2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) | 978-1-6654-1012-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/EuroSPW54576.2021.00013

Adopting Trusted Types in Production Web Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study

Pei Wang, Bjarki Ágúst Guðmundsson, Krzysztof Kotowicz
Security Engineering Research
Google
{pwng, bjarki, koto}@google.com

***Abstract*—Cross-site scripting (XSS) is a common security vulnerability found in web applications. DOM-based XSS, one of the variants, is becoming particularly more prevalent with the boom of single-page applications where most of the UI changes are achieved by modifying the DOM through in-browser scripting. It is very easy for developers to introduce XSS vulnerabilities into web applications since there are many ways for user-controlled, unsanitized input to flow into a Web API and get interpreted as HTML markup and JavaScript code.**

An emerging Web API proposal called Trusted Types aims to prevent DOM XSS by making Web APIs secure by default. Different from other XSS mitigations that mostly focus on post-development protection, Trusted Types direct developers to write XSS-free code in the first place.

A common concern when adopting a new security mechanism is how much effort is required to refactor existing code bases. In this paper, we report a case study on adopting Trusted Types in a well-established web framework. Our experience can help the web community better understand the benefits of making web applications compatible with Trusted Types, while also getting to know the related challenges and resolutions. We focused our work on Angular, which is one of the most popular web development frameworks available on the market.

***Index Terms*—web security, cross-site scripting, trusted types**

1. Introduction

Cross-site scripting, a.k.a. XSS, is known to be one of the most common security vulnerabilities on the web [1], [2], [3]. XSS is a type of code injection attack caused by web applications taking untrusted user inputs and interpreting them as HTML or JavaScript code without appropriate sanitization or escaping. A particular kind of XSS, called DOM-based XSS, occurs when there are bugs of this type in the client-side JavaScript part of a web application. As single-page web applications become more and more prevalent, client-side code is growing exceedingly large and complex [4], putting DOM-based XSS on the path to becoming the most common XSS variant.

In essence, DOM-based XSS exists because there lacks a mechanism in the JavaScript language or the Web APIs to distinguish trusted from untrusted data. There have been many attempts to establish this mechanism, e.g., by dynamic taint tracking [5], [6], [7] or static data-flow analysis [8], [9], [10]. Unfortunately, most of the dynamic methods suffer from scalability or performance issues when applied to large-sized code bases of modern web applications. Static methods, on the other hand, often have difficulties in analyzing the highly dynamic JavaScript and the templating languages commonly used in web development.

In recent years, a different approach to eradicating DOM-based XSS was developed at Google, called API hardening [11]. Different from traditional XSS countermeasures that focus on post-development mitigations, API hardening aims to prevent developers from writing XSS-prone code in the first place by replacing the original Web APIs with inherently safe APIs powered by a set of especially designed type contracts enforced at compile time.

The success of API hardening has inspired a new Web API proposal, called Trusted Types [12]. Trusted Types (TT) follow the idea of making Web APIs inherently secure, but they do so by introducing dynamic security enforcement inside the browser instead of performing static type checking like API hardening does. For TT-enabled applications, the browser will perform additional checks on values flowing to sensitive Web APIs and examine if they are produced by certain developer-endorsed factory functions, e.g., sanitizers. This effectively prevents malicious, user-injected values from being interpreted as HTML or JavaScript by the browser.

At the time of writing, Trusted Types are supported in Chromium-based browsers including Google Chrome and Microsoft Edge, and the API is work in progress under W3C [12]. Although it would be in the best interest of web developers to prevent XSS vulnerabilities from slipping into their applications, the process of adopting a new security mechanism can be lengthy and challenging [13]. Enabling Trusted Types typically requires developers to review, and very likely revise, the design and implementation of their applications. For large applications, this process is bound to be challenging if done without the right methodology or tooling support.

In this paper, we report our experience with making Angular, a widely used open source web framework, compatible with Trusted Types. Angular powers many high-profile web applications. Making Angular TT-compatible is the first step towards helping downstream applications embrace this new security feature and fend off the threats of DOM-based XSS.

Throughout this case study, we would like to shed

DOI 10.1109/EuroSPW54576.2021.00013

light on the costs and benefits of revising an existing code base for TT-compatibility. Hopefully, by following our path, other developers will be able to avoid many of the engineering obstacles we encountered if they decide to migrate their own frameworks or applications to Trusted Types. While the idea of integrating Angular with Trusted Types originated as part of a broader security improvement initiative at Google, the migration has been executed completely as an open source operation and our engineers have been working closely with Angular maintainers to fix problems related to the Angular ecosystem. Therefore, our experience should be inspiring to both the industry and the open source community.

In the course of this case study we made the following contributions:

- We refactored Angular, one of the most widely used web frameworks, to be compatible with Trusted Types, allowing downstream applications to adopt this new security feature to prevent XSS attacks. We demonstrated this by additionally migrating a medium-sized web application built upon Angular to Trusted Types.
- We introduced the methodology of the migration, enumerated the technical challenges, and shared the lessons we learned from this project. This first-hand experience can help other developers perform similar adoptions, which can lead to enhancing the security of a larger web community.
- We discovered and fixed a new Angular vulnerability during the migration, demonstrating how adopting Trusted Types can fundamentally improve the security of web software.

The rest of the paper is organized as follows: We begin with some relevant background knowledge in Section 2. We then introduce the motivation of our project and the encountered challenges in Section 3. Section 4 enumerates the tooling we utilized during the migration. We elaborate on the characteristics of existing violations in Angular in Section 5. Section 6 addresses the scope and design principles of the project, while Section 7 presents the rest of the migration details. We share the lessons learned from this process in Section 8. Section 9 and Section 10 discuss threats to validity and related work, respectively. We conclude the paper in Section 11. Appendix A describes the migration of an Angular web application to Trusted Types for the interest of certain readers.

2. Background

2.1. DOM-Based XSS

DOM-based XSS occurs when there are bugs in the client-side code of a website. By exploiting these bugs, attackers are able to inject executable content directly into the Web API injection sinks without needing to send malicious payload to the servers.

Figure 1 shows an example of DOM-based XSS. In this example, the vulnerability manifests when a malicious user visits the website with a URL parameter like the following:

```
?cat_name=<img src=x onerror=alert('xss')>
```

```
<html>
 <title>Cat Gallery</title>
 <body>
  <p>You want to see: <span id="cat_name"></span></p>
  <script>
    const catName = new URLSearchParams(
      location.search).get('cat_name') ?? 'signature cat';
    const nameArea = document.getElementById('cat_name');
    nameArea.innerHTML = catName; // XSS!
  </script>
 </body>
</html>
```

Figure 1. Example Web Application with a DOM-Based XSS Vulnerability

TABLE 1. MAJOR WEB APIS GATED BY TRUSTED TYPES

Web API	Required Type
`Document#write`	`TrustedHTML`
`Document#writeln`	`TrustedTHML`
`DomParser#parseFromString`	`TrustedHTML`
`innerHTML` and `outerHTML`	`TrustedHTML`
`srcdoc` on `<frame>`	`TrustedHTML`
`eval`	`TrustedScript`
`text` and `textContent` on `<script>`	`TrustedScript`
Constructor of `Function`	`TrustedScript`
`setTimeout` and `setInterval`	`TrustedScript`
`src` on `<script>` and `<embed>`	`TrustedScriptURL`
Constructor of `Worker`	`TrustedScriptURL`
Constructor of `SharedWorker`	`TrustedScriptURL`
`Element#setAttribute`	Element and attribute dependent

The root cause of the vulnerability is that the client-side code of the application does not properly check user input before assigning it to the `innerHTML` property of an HTML element, a DOM sink that interprets strings as HTML markup.

2.2. Trusted Types

In addition to `innerHTML`, there are numerous Web APIs that are prone to XSS. Trusted Types changes these APIs so they accept special types instead of strings.[1] These types can only be constructed in designated factory functions called policies. If this requirement is not met, the APIs will throw run-time exceptions.

2.2.1. Trusted Types policies. A Trusted Types policy is a set of factory functions that produce Trusted Types values from strings. There are three Trusted Types, for different contexts that can cause XSS:

- `TrustedHTML`, used for safely feeding DOM APIs that expect HTML, e.g., `innerHTML` and `document.write`.
- `TrustedScript`, for safely evaluating JavaScript dynamically with APIs like `eval`.
- `TrustedScriptURL`, for safely loading JavaScript code from a URL, e.g., the `src` attribute of `<script>` and `importScripts` in web workers.

Table 1 lists some of the most common Web APIs that Trusted Types currently monitor.

1. The complete roster can be found at https://w3c.github.io/webappsec-trusted-types/dist/spec/#integrations.

```
let sanitizeHTMLPolicy = {
    createHTML: str => DOMPurify.sanitize(
        // Sanitize HTML and only allow simple
        // "rich text"
        str, {ALLOWED_TAGS: ['b', 'q', 'i']});
};
if (window.trustedTypes && trustedTypes.createPolicy) {
  sanitizeHTMLPolicy = trustedTypes.createPolicy(
      'myEscapePolicy', escapeHTMLPolicy);
}

const sanitized = sanitizeHTMLPolicy.createHTML(
    '<b>See this<img src=x onerror=alert(1)></b>');
// If Trusted Types are available, 'sanitized' is
// now an instance of TrustedHTML.
el.innerHTML = sanitized;  // '<b>See this</b>'
```

Figure 2. Example of Using Trusted Types in Web Applications

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

Figure 3. Example of the "default" Trusted Types Policy

Figure 2 shows an example of a TT policy that produces `TrustedHTML`. The policy contains the factory function `createHTML` which sanitizes the input using DOMPurify, a library that strips executable parts off HTML markup [14]. To make use of this policy, developers need to wrap all such strings with a factory function before passing them to the Web API sinks, otherwise the browser will raise run-time exceptions. It is worth noting that Trusted Types fully fit in *progressive enhancement*, as also shown in Figure 2: With a minimal stub API (a.k.a. a tinyfill[2]) in place, data pass through the sanitization policy even in browsers that do not support Trusted Types. This means that when an application is Trusted-Types-compatible, it is resilient against DOM XSS in all browsers.[3]

Note that Trusted Types do not verify the semantics of the policies themselves and therefore Trusted Types alone do not offer any formal security guarantees. What Trusted Types ensure is that all values accepted by sensitive Web APIs must be processed according to one of the designated, centralized policies.

2.2.2. Default policy. A web application can define as many TT policies as it desires, but the policy named "default" is a special one. Due to various reasons, it may be extremely difficult, if not impossible, to modify certain pieces of code that cause TT violations. For example, the violation may come from a third-party library which the application developers have no control over. The default policy is a fail-safe policy which automatically applies to Web API sink usage not already covered by any other policy. In the example demonstrated by Figure 3, a default policy is created to ensure that all strings passed to HTML DOM sinks are sanitized by `DOMPurify` if they have not already been processed by other policies.

2. https://github.com/w3c/webappsec-trusted-types#tinyfill

3. See https://github.com/w3c/webappsec-trusted-types/wiki/FAQ#do-trusted-types-address-dom-xss-only-in-browsers-supporting-trusted-types for details regarding the security guarantees provided in this scenario.

2.2.3. Trusted Types enforcement. In Trusted Types, the availability of the JavaScript API to create policies is separated from the enforcement, which is controlled via Content Security Policy (CSP) in HTTP response headers. That allows the applications to introduce Trusted Types to their code gradually, without causing behavior changes, and allows for gradual rollout of the enforcement using the CSP report-only mode.

The CSP header has two separate directives[4] controlling Trusted Types:

- `trusted-types` specifies which policies can be defined, i.e. controlling the sources of Trusted Types instances.
- `require-trusted-types-for` controls which Web APIs require Trusted Types values, i.e., specifying the sinks of Trusted Types values.

If the application violates these restrictions, e.g., it tries to create a policy with a name not on the `trusted-types` allowlist, or a string is assigned to `innerHTML` without a policy to convert it to `TrustedHTML`, the respective JavaScript statement throws a `TypeError` and a CSP violation is dispatched. That makes introducing a security vulnerability a programming error, which is necessary to address early on.

Web navigation APIs can also cause XSS with `javascript:` schemed URLs. When Trusted Types are enforced, the browser will intercept `javascript:` URLs before the navigation and pass the code to the default policy's `createScript` factory function. If such a policy does not exist or if its `createScript` function rejects the URL, the browser stops the navigation attempt and dispatches a CSP violation.[5]

2.2.4. Comparison with other XSS mitigations. Trusted Types refine and complement existing XSS mitigation measures with the idea of managing security risks in the early stages of software development across the entire project. Their Web API is visible to static tooling. As such, it surfaces the XSS risks directly in the code and can leverage the tooling to address those risks when writing the application - contrasting it with other Content Security Policy directives, compliance with which can only be asserted after the code has been written.

Static tooling for DOM XSS issues already exists, but when used alone, it fails to detect all instances of the vulnerabilities [7]. Trusted Types, with their enforcement in the browser engine, act at run time, and can prevent the accidental DOM XSS risks that static analysis failed to find.

Trusted Types don't focus on the payloads (like XSS filters [6] and Web Application Firewalls) and impose constraints on all uses of dangerous Web APIs, even though it may not be immediately clear why a particular use can lead to XSS vulnerabilities. However, industry practice shows evidence that using this rigid approach can substantially reduce XSS across large code bases [11].

4. https://w3c.github.io/webappsec-trusted-types/dist/spec/#integration-with-content-security-policy

5. In such a setup, enforcing Trusted Types blocks all `javascript:` URLs, and the default policy can selectively allow certain known-to-be-secure payloads like `void(0)`.

```
import {Component} from '@angular/core';

@Component({
  selector: 'home-page',
  template: '<div [innerHTML]="content"></div>',
})
export class HomePageComponent {
  content =
    '<img src="logo.png" onerror=alert("error")></img>';
}
```

Figure 4. Angular Template Example

2.3. Angular

Angular [15] is a design framework and development platform for building single-page web applications. Originating from Google, Angular has become one of the most popular open source web frameworks. Angular has a powerful template system that allows developers to write a DSL that conveniently controls the UX/UI by coordinating data between the JavaScript code and the HTML page. Figure 4 demonstrates an example of this DSL, in which the `@Component` decorator takes an inline template that renders a `<div>` element, with its `innerHTML` property bound to the `content` property in the `HomePageComponent` class. When this page is rendered, Angular automatically updates the content of the rendered `<div>` with the value of the property bound to it.

There used to be a time when web frameworks failed to offer fundamentally better security than standalone applications [16]. To avoid these historical pitfalls, Angular implements strict contextual escaping [17] and treats all input values of the templates as untrusted by default. Take Figure 4 again for example, Angular treats the `innerHTML` property as a sensitive HTML DOM sink in the template's context, therefore automatically sanitizing the value bound to it when rendering the template. That means the inline event handler of the `<img>` tag in `content` will be stripped to prevent unexpected code execution.

3. Motivation and Challenges

This section discusses why we would like to migrate Angular to Trusted Types and reports the experience as a case study to a wider audience. It also discusses the encountered challenges of this project.

3.1. Motivations

Neither automated tools nor human reviews are sufficient to prevent XSS vulnerabilities in large-scale web applications with a high degree of confidence [18], [11]. Moreover, under intense time pressure to fix vulnerabilities that already manifest in production, the remediation itself can be error-prone and may introduce new vulnerabilities [19], [20], [21]. Even if such remediation is successful, without improvements of the development process, security regressions can occur at a later point in time.

Various web applications and libraries already use Trusted Types, e.g., several Google web applications, Visual Studio Code,[6] DOMPurify, Webpack, Lit, FAST.[7] In this paper, we describe the migration of a major web framework, which is quite different in many aspects. Empirical studies show that frameworks have significant impact on the security of downstream applications [22]. Meanwhile, refactoring a framework has some unique challenges.

Even with numerous security features in Angular, building applications on top of it does not automatically make the applications secure. For example, Angular applications can contain code that bypasses the framework's default security checks.[8] Angular applications can also directly interact with the error-prone Web APIs, including the DOM.[9] Finally, applications may include code from non-Angular dependencies that may be vulnerable to XSS. Trusted Types support in Angular enables downstream applications to contain all such security threats.

3.2. Challenges

From a software engineering point of view, adopting Trusted Types in an established project with many developers is non-trivial. Mechanically eliminating all incompatibilities with Trusted Types is not the purpose, as it can be done as easily as defining a trivial default policy (see Section 2.2.2) that automatically converts every string to a Trusted Types value whenever the browser is about to raise an exception. That, however, does not help improve the security of the project in any regard.

3.2.1. Violation Identification. The first step towards making an application or framework compatible with Trusted Types is to identify the code locations from which TT violations can emerge. For application migration, this process can be highly automated with sufficient technical support. There are both dynamic and static tools that can help detect TT violations in the code base, which will be introduced with more details in Section 4. For Angular, the challenge deepens due to the templates. Before the applications are deployed, Angular needs to perform code generation to transform templates into common HTML and JavaScript, which may contain TT violations. These violations cannot be directly identified or fixed. Instead, we need to identify the code locations that produce the violating code.

3.2.2. Fixing Direct Violations. It is not always immediately obvious how to fix a TT violation. For example, if the string assigned to `innerHTML` is a compile-time constant, wrapping that string into `TrustedHTML` is an adequate and secure solution as we know this string can never be controlled by attackers. If, however, it is not a compile-time constant, we have to take a different approach, like sanitizing the string or escaping it. It may require deep domain knowledge to decide which option to take.

6. https://code.visualstudio.com/
7. https://github.com/w3c/webappsec-trusted-types/wiki/Integrations
8. https://angular.io/guide/security#trusting-safe-values
9. https://angular.io/api/core/ElementRef

```
{
  "csp-report": {
    "document-uri": "https://my.url.example",
    "violated-directive": "require-trusted-types-for",
    "disposition": "report",
    "blocked-uri": "trusted-types-sink",
    "line-number": 39,
    "column-number": 12,
    "source-file": "https://my.url.example/script.js",
    "status-code": 0,
    "script-sample": "Element innerHTML <img src=x"
  }
}
```


Figure 5. Report of Trusted Types Violations

3.2.3. Fixing Indirect Violations. As mentioned in Section 3.2.1, Angular has a template language that is transpiled to TypeScript and then JavaScript code. That means we not only need to fix code that triggers TT violations, but also need to fix code that produces TT-violating code. As we will show in the later sections of this paper, we have to make a few significant structural changes to fix certain violations in Angular.

4. Tooling Support

Trusted Types mechanisms are controlled through Content Security Policies, which offer a report-only mode that helps developers safely identify code locations that do not conform to Trusted Types when they dynamically run the applications. Additionally, we have developed an analyzer to make it easier to identify those code locations statically.

4.1. Report-Only Mode

If Trusted Types are enforced in report-only mode, the browser will record all violations without actually raising exceptions. For example, when the application passes a string to `innerHTML`, the browser may dispatch a `SecurityPolicyViolation` event and send a report to a developer-specified endpoint.

The example report in Figure 5 notifies that in `https://my.url.example/script.js`, the `innerHTML` property of a DOM element was assigned to, on line 39, with a string beginning with `"<img src=x"`.[10]

4.2. Static Analyzer

Dynamically running an application to uncover all TT violations becomes unwieldy when the applications get large. Moreover, relying on dynamic detection means developers can introduce new violations which they only discover during testing or, even worse, in production. To alleviate this inconvenience, an analyzer that can statically identify violations in TypeScript code can be used. We created an open source analyzer for this purpose called tsec. [11]

In its current form, the analyzer is essentially an extended TypeScript compiler that additionally scans the abstract syntax tree of each source file to look for uses of XSS-prone APIs that can cause TT violations. If those XSS sinks do not already take Trusted Types values, the checker emits compiler errors. The analysis reuses the type information inferred by the compiler, but is otherwise flow-insensitive, and therefore extremely efficient [11]. For example, we have used tsec to analyze the entire code base of Visual Studio Code, which consists of almost a million lines of TypeScript code, in a few minutes.

Since TypeScript has an unsound type system to be compatible with JavaScript, tsec is also unsound and can miss code locations that cause TT violations. Our experience is that tsec can capture most of the TT violations before developers need to run any dynamic analysis (see Section 3.2.1 and Table 2).

TABLE 2. VIOLATIONS IDENTIFIED BY EACH METHOD

Method	Identified Violations
Static analysis (tsec)	31
Angular applications	13
Angular tests	2

5. Identified Violations

We took the following approaches to identifying potential TT violations in Angular:

- Ran tsec to analyze the source code of Angular (see Section 4.2 for details about tsec).
- Ran Angular's unit tests with Trusted Types enforced and looked for failures.
- Ran small- to medium-sized Angular applications with report-only Trusted Types enabled.

Each of these approaches excels at locating certain types of TT violations. For example, tsec can detect the majority of direct violations. Running Angular's unit tests can reveal a good amount of indirect violations, thanks to the good test coverage. Lastly, running representative Angular applications with TT enforced surfaces all the remaining violations, however it might be difficult to infer their root cause. Therefore, a manual source code review is still required and helps us understand why the violations are present - and that informs us how to fix them later. It might take a few iterations to identify all the violations, as some of them only emerge after others are removed.

Table 2 shows how many violations were identified by each method. While there were certain overlaps among the violations captured in different ways, we found it necessary to employ all of them. Static analysis performed best in finding direct violations in TypeScript code, but dynamic methods were necessary since they helped us better understand the context in which the violations emerge and therefore made it easier to devise solutions.

10. TT default policy can also be used to inspect string values passed to injection sinks at run time, and in some cases might offer a more convenient and programmatic way of collecting violation data without breaking the application.

11. The analyzer only processes TypeScript code, since a precise analysis requires type information. Available at https://github.com/google/tsec.

In the remainder of this section, we classify the identified violations into a few categories based on their root causes. We try to explain these root causes with minimum Angular-specific context. We believe at least some of them also apply to other modern web frameworks, if these frameworks are to be made compatible with Trusted Types.

Template constants. When an Angular template is compiled, attributes and properties of HTML elements that contain constant strings are hoisted into a local array structure. When the template is then rendered, these constant strings are passed directly to `setAttribute` calls for the corresponding element. These operations intentionally do not undergo any sanitization, as constants in Angular templates are developer-controlled and considered trusted. However, in cases where the given attribute on a particular element is an injection sink, this will raise a Trusted Types violation.

Bound values in templates. When an Angular template is compiled, attributes of HTML elements whose values are bound to a data model are converted to a function call that looks roughly as follows:

```
angularSetAttribute(
  "attributeName", valueExpression, angularSanitizer);
```

The third argument is only present when the given attribute or property is sensitive in the context of the given element, and dictates how the value of the expression should be sanitized before it is passed to the corresponding `setAttribute` call for the corresponding element. This will either use an internal sanitizer or a custom sanitizer defined by the application. The output from the sanitizer is a string that then gets passed to a `setAttribute` call for the corresponding element, which may result in a Trusted Types violation.

Inert DOM builder. Angular's internal HTML sanitizer, which is called when an attribute or property that takes HTML as input needs to be sanitized, uses an approach that begins by parsing the given HTML into an inert DOM tree, traverses that tree, and then generates the sanitized HTML as a product. To parse the HTML into an inert DOM tree, Angular provides an inert DOM builder, which either uses the `DOMParser` API, or creates a template element and sets its `innerHTML` property. Both approaches cause Trusted Types violations, since Trusted Types treat all DOM sinks as risks, even if the resulting DOM nodes are not yet attached to the main document of the page.

ICU messages and i18n constants. Angular templates support internationalization (i18n), both for text content of elements as well as values of attributes and properties. Text that is marked for i18n translation can contain International Components for Unicode (ICU) directives [23], which can be used to return different strings based on a condition, e.g. the plurality of a counter. ICU messages are parsed at runtime using an approach similar to the one used by Angular's HTML sanitizer, and thus also cause a Trusted Types violation due to their use of the inert DOM builder. However, in addition to that, the parsed ICU message can contain HTML that is converted into an intermediate opcode structure for creating and updating the corresponding DOM fragment. Values of attributes and properties in these ICU messages are parsed as strings, and may thus cause Trusted Types violations when the opcodes are executed and the corresponding `setAttribute` is called.

CSS keyframes driver. Angular has a component that facilitates the animation of element styles and dynamically creates a `<style>` element and populates it with CSS via `innerHTML`, which causes a Trusted Types violation.

JIT interpreter. Angular template compilation can be done either ahead of time (AOT) or just in time (JIT). Although a large majority of Angular applications have migrated to AOT compilation, there are still applications that rely on JIT compilation. Since templates may contain JavaScript code, e.g., in event handlers, Angular's JIT interpreter needs to call a DOM API that evaluates strings as executable code. When this happens, a TT violation emerges.

Development features. When an Angular application is served in development mode, additional code is included for debugging, profiling, and other development-related features. One of those features is the `named_array_type`, which creates custom named array classes to aid with profiling and debugging of otherwise anonymous array structures. To achieve this, it calls the `Function` constructor with the necessary class definition as a string, causing a Trusted Types violation.

JSONP. JSON with padding, aka JSONP, is a method for web applications to send and receive cross-domain data in JSON. Due to security concerns, reading cross-domain data in JavaScript is restricted by browsers. JSONP is a method to work around this restriction, based on an exception that requesting cross-domain script files is always allowed.[12] Angular provides a module for performing JSONP. To make such a request, the application provides a string URL pointing to the JSONP endpoint. The module then creates a `<script>` element and sets its `src` attribute to the given URL. This causes a Trusted Types violation.

Third-party libraries. Third-party libraries that the application depends on may have to perform dangerous DOM operations, and if they are not using Trusted Types, this will cause Trusted Types violations. It has been shown that third-party code is one of the major blockers of enforcing new security policies in web applications [24].

Integration and unit tests. When running Angular's test suite with Trusted Types enforced, there were a large number of failures due to Trusted Types violations. Most of them occurred within individual unit tests and associated test utilities related to rendering pages dynamically. There were also Trusted Types violations in the test suite drivers themselves.

12. Modern browsers support Cross-Origin Resource Sharing (CORS) which is in general more preferable for cross-domain data transfer. But there are still a good amount of applications using JSONP.

6. Design Considerations and Scope

Even without further inspection, it is clear that the challenges in removing the aforementioned violation categories can vary drastically. Before jumping into fixing, we first define the scope of our refactoring and key principles to follow during the process.

6.1. Design Considerations

We identified three principles that should be followed during the migration. These principles should be able to ensure that adding Trusted Types support into Angular will indeed improve its security capability without degrading its usability or performance.

Security. This is the most important principle we shall respect, as it is the whole purpose of the migration. In particular, values from an Angular application should not be automatically converted to Trusted Types unless the framework can guarantee that it is safe to do so, or the developer explicitly opts in to such a conversion. This is similar to the practice introduced by previous research called API hardening [11], i.e., any code that may introduce XSS vulnerabilities is explicitly marked by special program constructs easily recognizable by compilers and human reviewers.

Backward, Cross-Platform Compatibility. Considering that Angular is the foundation of many community creations, extreme caution must be taken to avoid introducing breaking changes that can stop applications built upon Angular from functioning correctly. Therefore, Trusted Types violations should be addressed in a manner that is as close to being completely backwards compatible as possible.

Also, Trusted Types are a relatively new feature and not every browser supports it. Besides, Angular can even run in a non-browser environment. We need to make sure that the refactored Angular and downstream applications affected by the refactoring can still function correctly on platforms without Trusted Types support.

Code Size. For some applications, it is critical to keep the size of served JavaScript code in check to speed up website loading. As a framework, Angular has many features and it is very common that a certain application only needs some of them. To avoid unnecessarily bloating the code size, Angular supports tree shaking, i.e., the ability to prune out chunks of the JavaScript bundle that are not used by the application, and thus reducing the size of the bundle. As an example, if an Angular application does not make use of any i18n features, great care has been taken to make sure that the entire i18n library can be pruned from the resulting JavaScript bundle. Good treeshakability is one of the main design goals of Angular, and is an important consideration when new components or features are added to Angular.

Any Trusted-Types-specific code introduced in Angular should be tree-shakable to the greatest extent possible, meaning that if an Angular application does not use any of the components or features listed in the previous section that could cause Trusted Types violations, then the TT-specific code will not be present in the application's JavaScript bundle.

6.2. Scope

Among the many classes of violations, some of them may be harder to address than others without compromising our design principles. In the interest of time and giving priority to features that are more widely used, we de-prioritized the following types of violations in our refactoring because they are not blockers for a minimum viable product (MVP) and can be addressed in the future:

- First-class TT support for the JIT interpreter is out of scope for this case study, as it requires a very complicated design and the feature itself is not too widely used among Angular developers. In this case study, we devised a solution that "just works" for JIT without conducting a thorough review from the security perspective. See Section 7.2 for details.
- JSONP does not need to be supported at all in the first iteration as it is on the path of deprecation in favor of modern alternatives such as CORS.

7. Migration

Among the Trusted Types violations discussed in Section 5, some can be rather easy and straightforward to fix. In this section, we focus primarily on the non-trivial fixes. To free readers from the rather complex technical details of Angular's internal gadgets, we have tried our best to distill the high-level ideas behind our migration strategy.

7.1. General Strategy

We would like Angular to run with Trusted Types whenever possible, with minimum user configuration, as misconfiguration of security policies has been a common source of insecurity [13]. Therefore, we decided that if Angular is being run in an environment that supports Trusted Types, Angular will automatically produce Trusted Types internally. This is internal-only behaviour unexposed to an Angular application. If the application itself does not enable Trusted Types enforcement by sending the CSP HTTP header, the internally enabled TT-related operations are effectively no-ops.

When Trusted Types are not available, our design makes sure that Angular will fall back to using plain strings, just as before Trusted Types support was added. This facilitates backwards compatibility with non-browser platforms and browsers that do not support Trusted Types.[13]

7.2. Managing Trusted Types Policies

Given the wide range of root causes of TT violations discussed in Section 5, it is beneficial to use different TT policies to handle different situations.

13. Applications can always use polyfills to enable Trusted Types on those platforms and browsers, but Angular as the framework cannot assume all applications will choose to do that.

TABLE 3. FIXES APPLIED TO ADDRESS TRUSTED TYPES VIOLATIONS

Violation category	Resolution	Pull request(s)
Template constants	Used `angular` policy and template tag functions verifying that the value is constant.	#40082
Bound values in templates	Used `angular#unsafe-bypass` policy when value comes from a custom sanitizer or `bypassSecurityTrust` function. Otherwise, `angular` policy.	#39217, #39218
Inert DOM builder	Used `angular` policy. The value is sanitized internally.	#39208
ICU messages	Dynamic values treated like bound values in templates. Constant values converted with `angular` policy.	#39221
CSS keyframes driver	Replaced `innerHTML` with `textContent`.	#37397
JIT interpreter	Used `angular#unsafe-jit` policy. This might be unsafe, but JIT usage in Angular is discouraged.	#39210
Development features	Used `angular` policy, asserting that Angular runs in development mode.	#39209
JSONP	n/a (out of scope)	-
Third-party libraries	Used dedicated policies in upstream packages (e.g. Webpack).	-
Integration and unit tests	Used dedicated policies in upstream packages (e.g. Karma).	-

During the migration, we defined three Trusted Types policies, one of which is the main policy and the other two are supplemental. The main policy - `angular` - is used when Angular can deduce that a given string is safe for use in a given security context. For example, constant strings are considered safe and thus can be processed by the main policy. Another example of this is the output of Angular's sanitizer pipeline that facilitates contextual auto-escaping.

There are cases where Angular cannot be sure that a given string is safe, but it seems that the application developer intentionally and explicitly bypassed a security measure in Angular such as its internal sanitization pipeline.[14] In those cases, Angular will use a separate `angular#unsafe-bypass` policy, specifically marked as insecure. Applications using the unsafe features and which are not willing to migrate off them, may allow this policy through server-side CSP headers. This allows Angular applications to start enforcing Trusted Types even when using custom security primitives, such as a custom HTML sanitizer. Since Trusted Types do not inspect the semantics of the security primitives behind the polices, the insecure policy, as the name suggests, cannot provide much security guarantee. As a bottom line, developers have to take extra steps to enable this policy and the bypasses can be easily discovered by security reviewers if the application maker indeed decides to turn on this feature.

The third policy (`angular#unsafe-jit`) we introduced through the migration is solely for the JIT interpreter. As mentioned in Section 6.2, full JIT support would not be available in the first iteration, so we defined this policy just to mitigate JIT-related violations. Different from the other two policies, the JIT policy can only produce `TrustedScript` values, which is sufficient to fix the violations.

Table 3 pairs the violation categories from Section 5 with the methods used to securely resolve them. Each row also specifies the Pull Requests in the Angular's GitHub repository[15] containing the relevant code changes.

It should be noted that all three internal policies we created for Angular only contain identity factory functions, i.e., they merely wrap the input strings into instances of corresponding TT types. To make sure those policies are invoked exclusively in a secure manner, they are made private and can only be invoked at limited code locations, e.g., inside Angular's sanitization pipeline and respective internal packages. In other words, we created a small, well designed, and thoroughly reviewed trusted computing base (TCB) in a large project to ensure Trusted Types values are always constructed securely and the policies are not abused. This is one of the typical patterns for designing Trusted Types policies, usually adopted by libraries and frameworks that are designed with established security principles and capable of offering security services to other applications. Regarding policies made for most ordinary applications, the sanitization logic should instead be specified *inside* the policy factory functions in order to ensure that Trusted Types are always mitigating incoming threats and to make it safe to invoke the policies without restrictions. Most of the time, this alleviates the burden of security reviews so that application developers can focus on implementing other features.

7.3. Verification

To verify if the migration had completed, we mostly relied on integration and unit tests, with Trusted Types enforced. To achieve this, we first needed to fix the violations in the tests themselves and the test suites Angular uses. Angular mostly depends on two open source tools for testing, i.e., Jasmine[16] and Karma.[17] We made both compatible with Trusted Types and pushed the changes upstream. Compared with the Angular migration, adding TT support for these two tools was a much more mechanical process, partially due to the fact that there are far fewer

14. https://angular.io/api/platform-browser/DomSanitizer#security-risk

15. https://github.com/angular/angular
16. https://github.com/jasmine/jasmine
17. https://github.com/karma-runner/karma

security concerns regarding testing infrastructure. It took a single pull request containing about 50 lines of code changes to patch each project.

In addition to the existing tests, we also created new integration tests to cover additional TT-related cases, making sure there will be no security regressions in the future.

7.4. Impact

After resolving the most important issues that blocked Trusted Types adoption, Angular released a version (v12.1.1) that officially supports Trusted Types.[18] Issues not included in this first iteration of adoption are planned to be addressed in the next release.

On the completion of the migration, we turned on Trusted Types enforcement for several Angular-based applications, as part of the ongoing effort to roll out Trusted Types for all Google's products. We also added Trusted Types to a medium-sized open source Angular application (see Appendix A). Trusted Types enforcement performed in this fashion guarantees that the DOM XSS-relevant code in these applications is minimal and sufficiently isolated from the rest of the code base. With the help of the static analyzer tsec and the new integration tests, we ensure that this much smaller code base will be subject to an efficient security review upon changes. This property holds true regardless of additional dependencies that the covered applications might acquire over time.

7.5. Engineering Effort

In total, it took a small team of four security engineers and an intern roughly six months to complete the migration, without working full-time on the project. Of those, only around six weeks were spent on the concrete design and implementation of the Trusted Types integration. As none of the engineers working on the project had contributed to Angular before, the rest of the time was spent on learning about Angular's internals from scratch and communicating with the Angular community for suggestions and reviews. The implementation load was also modest. The migration team sent out a total of 14 pull requests that were directly related to Trusted Types adoption, with about 2600 lines of code added and modified. The team sent another four pull requests to other open source projects to coordinate the changes made to Angular.

Angular is one of the most feature-rich modern frameworks, which largely contributed to the project scope and complexity. Comparative integrations, e.g. with Lit,[19] or React,[20] were noticeably easier. There, the overall time spent on design and implementation was in the order of days and weeks, respectively.

It should be emphasized that frameworks exist to address common issues, such that many end applications do not need to tackle them individually. That comes at a price: a bar for a framework feature development is high and introduces unique challenges like backwards compatibility, demanding performance goals or API surface design. These challenges are largely absent when changing regular web applications (See Appendix A).

Considering these factors, we believe that the engineering cost of the migration was fairly low in contrast to the benefits delivered by this project.

18. https://angular.io/guide/security#enforcing-trusted-types
19. https://github.com/lit/lit/pull/1772
20. https://github.com/facebook/react/pull/16157

8. Lessons Learned

During the adoption process, many unexpected situations emerged, both positive and negative. We believe that some of these situations and the ways we handled them are valuable experiences that can benefit similar work in the future. This section summarizes the lessons we learned.

8.1. Finding New Vulnerabilities

During the migration, one of the violations caught our particular attention. By studying Angular's code, we noticed that the mandatory sanitization of sensitive DOM attributes and properties in Angular templates could be unintentionally bypassed by the application developer. Consider the template below:

```
<iframe srcdoc="{{userInput}}" i18n-srcdoc></iframe>
```

The `i18n-srcdoc` custom attribute directs Angular to translate the `srcdoc` attribute for localization when rendering the template. Without the i18n translation, the `srcdoc` attribute is passed through Angular's sanitization pipeline, as it may contain unsafe user input that would result in XSS. With the i18n translation, however, no sanitization was performed; instead, the string was passed directly to the underlying DOM sink, causing a Trusted Types violation. We identified this as a security vulnerability and have since disallowed i18n translations on security-sensitive attributes and properties in Angular.

It is worth highlighting that if we had not discovered this vulnerability, triggering this behaviour in a production application with Trusted Types enforced would have caused a Trusted Types violation, instead of XSS. Such applications would therefore be immune to the vulnerability, even though it occurs deep within the internals of the Angular framework. This exhibits the great security benefits that Trusted Types bring to an application.

> Adopting Trusted Types brings new chances to review the security aspect of a project's design and implementation, possibly leading to the discovery of new vulnerabilities. With Trusted Types enforcement, applications can become more resilient to undiscovered vulnerabilities.

8.2. Third-Party Libraries

It has been reported that third-party libraries are one of the key blockers that prevent web applications from adopting security features like CSP and SRI [24].

As mentioned in Section 7.3, we had to patch the third-party test frameworks and runners to be able to verify our changes indeed added TT compatibility to Angular. The other blockers on the third-party side were Bazel, the build system that compiles and tests Angular, Webpack, a code

and asset bundling tool for distributing compiled Angular applications. These were the only third-party dependencies we had to patch and the fixes were straightforward, so our experience only partially aligns with historical observations. The reason could be that web frameworks typically have unique designs and most DOM-related operations they perform have very uncommon patterns; they do not have too many third-party libraries to rely on for manipulating the DOM. Also, web frameworks may intend to avoid critical run-time dependencies for the sake of ecosystem stability.

> Frameworks tend to have fewer dependencies than web applications. While third-party libraries and tools can block Trusted Types adoption for frameworks, the obstacle is less severe than for web applications. On the other hand, third-party blockers can be more subtle in the case of frameworks, as the conflicts can manifest during testing and distribution in addition to run time.

8.3. Backward and Cross-Platform Compatibility

When introducing new features to a framework that serves an inherently fragmented ecosystem, maintaining backward compatibility could lead to a "long tail" effect, i.e., a large amount of resources is spent on platforms with a small fraction of users. In this project, we noticed this effect due to a TT implementation bug in Chrome 83, a browser released in May, 2020. Due to this bug, string-to-code APIs like `eval` basically became no-ops when Trusted Types were enabled. At the time we noticed this issue, Chrome 83 only had about 0.2% market share worldwide [25], so the issue only affected a very small fraction of our users. Still, we made a commitment to retaining backward and cross-platform compatibility to the best of our abilities. We sent a dedicated pull request to add a workaround for this browser bug.[21] Although the code change is small, it took us quite some time to develop this solution.

> Adding new security features into web frameworks or applications may introduce unexpected backward compatibility issues that affect a small group of users but cost a notable amount of effort to fix.

8.4. Collaboration Model

Before we successfully added Trusted Types support to Angular, we made two similar attempts. In the first two trials, we mostly tried to work by ourselves and did not coordinate well with the Angular community. This time however, we followed community contribution guidelines and built good communication channels with the Angular technical leads. It turned out that the open source software development model works exceptionally well in this case. Previous studies suggest that the open source community acknowledges the importance of security but does not always prioritize security-related work due to various reasons [26]. Having a group of security experts drive the security evolution of open source projects with support from the community can be an effective collaboration model, even if they did not participate in community work before. Our case shows that this model works for a complex project like Angular, and we expect it can work for many other projects as well.

> With good collaboration and contribution practices, open source projects can benefit from security-improving efforts initiated by "outsiders."

21. https://github.com/angular/angular/pull/40815

9. Threats to Validity

Internal Validity. Although we have successfully adopted Trusted Types in Angular, it remains to be seen how effective the new security mechanism is against XSS threats. Previous research has shown that static enforcement of safe coding practices can effectively prevent XSS vulnerabilities [11], but we do not yet have quantitative measurement regarding the effectiveness of Trusted Types at a large scale.

External Validity. This study focuses on Angular, an open source web framework. Our migration work inevitably requires collaboration with the core Angular team. Although Angular contributors always advocate community-driven development and we have tried our best to follow this guideline during the adoption process, our experience may still be biased due to a similarity in the development environment and organization culture to the ones used by Angular. These factors may affect how generally our lessons learned can be applied to other scenarios.

10. Related Work

There has been a long history of preventing script injection attacks with additional security mechanisms built into browsers. The Browser-Enforced Embedded Policies (BEEP) [27] aimed to regulate script inclusion operations based on an application-controlled policy. A system called SOMA [28] extends the the scope of BEEP to other web resources in addition to scripts. These two systems inspired the creation of Content Security Policy (CSP) [29], which now has been standardized and implemented by many browsers. Trusted Types are the latest advancement in this direction, introducing much more fine-grained script injection prevention. Trusted Types can also be seen as a continuation of an attempt of eliminating XSS vulnerabilities by deploying a series of secure-by-default APIs [18], [11]. Trusted Types complement the effort by adding run-time checks in DOM APIs.

Despite being the first widely available in-browser mechanism for preventing script injections, CSP did not go through a smooth adoption process across the web [30], [31]. Many studies have suggested that websites face many problems when trying to enable CSP with an effective policy. One of the most important reasons appears to be that adopting CSP requires extensively rewriting the applications [32]. Third-party libraries being incompatible with CSP also plays a significant role in hindering the

adoption [24]. Browser extensions can also cast a negative effect on the effectiveness of CSP, as many of them tamper with the CSP policies of a page because policy makers of the page did not take extensions into consideration [33]. JSONP is yet another common blocker for CSP adoption, as the scripts are dynamically generated [30].

In response to the challenges in adopting new security features in established software projects, there have been various efforts in making it easier for developers to write secure code or harden their applications with automated tools. Xie et al. designed an IDE plugin that automatically prompts programmers with better security options including how to avoid XSS [34]. After studying the characteristics of a large amount of vulnerable code changes, Bosu et al. recommended that software projects should create or adapt secure coding guidelines and establish a dedicated security review team [35]. Parameshwaran et al. designed a tool to automatically patch web applications by replacing unsafe string interpolation with safer code patterns [36]. A system called ZigZag is capable of dynamically instrumenting a web application and performing invariant detection on security-sensitive code [37]. Musch et al. proposed a tool that automatically removes third-party code that facilitates string-to-code transformations from web applications [38].

There have been reports of cases where developers utilize features of specific programming languages in pursuit of additional security benefits. For example, Doligez et al. released a case study about developing secure XML validators with functional programming [39]; Anderson et al. reported their experience with building a new Browser Engine in the Rust Programming Language which is by default memory safe [40]; Narayan et al. refactored the code isolation mechanism of Firefox's render engine for improved security and performance, making use of advanced features of the C++ type system [41].

11. Conclusion

In this case study, we shared our experience with migrating an established web framework to adopt Trusted Types, the latest web API proposal for preventing XSS attacks. We introduced the procedure of adopting Trusted Types in a fairly complex project, detailing the root causes of security violations, and how we fixed them. We analyzed the benefits and cost of the migration and discussed the lessons learned from this process. We believe that with the right tooling and methodology, adding Trusted Types to a web application or framework should be fairly cost effective. We hope that this report can motivate more web developers to enhance the security of their web applications with Trusted Types and help them overcome the obstacles that we encountered.

References

[1] "Reshaping web defenses with strict content security policy," 2016. [Online]. Available: https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html

[2] "2019 application security statistics report," 2019. [Online]. Available: https://www.whitehatsec.com/resources/2019-application-security-statistics-report/

[3] "Web application attacks statistics," 2017. [Online]. Available: https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-application-attacks-2018-eng.pdf

[4] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in *Proceedings of the 26th USENIX Security Symposium*, ser. USENIX Security '17. Vancouver, BC: USENIX Association, Aug. 2017, pp. 971–987. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock

[5] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013, p. 1193–1204.

[6] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *Proceedings of the 23rd USENIX Security Symposium*, ser. USENIX Security '14. San Diego, CA: USENIX Association, Aug. 2014, pp. 655–670.

[7] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting," in *2018 Network and Distributed System Security Symposium (NDSS)*, ser. NDSS '18, 2018.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for javascript," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, p. 50–62.

[9] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, p. 177–187.

[10] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, p. 171–180.

[11] P. Wang, J. Bangert, and C. Kern, "If it's not secure, it should not compile: Preventing DOM-based XSS in large-scale web development with API hardening," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21, 2021.

[12] "Trusted types." [Online]. Available: https://w3c.github.io/webappsec-trusted-types/dist/spec/

[13] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.

[14] M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against xss and markup injection," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 116–134.

[15] "Angular," https://angular.io/.

[16] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *Proceedings of the 2011 European Symposium on Research in Computer Security*, ser. ESORICS '11. Springer, 2011, pp. 150–171.

[17] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011, p. 587–600.

[18] C. Kern, "Securing the tangled web," *Communications of the ACM*, vol. 57, no. 9, pp. 38–47, 2014.

[19] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, 2011, pp. 26–36.

[20] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama, "Fix that fix commit: A real-world remediation analysis of JavaScript projects," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '20, 2020, pp. 198–202.

[21] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017, pp. 2201–2215.

[22] K. Peguero, N. Zhang, and X. Cheng, "An empirical study of the framework impact on the security of javascript web applications," in *Companion Proceedings of the The Web Conference 2018*, ser. WWW '18, 2018, p. 753–758.

[23] "ICU - international components for unicode," http://site.icu-project.org/.

[24] M. Steffens, M. Musch, M. Johns, and B. Stock, "Who's hosting the block party? studying third-party blockage of CSP and SRI," in *Proceedings of the 2021 Network and Distributed Systems Security Symposium*, ser. NDSS '21, 2021.

[25] "Browser version market share worldwide," https://gs.statcounter.com/browser-version-market-share#monthly-202011-202012-bar, 2020.

[26] S.-F. Wen, M. Kianpour, and S. Kowalski, "An empirical study of security culture in open source software communities," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '19, 2019, pp. 863–870.

[27] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07, 2007, p. 601–610.

[28] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, 2008, p. 89–98.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, 2010, p. 921–930.

[30] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1376–1387.

[31] S. Calzavara, A. Rabitti, and M. Bugliesi, "Semantics-based analysis of content security policy deployment," *ACM Trans. Web*, vol. 12, no. 2, Jan. 2018.

[32] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is csp failing? trends and challenges in csp adoption," in *Proceedings of 17th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '14. Springer, 2014, pp. 212–233.

[33] D. Hausknecht, J. Magazinius, and A. Sabelfeld, "May i?-content security policy endorsement for browser extensions," in *Proceedings of the 2015 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '15. Springer, 2015, pp. 261–281.

[34] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "ASIDE: IDE support for web application security," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 267–276.

[35] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 257–268.

[36] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, p. 272–283.

[37] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "ZigZag: Automatically hardening web applications against client-side validation vulnerabilities," in *Proceedings of the 24th USENIX Security Symposium*, ser. USENIX Security '15, 2015, pp. 737–752.

[38] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "ScriptProtect: Mitigating unsafe third-party JavaScript practices," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19, 2019, p. 391–402.

[39] D. Doligez, C. Faure, T. Hardin, and M. Maarek, "Avoiding security pitfalls with functional programming: A report on the development of a secure xml validator," in *Proceedings of the 37th International Conference on Software Engineering Companion*, ser. ICSE '15, 2015, p. 209–218.

[40] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using Rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, pp. 81–89.

[41] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX Security '20, 2020, pp. 699–716.

Appendix A. Migrating Angular.io to Trusted Types

Trusted Types support in Angular enabled enforcing TT in web applications built on top of it. We have already started this process for internal applications, but wanted to showcase what such migrations might look like on an open source target. In this appendix, we demonstrate the Trusted Types migration of the web application powering the Angular documentation site at https://angular.io. The accompanying code is available at https://github.com/angular/angular/pull/42800.

A.1. Preliminary work

CSP header support. To enforce Trusted Types, the application needs to be served with the appropriate CSP header. The Angular CLI can be configured to serve custom headers during local development and in end-to-end tests by updating the `angular.json` file, specifically the `projects.site.architect.serve.options` configuration path, as follows.

```
"options": {
  "headers": {
    "Content-Security-Policy":
      "require-trusted-types-for 'script';"
  }
}
```

NPM dependencies. The following packages were added to the Angular.io application to facilitate the migration; `@types/trusted-types`[22] for the TypeScript type definitions for Trusted Types, `safevalues`[23] as a Trusted Types polyfill and utility library, and `tsec` for identifying violations and preventing regressions.

A.2. Identifying TT violations

While Angular framework solves many of the security issues, applications built on top of that framework are not completely free of DOM XSS risks (see Section 3.1).

22. https://npmjs.com/package/@types/trusted-types
23. https://npmjs.com/package/safevalues

With Trusted Types enforced, and a support for TT in the framework, all those risky code areas now surface as TT violations.

We used two approaches to identify TT violations in the Angular.io application; manual inspection by running the application in local development mode with report-only Trusted Types enabled, and static code analysis using tsec.

A total of 14 violations were identified in the application, and were all encountered during manual inspection in local development mode, while tsec was able to identify all 11 violations that occurred in the application's TypeScript code base, but missed violations that occurred elsewhere: in an external TypeScript dependency (`ServiceWorkerModule`), a bundled JavaScript library (`prettify.js`), and a static HTML page (Google Analytics snippet).

A.3. Removing TT violations

Out of the 14 identified violations, 7 were trivial usages of unsafe sinks that could be immediately refactored to use safe DOM APIs, such as replacing an `innerHTML` assignment with a `textContent` assignment or an equivalent construction using `document.createElement`. The remaining violations, listed below, required more work to address.

Inline SVG icons. The application used `innerHTML` to render various SVG icons defined statically as strings within the application bundle. Constant strings can usually be assumed safe as they are fully application-controlled, and cannot be injected by an attacker. To convert constant strings to `TrustedHTML`, a template tag[24] was defined. This tag, which verifies that the tagged string contains no dynamic data interpolation, was then used to convert the markup for each of the SVG icons to `TrustedHTML`.

```
- const svg1 = '<svg xmlns...></svg>';
+ const svg1 = svg`<svg xmlns...></svg>`;
```

Service worker creation. The application created a Service Worker. Since the Service Worker URL is a constant string, we used the same approach as with inline SVG icons, but this time leveraging the existing `scriptUrl` tag from the `safevalues` package that performs identical checks, but produces a `TrustedScriptURL`.

Content rendering. Each Angular documentation page is written in Markdown, compiled and then served as static HTML from the Angular.io server. The Angular.io client code then fetches the appropriate page from the server, and renders it inline using an `innerHTML` assignment. Since the Markdown is authored and reviewed by the Angular team, the static HTML fetched from the Angular.io server was deemed trusted and converted to `TrustedHTML` using a reviewed conversion[25] from the `safevalues` package.

After changing the type of the variable storing the page content to a `TrustedHTML`, the TypeScript compiler reported a type error due a string value being assigned to that variable. Inspecting the code, this assignment occurred when the application failed to fetch the given documentation page, and instead the string contained a user-friendly error message describing the failure.

Type errors from the compiler such as the above are commonly encountered when migrating an application to TT. The solution is often to follow the `string` value upstream and convert it to a `TrustedHTML` in a place where its safety is evident. In this case, however, the source of the string was a function similar to the following:

```
const FETCHING_ERROR = (path: string): string => `<p>
  We are unable to retrieve "${path}" at this time.
</p>`;
```

A keen reader may notice that the `path` variable, containing the path to the requested documentation page, is not escaped. Hence this user-controlled parameter is being injected into the current document as HTML without being escaped by the application. Fortunately, attempts to exploit this apparent XSS vulnerability turned out to be futile due to the way modern browsers encode requested URLs.

However, this near miss is a representative example of how implicit assumptions can lead to the introduction of vulnerabilities and how Trusted Types make these assumptions explicit. In this case, the documentation pages were assumed trusted since they were static HTML originating from the application development team. This assumption was violated when the error page, which was not static HTML, was introduced into the code base. With TT, the developer is required to prove that the error message is indeed trusted HTML. Furthermore, this proof is typically provided near the source of the data where it can be readily assessed by a security reviewer:

```
const FETCHING_ERROR = (path: string): TrustedHTML =>
  htmlFromStringKnownToSatisfyTypeContract(`<p>
    We are unable to retrieve "${htmlEscape(path)}"
    at this time.
  </p>`, 'inline HTML with interpolations escaped');
```

Syntax highlighting. Some documentation pages included code snippets that were highlighted using an old version of the `prettify.js` library.[26] During highlighting, the library assigned an escaped version of the code snippet to `innerHTML` after some minor processing. Since the code snippet was already a `TrustedHTML` due to the changes made in the previous section, the library was patched to pass that `TrustedHTML` directly to `innerHTML`, substituting the processing step for an equivalent use of safe DOM APIs.

Google Analytics loading. The application used a customized Google Analytics script loader. Since the code was defined as an inline script in one of the HTML templates, a single-use dedicated TT policy was created to convert the URL to the Google Analytics script to a `TrustedScriptURL`.

24. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates

25. https://github.com/google/safevalues#reviewed-conversions

26. https://github.com/googlearchive/code-prettify

A.4. Preventing regressions

Besides fixing the Trusted Types violations, additional measures were taken to prevent regressions related to Trusted Types in the future.

Enforcement in end-to-end tests. The end-to-end test suite of Angular.io, which runs in a browser environment that supports Trusted Types, was configured with Trusted Types in enforcement mode. Any Trusted Types violation encountered during a test would cause the test to fail.

Running tsec when linting. Angular.io's linter configuration was updated to run tsec on the entire code base, i.e., `tsec -p tsconfig.app.json`. Code locations that were security reviewed to safely create Trusted Type policies or instances were exempted in the tsec configuration.

Allowlisting policy names. A CSP directive was added to only allow the creation of Trusted Types policies that are used by the application and were reviewed to be safe. The respective CSP directive was `trusted-types angular angular#bundler angular#unsafe-bypass analytics google#safe`.

A.5. Summary

Angular.io is a medium-sized Angular application, with some cases of risky Web API usage, including fetching (originally Markdown) documentation pages as HTML from the server and rendering them inline. It also uses a code prettifier modifying the DOM in a non-obvious manner. Nevertheless, it turned out to be rather straightforward to fix all the Trusted Types violations, despite that the engineer working on the migration had no previous exposure to the code base. The effort only took:

- One day to configure TT enforcement in local development mode and fix all the encountered violations,
- One day to configure TT enforcement in end-to-end tests and fix all the violations there,
- One day to set up tsec and tailor the code commits into a publishable state.

This matches our experience in that while migrating framework code to Trusted Types has some unique challenges that take time and effort to appropriately resolve, subsequent migration of web applications is substantially simpler and faster.