{{alert('CSTI')}}: Large-Scale Detection of Client-Side Template Injection

Lorenzo Pisu
University of Cagliari
Cagliari, Italy
lorenzo.pisu@unica.it

Davide Balzarotti
Eurecom
Sophia Antipolis, France
davide.balzarotti@eurecom.fr

Davide Maiorca
University of Cagliari
Cagliari, Italy
davide.maiorca@unica.it

Giorgio Giacinto
University of Cagliari
Cagliari, Italy
National Interuniversity Consortium for Informatics
Rome, Italy
giacinto@unica.it

***Abstract*—Template engines are software components that enable the creation of reusable HTML elements containing special keywords that can dynamically alter the page's rendering based on the presented data. This technology is widely used in server-side applications and frameworks, and in recent years, it has also gained adoption on the client side through JavaScript frameworks and libraries. Client-Side Template Injection (CSTI) is a vulnerability that occurs when user input is reflected inside a template and rendered as part of it, allowing attackers to inject malicious instructions. This can trick the template engine into executing arbitrary JavaScript code, potentially leading to Cross-Site Scripting (XSS). Despite the widespread adoption of template engines in production websites, a comprehensive study of their characteristics remains absent. In our study, we begin by providing an overview of the main features of template engines, highlighting attributes that play a crucial role in escalating CSTI to XSS. We then use these extracted characteristics to develop a systematic methodology for detecting CSTI vulnerabilities. Based on this methodology, we create an automatic CSTI detection tool, `CSTI-Alert`. By running `CSTI-Alert` on the Tranco top 1 million domains, we identify 532 CSTI-vulnerable domains, with 72% directly leading to XSS through GET parameters or CSRF. Finally, we discuss potential approaches to defend against CSTI based on the result of our semi-automatic exploitability analysis.**

***Index Terms*—web security, client-side template injection, large-scale detection**

I. Introduction

Arbitrary code execution in client-side JavaScript applications is a critical security threat to modern websites, primarily through attacks such as Cross-Site Scripting (XSS) [1]–[10]. However, the evolving landscape of client-side libraries and frameworks has paved the way for new techniques to inject JavaScript code into websites. A popular technology widely used in server-side applications (and in recent years, also integrated into client-side applications) is the template engine, a piece of software designed to dynamically render data inside a predefined set of templates.

Template engines can be utilized as standalone libraries or integrated with JavaScript frameworks such as jQuery [11], Angular [12], and Vue [13]. Originally designed for server-side use, they serve the purpose of separating the logic and presentation layers by allowing developers to define HTML templates that dynamically present data [14]. With the increasing sophistication of client-side applications, which are composed of complex, dynamic User Interface (UI) components, template engines have become useful for defining reusable pieces of code. An advanced example of client-side template engine usage is the so-called Single-Page Application (SPA) [15], in which template engines are employed to continuously fetch data from the server-side using JavaScript to render it without the need to reload or redirect the user's page.

After Kettle showed the impact of Server-Side Template Injection (SSTI) [16], the dangers associated with this technology became evident. Since templates are not merely passive components but, when rendered, can manipulate objects and functions, they can become a potential source of vulnerabilities. Moreover, it was shown that SSTI can lead to Remote Code Execution (RCE) [17], [18], a critical security issue that can cause a complete takeover of the server by an attacker. Despite the significant impact of template injection, popular websites and frameworks are still found to be vulnerable to SSTI [19]–[22], showing that this vulnerability needs to be analyzed carefully.

Notably, Client-Side Template Injection (CSTI) was uncovered by Heiderich even before SSTI, under the general category of attacks against JavaScript Model-View-Controller (MVC) [23]. However, it was only after the research conducted by Heyes [24], [25] in 2016 and 2017 on injection attacks in Angular, Vue, and Mavo frameworks that this vulnerability became known as CSTI.

CSTI is an attack vector that occurs when the user input is parsed by client-side engines as part of their templates, allowing users to exploit its functionalities for malicious purposes.

An attacker can take advantage of this vulnerability and, by injecting specific payloads into template expressions, they can manipulate the client-side rendering process, often gaining the possibility to execute malicious JavaScript code. CSTI can be much more subtle than SSTI because many client-side template engines work differently than the server-side counterpart, making CSTI more difficult to prevent. Furthermore, CSTI does not necessarily need the injection of HTML or HTML-like tags: Even simple expressions such as {{`alert(1)`}}, which are not seen as malicious by general-purpose sanitizers, can trigger arbitrary JavaScript code execution.

Although the web security community has been discussing CSTI for many years now, the prevalence and large-scale detection of this vulnerability remains a completely uncharted topic in research. Notably, no measurements or large-scale analysis of CSTI have been conducted, and only one tool for CSTI detection has been developed (ACSTIS) [26], which supports only a single template engine (Angular) and is no longer functional due to a lack of updates. However, many bug bounty reports [27], [28] and CVEs [29]–[33] related to CSTI demonstrate a growing awareness among security practitioners on the topic.

In this paper, our goal is to provide a systematic overview of template engines and how their improper usage can lead to CSTI. To this end, we first perform a survey of the most popular client-side template engines. For each of them we devised a special payload that could be injected to verify whether a website using that engine can be vulnerable to CSTI.

Next, we present our methodology for detecting CSTI in the wild. Given a website URL, we crawl it to the desired depth, extracting a list of URLs for analysis. The analysis of each URL begins with detecting the template engine in use, which is determined by the presence of objects instantiated by the template engine. For each detected template engine, we inject a specific payload into the page, interacting with its components and checking whether the payload is executed. For each vulnerable website, we also conduct an automatic exploitability study, simulating an attack scenario to assess whether arbitrary JavaScript code can be executed on the target website.

Finally, to estimate the adoption of template engines and assess the presence of vulnerable websites, we deployed our tool, `CSTI-Alert` [34], on the Tranco top 1 million domains. To the best of our knowledge, this is the first large-scale analysis of CSTI. We identified 532 vulnerable domains, with 385 of them exploitable to achieve XSS. We further analyzed the remaining 178 non-exploitable domains to evaluate their defenses against CSTI, highlighting interesting scenarios where the vulnerability could not be exploited. Additionally, we discuss defenses against CSTI, showing that existing sanitizers are inadequate for mitigating this threat.

Our contributions can be summarized as follows:

- We conduct the first comprehensive and systematic study of Client-Side Template Injection (CSTI), covering vulnerability, injection techniques, detection, prevalence, impact, and defenses.
- We conduct an in-depth study of the most popular template engines. By analyzing their syntax, behavior, and possible payloads to trigger XSS, we designed a methodology to automatically detect CSTI vulnerabilities.
- We implemented our methodology in a tool, `CSTI-Alert`. To the best of our knowledge, no other tool is capable of performing this kind of detection on such a large number of template engines.
- We instantiate `CSTI-Alert` [34] to assess the prevalence and impact of CSTI on the Tranco top 1 million, finding that vulnerable domains either use Angular or Vue as their frameworks. Finally, we discuss defenses and mitigation strategies against CSTI, analyzing both the current possibilities and future directions for the prevention of this threat.

II. Background and Related Work

In this section, we provide an overview of the background concepts, explaining how client-side template engines work in II-A and how CSTI occurs in II-B.

A. Template Engines

Template engines are software components designed to generate dynamic HTML views from templates, allowing developers to present user data without rewriting HTML code every time the data changes. Templates typically contain a mix of static HTML and special symbols that the engine interprets as delimiters between HTML and executable instructions. The most common delimiter symbol, used by popular engines like Jinja2, Handlebars, and Angular, is the double curly brackets ({{ }}). Within these delimiters, variables or expressions are placed, which the engine replaces with user-provided data to produce the final HTML content.

Template engines can operate on both the server-side and client-side, with some JavaScript engines supporting usage on both ends. Although template engines were originally developed for server-side applications, their adoption in client-side frameworks has grown significantly. Client-side template engines function similarly to server-side ones but differ in the way templates are rendered. Specifically, client-side rendering can occur through three main approaches:

- **Declarative**: Similar to server-side rendering, templates are defined first and then programmatically fetched and compiled.
- **HTML Tag Attributes**: Specific HTML attributes can trigger the template compilation process.
- **Tag Mount**: The engine is attached to a tag that contains the template, automatically rendering the content.

We will provide a detailed analysis of these three client-side rendering approaches in Section IV-B.

B. Client-Side Template Injection

When the user input reaches a template definition and is parsed as part of the template itself, a template injection vulnerability occurs. In server-side template engines, this vulnerability is called SSTI, which often leads to RCE. On the client-side, this vulnerability is known as CSTI and can potentially

lead to XSS, compromising the security of legitimate users visiting the website.

The possibility of injecting template syntax can stem from improper handling or insufficient validation of user input on the server, which then reflects the input back to the client. This improper reflection allows an attacker to craft input that is interpreted as part of the template, enabling the execution of unintended commands.

The following example demonstrates how CSTI can arise in Angular using a PHP backend:

```
<html ng-app>
<head>
<script src="angular.js"></script>
</head>
    <body>
        <p>
            <?php
            $username = $_GET['username'];
            echo htmlspecialchars($username);
            ?>
        </p>
    </body>
</html>
```

Listing 1: CSTI Vulnerability Example

The code in Listing 1 shows a page in which the angular scope is the full HTML code (as the `ng-app` attribute is used in the `<html>` tag). The PHP code retrieves a username from the GET parameters and echoes it using the sanitization function `htmlspecialchars`, which prevents malicious HTML injection. However, this sanitization does not account for curly brackets, which Angular interprets as part of its template syntax, thereby exposing the application to XSS.

For instance, an attacker can inject the following payload `{{constructor.constructor('alert(1)')()}}` to achieve arbitrary JavaScript execution and compromise website users by stealing cookies or performing sensitive actions on their behalf. CSTI can also arise from poor coding practices in client-side JavaScript. If a template rendering function is used similarly to a sink (such as `eval` or `innerHTML`), a DOM-based CSTI can occur. We will explore CSTI scenarios in detail in Section V-A, focusing on both server-side reflections and DOM-based instances.

III. Research Questions

The paper aims to answer the following research questions:

(RQ1) What are the characteristics of the most popular template engines? Countless template engines have been developed, each with its own set of features and potential exploits. Despite this, a comprehensive analysis of the characteristics of template engines has never been carried out.

(RQ2) How can we detect reflected CSTI in a black-box scenario? Given the numerous cases of CSTI and the widespread adoption of template engines, the automatic detection of CSTI is essential. We present a systematic approach for CSTI detection in a black-box environment, where the primary cause of CSTI is the reflection of user inputs.

(RQ3) What is the prevalence of CSTI in the wild? How can websites defend against it? Although CSTI has been known for almost ten years, there is still no measurement of its prevalence, impact, or code patterns in the wild. In this paper, we aim to quantify the prevalence of CSTI, identify vulnerable behaviors, and examine their impact to shed light on the possible causes and factors behind this overlooked vulnerability. Moreover, we discuss possible defenses against CSTI, starting from scenarios where the vulnerability is present but not exploitable.

IV. Survey of Template Engines

The first part of our work aims to answer RQ1, collecting a set of popular template engines and performing a systematic evaluation of their properties. We selected the 26 most popular template engines based on GitHub stars, searching GitHub with the keyword ”template engine” and filtering by the JavaScript language. We excluded frameworks such as React [35], Svelte [36], Marko [37], and Ember [38], as performing CSTI on them is either not possible or extremely rare. This is primarily because these frameworks precompile their templates, preventing them from being compiled at runtime with user-supplied strings. While this applies to all four excluded frameworks, some have additional distinguishing features. Specifically, React uses JavaScript XML (JSX), which is not a template engine. Svelte and Marko have their own templating languages, and Ember uses Handlebars to handle templating features.

For each template engine, we created a local testing environment and developed a simple app vulnerable to a reflected CSTI. This helped us identify and extract the functions called during the template rendering process, if any are visible (i.e., if the functions are not anonymous) [39]. This set of function calls will be useful later when we want to assess if a target website is actually using the template engine or merely importing the library without utilizing it. If the template engine uses only anonymous functions, this process cannot be used for detecting the engine’s usage. In such cases, we rely on the presence of these functions in the JavaScript code of the page or on the presence of template-related attributes inside the tags. The selected engines, along with the result of our analysis, are summarized in Table I.

A. Extracted Characteristics

The key information that we extracted for each template engine is summarized in the following five features.

- **Syntax**. The syntax is often similar across various engines (e.g., curly brackets {{ }}), although it can slightly vary. It is crucial to understand which symbols are used by each engine to recognize instructions. From a detection perspective, this information can be leveraged to build payloads that the engine parses when a CSTI vulnerability is present. This information is easily found in the engine’s documentation.
- **Detection Payload**. To detect CSTI, we need to inject a payload that produces an easily identifiable result on the page. Since CSTI occurs when the user input is executed as part of a template, we can inject specific operations and

TE	Ref.	Popularity				Decl.	mount	attr.	TE object	Detection Payload	XSS Payload
		★			npm						
vue	[13]	208.2k	33.7k	-	23M		✓	✓	Vue	{{12345*54321}}	{{constructor.constructor('alert(1)')()}}
angular	[12]	96.8k	25.8k	3.8M	14M		✓	✓	angular	{{12345*54321}}	Version Dependant
alpine	[40]	28.9k	1.2k	179	86.8k			✓	Alpine	12345*54321	alert(1)
underscore	[41]	27.3k	5.5k	4.5M	50M		✓		_.template	<%=12345*54321%>	<%=alert(1)%>
pug	[42]	21.7k	1.9k	676k	6M	✓			pug	#{12345*54321}	#{alert(1)}
lit	[43]	19.1k	951	145k	8.6M	✓			litHtmlVersions	${12345*54321}	${alert(1)}
handlebars.js	[44]	18.1k	2k	4M	60M	✓			Handlebars	{{this}}	-
mustache.js	[45]	16.5k	2.3k	-	21M	✓			Mustache	{{.}}	-
art-template	[46]	9.8k	2.6k	16.8k	151.6k	✓			template	{{12345*54321}}	{{constructor.constructor('alert(1)')()}}
nunjucks	[47]	8.6k	641	271.3k	3.9M	✓			nunjucks	{{12345*54321}}	{{({}).constructor.constructor("alert(1)")()}}
ejs	[48]	7.8k	845	13M	75.9M	✓			ejs	<%=12345*54321%>	<%=alert(1)%>
swig	[49]	5.8k	1.2k	18	123.1k	✓			swig	{{12345*54321}}	{{alert(1)}}
hogan.js	[50]	5.1k	427	87.8k	2.5M	✓			Hogan	6705{{!csti}}92745	-
doT	[51]	5k	1k	35.8k	1.9M	✓			doT	{{=12345*54321}}	{{=alert(1)}}
jquery-tmpl	[52]	3.2k	1k	-	-	✓			$.tmpl	${12345*54321}	${constructor.constructor('alert(1)')()}
dustjs	[53]	2.9k	479	1	57.8k	✓			dust	{{.}}	-
mavo	[54]	2.8k	177	10	56		✓	✓	Mavo	[12345*54321]	[self.alert(1)]
jsrender	[55]	2.6k	340	3.3k	65.3k	✓			jsrender	{{:12345*54321}}	{{:"".toString.constructor.call("alert(1)")()}}
twig.js	[56]	1.8k	275	-	1M	✓			Twig	{{12345*54321}}	-
regular	[57]	1k	150	4	238	✓			Regular	{12345*54321}	{constructor.constructor('alert(1)')()}
transparency	[58]	967	112	236	208		✓	✓	Transparency	-	-
pure	[59]	922	92	119	1k		✓	✓	$p	-	-
Juicer	[60]	914	260	555	5.7k	✓			Juicer	${12345*54321}	${alert(1)}
ICanHaz.js	[61]	837	126	136	2.3k	✓			ich	6705{{!csti}}92745	-
tempo	[62]	708	72	-	-		✓	✓	Tempo	{{this}}	-
template7	[63]	658	164	2.9k	11.9k	✓			Template7	{{js "12345*54321"}}	{{js "alert(1)" }}
squirrelly	[64]	651	83	2.1k	98.4k	✓			Sqrl	{{12345*54321}}	{{alert(1)}}
jquery-template	[65]	603	200	49	-		✓	✓	loadTemplate	-	-
Markup.js	[66]	319	53	512	22.7k	✓			Mark	{{.}}	-

Legend:★=**GitHub Stars;** =**GitHub Forks;** =**GitHub UsedBy;** npm=**NPM Monthly Downloads; decl.=declarative; mount=tag mount; attr.=tag attribute**

TABLE I: Overview of the selected client-side template engines and their characteristics. Note that some of these engines (e.g., Handlebars) can also be used on the server-side, but we focus on their usage on the client side.

verify whether they have been executed. Therefore, we primarily use mathematical operations, as they generate predictable, unique, and easily detectable results on the target page. For instance, multiplying `12345*54321` yields 670592745. Testing for the presence of this number on a set of Tranco top 10k websites revealed that none of them contain this sequence, making this payload suitable for detecting the execution of our detection payload. For template engines incapable of performing mathematical operations, we check if any default objects are present in the template context and utilize them. In Handlebars, for example, the `this` object is available, while others (like Mustache) provide a dot (`.`) object. Using these objects produces the string `[object, Object]`, which rarely appears on webpages. If neither of the above techniques applies to the engine, we use template syntax that allows for writing comments inside the templates (e.g., Hogan templates can contain {{`!comment`}}). Since comments are removed after the template is compiled, we can insert the comment within our detection string. For example, the string `6705{{!comment}}92745` becomes `670592745`, enabling the detection of CSTI. If none of these techniques apply, it indicates that the engine is structured in such a way that CSTI is not feasible. For example, `PureJS` [59] associates data with tags but only allows data presentation without the capability to access object attributes or perform mathematical operations.

- **Rendering Type**. As mentioned in Section II-A, template engines operate in three main ways: declaratively, through tag attributes, or via tag mounts. We gathered this information by setting up a testing scenario that utilizes the engine. Understanding this helps us estimate the attack surface of a website using a particular engine. If it follows a declarative paradigm, the attack surface is limited to reflections within the template string. However, if it uses tag mounts or attributes, reflections can also occur within tag content or attributes.
- **Template Engine Object**. Since most template engines are implemented as client-side libraries, they typically export an object containing all the functions and attributes needed to compile and render templates. Identifying this object is crucial to determine if a target page uses the engine. Although its presence alone may not confirm the use of the engine, its absence confirms that the website does not import the engine library.
- **XSS Payload**. Escalating CSTI to arbitrary JavaScript code execution is not always possible. We analyzed each engine in a testing scenario to assess its capability in this regard. If the engine allows it, we extract and document an example of a payload that achieves XSS. Notably, in Angular, the payload depends on the version. Online resources authored by Heyes and Heiderich provide comprehensive lists of working payloads for each version of the framework [67]. Vue also features slight differences in its XSS payloads between V1, V2, and V3, but we decided to show the most common payload, which works

in V2.

B. Template engine analysis

To extract the syntax and functionalities of a template engine, we analyze its documentation. The documentation typically includes details on the possibility of performing mathematical operations, the presence of default objects, and how the template engine operates. Additionally, we extract example code snippets from the documentation that we can reuse to set up a vulnerable scenario. We also organize the way in which the target engine parses and renders templates in three main categories: (i) tag attributes; (ii) tag mount and (iii) declarative. More than one category can apply for the same engine, i.e. Angular can operate both with tag attributes and tag mount.

Declarative. The most common way for template engines to work is by declaring a template, either inside a JavaScript variable or as a tag inside the HTML code. The template is then passed to the engine function or class that parses it to identify specific keywords (such as {{}}) and executes instructions inside or replaces the variables with their corresponding values. Notably, with this mode of operation, the frontend developer narrows the possibility of CSTI by using very specific areas of the page to declare and use the template engines, reducing the possibility of introducing user inputs inside the templates. This type of template engine is the most common among those under analysis, accounting for 30 out of 40 (66%). An example of a declarative engine is `Handlebars`, in which the templates are often declared inside script tags, following is a brief example.

```
<script id="userTemplate" type="text/
    x-handlebars-template">
  <p>Hello {{user}}!</p>
</script>
```

Listing 2: Script tag containing the declaration of a `Handlebars` template

Using the HTML code in Listing 2, the browser will not execute the script element, since the type attribute suggests that it does not contain JavaScript code. However, the content of the tag can be retrieved and compiled using Handlebars. The following JavaScript code shows how.

```
var template = document.getElementById("
    userTemplate").innerText
output = Handlebars.compile(template)({user: '
    test'})
```

Listing 3: Compile process of a `Handlebars` template

The code in Listing 3 retrieves the text content of the tag `userTemplate`, which contains the template to be compiled. Using the document object `Handlebars`, we call the function `compile` by passing the template string as an argument. The call to `compile` returns another function that receives an object representing the available data in the template context. The output variable will now contain the string `<p >Hello, test!</p>` and can be placed inside the page. Notably, we could also have declared the template as a JavaScript string directly inside the code.

Tag Attributes. Many libraries enrich HTML tags with active attributes, i.e., attributes used by JavaScript to dynamically change the page behavior. Template engines are no exception, using specific tag attributes or even classes to perform template rendering operations. This mode of operation widens the attack surface because a user input that is reflected inside a tag as a class or attribute can be manipulated to trigger a template injection. Certain attributes are also valid for child elements (e.g., an `h1` tag inside a `div` with an active template engine attribute), meaning that a user input that is reflected inside a child element from the back-end will be parsed as a template instruction. This method of rendering templates is supported by 8 out of the 30 engines analyzed (26%). Angular is one of the most popular engines that use this kind of mechanism. In the following, we report an example that shows the main tag attribute that can be used in `Angular` to invoke the template engine.

```
<html>
    <head>
        <script src="angular.js"></script>
    </head>
    <body>
        <div ng-app>
            <input type="text" ng-model="name">
            <h1>Hello, {{ name }}!</h1>
        </div>
    </body>
</html>
```

Listing 4: How the `Angular` `ng-app` and `ng-model` attributes work

In the HTML code of Listing 4, we start by declaring our Angular app scope by adding the attribute `ng-app` to the `div` tag. We then use the Angular attribute `ng-model` to bind the value of an input tag to the template variable `name`. Since we are within the ng-app scope, the template syntax {{ `name` }} will be automatically replaced with the value entered by the user in the input tag.

Tag Mount. Embedding template keywords directly inside the HTML code of the page can be convenient, making the page both a template, when it is first loaded, and the rendered results, when the engine has finished parsing it. Engines can be mounted to watch a specific element and treat it as a template, rendering the keywords and showing the result directly inside the HTML code. This can be dangerous if the backend puts an untrusted input inside these template tags. In practice, many vulnerable websites mount the template engine inside very broad tags such as `html` or `body`, making it very easy for a user input to become part of the template. This strategy for rendering templates is supported by 8 out of the 30 engines analyzed (26%). `Vue` is an example of a template engine that is often used in this way. In the following, we report an example that shows a way to use `Vue` with this kind of technique.

```
<div id="app">
  <p>{{ user }}</p>
</div>

<script type="module">
import { createApp } from 'vue'

const app = createApp({
  data() {
    return {
      user: "test"
    }
  }
})

app.mount('#app')
</script>
```

Listing 5: How the `Vue` tag mount works

The code in Listing 5 contains both the HTML and the JavaScript needed to perform a template rendering with `Vue`. The template is contained inside the div tag and renders the variable `user`. The JavaScript code creates an `app` object which contains the data available to the templates, in this case the `user` variable with value `test`. To attach and, therefore, render the template inside the page we call the `mount` function, we pass as argument a selector that identifies the tag to which `Vue` will be attached.

These three modes of operation can impact how easy it can be for CSTI to arise, making it a fundamental feature to understand common pitfalls when using template engines.

C. Arbitrary Code Execution Prevention in Template Engines

While our focus in this work is on the exploitation of CSTI, this subsection takes a step back to compare how different template engines handle the risk of arbitrary code execution, which directly impacts exploitability. Notably, not all client-side template injections lead to XSS, and one of the main reasons is that not all template engines allow the execution of arbitrary JavaScript code within the templates. Below, we discuss three template engines, starting with one that (to the best of our knowledge) prevents arbitrary code execution, followed by one that attempted but failed, and finally, one that permits arbitrary code execution.

Handlebars is a template engine focused on speed, simplicity, and security. However, there have been security vulnerabilities in this engine due to the ease of performing a JavaScript sandbox escape. Nevertheless, Handlebars succeeded in preventing untrusted inputs from causing arbitrary code execution within templates. It achieves this by, first, restricting the possible operations that can be performed inside a template and, second, by prohibiting access to prototype properties of objects [68], [69]. Since JavaScript sandbox escapes often exploit prototype properties, this restriction effectively prevents arbitrary code execution.

Angular is an example of a framework that attempted to prevent XSS from CSTI but failed due to the known challenges of sandboxing JavaScript code [70]. From version 1.0 up to version 1.6, Angular tried to limit the possibility of executing arbitrary code using sandboxes of increasing complexity. However, all of these sandboxes were eventually breached, leading to the decision to abandon this approach. Consequently, Angular announced that starting with version 1.6, sandboxes would no longer be present [71].

Vue is one of the most popular frameworks according to GitHub stars; however, it does not provide any effective security features related to JavaScript code execution. This approach is shared with many other engines under analysis. When using such frameworks, developers must be extremely cautious to ensure that untrusted user input cannot reach the templates within the page. Although this approach is understandable given the difficulty of sandboxing JavaScript, our results show that many vulnerable websites are exploitable due to this issue.

V. Detection Methodology

In the second part of the paper, we address RQ2 by defining the possible causes of CSTI (V-A), how to assess whether a webpage is using a template engine (V-B), how to generate a payload that can be used to detect CSTI (V-C) and how to inject it inside the page (V-D). Finally, we put everything together into our tool `CSTI-Alert`, describing its modules and the flow it follows to detect CSTI (V-E). Figure 1 shows a summary of how the methodology works.

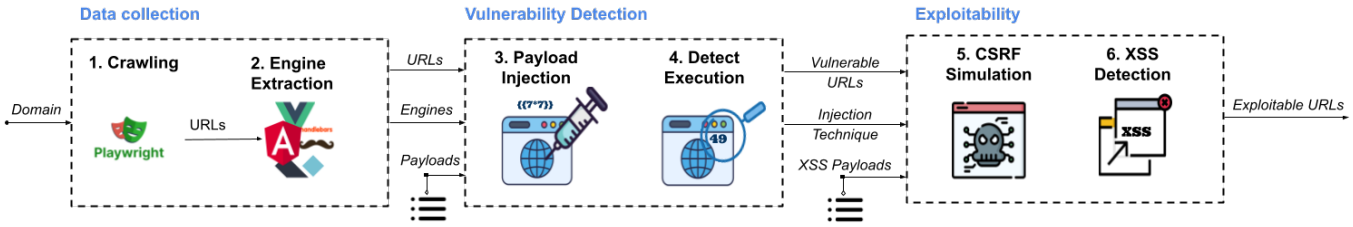
A. Potential Causes of CSTI

There are many potential ways in which CSTI can arise, depending both on the characteristics of the template engine and on the vulnerable coding practices that developers can use. We identify two main categories (server-side reflection and DOM-based), and for each of them, we list subcategories.

Server-Side reflection. In this category, we include cases where CSTI arises from a server-side reflection of user input. This reflection can occur in different parts of the page and with varying scopes (e.g., inside a tag attribute, within a template, or inside a tag). In the following, we analyze the different contexts in which this reflection can occur.

- **Inside a tag without HTML injection**. As shown in Section II-B, CSTI can occur when the server-side reflects user input inside a client-side template. This is the most common cause of CSTI found in the wild, highlighting both the lack of validation for user inputs against CSTI and the ease with which developers can make mistakes when using a template engine or client-side framework. Notably, this scenario can occur even if the server uses an HTML sanitizer.
- **Inside a tag with HTML injection**. In cases where the injection can also include HTML tags, it may be possible — depending on the template engine — to use these tags to create malicious templates that are automatically parsed by the engine. For example, in Angular, we can inject a tag with the `ng-app` attribute and place a template inside this tag to trigger XSS without using unsafe tags like `<script>` or attributes like `onload` (which we assume are filtered by sanitizers such as DOMPurify).

Fig. 1: CSTI Detection Methodology



- **Inside the tag attributes**. Some template engines, as discussed in Section II-A, have special attributes that can be used to declare templates, perform operations, or initialize template data. If a user can inject arbitrary attributes (e.g., by escaping the current attribute with quotes), they might be able to exploit CSTI using these engine-specific attributes. In some cases, the reflection may already occur inside a template attribute. Even if it is not possible to escape this context to inject other attributes, CSTI may still be possible if the attacker is in a context where the attribute can execute template operations. Some template engines, such as Angular, also allow template directives inside the `class` attribute of a tag, meaning that a reflection within such attribute can be escalated to CSTI (e.g., using the keyword `ng-init:<payload>`).

DOM-Based. In this category, we examine cases where CSTI can arise from poor coding practices in the client-side JavaScript code that handles the template rendering process.

- **User-controlled templates**. If JavaScript code that renders a template dynamically inserts user input into the template, CSTI can occur. This scenario can involve common sources of DOM-based vulnerabilities, such as URL parameters or fragments, but in this case, the sink is the string that is subsequently passed to the template render function.
- **Common sinks (innerHTML)**. Common XSS sinks, such as innerHTML, can also be sources of CSTI. Even if an HTML sanitizer is used to filter malicious input, characters like {{ }} are typically not sanitized. Additionally, if sanitizers such as DOMPurify are used, it is possible to inject specific tags that are not considered harmful and add dangerous attributes to them. These attributes might be interpreted by the engine as directives, thus triggering CSTI.
- **Uncommon sinks (innerText)**. With CSTI, attributes and functions that were not previously considered dangerous can become potential sinks. For example, the `innerText` attribute is typically not considered a sink because it does not evaluate HTML tags but treats the content as a plain string. However, in the case of CSTI, this can be problematic if the tag whose innerText is being manipulated is later parsed as a template by the library. Due to the asynchronous nature of JavaScript, templates are not always parsed immediately when the page loads. If the content of the templates is modified using user input before the engine parses it, the user input might be interpreted as part of the template and subsequently rendered.

B. Detecting a Client-Side Template Engine

The advantage of client-side templating vs server-side is that if a template is used, the library containing its code needs to be imported. This import can target a local copy of the library or Content Delivery Networks (CDNs) [72]. Detecting the presence of such imports can be one way to assess the presence of a client-side template. However, it can be prone to errors, especially if the JS file has been renamed or the source repository is unknown. The consequence of a template engine being imported is the presence of global objects that can be used to compile, create or simply obtain information about templates. For example, Angular provides a global object called `angular`: the presence of this object itself is a certainty that the library has been imported. By analyzing our set of selected template engines, we find that this kind of object exists for all engines, meaning that we can assess whether a website imports a specific template engine by checking the presence of this object. However, merely detecting a global template engine object does not guarantee that the engine is being used. Common scenarios where the object is present but not utilized include: **(i)** Libraries providing multiple utilities, like Underscore, where the template engine is just one of many features, but it may not be used. **(ii)** Websites that import the library on all pages but only rendering templates on specific ones. To perform a more accurate estimation of the effective usage of a template engine, we consider the following additional heuristics:

- **Function calls**. Template engines provide functions for compiling and rendering templates. We identify and extract the functions involved in the rendering process for each engine. If a page contains calls to these functions, it strongly suggests active template engine usage.
- **Page scripts analysis**. Some template engines use anonymous functions, which do not display names when extracted. In these cases, we inspect the page scripts for interactions with the global template engine object or its attributes. This includes looking for function calls related to template compilation and rendering.

- **Tag attribute detection**. This method applies to template engines that use custom attributes for template rendering (e.g., Alpine uses `x-` attributes). By detecting the presence of these attributes, we confirm the usage of the engine. Additionally, script tags defining templates can be detected by checking if the `type` attribute follows the pattern `template/<engine name>`.

C. Payload generation and reflection

To detect CSTI, our methodology is based on generating a payload, injecting it into the page, and then checking if it was reflected and executed. To generate a payload that can be used to detect CSTI, we need first to obtain some context information about the engine for which we are creating it.

- **Engine syntax**. Each template engine can use different symbols to mark expressions, for example, Angular and Vue use double curly brackets, which is one of the most common syntaxes.
- **Mathematical expressions**. Despite being a seemingly granted ability, not all engines allow for the execution of mathematical operations. Angular allows the execution of a simple {{`7*7`}} rendering a 49, while other engines (such as Handlebars) do not allow this kind of operation to be executed. In the case of Handlebars, we can use the payload {{`this`}} which will result in the string `[object Object]`.

For each engine, we need this information to correctly generate a payload that we can inject inside the page. Firstly, we need to adhere to its syntax, then we exploit mathematical operations to create a payload of which the result is easily identifiable on the page (e.g., a long number or a particular string). We consider a page vulnerable to CSTI if injecting a payload `P` we obtain a reflected string `S` which is the result of the execution of `P`.

D. Payload injection and submission

An important aspect of CSTI detection depends on the following question: How many ways to inject a payload can a page have? The answer is not trivial, since form tags are not the only way to submit user data. We find 4 different techniques for submitting our detection payload inside a target page.

- **Form submission**. Form tags are the standard way to trigger the parsing of user input. To test for CSTI, all forms on a page should be identified and the detection payload should be inserted into their input fields. It is important to consider specific formatting requirements, such as ensuring that email fields receive a valid input format (e.g., {{`12345*54321`}}`@test.com`.
- **Buttons**. Buttons can trigger JavaScript functions that collect user inputs from the page and send them to an endpoint. This is common in search bars that use a button to initiate server-side requests for search results without utilizing a traditional form tag.
- **Links**. Although rare, `a` tags can sometimes trigger JavaScript code execution, leading to the submission of input. These edge cases were observed in a few instances during our analysis
- **JavaScript events**. Events such as `onclick` and `onKeyPressed` can be used to trigger input submission. For example, a search bar might trigger a server-side request when the user presses the enter key. To account for this, it is necessary to select each input field, enter the detection payload, and simulate such events.

We also emphasize that the above injection and submission techniques are not harmful to the website or its users, as they do not execute dangerous operations or generate a high volume of requests to the server.

E. CSTI Detection Tool

To check the presence of CSTI in a target website, we created a tool that performs black box detection. The tool is composed of five main modules that we describe in the following.

- **Crawler**. The crawling module collects links from the website under analysis. It can be configured to either limit crawling to the main domain or include subdomains. The depth of crawling is adjustable, with depth 2 used in our analysis, although users can choose to go deeper if needed.
- **Engine Detection**. Before checking for CSTI vulnerabilities, the tool identifies the template engine present on each page. This is done by detecting global objects specific to template engines (e.g., Angular). Since different pages on the same website might use different engines, this check is repeated for every new page analyzed.
- **Payload Injection**. To detect CSTI vulnerabilities, the tool checks if the input is embedded in the template rendering process. Each template engine has its own syntax, so engine-specific payloads are used. These payloads include template syntax and instructions to perform operations, allowing the tool to (i) verify if the operation is executed and (ii) generate an uncommon output, making the detection more reliable.
- **Payload Submission**. After injecting the payload, the tool triggers processing, which can occur on either the client or server-side. For example, there might be a client-side script that parses the payload and reflects it within the page, or a server-side endpoint that receives the payload and subsequently reflects it inside the page. As detailed in Section V-D, this can happen in various ways, including form submissions, button clicks, link activations, or JavaScript event triggers. We handle the first three cases by executing the JavaScript functions that trigger them, namely the `submit` function for forms and the `click` function for `button` and `a` tags. The last case is handled using the Playwright [73] function `press` to generate the desired event.
- **Reflection Check**.The final step involves checking if the payload was reflected and executed on the page. Once the page is fully loaded, the tool searches for the result of the

payload's execution. If the expected result is found, the website is flagged as vulnerable to CSTI.

VI. CSTI in the Wild

In this third part of the paper, we address RQ3 by quantifying the prevalence and impact of CSTI on the top 1 million websites using the Tranco list [74] from October 13, 2024 (ID: YXZ5G). We performed a crawl with a maximum of 5 URLs per website in the top 1M and then examined 3 sub-ranges (top 100k, middle 50k, and bottom 50k) by crawling them at a depth of 1. Additionally, we ran the tool Amass [75] for 7 days on the top 100k, crawling the resulting domains and subdomains at a depth of 1. We also analyzed the top 5k and middle 5k by crawling them at a depth of 2. Our goal is to provide insights into the distribution of vulnerable domains within the list and to examine how crawling at different depths affects the number of vulnerable websites we discover.

In the first part of this section, we present a study on the usage of template engines in the top 100k websites VI-A. Next, we discuss the prevalence of CSTI within our pool of websites VI-B. We then present the results of our exploitability study on the vulnerable websites VI-C. Following this, we examine the distribution of Angular versions in the wild VI-D. Finally, we discuss defenses against CSTI VI-E.

A. Template engine usage

Before detecting CSTI, we narrowed our scope to websites that actively use a template engine. As discussed in Section II-A, client-side template engines can be initially identified by checking for exported global objects when the library is loaded. Notably, in Table II the count of URLs and domains using a template engine is derived from this heuristic. However, to provide a more accurate estimation of template engine usage, we conducted an additional study on the top 100k URLs, using the methodology discussed in Section V-B to estimate the actual number of websites using a template engine. Table III shows the results of this analysis. It is interesting to observe that 51.7% of the domains that were collected using only the object detection actually use a template engine.

B. CSTI Prevalence

In Table II, we present the number of domains found to be vulnerable to CSTI within each Tranco range that we selected, along with a breakdown of the specific template engines used on these vulnerable websites. Additionally, we compare the number of vulnerable domains detected against the total number of domains analyzed.

Our findings indicate that the depth of the crawling process significantly impacts the number of detected vulnerabilities. Specifically, performing a full depth-1 crawl revealed 41 additional vulnerable domains compared to analyzing only the first 5 URLs of the top 1 million websites. Similarly, enabling subdomain analysis led to the discovery of an additional 23 vulnerable domains.

However, it is important to note that increasing the crawling depth drastically expands the number of URLs to analyze. For example, in the top 5k (depth 2), we observed an average of 264 URLs per domain, whereas in the top 100k (depth 1), this ratio dropped to 89 URLs per domain. This demonstrates that deeper crawling, while more thorough, can quickly become infeasible due to the exponential increase in URLs to process.

To enhance our crawling strategy, we used Amass to identify additional subdomains, enabling a more comprehensive examination of each domain's attack surface. This approach not only uncovered subdomains using template engines, but also revealed previously unnoticed endpoints vulnerable to CSTI. Notably, extracting subdomains and then performing a depth-1 crawl efficiently expands the attack surface while maintaining a relatively low number of URLs per domain, balancing thoroughness and scalability.

Regarding the distribution of vulnerable template engines, our analysis shows that 70% of the vulnerable domains utilized Angular as the template engine, and the remaining 30% were found to use Vue. No vulnerabilities were detected for the other template engines under analysis. This result is particularly noteworthy, as it suggests that Angular and Vue, despite being among the most popular template engines, are also the most prone to CSTI vulnerabilities. One reason for this could be their design choice of parsing templates directly from the page tags, which makes it easier for developers to unintentionally reflect user inputs into templates without proper sanitization.

It is important to note, however, that identifying a vulnerable domain does not necessarily imply that it is exploitable. In the next section, we detail our approach to testing for exploitability and present our findings on the actual impact of these vulnerabilities.

C. Exploitability

To assess the exploitability of a target webpage for CSTI, we need to confirm two conditions: the ability to execute arbitrary JavaScript code and the feasibility of performing a Cross-Site Request Forgery (CSRF) attack that forces a victim to execute the malicious payload involuntarily. To achieve this, we designed a semi-automatic procedure that, given a vulnerable URL and the location of the vulnerable form or button, determines whether the page is exploitable. We emphasize that merely injecting the payload (as we did to verify the presence of SSTI) is not sufficient to determine the exploitability of the website. Servers may filter cross-origin requests using headers or tokens, or they might apply runtime filters that prevent payload execution.

Our approach involves a module that automates this verification process. Negative results are manually reviewed to check for false negatives, ensuring accuracy. Positive results are not manually verified, as the successful execution of JavaScript confirms the website's exploitability. Notably, this procedure is harmless as it has no side effects on the server and does not involve real users.

The procedure is as follows:

Tranco Range	Crawl Depth	URLs		TE URLs	#domains	#TE domains		Vuln.	Engine	
		Total	avg.			#	%		Angular	Vue
0-1M	1 (max 5 urls)	2,405,504	5.4	406,671	444,949	72,850	16.37	439	299	140
0-100k	1	4,385,138	89.66	818,648	48,908	21,060	43.06	84	69	15
0-100k amass	1	181,224	5.34	520,620	33,876	13,835	40.84	77	56	21
0-5k	2	534,365	264.27	70,259	2022	1041	51.48	2	2	0
475K-525K	1	1,763,751	66.9	314,470	26,361	7659	29.05	47	35	12
497.5K-502.5K	2	482,340	234.94	43,643	2053	562	27.37	7	5	2
950K-1M	1	968,652	51.4	160,898	18,845	5408	28.69	7	5	2
Total		8,672,394	17.53	1,929,846	494,670	96,973	19.6	532	374	158

TABLE II: Vulnerability Detection Results. The URLs column lists the total number of URLs crawled along with the average number of URLs crawled per domain (shown in the avg. column). The TE domains column represents the total number of domains using a TE, along with the percentage relative to the total number of domains retrieved. In the Total row, the vuln column indicates the total number of unique domains that were found to be vulnerable.

Engine	URLs	URLs eff.	Domains	Domains eff.
underscore	350,203	110,745	9223	6205
lit	127,640	1377	4425	123
angular	69,500	16,987	3150	1545
vue	60,449	15,427	1714	1071
handlebars	49,663	10,959	1320	756
alpine	34,320	24,627	666	608
mustache	30,502	6734	625	390
art-template	13,048	5333	380	256
tmpl	12,971	4163	293	193
ejs	7733	1753	111	109
dot	3372	1110	100	73
template7	32,347	718	83	54
pure	3201	1666	70	98
hogan	2899	12	57	7
dust	859	107	32	16
juicer	687	583	29	24
twig	485	306	25	24
nunjucks	318	128	20	0
loadTemplate	263	219	16	6
regular	230	24	10	2
tempo	223	200	5	3
swig	74	24	4	4
transparency	71	40	3	3
squirrelly	17	1	2	1
jsrender	14	8	2	2
icanhaz	10	7	2	3
pug	2	0	1	0
Total	801,101	203,258	22,366	11,576

TABLE III: Template engines usage in the Tranco top 100k. The URLs and Domains columns rely solely on the object detection heuristic, while the URLs eff. and Domains eff. columns additionally incorporate function calls, page scripts, and tag attributes. This measurement was taken separately from the CSTI detection.

- **Creating a Malicious Form**: We generate a local HTML page containing the target form's code and embed the payload. The form is automatically extracted from the vulnerable page using data provided by our detection tool, `CSTI-Alert`. The form's action is modified to point to the URL of the vulnerable website instead of a relative path. Furthermore, the form values are altered to include the CSTI payload that triggers an XSS in the specific template engine used by the vulnerable website. Finally, malicious JavaScript code is embedded in the page to automatically submit the form upon opening.
- **CSRF Simulation**: Using the `Playwright` framework, we simulate the victim's browser visiting the CSRF form. A malicious JavaScript code automatically submits the form, redirecting the victim to the vulnerable website.
- **Payload Execution Check**: Leveraging `Playwright`'s event handlers to detect whether a specific function is executed, we can test if the payload (a simple `alert()` call) runs. If it does, we mark the website as exploitable, otherwise, we perform a manual analysis to determine why the payload was not executed (e.g., it was sanitized or the attack attempt was blocked).
- **GET Request Handling**: In cases where the vulnerable form uses a GET request and reflects the payload in the URL (e.g., `vulnerable.com?search={{alert(1)}}`), we visit this URL directly without hosting a CSRF form. If the alert is triggered, the site is flagged as exploitable.

This method effectively mimics a real-world CSRF scenario, where a victim could be lured into visiting a malicious website that submits the payload on their behalf. By automating the process while manually validating the negative cases, we ensure a thorough and accurate assessment of exploitability. Notably, we can also detect sanitization or firewall mechanisms by checking whether the response still contains the full payload or if parts of it have been escaped, encoded, or removed.

Table IV presents the results of our exploitability analysis, revealing that 385 out of 532 (72%) vulnerable domains were exploitable. Among them, 62% used Angular as the template engine, while the remaining 38% employed Vue. Notably, both the employed engine and the implemented security mechanisms can impact the exploitability of a website. Vue is generally easier to exploit as its payloads are simpler, whereas Angular requires different payloads depending on the version used. Therefore, if the security mechanism implemented by the website sanitizes or blocks payloads containing characters necessary to achieve XSS, the website becomes non-exploitable.

The table also highlights the presence of security mechanisms:

- 13.9% of vulnerable websites implemented input sanitization techniques, categorized as follows:
  - 79% performed HTML encoding on quotes and double quotes (e.g., converting " to ").
  - 14% escaped quotes with a backslash (e.g., \').

  - 3% stripped out certain characters, such as commas and quotes. While quotes or double quotes are not mandatory for exploiting CSTI with certain engines, they are often required when a sandbox escape must be performed and an argument needs to be passed to a function. The following payload is one of the most common examples: `constructor.constructor('alert(1)')()`.
  - The remaining 4% used Unicode escaping (e.g., \u0027).
- Additionally, 8.4% of websites had some form of firewall protection. Of these:
  - 93% returned a 403 status code when the payload was submitted, effectively blocking the request.
  - The remaining 7% displayed a firewall-specific page, indicating the request was flagged as malicious.
- 3.7% of the websites returned a non-200 status code, suggesting either a firewall rejection or a server-side error triggered by the payload. Among these:
  - 400 and 500 status codes were the most common (12 and 7 occurrences, respectively).
  - Only one website responded with a 404 status code.
- Other cases included:
  - 5 websites where the payload failed due to syntax errors in the Angular parser.
  - 3 websites where the payload executed only when it produced search results (the detection payload worked, but the XSS payload did not).

Finally, during the manual validation phase, we discovered 40 websites that were initially flagged as non-exploitable but were actually vulnerable after making minor adjustments to the payload. Specifically:

- 16 websites were filtering for the presence of certain keywords such as `alert` or `prototype` in the payload. By substituting `alert` with other functions (such as `console.log`) and using hex notation to access attributes as strings (e.g., `''.constructor['\x70rototype']`), we successfully triggered the XSS.
- 13 websites restricted the use of single quotes but allowed double quotes, enabling the payload to execute.
- 11 website blocked the use of dots. We bypassed this restriction by modifying the part of the payload containing the dot from: `constructor.constructor` to: `constructor['constructor']`.

D. Angular versions in the wild

Table V presents the distribution of Angular versions across the Tranco top 100k websites. Our analysis reveals the following insights:

- **1.5.X Versions**: These are the most commonly used, accounting for 27% of the websites. Notably, 2% of these are vulnerable despite the fact that 1.5.X is a sandboxed version series known for having the most restrictive sandboxes and the longest payloads required to bypass them (e.g., 326 characters for versions 1.5.9 to 1.5.11 [67]). This makes exploitation more challenging but not impossible.
- **1.8.X Versions**: The second most popular, used by 24% of the websites. Unlike 1.5.X, version 1.8.X is not sandboxed, making it more susceptible to CSTI exploitation. Consequently, 7% of the websites using 1.8.X were found to be vulnerable.

Overall, the data highlights that while newer versions (like 1.8.X) offer more functionality, they also present a larger attack surface due to the absence of a sandbox, increasing the risk of CSTI exploitation.

Exploitability Results	#Domains	Engine	
		Angular	Vue
Exploitable	385	242	143
Sanitization	74	72	2
WAF	45	39	6
400/404/500 Status Code	20	16	4
Other	8	5	3
Total	532	374	158

TABLE IV: Exploitability analysis results

	Angular Version	#Domains		Vulnerable	
Sandboxed	1.0.X	12	1101	2	45
	1.1.X	7		0	
	1.2.X	158		8	
	1.3.X	141		14	
	1.4.X	216		5	
	1.5.X	567		16	
Not sandboxed	1.6.X	226	967	10	55
	1.7.X	115		7	
	1.8.X	514		37	
	1.9.X	112		1	
Total		2068		100	

TABLE V: Most used versions of angular in the Tranco top 100k

E. Defending against CSTI

Popular libraries that sanitize user input against XSS, both server-side and client-side, are generally ineffective against CSTI. This is mainly because the most common template engine syntax (specifically, curly brackets) is not recognized as malicious by these sanitizers. To mitigate certain cases of CSTI using a sanitization approach, developers would need to include the specific syntax of their template engine in the application's input filters.

For example, if Angular is used on the website, the backend should filter curly brackets from user inputs. The only sanitizer capable of handling specific template engine syntaxes ({{ }} and <% %>) is DOMPurify [6], which offers a `SAFE_FOR_TEMPLATE` option that strips template expressions. Notably, this option is disabled by default, and the sanitization of template expressions is not a priority for DOMPurify [76]. This gap highlights the need for CSTI-specific tools that can

perform sanitization based on a wide range of template engine syntaxes. However, a sanitization approach does not guarantee complete protection against CSTI. The most effective way to prevent CSTI is to ensure that user-controlled input never reaches the templates.

Since curly brackets are commonly used in template syntax, filtering this character from inputs where it is unnecessary can be beneficial. However, the internet message format RFC [77] specifies that curly brackets can appear in the local part of an email address, making such filtering more complex in this kind of input. Additionally, there are other scenarios, such as text areas, where curly brackets might be legitimately allowed in the input.

Nevertheless, our results show that most CSTI cases are linked to improper use of Angular and Vue, which are often mounted on the `html` or `body` tags. This practice significantly broadens the scope where user input can be interpreted as part of a template. To prevent such mistakes, templates should be properly separated from the main body of the page and should not include tags where a user-input is reflected.

F. Case study

By analyzing the vulnerable instances we found in our experiments, we identified interesting cases in which CSTI arose in more subtle ways. One notable example, observed on two websites, involved CSTI triggered by the reflection of an input stored within the user session. When a user performed a search, the website saved the search query by associating it with the user's session ID. Later, when the user visited another page containing the search bar, the last search query was reflected inside the bar, triggering CSTI. Our tool correctly identified this vulnerability because it performs multiple interactions with the website. However, the exploitability tool did not detect an immediate reflection, leading to the website being mistakenly marked as not exploitable. Upon manual validation, we discovered that if a victim visited a malicious website containing a CSRF form that injected a CSTI payload, the vulnerability would be triggered when the victim navigated to another page on the affected website. This resulted in a persistent XSS that activated whenever the victim visited a page containing the compromised search bar. This case illustrates a more subtle form of input reflection, which could easily be overlooked by developers or automated tools that only check for immediate reflections on the result page.

VII. Ethical considerations.

Our experiments on live sites did not target any real users. We sought to avoid tests that required data persistence (e.g., storing a payload). Tests on public functionalities were conducted as much as possible without persistently injecting any payload. The vulnerabilities and security risks identified in this paper affect 532 domains. We began the process of notifying the affected parties in February 2025, following best disclosure practices [78], [79]. Initial notifications were sent via email or through bug bounty program platforms, including vulnerability details or a proof-of-concept exploit. Additional reminders were sent every three weeks to maximize the remediation rate. At the time of preparing the camera-ready version, we attempted to notify all affected parties at least once. Of these, two have fixed the issue, one of which awarded us a bounty of 800 dollars, and one acknowledged the vulnerability but subsequently stopped providing updates. A total of 194 notification emails bounced with errors, while the remaining 335 recipients did not respond.

VIII. Summary and Discussion

CSTI is present and impactful. We found 532 domains that are vulnerable to CSTI in our analysis, with 72% of them leading to an exploitable XSS, exposing their user's data to theft and manipulation.

Defenses are absent or inadequate. Despite the high number of vulnerable websites, only 17.7% have firewalls, and only 13.9% employ sanitization techniques, although these are not specifically designed to prevent CSTI. The absence of CSTI-specific sanitizers and the limitations of existing sanitization measures leave websites highly susceptible to the exploitation of this vulnerability. Moreover, we identified 10 instances of CSTI where it was possible to bypass firewalls or sanitizers to achieve XSS.

False positives. We manually validate at least one vulnerable URL per domain. We found only two false positives on vulnerable websites that did not use Angular or Vue. Upon inspection, we discovered that these were actual vulnerabilities but due to SSTI rather than CSTI. This situation can occur when a website uses two different template engines: one on the client-side and another on the server-side that share the same syntax. For example, the client-side engine `lit` uses the same syntax as the server-side Java engine `Spring Expression Language`.

Most common occurrences. Upon manual inspection, we discovered that the most common instance of CSTI was triggered by a search bar that reflected user input inside a tag interpreted as part of a template. In most of these cases Angular (using the `ng-app` attribute) or Vue were mounted on either the main HTML tag or the body tag. This practice caused user input to be treated as part of the template and executed, making it the most common mistake leading to CSTI. Furthermore, because search bars are often reused across multiple pages, the vulnerability was present on most pages of the affected websites, significantly increasing the potential for exploitation.

Limitations. Our large-scale detection study has three main limitations. First, since in our experiment we limit site crawling to a depth of 1 or 2, we may exclude pages that are potentially vulnerable from our analysis. Second, we do not perform authentication. Vulnerabilities may exist on pages or within functionalities that require authentication, which limits the number of vulnerable websites we can identify. Finally, while our approach rarely produces false positives, our measurement focuses on the detection of reflected CSTI instances, and thus it can miss vulnerabilities that occur in different contexts.

IX. Related Work

CSTI can be associated inside the broader category of scriptless injection attacks [80], many research endeavors were made on this category of attacks, among them we find popular vulnerabilities such as DOM Clobbering [81], Dangling Markup [82], and certain classes of Cross-Site Leaks (XS-Leaks) [83]–[85] such as CSS injection.

Some papers related to scriptless attacks that also contain CSTI-related exploits can be found in the past under the category of script gadget attacks, a class of vulnerabilites that exploited frameworks and libraries to achieve arbitrary JavaScript code execution. Lekies et. al. [86] have explored this issue, and among their findings, there are some framework-related gadgets that can be exploited through what we now refer to as CSTI. It was also shown by Roth et. al. [87] that this class of attacks can bypass CSP protections [88] and HTML sanitizers.

The name CSTI only comes from recent work by Heyes [24], [25], which mainly explores CSTI in Angular, Vue and Mavo. However, the presence of works in literature that explore CSTI in a more specific and in-depth way is conspicuously absent. Despite this absence of research work, this vulnerability has attained renewed attention in the bug bounty and security practicioners community [27]–[33].

On the CSTI detection side, a tool called `ACSTIS` (Angular CSTI Scanner) [26] was released in 2017. The last commit in the official repository of the tool dates back to 2019. We tested this tool and identified a set of issues that make its usage challenging. We also observed key differences between the features of our tool and `ACSTIS`.

The first obstacle in running `ACSTIS` is that it requires a Python 3.8 environment, as one of its dependencies, namely `nyawc` [89], does not function properly with newer Python versions. Moreover, it uses Puppeteer [90] with the PhantomJS [91] browser, which is now deprecated [92] and no longer usable. To overcome these issues, we patched the tool to use a Chromium driver instead and created a Docker environment with Python 3.8. While these problems are not overly difficult to resolve, they may discourage users, who must debug and solve these issues before the tool can function properly.

Next, we tested the tool against simple test code suggested in the repository [93] and confirmed that it correctly detects CSTI. Notably, `ACSTIS` supports CSTI detection exclusively for Angular, while our tool extends coverage to 30 different template engines. Additionally, `ACSTIS` focuses primarily on exploiting CSTI by directly attempting payloads designed to trigger the alert function. In contrast, our tool adopts a staged approach, first detecting CSTI with simpler payloads such as $\{\{7*7\}\}$. As discussed in Section VI-C, because XSS payloads are often sanitized or blocked by websites, our approach helps ensure CSTI vulnerabilities are not overlooked even when WAFs or sanitizers are in place.

Overall, our paper consolidates the existing pieces of information and fills the gaps that are present in CSTI research, providing a systematic and comprehensive overview of this vulnerability.

X. Conclusion

In this paper, we conducted what is, to the best of our knowledge, the first comprehensive study of Client-Side Template Injection (CSTI), exploring how template engines operate, how CSTI emerges, its prevalence, and potential defenses. We began by surveying existing template engines, highlighting their characteristics and identifying features that can contribute to XSS escalation due to CSTI.

Next, we introduced our detection methodology and presented `CSTI-Alert`, the first CSTI detection tool supporting a wide range of template engines. We applied `CSTI-Alert` to the Tranco top 1 million sites, revealing the widespread presence of CSTI vulnerabilities. To support future research efforts, we publicly release `CSTI-Alert` [34].

Our findings indicate that existing countermeasures are inadequate for mitigating a substantial portion of these vulnerabilities. Consequently, there is a need for tailored sanitizers and higher awareness of the dangers related to CSTI. We hope that our work will help future efforts to build stronger security measures against this vulnerability.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] 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 & communications security*, 2013, pp. 1193–1204.

[2] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense." in *NDSS*, vol. 20, 2009.

[3] J. Grossman, *XSS attacks: cross site scripting exploits and defense*. Syngress, 2007.

[4] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild." 2019.

[5] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mxss attacks: Attacking well-secured web-applications by using innerhtml mutations," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 777–788.

[6] M. Heiderich, C. Späth, and J. Schwenk, "Dompurify: Client-side protection against xss and markup injection," in *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 2017, pp. 116–134.

[7] 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*, 2011, pp. 587–600.

[8] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 601–614.

[9] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 91–100.

[10] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *2009 ICSE Workshop on Software Engineering for Secure Systems*. IEEE, 2009, pp. 33–39.

[11] JQuery. Jquery. [Online]. Available: https://github.com/jquery/jquery

[12] angular. angular. [Online]. Available: https://github.com/angular/angular/
[13] vuejs. vue. [Online]. Available: https://github.com/vuejs/vue/
[14] T. J. Parr, "Enforcing strict model-view separation in template engines," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 224–233.
[15] M. A. Jadhav, B. R. Sawant, and A. Deshmukh, "Single page application using angularjs," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, pp. 2876–2879, 2015.
[16] J. Kettle, "Server-side template injection: Rce for the modern webapp," *Black Hat USA*, 2015.
[17] Y. Zhao, Y. Zhang, and M. Yang, "Remote code execution from {SSTI} in the sandbox: Automatically detecting and exploiting template escape bugs," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3691–3708.
[18] L. Pisu, D. Maiorca, and G. Giacinto, "A survey of the overlooked dangers of template engines," *arXiv preprint arXiv:2405.01118*, 2024.
[19] battle_angel, "Server side template injection on name parameter during sign up process (glovo)," https://hackerone.com/reports/1104349.
[20] zombiehelp54, "Server side template injection in return magic email templates? (shopify)," https://hackerone.com/reports/423541.
[21] O. Tsai, "uber.com may rce by flask jinja2 template injection," https://hackerone.com/reports/125980.
[22] yaworsk, "Server side template injection via smarty template allows for rce (unikrn)," https://hackerone.com/reports/164224.
[23] Mario Heiderich. A wiki dedicated to javascript mvc security pitfalls. [Online]. Available: https://github.com/cure53/mustache-security
[24] Gareth Heyes. Xss without html: Client-side template injection with angularjs. [Online]. Available: https://portswigger.net/research/xss-without-html-client-side-template-injection-with-angularjs
[25] ——. Abusing javascript frameworks to bypass xss mitigations. [Online]. Available: https://portswigger.net/research/abusing-javascript-frameworks-to-bypass-xss-mitigations
[26] Tijme Gommers. Abusing javascript frameworks to bypass xss mitigations. [Online]. Available: https://github.com/tijme/angularjs-csti-scanner
[27] europa, "Csti in rockstar games," https://hackerone.com/reports/271960.
[28] themarkib0x0, "Stored csti in mars.com," https://hackerone.com/reports/2234564.
[29] "CVE-2024-46366." Available from MITRE, CVE-ID CVE-2024-46366., 2024. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
[30] "CVE-2024-37846." Available from MITRE, CVE-ID CVE-2024-37846., 2024. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-37846
[31] "CVE-2023-26060." Available from MITRE, CVE-ID CVE-2023-26060., 2023. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26060
[32] "CVE-2022-27665." Available from MITRE, CVE-ID CVE-2022-27665., 2022. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-27665
[33] "CVE-2022-22112." Available from MITRE, CVE-ID CVE-2022-22112., 2022. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22112
[34] Lorenzo Pisu, Davide Balzarotti, Davide Maiorca, Giorgio Giacinto. Csti-alert. [Online]. Available: https://github.com/lpisu98/CSTI-Alert
[35] React. React. [Online]. Available: https://reactjs.org/
[36] Svelte. Svelte docs. [Online]. Available: https://svelte.dev/docs
[37] Marko. Markojs. [Online]. Available: https://github.com/marko-js/marko
[38] EmberJS. Ember. [Online]. Available: https://emberjs.com/
[39] Mozilla. Javascript functions. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions
[40] alpinejs. alpine. [Online]. Available: https://github.com/alpinejs/alpine/
[41] jashkenas. underscore. [Online]. Available: https://github.com/jashkenas/underscore/
[42] pugjs. pug. [Online]. Available: https://github.com/pugjs/pug/
[43] lit. lit. [Online]. Available: https://github.com/lit/lit/
[44] handlebars-lang. handlebars.js. [Online]. Available: https://github.com/handlebars-lang/handlebars.js/
[45] janl. mustache.js. [Online]. Available: https://github.com/janl/mustache.js/
[46] goofychris. art-template. [Online]. Available: https://github.com/goofychris/art-template/
[47] mozilla. nunjucks. [Online]. Available: https://github.com/mozilla/nunjucks/
[48] mde. ejs. [Online]. Available: https://github.com/mde/ejs/
[49] swig. swig. [Online]. Available: https://github.com/swig/swig/
[50] twitter. hogan.js. [Online]. Available: https://github.com/twitter/hogan.js/
[51] olado. dot. [Online]. Available: https://github.com/olado/doT/
[52] BorisMoore. jquery-tmpl. [Online]. Available: https://github.com/BorisMoore/jquery-tmpl/
[53] linkedin. dustjs. [Online]. Available: https://github.com/linkedin/dustjs/
[54] mavoweb. mavo. [Online]. Available: https://github.com/mavoweb/mavo/
[55] BorisMoore. jsrender. [Online]. Available: https://github.com/BorisMoore/jsrender/
[56] twigjs. twig.js. [Online]. Available: https://github.com/twigjs/twig.js/
[57] regularjs. regular. [Online]. Available: https://github.com/regularjs/regular/
[58] leonidas. transparency. [Online]. Available: https://github.com/leonidas/transparency/
[59] pure. pure. [Online]. Available: https://github.com/pure/pure/
[60] PaulGuo. Juicer. [Online]. Available: https://github.com/PaulGuo/Juicer/
[61] HenrikJoreteg. Icanhaz.js. [Online]. Available: https://github.com/HenrikJoreteg/ICanHaz.js/
[62] twigkit. tempo. [Online]. Available: https://github.com/twigkit/tempo/
[63] nolimits4web. template7. [Online]. Available: https://github.com/nolimits4web/template7/
[64] squirrellyjs. squirrelly. [Online]. Available: https://github.com/squirrellyjs/squirrelly/
[65] codepb. jquery-template. [Online]. Available: https://github.com/codepb/jquery-template/
[66] adammark. Markup.js. [Online]. Available: https://github.com/adammark/Markup.js/
[67] Gareth Heyes, Mario Heiderich. Angularjs sandbox escapes reflected. [Online]. Available: https://portswigger.net/web-security/cross-site-scripting/cheat-sheet#angularjs-sandbox-escapes-reflected
[68] Mahmoud Gamal. Prototype pollution in handlebars. [Online]. Available: https://www.npmjs.com/advisories/755
[69] ——. Handlebars template injection and rce in a shopify app. [Online]. Available: https://mahmoudsec.blogspot.com/2019/04/handlebars-template-injection-and-rce.html
[70] A. Alhamdan and C.-A. Staicu, "{SandDriller}: A {Fully-Automated} approach for testing {Language-Based}{JavaScript} sandboxes," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3457–3474.
[71] Angular. Developer guide - sandbox removal. [Online]. Available: https://docs.angularjs.org/guide/security
[72] R. Buyya, M. Pathan, and A. Vakali, *Content delivery networks*. Springer Science & Business Media, 2008, vol. 9.
[73] Microsoft. Playwright. [Online]. Available: https://github.com/microsoft/playwright
[74] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *arXiv preprint arXiv:1806.01156*, 2018.
[75] OWASP. Amass. [Online]. Available: https://owasp.org/www-project-amass/
[76] cure53. Xss via template expressions is not a key goal for dompurify. [Online]. Available: https://github.com/cure53/DOMPurify/issues/698
[77] B. Leiba, "Update to Internet Message Format to Allow Group Syntax in the "From:" and "Sender:" Header Fields," RFC 6854, Mar. 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc6854
[78] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of {Large-Scale} web vulnerability notification," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1015–1032.
[79] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability notifications," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1033–1050.
[80] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 760–771.
[81] S. Khodayari and G. Pellegrino, "It's (dom) clobbering time: Attack techniques, prevalence, and defenses," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1041–1058.

[82] F. Hantke and B. Stock, "Html violations and where to find them: a longitudinal analysis of specification violations in html," in *Proceedings of the 22nd ACM Internet Measurement Conference*, 2022, pp. 358–373.
[83] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks," *arXiv preprint arXiv:1908.02204*, 2019.
[84] T. Van Goethem, G. Franken, I. Sanchez-Rola, D. Dworken, and W. Joosen, "Sok: Exploring current and future research directions on xs-leaks through an extended formal model," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 784–798.
[85] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, "Xsinator. com: From a formal model to the automatic evaluation of cross-site leaks in web browsers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1771–1788.
[86] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1709–1723.
[87] S. Roth, M. Backes, and B. Stock, "Assessing the impact of script gadgets on csp at scale," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 420–431.
[88] M. West, "Initial Assignment for the Content Security Policy Directives Registry," RFC 7762, Jan. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7762
[89] Tijme Gommers. Nyawc. [Online]. Available: https://pypi.org/project/nyawc/
[90] Puppeteer. Puppeteer. [Online]. Available: https://github.com/puppeteer/puppeteer
[91] ariya. Phantomjs. [Online]. Available: https://github.com/ariya/phantomjs
[92] ——. Phantomjs deprecation issue. [Online]. Available: https://github.com/ariya/phantomjs/issues/15344
[93] dabula-s. On what vulnerable website can i test your scanner? [Online]. Available: https://github.com/tijme/angularjs-csti-scanner/issues/14