RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Improving the Detection and Identification of Template Engines for Large-Scale Template Injection Scanning

Maximilian Hildebrand

Master's Thesis – September 19, 2023
Chair for Network and Data Security.

Supervisor: Prof. Dr. Jörg Schwenk
Advisor: M. Sc. Lukas Knittel
Advisor: Prof. Dr.-Ing. Juraj Somorovsky

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

In this thesis, the currently most relevant template engines are enumerated and intentionally implemented in a manner vulnerable to template injection in a playground with various features. Based on this, new error polyglots are created that surpass the known ones in brevity and template engine coverage. Furthermore, a new type of template injection polyglots is constructed, called non-error polyglots. These non-error polyglots ensure that no error is thrown that might be caught, but that a template engine renders them modified. To make these polyglots easy to use for template injection detection and template engine identification, a web page with an interactive table is created. Furthermore, a new type of template injection scanner (*TInjA*) is created, which significantly outperforms the previous scanners in several categories, such as number of requests sent and identified template injections. This outperformance is mainly due to the newly created polyglots. Finally, 81,937 URLs from 69 apex domains of the Tranco Top 1000 list are scanned for template injections with TInjA. During the analysis of the scan results, a new "hybrid approach" is developed. This approach combines the speed and effectiveness of novel polyglots with the very low false positive probability of a template expression tailored to a specific template engine.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

19.09.23	Hildebrand
DATE	MAXIMILIAN HILDEBRAND

Erklärung

Ich erkläre mich damit einverstanden, dass meine Abschlussarbeit am Lehrstuhl Netz- und Datensicherheit dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen.
Weiterhin erkläre ich mich damit einverstanden, dass die Abschlussarbeit auf die Webseite des Lehrstuhls veröffentlicht werden darf.

19.09.23	Hildebrand
DATE	MAXIMILIAN HILDEBRAND

Contents

1 Introduction

This chapter serves as an introduction to the master's thesis. First, we delve into the motivating factors behind the creation of this thesis. Subsequently, we provide a concise review of the relevant prior research, setting the stage for our unique contributions. Lastly, we present an outline of the thesis structure.

1.1 Motivation

In an ever-evolving landscape of cyber threats, the discovery of novel attack vectors and exploitation techniques remains a persistent challenge. The need to comprehensively test for these vulnerabilities is paramount in order to safeguard the continuously expanding attack surface. Among these threats, template injections stand out as a significantly underestimated vulnerability within web applications. Despite their potential to compromise security, their significance has been overlooked. Over recent years, a consistent pattern has emerged, with approximately 30 template injection Common Vulnerabilities and Exposures (CVEs) being disclosed annually. Remarkably, a majority of these vulnerabilities grant attackers the ability to achieve Remote Code Execution (RCE), often without the need of authentication. The exploitation of template injection vulnerabilities necessitates a multi-step process. Initially, it is imperative to detect instances where user input is processed by a template engine. Subsequently, the specific template engine in use must be identified. At this juncture, the application of polyglots proves instrumental in facilitating efficient testing. However, existing polyglots exhibit inherent limitations. Some are concise but confined to a narrow spectrum of commonly used template engines. Others offer broader coverage but at the expense of length and complexity. This master's thesis addresses the shortcomings of established polyglots by introducing novel polyglots that overcome the limitations of the existing ones. In addition, the thesis introduces *TInjA*—a novel template injection scanner that outperforms its predecessors in terms of efficiency. TInjA harnesses the power of the newly developed polyglots to conduct scans that are considerably faster and more effective.

1.2 Related Work

Template injection vulnerabilities were investigated for the first time in 2013. The *mustache-security* project analyzed several JavaScript frameworks and template engines regarding their security and defined security criteria [3].

In 2015, it was shown that template injections can be executed not only client-side in the browser, but also server-side, thus achieving RCE [6]. RCE payloads have been developed for several template engines, even working in the sandbox mode of the template engine, which is a restricted mode for security reasons. The three-step template injection methodology was also presented.

In 2018, new template injection polyglots were created, which have a higher coverage of template engines than the previous polyglots [8]. These polyglots were then used to create a scanner that is more efficient at detecting template injection possibilities than the previous scanners.

In 2021, a short research paper on an SSTI countermeasure using instruction set randomization was published [13]. Using the PHP template engine *Twig* as an example, it promises to be able to prevent SSTI without performance loss.

In 2022, an evaluation of how well different web application security scanners are able to find different injection vulnerabilities was published [1]. In the case of SSTI, none of the scanners evaluated were able to detect the vulnerability.

In 2023, a paper was published on how to automatically bypass a sandbox to achieve RCE in the case of SSTI [14]. The tool created for this purpose was also published[1].

In addition, new sandbox escapes have been published over the years [5][4][9].

1.3 Contribution

This thesis presents a comprehensive set of six distinct contributions, each addressing important aspects within the domain of template injection vulnerabilities:

Template Injection Playground A novel template injection playground is developed and published. Unlike existing playgrounds this playground exhibits notable advancements. It strategically focuses on integrating only the most relevant template engines after careful analysis. In addition, the playground offers a wide range of implemented template engines. A special feature is the configurable behavior, which allows users to activate various countermeasures, for example.

[1]`https://github.com/seclab-fudan/TEFuzz/`

Novel Improved Polyglots Building on the foundation laid by the polyglots published in 2018 [8], this research identifies opportunities for refinement. The result culminates in the creation of polyglots that are not only shorter, but also offer broader coverage. An innovative type of template injection is introduced that is characterized by a departure from error-inducing behavior. These "non-error" polyglots bring two benefits: immunity to scenarios where errors are common, and the ability to identify template engines in a significantly more efficient manner.

Template Injection Table An interactive *Template Injection Table* is created and published. Enriched with the novel and efficient polyglots and expected template engine responses, this resource accelerates the detection of template injection possibilities and the identification of template engines used.

TInjA Template Injection Scanner TInjA, an innovative template injection scanner, is developed and published. Characterized by the use of novel and efficient polyglots, TInjA surpasses its predecessors in several dimensions. It excels in detecting a greater number of template injection possibilities, exhibits improved accuracy in identifying template engines, and significantly reduces the number of requests and time required to perform these tasks.

Scanner Performance Evaluation A comparative evaluation is conducted that includes TInjA and various other template injection scanners. This evaluation unfolds across two different template injection playgrounds and an SSTI vulnerability from the scanner evaluation published in 2022 [1]. The results will provide insight into the current state of scanner effectiveness.

Large-Scale Template Injection Scanning This will be followed by a scan of some of the world's most popular domains using TInjA. This initiative aims to identify the challenges of performing large-scale template injection scans while proposing viable solutions.

1.4 Organization of this Thesis

Chapter 2 provides the necessary foundations needed throughout this thesis.

In Chapter 3 the *Template Injection Playground* is introduced, which can be used to test a variety of template engines and is important for the following chapters. The selection of template engines is discussed in detail. This is crucial for the creation of new and improved polyglots.

Chapter 4 discusses the creation of the novel polyglots, which are shorter and cover a wider range of template engines. In particular, a new type of template injection polyglots is introduced. These "non-error" polyglots have significant advantages over the "error" polyglots commonly used so far. Further, the *Template Injection Table* is introduced, which accelerates the execution of the template injection methodology.

Chapter 5 introduces the template injection scanner *TInjA*. TInjA uses the novel polyglots to quickly and efficiently detect template injection possibilities and even identify the template engines used. The use of polyglots to identify template engines is a novel approach for scanners.

In Chapter 6 a comparative evaluation is conducted that includes TInjA and various other template injection scanners. This evaluation unfolds across two different template injection playgrounds and an SSTI vulnerability from the scanner evaluation published in 2022 [1]. In addition to the number of detected template injection possibilities and identified template engines, other metrics such as the duration and number of requests are also measured.

In Chapter 7, a large-scale template injection scan is performed. For this purpose, TInjA is used to scan some of the world's most popular domains. Problems encountered during the scan as well as causes for false positives are identified and solutions are presented.

Chapter 8 discusses the key insights and main contributions of this thesis.

2 Background

This chapter provides the necessary background needed throughout this thesis in four sections. The first section describes what template engines are and how they can be used. The second section explains the template injection vulnerability class that can be exploited by an attacker if a template engine is used insecurely. The third section explains the template injection methodology that can be used to find and exploit template injection vulnerabilities. The fourth section discusses several countermeasures that can reduce the risk of a template injection or, in some circumstances, prevent it altogether.

2.1 Template Engine

Template engines allow static template files or strings containing template language to be used in web applications. The function of a template engine is to dynamically interpret the template language at runtime and generate an output format, such as HTML, from the template. Template engines often have features similar to a high-level programming language, such as variables, functions, loops, conditions, and access to files. The processing flow of a template engine is shown in Figure 2.1. The template engine interprets a template containing template statements and combines it with a data model, such as a relational database or other data source. The result is a document or part of a document.

This simplifies the programming of web applications because data can be dynamically read from databases and the design principle of separation of concerns can be achieved. This allows the use of the MVC (Model-View-Controller) pattern or a similar approach where the appearance of the web application is independent of the data to be used. The MVC approach allows designers to create templates with placeholders and developers to create the logic that replaces the placeholders. This makes the work of both designers and programmers easier and more efficient. Many web frameworks that simplify the programming of dynamic web applications use a third-party or even custom template engine by default. In addition, template engines can often be easily integrated into an existing project. As a result, template engines are widely used in web applications.

The functionality and syntax of different template engines can be very similar, but also very diverse. Some want to be all-inclusive solutions and support loops, conditions, expressions, filters, and much more (e. g., *AngularJS*). Others have a logic-less

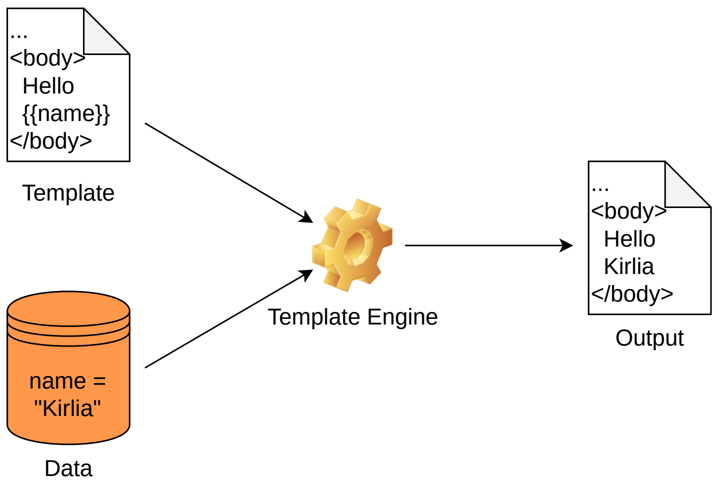


Figure 2.1: The processing flow of a template engine.

approach and do not want to integrate any logic and control flow into the templates (e. g., *Mustache*). Also, depending on the template engine, the tags that enclose a template statement vary. Commonly used are `{{ statement }}`, `${ statement }`, and `<%= statement %>`. Integrating a JavaScript template engine into a web application can be done in two different ways. Either client-side by integrating it into the web page with a "script" tag, or server-side by using a back-end JavaScript runtime environment, such as *Node.js*. Template engines for programming languages other than JavaScript are always server-side, since they are installed and run on a server and cannot be included with a script tag in the web page.

2.2 Template Injection

Template injection vulnerabilities occur when user input is embedded into a template and, thus, can manipulate or add template statements. A distinction is made between client-side template injection (CSTI) and server-side template injection (SSTI), depending on whether a client-side or server-side template engine is used. When exploiting an SSTI, an attacker may be able to, among other things, execute code on the server (remote code execution, or RCE for short), read or write arbitrary files, or generate HTML containing JavaScript that is then executed in a victim's browser (cross-site scripting, or XSS for short).

When exploiting a CSTI, the template is executed by a JavaScript template engine in the browser. This means RCE or reading or writing arbitrary files on the server is not possible. However, the template engine can be instructed to generate HTML with malicious JavaScript (XSS) or to execute JavaScript directly, for example. This means that the only difference between CSTI and SSTI is the potential

```
{ system('whoami') }
```

Listing 1: Smarty template expression to run the `whoami` command on the server

```
<%= system("whoami") %>
```

Listing 2: ERB template expression to run the `whoami` command on the server

impact. The process for detecting and exploiting both types of template injection is identical.

Template injection payloads can look very different depending on the template engine. In the following, we will look at four different server-side template engines as examples. Consider the `whoami` command, which prints the current user name when invoked on most Unix and modern Microsoft Windows operating systems. To achieve RCE and run `whoami` using *Smarty*, a PHP template engine, the simple template statement shown in Listing 1 can be used. For the Ruby template engine *ERB* there also exist simple template statements for RCE, as shown in Listing 2. Other template engines, such as *Jinja2* (Python) and *Nunjucks* (Node.js), also allow commands to be executed, but the template statements are much more complex (see Listing 3 and Listing 4). This is due to the fact that they do not provide direct command execution functionality; however, objects can be accessed in an indirect way, which then allows commands to be executed.

2.3 Template Injection Methodology

As mentioned in Section 1.2, Kettle has established a template injection methodology that can be used to detect and exploit template injection vulnerabilities. This methodology consists of three main steps: *Detect*, *Identify*, and *Exploit*. The third step, *Exploit*, can be further subdivided into the three sub-steps *Read*, *Explore*, and *Attack*.

The steps of the methodology are presented in the following.

```
{{ self._TemplateReference__context.cycler.__init__.__globals__.os
↪  .popen('whoami').read()
↪  }}
```

Listing 3: Jinja2 template expression to run the `whoami` command on the server

```
{{ range.constructor("return
↪  global.process.mainModule.require('child_process').execSync('whoami')")() }}
```

Listing 4: Nunjucks template expression to run the `whoami` command on the server

2.3.1 Detect

The first step is to find out if template injection is possible, in other words, if unsafe user input is being inserted into a template. To do this, it is useful to use a template statement that can be interpreted by as many template engines as possible. Many template engines use `{{` as opening and `}}` as closing tag for their statements and a `*` for multiplication. Therefore, the statement `{{7*7}}` is often referenced, which should be rendered as `49` by a template engine that uses this syntax. It is also possible to make a template engine reveal itself by throwing an error. This can be done, for example, by provoking a division error with `{{1/0}}` if the thrown error is returned and not caught.

A template injection can happen in two different contexts, which must be distinguished:

Plaintext context Here, the user input is free-standing in the template and not in any template statement. Such a plaintext context is shown in Listing 5. If the user input is the string “User”, for example, then `Hello User` is printed in bold. If the user input is a template statement like `{{7*7}}` instead and it is interpreted by the Python template engine Jinja2, Jinja2 will generate the result `Hello 49`. The plaintext context often results in XSS because a user input of `<script>alert(1)</script>` would be returned unmodified after the `Hello` and executed by the browser. Accordingly, the XSS vulnerability should be easily found by an automated scanner. The same is true for template injections, as long as the scanner uses correct statement syntax or causes the template engine to throw an error.

Code context Here, the user input is placed inside a template statement. Such a code context is shown in Listing 6. If the user input is `7*7` and interpreted by the Python template engine Jinja2, Jinja2 will generate the result `Hello 49`. However, the user input `{{7*7}}` as in the plaintext context would now throw a syntax error, because Jinja2 does not expect an opening tag (`{{`) after another opening tag (`{{`). If the syntax error is returned, it is easy to see that the user input ends up in a template. Otherwise, if it the error is caught, `{{7*7}}` would not be suitable to identify the template injection. The same is true for XSS. An XSS payload like `<script>alert(1)</script>` would not succeed in this context because Jinja2 would throw a syntax error. However, the template statement can be closed to insert the XSS payload and then start a second template statement to avoid errors. Therefore, for the user input `4}}`

```
template = "<b>Hello " + userinput + "</b>"
```

Listing 5: The user input is inserted free-standing into the template. This is called “plaintext context”

```
template = "<b>Hello {{" + userinput + "}} </b>"
```

Listing 6: The user input is inserted into the template within an expression. This is called “code context”

`<script>alert(1)</script>{{9` the text `Hello 49` is shown and the script executed by the browser.

2.3.2 Identify

Once the possibility of a template injection has been detected, the next step is to identify which template engine is used for processing the template. For example, if an error is caused by a division by zero, the template engine can probably be identified very easily from the error message. However, if errors are caught, this is no longer possible. To solve this issue the following approach can be applied: A statement is used that is only valid for a subset of template engines. Based on the results template engines are excluded from the list of possible template engines. This process is repeated iteratively until only one template engine is left on this list. For example, `{{7*'7'}}` is interpreted as `7777777` by some template engines and as `49` by others. Figure 2.2 shows a decision tree that can be used to distinguish between the four template engines Blade, Twig, Jinja2 and Tornado.

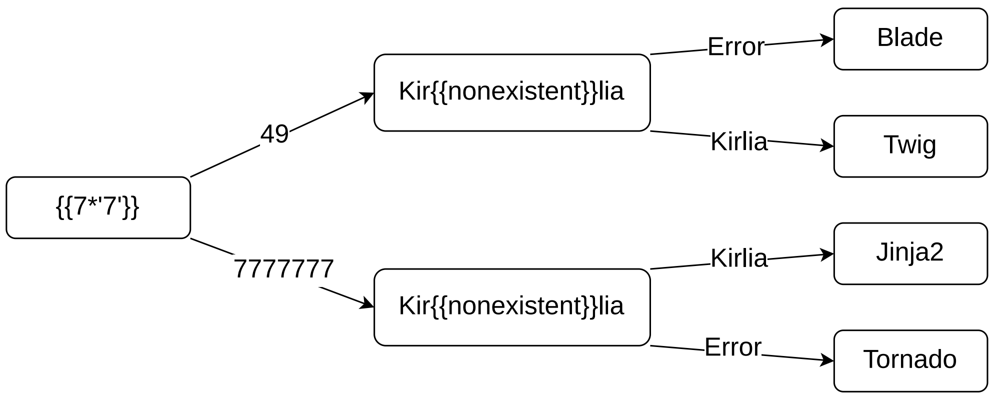


Figure 2.2: Decision tree to distinguish between four template engines.

2.3.3 Exploit

After identifying the template engine in use, the next step is to exploit the template injection vulnerability. Nowadays, there are many cheat sheets that show possible template injection payloads for different template engines[1,2]. To create such payloads or modify existing ones, the following three steps are required.

Read The first step is to read the documentation carefully. Besides the basic syntax are variables, methods, and other functions of interest. Furthermore, it can be very promising to look for security considerations, as these can point out possible dangers and attack vectors.

Explore The second step is to find out what variables, objects, and methods are accessible. Many template engines provide a "self" object that contains all other possible objects. In the case of Jinja2, for example, other possible objects can be listed with the template expression `{{self.__dict__}}` (see Figure 2.3). This is also an easy way to find out about objects that are not accessible by default, but are passed to the template by the developer. If such a self object is not accessible, possible objects can be enumerated using a brute force approach.

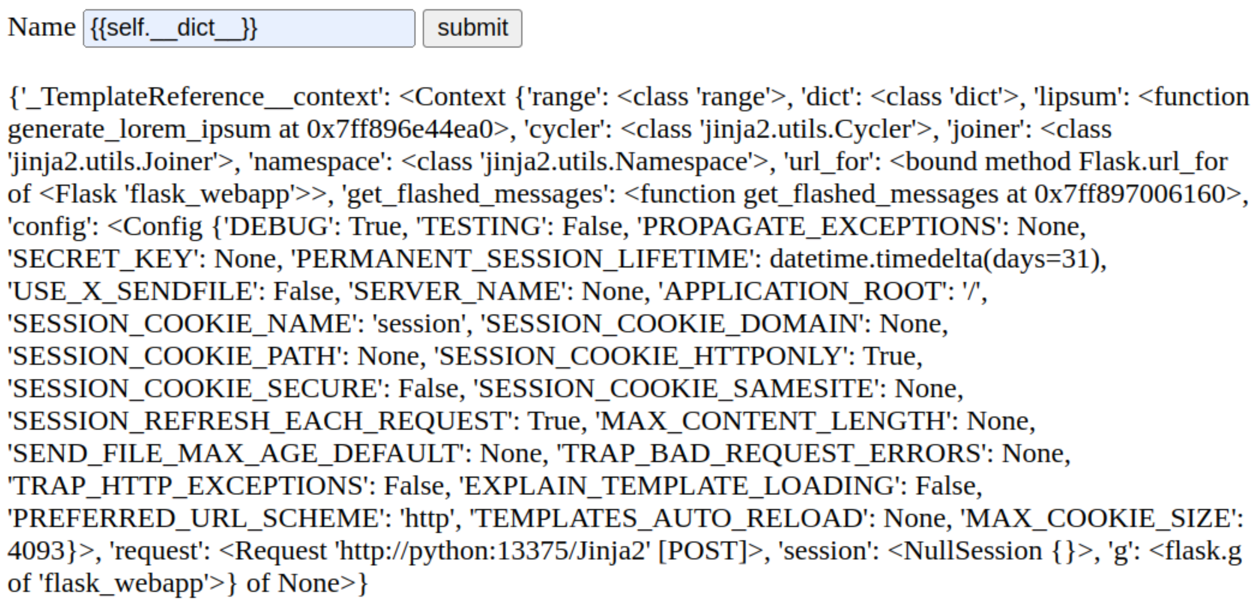
Attack In the third and final step, the knowledge gained from the *Read* and *Explore* steps can be utilized to call objects or methods that can be used to execute malicious actions. For example, Listing 3 shows an SSTI payload for Jinja2 which allows RCE. In the depicted statement various objects are accessed iteratively starting at the self object until the Python `os` module can be accessed. The `os` module provides the `popen` method, which can be used to execute a command in a subprocess.

2.4 Template Injection Countermeasures

There are several ways to prevent template injections, or at least make them more difficult to achieve.

Sandboxing Some template engines offer a sandbox mode that restricts functionality to prevent malicious statements from being executed. However, these sandboxes are not foolproof, as research is repeatedly showing how such sandboxes can be bypassed [6, 4, 5, 14]. Nevertheless, they make template injections more difficult, even if they do not necessarily guarantee that no malicious statements can be executed.

[1]`https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection`
[2]`https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Template%20Injection`



```
{'_TemplateReference__context': <Context {'range': <class 'range'>, 'dict': <class 'dict'>, 'lipsum': <function
generate_lorem_ipsum at 0x7ff896e44ea0>, 'cycler': <class 'jinja2.utils.Cycler'>, 'joiner': <class
'jinja2.utils.Joiner'>, 'namespace': <class 'jinja2.utils.Namespace'>, 'url_for': <bound method Flask.url_for
of <Flask 'flask_webapp'>>, 'get_flashed_messages': <function get_flashed_messages at 0x7ff897006160>,
'config': <Config {'DEBUG': True, 'TESTING': False, 'PROPAGATE_EXCEPTIONS': None,
'SECRET_KEY': None, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
'USE_X_SENDFILE': False, 'SERVER_NAME': None, 'APPLICATION_ROOT': '/',
'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_DOMAIN': None,
'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_HTTPONLY': True,
'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': None,
'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
'SEND_FILE_MAX_AGE_DEFAULT': None, 'TRAP_BAD_REQUEST_ERRORS': None,
'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
'PREFERRED_URL_SCHEME': 'http', 'TEMPLATES_AUTO_RELOAD': None, 'MAX_COOKIE_SIZE':
4093}>, 'request': <Request 'http://python:13375/Jinja2' [POST]>, 'session': <NullSession {}>, 'g': <flask.g
of 'flask_webapp'>} of None>}
```

Figure 2.3: Jinja2 provides a dictionary object that contains all other available objects and functions.

User input validation Similar to other injection vulnerabilities, such as SQL injection (SQLi) and XSS, template injections can be made more difficult or even prevented by carefully validating user input. An allow list is preferable to a deny list. This is because with a deny list, the prohibited expressions and characters can often be replaced with others. Thus, deny lists become very high-maintenance and are not sufficient to prevent template injection attacks. With an allow list, only certain expressions and characters are allowed, which can effectively reduce the attack surface. Regex expressions are well suited for this purpose; however, it should be noted that regex validations can potentially lead to a denial of service if vulnerable regex expressions are used.

No user input in templates If possible, user input should not be inserted directly into templates. Many, but not all, template engines have a feature called "context". Instead of the user input, placeholders can be inserted into the template and the user input can be passed as parameters to the template engine. Similar to a "prepared statement" in SQL[12], the template is then precompiled with the placeholders, and afterwards the placeholders are replaced with the parameters. Therefore, a user is not able to manipulate the template and a template injection is prevented.

Usage of a logic-less template engines Logic-less template engines do not offer logic and control flows. This means that often only placeholders can be replaced by parameters, but no loops, expressions, or filters can be used. As a result, these template engines have only a small attack surface. One of the most widely used logic-less template engines is *Mustache*, which has implementations in many programming languages[11]. However, recent research has shown that at least the PHP implementation of Mustache was vulnerable to RCE [14], which has been fixed in Version 2.14.1[10].

Isolation of the template engine The template engine can be isolated to prevent code execution or access to sensitive data in the event of a successful SSTI. For example, it can run in a hardened Docker container configured to prevent breakouts.

3 Template Injection Playground

This chapter introduces the *Template Injection Playground*, which can be used to test a variety of template engines and is important for the following chapters. First, we describe the use cases for which the playground was designed. Then, we explain the features of the playground that are determined by the requirements of the use cases. After that, we break down the selection of the template engines implemented in the playground. This selection is very important because the results of the following chapter depend on it. Next, we outline the structure of the playground. Finally, we describe how to set up and run the playground.

3.1 Use-Cases and Requirements

The Template Injection Playground was built to create the polyglots for template injection detection and template engine identification. The playground was also used to create and test the *TInjA* template injection scanner which will be presented in Chapter 5. Finally, it has also been used to benchmark TInjA and other template injection scanners. The playground is released as an open source project so that other cybersecurity enthusiasts and professionals can use it for any of the following use cases:

UC1: Programming a template injection scanner. In this use case someone wants to program a scanner that detects template injection vulnerabilities.

UC2: Compare template injection scanners. In this use case someone wants to compare template injection scanners to find out which one is the best. These comparison criteria could include, for example, the number of false positives/negatives, blind SSTI detection, and automated CSRF token extraction.

UC3: Create polyglots. In this use case, someone wants to create polyglots for template engine detection that multiple template engines will react to.

UC4: Creating template expressions. In this use case, someone wants to create template expressions that can be used to distinguish between different template engines.

UC5: Template injection methodology. In this use case, someone wants to manually test the template injection methodology on different template engines.

Table 3.1 summarizes the requirements towards the playground for each use case. These requirements are:

Many TEs Many template engines (short TE) are available because each may behave in a different way.

Information about TEs Information about the template engines is available. This includes links to the template engines's documentation and example template expressions.

Countermeasures Different countermeasures can be activated in order to simulate different countermeasures used in the wild.

Scenarios There are different scenarios that change the behavior of the template engine or the web page. Examples for such scenarios are blind template injections or that the user input is being rendered on another page.

Table 3.1: The requirements of the use cases of the Template Injection Playground.

	UC1	UC2	UC3	UC4	UC5
Many TEs	x	x	x	x	x
Information about TEs	x		x	x	x
Countermeasures	x	x			x
Scenarios	x	x			x

3.2 Features

A total of 46 template engines for 8 different programming languages can be tested for template injection. For each template engine, there is a simple web page that inserts the content of a text box into a template. The template is then processed by the TE and rendered on the web page. Due to this very simple scenario and the insecure insertion of unprocessed user input into a template, it is easy to find and exploit these template injection possibilities.

JavaScript template engines can often be implemented on both the server and client sides. In the Template Injection Playground, the JavaScript TEs were implemented server-side because the effects of SSTI are more dangerous and extensive than those of CSTI, as described in Section 2.2. The only exception is the AngularJS template engine. It is implemented client-side, as it does not offer a server-side implementation. SSTIs are also easier for scanners to detect because they do not require JavaScript to be executed by the scanner itself. In order to help one get started with a template engine, helpful links are provided for each TE, for example, to the official documentation or to blog posts containing information about template injections for the specific TE. A collection of template expressions, syntax examples, and special features is also provided for each TE.

In order to practice the manual procedure of the template injection methodology in a completely unbiased way, a randomly selected template engine can be tested. If desired, the selected TE can be revealed or another TE can be selected randomly.

To make it more difficult to find and exploit template injections, there are global options that can be set and then applied to the web page for all TEs. This can be used to validate whether a scanner is still able to detect template injections under more difficult conditions. These global options are as follows:

1. Strings can be added to a deny list so that they are removed from user input. This can invalidate template expressions that use these strings.
2. HTML encoding of the five HTML special characters `&`, `<`, `>`, `'`, and `"` can be enabled. This can invalidate template expressions that use these characters.
3. Error message interception can be enabled. The common polyglots for detecting template injections are based on a template engine throwing an error. However, if this error is caught and not displayed to the user, it appears that the template injection was not successful. Instead of common polyglots, template expressions that can be processed correctly by the TE and do not throw an error must be used.
4. Four special scenarios can be enabled:
   a) In the case of the "originHeader" scenario, the value of the `Origin` header is inserted into the template in place of the `name` parameter.
   b) The "blind" scenario ensures that the output of the template engine is not returned. This makes detection and identification of the template engine much more difficult. If, in addition, the errors are suppressed and hidden, then the only option is to try out-of-band SSTI payloads for different template engines that might be used on the website. For example, a possible real-world scenario is that the user input, after being processed by a TE, is only visible in an admin panel to which the user has no access.
   c) The "otherLocationDirect" scenario ensures that the output of the TE is not returned directly in the response, but is provided at the relative URL `/otherlocationdirect/`, which is referenced in the response. A possible real-world scenario is when one or more fields in a registration form are processed by a TE, and the output is only visible when the user clicks on their profile link after successful registration.

   d) The “otherLocation” scenario is similar to “otherLocationDirect”, but the TE output is provided at the relative URL `/otherlocation/` and this is not referenced in the response. Instead, this URL is specified in a list of URLs that can be passed to a scanner. This means that a scanner must be able to look for the TE output not only in the direct response or in a URL referenced in the response, but also in other URLs passed to the scanner by, for example, a crawler.

5. Two different versions of Cross-Site Request Forgery (short CSRF) protection can be enabled. In the first case, a random but fixed CSRF token is generated, which must be included in every POST request. If the correct value is not provided, the user’s input will not be processed and an error message will be returned. Such a static CSRF token is not recommended in a production environment, as it does not effectively prevent CSRF attacks if the attacker knows the fixed value. However, for this playground, it is sufficient to simulate fixed but random and session-bound CSRF tokens without the need to implement actual session handling, which is otherwise not required for the playground. In the second case, a new random CSRF token is not generated once, but on every request. This means that the last generated CSRF token must be included in every POST request.
6. A user input length limit can be set. If the user input is longer than the specified length limit, an error message is returned.

Five of the 46 implemented template engines offer a sandbox mode in which objects and features classified as dangerous are prohibited. The Twig, Smarty, Latte, and Jinja2 template engines can also be tested with their sandbox mode enabled. Unfortunately, it was not possible to implement the sandbox mode of the RazorEngine in the playground without causing critical errors. The sandbox modes were implemented with the unchanged default settings. Since Twig and RazorEngine do not have default sandbox settings, but require explicit naming of objects and features to be prohibited, the settings have been taken from the examples in the respective documentation[1,2].

Furthermore, the playground counts the number of requests that are sent to it. This is a simple way to measure how many requests a scanner sends.

3.3 Implemented Template Engines

For the playground, it was important to implement the most relevant and widely used template engines. This is necessary so that the polyglots used to detect template injections and identify template engines can be used for as many websites as

[1]`https://twig.symfony.com/doc/3.x/api.html#sandbox-extension`
[2]`https://antaris.github.io/RazorEngine/Isolation.html`

possible. The procedure to achieve this goal can be summarized as follows:
The most commonly used programming languages for website backends and the most commonly used web frameworks were identified from various sources. Further, the package managers of the respective programming languages were then used to find the most commonly used template engines. We also considered which template engines were used by default by the most popular web frameworks. In the end, 47 template engines were selected as the most relevant according to our criteria; however, one of the template engines—the template engine of the Angular web framework—could not be implemented within a reasonable timeframe. As a result, Angular was not included in the creation of the polyglots. The choice of programming languages and template engines, as well as the shortcomings of the process, are explained in more detail below.

3.3.1 Selection of Programming Languages

Selection Three different sources were used to find out the most popular programming languages for websites.

1. “Usage statistics of server-side programming languages for websites”[3] by w3techs. According to their methodology[4], w3techs analyzes the most relevant websites to enumerate the most used server-side programming languages. For this purpose, the top 1000 websites of the Tranco list[5] are used. The most frequently used server-side programming languages with more than one percent were PHP, ASP.NET, Ruby, Java, Scala, JavaScript, and Python. All of these were selected as candidates.

2. “Web framework and technologies survey 2022”[6] by Stackoverflow. Stackoverflow, one of the most popular platforms for developers to ask questions, conducts surveys every year. The latest 2022 results list 25 of the most used web frameworks and technologies according to the votes. JavaScript clearly leads the pack with Node.js, React.js, and many other frameworks and technologies. ASP.Net, PHP, Python, Ruby, and Elixir (an Erlang-based programming language) are also represented. This means that all the programming languages selected in the previous step are represented, with the exception of Java. Elixir was added to the list of candidates because of the Phoenix web framework, which is ranked 22nd in the survey.

3. “Most Popular Backend Frameworks 2022”[7] by Statistics and Data. Statistics and Data lists web frameworks based on the stars they have accumulated on Github. PHP, Python, JavaScript, Ruby, Java, ASP.NET, and

[3]https://w3techs.com/technologies/overview/programming_language
[4]https://w3techs.com/technologies
[5]https://tranco-list.eu/
[6]https://survey.stackoverflow.co/2022#section-most-popular-technologies-web-frameworks-and-technologies
[7]https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/

Golang are represented. As Golang appeared twice, it was also selected as a candidate.

Shortcomings The w3tech statistics are only partially representative. This is because it is difficult to extrapolate from the top 1000 websites to the rest of the web. In addition, some programming languages may be much more difficult or even impossible to detect, while others may be very easy to detect. An example of this is PHP and Golang. With a file extension like `.php` or a session cookie like `PHPSESSID`, it is easy to conclude that PHP is involved. Go, on the other hand, does not use special file extensions and session cookie names, and is therefore in comparison much more difficult or even impossible to detect.
The Stackoverflow survey is only partially representative, as the results depend heavily on the respondents.
The list by Statistics and Data is only partially representative, as not all web frameworks are published as public repositories on Github.

3.3.2 Selection of Template Engines

PHP, ASP.NET, Ruby, Java, Scala, JavaScript, Python, Elixir, and Golang are the programming languages that were found to be the most relevant server-side programming languages and were examined in more detail. In order to find the most relevant template engines, the package managers of the respective programming languages were used and searched with different terms such as “template engine” or “templating engine”. The results were sorted by the number of downloads, number of dependencies, or other criteria, depending on the package manager’s capabilities. As some template engines may not be found due to the search terms used, an additional search for web frameworks was performed for each programming language to see which template engines they support according to their documentation or even use by default. If a web framework has its own template engine, which cannot be used independently of the web framework, the statistics of the web framework were used to represent the template engine.
Table 3.2 shows the statistics for the selected template engines grouped by programming language as of May 26, 2023. While Python has the most downloads overall, JavaScript leads in the number of monthly downloads, packages that have the template engine as a dependency, CVEs, and number of template engines which were analyzed during this thesis. Unfortunately, not all package managers publish download counts. In addition, Golang and Ruby have their own template engines that ship with the language by default, so only limited statistics are available.

The top-ranked Template Engines according to programming language are shown below, along with more detailed statistics.

Table 3.2: The statistics of the selected template engines grouped by programming language and sorted by total downloads as of May 26, 2023.

	Downloads				
TE	Total	Monthly	Dependants	CVEs	Selected TEs
Python	5.493.037.118	171.278.046	n/a[a]	21	7
JavaScript[b]	3.586.370.264	199.896.868	145.291	62	14
Ruby	822.357.421[c]	n/a[a]	n/a[a]	7	7
PHP	550.034.563	10.304.467	21.528	19	5
Elixir	110.175.420	108.141	n/a[a]	1	3
ASP.NET	69.500.000	2.365.716	447	1	5
Golang	n/a[a]	n/a[a]	49.113	0	1
Java	n/a[a]	n/a[a]	15.001	5	4
Scala	n/a[a]	n/a[a]	319	0	3

[a]This information is not disclosed by the package manager.
[b]Client-side implemented template engines are typically provided by content delivery networks and therefore do not need to be downloaded with the package manager.
[c]Ruby has a default TE called ERB that ships with Ruby. Therefore, only explicit downloads of another ERB version are included in this statistic.

Python Python uses the pip package manager, which can install packages from several indexes. The largest package index is pypi[8]. The pypi website does not give any information about the number of downloads, but a third-party website[9] can be used to show the number of total and monthly downloads of a pypi package. The template engines with more than four million downloads are listed in Table 3.3.

The web frameworks Flask (2012 million downloads) and Sanic (122 million downloads) use Jinja2 as the default TE. Tornado (803 million downloads) has its own template engine, while Django (377 million downloads) uses both its own template engine and Jinja2. Fastapi (204 million downloads) does not use a default TE, but recommends using Jinja2. Bottle (86 million downloads) also has its own template engine called SimpleTemplate.

All template engines listed in Table 3.3 have been implemented in the playground.

JavaScript npm is a package manager for JavaScript. The official website[10] shows the number of weekly downloads and the number of dependents for each package. To compare the number of monthly downloads with other template engines, the

[8]`https://pypi.org/`
[9]`https://pepy.tech/`
[10]`https://www.npmjs.com/`

Table 3.3: The statistics of the selected Python template engines sorted by total downloads as of May 26, 2023.

	Downloads			
TE	Total	Monthly	CVEs	Latest Release
Jinja2	3.526.760.554	110.003.992	5	04.2022 (3.1.2)
Tornado	797.201.003	24.901.269	5	05.2023 (6.3.2)
Mako	687.082.652	24.085.174	2	11.2022 (1.2.4)
Django	374.904.154	9.772.203	5	05.2023 (4.2.1)
SimpleTemplate (Bottle)	86.281.975	2.030.476	4	03.2023 (0.12.25)
Pystache	81.205.055	1.396.423	0	12.2021 (0.6.0)
Cheetah3	16.229.032	408.409	0	07.2021 (3.2.7b1)
Chameleon	4.577.748	76.523	0	03.2023 (4.0.0)

number of weekly downloads was multiplied by four. A third party website[11] can be used to find out the number of total downloads. The template engines with more than 10 million downloads are listed in Table 3.4.

For the JavaScript TEs, it should be noted that except for Angular, all other TEs can be embedded with a script tag on the client side. This means that the package does not need to be downloaded. For example, there are several Content Delivery Networks (CDNs), such as cdnjs[12], that provide the JavaScript file to be imported with the script tag. Because of this client-side integration, the usage of the JavaScript template engines can be much higher than the number of downloads provided by npm would suggest. Another important note is that AngularJS has not been maintained since January 2022 and the current version is affected now by five CVEs. Nevertheless, this TE still is downloaded almost two million times per week using npm.

There are many large web frameworks for JavaScript. Most of them (including react.js, express.js, and next.js) do not have a template engine by default. The three web frameworks Vue.js, AngularJS, and Angular each have their own template engine.

All TEs listed in Table 3.4 have been implemented in the playground except for Angular. Angular could not be successfully implemented in the playground within a given time frame. AngularJS does not support server-side usage, so it was included client-side via a script tag. The other TEs were implemented server-side because the potential consequences of an SSTI are more extensive than those of a CSTI.

[11] `http://npm-stats.org/`
[12] `https://cdnjs.com`

Table 3.4: The statistics of the selected JavaScript template engines sorted by total downloads as of May 26, 2023.

	Downloads				
TE[a]	Total	Monthly	Dependents	CVEs	Latest Release
EJS	846.918.758	49.905.284	12.493	5	03.2023 (3.1.9)
Handlebars	817.379.329	47.418.028	13.123	11	02.2021 (4.7.7)
Underscore	746.142.833	44.259.608	22.863	1	09.2022 (1.13.6)
Vue.js	417.686.075	15.639.032	67.457	4	05.2023 (3.3.4)
Angular	233.824.845	12.862.320	13.454	1	05.2023 (16.0.2)
Mustache.js	220.073.918	13.355.200	4.554	2	03.2021 (4.2.0)
Pug	100.277.984	5.148.800	3.206	1	02.2021 (3.0.2)
AngularJS	40.540.404	1.954.520	4.132	31[b]	04.2022 (1.8.3)
Hogan.js	34.684.968	1.764.864	540	0	06.2014 (3.0.2)
Nunjucks	33.668.137	1.992.048	2.293	3	04.2023 (3.2.4)
Dot	32.608.519	1.855.052	542	1	07.2020 (1.1.3)
Velocityjs	32.236.352	1.731.864	140	0	01.2022 (2.0.6)
Eta	16.549.758	1.314.224	129	2	05.2022 (2.2.0)
Twig.js	13.778.384	696.024	365	0	02.2023 (1.16.0)

[a]Client-side implemented template engines are typically provided by content delivery networks and therefore do not need to be downloaded with the package manager.
[b]Five of the 31 CVEs affect even the latest version (1.8.3).

Ruby Rubygems[13] is Ruby's package ("gem") hosting service. Among other statistics, it shows the total number of downloads. The template engines with more than 30 million downloads are shown in Table 3.5.

Table 3.5: The statistics of the selected Ruby template engines sorted by total downloads as of May 26, 2023.

TE[a]	Total Downloads	CVEs	Latest Release
Erubi	274.457.236	0	12.2022 (1.12.0)
Erubis	252.591.965	1[b]	04.2011 (2.7.0)
Haml	122.507.906	4	12.2022 (6.1.1)
Liquid	64.527.452	2	07.2022 (5.4.0)
Slim	55.831.034	0	05.2023 (5.1.1)
Mustache	51.572.186	0	12.2019 (1.1.1)
ERB (Ruby's Default)	869.642[c]	0	11.2022 (4.0.2)

[a]Client-side template engines are typically provided by content delivery networks and therefore do not need to be downloaded with the package manager.
[b]This CVE affects even the latest version (2.7.0).
[c]ERB ships with Ruby. Therefore, only explicit downloads of another ERB version are included in this statistic.

ERB is a template engine that is a so-called "standard gem". Standard gems come with the Ruby programming language by default, similar to a standard library. However, it is possible to download standard gems using the package manager if a different version is required. Only in this case it is counted as a download in the statistics. So the actual usage of ERB is probably much higher.

For Ruby two big web frameworks exist: Rails (434.9 million downloads), which uses Erubi as TE, and Sinatra (223.9 million downloads), which has no default TE.

All TEs listed in Table 3.5 have been implemented in the playground.

PHP Composer is a dependency manager for PHP. The main repository for Composer is Packagist[14]. Packagist shows, among other statistics, the number of installs for each package. The most installed template engines, with more than 5 million installs, are shown in Table 3.6.

The most installed web frameworks for PHP are Laravel (254 million downloads), which uses its own Blade template engine, Zend (75 million downloads) without a default template engine, and Symfony (74 million downloads) and slim (27 million downloads), both using Twig as default.

[13]`https://rubygems.org/`
[14]`https://packagist.org/explore/popular`

Table 3.6: The statistics of the selected PHP template engines sorted by total downloads as of May 26, 2023.

	Downloads				
TE	Total	Monthly	Dependants	CVEs	Latest Release
Blade (Laravel)	254.530.322	4.839.141	14.100	2	05.2023 (10.12.0)
Twig	248.482.851	4.237.630	5.935	5	05.2023 (3.6.0)
Mustache.php	22.488.094	550.125	332	1	08.2022 (2.14.2)
Smarty	17.298.966	568.661	488	9	03.2023 (4.3.1)
Latte	7.234.330	108.910	673	2	03.2023 (3.0.6)

All TEs listed in Table 3.6 have been implemented in the playground.

Elixir Hex[15] is the package manager for both Elixir and Erlang. In addition to the number of total and weekly downloads, the dependencies are also shown. Based on the number of weekly downloads, the number of monthly downloads have been calculated so that they can be compared with the TEs of the other programming languages. The template engines with more than five million downloads are listed in Table 3.7.

Table 3.7: The statistics of the selected Elixir template engines sorted by total downloads as of May 26, 2023.

	Downloads				
TE	Total	Monthly	Dependants	CVEs	Latest Release
EEx	n/a	n/a	n/a	0	n/a
LEEx/HEEx (Phoenix)	110.175.420	108.141	403	1	03.2023 (1.7.2)

EEx stands for "Embedded Elixir" and is the default template engine of Elixir. Since it comes with Elixir by default, there is no number of downloads available. Both LEEx and HEEx are template engines from the Phoenix web framework. HEEx is the successor of LEEx. However, both are available as default template engines along with EEx.

All template engines listed in Table 3.7 have been implemented in the playground.

[15]`https://hex.pm/`

ASP.NET NuGet[16], the package manager for ASP.NET, shows, among other statistics, the number of downloads for the packages. The template engines with more than 5 million downloads are shown in Table 3.8.

Table 3.8: The statistics of the selected ASP.NET template engines sorted by total downloads as of May 26, 2023.

	Downloads				
TE	Total	Monthly	Dependants	CVEs	Latest Release
RazorEngine	24.300.000	391.809	172	1[a]	09.2017 (4.5.1-a001)
DotLiquid	16.400.000	499.391	75	0	03.2023 (2.2.692)
Scriban	15.800.000	795.178	111	0	02.2023 (5.7.0)
RazorEngine.NetCore	6.600.000	283.274	53	0	06.2020 (3.1.0)
Fluid	6.400.000	396.065	447	0	03.2023 (2.4.0)

[a]This CVE affects even the latest version (4.5.1-a001).

It is particularly noteworthy that the RazorEngine has not been updated since September 2017, and that a critical CVE affecting all released versions exists. The vulnerability identified by the CVE is an SSTI RCE, which is exploitable even when sandbox mode is enabled.

The default web framework for ASP.NET is Blazor. It uses the Razor view engine, which allows templating. However, dynamic templates are not possible as the templates must be pre-compiled. RazorEngine and its fork RazorEngine.NetCore are template engines that use the Razor syntax to build dynamic templates. There are several other web frameworks for ASP.NET available. ABP (16.8 million downloads) uses the Razor view engine and Scriban as the default TE. ASP.NET Boilerplate (12.6 million downloads) has no default TE. Nancy (10.5 million downloads) uses its own SuperSimpleViewEngine which is based on the Razor view engine. service stack (8.9 million downloads) uses the Razor view engine.

All TEs listed in Table 3.8 have been implemented in the playground. except for RazorEngine. While RazorEngine.NetCore—a fork of RazorEngine—was easy to implement, RazorEngine was not compatible with the ASP.NET project used for the playground.

Golang Golang provides a built-in package manager and all available packages can be found on its website[17]. There are no download counts, but the number of dependents is shown as “imported by”. The template engines with more than 240 dependants are shown in Table 3.9.

[16]`https://www.nuget.org/`
[17]`https://pkg.go.dev/`

Table 3.9: The statistics of the selected Golang template engines sorted by dependants as of May 26, 2023.

TE	Dependants	Github Stars	CVEs	Latest Release
text/template	60249	n/a	0	05.2023 (1.20.4)
html/template	49221	n/a	0	05.2023 (1.20.4)
fasttemplate	734	0.7k	0	10.2022 (1.2.2)
pongo2	689	2.5k	0	06.2022 (6.0.0)
jet	527	1k	0	12.2022 (6.2.0)
ace	459	0.8k	0	07.2016 (0.0.5)
quicktemplate	316	2.8k	0	09.2021 (1.7.0)
amber	244	0.9k	0	10.2017 (0.0.0)

Both text/template and html/template are standard Golang packages that ship with the language.

Most of the major Golang web frameworks, as measured by Github stars, use html/template as their template engine. These include Gin (69k stars), Beego (29.8k stars), echo (25.8k stars), and Revel (12.9k stars). In contrast, Fiber (26.k stars) and Iris (24k stars) have several TEs integrated. These include html/template and most of the other TEs listed in Table 3.9, as well as some TEs with less than 500 stars on Github.

Only text/template and html/template have been implemented in the playground, as the other template engines listed in Table 3.9 are not relevant enough—at least at this point—compared to the two standard packages.

Java For Java two major package managers, Maven[18] and Gradle[19], exist. Since there is no practical way to search for Gradle packages, Maven was used. Unfortunately it does not provide the number of downloads. The only comparable value is the number of usages. These are equivalent to dependents, as they also indicate how many other packages use a package. The template engines with more than 400 usages are shown in Table 3.10.

Of particular note is that 1815 of Velocity's 2850 dependents use deprecated versions, all of which are affected by CVEs.

For Java, there are two major web frameworks. Spring (28 thousand usages) is the most used web framework and has no default TE. However, there are four packages that combine the Spring Web Framework with Thymeleaf. These four combined

[18]`https://mvnrepository.com/`
[19]`https://gradle.org/`

Table 3.10: The statistics of the selected Java template engines sorted by dependants as of May 26, 2023.

TE	Dependants	CVEs	Latest Release
Groovy	8997	2	05.2023 (4.0.12)
Freemarker	2716	1	01.2023 (2.3.32)
Velocity	2850	2	03.2021 (2.3)
Thymeleaf	438	0	12.2022 (3.1.1.RELEASE)

have just over one thousand usages. GWT (2.3 thousand usages) is the second most used web framework and has no default TE.

All TEs listed in Table 3.10 have been implemented in the playground.

Scala The Scala Library Index[20] indexes libraries and projects for Scala. Only the dependents, dependencies, and Github statistics are provided. The template engines with more than 100 stars on Github are listed in Table 3.11.

Table 3.11: The statistics of the selected Scala template engines sorted by dependants as of May 26, 2023.

TE	Dependants	Github Stars	CVEs	Latest Release
Twirl	289	522	0	12.2022 (1.5.2)
Scalate	30	598	0	03.2022 (1.9.8)
Beard	0	121	0	03.2020 (0.3.1)

None of the Scala template engines listed in Table 3.11 have been implemented in the playground. Both Twirl and Scalate do not allow dynamic templates. This is because they have a template compiler that is only executed when the whole project is compiled. This means that to inject a template, user input must first be entered into a template file, and then the project must be recompiled and restarted. Since this is very cumbersome and not suitable for automated scanning, the template engines were not implemented. With Beard dynamic templates should be possible, but it is far away from the relevance of the other template engines and has not been maintained for more than 3 years.

[20] `https://index.scala-lang.org/`

3.4 Design and Implementation

In the following, the structure of the website, the servers and the project is explained.

3.4.1 Website

The website consists of the following pages:

Index The index page gives a brief introduction to the playground and explains the other pages available.

TEs All implemented template engines are listed here. The template engines are grouped based on their programming languages and sorted in descending order by the number of downloads (or similar factors) for each programming language. Selecting a TE will take one to its details page.

TE details Besides a short description of the TE, this page provides the following details:

1. Links to the documentation and blog posts that contain information about template injections for the TE.
2. Links to the different "modes" implemented for the TE.
3. A collection of template expressions and syntax examples for the TE.

TE modes Every TE has the "Default" mode, where the TE has been implemented intentionally in an insecure way and without any countermeasures applied. Some TEs also have the "Sandbox" mode, where the TE was also implemented in an insecure way, but the sandbox mode of the TE was enabled. The TEs are implemented in an insecure way by concatenating user input into the string that is passed to the template engine as a template. Selecting a mode brings up a simple page that always looks the same, containing a textbox and a button. When the button is clicked, the content of the textbox is inserted into the template string and passed to the TE. The result is then displayed below the textbox.

Config This page provides access to global settings which have an effect on all modes of all template engines. The current config is displayed and the config options are explained. These five settings that can be set:

1. `hideError` can be enabled. This will intercept error messages thrown by the TE or the web server. Instead, a response is generated in which the user input is inserted without being processed by the TE. Thus, it is not possible to detect the presence of a TE or identify which TE is used from a template expression that causes an error.

2. `removeArray` allows to define a deny list containing an arbitrary number of characters and strings. The entries of the deny list will be removed from the input before it is passed to the TE. Using such a deny list approach is not atypical for user input on web pages. Depending on the length of the deny list, some or even all template injection payloads can be mitigated.
3. `htmlEncode` can be enabled, which encodes the five HTML special characters `&`, `<`, `>`, `'`, and `"` into HTML entities. This is a common countermeasure to prevent XSS and can mitigate some template injection payloads.
4. `special` is used to enable the special scenarios described in Section 3.2. It can either be an empty string or have one of the following values:

   blind ensures that the output of the TE is not returned.

   otherLocationDirect ensures that the TE output is not returned directly in the response, but at the relative URL `/otherlocationdirect/`, which is referenced in the response.

   otherLocation is similar to "otherLocationDirect", but the TE output is provided at the relative URL `/otherlocation/`, which is not referenced in the response.
5. `csrfProtection` can either be an empty string or have one of the following values:

   static In this case, a random but fixed CSRF token is generated, which must be provided with every POST request.

   nonce Each time a page containing a POST form is requested, a new random CSRF token is generated, which must be provided in the next POST request.

Links This page contains a collection of links to the different modes of all TEs. The following two special URLs are also provided:

1. `/otherlocation/` contains the last generated TE output, if the config setting `special` contains the value `otherlocation`.
2. `/echo/` is a page structured exactly like the pages where user input is processed by a TE. However, instead of the user input being processed by a TE, the user input is inserted into the template and returned directly. If `removeArray` or `htmlEncode` is enabled in the config, the user input may have been modified. The echo page is provided in order to potentially trigger a false positive if the user input has been modified in any way other than by a TE.

Random TE This provides a page with a randomly selected TE mode. Anything that points to the template engine will be removed.

3.4.2 Servers

The playground consists of a total of nine Docker containers, each of which contains a web server. Eight of these web servers serve web pages with all template engines for one of the eight programming languages. The ninth web server is NGINX[21], which is very versatile and can also be used as a reverse proxy and load balancer. While the other web servers are only accessible within the Docker network, NGINX is also accessible from the host machine. NGINX performs several tasks within the playground:

- NGINX serves all the pages of the playground, except for the pages with the built-in template engines, which are served by the other web servers.
- NGINX handles the routing of HTTP requests and responses. For example, if the path of a URL in the playground starts with `/python/`, `/php/`, or another programming language, NGINX routes the request to the appropriate web server. This allows web pages with the TEs to be requested from the host machine, and at the same time, everything can be served from a single host. This ensures that the playground is a unified website and does not need to switch between multiple hosts.
- The global settings are implemented with NGINX altering the HTTP requests and responses. This is made possible by the njs scripting language[22], which is based on JavaScript. HTTP requests are altered in different cases. For example, NGINX decodes or removes HTML characters before sending a request to one of the web servers, if this is desired according to the configuration. NGINX also checks if CSRF protection is enabled and if the CSRF token is correct.
- The HTTP responses of the web servers are altered, for example, if the config specifies that the TE output should not be returned (blind scenario) or should be rendered on another page (otherLocation or otherLocationDirect scenario). In the first case, the TE output is removed; in the second case, the output is saved to a file and also removed from the response. When the `/otherLocation/` page is requested, the last saved TE output can be read from the file. In the otherLocationDirect scenario, the saved TE output can be read from `/otherLocationDirect/` and a link to this path is also added to the returned response.

Since NGINX does not provide any functionality to set a variable with njs in one request and read it again in another request, saving and reading files is used as a workaround.

Figure 3.1 shows a sample request to `/python/Jinja2`. An HTTP GET request is sent to NGINX, which is accessible from the local machine. NGINX recognizes

[21]`https://www.nginx.com/`
[22]`https://NGINX.org/en/docs/njs/`

from the path beginning with `/python/` that the request should be forwarded to the python web server. The Docker network is configured so that the python webserver can be reached internally via `http://python:13375`. Before forwarding the request, NGINX may make changes to the request, such as removing characters from the user input that are on the deny list. When NGINX receives a response from the python web server, it can again make changes, such as removing template engine output if the blind scenario is enabled. Finally, the response is passed to the browser. It is not possible to call the web servers directly because they are in an internal Docker network and only NGINX can be used as an interface to the outside.

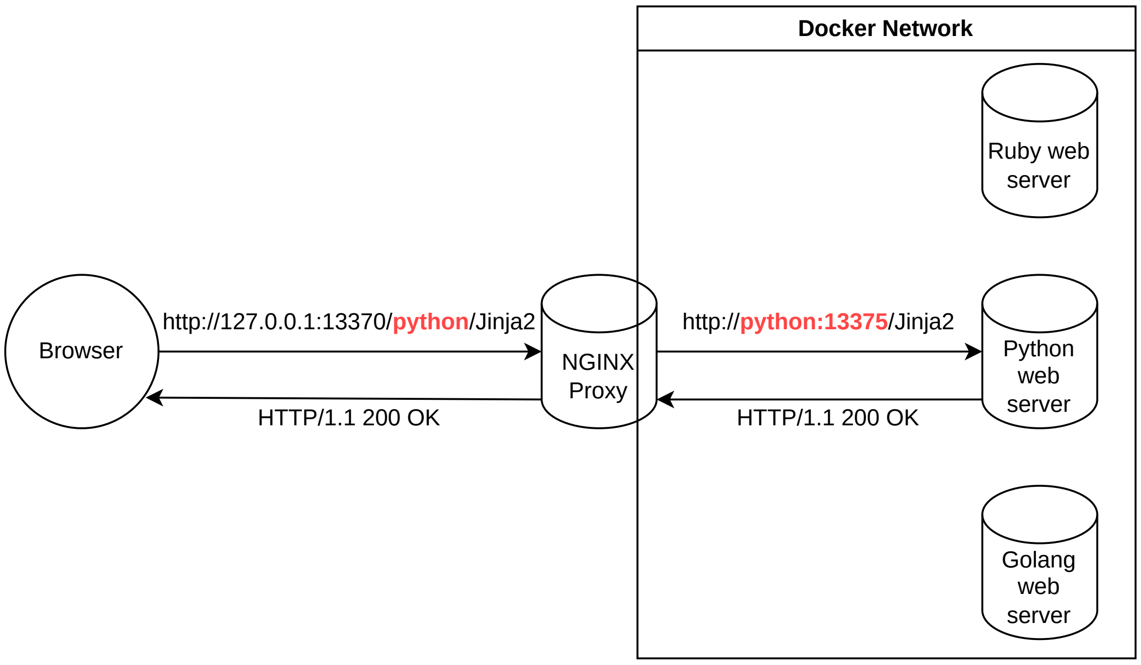


Figure 3.1: A sample request to `/python/Jinja2` that NGINX passes to the Python web server.

3.4.3 Project

The main directory of the project consists of nine folders, one folder for each web server, and the two files `docker-compose.yml` and `README.md`. `README.md` contains the project description written in Markdown. `docker-compose.yml` contains the path to the `Dockerfiles` for all web servers. The web server folders contain the `Dockerfile`, which contains all the information Docker needs to build and start the web server. The web server folders also contain a folder with the project files for each web framework used, which are imported into each Docker container when it is created. The `nginx` folder contains the `Dockerfile` and two other folders. The `root` folder contains the HTML, CSS, JavaScript, and image files for the playground website. The `njs` folder contains JavaScript files, which contain the methods that NGINX uses to manipulate the HTTP requests and responses, and the files used to store the CSRF token or config.

3.5 Setup and Usage

Shortly after the thesis is published, the playground will be made available to the public. To use the playground, there are two requirements. First, a Docker Compose installation must be in place[23]. Second, the Template Injection Playground Github repository needs to be downloaded. It is recommended to download the latest release.

In the root directory (where the `docker-compose.yml` file is located), there are two commands that need to be run to get the playground up and running.

docker compose build This command builds the services needed for the playground.

docker compose up This command builds the containers and starts them.

Finally, the playground is served at `http://127.0.0.1:13370`.

[23] `https://docs.docker.com/compose/install/`

4 Improving the Template Injection Methodology

This chapter describes the procedure for improving the *Detect* and *Identify* steps of the template injection methodology and the corresponding results. First, we define four different types of polyglots for template injection detection. Then, we compare the existing polyglots and create new polyglots that are shorter and work for more template engines. In particular, we design a new type of template injection polyglots that does not aim to throw errors. Next, we explain how these detection polyglots can also be used to efficiently identify template engines. Finally, we present the *Template Injection Table*. This is a web page with an interactive table containing the analyzed polyglots and the expected responses of the tested template engines. Among other things, the table can be used as an aid when performing the template injection methodology in order to use the efficient polyglots.

4.1 Detection of Template Injection Possibilities

The possibility of template injection can be detected by trying different template expressions as an input for the parameter in question. However, there are a large number of possible template engines that use different syntaxes and provide different features. Thus, the number of template expressions that need to be tested to ensure that no template injection is possible increases almost linearly. This is where polyglots come in handy. In computer science, the term *polyglot* is used to describe a type of software that can be interpreted by multiple programming languages. In the case of template injections, polyglots are template expressions that can be interpreted by multiple different template engines. This can significantly reduce the number of template expressions required when testing whether a template engine is in use. This is especially important for large-scale scans, since each additional HTTP request must be multiplied by the number of parameters to be tested per URL. In this thesis, there are four different types of polyglots distinguished:

1. Universal Error
2. Language-Specific Error
3. Universal Non-Error
4. Language-Specific Non-Error

An error polyglot causes template engines to throw an error.

A non-error polyglot, on the other hand, avoids causing template engines to throw an error. Instead, it aims to have the template engine successfully render an output.

A universal polyglot is intended to be interpreted by as many different template engines as possible.

A language specific polyglot, on the other hand, limits the set of targeted template engines to one programming language. Because of the smaller set of template engines to be addressed, it is possible to create much shorter polyglots. The polyglot length is relevant if the input length is limited.

All of the polyglots examined and created during this thesis work the same way for JavaScript template engines regardless of whether the engine is running client-side or server-side. This is because the polyglots only use basic features such as printing numbers, comments, and simple comparisons that are supported by both clients and servers.

4.1.1 Universal Error Polyglots

Searching for template injection polyglots in the public domain using articles, blog posts, and cheat sheets about template injection, only two polyglots could be found. Both polyglots are classified as universal error polyglots. The first one is `${<%[%'"}}%\` (PG1) and the second one is `${<%[%'"}}%\.` (PG2). The only distinction between them is that the latter has a dot appended to it. Unfortunately, the origin of these two polyglots could not be determined. In addition, Miguel Reis Silva created three polyglots in his master thesis using 15 template engines [8]. These are `<#set($x<%={{@{#{${xux}}%>)` (PG3), `<%={{@{#{${xu}}%>` (PG4), and `<th:t=\"${xu}#foreach` (PG5). However, we could not find any mention of these polyglots in any of the common cheat sheets, articles, and blog posts.

All five polyglots were tested in the 51 different test cases provided by the Template Injection Playground. These test cases consist of all modes of 44 template engines implemented in the playground. Some template engines have an additional mode in addition to the default mode in which, for example, the sandbox is activated. Therefore there are a few more test cases than template engines. The two template engines LEEx and HEEx were not included because they do not support dynamic templates. In addition, the test was carried out with the default configuration, so that there were no countermeasures in place. Finally, the polyglots were evaluated according to the following criteria:

Error The number of template engines throwing an error.

Modified The number of template engines rendering something other than the unmodified polyglot.

Total detected The total number of detected template engines. This is the sum of the two criteria above.

Length The number of characters the polyglot consists of. Since the possible input length may be limited, it is important that a polyglot is as short as possible.

The results are shown in Table 4.1. In addition to the five polyglots mentioned above (PG1 - PG5), two new universal error polyglots were created and evaluated: `<%'${{#{@}}%>` (PG6) and `<%'${{/#{@}}%>{{` (PG7). PG6 led to the detection of template injection in all 51 test cases and is even—together with PG1—the shortest polyglot. For comparison: PG3 detects only one template injection less, but it is more than twice as long. The polyglots PG1 - PG6 do not throw an error in all template engines. However, some render the polyglot modified. Therefore, PG7 was developed based on PG6. With PG7, the template engines throw an error in all 51 test cases. However, it is also three characters longer than PG6, for which only four template engines do not throw an error, but instead render the polyglot modified. PG1 and PG2 behaved exactly the same during testing. PG2 may originate in a copy-paste error: PG1 might have been at the end of a sentence, and the dot referred to the end of the sentence and was not supposed to be part of the polyglot.

Table 4.1: The evaluation results of the seven examined universal error polyglots, measured on the template injection playground. `Abbr.` is the abbreviation of the respective polyglot. `Mod.` indicates in the number of test cases in which the polyglot was rendered modified. `Error` indicates the number of test cases in which the polyglot caused the template engine to throw an error. `Total Det.` is the sum of `Mod.` and `Error`. `Length` is the length of the polyglot.

Polyglot	Abbr.	Mod.	Error	Unmod.	Total Det.	Length
`${<%[%'"}}%\`	PG1	2	37	12	39	**13**
`${{<%[%'"}}%\.`	PG2	2	37	12	39	14
`<#set($x<%={{={@{#{${xux}}%>)`	PG3	4	46	1	50	29
`<%={{={@{#{${xu}}%>`	PG4	4	44	3	48	19
`<th:t=\"${xu}#foreach.`	PG5	1	15	35	16	22
`<%'${{#{@}}%>`	PG6	4	47	0	**51**	**13**
`<%'${{/#{@}}%>{{`	PG7	0	**51**	0	**51**	16

In his master's thesis, Miguel Reis Silva described in detail how he created his polyglots. Among other things, the opening and closing tags of 15 template engines were strung together and shortened by rearranging and summarizing. The template engines would then throw an error due to unexpected special characters or an unknown variable between the opening and closing tags. The resulting PG3 was able to trigger an error message for 12 of the 15 template engines, while the other three rendered the polyglot unprocessed. The three template engines that rendered the polyglot unchanged were FreeMarker, Velocity, and Thymeleaf [8]. However, when PG3 was tested with the Template Injection Playground, FreeMarker and Velocity threw an error. When checking the different behaviors, it came out that Silva had a bug in his test setup. The web application catches error messages for both FreeMarker[1] and Velocity[2] and does not return the error message. Accordingly, it was not possible to receive an error message with these two template engines.

The polyglots PG6 and PG7 created for this thesis were created in a similar manner to Silva's PG3. First, the opening and closing tags of all 45 implemented template engines were collected. Most template engines use `<% %>`, `${ }`, `{{ }}`, and `#{ }`. The resulting polyglot `<%${#{}}}}%>` can be shortened by merging the two opening tags `${` and `{{` into `${`. Similarly, the multiple closing tags `}}}}` can be shortened to just `}}`. It is not possible to remove either of the closing tags `}}` or `%>`, because some template engines only recognize something as a template expression if both opening and closing tags are present. This results in `<%${#{}}%>`, which is very close to the polyglot PG6 except for two crucial characters. RazorEngine uses `@( )` for statements or just an `@` followed by a string to reference variables. For example, an `@` can be added between `#{` and `}}`, resulting in `<%${#{@}}%>`. This will cause RazorEngine to throw an error because `}` is an unexpected special character after an `@`. At the same time, the engines with the opening tag `#{` throw an error because of the unexpected special character `@`. The only template engine for which no opening tag is present in the polyglot yet, and therefore would not react at all, is Thymeleaf. This template engine uses either the HTML tag attribute `th:text="${ }"` or the opening and closing tags `[[${ }]]` or `[(${ })]`. To add Thymeleaf's opening and closing tags would require at least four additional characters. However, Thymeleaf throws an error when it encounters the string `<'`, as well as `<%'`. Why the error is thrown is not clear from the error message or the documentation. Nevertheless, this behavior requires only one additional character instead of four. This procedure finally resulted in the polyglot PG6 `<%'${{#{@}}%>`.

Although we used the same methodology as Silva for his PG3 (`<#set($x<%={{@{#{${xux}}%>))`), our PG6 (`<%'${{#{@}}%>`) is less than half as long and detects one template engine more. This is mainly due to four differences:

1. Silva used opening tags for expressions where the result should be rendered. For example, `<%= 7*7 %>` would compute the result `49` and render it, while `<% 7*7 %>` would also compute `49` but not render the result. However, to

[1]`https://github.com/DiogoMRSilva/websitesVulnerableToSSTI/blob/master/java/springBased/src/src/main/java/Main.java#L66C13-L66C30`

[2]`https://github.com/DiogoMRSilva/websitesVulnerableToSSTI/blob/master/java/springBased/src/src/main/java/Main.java#L143`

throw an error, it is not relevant to instruct the template engine to render the result, since it is supposed to throw an error anyway.

2. `xux` is supposed to refer to a non-existent variable, which should throw an error. Instead, an unexpected special character that is already needed for other reasons can be used. For example, the three extra characters can be avoided by using the `@` character instead of `xux`.

3. `#set($x)` is only available for the template engine Velocity and is an expression to set the value of the variable `x`. However, it is possible to make Velocity throw an error in a much shorter way. `#DIRECTIVE` or `#{DIRECTIVE}` tells Velocity to use a directive such as "set" to set variables. If an unknown directive name such as `#{foo}` is used, nothing happens and the expression is rendered unchanged. However, if the directive name contains an unexpected special character such as `@`, Velocity will throw an error. Thus, `#{@}` in PG6 causes Velocity to throw an error without requiring additional characters just for Velocity, since they are already present for other template engines.

4. The fact that Thymeleaf throws an error for `<'` or `<%'` was probably not known, nor were the alternative tags `[[${ }]]` and `[(${ })]`.

PG6 has a detection rate of 100% only if the user input is reflected for four of the template engines (Liquid, DotLiquid, Scriban, Pystache). If this is not the case, and the template injection can only be detected by triggering errors, the polyglot is successful in only 47 of the 51 test cases. To overcome this shortcoming, `<%'${{/#{@}}%>{` (PG7) was created to trigger an error message instead of a modified response for the remaining four test cases. Liquid and Pystache turned out to be the most difficult ones to make throwing errors. In our tests, Liquid only failed when an opening tag (`{{`) was not followed by a closing tag (`}}`). In other cases, such as division by zero or references to non-existent variables, only an empty string was returned instead of an error message. To address these issues, PG6 was appended with `{{` resulting in `<%'${{#{@}}%>{{`. Fortunately, not only Liquid, but also DotLiquid and Scriban throw an error when using this polyglot. This is because they use the same opening and closing tags and also throw an error if an opening tag is not followed by a closing tag. Pystache is a Mustache implementation for Python and thus belongs to the logic-less template engines. These engines offer only a very small range of functions. Therefore the possibilities to throw errors are rather limited. However, Mustache, and therefore Pystache as well, does offer a feature called "sections". Everything within a section can be rendered any number of times. For example, a section named "person" starts with `{{#person}}` and ends with `{{/person}}`. If a section that was not open before is closed, Pystache throws an error. In the case of the polyglot `<%'${{#{@}}%>{{` another `/` was added after the first `{{`, resulting in PG7 `<%'${{/#{@}}%>{{`. PG7 is the only polyglot to throw an error in all 51 test cases, and the only one of the universal error polyglots examined to throw an error in Liquid and Pystache.

4.1.2 Language-Specific Error Polyglots

The language-specific polyglots each cover all template engines of one specific programming language. Focusing on these subsets of the 51 test cases enables shorter polyglots. This can be relevant, for example, if the universal error polyglot PG6 with its 13 characters is too long. Even with less than 13 characters, template injection vulnerabilities can be exploited. Two examples are the nine-character expression `<%=`ls`%>` of the ERB template engine, which returns the files in the directory in which the web server is running, and the 10-character expression `{{config}}` of the Jinja2 template engine, which returns the global config variable. In the case of the Flask web server this variable includes the `SECRET_KEY` used for signing.

Table 4.2 shows the language-specific polyglots created. The average length of these polyglots is only 5.75 characters. Compared to the 13 characters of polyglot PG6 or the 16 characters of polyglot PG7, this is a significant reduction.

Table 4.2: The evaluation results of the eight created language-specific error polyglots, measured on the template injection playground. `Language` specifies the programming language of the template engines for which the polyglot was designed. `Modified` indicates in the number of test cases in which the polyglot was rendered modified. `Error` indicates the number of test cases in which the polyglot caused the template engine to throw an error. `Unmodified` indicates the number of test cases in which the polyglot was rendered unmodified. `Total Detected` is the sum of `Modified` and `Error`. `Length` is the length of the polyglot.

Polyglot	Language	Modified	Error	Unmodified	Total Detected	Length
`${{/#}}`	Python	3	35	13	38	7
`<%${{#{%>}}`	JavaScript	4	44	3	48	11
`<%{{#{%>}`	Ruby	3	37	11	40	9
`{{/}}`	PHP	4	28	19	32	5
`<%`	Elixir	2	8	41	10	2
`{{@`	ASP.NET	1	25	25	26	3
`{{`	Golang	3	22	26	25	2
`<%'#{@}`	Java	2	19	30	21	7

The language-specific polyglots are based on PG7 (`<%'${{/#{@}}%>{{`). Characters unnecessary for the specific TEs were removed and the order was partially changed. This simple but effective approach allowed to quickly create the language-specific polyglots.

4.1.3 Universal Non-Error Polyglots

As long as the errors are not caught, the error polyglots are very efficient. However, they can usually be mitigated by a simple measure, namely catching the errors instead of returning them to the user. In this case, the error polyglots cannot be used to detect template injection possibilities unless there are behavioral differences. Examples of such differences include:

- faster response times, because the template engine stops processing the template earlier due to the error.
- a generic error page.
- not only the user input, but also the rest of the template is returned without having been processed by the template engine.

To be able to detect template injection possibilities if errors are caught, non-error polyglots were created. The big challenge with non-error polyglots is that as many template engines as possible should render the polyglot modified instead of throwing an error. This is where problems quickly arise, as Blade renders a `1` when encountering `{{1}}`, while Django throws an error, for example. This can be worked around with comments, for example by using the expression `{#{1}}#}`. In Django, `{#` is the start and `#}` is the end of a comment. So Django will render nothing when it encounters `{#{1}}#}` instead of throwing an error. Blade, on the other hand, renders `{#` and `#}` unchanged, since it does not know this syntax, and therefore ends up rendering `{#1#}`.

In the end, three universal non-error polyglots were created. All three together render at least one modified response in all 51 test cases, while being only 12-14 characters long. For most template engines, a `1` was placed between various opening and closing tags. This makes the expression between the opening and closing tags as short as possible to reduce the likelihood that a template engine using these opening and closing tags will throw an error. However, the template engine mustache.js already throws an error when encountering `{{1}}`. The shortest expression that is rendered modified for mustache.js is `{{.}}`. For other template engines, the comments `{##}`, `##`, `/**/`, and `@*` were used because they were significantly shorter and less error-prone than other approaches, such as rendering a number. Finally, `">` was needed for the test case where Thymeleaf inserts the user input into an inline expression to close it, and `p ` is needed for Pug, which throws an error for special characters at the beginning of lines. At this point, the individually created template expressions such as `#{1}`, `/**/`, and `@*` were combined in various ways until a combination was created in which each template engine rendered the polyglot at least once modified. However, since some template engines consistently throw errors when encountering template expressions added for other template engines, three separate polyglots had to be created. These three universal non-error polyglots are shown in Table 4.3, along with their benchmarks measured on the playground.

Table 4.3: The evaluation results of the three created universal non-error polyglots, measured on the template injection playground. `Modified` indicates in the number of test cases in which the polyglot was rendered modified. `Error` indicates the number of test cases in which the polyglot caused the template engine to throw an error. `Unmodified` indicates the number of test cases in which the polyglot was rendered unmodified. `Total Detected` is the sum of `Modified` and `Error`. `Length` is the length of the polyglot.

Polyglot	Modified	Error	Unmodified	Total Detected	Length
`p ">[[${{1}}]]`	34	5	12	39	14
`<%=1%>@*#{1}`	22	2	27	24	12
`{##}/*{{.}}*/`	15	20	16	35	13

4.1.4 Language-Specific Non-Error Polyglots

If the programming language in which a website is written is known, for example by enumerating the web framework in use, the language-specific non-error polyglots can be used instead of the universal non-error polyglots. These polyglots cover all test cases for one specific programming language with only one polyglot; an exception is the JavaScript polyglot, which renders the polyglot modified for all JavaScript template engines except AngularJS and Pug (Inline). Pug (Inline) is an alternative mode to Pug's default mode. In this mode, the user input is inserted after a `p` followed by a space that instructs Pug to create a paragraph. For six programming languages, the polyglot is shorter than the universal ones, for Ruby it is a few characters longer, and for JavaScript it is significantly longer. Especially for JavaScript template engines the problem exists that many engines use identical opening and closing tags, but a partially different expression syntax. Thus, many comments had to be used to comment out error throwing expressions for certain template engines. These eight language-specific non-error polyglots are shown in Table 4.4, along with their benchmarks measured on the playground.

4.2 Identification of Template Engines

After detecting a template injection possibility, it is important to identify the template engine in use in order to use template injection payloads designed for that specific template engine. Similar to the template injection detection, polyglots can be used to identify the specific template engine, significantly reducing the number

Table 4.4: The evaluation results of the eight created language-specific non-error polyglots, measured on the template injection playground. `Language` specifies the programming language of the template engines for which the polyglot was designed. `Mod.` indicates in the number of test cases in which the polyglot was rendered modified. `Error` indicates the number of test cases in which the polyglot caused the template engine to throw an error. `Unmod.` indicates the number of test cases in which the polyglot was rendered unmodified. `Total Det.` is the sum of `Mod.` and `Error`. `Length` is the length of the polyglot.

Polyglot	Language	Mod.	Error	Unmod.	Total Det.	Length
`{#${{1}}#}}`	Python	28	11	12	39	11
`//*<!--{##<%=1%>{{!--{{1}}--}}-->*/#}`	JavaScript[a]	31	14	6	45	37
`<%=1%>#{2}{{a}}`	Ruby	32	14	5	46	15
`{{7}}}`	PHP	26	6	19	32	6
`<%%a%>`	Elixir	8	6	37	14	6
`{{1}}@*`	ASP.NET	28	5	18	33	7
`{{.}}`	Golang	11	20	20	31	5
`a">##[[${1}]]`	Java	15	2	34	17	13

[a]Only 12 of the 14 JavaScript test cases rendered the polyglot modified. AngularJS and Pug (Inline) rendered the polyglot unmodified

of requests required to identify the template engine. Thus, it is not necessary to create a template expression for each single template engine that provides a unique result. Instead, a small number of polyglots can be used to iteratively eliminate more and more template engines from the list of possible candidates until only one remains. This way, the polyglots created for template injection detection can also be used for template engine identification. In 38 of the 51 test cases, an unambiguous identification of the template engine is possible if errors are caught. In addition, unambiguous identification is possible in six more test cases when errors are not caught. No further distinction was made between ERB, Erubi and Erubis, since all three are identical except for minor differences; especially the existing exploits are identical. In order to be able to clearly identify the template engine in use in the remaining 13 test cases, three more polyglots were created.

The first polyglot is `{{1in[1]}}`. It allows to distinguish Blade from SimpleTemplate, Tornado from Twig (Sandbox), and the three template engines Jinja2, Nunjucks, and Twig from each other.

The second polyglot is `${"<%-1-%>"}`. It allows to distinguish Underscore from Eta, Mako from Chameleon, and the three template engines EEx, EJS, and ERB from each other.

The third polyglot is `#evaluate("a")`. It allows to distinguish Velocity and Velocity.js from each other.

Based on the results, a decision path graph could be created, as Kettle did to distinguish four template engines from each other[6]. However, with 51 different test cases for 44 different template engines, such a graph would be far too cluttered. Therefore, a web page with an interactive table was created, which is presented in the following Section 4.3.

4.3 Template Injection Table

The findings from the polyglots, which were created to detect template injection possibilities and identify template engines, have been summarized in an interactive table on a web page. This table can be used as an aid to first detect a template injection possibility and then gradually exclude template engines from the list of possible candidates with few polyglots until only one template engine remains. The web page will be made available to the public shortly after the thesis is published.

Each polyglot is a column header and each template engine has its own row. In this way, either the answers of all template engines can be viewed vertically for each polyglot, as well as the answers of all polyglots can be viewed horizontally for each template engine. Furthermore, each polyglot can be easily copied to the clipboard using a copy icon, and one of the answer options can be selected for each polyglot using drop-down menus. This filters out the template engines to which the selected answers do not apply. Furthermore, for each template engine, information such as programming language, tested version, and links to the documentation and package manager are given. Especially the information about the tested versions can be relevant, as shown in the following.

Limitations The polyglots and all other tests were created with the latest version of the template engines at time of writing. The versions used are listed in the Template Injection Table and in the tables in Subsection 3.3.2. It is possible that the polyglots will return a different result with an older or newer version of a template engine. This is rather unlikely, since simple expressions were used for the most part, and "breaking changes" to the template engine syntax will probably be avoided by the developers. However, this possibility does exist.

Furthermore, the template engines were used in their default configuration. Any changes to the configuration may cause the template engine to react differently to a polyglot than expected. For example, Jinja2 is one of the few template engines where the opening and closing tags can be configured. `${`, `#{`, or something arbitrary can be defined as an opening tag instead of `{{`.

Other factors that can alter template engine responses in the wild include changes to user input or template engine output by the website, proxies, or other components.

If certain characters are removed or changed from the user input, or even if new characters are added, the template engine no longer receives the original polyglot. As a result, the template engine's response may be completely different from what was anticipated.

5 Template Injection Scanner

This chapter introduces the template injection scanner, called *TInjA* (short for Template Injection Analyzer), which allows to automatically perform the first two steps of template injection methodology—detection and identification. First, we present the two use cases for which the scanner was developed. Then we list the key features of TInjA. This is followed by an explanation of the program flow, which consists of a main flow and a scan flow. Next, we explain the measures how the scanner detects template injection possibilities and identifies template engines. Finally, we describe the setup and usage of the scanner.

5.1 Use Cases

TInjA is designed to scan single web pages as well as an arbitrary number of web pages for possible template injection possibilities. The scanner can detect both client-side and server-side template injection possibilities. Once a template injection has been detected, further tests are performed to identify the template engine in use. Whether a single web page or a large number of web pages are to be scanned are different use cases, which can significantly differ in the requirements for the scanner.

If a single web page is to be scanned, the scanner can be manually tailored to the specific web page. This includes, for example, setting query or POST parameters of the web page or specifying a session cookie.

With a large number of web pages that may originate from many different websites, it would be impractical to manually set such parameters, cookies, etc. for each individual web page. Therefore, TInjA allows importing a JSONL file that contains this information for each web page to be tested. A crawler such as *Katana*[1], which was used for the large-scale scan described in Chapter 7, can automatically crawl a list of web pages and add this required information to a JSONL file. The JSONL file must contain a JSON object with the structure shown in Listing 7 on each line.

[1]`https://github.com/projectdiscovery/katana`

```
{
        "request":{
                "method":"POST",
                "endpoint":"http://example.com/path",
                "body":"name=kirlia",
                "headers":{
                        "Content-Type":"application/x-www-form-urlencoded"
                }
        }
}
```

Listing 7: The structure of JSON objects expected by Tinja in a JSONL file

5.2 Key Features

TInjA provides the following key features:

Automated execution of the first two steps of the template injection methodology The scanner automatically performs the detection of template injections. Once a template injection is detected, the template engine is identified.

Support for the most relevant template engines The scanner supports the 44 most relevant template engines and can uniquely identify each one. The most relevant template engines are discussed in chapter 3.4.

Report If desired, a report can be generated that contains information about template injection vulnerabilities identified, error messages, and general information such as the settings with which the scanner was initiated. The report is appended with the results of a URL after the scan for this specific URL has been finished. This ensures that the performance required to update large reports remains low compared to, for example, keeping a large ever-growing JSON object in memory and rewriting the entire file.

Efficiency Polyglots are used for detection and identification. Thus, only a few requests are needed to identify a large number of template engines. In addition, the average RAM usage during an active scan is typically between 20-40 MiB.

Customization A total of 23 flags can be used to customize the behavior of the scanner. For example, it is possible to set custom headers and parameters, or to set a rate limit. However, except for the one flag that is used to provide the URLs to be tested, all flags are optional.

5.3 Program Flow

To illustrate TInjA's program flow, two simplified flow charts have been created. Figure 5.1 shows the main program flow, while Figure 5.2 shows the program flow of scanning a URL in more detail. Both flows are described in the following.

5.3.1 Main Program Flow

First the report file is created and some information is written into it; for example, the settings and URLs with which TInjA was initiated. Then a loop is started which scans all URLs one by one. The more detailed scan process is described in the scan flow below. When the scan of a URL is finished, the results are appended to the report file. If there are still URLs left that have not been scanned yet, the scan of the next URL is started. Otherwise, information such as how many template injections were found and how long the scan took in total is added to the report file. After that, the process is terminated successfully.

The report file is only generated if the `--reportpath` flag is set. Otherwise, the program flow skips the initial creation, appending, and finishing steps of the report.

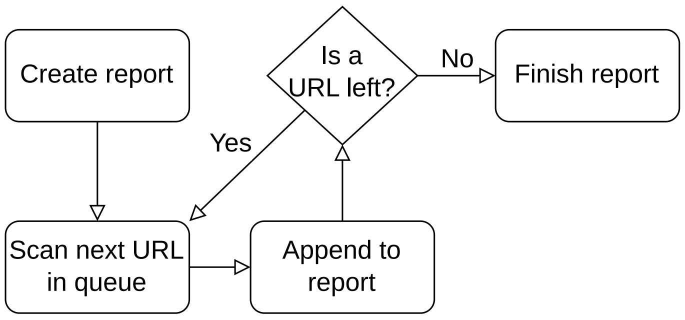


Figure 5.1: TInjA's main program flow summarized.

5.3.2 Scan Flow

The scan flow is executed one by one for each parameter and some headers of a request. The `Host`, `X-Forwarded-For`, `Origin` and other headers which have a higher chance of being inserted into a template, for example, to reference the domain a template engine is rendering output for, get scanned. First a "default request" is sent. Default in this case means that the URL will be called as specified by the command line flags. That is, if custom headers, cookies, parameters, or the like have been specified, they will be added to the request, but no payloads will be inserted yet. The response to the default request is important as a comparison to the responses of the requests where the polyglots were inserted later. After the default request has finished, a "reflection request" is sent. This checks if the user input is reflected in the response by replacing the user input with a randomly generated string, which is sixteen characters long by default. The response to the reflection request will then be searched for this random string. The information if and where the user input is reflected is important for evaluating whether a polyglot is rendered modified or not.

Next, the first polyglot is sent, namely the universal error polyglot PG7 (`<%'${/#{@}} %>{{`). This will result in an error message for all 44 examined template engines. At this point, there are four different possibilities:

1. There is no error response and no user input is reflected:
   The scan flow is terminated because there appears to be no reflected or error-based template injection.

2. There is no error response and user input is reflected:
   The scanner now knows that errors are being caught and will take this into account when evaluating all subsequent polyglot responses. Next, requests with the three universal error-free polyglots are sent and their responses are evaluated.

3. There is an error response and no user input is reflected:
   The three universal non-error polyglots are skipped because the user input is not being rendered. Instead, the scanner sends error polyglots.

4. There is an error response and user input is reflected:
   Requests with the three universal non-error polyglots are sent and their responses are evaluated in order to check if at least one of them is being rendered modified.

If the polyglots sent so far have only generated responses indicating that there is no reflected or error-based template injection, the scan flow is terminated.

Otherwise, depending on the case, reflected or error-based template injection seems to be present and TInjA checks whether a template engine can already be uniquely identified. This is done by evaluating the response to each polyglot and checking which template engines it applies to. In addition, TInjA checks before a polyglot is sent whether it contributes at all to the exclusion of further template engines. This ensures that with each additional polyglot sent, the possible template engines can be further restricted. At this point there is a loop with again four possibilities:

1. All supported template engines have been excluded:
   This means that an unsupported template engine is used. The scanner returns

that a template injection was found for an "unknown" template engine, and the scan process is terminated.

2. A template engine was uniquely identified:
   The scanner reports that a template injection exists for the identified template engine, and the scan process is terminated.

3. More than one template engine is possible, but there is no polyglot left to exclude another one:
   The scanner returns both that a template injection has been detected and the list of possible template engines. The scan process is terminated.

4. More than one template engine is still possible, and there are still polyglots left to exclude another one:
   The next polyglot is sent and the response is evaluated.

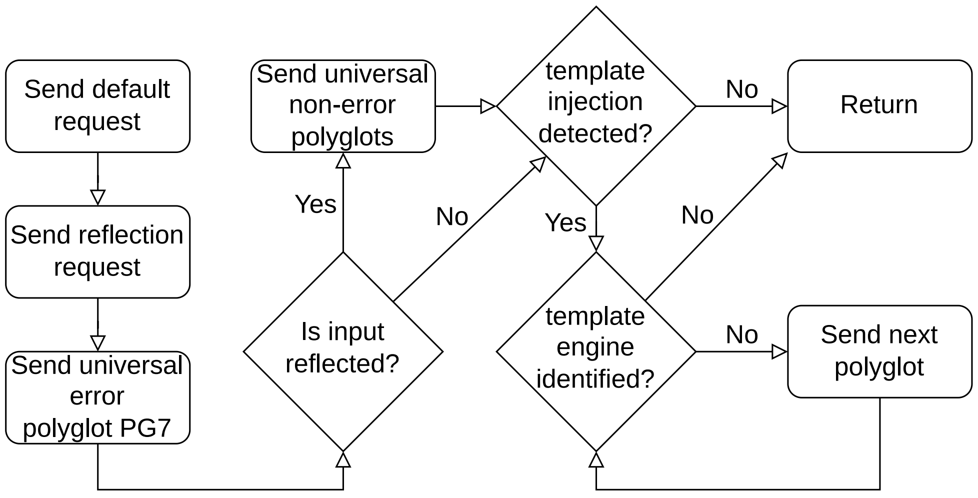

Send default request
Send reflection request
Send universal error polyglot PG7
Is input reflected?
Yes
No
Send universal non-error polyglots
template injection detected?
No
Return
Yes
template engine identified?
No
No
Send next polyglot

Figure 5.2: TInjA's scan flow summarized.

5.4 Template Injection Detection and Template Engine Identification

Tinja needs only four polyglots to detect template injection. The first is the universal error polyglot PG7, which triggers an error in all test cases of the template injection playground. In addition, three universal non-error polyglots are used which ensure that the polyglot is rendered modified at least once in each of the test cases. For template engine identification, all polyglots examined in Chapter 4 are used. For each supported template engine, the scanner has a list of expected responses for the polyglots. The lists of expected polyglot responses are further compared to the received polyglot responses to exclude template engines that would respond differently than expected. For each of the template engines examined, a unique identification is possible using the polyglots. On average, TInjA requires 4.75 polyglots to detect a template injection and identify the template engine when error messages are not caught. When error messages are caught, the average number of polyglots required increases by only one to 5.75.

In detecting rendered template injections and identifying template engines, TInjA differs from scanners that do not rely on polyglots. For example, these scanners send template expressions with mathematical calculations with long results for many different template engine syntaxes. The advantage of this approach is that it makes it very easy to detect template injections. This is because the entire response can be searched for the long result of the mathematical calculation, since it is very unlikely that this number would occur there by chance. However, this has a distinct disadvantage: Due to the different syntaxes of template engines, many more requests are needed to cover a large number of template engines, as with polyglots.

Unfortunately, it is not possible with polyglots to search the entire response for all possible expected template engine responses. This would lead to too many false positives. Further, the non-error polyglots need to use as few features as possible in order to avoid triggering errors. For example, when using polyglots with mathematical calculations, the syntax may vary too much from template engine to template engine, leading to more error messages. Also, polyglots should be as short as possible, so that a web page with a limit on the length of user input causes fewer problems. For these reasons, the template engines often render the polyglots only as a single-digit number, for example. Searching for a complete answer will most likely result in false positives.

Therefore, before sending polyglots and evaluating their responses, TInjA sends a reflection request. This contains a long random string that can be searched for in the response. TInjA then stores the characters before and after this string for all places where it is reflected. When parsing a polyglot response, it can search for the characters it stored before. Everything in between these characters is the rendered polyglot. Figure 5.3 shows the response to a reflection request with the random string `2N3A0T7A2L0I1E7`. If the response to a polyglot request is then received as in Figure 5.4, the characters preceding and following the random string can be used to determine that the template engine rendered the polyglot as `1`.

Furthermore, TInjA tries to keep the rate of false positives as low as possible for supposed error-based template injections where the input is not reflected. For this purpose, as soon as a polyglot triggers an error message, the same polyglot is sent again, but with a backslash (\) in front of each character. This invalidates the polyglot and it is no longer recognized as a template expression by template engines. If the “backslashed” polyglot also throws an error message, the scanner concludes

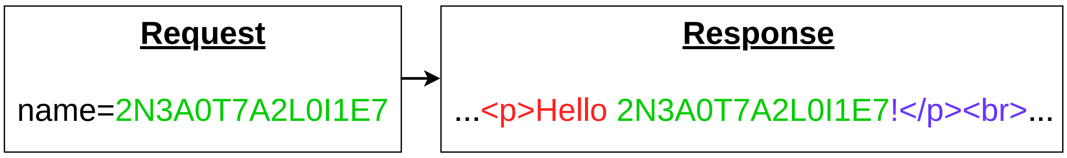


Figure 5.3: A random string is used as user input in order to determine where it is reflected.

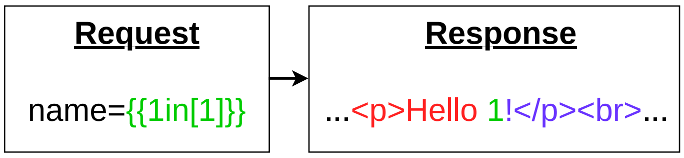


Figure 5.4: The saved preceding and subsequent strings are used to determine the rendered polyglot.

that the error message was not generated by a template engine, but occurs for other reasons. One possible reason might be that the website or web application firewall is blocking the use of certain characters. On the other hand, if the backslashed polyglot does not throw an error message, the scanner concludes that a template engine is indeed processing the user input.

TInjA reports a suspected template injection if at least one polyglot was rendered modified or caused an error response. Depending on how likely it is that a template injection has been found, TInjA will report one of three possible levels of certainty:

Low Every supported template engine has been excluded. If the polyglot is encoded only once, either in URL or HTML, it is not counted as modified rendered. This avoids a lot of false positives, since many web pages encode reflected user input, as discovered in tests for the large-scale scan. However, there is still a high risk of false positives. For example, if a web page or web application firewall removes some special characters or entire template expressions, uses an encoding other than URL or HTML, or encodes multiple times. Another source of false positives is cases where user input is not rendered. In the case of error-based template injections, the probability that a polyglot request will return an error response for other reasons is not negligible.

Medium There are two options here:

1. Every supported template engine has been excluded, but at least one response expected from one of the supported template engines has been rendered. If one of the expected responses is rendered, the probability of a false positive is reduced. On the one hand, this is because it is unlikely that a web page will modify the polyglot exactly as one (of the supported)

template engines does. On the other hand, this eliminates the possibility that the template injection was only detected by error responses.

2. Not all template engines have been excluded, but none of the expected responses have been rendered. If no user input is reflected, and thus only error-based template injection can be detected, TInjA cannot uniquely determine the template engine in all cases. Nevertheless, the set of possible template engines can be narrowed down. Furthermore, there is still the problem that a polyglot request may return an error response for other reasons.

High Not all template engines were excluded, and at least one of the expected responses was rendered. Thus, the user input is reflected and the three universal error-free polyglots, as well as any other polyglots sent, are responded to as expected. If one or more responses to polyglot requests were not received, for example, due to a timeout, it is possible that not all template engines could be excluded, and thus several are still possible.

5.5 Setup and Usage

Setup TInjA is released in its own Github repository[2]. The repository will be made available to the public shortly after the thesis is published. Since the scanner is developed in Go, there are two different recommended setup options.

- Standalone binaries for different operating systems are available on the release page of the repository. These can be downloaded and run immediately without any installation or dependencies. Among others, binaries for windows/amd64, linux/amd64 and darwin/amd64 are available.
- If Golang is installed, the scanner can be installed via Golang. The command `go install -v github.com/Hackmanit/TInjA` instructs Golang to compile a standalone binary of TInjA and place it in `$GOPATH/bin`.

The advantage of the first method is that no Golang or other dependencies are needed. However, this is only possible if a binary is available for the used operating system. The second method has the advantage that it works independent of the operating system, since it instructs Golang to compile a standalone binary for the operating system.

It is also possible to download the repository and compile a standalone binary by oneself. This allows to compile binaries for different computer architectures supported by Golang[2].

[2]`https://github.com/Hackmanit/TInjA`

Usage The scanner needs at least one URL, the path to a list of URLs, or the path to a JSONL file, which can contain additional information like the headers to use or a request body. To test the URL "example.com" with default settings, the scanner can be started like this `tinja url -u http://example.com`. For additional testing of the URL "example.com/url2", `-u http://example.com/url2` can be appended. Similarly, |tinja url -u file:/path/to/file| can be used to specify the path to a file containing a URL in each line. Since a page may be crawled for URLs first, especially for larger fully automated scans, a JSONL file can also be imported with `tinja jsonl -j /path/to/file`. The file is then searched line by line for a JSON object named `request`, which can contain the string keys `method`, `endpoint`, and `body`, as well as the array `headers`. This corresponds to the JSON output of the Katana[3] crawler, which was also used for the large-scale scan described in Chapter 7. Using such a file allows to specify different headers, HTTP methods and request bodies for each URL (or endpoint).

[3]`https://github.com/projectdiscovery/katana`

6 Template Injection Scanner Comparison

This chapter deals with the comparison of *TInjA* and four other template injection scanners. First, we introduce all the contestants and compare several features and characteristics. Next, we compare benchmarks determined using the *Template Injection Playground*, such as detection and identification rate, number of requests sent, or RAM usage. Another playground called *websitesVulnerableToSSTI*[1] is also used to compare the scanners. As a final practical comparison, we examine which of the scanners are able to detect and identify an SSTI vulnerability in the intentionally vulnerable web application *Juice Shop*[2]. Finally, we draw a conclusion from the comparison and address identified opportunities to improve TInjA.

6.1 Overview of the Contestants

A total of six scanners were found, which either have been specifically developed for template injections or are capable of detecting template injections in addition to other vulnerabilities. One of these scanners, tplmap[3], has not been maintained for a long time and during this master's thesis it was not possible to successfully run it. Therefore it was not considered further. The five scanners considered for the comparison are:

TInjA This free command-line template injection scanner was developed during this master's thesis. TInjA was tested in version 1.0.5.

SSTImap A free command line template injection scanner based on tplmap. SSTImap has been tested in version 1.1.4[4].

gossti A free command line template injection scanner. gossti was tested in version 1.0.0[5]. Unfortunately, the tested version, which was the only version available at the time of writing, does not seem to work correctly. For example, gossti does not replace the values of the POST parameters for POST requests. Also,

[1]`https://github.com/DiogoMRSilva/websitesVulnerableToSSTI`
[2]`https://owasp.org/www-project-juice-shop/`
[3]`https://github.com/epinna/tplmap`
[4]`https://github.com/vladko312/SSTImap/tree/1619ed2161dc43f3676aef43b3a8ddfdd16430d7`
[5]`https://github.com/LeoFVO/gossti/releases/tag/v1.0.0`

it only replaces the values of the query parameters with the payloads for the Tornado template engine, not with those of the other supported template engines. As a result, gossti has not been considered for the practical tests.

ZAProxy A free, full-featured web application security toolkit that includes a scanner and many other features. This scanner can detect template injections and many other vulnerabilities. In addition to the GUI, ZAProxy can also be used from the command line. ZAProxy has been tested in version 2.13.0[6] with the active scanner rules v56. The template injection plugins for the ZAProxy scanner were developed in the thesis of Miguel Reis Silva[8].

BurpSuite Professional A full-featured web application security toolkit that includes many features and a scanner. This scanner can detect template injections and many other vulnerabilities. There is a free community edition as well as a paid professional edition, but only the paid edition contains the scanner. The cost for the professional edition is currently 449€ per year. Besides the GUI, BurpSuite can also be used via a GraphQL API. BurpSuite professional was tested with version 2023.7.2[7].

Table 6.1 compares some features and characteristics of the scanners. Besides the already mentioned features like program type, usage, and price, some other features are compared. All scanners can detect SSTI, but only TInjA and BurpSuite can detect CSTI. “Reflection Other Site” describes a scenario where a request is sent to a web page, but the reflection of the user input can be seen on another web page. An example would be a registration form where the email address entered during registration is only visible on the profile page later. Only SSTImap and gossti cannot detect a template injection in such a scenario. TInjA allows specifying an additional URL that will be scanned for reflections. ZAProxy provides a plugin with a scripting language called zest that can be used to send a request to a different URL and replace the response with the actual response. BurpSuite provides macros that, similar to ZAProxy, can be used to set the response to be replaced with the response from another URL. All scanners except SSTImap and gossti can generate a report summarizing the results. ZAProxy and BurpSuite use their integrated crawlers. TInjA does not have an integrated crawler, but allows importing the output of a crawler in JSONL format, as generated by the *katana*[8] crawler. Rate limiting is provided by TInjA, ZAProxy, and BurpSuite, but for BurpSuite an additional plugin is required.

[6]`https://www.zaproxy.org/docs/desktop/releases/2.12.0/`

[7]`https://portswigger.net/burp/releases/professional-community-2023-7-2?requestededition=professional&requestedplatform=`

[8]`https://github.com/projectdiscovery/katana`

Table 6.1: Overview of the competing scanners and their features.

Scanner	Usage	Templ. Inj.	Tactic[a]	Refl. URL	Report	Crawler	Ratelimit.
TInjA	CLI	SSTI+CSTI	Pol.	yes (flag)	yes	import	yes
SSTImap	CLI	SSTI	Spec.	-	-	-	-
gossti	CLI	SSTI	Spec.	-	-	-	-
ZAProxy	GUI+CLI	SSTI	Pol.+Blind	yes (zest)	yes	yes	yes
BurpSuite Pro	GUI+API	SSTI+CSTI	Spec.	yes (macro)	yes	yes	yes (plugin)

[a]Pol = Polyglots. Spec = template expression—such as mathematical calculations—tailored to a specific TE. Blind = Specific template expressions for blind scenarios—such as RCE with a sleep statement—tailored to a specific TE.

6.2 Template Injection Test Approach

To compare how good the scanners are at detecting template injections, the Template Injection Playground which was used to develop TInjA was used. Here, 50 of the 51 test cases were tested with each scanner in two separate runs. One test case, for the *Vue.js* template engine, was not included because the Vue.js implementation can crash the JavaScript web server in case of an error. In the first test run, the playground was used with its default settings. In the second test run, error messages were intercepted and in case of an error message, the user input was displayed unchanged instead. During the test runs, the CPU and RAM usage caused by the scanners was monitored. Furthermore, the time needed for the scan to finish was measured and the requests sent to the playground were counted by the playground itself. Both BurpSuite and ZAProxy scan a large number of vulnerabilities by default. To make their results comparable, a custom scan configuration was created for both scanners. For ZAProxy, all tests except "Server Side Template Injection" and "Server Side Template Injection (blind)" were disabled. In addition, the strength was set to "Insane" for both test cases. The strength of a test is the maximum number of requests that are sent by ZAProxy. At "Insane" the number of requests is not limited. For BurpSuite, all tests were disabled except for "server-side template injection" and "client-side template injection".

TInjA was developed and tested on the Template Injection Playground. Therefore, it has a non-negligible advantage over the other scanners when compared using this playground. To mitigate this the scanners were also tested using another playground. This other playground is called *websitesVulnerableToSSTI*[9] and it is the only other Template Injection Playground that could be found. It was also used to develop the template injection scanner plugins for ZAProxy. The playground offers 18 different template engines with a total of 20 test cases. In contrast to the Template Injection Playground, the template engines Dust and Jade (an older version of Pug) are included. Furthermore, some of the template engines are available in a

[9]`https://github.com/DiogoMRSilva/websitesVulnerableToSSTI`

different version or configured differently. In addition, the web servers used or their configuration is different. Due to these differences, the test results may differ from those of the Template Injection Playground.

In addition to the two playground tests, each scanner was also challenged to find the SSTI vulnerability in the *OWASP Juice Shop*[10]. The OWASP Juice Shop is an intentionally vulnerable but realistic web application that has all sorts of vulnerabilities. Among other things, it has an SSTI vulnerability. In the paper "Evaluation of Black-Box Web Application Security Scanners in Detecting Injection Vulnerabilities", five different vulnerability scanners—including ZAProxy and BurpSuite—were tested to see if any of them could detect this SSTI[1]. The result was that none of the scanners were able to detect it. The current test with BurpSuite and ZAProxy will show whether they have improved in this regard.

6.3 Template Injection Test Results

A total of four different practical tests were performed with the four scanners TInja, ZAProxy, SSTImap, and BurpSuite. gossti was not included in the practical test because the only version available at this time does not work properly, as already mentioned.

6.3.1 Template Injection Playground

The results of the first test run of the Template Injection Playground are shown in Table 6.2. The playground was configured with its default settings. This means that the user input was inserted into a template without any changes and was rendered by the template engine. Error messages were not caught, but rendered as well. TInjA detected template injection in all test cases and correctly identified all template engines. ZAProxy was able to detect the template injection in 70% of the test cases. However, ZAProxy does not try to determine which template engine is used. SSTImap detects the template injection in only one test case less than ZAProxy. In almost 53% of the detected template injections, SSTImap identifies the correct template engine. However, in the remaining 47% of cases, the wrong template engine is specified. BurpSuite detects the template injection possibility in 62% of the test cases, not far behind ZAProxy and SSTImap. BurpSuite identifies the correct template engine in 39% of detected template injections and a wrong template engine in another 39%. In the remaining cases, BurpSuite does not specify a template engine at all.

[10]`https://owasp.org/www-project-juice-shop/`

Table 6.2: Evaluation results on the Template Injection Playground.

Scanner	Detected	Correct TE	Wrong TE	Requests	Time	CPU	RAM
TInjA	**50**	**50**	**-**	**698**	**00:22**	**0.5%**	50.4 MiB
SSTImap	34	18	16	12,002	05:27	2%	**24.7 MiB**
ZAProxy	35	n/a	n/a	3,554	00:57	13%	1.1 GiB
BurpSuite Pro	31	12	12	7,116	03:35	15%	1.5 GiB

Since one of the features of the Template Injection Playground is that it counts the number of requests, the number of requests sent by each scanner can be easily gathered. TInjA clearly sends the least number of requests with 698 requests. ZAProxy, which also uses polyglots, requires the second fewest requests with 3554. BurpSuite sends 7116 requests, while SSTImap sends by far the most requests with 12,002. The duration of the scans were also measured. TInjA was the fastest with 22 seconds, followed by ZAProxy with 57 seconds. BurpSuite takes much longer with 3 minutes and 35 seconds and SSTImap takes even longer with 5 minutes and 27 seconds.

Furthermore, the maximum CPU and RAM usage of the scanners was monitored using the task manager of the Manjaro test system. The CPU usage is of course highly dependent on the CPU in use, but there is still a clear difference between the measured results. TInjA causes a CPU usage of only 0.5%, while SSTImap causes the second lowest CPU usage with 2%. ZAProxy with 13% and BurpSuite with 15% cause much higher utilization as full featured toolkits. RAM usage shows similar results. SSTImap uses the least RAM with 24.7MiB. TInja uses slightly more than twice as much with 50.4MiB, but more than half of it (29.4MiB) is used by the headless browser needed to detect CSTI. ZAProxy uses 1.1 GiB of RAM and BurpSuite uses the most with 1.5GiB.

The results of the second test run of the Template Injection Playground are shown in Table 6.3. In the second run, the Template Injection Playground option to catch error messages and instead reflect the user input unchanged was enabled. The results are mostly the same as in the first run. Only for TInjA the number of requests decreased. This is due to the fact that TInjA sends up to two more requests in case of an error message to check whether the error message was caused by the polyglot or by another reason. Accordingly, the number of requests in the second run is lower. The time differences compared to the first run are minimal and probably due to normal fluctuations. There was no measurable difference in the maximum CPU and RAM usage.

Table 6.3: Evaluation results on the Template Injection Playground with errors being caught.

Scanner	Detected	Correct TE	Wrong TE	Requests	Time	CPU	RAM
TInjA	**50**	**50**	**-**	**652**	**00:20**	**0.5%**	50.4 MiB
SSTImap	34	18	16	12,002	05:17	2%	**24.7 MiB**
ZAProxy	35	n/a	n/a	3,552	01:09	13%	1.1 GiB
BurpSuite Pro	31	12	12	7,215	03:06	15%	1.5 GiB

6.3.2 *websitesVulnerableToSSTI*

The results of the test run on the websitesVulnerableToSSTI playground are shown in Table 6.4. TInjA and ZAProxy are the only scanners that detected the template injection in all 20 test cases. BurpSuite was able to do so in 17 of the test cases and SSTImap in only 14. TInjA was able to correctly identify the template engine for 13 of the 20 template injections detected in this playground. In the test case where the *Mako* template engine is used and the user input is filtered, a wrong template engine was identified caused by the filtering. In the other six cases, TInjA reported that the template engine used was unknown. In the case of the *Dust* template engine, this is indeed true. Dust was not taken into account when generating the polyglots, since it was not considered relevant (anymore). In the other test cases, where TInjA could not identify the template engine, this is either because the template engine versions used are significantly older than in the Template Injection Playground, or the implementation is different. In both cases, the responses of individual polyglots may be different from those expected by TInjA.

SSTImap, like TInjA, correctly identifies 13 template engines and specifies an incorrect template engine in one test case. BurpSuite identifies the correct template engine for only nine of the detected template injections. For five detected template injections BurpSuite does not specify a template engine and for three others it specifies an incorrect template engine. TInjA is again by far the fastest in scanning this playground with only two seconds. ZAProxy is once again in second place with 26 seconds. SSTIMap is no longer the slowest by far and is close behind ZAProxy with 36 seconds. BurpSuite takes the longest time with 49 seconds. The CPU and RAM usage of the scanners are largely in line with the Template Injection Playground measurements. The biggest difference can be seen in TInjA's RAM usage. This is only 14.6 MiB for WebsitesVulnerableToSSTI. For the Template Injection Playground it is much higher at 50.4 MiB. This is mainly due to the fact that websitesVulnerableToSSTI does not embed script tags in its web pages and therefore TInjA does not run the headless browser. The number of requests has not been counted in this playground, as this is only a feature of the Template Injection Playground.

Table 6.4: Evaluation results on the websitesVulnerableToSSTI playground.

Scanner	Detected	Correct TE	Wrong TE	Time	CPU	RAM
TInjA	**20**	**13**	**1**	**00:02**	**0.5%**	**14.6 MiB**
SSTImap	14	**13**	**1**	00:36	1.5%	18.5 MiB
ZAProxy	**20**	n/a	n/a	00:26	12.5%	1.2 GiB
BurpSuite Pro	17	9	3	00:49	14%	1.5 GiB

6.3.3 Juice Shop

The Juice Shop contains an SSTI vulnerability where the configurable username is inserted into a template without further validation. This template is rendered by the server-side *Pug* template engine and returned as a response to the request used to change the username. The results of scanning Juice Shop with the four scanners are presented in Table 6.5. TInjA detects the template injection and is the only one of the tested scanners to identify the correct template engine. SSTImap and BurpSuite also detect the template injection, but both report an incorrect template engine. Thus, BurpSuite's SSTI detection seems to have improved since the test in "Evaluation of Black-Box Web Application Security Scanners in Detecting Injection Vulnerabilities"[1], as BurpSuite did not find the SSTI at that time. However, ZAProxy still does not detect this specific template injection vulnerability. SSTImap and, of course, TInjA were not part of the 2022 evaluation, as only scanners covering a wider range of injection vulnerabilities were evaluated.

Table 6.5: Evaluation results for the Juice Shop SSTI vulnerability.

Scanner	This Evaluation	Evaluation 2022[1]
TInjA	**Detected + Correct TE**	n/a
SSTImap	Detected + Wrong TE	n/a
ZAProxy	Not detected	Not detected
BurpSuite Pro	Detected + Wrong TE	Not detected

6.4 Conclusion

It is not surprising that the scanners focused on template injection—TInjA and SSTImap—cause a significantly lower CPU and RAM load than the full-featured web application security toolkits ZAProxy and BurpSuite. However, practical tests also show that TInjA requires significantly fewer requests than the other scanners due to the newly designed polyglots. This also results in TInjA being the fastest for all scans, in addition to being single threaded. Furthermore, TInjA clearly leads in the number of detected template injections and correctly identified template engines. Nevertheless, there is room for improvement for TInjA, as the evaluation of the test run for the websitesVulnerableToSSTI playground showed. For example, differences in the implementation of the template engine or large differences in the versions caused the answers of a template engine to be different than expected for some polyglots. An example of this is the Django template engine and the universal no-error polyglot `p ">[[${{1}}]]`. In the Template Injection Playground, the error `AppRegistryNotReady("Apps aren't loaded yet.")` is thrown because no settings file, which contains information about the apps to use, was loaded beforehand. In the websitesVulnerableToSSTI playground, however, such a settings file is loaded, so no error is thrown and `p ">[[$1]]` is rendered. Since TInjA contains the polyglot responses from the Template Injection Playground, it expects an error message in response to the polyglot, provided no error messages are caught. Therefore, Django is excluded as a possible template engine after the polyglot response is evaluated. This difference also occurs with the PHP-specific non-error polyglot `{{7}}}` and the Dotnet-specific non-error polyglot `{{1}}@*`. All other polyglots are rendered the same by Django in both playgrounds. Such differences that may occur with some polyglots need to be considered and, if discovered, incorporated into future enhancements to TInjA and also the interactive template injection table.

7 Large-Scale Template Injection Scan

This chapter describes the large-scale scan that was performed during this thesis and the insights gained from both the process and the results of the scan. First, we discuss how the domains to be scanned were selected. Then we outline the approach of the scan, with a focus on what tools were used and how they were used. Next, we address some problems that occurred during the scan and made it necessary to restart the scan several times. After that, we discuss various statistics, such as how many URLs were tested and what the results of the scan were. Subsequently, we describe improvements to TInjA that can be implemented to avoid many of the false positives. Finally, we draw a conclusion from the large-scale scan.

7.1 Selection of Domains

The top 1000 apex domains on the Tranco list generated on 11 July 2023[1] were selected as the target of the large-scale scan for template injections. The Tranco list is a ranking of the world's most trafficked apex domains and uses several methods to prevent statistical manipulation [7]. To ensure that the scan is legal and ethical, only domains that explicitly allow scanning with automated tools were tested. To filter out all other domains, four of the largest bug bounty platforms were searched for the top 1000 domains. Among other things, bug bounty platforms allow companies to expose their domains and websites for testing under certain conditions. If someone finds a vulnerability, they can receive a "bug bounty," which can include monetary rewards, merchandise, or virtual recognition, such as points on the platform or entry into a hall of fame. Some of the companies with one of the top 1000 domains have their own bug bounty programs, such as Google and Facebook. However, only bug bounty programs on the four selected platforms were considered. This was mainly due to time constraints, as these platforms present the conditions in a unified and clear manner, making it possible to quickly find out whether a domain is eligible or not. In addition, the effort required to check whether a domain appears in a bug bounty program on a bug bounty platform is significantly less than the effort required to find out whether a domain generally is part of an individual bug bounty

[1]`https://tranco-list.eu/list/W9Z99/1000`

program and where to find it. The four bug bounty platforms used are Hackerone[2], Bugcrowd[3], Intigriti[4], and YesWeHack[5].

In total, 80 of the top 1000 apex domains were part of a bug bounty program on one of the four bug bounty platforms. Of these 80, 72 allowed automated scanning without restriction or under certain conditions, while eight explicitly prohibited it. The specific restrictions for automated scanning were as follows:

Rate limiting Some bug bounty programs required rate limiting, meaning that a maximum number of requests were allowed to be sent per second or minute. The rate limits specified for various domains range from one to five requests per second.

Custom header or user agent Some bug bounty programs require a specific user agent or header to be set. This header should include, for example, the name of the bug bounty platform or one's username on the bug bounty platform.

Some programs only allowed one or a few subdomains of the apex domain to be tested, while most allowed any subdomain.

7.2 Scan Approach

First, the subdomains of the 72 apex domains in question were enumerated, provided that each subdomain could be crawled. Next, these domains were crawled and the crawl results were passed to TInjA. Finally, the results of the TInjA report were evaluated. This approach is described in greater detail below:

Subdomain enumeration For the apex domains where scanning of arbitrary domains was allowed, Subfinder[6] was used to enumerate subdomains. Subfinder is a passive subdomain enumeration tool that uses various sources to find subdomains of a given domain. Compared to an active subdomain enumeration, where for example subdomains are tried by a brute force approach, the passive enumeration has the advantage that it is much faster. This is due to the fact that existing databases are queried for subdomains of a given domain. Besides the standard sources like waybackarchive and crtsh, shodan was included by adding a shodan API key to the subfinder configuration file. With `subfinder -d example.com` the subdomain enumeration was started for each apex domain in question.

Next, the enumerated subdomains were filtered. Subdomains that did not respond at all or had a status code other than 200, 301 or 302 were discarded.

[2]`https://www.hackerone.com/`
[3]`https://www.bugcrowd.com/`
[4]`https://www.intigriti.com/`
[5]`https://www.yeswehack.com/`
[6]`https://github.com/projectdiscovery/subfinder`

Thus, subdomains that are no longer used or that only display an error page, for example, due to missing authorization, are discarded. To do this, the httpx[7] tool was used with the following command: `httpx -list /list/of/subdomains -silent -mc 200,301,302 -rl 1 -fr -o /list/of/subdomains-active`
This creates a filtered list of subdomains, including the destination of a redirect. Next, it is useful to filter out redirects that redirect to the same subdomain to avoid duplicates, or redirects that redirect to a different apex domain to avoid leaving the allowed scope. To automate this filtering, a Python script was written (see Appendix B).

Crawling The katana[8] crawler was used to crawl the subdomains. A separate crawler was started for each apex domain. The crawler was set to crawl a maximum of 250 URLs per domain, to automatically fill web page forms, and to save the results to a file in JSONL format. In addition, the crawler was instructed to maintain a rate limit of one request per second to avoid being blocked by a web application firewall, for example, and to not leave the specified domain to stay in scope. Furthermore, a breadth-first search with a maximum depth of five was chosen. The command used is: `katana -list /list/of/subdomains-active-filtered -automatic-form-fill -jsonl -omit-raw -omit-body -strategy breadth-first -field-scope fqdn -rate-limit 1 -concurrency 1 -parallelism 1 -depth 5 -ct 250 -o /list/of/subdomains-active-filtered-crawled`

If a bug bounty program specified that a particular user-agent or other header should be set, this was added to the command. In addition to the URLs, the crawl results include other information such as the HTTP methods used and the request body. This means that not only GET, but also POST requests can be recreated.

Scanning TInjA allows to import crawl results in JSONL format, the same way the crawler Katana stores them. An individual scanner was started for each file containing the crawl results of one of the apex domains. TInjA was started with the following command: `tinja jsonl --jsonl /list/of/subdomains-active-filtered-crawled --csti --reportpath /list/of/subdomains-active-filtered-crawled-scanned -r 1`
Thus, a general rate limit of one request per second was used, regardless of whether the bug bounty program requested one or not, to keep the risk of being blocked as low as possible. In addition to SSTI, CSTI was also scanned by enabling the use of a headless browser.

[7]`https://github.com/projectdiscovery/httpx`
[8]`https://github.com/projectdiscovery/katana`

7.3 Problems During the Scan

The scan needed to be restarted several times as the large number of diverse websites resulted in several edge cases that were not apparent during the playground tests. Most of the problems were due to use of the library *rod*[9]. This allows the use of a headless browser, which is needed to detect CSTI. For example, one problem was that code examples and documentation often use methods with a "must" prefix. These methods panic in the event of an error because they are meant for quick testing, not production use. This was not obvious during the implementation in TInjA, causing the scanner to panic a few times.

There were also some RAM usage improvements implemented because some URLs were causing very high RAM usage. In the last started scan each TInjA instance had a RAM usage of 30-60 MiB. However, depending on the URL, this could increase up to 250MiB in marginal cases. Though the CPU load remained minimal as expected.

7.4 Statistics and Results

A total of 160,319 domains were enumerated for the 72 eligible apex domains. After filtering these domains, 6,227 domains remained out of a total of 69 apex domains. Three apex domains blocked httpx requests, resulting in all of their domains being discarded. Based on the 6,227 domains, 81,937 URLs were crawled. Of these, 27,003 URLs had a total of 51,728 query parameters. Further, 1,882 URLs were from POST requests containing a total of 9,094 POST parameters.

The scan of the 81,937 URLs was completed within five days with some breaks in between. In total, TInjA reported 10,438 possible template injections, of which 854 were high certainty, 840 were medium certainty, and 8,744 were low certainty. For those with a high certainty, 804 times Velocity, 46 times VelocityJS, two times Pystache, and two times Thymeleaf (Inline) were reported as the identified template engines. The low confidence results were not considered further. This is because they are rather information that the user input is rendered in some way modified and therefore the probability of false positives is very high. The high and medium confidence findings were manually checked on a spot check basis, as the number of findings is very high for the given time frame. All checked results were classified as false positives.

[9] `https://github.com/go-rod/rod`

7.5 False Positives

Improvements for TInjA have been identified to eliminate all of the high certainty false positives and most of the medium certainty false positives. These improvements are:

Hybrid Approach Scanning with polyglots is very efficient, as shown in the scanner comparison. However, there is a significant drawback: It is much more prone to false positives than, for example, scanning with mathematical template expressions in a large number of different syntaxes. Therefore, a hybrid approach can combine the advantages of both approaches. As before, the polyglots are used to detect a possible template injection and to identify the template engine. In order to ensure that it is not a false positive, a template expression specifically adapted to the identified template engine is then sent. This template expression is one that has a very small chance of being a false positive, such as a mathematical calculation. This hybrid approach thus combines the efficiency of polyglots with the extremely low false positive probability of tailored template expressions, such as mathematical calculations.

Additional Checks Some polyglots are significantly more prone to false positives than others. This includes, for example, the universal non-error polyglot `{##}/*{{.}}*/`. This polyglot was responsible for almost all of the high certainty false positives and also a proportion of the medium certainty false positives. The reason for this is as follows: Some template engines, including Velocity and VelocityJS, use `##` to start a comment, resulting in only a `{` being rendered. However, during the large-scale scan, some sites removed one `#` and everything that followed it if there was a `#` in the user input. This resulted in only a `{` being rendered, too. To detect these cases, if only a `{` is rendered when the polyglot is sent, the polyglot can be sent again, but with only a single `#`. If again only a `{` is rendered, this is a false positive, as the template engines tested only interpret two `#` characters as comments.

7.6 Conclusion

Over a period of five days, 81,937 URLs from 69 apex domains on the Tranco Top 1000 list were scanned for template injections. 10,438 possible template injections were reported, of which 854 had a high, 840 a medium, and 8,744 a low certainty. However, no template injections could be confirmed during manual verification. There may be several reasons for this. For example, template injection vulnerabilities may be present in the URLs tested, but not identified by TInjA. Another possibility is that there are no template injection vulnerabilities at all, because no insecure user input is inserted into templates, or simply no template engines are used at all.

Nevertheless, the scanning of the many different domains has been very instructive. Bugs in TInjA that had not been found in the previous tests on the playgrounds and the juice shop were identified and could be fixed. In addition, improvements for TInjA have been identified that can eliminate all of the high confidence and most of the medium confidence false positives reported during the large-scale scan. One of these improvements is a new *hybrid approach*. This combines the efficiency of polyglots with the very low false positive probability of a template expression tailored to a specific template engine such as a mathematical calculation. The polyglots are used to detect possible template injection and to identify the template engine. Subsequently, a template expression tailored to the identified template engine is sent to verify the finding and eliminate the potential for a false positive.
Once these false positive mitigations are implemented, TInjA will be well prepared for a new large-scale scan.

8 Conclusion

In the context of this master thesis, an extensive template injection playground was created. In addition to various features that can be used to modify the behavior of the playground, such as several optional countermeasures, another unique feature is the large number of 46 implemented template engines. For the selection of the template engines, several sources were used to determine which template engines are currently the most relevant. Based on these most relevant template engines, “error” polyglots were developed that differ from previous polyglots in two main ways: They cover all template engines on the playground, and they are much shorter. However, error polyglots are not able to detect template injections when errors thrown by a template engine are caught. Therefore, a new type of template injection polyglots has been created, the “non-error” polyglots. These are based on the idea that as many template engines as possible render the polyglot modified, for example, by removing parts of it. Three non-error polyglots were created, covering all template engines of the playground. In addition, language-specific polyglots have been created, both of the type error and non-error. These often have the advantage of being much shorter and can be used if the programming language of the web application is known, for example, by the framework used. In order to be able to use these polyglots in a meaningful way, a web page with an interactive template injection table has been created. It contains all the polyglots that have been examined and newly created, including all the responses from the examined template engines. When going through the template injection methodology, this table can be used to quickly and efficiently detect template injection possibilities and identify the specific template engines using the polyglots.

Furthermore, the novel polyglots have been used to program the template injection scanner TInjA. TInjA is the only scanner we know of that uses polyglots to both detect template injection possibilities and identify template engines. In various tests, it has proven to be significantly more efficient than the other template injection scanners tested. TInjA has by far the best detection and identification rate on two different playgrounds, and is also the only scanner that correctly identifies the template engine used in the SSTI vulnerability of the realistic but intentionally vulnerable web application *Juice Shop*. In addition, TInjA requires significantly fewer requests—at least five times fewer than the second place and at least 17 times fewer than the last place—making its scanning significantly faster. The Template Injection Playground, the Template Injection Table, and the Template Injection Scanner TInjA are made available to the public. This makes them available for future research as well as for cybersecurity enthusiasts and professionals who want to learn about or detect template injection possibilities.

Finally, a large-scale scan was performed on 81,937 URLs from 69 apex domains on the Tranco Top 1000 list. Due to the high number of 854 high certainty findings and 840 medium certainty findings, these were only manually verified on a sample basis. All spot checks were evaluated as false positives and therefore no template injection vulnerability was verified. Nevertheless, the scan was very instructive as some edge case bugs were identified and fixed. Furthermore, countermeasures for the false positives were found. One of them is a "hybrid approach". This combines the efficiency of polyglots with the very low false positive probability of a template expression tailored to a template engine such as a mathematical calculation. The polyglots are used to detect possible template injection vulnerabilities and to identify the template engine. Subsequently, a template expression tailored to the template engine is sent to verify the finding and eliminate the potential for a false positive.

Glossary

CSRF A Cross-Site Request Forgery is an attack that tricks an authenticated user into sending a malicious request. 16, 29, 71

CSRF token A Cross-Site Request Forgery token is a secret value generated by a web server and embedded in a response. The web server then verifies that the following request contains the correct secret value. This can be used to mitigate CSRF attacks. 13, 16, 28, 29, 31

CSTI Client-side template injection is a web application vulnerability which enables an attacker to inject template expressions into a template rendered by a client-side template engine. 6, 14, 20, 56, 59, 65, 66

JSONL JSON Lines—also known as newline-delimited JSON—is a data format where each line contains a JSON object. It is commonly used for log files or other large files. 45, 46, 53

RCE Remote code execution is an application vulnerability which enables an attacker to execute code on the vulnerable system. 6, 7, 10, 24, 57

SQLi SQL injection is an application vulnerability which enables an attacker to alter an SQL statement; eventually giving the attacker unauthorized access to the database. 11

SSTI Server-side template injection is a web application vulnerability which enables an attacker to inject template expressions into a template rendered by a server-side template engine. 2–4, 6, 10, 12–15, 20, 24, 55, 56, 58, 61, 65, 69

TE A template engine dynamically interprets a template containing template language at runtime and generates an output format, such as HTML. 14–16, 19, 20, 22–29, 57, 59–61

XSS Cross-Site-Scripting is a web application vulnerability which tries to execute an attacker script within the context of the website owner within the user's browser. 6, 8, 11, 28

List of Figures

List of Tables

Bibliography

[1] Muzun Althunayyan, Neetesh Saxena, Shancang Li, and Prosanta Gope. "Evaluation of Black-Box Web Application Security Scanners in Detecting Injection Vulnerabilities". In: *Electronics 2022, Vol. 11, Page 2049* 11.13 (June 2022), p. 2049. ISSN: 2079-9292. DOI: `10.3390/ELECTRONICS11132049`. URL: `https://www.mdpi.com/2079-9292/11/13/2049/htm%20https://www.mdpi.com/2079-9292/11/13/2049`.

[2] *Building Go Applications for Different Operating Systems and Architectures | DigitalOcean.* URL: `https://www.digitalocean.com/community/tutorials/building-go-applications-for-different-operating-systems-and-architectures`.

[3] Mario Heiderich. *mustache-security: A wiki dedicated to JavaScript MVC security pitfalls.* 2013. URL: `https://code.google.com/archive/p/mustache-security/`.

[4] Gareth Heyes. *DOM based AngularJS sandbox escapes | PortSwigger Research.* 2017. URL: `https://portswigger.net/research/dom-based-angularjs-sandbox-escapes`.

[5] Gareth Heyes. *XSS without HTML: Client-Side Template Injection with AngularJS | PortSwigger Research.* 2016. URL: `https://portswigger.net/research/xss-without-html-client-side-template-injection-with-angularjs`.

[6] James Kettle. "Server-Side Template Injection: RCE for the modern webapp". In: *Black Hat USA* (2015).

[7] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczy, Wouter Joosen, and Ku Leuven. "TRANCO: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: (). DOI: `10.14722/ndss.2019.23386`. URL: `https://dx.doi.org/10.14722/ndss.2019.23386`.

[8] Diogo Miguel Reis Silva. "ZAP-ESUP: ZAP Efficient Scanner for Server Side Template Injection Using Polyglots". PhD thesis. 2018.

[9] Oleksandr Mirosh. "Room for Escape: Scribbling Outside the Lines of Template Security". In: *Black Hat USA* (2020).

[10] *Mustache.php Release v2.14.1 — Security release.* URL: `https://github.com/bobthecow/mustache.php/releases/tag/v2.14.1`.

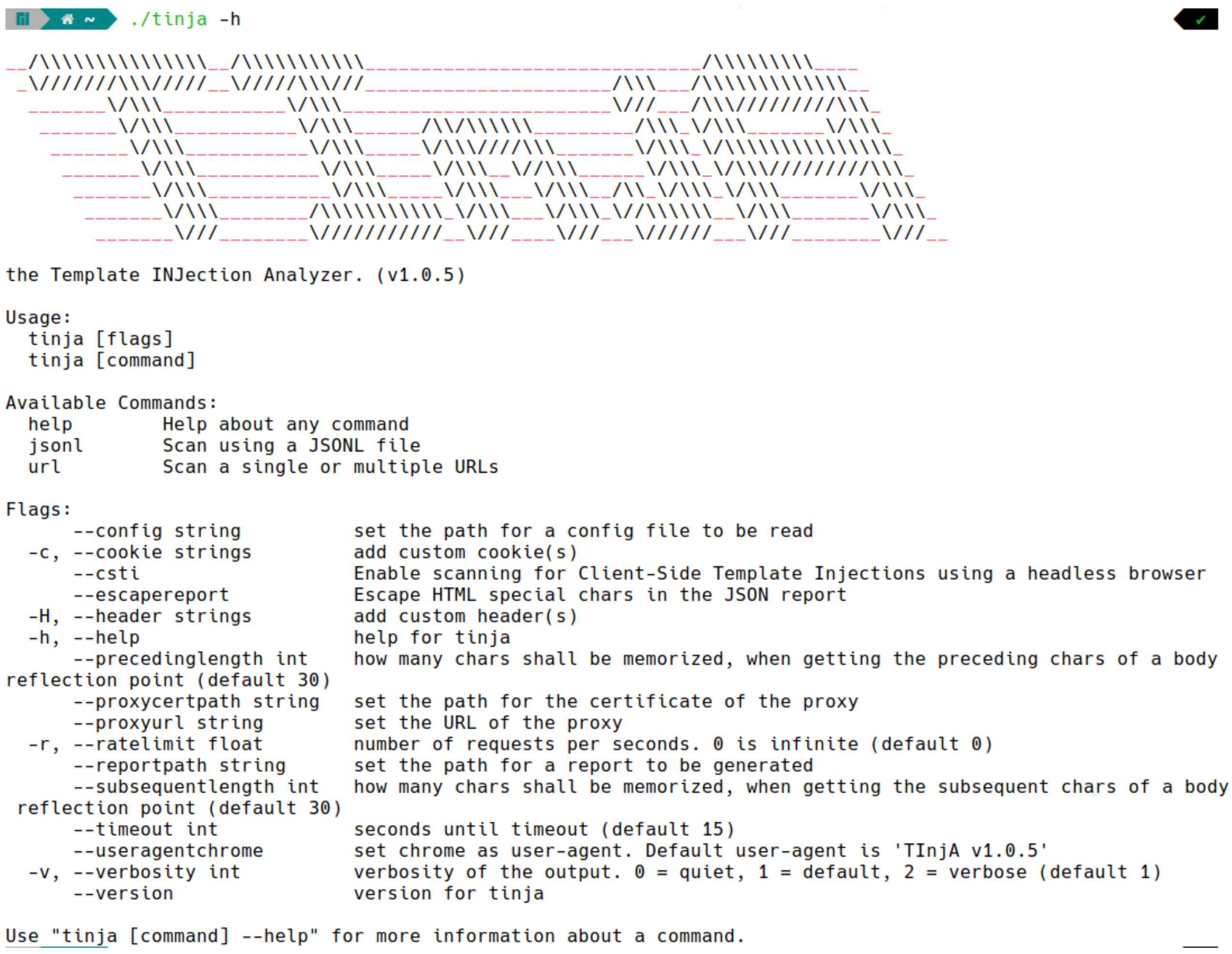
[11] *mustache's website.* URL: `https://mustache.github.io/`.

[12] *SQL Injection Prevention - OWASP Cheat Sheet Series.* URL: `https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html#defense-option-1-prepared-statements-with-parameterized-queries`.

[13] Jiang Wang, Zheng Zhang, Bolin Ma, Yuan Yao, and Xinsheng Ji. "Research on SSTI attack defense technology based on instruction set randomization". In: *ACM International Conference Proceeding Series* (May 2021). DOI: `10.1145/3469213.3471315`.

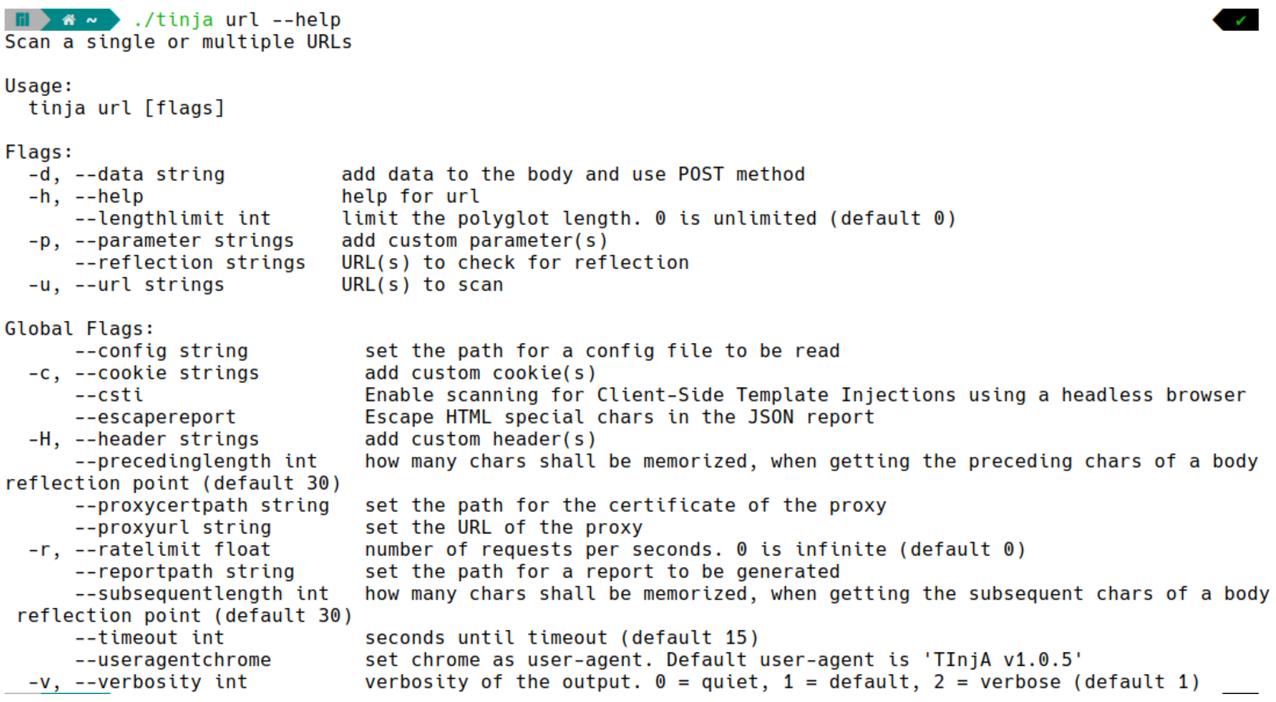
[14] Yudi Zhao, Yuan Zhang, and Min Yang. "Remote Code Execution from SSTI in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs". In: *USENIX Security Symposium* (2023).

A TInjA’s Help Output



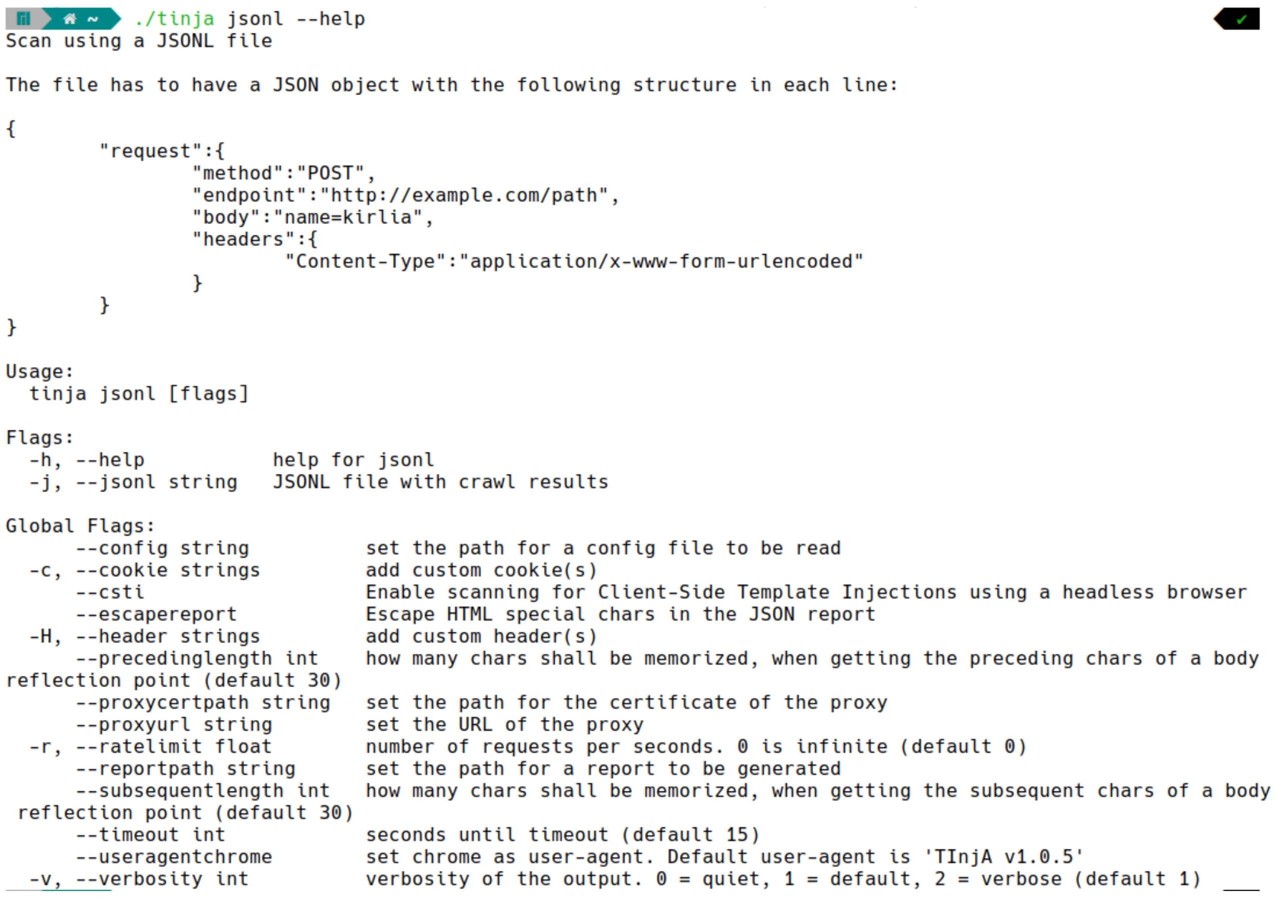
```
./tinja -h

__/\\\\\\\\\\\\\\\__/\\\\\\\\\\\______________________________/\\\\\\\\\____
 _\///////\\\/////__\/////\\\///______________________/\\\___/\\\\\\\\\\\\\__
  _______\/\\\___________\/\\\________________________\///___/\\\/////////\\\_
   _______\/\\\___________\/\\\______/\\/\\\\\\_________/\\\_\/\\\_______\/\\\_
    _______\/\\\___________\/\\\_____\/\\\////\\\_______\/\\\_\/\\\\\\\\\\\\\\\_
     _______\/\\\___________\/\\\_____\/\\\__\//\\\______\/\\\_\/\\\/////////\\\_
      _______\/\\\___________\/\\\_____\/\\\___\/\\\__/\\_\/\\\_\/\\\_______\/\\\_
       _______\/\\\________/\\\\\\\\\\\_\/\\\___\/\\\_\//\\\\\\__\/\\\_______\/\\\_
        _______\///________\///////////__\///____\///___\//////___\///________\///__

the Template INJection Analyzer. (v1.0.5)

Usage:
  tinja [flags]
  tinja [command]

Available Commands:
  help        Help about any command
  jsonl       Scan using a JSONL file
  url         Scan a single or multiple URLs

Flags:
      --config string          set the path for a config file to be read
  -c, --cookie strings         add custom cookie(s)
      --csti                   Enable scanning for Client-Side Template Injections using a headless browser
      --escapereport           Escape HTML special chars in the JSON report
  -H, --header strings         add custom header(s)
  -h, --help                   help for tinja
      --precedinglength int    how many chars shall be memorized, when getting the preceding chars of a body
reflection point (default 30)
      --proxycertpath string   set the path for the certificate of the proxy
      --proxyurl string        set the URL of the proxy
  -r, --ratelimit float        number of requests per seconds. 0 is infinite (default 0)
      --reportpath string      set the path for a report to be generated
      --subsequentlength int   how many chars shall be memorized, when getting the subsequent chars of a body
 reflection point (default 30)
      --timeout int            seconds until timeout (default 15)
      --useragentchrome        set chrome as user-agent. Default user-agent is 'TInjA v1.0.5'
  -v, --verbosity int          verbosity of the output. 0 = quiet, 1 = default, 2 = verbose (default 1)
      --version                version for tinja

Use "tinja [command] --help" for more information about a command.
```

Figure A.1: The main help output of TInjA.



```
./tinja url --help
Scan a single or multiple URLs

Usage:
  tinja url [flags]

Flags:
  -d, --data string          add data to the body and use POST method
  -h, --help                 help for url
      --lengthlimit int      limit the polyglot length. 0 is unlimited (default 0)
  -p, --parameter strings    add custom parameter(s)
      --reflection strings   URL(s) to check for reflection
  -u, --url strings          URL(s) to scan

Global Flags:
      --config string          set the path for a config file to be read
  -c, --cookie strings         add custom cookie(s)
      --csti                   Enable scanning for Client-Side Template Injections using a headless browser
      --escapereport           Escape HTML special chars in the JSON report
  -H, --header strings         add custom header(s)
      --precedinglength int    how many chars shall be memorized, when getting the preceding chars of a body
reflection point (default 30)
      --proxycertpath string   set the path for the certificate of the proxy
      --proxyurl string        set the URL of the proxy
  -r, --ratelimit float        number of requests per seconds. 0 is infinite (default 0)
      --reportpath string      set the path for a report to be generated
      --subsequentlength int   how many chars shall be memorized, when getting the subsequent chars of a body
 reflection point (default 30)
      --timeout int            seconds until timeout (default 15)
      --useragentchrome        set chrome as user-agent. Default user-agent is 'TInjA v1.0.5'
  -v, --verbosity int          verbosity of the output. 0 = quiet, 1 = default, 2 = verbose (default 1)
```

Figure A.2: The url command help output of TInjA.



```
./tinja jsonl --help
Scan using a JSONL file

The file has to have a JSON object with the following structure in each line:

{
        "request":{
                "method":"POST",
                "endpoint":"http://example.com/path",
                "body":"name=kirlia",
                "headers":{
                        "Content-Type":"application/x-www-form-urlencoded"
                }
        }
}

Usage:
  tinja jsonl [flags]

Flags:
  -h, --help            help for jsonl
  -j, --jsonl string    JSONL file with crawl results

Global Flags:
      --config string           set the path for a config file to be read
  -c, --cookie strings          add custom cookie(s)
      --csti                    Enable scanning for Client-Side Template Injections using a headless browser
      --escapereport            Escape HTML special chars in the JSON report
  -H, --header strings          add custom header(s)
      --precedinglength int     how many chars shall be memorized, when getting the preceding chars of a body
reflection point (default 30)
      --proxycertpath string    set the path for the certificate of the proxy
      --proxyurl string         set the URL of the proxy
  -r, --ratelimit float         number of requests per seconds. 0 is infinite (default 0)
      --reportpath string       set the path for a report to be generated
      --subsequentlength int    how many chars shall be memorized, when getting the subsequent chars of a body
 reflection point (default 30)
      --timeout int             seconds until timeout (default 15)
      --useragentchrome         set chrome as user-agent. Default user-agent is 'TInjA v1.0.5'
  -v, --verbosity int           verbosity of the output. 0 = quiet, 1 = default, 2 = verbose (default 1)
```

Figure A.3: The jsonl command help output of TInjA.

B filter-urls.py

```python
import os
from urllib.parse import urlparse

def get_fqdn(url):
    parsed_url = urlparse(url)
    return parsed_url.netloc

def get_second_level_domain(url):
    parsed_url = urlparse(url)
    domain_parts = parsed_url.netloc.split('.')
    if len(domain_parts) >= 2:
        return '.'.join(domain_parts[-2:])
    return parsed_url.netloc

def remove_mismatched_urls(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    output_lines = []
    existing_domains = set()  # Set to save the already present domains

    for line in lines:
        line = line.strip()
        if '[^^[[35m' in line and '^^[[0m]' in line:
            start = line.index('[^^[[35m') + len('[^^[[35m')
            end = line.index('^^[[0m]')
            first_url = line[:start-6].strip()
            second_url = line[start:end].strip()
            first_domain = get_second_level_domain(first_url)
            second_domain = get_second_level_domain(second_url)
            if first_domain == second_domain:
                second_domain_fqdn = get_fqdn(second_url)
                if second_domain_fqdn not in existing_domains:
                    output_lines.append(second_url)
                    existing_domains.add(second_domain_fqdn)
        else:
            domain = get_fqdn(line)
            if domain not in existing_domains:
                output_lines.append(line)
                existing_domains.add(domain)

    filtered_filename = filename + '.filtered'
    with open(filtered_filename, 'w') as file:
        file.write('\n'.join(output_lines))

# Search for files in the current directory with the ".active" extension
directory = os.getcwd()
for filename in os.listdir(directory):
    if filename.endswith('.active'):
        full_path = os.path.join(directory, filename)
        remove_mismatched_urls(full_path)
```

Listing 8: Script to filter out duplicate domains that redirect to the same subdomain and domains that redirect to a different apex domain