ZAP-ESUP: ZAP Efficient Scanner for Server Side Template Injection Using Polyglots

Diogo Silva
diogo.m.r.silva@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2018

Abstract

Recently, Kettle [5] exposed the discovery of a new type of vulnerability which he called Server Side Template Injection (SSTI). This can be considered an A1-Injection that is the class with the highest security risk according to Open Web Application Security Project (OWASP) Top 10 2017 [8]. To the best of my knowledge only 2 solutions have been developed to detect or exploit SSTI, Burp Suite and Tplmap. These solutions are either proprietary software (Burp Suite), or have a limited amount of (fixed) payloads and consequently restricted to a limited number of template engines (Tplmap). None of them can find vulnerabilities when the input is stored and used in other pages (Stored SSTI). In this work, I studied the situations where SSTI may be present, developed a scanner that automatically detects SSTI vulnerabilities in a broader range of situations (reflected, stored, and blind), introduced an efficient technique that uses polyglot payloads to detect SSTI that requires less than 25% of the requests made by the other scanners, and concluded by developing and using a set of vulnerable web applications to compare with the existent solutions. The solution will be made available as a plug-in for OWASP Zed Attack Proxy, a widely used open-source penetration testing tool to find vulnerabilities in web applications.

Keywords: security, web application, SSTI, injection, automated vulnerability scanner

1. Introduction

Manual security testing is a reliable way of testing software, but it is neither efficient nor scalable. If the application has a large set of functionalities or a big codebase, manual testing will require several people and a significant amount of time. A fast and efficient way to perform security testing is to use automated vulnerability scanners that can perform many more tests in a shorter period of time.

Recently, Kettle [5] exposed the discovery of a new type of vulnerability which he called Server Side Template Injection. With this vulnerability it is possible to achieve remote code execution on the server. A Template engine, is a software used to combine templates with data models. Web applications templates contain both static Hyper-text Markup Language (HTML) and template code. This template code is what defines how the dynamic HTML is generated depending on the given data model and can have simple functionalities or the full power of a programming language. The expected usage of these templates is to use a render function that receives 2 arguments: the template and the data model. If the programmer wants to use the user input in the resulting page, he should add it to the data model as represented in Listing 1.

```
template = '<html>${data} </html>'
render(template, USERINPUT)
```

Listing 1: Correct usage of user input.

```
template = '<html>'+ USERINPUT +'</html>'
render(template)
```

Listing 2: Insertion causing SSTI.

SSTI vulnerabilities exist when the user input is incorrectly inserted in the middle of the template instead of being used as the data model argument in the rendering function as represented in Listing 2. Since the user input is inserted in the template, an attacker can inject template code. If the template code has the full programming language functionality, he can run malicious code in the server. SSTI can be considered an A1-Injection that is the class with the highest security risk according to OWASP Top 10 2017 [8].
To the best of my knowledge, there are 2 automated vulnerability scanners able to find SSTI. The first is Burp Suite (Burp) [11]. Burp has a crawler able to automatically obtain the locations it should test and has a vulnerability scanner for the more common vulnerabilities. The second is Tplmap [10], an open-source tool to find and exploit SSTI and code injection vulnerabilities. It is a command line interface tool that receive as input the locations to test. Although Burp is one of the cheapest commercial vulnerability scanners, it may not be accessible to everyone. Tplmap is an open-source tool but contrary to Burp it does not have crawling capabilities, so the locations to test need to be declared manually or by another tool. Another problem of Tplmap is that it is a tool specifically designed for this vulnerability and, according to [1], generic tools are able to detect with a wider spectrum of vulnerabilities are often preferred to others due to cost and time restrictions. Neither Burp nor Tplmap have the capability to detect SSTI when the payload is stored and later injected.

From the 90 template engines enumerated in [16], Tplmap can only detect SSTI in 15 of them as it needs to develop a plugin for each of the template engines, and each of them contains tests made taking into account only the template itself. The tests in Section 5 show that Burp is able to generalise in some cases but may not have been designed with the intention to find vulnerabilities in unknown template engines.

This thesis proposes a solution that automatically searches for SSTI in a more efficient way using polyglots to do fault injections, extending the capabilities of previous tools to stored and blind SSTI, and is able to find SSTI in a bigger number of templates by generalising the payloads sent.

Prior to developing this solution, I conducted a study of the possible situations where this vulnerability may happen based on the existing known cases of successful exploitation of the vulnerability, on other already well-studied vulnerabilities, and on other web vulnerability scanners.

I wanted to create a tool with impact and useful to the community, so I decided to develop my ideas in the format of a plugin to OWASP Zed Attack Proxy (ZAP) that is already a well know and widespread tool for web security testing.

Thus, the 5 main contributions of this project are the following:

1. Study of the situations where SSTI is present.
2. Development of a vulnerability scanner able to detect the cases found in my study.
3. Development of an efficient way to find SSTI using polyglot payloads.
4. Development of web applications to evaluate the efficiency and correctness of SSTI vulnerability scanners.
5. Integration of the tool as a plugin in the ZAP scanner.

2. Background

2.1 Vulnerability classification

A software security vulnerability can be defined as a flaw in the software design, implementation, or operation that can be exploited to violate the security policy [13]. According to [6], input validation vulnerabilities are caused by using malicious inputs without filtering or validating them before, leading to non-intended actions.

I use the word test for each action or group of actions with the purpose of finding a unique vulnerability. The results of a test can be classified according to Table 1.

<table>
<thead>
<tr>
<th>Says is vuln.</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Says non vuln.</td>
<td>True negative</td>
<td>False positive</td>
</tr>
</tbody>
</table>

Table 1: Possible test result classification.

2.2 Vulnerability detection

Depending on the information we have about the tested entity, security testing can be considered white-box testing if it is done with access to the source code of the program, black-box testing if it is done with only an external description of the software, and grey-box if it is a combination of the capabilities of the 2 previous categories.

White-box approach allows a better insight to how the program works compared to black-box, but to have this insight it needs to better understand the semantics of the programming language used. In our case each template engine has its own semantic it would be difficult to create a white-box tool for all the templates. Since I want to make my tool as broad as possible, developing a black-box scanner is a better choice because it does not need to know each of the languages syntax in deep.

Doupé et al. [2] state that web application scanners constituted by 3 modules: a crawler, an attacker module, and an analysis module. The crawler is responsible to discover all the reachable pages and their respective input points. It receives as input one or more URLs and parses the available content in those locations to obtain new URLs to explore. In my solution I will not develop this module since it is going to be integrated in ZAP that already has one. The attacker module analyses the input points found by the crawler. For each of the entry points and for each of the vulnerabilities it tests, the at-
tacker module generates payloads that are likely to trigger the vulnerability. The last module is the analysis module which analyses the result of the requests of the attacker module to detect possible vulnerabilities.

2.3 SSTI exploitation

If we have a vulnerable web application that uses Mako template engine as in Listing 3 we can run Python code in the server by injecting template code. The payload we need to inject in order to execute python in this example is constituted by the template start tag ${ followed by the Python code we intend to execute and at the end the template closing tag } resulting in ${PYTHON code}. The only restriction we have compared to normal Python code is that it needs to be made in a single expression. Then, to execute operating system commands an attacker could send: ${__import__('subprocess').check_output('ls')}$.

Even if the template engine does not allow to achieve Remote Code Execution (RCE), it may allow to cause denial of service, to bypass cross site scripting protections, read or write the system files, or leak application security configurations.

3. Analysis of situations where the vulnerability can happen

One important piece of information to understand this vulnerability is the public cases (Common Vulnerabilities and Exposures (CVE)s and the available bug-bounty reports) where the vulnerability was found. In some of the real cases the vulnerabilities were found by investigating Cross-site Scripting (XSS) vulnerabilities which indicate that they are closely related and consequently the situations where SSTI exist are like the ones of XSS. There are 3 types of XSS: reflected XSS, stored XSS, and blind XSS. Reflected XSS vulnerabilities exist when the user input is reflected in the response without filtering. Stored XSS happens when the user input is stored by the web application and then it is inserted without filtering in another page leading to code execution in that location. Blind XSS is a case of stored XSS but with the additional condition that the place where the injection happens is not accessible to the attacker. I believe that these subvariants also exist in SSTI.

3.1 Result Location and Time of Rendering

In order for an SSTI vulnerability to occur, the user input needs to be inserted in a template and that template needs to be rendered. Based on the location where the rendering result is shown, I divided SSTI in 3 sub-classes: reflected SSTI, stored SSTI, and blind SSTI.

**Reflected SSTI** If we send an SSTI piece of code to the web application and this code is inserted in the template that is used to create the response, we can immediately see the rendering result in the response and consequently identify the vulnerability. In the real examples we can see several cases where this happen [3, 12, 15]. The representation of reflected SSTI is depicted in Figure 1.

**Stored SSTI** Sometimes the user input is stored in some persistent way and may be used in another page of the website and if it is inserted in the template in an unsafe way it can also lead to SSTI. We will refer to these cases as Stored SSTI with posterior injection and rendering. I did not find any real case where a stored input caused an injection in another page. Nonetheless, stored XSS vulnerabilities exist, when user input is stored and later rendered in pages other than the one where the input was sent. In this case to detect and exploit the SSTI an attacker needs to send the payload in one page and then go to the other where the input will be injected and rendered. The timeline representation of the stored SSTI is at Figure 2.

Template engines are not only used to generate web pages, they can also be used to generate emails.
or any other kind of texts. The rendering of this content with injected user input can happen during the processing of the client request but shown somewhere else. We will refer to it as *Stored SSTI with immediate injection and rendering*. Two possible examples of this injection cases are [7, 9], where the rendering input was received in an email. In Figure 3 it is represented a timeline of *Stored SSTI* with immediate injection and rendering.

![Figure 3: Stored SSTI with immediate injection and rendering timeline.](image)

**Blind SSTI** In the previous cases, where I talked about the user input being stored and later causing an injection, I considered that it was always possible for the attacker to see the rendering result. Nonetheless, the existence of blind XSS attacks shows us that the user input may end up in a restricted page as an administrator panel. *Blind SSTI with posterior injection and rendering* is similar to the situation pictured in Figure 2 but the page /b is not accessible to the attacker and later is accessed by some person or program with permission to access it. *Blind SSTI with immediate injection and rendering* is similar to the case pictured in Figure 3 but the email in sent to another person that is not the attacker so he cannot access it.

### 3.2 Location in the Template Where Injection Occur

Another factor that differentiates the possible cases is if the input is inserted in the middle of HTML content (Listing 4) or if it is inserted in the middle of existing code (Listing 5). In the first case a simple template code would result in a correct syntax, but in the second the template we can have an invalid syntax causing the rendering to fail. This factor is completely independent of the previously described cases and it is possible to happen in all of them.

#### 4. Implementation

In this chapter I am going to present my solution and the reasons behind my decisions. The main goals for the scanner are: detecting SSTI in the largest number of engines, including unknown template engines; cover all the cases described in Section 3, i.e., reflected, stored, blind, and inside or outside template code; and be efficient.

```python
name = request.form['name']
template = '...<h1>Hello'+ name +'</h1>...'
return Template(template).render()
```

Listing 4: Insertion of input in HTML zone.

```python
name = request.form['name']
template = '...'${name}=""' + name + '""'}...'
return Template(template).render()
```

Listing 5: Insertion of input in template code zone.

All ZAP plugins should follow some guidelines and one of them is the maximum number of requests done in each of the possible strength values. In my solution, the included capabilities vary between the strength levels, allowing us to follow this guideline and at the same time obtaining the best possible performance. The capabilities in each configuration are represented in Table 2. With these several possible configurations the scanner can fulfil the needs of broader types of users, from the ones that can only make a reduced number of requests to the ones that have time and resources to do a more intense scan. In all these possible configurations it is possible to find reflected and stored SSTI. At the levels “high” and “insane” I added the capability to find blind SSTI. The scanner does this by sending payloads that make callbacks to the scanner. The Insane level includes the ability to find some vulnerabilities where the input is inserted in the middle of existing template code. It consists in fixing the syntax in simple cases. What distinguishes the “low” definition from the “medium” is the usage of polyglots to predict if the vulnerability exists this way saving requests.

<table>
<thead>
<tr>
<th>Poly</th>
<th>Refle</th>
<th>Stor</th>
<th>Blind</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Insane</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2: Capabilities depending on the ZAP Strength configuration

The scanner interaction with the user and the server is done through the ZAP interface. To scan an endpoint, ZAP calls each active scanner indicating the original user Hypertext Transfer Protocol (HTTP) request, the location to inject, and the
original value.

4.1 Architecture and Interactions
The plugin can be divided in separate components that interact with ZAP, with the server under test, and between them. These interactions are represented in Figure 4.

The Sink Manager (SinkM) has the responsibility of storing and retrieving information from all the known sink points which are the locations where the payloads appear and an injection may occur. The SinkM obtains the stored sinks from the Stored XSS scanner of ZAP. The efficient vulnerability detector sends polyglot payloads in the parameter under test, then asks SinkM the normal similarity between responses with different inputs in that sink, and the similarity between the new response and the original one. With this it can define if there was a strange behaviour on the web application. The SinkM uses the Message Comparator (MComparator) to obtain the similarity between the responses. If the Efficient Vulnerability Detector (EfficientVD) considers that existed a strange behaviour, the scanner calls the Arithmetic Evaluation Detector (ArithmeticED). The ArithmeticED tries to find the vulnerability by sending multiplication code as \(777*222\) and then asks SinkM for the string representation of the actual state of the sinks. If it includes the multiplication result it considers that a vulnerability was found. The Syntax Fixer (SyntaxFix) module is a type of wrapper to the ArithmeticED that sends the ArithmeticED requests with prefixes to fix the syntax before rendering the normal payload. It sends them several times with different prefixes. The Blind Vulnerability Detector (BlindVD) operates by sending payloads that make a callback to ZAP ChallengeCallback plugin. This module receives requests to register a certain token and returns an Uniform Resource Locator (URL) to which the callback should be done. The plugin sends the payload with the intended URL and whenever ZAP receives one request in that endpoint it informs the plugin.

4.2 Sinks Manager
The SinkM has the responsibility of storing and retrieving information from all the known sink points. At this moment these sink points only include the response to the request and the pages of the website where the input is shown later, but it is easy to implement a sink abstraction for emails. The sinks used to test for stored vulnerabilities are obtained from the Stored XSS scanner plugin. Thanks to this, it is possible to know the locations where the input is going to be reflected without any additional requests. To obtain these locations SinkM makes a function call to the plugin with the endpoint and the parameter tested and it returns the sink locations. The stored sink points should be created immediately after sending one reference request to the server because once created they will obtain the current state of the sink and compare it to the one stored by the XSS scanner. The result of this comparison will be considered the normal variation of the response between 2 different inputs. Then, each time the sink is updated the actual similarity is stored. The same happens with the reflected sinks (the responses to the requests where the payload is sent), that at creation receive the original user request and one request created to serve as reference. The similarity between the two is stored to be the reference similarity.

4.3 Efficient Vulnerability Detector
This component has the objective of deciding if it is worth to send all the payloads from ArithmeticED that will be described in Section 4.5. It allows us to save a large number of requests if there is no vulnerability. Since the case where the vulnerability does not exist is the most common case by far, this will enhance my scanner performance.

Polyglot payloads allow to exploit several contexts where a vulnerability can exist with just one payload. Inspired by this idea I decided to create one polyglot to identify SSTI. Contrary to XSS that needs to be aware only of HTML and JavaScript, this polyglot needs to work on several languages and templates making the task impossible within a reasonable size. However, to detect SSTI the polyglot does not need to render in the server, it just needs to prove that the engine is trying to render it. The easier way to detect that the input is being rendered in a template is the failure in rendering due to syntax errors. Causings errors is easier to achieve than creating a correct syntax because it can be independent of the rest of the code. If a normal SSTI payload is inserted in the middle of the HTML it will render and if it is inserted in the template code it will probably fail, but if the EfficientVD can create an incorrect syntax in the first it will also be incorrect in the second. Another advantage of creating an incorrect syntax is that a payload that causes errors in one template may also cause errors in another template engines with a similar tag syntax. I believe that even if the error is correctly caught there will also exist a deviation from the normal behaviour.

With all these facts I decided to detect SSTI by causing errors using the same polyglot payload for all the template engines. Main logic of the scanner My idea assumes that it is possible to detect errors because if I can cause errors but not detect them then this idea would not work. Some of the more evident errors can be detected by humans. For instance, if we see an er-
error page we know that something unexpected happened and the same happens to a scanner if an error HTTP code is returned. In the subtler cases that it is difficult for humans to detect errors, what we would do in these situations would be to compare the response when we try to cause the error with the response of a normal utilisation. This is exactly the idea I try to implement in my efficient scanner inspired by [4]. The objective of the Backslash scanner is to detect interesting behaviours while the EfficientVD objective is to improve performance without increasing the false negative rate, consequently the requirements will also be different. Some parameters, such as search inputs, cause big differences in the response so the difference in HTML structure will always happen and the responses will always be considered different. When following the logic of the backslash scanner the result obtained from the safe element of the probe pair will be different from the base request and consequently not interesting what could result in a false positive. To avoid this problem, instead of comparing the responses with a reference payload created by us we compare them with the original user request taking in consideration the alterations the reference request has from the original. My strategy is depicted in Figure 5.

The first step is to send a reference request and compare it with the original user request. Then the EfficientVD sends the polyglot and compares the response with the original user request. If the difference between the similarity of these pairs is low, it means that the variation obtained from the original request is the normal one when a different input is sent. If the difference in similarity between the pairs is big it means that the scanner considers that a strange behaviour occurred. One possible problem is that the characters used in the template engines are usually special characters, including the ones used in XSS < and >, that may cause the request to fail not because of rendering but because some character was being blocked by the application or a Web Application Firewall (WAF). To verify that the error is not due to these character problems the EfficientVD sends a safe version of the polyglot in which the special characters are escaped to make them innocuous to the template. If the similarity of this request with the user original request is also very different from the reference one, the EfficientVD considers that the error is not caused by the template engine and it is not vulnerable. Otherwise, it considers there exists a strange behaviour and SSTI might be possible. Details about how the comparison between 2 responses is done is discussed in the following Subsection 4.4.

The reference request has the objective of detecting the normal variations caused by sending a different value in the parameter. To create a reference payload that is different but close to the one the user would send, we use characters already present
in the user original request.

**Developing a SSTI Polyglot to Cause Errors** Before creating the polyglot, I needed to know which is the most effective way of causing errors. To causes errors we need to make an incorrect syntax so I tested and compared 4 combinations of template tags: template start tag (ex: `_${}`); ending tag (ex: `'}'); start tag followed with an ending tag (ex: `_${}`); and start tag, a variable name, and the ending tag (ex: `_${foo}'`). By testing these combinations of tags in 18 web applications [14] using different template engines, we concluded that the best way of causing errors was to send start tag, a variable name, and the ending tag which caused a strange behaviour in 16 of the 18 applications. The variable inside the tags should not exist in the context because its nonexistence in the context is what causes the error.

The applications where I did not cause strange behaviour were all Java because these engines must handle errors better that the ones from other languages and calling an existing variable is not enough to cause a visible error. To solve this problem, I created a specific polyglot for them constituted with code that disrespect their syntax in other ways that cause errors: `<th t="${xu}`

Since the best way of causing error is to have the initial and end tag with something invalid inside, I created one polyglot getting each template start tag and end tag around the payload already existing. The generated payload was:

```html`
<%=\{\{=${{xux}}%>
```

The resulting payload contained repeated combinations at the left and right caused by the usage of the same tags by several template engines. These characters were redundant. To reduce the payload size, I removed repeated tags. By reordered the tags in a way they could share the common characters I obtained a smaller payload that was:

```html`
<%=\{\{=${{xux}}%>
```

which I call general polyglot. The escaped version of the general polyglot was backslashed to avoid rendering, resulting in:

```html`
\%\{\%\{=${{xux}}%}\%
```

Smarty returned errors with the backslashed polyglot, so I added the Twig commentary tags `{*...*}` around the polyglot solving the problem. I did no created a unique polyglot because one of the common restrictions in parameters is the maximum size and it would create a big payload.

**4.4 Message Comparator**

The objective of the Message Comparator is to compare 2 different responses and return a value that represents how similar they are from each other.

To compare different responses R1 and R2, the MComparator uses heuristics and each heuristic has one weight associated. The variable representing the final result is between 0 and 1: 0 means that the responses are very different, and 1 that are very similar. Every evaluation starts with the total variable set to 1 and then is multiplied with each of the heuristic (Heu) result taking in account its weight (w), leading to the formula:

\[
Total = \prod_{H \in Heu} H(R_1, R_2) \times w(H) + (1 - w(H)).
\]

The MComparator has 6 heuristics with different weights. The HTML pages are structured by elements and each element may contain one or more children elements. It is possible to define one path between the HTML element that contains all the others (`<HTML>`) and each of the elements that do not contain any children including all the elements between them. **Body trees structure heuristic** is calculated by comparing the paths of 2 responses HTML. The status code heuristic compares the status code of 2 responses that when different always means that the responses are different. Besides the status code, other response headers may be indicators of different behaviour in the server. One of the possible effects the polyglot causes in the server is the disappearance of the input caused by rendering or errors, so that is what input reflection heuristic tries to detect. The line and word count heuristic is used to detect subtle changes in the HTML. Some English words are very common in error messages, so if they appear in one message and not in the other is probable to exist an error. This is captured by the relevant keywords count heuristic.

The names and main idea from the heuristics used are the same as the ones in [4], but they are calculated and implemented in different ways.

**4.5 Arithmetic Evaluation Detector**

What the ArithmeticED does is simply send all the tags it knows with arithmetic operations inside and observes if the expected operation result is in any of the sinks response. One example of payload is `$\{\{N1+N2\}\}$`, where the numbers are randomly generated for each request and the result it searches in the sinks response is the result of the operation $N1 \times N2$.

This method is more generic than other solutions because it allows to find the vulnerability in any existent template engine that respect 2 requirements: have one of the tags we collected in our engines examples, and evaluate and return the result if we send a multiplication operation inside the tags. The biggest drawback of this solution is the existence of template engines that do not follow the second restriction as is the case of Django, DustJs and Golang. For these template engines, I created specific test cases similarly toTplmap.
4.6 Blind Vulnerability Detector
To detect blind SSTI we use the callback module of ZAP. First, the BlindVD registers a certain token in the callback module and receives an URL with a specific path that should be accessed by the vulnerable server in the case the vulnerability exists. Secondly, it adds the URL to payloads that when rendered make a request to the specified URL. When ZAP receives one request in that endpoint it informs the BlindVD about it. If the BlindVD receives the information, it considers that the web application is vulnerable. The payloads that make the callbacks are made specifically to each template engine and are all predefined.

4.7 Syntax Fixer
The main objective of this module is to generate payloads able to execute if the payload is inserted inside existing template tags as in Listing 5. Inserting ArithmeticED payloads in the middle of the template code will create an incorrect syntax, so SyntaxFix adapts the ArithmeticED payloads by adding one prefix that fixes the current syntax.

The first thing needed to fix the syntax of global template is to fix the code where the input is inserted. To escape and fix the code, the SyntaxFix uses one of the 3 payloads: (doublequote), (quote), and number. Then, SyntaxFix needs to end the current template code to start a new one. To end the template code, it adds the end tag used in the template. At this point, the SyntaxFix is in a clean stage, so it can finally send the usual payload generated by the ArithmeticED.

The payload generated to detect the vulnerability in Listing 5, would be ")${99*99}. This payload can be split in 3 parts: the first is to escape the string in the code ", the second is to fix the template syntax }, and the last is ${99*99} to execute code and detect SSTI.

5. Results
To evaluate my solution, I developed several web applications [14] to test its abilities in the several cases described at Section 3 as well as its performance. I considered as the main evaluation metrics the vulnerability detection rate which is calculated by dividing the number of vulnerabilities found by the number of existing ones and number of HTTP requests. I will not consider the number of false positives because none of the scanners had them. The result of those metrics will be compared against the same results using the scanners Burp Suite Pro and Tplmap. Since my plugin will be part of ZAP, in this chapter I will refer to my scanner plugin as ZAP.

If not stated otherwise the configuration of the scanners was: BurpSuite PRO 2.0 scanner with the “accuracy” option to minimise the false negative rate, and the “Audit speed” as fast; Tplmap was set with level equal to 0 which is the number of levels of nested code to fix before the payload. Later in the fixing code tests this value was set to 1; ZAP was configured with attack strength to “low” to test its optimisation, except for the test where the rendering result is never visible to the attacker (blind SSTI). In that case the chosen configuration was “high” because it is the lowest level where my tool uses payloads that make callbacks.

5.1 Simple Tests - Reflected Results
The first test intends to know if the scanners are able to detect simple SSTI vulnerabilities. Failing in these tests implies that they are not able to detect the vulnerability in a more complex situation with the same template engine. I created 19 vulnerable web applications using six different programming languages. The only thing the web application does is to receive one parameter “name” in the request, insert the parameter in the template before rendering, and return the rendering result in the response.

The results obtained from the test show that all scanners had a similar vulnerability detection rate in the simple vulnerabilities. ZAP found 84.2% of the vulnerabilities, Burp 73.7%, and Tplmap 78.9%. This result should not be used as a strict distinction between all the scanners because they can detect vulnerabilities in template engines which I did not contemplated.

Some template engines in the tests do not execute arithmetic operations, e.g., Django, making ZAP (before including specific tests) unable to find the vulnerability while Burp has specific tests for Django and finds it. The Golang web application vulnerability was not found by any of the scanners. ZAP was unable to find it because the template engine does not execute arithmetic operations failing in one of the conditions, but the others fail because they did not have a specific test to it.

5.2 Stored and Blind SSTI Test Cases
As we have seen in my analysis of the possible subvariants of SSTI performed in Section 3, just looking for reflected SSTI in the server response to the request is not enough to find all cases of SSTI. Therefore, I created tests for stored and blind SSTI.

To test the ability of detecting the vulnerability in special cases and not having the result influenced by the capability of each scanner to detect the vulnerability for a certain template engine I created these web applications using the Mako template engine. All tested scanners were able to detect the vulnerability in the simple test using this template engine.

The first test, Test1, tries to represent the stored SSTI with posterior injection and rendering represented in Figure 2. This web application is similar...
to the simple tests performed in Section 5.1 as it receives a single parameter (name). However, instead of inserting it in the template rendered for the response it is stored in a file. Later, when the "/stored" endpoint is requested the content of the file is read and inserted in the template before being rendered.

The second test simulates stored and blind SSTI with immediate injection and rendering represented in Figure 3. The test code is exactly the same as the one from Mako simple test but instead of using the user input in the template of the response, it is used in a template which result is never shown.

<table>
<thead>
<tr>
<th>Test</th>
<th>Burp</th>
<th>ZAP</th>
<th>Tplmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1 Stored SSTI</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Test 2 Blind SSTI</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3: Stored and Blind SSTI tests results

Tplmap has no way to declare possible result locations or crawling capabilities so it was not able to detect the stored SSTI. I do not know for sure why Burp did not find this vulnerability, but the most probable reason is that it only searches for input rendering in the response of the request itself. This means that Burp is unable to find stored SSTI. ZAP was able to find this vulnerability since it searches on each of the sink points (endpoints where that parameter value is posteriorly shown) for signs of input rendering.

In the case of stored and blind SSTI with immediate injection and rendering Burp was also not able to detect the vulnerability. The other two scanners detected the vulnerability but using 2 different strategies. While ZAP was able to detect the vulnerability by sending requests that make callbacks to the attacker, the Tplmap found the vulnerability by sending requests with template code that caused a time delay. If the delay was noticed in the response the Tplmap considers it as vulnerable.

5.3 Injection inside template code tests

As seen when defining the several variants of SSTI in Section 3, SSTI payloads can be injected inside or outside the template code. For that, I created two tests with Mako template engine to evaluate the scanners in this area.

The payload insertions in the template were:

\[\ldots<h2>\{1234 == %s\} \</h2\>\ldots\]

and

\[\ldots<h2>\{"prefix"+"%s"\} \</h2\>\ldots\]

The user input will replace the %s in the template.

All the scanners detected the vulnerability when the input was inserted in the middle of template code inside a string. Burp detected this vulnerability as RCE.

In the case the input is inserted in code doing an arithmetic operation Burp was the only one that did not find it. ZAP was able to find the vulnerability in these two simple cases of injection inside existing code, but it will fail in more complex situations. On the other hand, Tplmap has a deeper knowledge about each of the template engines and makes several possible combinations with the elements of each engine to detect the vulnerability in these situations. This is, however, at the cost of losing generality and the need to have such payloads defined manually for each template engine.

5.4 Performance Tests

Table 4: Injection inside template code tests results

<table>
<thead>
<tr>
<th>Test</th>
<th>Burp</th>
<th>ZAP</th>
<th>Tplmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inserted in math</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Inserted in text</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4: Injection inside template code tests results

It is possible to see that Tplmap is the scanner that does the highest number of requests to a vulnerable endpoint having an average of 29 requests and 55 when it is not vulnerable. This is due to the effort to find vulnerabilities blindly, and due to the number of template engines contemplated since it does specific tests for each one. The average number of requests Tplmap does when the site is vulnerable is not so far away from the requests that ZAP does when the settings are set "high" for strength (22 when vulnerable and 24 otherwise). That is the level where ZAP starts to also detect blind vulnerabilities making the comparison of the two more fair.

ZAP’s results may seem contradictory at first. Comparing the average of requests in “low” and “medium”, the reader sees that the values in “medium” are lower than in “low”. This happens because when “low” is selected my solution first

<table>
<thead>
<tr>
<th></th>
<th>Burp</th>
<th>Tplmap</th>
<th>ZAP low/med</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg vuln</td>
<td>12</td>
<td>29</td>
<td>12/8</td>
</tr>
<tr>
<td>Max</td>
<td>24y</td>
<td>55n</td>
<td>17n/12n</td>
</tr>
<tr>
<td>Min</td>
<td>8y</td>
<td>2y</td>
<td>8y/3y</td>
</tr>
<tr>
<td>Non vuln</td>
<td>17</td>
<td>55</td>
<td>4/12</td>
</tr>
<tr>
<td>Two sinks</td>
<td>14</td>
<td>55</td>
<td>16/7</td>
</tr>
</tbody>
</table>

Table 5: Performance Tests Table. y: result obtained in situation where found vulnerability; n: the opposite of y. ZAP requests do not include the one necessary to find the sinks. avg is average. vuln is vulnerable

9
uses the polyglots to know if the web application may be vulnerable. If the application is vulnerable it sends the same requests that “medium” sends. These actions increase the number of requests by 3 if it is a non Java engine and 5 if it is. This average is from the cases when the endpoint is vulnerable. Since the majority of the entry points in the web application will not be vulnerable, the most relevant value is the number of requests made when there is no vulnerability.

Considering a case where the “low” setting is used to test a web application with 50 entry points, 1 of which is vulnerable and 9 other have strange behaviour. The number of requests made by Burp should be 12 + 17 × 49 = 845, Tplmap 29 + 55 × 49 = 2724, ZAP “medium” 8 + 12 × 49 = 596 requests, and ZAP “low” 12 + 17 × 9 + 4 × 40 = 325.

The number of requests made by ZAP in the “low” setting are 38% of the ones made by Burp and 12% of the ones done by Tplmap. In an optimistic scenario where it does not exist any SSTI and the scanner predictor works correctly, the number of requests made by ZAP in the “low” setting is 4/17 = 24% of the ones made by Burp and is 4/55 = 7% of the ones done by Tplmap.

**Impact of the number of sinks in ZAP performance**

Another important factor that needs to be taken into consideration is that in order to test stored SSTI, the number of requests performed by ZAP grows linearly with the number of existent sinks for the entry-point under test. My scanner sends one payload and then searches on each of the sink points for signs of a possible SSTI. Thus to get the number of requests made by ZAP to test stored SSTI in “low” and “medium” I multiply the number of sinks by the number of requests stated in Table 5. For instance, for an entry-point with 6 sinks, ZAP needs to perform 6 times the requests that would make whenever there was a single sink.

When ZAP searches for reflected SSTI there is only a single sink hence the values are the ones stated in Table 5.

### 6. Conclusions

This thesis presented the study of Server Side Template Injection which is a recent vulnerability, the development of a black-box web application vulnerability scanner, and the evaluation of my solution as well as the comparison with other SSTI scanners.

I categorised SSTI variants depending on the time of rendering, location of rendering, and the location inside the template where the input may be injected. The study was based on real cases and well studied vulnerabilities which allowed me to define cases not contemplated by any of the already existent solutions.

Then, I developed a black-box web application vulnerability scanner in the format of a plugin for OWASP-ZAP which had as main objectives the efficiency, and the ability to generalise the tool to the largest number of engines possible, the ability to cover all the types of SSTI previously studied.

To conclude my work, I have done several tests to the scanners and thanks to the previous study, my scanner was able to find stored SSTI which none of the previously existent solutions were. This is an important feature since two of the collected real cases were of this type. Additionally, the performance tests proved that our efficient scanner using polyglots is able to find the vulnerability with just 24% of the requests that Burp uses and 7% of the requests that Tplmap uses.

### References


