CTF Challenge: "Sasti" - Exploiting SSTI Vulnerability

-By Vedant Shanker (24/B03/056)

Introduction

In this challenge, we were provided with a web application that appeared to be vulnerable to Server-Side Template Injection (SSTI). The objective was to identify and exploit this vulnerability to extract sensitive information, ultimately obtaining a flag.

Understanding the Technologies Involved

Before diving into the details of the vulnerability and exploitation process, it's important to understand the underlying technology and the type of vulnerability we are dealing with. This section will cover the basics of Flask and Server-Side Template Injection (SSTI).

What is Flask?

Flask is a lightweight web framework written in Python. It's often referred to as a microframework because it provides the essentials for web development but leaves room for developers to choose additional components they need. Flask is widely used due to its simplicity, flexibility, and ease of use, especially for smaller applications or services.

Key Features of Flask:

- **Routing:** Flask allows you to define routes for your web application using decorators. For example, @app.route('/') defines the route for the home page, mapping it to a specific function.
- **Templating with Jinja2:** Flask uses the Jinja2 template engine to allow dynamic content generation within HTML templates. This means that you can insert variables and even logic into your HTML, which Flask will process on the server-side before sending the final page to the user.
- **Configuration Management:** Flask applications can load configuration settings from various sources, such as files or environment variables. This flexibility allows developers to manage different settings for development, testing, and production environments.

Understanding Flask is essential because it helps us grasp how the web application processes requests, handles user input, and generates responses. In this challenge, recognizing that Flask uses the Jinja2 template engine led us to suspect and eventually confirm a Server-Side Template Injection (SSTI) vulnerability.

What is Server-Side Template Injection (SSTI)?

Server-Side Template Injection (SSTI) is a critical web vulnerability that occurs when user input is improperly handled by a server-side template engine, such as Jinja2, which Flask uses. This vulnerability allows attackers to inject and execute arbitrary code on the server, potentially leading to full system compromise.

How SSTI Works:

- **Templates and Dynamic Content:** Web applications often use templates to generate HTML that includes both static content and dynamic data, such as user-specific information or data fetched from a database.
- **Injection Point:** If an application fails to properly sanitize user input that is incorporated into a template, an attacker can inject malicious code into the template. The template engine processes this code on the server, which can result in the execution of arbitrary commands or the exposure of sensitive data.

Code Provided

We were given the following Python code running on a Flask web server :

```python
#!/usr/bin/env python3

from flask import Flask, render_template_string, request, Response

app = Flask(__name__)
app.config.from_pyfile('topsecret.py')
@app.route('/')
def index():
    return Response(open(__file__).read(), mimetype='text/plain')

@app.route('/vuln')
def vuln():
    query = request.args['query'] if 'query' in request.args else 'eh?'
    if len(query) > 21:
        return "nononono"
    return render_template_string(query)

app.run('0.0.0.0', 1337)
```

Step-by-Step Process

1. Initial Analysis of the Code

Upon reviewing the provided code, the following key points stood out:

- The `@app.route('/')` route returns the content of the script itself. This could be useful if we didn't have the source code, as it would allow us to read it directly from the server.
- The `@app.route('/vuln')` route takes a query parameter from the URL and passes it to the render_template_string function. This function is typically used to render templates in Flask, and passing user input directly to it without sanitization suggests a potential SSTI vulnerability.
- The line `app.config.from_pyfile('topsecret.py')` loads configuration from a file named topsecret.py, indicating that the flag might be stored in the application's configuration. This means that any sensitive information defined in that file, including the secret key, is stored in the Flask application's configuration dictionary.

2. Injecting the Payload into the URL Bar

Understanding the URL Structure

Before diving into how the payload was injected, it's crucial to understand the structure of a URL and how each part functions. A typical URL consists of several components:

`http://localhost:8003/vuln?query={{payload}}`

- **Protocol (http://):** Specifies the protocol used to access the resource, in this case, HTTP.
- **IP Address ($ip):** This is the IP address of the server hosting the web application which in this case is localhost (hosted on a local machine, that other computers that are not on the same network can't access). It can also be a domain name like www.example.com.
- **Port ($port):** The port number specifies the network port on which the server is listening. For HTTP, the default is 80, but in this case, it's set to 8003.
- **Path (/vuln):** This is the path to the specific resource on the server. In this challenge, /vuln refers to the vulnerable route defined in the Flask application.
- **Query String (?query={{payload}}):** The query string is a part of the URL that contains data to be sent to the server. It begins with a "?" and is followed by key-value pairs, such as `query={{payload}}`.

Injecting the Payload

In this challenge, the payload was injected into the query parameter in the URL's query string. Here's how it was done:

1. **Identify the Vulnerable Route:** The /vuln route was identified as the vulnerable route that could potentially execute user-provided input. The Flask code snippet showed that it accepted a query parameter.
2. **Craft the Payload:** The payload was crafted to exploit the SSTI vulnerability. For example, to evaluate a simple arithmetic expression, the payload {{23*3}} was used.
3. **Construct the URL:** The payload was inserted into the URL like this:

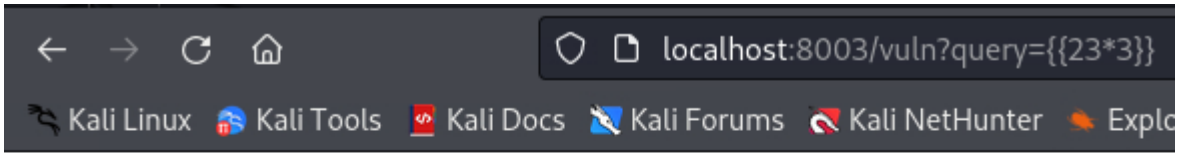
`http://localhost:8003/vuln?query={{23*3}}`

This URL was then entered into the browser's address bar.

Execution: When the URL was accessed, the Flask application processed the query parameter and executed the payload due to the SSTI vulnerability.

Result:

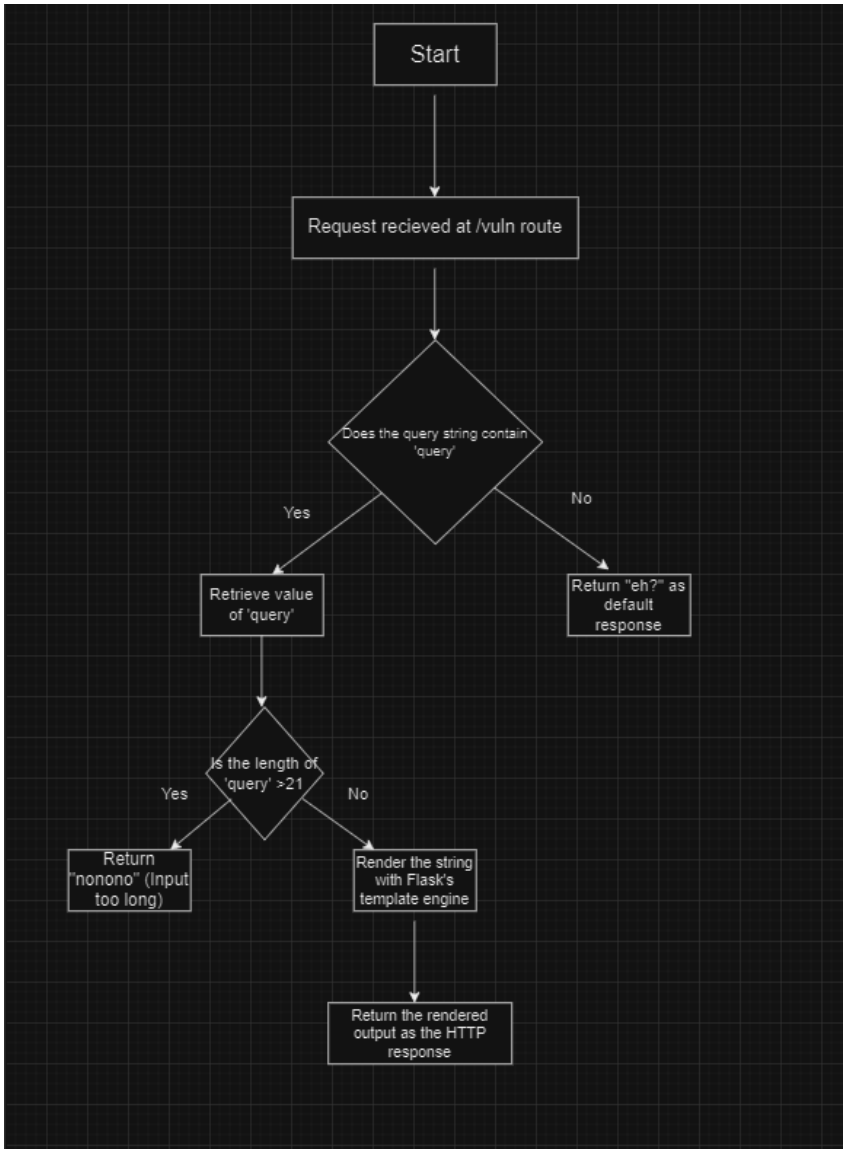
The result of the expression 23*3 =69 (nice) was returned as the response.



69

Flowchart: How the Code Processes Each Query Case

To visualize how the Flask code processes different queries, here's a simple flowchart that illustrates the logic:



3. Exploration of the Flask Configuration

With the SSTI confirmed, the next step was to explore the Flask application's configuration to locate the flag. The `app.config.from_pyfile('topsecret.py')` line suggests that sensitive information might be stored there, like stated before!

I then attempted to access the configuration items using another SSTI payload:

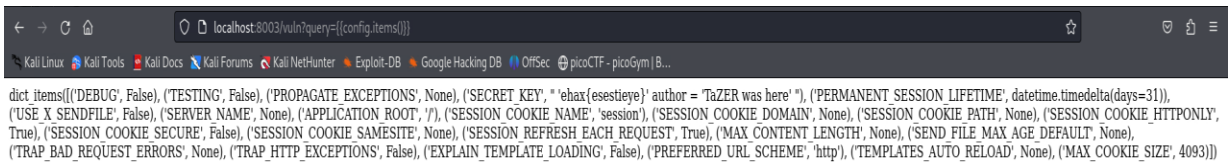
`http://localhost:8003/vuln?query={{config.items()}}`

Expected Result:

- This should list all configuration items, which might include the flag.

Result:

- The output included the flag as part of the configuration.



Here, browsing the text that was rendered, I find the flag:

"ehax{esestieye}"

Missteps and Exploration

During the process of solving the challenge, I took several exploratory steps to understand the application's behaviour and test different hypotheses. Here are some of the paths I explored, including some missteps that ultimately led to the final solution:

Initial Thought Process:
Given that web applications can have multiple types of vulnerabilities, I wanted to ensure that SSTI was the primary issue before focusing all my efforts there. I briefly considered the possibility of other common web vulnerabilities, such as SQL Injection, Cross-Site Scripting (XSS), or Path Traversal.

Action Taken:

- SQL Injection: I attempted to pass SQL-related syntax through the query parameter to see if there were any backend database interactions that could be exploited. For example, I tried injecting `?query={{' OR 1=1 --}}` to test for potential SQL injection.
- XSS: I experimented with injecting JavaScript code to see if it would be reflected in the response, which might indicate an XSS vulnerability. For instance, `?query=<script>alert('XSS')</script>`.
- Path Traversal: I tested for path traversal by trying payloads like `?query={{open('../etc/passwd').read()}}` to check if I could access files outside the intended directory.

Result:

- SQL Injection: The application didn't interact with a database in a way that allowed for SQL Injection, as Flask's render_template_string was the primary function handling input.
- XSS: The injected JavaScript was not executed in the browser, indicating that the server's response wasn't directly rendering HTML but rather processing it server-side through the template engine.
- Path Traversal: The path traversal attempts did not succeed, likely due to the application's design and environment. However, this reinforced the hypothesis that SSTI was the main vulnerability to exploit.

Conclusion

This challenge demonstrated the risks associated with improperly handling user input in template engines. By identifying and exploiting the SSTI vulnerability, I was able to retrieve the flag from the application's configuration.

Key Takeaways:

- Always sanitize user inputs, especially when they are used in functions that can execute code.
- Understand the underlying frameworks and their configurations, as they might expose sensitive information if not properly secured.