# Server-Side Template Injection (SSTI)

This technique was first documented by PortSwigger Research in the conference presentation [Server-Side Template Injection: RCE for the Modern Web App](https://portswigger.net/research/server-side-template-injection) (2015).

Server-side template injection is when an attacker can use native template syntax to inject a malicious payload into a template, which is then executed server-side.

## Template

Simply putting it, it’s an HTML file that contains variables. Something like

<h1>{{greeting}}!</h1>

Depending on the template type, a variable greeting is defined between {{ }}.

If we pass “hello username” to greeting, then the HTML would be:

<h1>hello username!</h1>

A common example is, when a user login into app, the app fetches the name of the user and pass it to greeting variable. The user will see:

hello username!

So, templates are used by backend app to render data dynamically into HTML.

## Essentially…

If the app blindly takes a user input (such as username) and render it into a template. Then the user can inject arbitrary code which the template will evaluate. Such injection will allow the user to access some APIs and methods which are not supposed to.

## Constructing an SSTI attack

## A diagram of a computer Description automatically generatedDetect

### Plaintext Context

As with any vulnerability, the first step towards exploitation is being able to find it. Perhaps the simplest initial approach is to try fuzzing the template by injecting a sequence of special characters commonly used in template expressions, such as the polyglot ${{<%[%'"}}%\.

To check if the server is vulnerable, you should spot the differences between the response with regular data on the parameter and the given payload.

The given input is being rendered and reflected into the response. This is easily mistaken for a simple XSS vulnerability, but it's easy to differentiate if you try to set mathematical operations within a template expression.

### Code Context

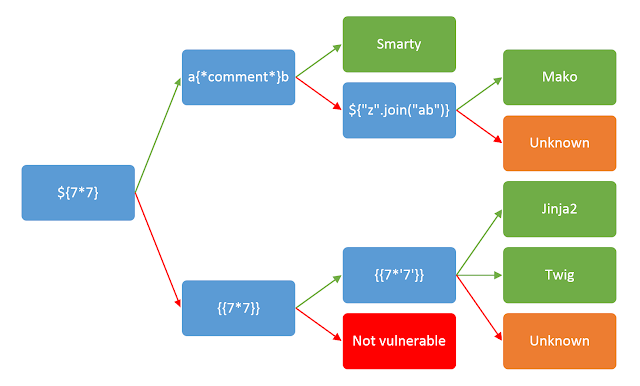
Once you have detected the template injection potential, the next step is to identify the template engine.

Although there are a huge number of templating languages, many of them use very similar syntax that is specifically chosen not to clash with HTML characters.

If you are lucky the server will be printing the errors and you will be able to find the engine used inside the errors. Some possible payloads that may cause errors:

|  |  |  |
| --- | --- | --- |
| ${} | {{}} | <%= %> |
| ${7/0} | {{7/0}} | <%= 7/0 %> |
| ${foobar} | {{foobar}} | <%= foobar %> |
| ${7\*7} | {{7\*7}} | `` |
| #{7\*7} | \*{7\*7} |

Otherwise, you'll need to manually test different language-specific payloads and study how they are interpreted by the template engine. A common way of doing this is to inject arbitrary mathematical operations using syntax from different template engines. You can then observe whether they are successfully evaluated. To help with this process, you can use a decision tree like the following:



You should be aware that the same payload can sometimes return a successful response in more than one template language. For example, the payload {{7\*'7'}} returns 49 in Twig and 7777777 in Jinja2. Therefore, it is important not to jump to conclusions based on a single successful response.

## Exploit

### Read

The first step after finding template injection and identifying the template engine is to read the documentation. Key areas of interest are:

* 'For Template Authors' sections covering basic syntax.
* 'Security Considerations' - chances are whoever developed the app you're testing didn't read this, and it may contain some useful hints.
* Lists of built-in methods, functions, filters, and variables.
* Lists of extensions/plugins - some may be enabled by default.

### Explore

Assuming no exploits have presented themselves, the next step is to explore the environment to find out exactly what you have access to. You can expect to find both default objects provided by the template engine, and application-specific objects passed into the template by the developer. Many template systems expose a 'self' or namespace object containing everything in scope, and an idiomatic way to list an object's attributes and methods.

If there's no built-in self object, you're going to have to brute force variable names using SecLists and Burp Intruder's wordlist collection.

Developer-supplied objects are particularly likely to contain sensitive information, and may vary between different templates within an application, so this process should ideally be applied to every distinct template individually.

### Attack

At this point you should have a firm idea of the attack surface available to you and be able to proceed with traditional security audit techniques, reviewing each function for exploitable vulnerabilities. It's important to approach this in the context of the wider application - some functions can be used to exploit application-specific features. The examples to follow will use template injection to trigger arbitrary object creation, arbitrary file read/write, remote file include information disclosure and privilege escalation vulnerabilities.

## How is that exploitable?

After an attacker figures out template injection, then what? The template evaluation happens on the server side. Meaning if the attacker somehow finds a way to make the template access the underlying operating system, the user can take over the server.

Injecting direct os commands like ls or even using Python OS module:

* {{ ls }}
* {{ import os; os.system("ls") }}
* {{ import os }}

❌ Is not going to work in jinja. And if the web developer doesn’t handle exceptions properly, the app will return an exception:

A blue and white logo

Description automatically generated

So Jinja engine limits what we can inject. If we can’t import modules, then what can we do?

If we try with adding a simple Python datatype like a string:

{{"foo"}}

✅ It gets evaluated as normal string foo.

What if we use a built-in method for string, like convert to upper case:

{{"foo".upper()}}

✅ It gets evaluated to uppercase: FOO

## Remote Code Execution

Python is an Object-Oriented Programming. It has objects, classes, class inheritance, ..etc.

Everything in Python is an object. When you create a string, try to print out its type, you will see it’s an object that belongs to class str.

foo = "myString"

print(type(foo))

<class 'str'> # output

Since everything is an object, Python by default provides some built-in methods called magic methods (which starts and ends with double underscore) such as:

\_\_init\_\_

We saw that we could access built methods (like "string".upper()).

Hence, injecting this Python snippet:

{{ "foo".\_\_class\_\_.\_\_base\_\_.\_\_subclasses\_\_()[182].\_\_init\_\_.\_\_globals\_\_['sys'].modules['os'].popen("ls").read() }}

will result with a remote code execution and the server will execute “ls” command and list back files and folders (play.py, static, template).

A screenshot of a computer

Description automatically generated

A diagram of a wiring diagram

Description automatically generated

Explanation:

* Give me the class for “foo” string, it returns:

<class 'str'>

* Give me the name of the base class. In other words, give me the parent class that child class ‘str’ inherits from, it returns:

<class 'object'>

👉 At this point, we are at class ‘object’ level.

* Give me all the child classes that inherits ‘object’ class, it returns a list:

[<class 'type'>, <class 'weakref'>, ....etc

* Give me the class that is in index #182, this class is:

<class 'warnings.catch\_warnings'>

We chose this class, because it imports Python ‘sys’ module, and from ‘sys’ we can reach out to ‘os’ module.

* Give me the class constructor (\_\_init\_\_). Then call (\_\_globals\_\_) which returns a dictionary that holds the function’s global variables. From this dictionary, we just want [‘sys’] key which points to the sys module:

<module 'sys' (built-in)>

👉 At this point, we have reached the ‘sys’ module.

* ‘sys’ has a method called modules; it provides access to many built-in Python modules. We are just interested in ‘os’:

<module 'os' from '/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/os.py'>

👉 At this point, we have reached the ‘os’ module.

* Now we can invoke any method provided from the ‘os’ module. Just like the way we do it form the Python interpreter console. So, we execute os command “ls” using popen and read the output🎉.

## How to prevent server-side template injection vulnerabilities

The best way to prevent server-side template injection is to not allow any users to modify or submit new templates. However, this is sometimes unavoidable due to business requirements.

One of the simplest ways to avoid introducing server-side template injection vulnerabilities is to always use a "logic-less" template engine, such as Mustache, unless necessary. Separating the logic from presentation as much as possible can greatly reduce your exposure to the most dangerous template-based attacks.

Another measure is to only execute users' code in a sandboxed environment where potentially dangerous modules and functions have been removed altogether. Unfortunately, sandboxing untrusted code is inherently difficult and prone to bypasses.

Finally, another complementary approach is to accept that arbitrary code execution is all but inevitable and apply your own sandboxing by deploying your template environment in a locked-down Docker container, for example.

# Server-Side Template Injection (SSTI) - Simplified

## Understanding the Difference between XSS and SSTI Attacks

Web applications are essential tools for businesses and individuals alike, but they can also be susceptible to various security vulnerabilities. Two common types of vulnerabilities are Cross-Site Scripting (XSS) and Server-Side Template Injection (SSTI). While both involve injection attacks, they target different components of a web application and can have distinct consequences.

XSS is a vulnerability that occurs when an attacker injects malicious scripts into a web application, which then get executed in the browsers of unsuspecting users. This type of attack targets the client-side of the application and aims to manipulate user interactions.

SSTI, on the other hand, focuses on the server-side of the web application and targets the templates used to dynamically generate content. In this vulnerability, an attacker injects malicious code into templates, which are later executed on the server. The primary goal of an SSTI attack is to manipulate the application's rendering process and potentially achieve remote code execution.

## What are Templates?

Templates are files that contain placeholders for dynamic data, such as HTML files with variables like {{greeting}}. The backend app replaces these placeholders with actual data, creating dynamic content to display on the web page.

## Exploiting SSTI

A diagram of a computer

Description automatically generated

### 3.1 Detect

To detect SSTI, try injecting special characters like $, {{, <%= %> into input fields and observe any differences in the server's response. Mathematical operations like ${7/0} can also help identify the vulnerability.

|  |  |  |
| --- | --- | --- |
| ${} | {{}} | <%= %> |
| ${7/0} | {{7/0}} | <%= 7/0 %> |
| ${foobar} | {{foobar}} | <%= foobar %> |
| ${7\*7} | {{7\*7}} | `` |
| #{7\*7} | \*{7\*7} |

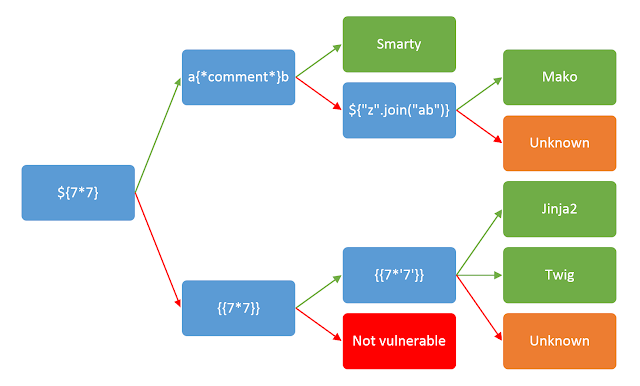
Otherwise, you'll need to manually test different language-specific payloads and study how they are interpreted by the template engine. A common way of doing this is to inject arbitrary mathematical operations using syntax from different template engines. You can then observe whether they are successfully evaluated. To help with this process, you can use a decision tree like the following:

A diagram of a computer program

Description automatically generated

You should be aware that the same payload can sometimes return a successful response in more than one template language. For example, the payload {{7\*'7'}} returns 49 in Twig and 7777777 in Jinja2. Therefore, it is important not to jump to conclusions based on a single successful response.

### Identify



Find the template engine being used by causing errors with payloads like ${foobar}. Errors may reveal the template engine.

### 3.3 Explore & Exploit

Read the template engine's documentation and find available functions and objects. Attackers can then craft payloads to exploit vulnerabilities, like arbitrary object creation or remote code execution.

## How is that exploitable?

If attackers manage to inject malicious code that executes on the server, they can gain control over it. For instance, injecting OS commands like {{ import os; os.system("ls") }} can allow them to list files on the server.

## Prevention

* Jinja’s WTF forms provide extra protection by sanitizing input, escaping characters, and preventing malicious code execution.
* Avoid user-modifiable templates whenever possible.
* Use "logic-less" template engines, like Mustache, to separate logic from presentation.
* If you must use dynamic templates, sandbox untrusted code to limit its capabilities.
* Consider deploying the template environment in a secure container, like Docker, to add an extra layer of protection.
* By following these steps, you can better understand and protect your web application from SSTI attacks. Always prioritize security to keep your users and data safe.

## Try it!

https://ssti.secure-cookie.io/

## References

- https://secure-cookie.io/attacks/ssti/

- https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection

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