

Python安全编码指南

Larry (/author/Larry) · 2015/11/12 10:58

0x00 前言

from:http://sector.ca/Portals/17/Presentations15/SecTor_Branca.pdf
(http://sector.ca/Portals/17/Presentations15/SecTor_Branca.pdf)

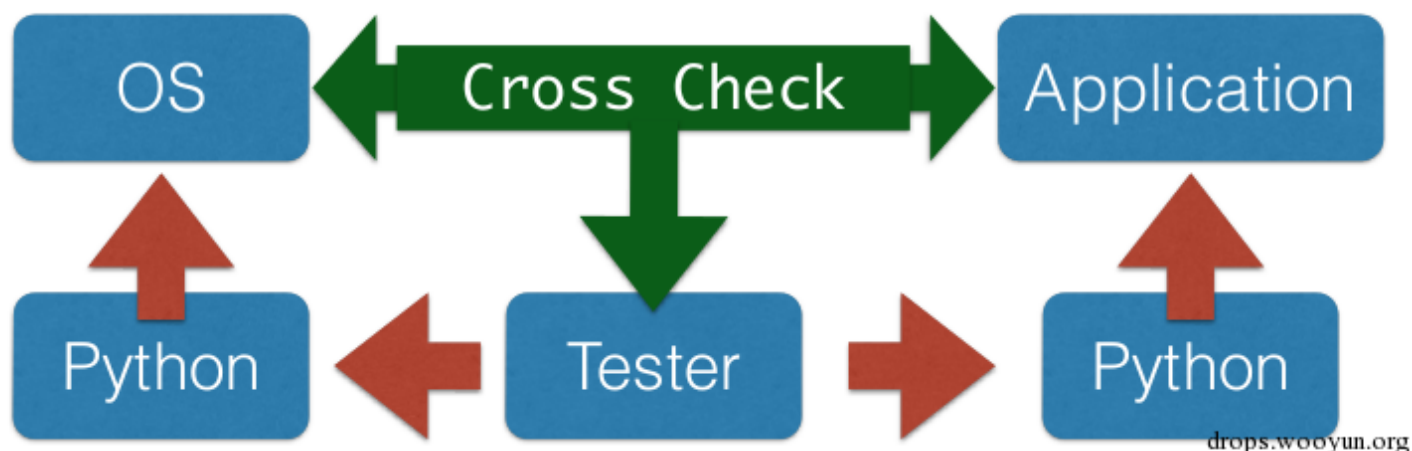
这个pdf中深入Python的核心库进行分析，并且探讨了在两年的安全代码审查过程中，一些被认为是最关键的问题，最后也提出了一些解决方案和缓解的方法。我自己也在验证探究过程中添油加醋了一点，如有错误还请指出哈。

下面一张图表示他们的方法论：

Normal approach



Our approach



探究的场景为：

- 输入的数据是"未知"的类型和大小
- 使用RFC规范构建Libraries
- 数据在没有经过适当的验证就被处理了
- 逻辑被更改为是独立于操作系统的

0x01 Date and time —> time, datetime, os

time

asctime

```
import time
initial_struct_time = [tm for tm in time.localtime()]

# Example on how time object will cause an overflow
# Same for: Year, Month, Day, minutes, seconds
invalid_time = (2**63)

# change 'Hours' to a value bigger than 32bit/64bit limit
initial_struct_time[3] = invalid_time

overflow_time = time.asctime(initial_struct_time)
1
```

这里面 `asctime()` 函数是将一个tuple或者是 `struct_time` 表示的时间形式转换成类似于 Sun Jun 20 23: 21: 05 1993 的形式，可以 `time.asctime(time.localtime())` 验证一下。对 `time.struct_time(tm_year=2015, tm_mon=11, tm_mday=7, tm_hour=20, tm_min=58, tm_sec=57, tm_wday=5, tm_yday=311, tm_isdst=0)` 中每一个键值设置 `invalid_time` 可造成溢出错误。

- 在 **Python 2.6.x** 中报错为 *OverflowError: long int too large to convert to int*
- 在 **Python 2.7.x** 中报错为
 - *OverflowError: Python int too large to convert to C long*
 - *OverflowError: signed integer is greater than maximum*

自己在64位Ubuntu Python2.7.6也测试了一下，输出结果为：

```
[ - ] hour:
[ + ] OverflowError begins at 31: signed integer is greater than maximum
[ + ] OverflowError begins at 63: Python int too large to convert to C long
...
```

gmtime

```
import time
print time.gmtime(-2**64)
print time.gmtime(2**63)
```

`time.gmtime()` 为将秒数转化为 `struct_time` 格式，它会基于 `time_t` 平台进行检验，如上代码中将秒数扩大进行测试时会产生报错 *ValueError: timestamp out of range for platform time_t*。如果数值在 -2^{63} 到 -2^{56} 之间或者 2^{55} 到 2^{62} 之间又会引发另一种报错 *ValueError: (84, 'Value too large to be stored in data type')*。我自

己的测试结果输出如下：

```
[ - ] 2 power:
[+] ValueError begins at 56: (75, 'Value too large for defined data ty
[+] ValueError begins at 63: timestamp out of range for platform time_
[ - ] -2 power:
[+] ValueError begins at 56: (75, 'Value too large for defined data ty
[+] ValueError begins at 64: timestamp out of range for platform time_
```

OS

```
import os
TESTFILE = 'temp.bin'

validtime = 2**55
os.utime(TESTFILE, (-2147483648, validtime))
stinfo = os.stat(TESTFILE)
print(stinfo)

invalidtime = 2**63
os.utime(TESTFILE, (-2147483648, invalidtime))
stinfo = os.stat(TESTFILE)
print(stinfo)
2
```

这里的 `os.utime(path, times)` 是设置对应文件的access和modified时间，时间以 `(atime, mtime)` 元组的形式传入，代码中将modified time设置过大也会产生报错。

- 在**Python 2.6.x**中报错为*OverflowError: long int too large to convert to int*
- 在**Python 2.7.x, Python 3.1**中报错为*OverflowError: Python int too large to convert to C long*

如果我们将其中的modified time设置为 2^{55} ，ls 后会有：

```
$ ls -la temp.bin
-rw-r--r-- 1 user01 user01 5 13 Jun 1141709097 temp.bin
$ stat temp.bin
A:"Oct 10 16:31:45 2015"
M:"Jun 13 01:26:08 1141709097"
C: "Oct 10 16:31:42 2015"
```

在某些操作系统上如果我们将值设为 2^{56} ，将会有以下输出（也有造成系统崩溃和数据丢失的风险）：

```
$ ls -la temp.bin
Segmentation fault: 11
$ stat temp.bin
A:"Oct 10 16:32:50 2015"
M:"Dec 31 19:00:00 1969"
C:"Oct 10 16:32:50 2015"
```

Modules通常没有对无效输入进行检查或者测试。例如，对于64位的操作系统，最大数可以达到 $2^{63}-1$ ，但是在不同的情况下使用数值会造成不同的错误，任何超出有效边界的数字都会造成溢出，所以要对有效的数据进行检验。

0x02 Numbers —> ctypes, xrange, len, decimal

ctype

ctypes是Python的一个外部库，提供和C语言兼容的数据类型,具体可见官方文档(<https://docs.python.org/2/library/ctypes.html#fundamental-data-types>)

测试代码：

```
import ctypes

#32-bit test with max 32bit integer 2147483647
ctypes.c_char * int(2147483647)

#32-bit test with max 32bit integer 2147483647 + 1
ctypes.c_char * int(2147483648)

#64-bit test with max 64bit integer 9223372036854775807
ctypes.c_char * int(9223372036854775807)

#64-bit test with max 64bit integer 9223372036854775807 + 1
ctypes.c_char * int(9223372036854775808)
3
```

举个栗子，可以在64位的操作系统上造成溢出：

```
>>> ctypes.c_char * int(9223372036854775808)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: cannot fit 'long' into an index-sized integer
```

Python ctypes 可调用的数据类型有：

Python ctypes calls

c_byte	c_char	c_char_p	c_double
c_longdouble	c_float	c_int	c_long
c_longdouble	c_longlong	c_short	c_wchar_p
c_void_p			

drops.wooyun.org

问题在于：

- ctypes对内存大小没有限制
- 也没有对溢出进行检查

所以，在32位和64位操作系统上都可以造成溢出，解决方案就是也要对数据的有效性和溢出进行检查。

xrange()

演示代码：

```
valid = (2 ** 63) - 1
invalid = 2 ** 63

for n in xrange(invalid):
    print n
```

报错为：*OverflowError: Python int too large to convert to C long*。虽然这种行为是“故意”的和在预期之内的，但在这种情况下依旧没有进行检查而导致数字溢出，这是因为 `xrange` 使用 `Plain Integer Objects` 而无法接受任意长度的对象。解决方法就是使用Python的 `long integer object`，这样就可以使用任意长度的数字了，限制条件则变为操作系统内存的大小了。

len()

演示代码：

```
valid = (2**63)-1
invalid = 2**63

class A(object):
    def __len__(self):
        return invalid

print len(A())
```

这里也会报错：*OverflowError: long int too large to convert to int*。因为 `len()` 函数没有对对象的长度进行检查，也没有使用python int objects（使用了就会没有限制），当对象可能包含一个“`.length`”属性的时候，就有可能造成溢出错误。解决办法同样也是使用python int objects。

Decimal

```
from decimal import Decimal
try:
    # DECIMAL '1172837167.27'
    x = Decimal("1172837136.0800")
    # FLOAT '1172837167.27'
    y = 1172837136.0800
    if y > x:
        print("ERROR: FLOAT seems comparable with DECIMAL")
    else:
        print("ERROR: FLOAT seems comparable with DECIMAL")
except Exception as e:
    print("OK: FLOAT is NOT comparable with DECIMAL")
2
```

以上代码是将Decimal (<https://docs.python.org/2/library/decimal.html#decimal-objects>)实例和浮点值进行比较，在不同Python版本中如果无法比较则用except捕获异常，输出情况为：

- 在**Python 2.6.5, 2.7.4, 2.7.10**中输出*ERROR: FLOAT seems comparable with DECIMAL (WRONG)*
- 在**Python 3.1.2**中输出*OK: FLOAT is NOT comparable with DECIMAL (CORRECT)*

Type Comparsion

```
try:
    # STRING 1234567890
    x = "1234567890"
    # FLOAT '1172837167.27'
    y = 1172837136.0800
    if y > x:
        print("ERROR: FLOAT seems comparable with STRING")
    else:
        print("ERROR: FLOAT seems comparable with STRING")
except Exception as e:
    print("OK: FLOAT is NOT comparable with STRING")
1
```

以上代码是将字符串和浮点值进行比较，在不同Python版本中如果无法比较则用**except**捕获异常，输出情况为：

- 在**Python 2.6.5, 2.7.4, 2.7.10**中输出*ERROR: FLOAT seems comparable with STRING (WRONG)*
- 在**Python 3.1.2**中输出*OK: FLOAT is NOT comparable with STRING (CORRECT)*

在使用同一种类型的对象进行比较之后，Python内置的比较函数就不会进行检验。但在以上两个代码例子当中Python并不知道该如何把STRING和FLOAT进行比较，就会直接返回一个FALSE而不是产生一个Error。同样的问题也发生于在将DECIMAL和FLOATS时。解决方案就是使用强类型（strong type）检测和数据验证。

0x03 Strings —> input, eval, codecs, os, ctypes

eval()

```
import os
try:
    # Linux/Unix
    eval("__import__('os').system('clear')", {})
    # Windows
    #eval("__import__('os').system(cls')", {})
    print "Module OS loaded by eval"
except Exception as e:
    print repr(e)
```

关于 eval () 函数，Python中eval带来的潜在风险 (<http://drops.wooyun.org/tips/7710>)这篇文章也有提到过，使用 `__import__` 导入 os ,再结合eval()就可以执行命令了。只要用户加载了解释器就可以没有限制地执行任何命令。

input()

```
Secret = "42"
```

```
value = input("Answer to everything is ? ")

print "The answer to everything is %s" % (value,)
```

在以上的代码中()会接受原始输入，如何这里用户传入一个dir()再结合print，就会执行dir()的功能返回一个对象的大部分属性：

```
Answer to everything is ? dir()
The answer to everything is
['Secret', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__']
```

我在这里看到了有一个**Secret**对象，然后借助原来程序的功能就可以得到该值：

```
Answer to everything is ? Secret
The answer to everything is 42
```

codecs

```
import codecs
import io

b = b'\x41\xF5\x42\x43\xF4'
print("Correct-String %r") % ((repr(b.decode('utf8', 'replace'))))

with open('temp.bin', 'wb') as fout:
    fout.write(b)
with codecs.open('temp.bin', encoding='utf8', errors='replace') as fin:
    print("CODECS-String %r") % (repr(fin.read()))
with io.open('temp.bin', 'rt', encoding='utf8', errors='replace') as fin:
    print("IO-String %r") % (repr(fin.read()))
2
```

以上的代码将 \x41\xF5\x42\x43\xF4 以二进制的形式写入文件，再分别用 codecs 和 io 模块进行读取，编码形式为utf-8，对 \xF5 和 \xF4 不能编码的设置 errors='replace'，编码成为 \ufffd，最后结果如下：

```
Correct-String → "u' A\\ufffdBC\\ufffd' "
CODECS-String → "u' A\\ufffdBC' " (WRONG)
IO-String → "u' A\\ufffdBC\\ufffd' " (OK)
```

当codecs在读取 \x41\xF5\x42\x43\xF4 这个字符串的时候，它期望接收到包含4个字节的序列，而且因为在读入 \xF4 的时候它还会再等待其他3个字节，而没有进行编码，结果就是得到的字符串有一段被删除了。更好且安全的方法就是使用 os 模块，读取整个数据流，然后进行解码处理。解决方案就是使用 io 模块或者对字符串进行识别和确认来检测畸形字符。

OS

```
import os
os.environ['a=b'] = 'c'
```

```
try:
    os.environ.clear()
    print("PASS => os.environ.clear removed variable 'a=b'")
except:
    print("FAIL => os.environ.clear removed variable 'a=b'")
    raise
```

在不同的平台上，环境变量名的名称和语法都是基于不同的规则。但Python并不遵守同样的逻辑，它尽量使用一种普遍的接口来兼容大多数的操作系统。这种重视兼容性大于安全的选择，使得用于环境变量的逻辑存在缺陷。

```
$ env -i =value python -c 'import pprint, os;
pprint.pprint(os.environ); del os.environ[""]'

environ({'': 'value'})
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "Lib/os.py", line 662, in __delitem__
    self.unsetenv(encodedkey)
OSError: [Errno 22] Invalid argument
```

上面的代码使用 `env -i` 以一个空的环境开始，再设置一个键为空值为value的环境变量，使用python打印出来再删除。这样就可以定义一个键为空的环境变量了，也可以设置在键名中包含"="，但是会无法移除它：

```
$ env -i python -c 'import pprint, posix, os;
os.environ["a="]="1"; print(os.environ); posix.unsetenv("a=")'

environ({'a=': '1'})
Traceback (most recent call last):
  File "<string>", line 1, in <module>
OSError: [Errno 22] Invalid argument
```

根据不同的版本，Python也会有不同的反应：

- Python 2.6 —> **NO ERRORS**，允许无效操作！
- PYTHON 2.7 —> *OSError: [Errno 22] Invalid argument*
- PYTHON 3.1 —> **NO ERRORS**，允许无效操作！

解决方案是对基础设施和操作系统进行检测，检测和环境变量相关的键值对，阻止一些对操作系统为空或者无效键值对的使用。

ctypes

```
buffer=ctypes.create_string_buffer(8)

buffer.value='a\0bc1234'

print "Original value => %r" % (buffer.raw,)
print "Interpreted value => %r" % (buffer.value,)
```

ctypes模块在包含空字符的字符串中会产生截断，上面代码输出如下：

Original value => 'a\x00bc1234'

Interpreted value => 'a'

这一点和C处理字符串是一样的，会把空字符作为一行的终止。Python在这种情况下使用 `ctypes`，就会继承相同的逻辑，所以字符串就被截断了。解决方案就是对数据进行确认，删除字符串中的空字符来保护字符串或者是禁止使用 `ctypes`。

Python Interpreter

```
try:
    if 0:
        yield 5
    print("T1-FAIL")
except Exception as e:
    print("T1-PASS")
    pass

try:
    if False:
        yield 5
    print("T2-FAIL")
except Exception as e:
    print(repr(e))
    pass

5
```

以上的测试代码应该返回一个语法错误: *SyntaxError: 'yield' outside function*。在不同版本的Python上运行结果如下:

Python Version	Result Test 1	Result Test 2
2.6.5	<nothing>	ERROR
2.7.4	T1-FAIL	ERROR
2.7.10	ERROR	ERROR
3.1.4	T1-FAIL	T2-FAIL drops.wooyun.org

这个问题在最新的Python 2.7.x版本中已经解决，而且避免使用像"if 0:"，"if False:"，"while 0:"，"while False:"之类的结构。

0x04 Files —> sys, os, io, pickle, cpickl

pickle

```

import pickle
import io
badstring = "cos\nsystem\n(S'ls -la /\n'\nR."
badfile = "./pickle.sec"
with io.open(badfile, 'wb') as w:
    w.write(badstring)
obj = pickle.load(open(badfile))
print "== Object =="
print repr(obj)

```

这里构造恶意序列化字符串，以二进制的形式写入文件中，使用 `pickle.load()` 函数加载进行反序列化，还原出原始python对象，从而使用os的 `system()` 函数来执行命令 `"ls -la /"`。由于 `pickle` 这样不安全的设计，就可以借此来执行命令了。代码输出结果如下：

- Linux

```

total 104
drwxr-xr-x 23 root root 4096 Oct 20 11:19 .
drwxr-xr-x 23 root root 4096 Oct 20 11:19 ..
drwxr-xr-x 2 root root 4096 Oct 4 00:05 bin
drwxr-xr-x 4 root root 4096 Oct 4 00:07 boot
...

```

- Mac OS X

```

total 16492
drwxr-xr-x 31 root wheel 1122 12 Oct 18:58 .
drwxr-xr-x 31 root wheel 1122 12 Oct 18:58 ..
drwxrwxr-x+ 122 root wheel 4148 10 Oct 15:19 Applications
drwxr-xr-x+ 68 root wheel 2312 3 Sep 10:47 Library
...

```

pickle / cPickle

```

import cPickle
import traceback
import sys
# bignum = int((2**31)-1) # 2147483647 -> OK
bignum = int(2**31) # 2147483648 -> Max 32bit -> Crash
random_string = os.urandom(bignum)
print ("STRING-LENGTH-1=%r") % (len(random_string))
fout = open('test.pickle', 'wb')
try:
    cPickle.dump(random_string, fout)
except Exception as e:
    print "##### ERROR-WRITE #####"
    print sys.exc_info()[0]
    raise
fout.close()
fin = open('test.pickle', 'rb')
try:
    random_string2 = cPickle.load(fin)

```

```

except Exception as e:
    print "##### ERROR-READ #####"
    print sys.exc_info()[0]
    raise
print ("STRING-LENGTH-2=%r") % (len(random_string2))
print random_string == random_string2
sys.exit(0)
5

```

在上面的代码中，根据使用的Python版本不同，`pickle` 或 `cPickle` 要么保存截断的数据而没有错误要么就会保存限制为32bit的部分。而且根据Python在操作系统上安装时编译的情况，它会返回在请求随机数据大小上的错误，或者是报告无效参数的OS错误：

- `cPickle` (debian 7 x64)

```

STRING-LENGTH-1=2147483648
##### ERROR-WRITE #####
<type 'exceptions.MemoryError'>
Traceback (most recent call last):
....
    pickle.dump(random_string, fout)
SystemError: error return without exception set

```

- `pickle` (debian 7 x64)

```

STRING-LENGTH-1=2147483648
##### ERROR-WRITE #####
<type 'exceptions.MemoryError'>
Traceback (most recent call last):
....
File "/usr/lib/python2.7/pickle.py", line 488,
in save_string
self.write(STRING + repr(obj)+ '\n')
MemoryError

```

解决方案就是执行强大的数据检测来确保不会执行危险行为，还有即使在64位的操作系统上也要限制数据到32位大小。

File Open

```

import os
import sys
FPATH = 'bug2091.test'
# =====
print 'wa (1)_write1'
with open(FPATH, 'wa') as fp:
    fp.write('test1-')
with open(FPATH, 'rb') as fp:
    print repr(fp.read())
# =====
print 'rU+_write2'
with open(FPATH, 'rU+') as fp:
    fp.write('test2-')
with open(FPATH, 'rb') as fp:
    print repr(fp.read())
# =====
print 'wa (2)_write3'

```

```

with open(FPATH, 'wa+') as fp:
    fp.write('test3-')
with open(FPATH, 'rb') as fp:
    print repr(fp.read())
# =====
print 'aw_write4'
with open(FPATH, 'aw') as fp:
    fp.write('test4-')
with open(FPATH, 'rb') as fp:
    print repr(fp.read())
# =====
print 'rU+_read1',
with open(FPATH, 'rU+') as fp:
    print repr(fp.read())
# =====
print 'read_2',
with open(FPATH, 'read') as fp:
    print repr(fp.read())
# =====
os.unlink(FPATH)
sys.exit(0)
8

```

以上代码主要是测试各种文件的打开模式，其中 U 是指以统一的换行模式打开（不赞成使用），各个平台的测试结果如下：

- Linux and Mac OS X

Test String	Flags	Operation	Expected Result	Test Result (LINUX-OS X)
test1-	wa	1. truncate and write 2. write in append mode	Invalid Mode	test1-
test2-	rU+	1. read (Universal Newline) 2. open file in read and write	test2-	test2-
test3-	wa+	1. truncate and write 2. write in append mode	Invalid Mode	test3-
test4-	aw	1. write in append mode 2. truncate and write	Invalid Mode	test3-test4-
	rU+	1. read (Universal Newline) 2. open file in read and write	test2-	test3-test4-
	read	read, ?, append, ?	Invalid Mode	test3-test4- drops.wooyun.org

- Windows

Test String	Flags	Operation	Expected Result	Test Result (WINDOWS)
test1-	wa	1. truncate and write 2. write in append mode	Invalid Mode	Invalid Mode
test2-	rU+	1. read (Universal Newline) 2. open file in read and write	test2-	test2-
test3-	wa+	1. truncate and write 2. write in append mode	Invalid Mode	Invalid Mode
test4-	aw	1. write in append mode 2. truncate and write	Invalid Mode	Invalid Mode
	rU+	1. read (Universal Newline) 2. open file in read and write	test2-	test2-
	read	read, ?, append, ?	Invalid Mode	Invalid Mode drops.wooyun.org

INVALID stream operations - Linux / OS X

```
import sys
import io
fd = io.open(sys.stdout.fileno(), 'wb')
fd.close()
try:
    sys.stdout.write("test for error")
except Exception:
    raise
```

代码在这里使用`fileno()` (<https://docs.python.org/2/library/stdtypes.html?highlight=fileno#file.fileno>)来获取 `sys.stdout` 的文件描述符，在读写后就关闭，之后便无法从标准输入往标准输出中发送数据流了。输出如下：

- 在**Python 2.6.5, 2.7.4**中

```
close failed in file object destructor:
sys.excepthook is missing
lost sys.stderr
```

- 在**Python 2.7.10**中

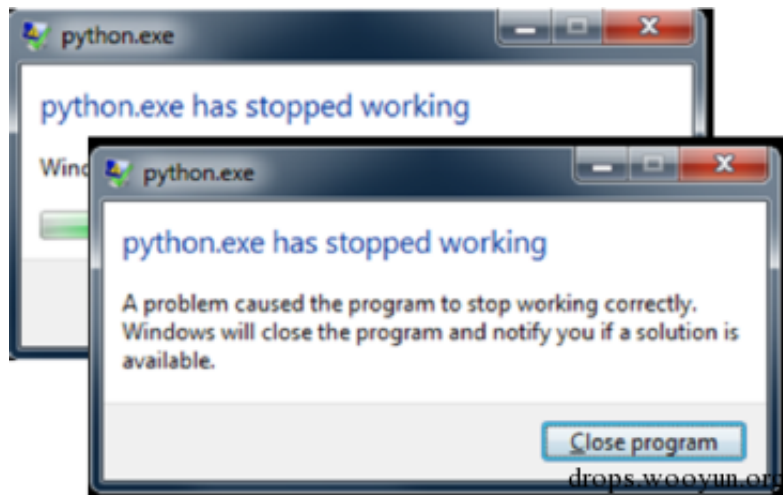
```
Traceback (most recent call last):
  File "tester.py", line 6, in <module>
    sys.stdout.write("test for error")
IOError: [Errno 9] Bad file descriptor
```

INVALID stream operations - Windows

```
import io
import sys

fd = io.open(sys.stdout.fileno(), 'wb')
fd.close()
sys.stdout.write("Crash")
```

在windows上也是类似的，如图：



解决方案就是file和stream库虽然不遵循OS规范，但它们使用一个通用的逻辑，有必要为每个OS使用有处理能力的库，来设置正确的调用过程。

File Write

```
import os
import sys
testfile = 'tempA'
with open(testfile, "ab") as f:
    f.write(b"abcd")
    f.write(b"x" * (1024 ** 2))
#####
import io
testfilea = 'tempB'
with io.open(testfilea, "ab") as f:
    f.write(b"abcd")
    f.write(b"x" * (1024 ** 2))
2
```

我们在Linux上使用 `strace python -OOBrttu script.py` 来检测Python的写文件行为：

在这里我们想要写入的字符数目是 $4 + 1048576 = 1048580$ ，在不同的版本上对调用 `open()` 和使用 `io` 模块进行比较：

- **PYTHON 2.6**

- 调用 `open()` 的输出为：

[illegible]

第一次调用的时候被缓冲，不仅仅是写入了4个字符（abcd），还写入了4092个x；第2次调用总共写入1044480个x。这样加起来 $1044480 + 4096 = 1.048.576$ ，相比1048580就少了4个x。等待5秒就可以解决这个问题，因为操作系统flush了缓存。

- 调用 i o 模块的输出为:

[illegible]

这样一切就很正常

- **PYTHON 2.7**

- 用 `open()` 的输出为:

```
write(3, "abcdxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx", ..., 4096) = 4.096
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx", ..., 1044480) = 1.044.480
write(3, "xxxx", 4) = 4
```

在这里进行了三次调用，最后再写入4个 **x**，保证整体数据的正确性。问题就在于这里使用了**3**次调用而不是我们预期的**2**次调用。

- 调用 i o 模块则一切正常

- **PYTHON 3.x**

在Python3中用 `open()` 函数和 `io` 模块则一切都很正常

在Python2中没有包含原子操作，核心库是在使用缓存进行读写。所以应该尽量去使用 `i o` 模块。

0x05 Protocols —> socket, poplib, urllib, urllib2

httplib, smtplib, ftplib...

核心库是独立于操作系统的，开发者必须要知道如何为每一个操作系统构建合适的通信通道，而且这些库将会运行执行那些不安全且不正确的操作

```
import SimpleHTTPServer
httplib, smtplib, ftplib...
import SocketServer
PORT = 45678
def do_GET(self):
```

```

        self.send_response(200)
        self.end_headers()
Handler = SimpleHTTPServer.SimpleHTTPRequestHandler
Handler.do_GET = do_GET
httpd = SocketServer.TCPServer("", PORT), Handler)
httpd.serve_forever()
1

```

在上面的代码中构造了一个HTTP服务端，如果一个客户端连接进来，再去关闭服务端，Python将不会释放资源，操作系统也不会释放socket，引发报错为`socket.error: [Errno 48] Address already in use`。可以通过以下代码来解决：

```

import socket
import SimpleHTTPServer
import SocketServer
PORT = 8080
# ESSENTIAL: socket reuse is setup BEFORE it is bound.
# This will avoid TIME_WAIT issues and socket in use errors
class MyTCPServer(SocketServer.TCPServer):
    def server_bind(self):
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.socket.bind(self.server_address)
def do_GET(self):
    self.send_response(200)
    self.end_headers()
Handler = SimpleHTTPServer.SimpleHTTPRequestHandler
Handler.do_GET = do_GET
httpd = MyTCPServer("", PORT), Handler)
httpd.serve_forever()
6

```

解决方案就是每一个协议库都应该由这样的库封装：为每一个OS和协议都适当地建立和撤销通信，并释放资源

poplib, httplib ...

服务端：

```

import socket
HOST = '127.0.0.1'
PORT = 45678
NULLS = '\0' * (1024 * 1024) # 1 MB
try:
    sock = socket.socket()
    sock.bind((HOST, PORT))
    sock.listen(1)
    while 1:
        print "Waiting connection..."
        conn, _ = sock.accept()
        print "Sending welcome..."
        conn.sendall("+OK THIS IS A TEST\r\n")
        conn.recv(4096)
        DATA = NULLS
        try:
            while 1:
                print "Sending 1 GB..."
                for _ in xrange(1024):
                    conn.sendall(DATA)
        except IOError, ex:
            print "Error: %r" % str(ex)

```



```

        print "End session."
    print
finally:
    sock.close()
print "End server."
7

```

客户端:

```

import poplib
import sys
HOST = '127.0.0.1'
PORT = 45678
try:
    print "Connecting to %r:%d..." % (HOST, PORT)
    pop = poplib.POP3(HOST, PORT)
    print "Welcome:", repr(pop.welcome)
    print "Listing..."
    reply = pop.list()
    print "LIST:", repr(reply)
except Exception, ex:
    print "Error: %r" % str(ex)
print "End."
sys.exit(0)
5

```

以上代码当中，首先开启一个虚拟的服务端，使用客户端去连接服务端，然后服务端开始发送空字符，客户端持续性接收空字符，最后到客户端内存填满，系统崩溃，输出如下：

- 服务端

```

Waiting connection...
Sending welcome...
Sending 1 GB...
Error: '[Errno 54] Connection reset by peer'
End session.

```

- 客户端

- **Python >= 2.7.9, 3.3**

```

Connecting to '127.0.0.1':45678...
Welcome: '+OK THIS IS A TEST'
Listing...
Error: 'line too long'
End.

```

- **Python < 2.7.9, 3.3**

```

Client!
Connecting to '127.0.0.1':45678...
Welcome: '+OK THIS IS A TEST'
.....
Error: 'out of memory'

```

解决方案就是如果无法控制检查数据的类型和大小，就使用Python > 2.7.9'或者'Python > 3.3'的版本

对数据没有进行限制的库：

Library	Link to Python bug
HTTPLIB	http://bugs.python.org/issue16037
FTPLIB	http://bugs.python.org/issue16038
IMAPLIB	http://bugs.python.org/issue16039
NNTPLIB	http://bugs.python.org/issue16040
POPLIB	http://bugs.python.org/issue16041
SMTPLIB	http://bugs.python.org/issue16042
XMLRPC	http://bugs.python.org/issue16043

drops.wooyun.org

urllib, urllib2

```
import io
import os
import urllib2 #but all fine with urllib
domain = 'ftp://ftp.ripe.net'
location = '/pub/stats/ripencc/'
file = 'delegated-ripencc-extended-latest'
url = domain + location + file
data = urllib2.urlopen(url).read()
with io.open(file, 'wb') as w:
    w.write(data)
file_size = os.stat(file).st_size
print "Filesize: %s" % (file_size)
2
```

`urllib2` 并没有合适的逻辑来处理数据流而且每次都会失败，将上次代码运行三次都会得到错误的文件大小的输出：

```
Filesize: 65536
Filesize: 32768
Filesize: 49152
```

如果使用以下的代码则会产生正确的输出：

```
import os
import io
import urllib2
domain = 'ftp://ftp.ripe.net'
location = '/pub/stats/ripencc/'
```

```

file = 'delegated-ripenc-extended-latest'
with io.open(file, 'wb') as w:
    url = domain + location + file
    response = urllib2.urlopen(url)
    data = response.read()
    w.write(data)
file_size = os.stat(file).st_size
print "Filesize: %s" % (file_size)
3

```

输出为:

File size: 6598450
File size: 6598450
File size: 6598450

通过以上的例子可以看出，解决方案为利用操作系统来保证数据流的正确性

已知不安全的库:

ast	multiprocessing	rexec
bastion	os.exec	shelve
commands	os.popen	subprocess
cookie	os.spawn	tarfile
cPickle / pickle	os.system	urllib2
eval	parser	urlparse
marshal	pipes	yaml
mktemp	pty	zipfile

drops.wooyun.org

最后，当数百万人在使用它的时候，永远不要以为它会一直按你期望的那样运作，也绝对不要以为在使用它的时候是安全的