Python Security

Introduction to Python Secure Coding
Deep dive into Python’s core libraries.

We will talk about some of the most critical issues that have been identified during a two year security code review.

Each issue will be analyzed and when possible we will provide a solution or a mitigation strategy.
Proper state of mind...

<table>
<thead>
<tr>
<th>If we expect ..</th>
<th>but we get ..</th>
<th>for DEVELOPER this is..</th>
<th>but for SECURITY this is..</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>PASS</td>
<td>GOOD</td>
<td>USELESS</td>
</tr>
<tr>
<td>PASS</td>
<td>FAIL</td>
<td>BAD</td>
<td>GOOD</td>
</tr>
<tr>
<td>FAIL</td>
<td>PASS</td>
<td>VERY BAD</td>
<td>VERY GOOD</td>
</tr>
<tr>
<td>FAIL</td>
<td>FAIL</td>
<td>GOOD</td>
<td>USELESS</td>
</tr>
</tbody>
</table>
DATE and TIME

time, os
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)

Python 2.6.x
OverflowError: long int too large to convert to int

Python 2.7.x
OverflowError: Python int too large to convert to C long
OverflowError: signed integer is greater than maximum
“time.gmtime” has a check against platform time_t

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

ValueError: timestamp out of range for platform time_t

But if value is between (-2^63) and (-2^56) or is between (2^55) to (2^62) then another type error is generated

import time
print time.gmtime(-2**63)
print time.gmtime(2**62)

ValueError: (84, 'Value too large to be stored in data type')
```python
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)
```

Python 2.6.x,
OverflowError: long int too large to convert to int

Python 2.7.x, Python 3.1
OverflowError: Python int too large to convert to C long
But in some systems we can also have **OS** related issues:

```
$ 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"
```

!! WARNING !!
RISK OF SYSTEM CRASH
RISK OF DATA LOSS

Do **NOT** play with “os” module.
Modules do **not** include exhaustive tests for edge cases.

The maximum value for a 64bit system would be $[2^{63}-1]$, but different errors will be generated depending on the used values.

Any number outside the valid range will generate an Overflow.

**SOLUTION**

Implement proper data validation.
NUMBERS

cypes, xrange, len, decimal
Example of overflow message in a 64bit system:

```python
>>> 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
```
• ctypes are not limited to size of memory
• overflow checks are mostly missing.

An overflow will occur in both 32bit and 64bit systems.

**SOLUTION**
Implement Overflow checking and data validation.
OverflowError: Python int too large to convert to C long

This happens because `xrange` uses "Plain Integer Objects" and cannot accept objects of arbitrary length.

**SOLUTION**
Create function that uses python only "long integer object".
len() does not check for the length of the object and does not use "python int objects" (unlimited). This can cause an Overflow error as the object may contain a ".length" property.

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

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

print len(A())
```

**OverflowError: long int too large to convert to int**

SOLUTION

Use python “python int objects” that will allow numbers of arbitrary length as the limit will be the system's memory.
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")

Python 2.6.5, 2.7.4, 2.7.10
ERROR: FLOAT seems comparable with DECIMAL (WRONG)

Python 3.1.2
OK: FLOAT is NOT comparable with DECIMAL (CORRECT)
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")

Python 2.6.5, 2.7.4, 2.7.10
ERROR: FLOAT seems comparable with STRING (WRONG)

Python 3.1.2
OK: FLOAT is NOT comparable with STRING (CORRECT)
Python does not know how to compare STRING and FLOAT and instead of returning an Error returns a FALSE.

Same problem if we try to compare DECIMAL and FLOATS, python does not know how to compare this objects and returns a FALSE instead of returning an Error.

**SOLUTION**
Implement strong type checking and perform data validation.
STRINGS

input, eval, codecs, os, ctypes
How do I correctly pass the string “Null” (an employee’s proper surname) to a SOAP web service from ActionScript 3?

We have an employee whose last name is Null. Our employee lookup application is killed when that last name is used as the search term (which happens to be quite often now). The error received (thanks Fiddler!) is:

```xml
<soapenv:Fault>
  <faultcode>soapenv:Server.userException</faultcode>
  <faultstring>coldfusion.xml.rpc.CFCInvocationException: [coldfusion.runtime.Missi
```

Cute, huh?

The parameter type is string.

I am using:

- WSDL (SOAP).
- Flex 3.5
- ActionScript 3
- ColdFusion 8

Note that the error DOES NOT occur when calling the webservice as an object from a ColdFusion page.

How bad it can be….

http://cdn.inquisitr.com/wp-content/uploads/2015/05/iphone-crash.jpg

http://hubpages.com/autos/10-fun-facts-us-license-plates
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)

Any code will be executed without limits in the context of the user that loaded the interpreter.
Answer to everything is ? dir()
The answer to everything is
['Secret', '__builtins__', '__doc__', '__file__', '__name__', '__package__']

The dir() function returns “most” of the attributes of an object, and as a result we obtain the “Secret” object.

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

Python 2.x
Use raw_input()

Python 3.x
Not vulnerable
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()))

- **Expected UNICODE:**
  - Two characters, each of 4 bytes

- **Test UNICODE:**
  - **One valid** character (4 bytes), **one invalid** character (1 byte)
Read by the OS:
\[
\text{read(3, "A\365BC\364", 8192)} = 5
\]

Read by the Python:
\[
u'A\ufffdBC\ufffd'
\]

The original string will be silently truncated at the \textbf{first} byte.

Correct-String $\rightarrow$ “u'A\ufffdBC\ufffd''
CODECS-String $\rightarrow$ “u'A\ufffdBC'' \quad \text{(WRONG)}$
IO-String $\rightarrow$ “u'A\ufffdBC\ufffd'' \quad \text{(OK)}$

\textbf{SOLUTION}

Either use the “io” module or implement string recognition and validation to detect malformed characters.
Names and syntax of environment variables names are also based on the specific rules used in each platform.

Python does not share the same logic and tried to implement a generic interface compatible with most operating systems.

This choice of **preferring compatibility over security** have allowed the existence of cracks in the logic used for environment variables.
It is possible to define an environment variable with an empty key, or a variable that contains "="", but not to remove it.

```
$ 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
```

```
$ 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
```
Python behaviour changes, depending on the version:

- Python 2.6 —> NO ERRORS, allows invalid operations !
- PYTHON 2.7 —> OSError: [Errno 22] Invalid argument
- PYTHON 3.1 —> NO ERRORS, allows invalid operations !

**SOLUTION**

Implement a solution to detect architecture and OS, then for each case prevent the usage of 'key-value' pairs associated to environment variable that are empty or invalid for several OS.
```python
import ctypes
buffer=ctypes.create_string_buffer(8)
buffer.value='a\0bc1234'
print "Original value    => %r" % (buffer.raw,)
print "Interpreted value => %r" % (buffer.value,)
```

The ctypes module **truncates** NUL-containing strings.

Original value    => 'a\x00bc1234'
Interpreted value => 'a'

This behaviour is consistent with how C handles string, by considering a NUL character as a line terminator. Python in this case, by using ctypes, is inheriting the same logic therefore the string is silently truncated.

**SOLUTION**

Implement data validation to detect NUL-containing strings to protect them, or avoid using ctypes.
try:
    if 0:
        yield 5
        print("NO-ERR")
    print("PASS")
pass

try:
    if False:
        yield 5
        print("NO-ERR")
    print(repr(e))
pass

Python Interpreter

Test should return syntax error like:
SyntaxError: 'yield' outside function

<table>
<thead>
<tr>
<th>Python Version</th>
<th>Result Test 1</th>
<th>Result Test 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.5</td>
<td>&lt;nothing&gt;</td>
<td>ERROR</td>
</tr>
<tr>
<td>2.7.4</td>
<td>NO-ERR</td>
<td>ERROR</td>
</tr>
<tr>
<td>2.7.10</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>3.1.4</td>
<td>NO-ERR</td>
<td>NO-ERR</td>
</tr>
</tbody>
</table>

**SOLUTION**

Solved in latest Python 2.7.x, avoid constructs like “if 0:”, “if False:”, “while 0:” “while False:”.

30
FILES

sys, os, io, pickle, cpickle
We are asking to pickle to load a string specially formatted that makes it executable by python.

Pickle is **NOT** designed to be safe/secure, we can make it execute **whatever we want**.

Pickle loads the string and by processing it executes "ls -la /".
Result of pickle crafted string

Linux

total 104
-rwxr-xr-x  24 root root  4096 Feb 28  01:42 .
-rwxr-xr-x  24 root root  4096 Feb 28  01:42 ..
-drwxr-xr-x   2 root root  4096 Feb 28  01:14 bin
-drwxr-xr-x   3 root root  4096 Feb 28  01:57 boot
-drwxr-xr-x  14 root root  3680 May  2 14:28 dev
-drwxr-xr-x 158 root root 12288 Apr 30 22:16 etc
-drwxr-xr-x   3 root root  4096 Feb 28  00:45 home
-lrwxrwxrwx   1 root root    30 Feb 27 23:29 initrd.img -> /boot/initrd.img-3.2.0-4-amd64
-drwxr-xr-x  18 root root  4096 Feb 28  01:54 lib
-drwxr-xr-x  31 root root  1122  12 Oct 18:58 Applications
-drwxr-xr-x+  68 root root  2312  03 Sep 10:47 Library
-drwxr-xr-x+   2 root root     68  24 Aug  2013 Network
-drwxr-xr-x+   4 root root  136  13 Jul  07:28 System
-drwxr-xr-x    7 root admin 238  08 Oct 11:23 Users
-drwxrwxrwt@   5 root admin 170  14 Oct 10:41 Volumes
-drwxr-xr-x@   39 root wheel 1326  13 Jul 14:14 bin
-drwxrwxr-t@   2 root admin  68  24 Aug 2013 cores
-dr-xr-xr-x    3 root wheel  7937  12 Oct 18:57 dev

Mac OS X

total 16492
-drwxr-xr-x   31 root wheel  1122  12 Oct 18:58 ..
-drwxrwxr-x+  122 root admin  4148  10 Oct 15:19 Applications
-drwxr-xr-x+   68 root wheel  2312  03 Sep 10:47 Library
-drwxr-xr-x@   2 root wheel   68  24 Aug  2013 Network
-drwxr-xr-x+   4 root wheel  136  13 Jul  07:28 System
-drwxr-xr-x    7 root admin  238  08 Oct 11:23 Users
-drwxrwxrwt@   5 root admin  170  14 Oct 10:41 Volumes
-drwxr-xr-x@   39 root wheel  1326  13 Jul 14:14 bin
-drwxrwxr-t@   2 root admin  68  24 Aug 2013 cores
-dr-xr-xr-x    3 root wheel  7937  12 Oct 18:57 dev
import os
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)

pickle / cPickle

Depending on the Python version used, pickle or cPickle will either save truncated data without error, or save a portion with a max size limited to 32bit size.

And depending on how Python has been compiled when installed in the system, it may return errors on either the size of random data requested, or report an OS error as invalid argument.
SOLUTION

Implement strong data validation to be sure that nothing dangerous will ever be processed, and limit data size to 32bit sizes even in 64bit systems.
To check Python behaviour with file writes (on Linux):

```
strace python -OOBRttu script.py
```
PYTHON 2.6

Amount of data we want to write = 4 + 1.048.576 = 1.048.580

**Expected results (using ‘io’ module):**
write(3, "abcd", 4)                     = 4
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1048576) = 1.048.576
All is fine if we use the ‘io’ module.

**With normal calls (without ‘io’ module):**
Results of ‘strace’ with standard ‘open’ call
write(3, "abcdxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 4096) = 4.096
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1044480) = 1.044.480

First call is buffered, instead of writing only 4 (abcd) it writes 4.092 ‘x’
Second call writes ‘x’ for a total of 1.044.480.
Checking the total data written something is not right.
  • 1044480 + 4096 = 1.048.576 (missing 4, expected 1.048.580)
Waiting 5 second ‘fix’ the problem as the OS has flushed the cache.
PYTHON 2.7
Amount of data we want to write = 4 + 1.048.576 = 1.048.580

Expected results (using ‘io’ module):
write(3, "xxxx", 4) = 4
write(3, "abcdxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1048580) = 1048580
All is fine if we use the ‘io’ module.

With normal calls (without ‘io’ module):
Results of ‘strace’ with standard ‘open’ call
write(3, "abcdxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 4096) = 4.096
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1044480) = 1.044.480
write(3, "xxxx", 4) = 4

First call is buffered, instead of writing only 4 (abcd) it writes 4.092 ‘x’
Second call writes ‘x’ for a total of 1.044.480.
Third call will write the remaining ‘x’, and written data is correct.
Only ‘problem’ is that we were expecting ‘2’ calls and NOT ‘3’. 
PYTHON 3.x
Amount of data we want to write = 4 + 1.048.576 = 1.048.580

Expected results (using ‘io’ module):
write(3, "abcd", 4) = 4
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1048576) = 1.048.576
All is fine if we use the ‘io’ module.

With normal calls (without ‘io’ module):
Results of ‘strace’ with standard ‘open’ call
write(3, "abcd", 4) = 4
write(3, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 1048576) = 1.048.576
All is fine if we use the standard ‘open’ call.

**SOLUTION**
Atomic operation are NOT guaranteed in Python 2, core library are using the cache to read and write. The ‘io’ module should be used when possible.
PROTOCOLS

socket, poplib, urllib, urllib2
Core libraries are OS independent, developer must know how to create proper communication channels for each OS, the library will permit to execute operation that are not safe and not correct.

```
import SimpleHTTPServer
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()
```

`socket.error: [Errno 48] Address already in use`

If a client connects to the HTTP server and then we close the server, python will **NOT** release resources, the OS will **NOT** release the socket.
import socket
import SimpleHTTPServer
import SocketServer

PORT = 8080

# ESSENTIAL: socket resuse is setup BEFORE it is 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()

SOLUTION

Each protocol library should be wrapped by a library that, for each OS and each protocol, is properly setting up and tearing down communications, and releasing resources.
Simple test

1. Start a dummy server
2. Use client to connect to server
3. Server sends NULs
4. Client will keep receiving NULs
5. Client memory if full....
6. OS crash!
If using 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.

If using Python < 2.7.9, 3.3:

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

**SOLUTION**

Use ‘Python > 2.7.9’ or ‘Python > 3.3’, if not possible implement controls to check for data type and size.
## Libraries with “Unlimited data“ issues

<table>
<thead>
<tr>
<th>Library</th>
<th>Link to Python bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPLIB</td>
<td><a href="http://bugs.python.org/issue16037">http://bugs.python.org/issue16037</a></td>
</tr>
<tr>
<td>FTPLIB</td>
<td><a href="http://bugs.python.org/issue16038">http://bugs.python.org/issue16038</a></td>
</tr>
<tr>
<td>IMAPLIB</td>
<td><a href="http://bugs.python.org/issue16039">http://bugs.python.org/issue16039</a></td>
</tr>
<tr>
<td>NNTPLIB</td>
<td><a href="http://bugs.python.org/issue16040">http://bugs.python.org/issue16040</a></td>
</tr>
<tr>
<td>POPLIB</td>
<td><a href="http://bugs.python.org/issue16041">http://bugs.python.org/issue16041</a></td>
</tr>
<tr>
<td>SMTPLIB</td>
<td><a href="http://bugs.python.org/issue16042">http://bugs.python.org/issue16042</a></td>
</tr>
<tr>
<td>XMLRPC</td>
<td><a href="http://bugs.python.org/issue16043">http://bugs.python.org/issue16043</a></td>
</tr>
</tbody>
</table>
import io
import os
import urllib2

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)

Wrong file sizes
Filesize: 65536
Filesize: 32768
Filesize: 49152

urllib2 does NOT have proper logic to handle data streams and fails silently.

Every proper size
Filesize: 6598450
Filesize: 6598450
Filesize: 6598450

SOLUTION
Make use of the OS.
PROTOCOL logics

How Python is handling the FTP protocol..

Step 1: Say HELLO

Step 2: Ask LIST

But to have something useful you need..
Proper FTP LOGIC (say HELLO)

Start
- Import
  modules
- Set
  Vars
  DNScheck()
  conprot
  FTP_CLR=0
  FTP_SEC=1
- Set vars
  ftpBASE()
  ftpFEAT()
  Commander(SYST)
  commander(QUIT)
  TLS Error
  TO DO:
  1) Check SSL cert
  2) Negotiate Ciphers
  3) IF TLS error
     3.1) Switch to SSL
  4) IF PROT error
     4.1) Switch schema
  TLS Error
  ftppASS()
  var
  toFTPS=1
  ransleep()
  ftpFEAT()
  ftpCLNT()
  Commander(UTF8)
  Commander(MLST)
  Commander(PASV)
  Commander(ACCT)
  sys.exit()
Proper FTP LOGIC (ask LIST)

- commander(TEST_CWD)
  - CLIST=1
    - ransleep()
  - CLIST=0
    - ransleep()
- commander(TEST_PWD)
  - PLIST=1
    - ransleep()
  - PLIST=0
    - ransleep()
- commander(TEST_LIST)
  - SLIST=1
    - ransleep()
  - SLIST=0
    - ransleep()
- commander(TEST_LISTa)
  - LISTa=1
    - ransleep()
  - LISTa=0
    - ransleep()
- commander(TEST_NLIST)
  - NLIST=1
    - ransleep()
  - NLIST=0
    - ransleep()
- commander(CUIT)
  - sys.exit()
# Known Unsafe Libs

<table>
<thead>
<tr>
<th>ast</th>
<th>multiprocessing</th>
<th>reexec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bastion</td>
<td>os.exec</td>
<td>shelve</td>
</tr>
<tr>
<td>commands</td>
<td>os.popen</td>
<td>subprocess</td>
</tr>
<tr>
<td>cookie</td>
<td>os.spawn</td>
<td>tarfile</td>
</tr>
<tr>
<td>cPickle / pickle</td>
<td>os.system</td>
<td>urllib2</td>
</tr>
<tr>
<td>eval</td>
<td>parser</td>
<td>urlparse</td>
</tr>
<tr>
<td>marshal</td>
<td>pipes</td>
<td>yaml</td>
</tr>
<tr>
<td>mktemp</td>
<td>pty</td>
<td>zipfile</td>
</tr>
</tbody>
</table>
Closing comments:

- Security is VERY hard.
- Python is a great language, we like it very much and we will keep using it.
- Everything used to make this slides has been in the public domain for years, is just difficult to find.
- NEVER assume something is working as it should just because millions of people are using it.
Thank You

Enrico Branca
Security Researcher
enrico.branca@awebof.info
enrico.branca@owasp.org

OWASP Python Security project
https://github.com/ebranca/owasp-pysec/wiki