• About Me

Started to work in IT in 1997, moved to information security in 2001. Working in information security for over a decade with experience in software security, information security management, and information security R&D.

Worked in many roles like Senior Security Engineer, Security Architect, Disaster Recovery Specialist, Microsoft Security Specialist, etc... etc...

Leader of “OWASP Python Security” Project
• [http://www.pythonsecurity.org/](http://www.pythonsecurity.org/)

Co-Leader of “OWASP Project Metrics” Project
• [https://github.com/OWASP/OWASP-Project-Metrics](https://github.com/OWASP/OWASP-Project-Metrics)
• **OWASP Python Security Project**

A new ambitious project that aims at making python more secure and viable for usage in sensitive environments.

• We have started a full security review of python by checking core modules written in both C and python
• First goal is to have a secure layer of modules for LINUX

The security review takes a lot of time and we are slowly publishing libraries and tools, documentation will follow 😊
• **OWASP Python Security Project**

Python Security is a free, open source, OWASP Project that aims at creating a hardened version of python that makes it easier for security professionals and developers to write applications more resilient to attacks and manipulations.

Our code in GITHUB:
• [https://github.com/ebranca/owasp-pysec/](https://github.com/ebranca/owasp-pysec/)

Known Issues in python modules concerning software security:
After checking statistics generated from vendors we have to also check data generated by the community at large.

Statistics on publicly disclosed vulnerabilities are available at the site “NIST.gov” under the name “National Vulnerability Database”


We will review vulnerability stats:
- By Access vector
- By Complexity
- By Severity
- By Category

Then we will formulate some conclusions.
Number of Software Flaws (CVE) by Access Vector

Trend of Software Flaws (CVE) By Access Vector

Number of Software Flaws (CVE) by Complexity


Trend of Software Flaws (CVE) by Complexity

• Initial review of “National Vulnerability Database” statistics revealed:

  – Number of public vulnerabilities relaying on “network” is decreasing
  – Number of public vulnerabilities relaying on “local network” access (adjacent networks) in increasing
  – Number of public vulnerabilities relaying on “local access only” access in increasing
  – Medium or low complexity Vulnerabilities are preferred
• Analysis of the “Web Application Vulnerability Statistics 2013” report revealed:
  – Rate of server misconfigurations is increasing
  – Authentication issues are increasingly not checked
  – Authorization issues are increasingly not checked
  – Server application headers are not sanitized
  – Server application error are not filtered
  – Server default files/dirs are left accessible
• How network configurations can impact internal code operations?

• IP Fragmentation
  – https://isc.sans.edu/forums/diary/IP+Fragmentation+Attacks/13282

• Depending on the system reading the fragmented packets arriving at the NIC, the reassembly process can either DESTROY or REASSEMBLE the original stream, as an application may have sent valid data but the receiving end may see only random data.
def genjudyfrags():
    pkts=scapy.plist.PacketList()
    pkts.append(IP(flags="MF",frag=0)/("1"*24))
    pkts.append(IP(flags="MF",frag=4)/("2"*16))
    pkts.append(IP(flags="MF",frag=6)/("3"*24))
    pkts.append(IP(flags="MF",frag=1)/("4"*32))
    pkts.append(IP(flags="MF",frag=6)/("5"*24))
    pkts.append(IP(frag=9)/("6"*24))
    return pkts

This section of code will generate six packet fragments as outlined in “IP Fragment Reassembly with scapy“ with the offsets specified in the Shankar/Paxson and Novak papers.

The picture is taken from the Novak paper and represent the final packet order per each reassembly policy.

http://www.snort.org/assets/165/target_based_frag.pdf
python -OOBR reassembler.py –demo
Reassembled using policy: First (Windows, SUN, MacOS, HPUX)

```
111111111111111111111111111111111
4444444442222222222222222222222
333333333333333333333333333333333
666666666666666666666666666666666
```

Reassembled using policy: Last/RFC791 (Cisco)

```
111111114444444444444444444444444
2222222222222222222222222222222
555555555555555555555555555555555
666666666666666666666666666666666
```

Reassembled using policy: Linux (Umm.. Linux)

```
1111111111111111111111111111111
4444444444444444444444444444444
2222222222222222222222222222222
555555555555555555555555555555555
666666666666666666666666666666666
```

Reassembled using policy: BSD (AIX, FreeBSD, HPUX, VMS)

```
1111111111111111111111111111111
4444444444444444444444444444444
2222222222222222222222222222222
333333333333333333333333333333333
666666666666666666666666666666666
```

Reassembled using policy: BSD-Right (HP Jet Direct)

```
111111114444444444444444444444444
2222222222222222222222222222222
555555555555555555555555555555555
666666666666666666666666666666666
```

24th October 2014, București, România
• What about numeric operations?

As an example we will take in consideration LINUX.

Many security operations are based on random numbers and every linux system using any cryptographic function can be impacted by the lack of good entropy.

What is generally overlooked is that under linux almost every process uses entropy when is created and even the network stack uses entropy to generate the “TCP-syn cookies”.
• This is an expected behavior and is working as designed.
• Spawning a process uses (on average) 16 bytes of entropy per “exec()”, therefore when server load spikes entropy is quickly depleted as the kernel is not generating entropy fast enough.
• Also when a system is built to use “Stack Smashing Protector” (SSP) by default it uses “/dev/urandom” directly, this tends to consume all the kernel entropy.
• Almost all modern Linux systems use “Address space layout randomization“ (ASLR) and stack protections that need a small amount of entropy per process. Since “/dev/urandom” always remixes, it doesn't strictly run out, but the entropy drops.
• In fact many Linux commands used to check the amount of entropy are “consuming” it and may lead to its depletion.

• For example this command will “consume” entropy
  – `watch cat /proc/sys/kernel/random/entropy_avail`

• But this Python one-line script will NOT use entropy:
  – `python -c "$(echo -e "import time\nwhile True:\ntime.sleep(0.5)\nprint open('/proc/sys/kernel/random/entropy_avail', 'rb').read(),")"`

• Also the command “`inotifywatch -v -t 60 /dev/random`” will monitor the access to “/dev/random” without using entropy
What happens to the entropy level in a working Linux server under average load?

150-200 bits = Entropy lowest limit

Generate 1024 bits SSL key

Generate 128 bits SSL key
Under Linux every process uses entropy and every server “should” not have less than 200 bits. It is possible to increase the entropy level using entropy daemons like the package “haveged”. (http://www.issihosts.com/haveged/)
**PYTHON for networking?**

<table>
<thead>
<tr>
<th>Scapy</th>
<th>libdnet</th>
<th>dpkt</th>
<th>Impacket</th>
</tr>
</thead>
<tbody>
<tr>
<td>pypcap</td>
<td>pynids</td>
<td>Dirtbags py-pcap</td>
<td>flowgrep</td>
</tr>
<tr>
<td>Mallory</td>
<td>Pytbull</td>
<td>Otrace</td>
<td></td>
</tr>
</tbody>
</table>

**PYTHON for fuzzing?**

<table>
<thead>
<tr>
<th>Sulley</th>
<th>Peach Fuzzing</th>
<th>antiparser</th>
<th>TAOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>untidy</td>
<td>Powerfuzzzer</td>
<td>Mistress</td>
<td>Fuzzbox</td>
</tr>
<tr>
<td>WSBang</td>
<td>Construct</td>
<td>Fusil</td>
<td>SMUDGE</td>
</tr>
</tbody>
</table>
• OWASP Secure Coding Principles

1. Minimize attack surface area
2. Establish secure defaults
3. Principle of Least privilege
4. Principle of Defence in depth
5. Fail securely

6. Don’t trust services
7. Separation of duties
8. Avoid security by obscurity
9. Keep security simple
10. Fix security issues correctly
• In reality “Secure coding” is a **PRACTICE**

**Practice**: “the actual application or use of an idea, belief, or method, as opposed to theories relating to it”

The definition of “secure coding” _changes over time_ as each person/company has different ideas.

• **Is about how to DESIGN code to be inherently secure and NOT on how to write secure code**
As a PRACTICE secure coding includes but is not limited to:

- Definition of areas of interest
- Analysis of architectures involved
- Review of implementation details
- Verification of code logic and syntax
- Operational testing (unit testing, white-box)
- Functional testing (black-box)
Secure coding depends on “functional testing”

- **Functional testing**: “verifies a program by checking it against ... design document(s) or specification(s)”
- **System testing**: “validate[s] a program by checking it against the published user or system requirements”


- **Operational testing** = white-box testing → unit-test
- **Functional testing** = black-box testing
• PYTHON → use with moderation

We have seen some powerful tools written in python but what about the security of python itself?

• Are there operations to avoid?
• Any module or core library to use with caution?
• Something to know before writing code for security?
• EXAMPLE – numeric overflow

```python
N = 2 ** 63
for n in xrange(N):
    print n
RESULT (debian 7 x64)
Traceback (most recent call last):
  File "xrange_overflow.py", line 5, in <module>
    for n in xrange(N):
OverflowError: Python int too large to convert to C long
```

**PROBLEM:** xrange uses "Plain Integer Objects" created by the OS

**SOLUTION:** Use python "long integer object“ that will allow numbers of arbitrary length as the limit will be the system's memory.
• EXAMPLE – operations with file descriptors

```python
import sys
import io

fd = io.open(sys.stdout.fileno(), 'wb')
fd.close()

try:
    sys.stdout.write("test for error")
except Exception:
    raise
```

**RESULT**

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

Code is trying to write a non-zero amount of data to something that does not exists. The file descriptor has been closed and nothing can be sent, but python has no control over it and returns a system error.
• EXAMPLE - File descriptors in Windows

C:\Python27>python.exe -V
Python 2.7.6

python.exe -OObtt "winfd_1.py"

import io
import sys

fd = io.open(sys.stdout.fileno(), 'wb')
f.close()

sys.stdout.write("Now writing to stdout closed FD will cause a crash")
• EXAMPLE – string evaluation

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

The function "eval" executes a string but is not possible to any control to the operation. Malicious code is executed without limits in the context of the user that loaded the interpreter.

**REALLY DANGEROUS**
• EXAMPLE – input evaluation

Secret = "A SECRET DATA"
Public = "a COCONUT"
value = input("Please enter your age ")
print "There are",value,
print "monkeys looking for",Public

What you type as input is interpreted through an expression and the result is saved into your target variable with no control or limits.

python -OOBRtt input_1.py
Please enter your age 32
There are 32 monkeys looking for a COCONUT

The dir() function returns “most” of the attributes of an object.

python -OOBRtt input_1.py
Please enter your age dir()
There are ['Public', 'Secret', '__builtins__', '__doc__', '__file__', '__name__', '__package__'] monkeys looking for a COCONUT

python -OOBRtt input_1.py
Please enter your age Secret
There are A SECRET DATA monkeys looking for a COCONUT
• Unicode string encode/decode

RESULT

Correct-String "u'A\ufffdBC\ufffd"

CODECS-String "u'A\ufffdBC"

IO-String "u'A\ufffdBC\ufffd"

→ KNOWN GOOD STRING
→ WRONG
→ OK

The problem is due to a bug in the "codec" library that detects the character "F4" and assumes this is the first character of a sequence of characters and wait to receive the remaining 3 bytes, and the resulting string is truncated.

A better and safer approach would be to read the entire stream and only then proceed to the decoding phase, as done by the "io" module.
• CODE — Unicode string encode/decode

```python
import codecs
import io
try:
    ascii
except NameError:
    ascii = repr
b = b'\x41\xF5\x42\x43\xF4'
print("Correct-String %r") % ((ascii(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") % (ascii(fin.read()))
with io.open('temp.bin', 'rt', encoding='utf8', errors='replace') as fin:
    print("IO-String %r") % (ascii(fin.read()))
```

< ISSUE HERE >
import os
import cPickle
import traceback

random_string = os.urandom(int(2147483648))
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

pickle/CPICKLE (debian 7 x64)
LIMIT = 2147483648 - 1 = 2147483647
(32bit integer object)
TEST WITH STRING SIZE "2147483647"
ALL OK

TEST using cPickle (data corruption)
TEST WITH STRING SIZE "2147483648"
######## ERROR-WRITE #######
<type 'exceptions.SystemError'>
....
File "pickle_2.py", line 18, in <module>
pickle.dump(random_string, fout)
SystemError: error return without exception set
• EXAMPLE – data corruption with “pickle”

```python
import os
import pickle
import traceback
random_string = os.urandom(int(2147483648))
print("STRING-LENGTH-1=%r") % (len(random_string))
fout = open('test.pickle', 'wb')
try:
    pickle.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 = pickle.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
```

pickle/CPICKLE (debian 7 x64)
LIMIT = 2147483648 - 1 = 2147483647
(32bit integer object)
TEST WITH STRING SIZE "2147483647"
ALL OK

TEST using pickle (data corruption)
TEST WITH STRING SIZE "2147483648"
############ ERROR-WRITE ############
<type 'exceptions.MemoryError'>
....
File "/usr/lib/python2.7/pickle.py", line 488, in save_string
    self.write(STRING + repr(obj) + '\n')
MemoryError
• EXAMPLE – unrestricted code in “pickle”

```python
import pickle
obj = pickle.load(open('./bug.pickle'))
print "== Object =="
print repr(obj)
```

**WARNING:** pickle or cPickle are NOT designed as safe/secure solution for serialization
• EXAMPLE – inconsistent “pickle” serialization

**RESULT**

```
# python 3
import pickle
import collections
dct = collections.defaultdict()
f = pickle.dumps(dct, protocol=1)
print (repr(f))
g = pickle.dumps(dct, protocol=1, fix_imports=False)
print (repr(g))
h = pickle.dumps(dct, protocol=2)
print (repr(h))
i = pickle.dumps(dct, protocol=2, fix_imports=False)
print (repr(i))
```

(http://hg.python.org/cpython/file/7272ef213b7c/Li
b/_compat_pickle.py at line 80)

If there's a “collections.defaultdict” in the pickle dump, python 3 pickles it to “UserString.defaultdict” instead of “collections.defaultdict” even if python 2.7 and 2.6 do not have a “defaultdict” class in “UserString”.

24th October 2014, București, România
• **EXAMPLE – review of pickle/cPickle**
  
  — Main problems: code injection, data corruption

• cPickle: severe errors as exceptions are "lost" even if an error is generated and signalled by the O.S.
• pickle: no controls on data/object integrity
• pickle: no control on data size or system limitations
• pickle: code evaluated without security controls
• pickle: string encoded/decoded without verification
• EXAMPLE – socket remains open after error..

OPEN IN TERMINAL 1 (one line):
python -m smtpd -n -c DebuggingServer localhost:45678

OPEN IN TERMINAL 2:
python -OOBRtt smtplib_1.py

import smtplib
try:
s = smtplib.SMTP_SSL("localhost", 45678)
except Exception:
    raise

RESULT:

lsof -P | grep python | grep ":45678"
python    16725 user01  3u  IPv4  31510356  0t0  TCP localhost:45678 (LISTEN)

The underlying socket connection remains open, but you can't access it or close it.
• EXAMPLE – “unlimited data” in POP3

```python
import poplib
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."
```

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

```python
$ python -OOBRtt pop3_client.py
Connecting to '127.0.0.1':45678...
Welcome: '+OK THIS IS A TEST'
Error: 'out of memory'
```
**EXAMPLE – leaks in poplib/urllib/smtplib …**

```
python -OOBRtt pop3_server.py
Traceback (most recent call last):
  File "pop3_server.py", line 12, in <module>
    sock.bind((HOST, PORT))
  File "/usr/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 98] Address already in use
```

If python process has an error the **exception** will not reliably close all file and socket file descriptors (handles) leading to **leaks** and **uncontrollable** background processes

```
ps aux | grep pop3
user01  30574  0.0  0.0  33256  6052 ?        S    19:34   0:00 /usr/bin/python –OOBRtt pop3_server.py

lsf -P | grep python | grep pop3
pop3_serv 30574 user01 txt /usr/bin/python2.7
pop3_serv 30574 user01 mem REG /usr/lib/python2.7/lib-dynload/_ssl.so
```
• EXAMPLE – libs with “unlimited data“ issues

HTTPLIB → http://bugs.python.org/issue16037 (fixed)
FTPLIB → http://bugs.python.org/issue16038 (fixed)
IMAPLIB → http://bugs.python.org/issue16039 (fixed)
NNTPLIB → http://bugs.python.org/issue16040 (fixed)
POPLIB → http://bugs.python.org/issue16041
SMTPLIB → http://bugs.python.org/issue16042
XMLRPC → http://bugs.python.org/issue16043
Small list of **KNOWN UNSAFE** python components

- ast
- bastion commands
- cookie
- cPickle
- eval
- marshal
- mktemp
- multiprocessing
  - os.exec
  - os.popen
  - os.spawn
  - os.system
  - parser
  - pickle
  - pipes
- pty
- rexec
- shelf
- subprocess
- tarfile
- yaml
- zipfile
• PYTHON for the web?

<table>
<thead>
<tr>
<th>Requests</th>
<th>HTTPie</th>
<th>ProxMon</th>
<th>WSMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twill</td>
<td>Ghost</td>
<td>Windmill</td>
<td>FunkLoad</td>
</tr>
<tr>
<td>spynner</td>
<td>mitmproxy</td>
<td>pathod/pathoc</td>
<td>scrapy</td>
</tr>
</tbody>
</table>

• PYTHON for offensive actions?

Plenty of **dangerous** python tools in “packet storm security” website:

More general tools:
• [http://pythonsource.com/](http://pythonsource.com/)
### PYTHON for reverse engineering?

<table>
<thead>
<tr>
<th>Tool</th>
<th>IDAPython</th>
<th>pyasm2</th>
<th>pype32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Androguard</td>
<td>libdisassemble</td>
<td>PyBFD</td>
<td>python-adb</td>
</tr>
<tr>
<td>apkjet</td>
<td>llvmpy</td>
<td>PyCodin</td>
<td>python-ptrace</td>
</tr>
<tr>
<td>AsmJit-Python</td>
<td>Miasm</td>
<td>pydasm</td>
<td>PythonGdb</td>
</tr>
<tr>
<td>BeaEnginePython</td>
<td>ollydbg2-python</td>
<td>PyDBG</td>
<td>PyVEX</td>
</tr>
<tr>
<td>Binwalk</td>
<td>OllyPython</td>
<td>pydbgr</td>
<td>pywindbg</td>
</tr>
<tr>
<td>Buggery</td>
<td>PDBparse</td>
<td>PyELF</td>
<td>Rekall</td>
</tr>
<tr>
<td>cuckoo</td>
<td>pefile</td>
<td>pyew</td>
<td>Vivisect</td>
</tr>
<tr>
<td>Disass</td>
<td>PIDA</td>
<td>pygdb2</td>
<td>Volatility</td>
</tr>
<tr>
<td>ElfParserLib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frida</td>
<td>PyADB</td>
<td>pyMem</td>
<td>WinAppDbg</td>
</tr>
</tbody>
</table>

24th October 2014, București, România
• **Closing Summary**

• Python is a *powerful* and *easy to learn* language **BUT** has to be used with care.

• There are *no limits* or controls in the language, *is responsibility of the coder to know what can be done and what to avoid.*
Secure Coding Review

**Server Issues**
- Misconfiguration
- Application headers
- Application Errors
- Default files
- Default Locations
- Traffic in clear text
- Vulnerable to DoS
- Vulnerable to MITM

**Crypto Issues**
- Weak ciphers
- Small keys
- Invalid SSL certs

**Access class to Monitor**
- Local network
- Local access only
- Remote Network Access

**Vulnerabilities to Check**
- Format String
- Buffer Errors
- Credentials Management
- Cryptographic Issues
- Information Leak
- Input Validation
- OS Command Injections
- SQL Injection

**Architectural Aspects**
- Kernel Architecture
- Data write policy
- NIC configuration
- Entropy pool

**Language Issues**
- File operations
- Object evaluations
- Instruction Validation
- Variable Manipulation
- String/Input Evaluation
- Unicode encode/decode
- Serialization
- Data limits

24th October 2014, Bucharest, Romania
Contact

Enrico Branca

“OWASP Python Security Project”
http://www.pythonsecurity.org/

Email: enrico.branca@owasp.org
Linkedin: http://fr.linkedin.com/in/ebranca