avet
Information and Network Security

Python source code review - Best practices version 0.9

AVET Information and Network Security Sp. z o.o.,
Belgijska 11, 02-511 Warsaw, Poland
Tel. +48 (22) 88 00 220, Fax. +48 (22) 88 00 726,

ZSJZ
AQAP 2110:2009
PN-EN ISO 9001:2009
PN-ISO/IEC 27001:2007

Table of Contents

Python source code review - Best practices version 0.9

Below we present some of best practices we use at AVET INS when reviewing Python code. Please note that we are focusing on Python only, so if you are using some 3rd party complex modules or packages like Django additional rules apply. Also please take in account that we are not discussing in this paper general code review / code audit rules and best practices but they do apply to the overall process.

Why source code audit?

In case of dynamic languages compiled to bytecode like Python this is quite valid question: after all, it is easy to recover at least partially functional source code from bytecode and unless Python code has been compiled into native executable form (with py2exe for example) dynamic analysis can be conducted even with Python build-in tools. Nevertheless, operating on the bytecode level using dynamic analysis techniques always raises the question of real code coverage. In case of static source code review – which can be aided by some dynamic analysis techniques, especially in case of languages like Python - we can be sure of 100% code coverage. From security perspective this is a crucial advantage. One could argue that if you employed static analysis of bytecode you could achieve comparable results. While it is true in terms of possible code coverage, assuming ignoring Control Flow Graphs (CFGs) during analysis, you are still missing important security and quality related information that you can see at source code level. You are missing comments, some naming schemes etc. All those information can be useful for analysis going beyond simple static checking of particular source code line, looking for insecure API calls. You gain insight regarding code quality, possible code refactoring needs, coding skills and culture – all of those elements have direct impact on overall security level of your system or application.

Running code first

It is a good general practice to compile and run the code first - especially when we are dealing with dynamic language like Python – before starting manual or automatic source code review process. If code fails at run-time or even earlier during compilation we can easily gain tips for places required further, more detailed source code review. Do not run python modules from byte code compiled files[1]. Instead force complete byte code compilation by deleting all .pyc and .pyo files.

[1] *.pyc and *.pyo are optimized compilation output files

How to run Python code

Python interpreter supports number of useful switches that can be aid in our process. Therefore instead of simple python [filename] usage we will use a bit more advanced method:

python -OOBRstt test.py

-OO	Turn basic optimization and discard docstrings
-B	Python won't try to write .pyc or .pyo files during import of modules (new in 2.6)
-R	Turns on hash randomization so that the __hash__() values of str, bytes and datetime objects are salted with an unpredictable random value. Those values remain constant within and individual Python process but they are not predictable between repeated Python interpreter invocations.
-s	Don't add the user site-packages directory to sys.path (new in 2.6).
-tt	Issue an error when source file mixes tabs and spaces for indentation in a way that makes it depend on the work of tab expressed in spaces.

Typical defects

Below we present of most typical defects we've seen during source code auditing. Please note that some of defects described are not possible to detect directly using only automatic, static source code analysis. Manual source code review may be required to uncover and properly address some of the issues.

For some points we also present vulnerable code to demonstrate described problems. Do not use this code in your projects under no circumstance.

Modules instead of packages

Some Python programmers may disagree on this point. In fact there is nothing wrong about using modules instead of packages. However packages through their __init__.py interface allow in our opinion better segregation and separation of privileges and functionality providing overall better architecture. Designing application in packages in mind is a good strategy, especially for more complex projects.

The __init__.py package interface allows better control over imports and exposing interfaces like variables, functions and classes.

We consider code build around thousand different modules not grouped into packages as both: bad designed practice and poor quality under most circumstance.

Lack of unit tests

Since version 2.1 Python comes as part of Python Standard Library with unittest: unit testing framework. Yet we've seen relatively few projects actually using this or as matter of fact any other unit testing solution. The simple rule is: if you are writing complex code, use unit tests. We consider code lacking unit test as poor quality in most circumstances.

Improper input/output validation

Improper input and output validation and sanitization is one of most critical and most often found problems leading to security vulnerabilities according to our statistics. Below we present some simple code snippets demonstrating such classes of defects.

```python
def main():
    for arg in sys.argv[1:]:
        os.system(arg)
```

In the above example *arg* argument is being passed to function considered as insecure (look below) without any validation.

Another issue is lack of output validation. Below is example (also used further to demonstrate another class of defects) using print without validation of variable filename controlled by user input:

```python
filename = sys.argv[1] + '\\' + sys.argv[2]

if os.path.isfile(filename):
    print filename, 'exists'
else:
    print filename, 'not found'
```

Insecure API usage

This class of vulnerabilities is based around usage of either insecure API calls or insecure usage of APIs. For example *os.system()* is considered insecure and should be replaced with *subprocess* module from Python Standard Library.

```
def main():
    for arg in sys.argv[1:]:
        os.system(arg)
```

Other examples of insecure APIs from Python Standard Library include:

- *os.tempnam()*
- *os.tempnam()*
- *os.tmpname()*
- *random.random()*
- *tempfile.mktemp()*

Also, as noted further, any *ctypes* based calls should be automatically consider dangerous without proper further inspection.

Improper execution environment sanitization and excessive trust

Modules and packages should not trust each other and their execution environment. Therefore when starting module, sanitization of execution environment – like validation and cleaning of environment variables - should be required.

```
def main():
    appdir = os.getenv('APPDIR')
    apppath = os.path.join(appdir,sys.argv[0])
```

Improper error handling

Handling errors and stressful situation correctly and in secure manner is not a trivial task, therefore it is a constant stream of software defects, some leading to exploitable vulnerabilities. This is not just Python code problem, however there is one code construction typical to this dynamic language employing exception handling

```
try:
    fin = open(sys.argv[1], 'r')
    data = fin.readlines()
    fin.close()
except:
    pass

print 'Data size = ', len(data)
```

The empty except clause is not the only way of handling errors badly in Python. Another example is not really Python-centric, but we still find it often:

```
def main():
    appdir = os.getenv('APPDIR')
    apppath = os.path.join(appdir,sys.argv[0])
```

Please take a note, that *os.getenv()* call is never checked for an error.

Type protection and error detection

It is generally true and applies to both programming languages that are compiled in time (like Python) and also compiled before deployment (like Java). When obtaining parameter from user HTTP request or database instance, its type is generally unknown. Variable casting to what author assumes it contains should be performed precisely with correct error handling in place.

Take a look at the following code:

```
variable = 0
test = "abc"

if variable == 1:
    print "equals 1"
else:
    print "%.2f" % (variable/test)
```

Error reported by interpreter:

```
print "%.2f" % (variable/test)
ZeroDivisionError: integer division or modulo by zero
```

Error detection / scope issues

Python is not compiled before deployment, so it may contain obvious errors that will occur if user supplies certain data or execution hits block that was not fully tested. The following block of code is syntactically correct, but will work properly only when variable is equal to 1, otherwise undefined error occurs. Compiled languages are usually free of such errors as compiler provides message to a programmer when such error is discovered in the build process.

```
variable = 0

if variable == 1:
    wrong = "wrong"
else:
    print "uninitialized variable: %s" % wrong
```

Error reported by interpreter:

```
print "uninitialized variable: %s" % wrong
NameError: name 'wrong' is not defined
```

TOCTOU problems

Time of Check Time of Use (TOCTOU) problems are not only nightmare for Python developers but they affect basically any language in which you access some kind of objects. A typical case is a symlink vulnerability where after check of availability of an object but before accessing it's data attacker creates a symlink pointing to his object, a different one than during checking phase. Here is a simple example:

```
def main():
    for arg in sys.argv[1:]:
        if os.path.isfile(arg):
            m = hashlib.md5()
            fin = open(arg, 'r')
            data = fin.read()
            m.update(data)
            print arg, ' ', m.hexdigest()
```

Please note that the above code also contains another defect: it never closes fin file handle.

Insecure temporary file usage

There are number of possibilities for introducing such vulnerability into your Python code. The most basic one is to actually use deprecated and not recommend temporary files handling function:

- *os.tempnam()* – this function is vulnerable to symlink attacks and should be replaced with tempfile module functions (see below!)
- *os.tmpname()* - – this function is vulnerable to symlink attacks and should be replaced with tempfile module functions (see below!)
- *tempfile.mktemp()* – this function has been deprecated since version 2.3. Use tempfile.mkstemp() instead

Compatibility and porting issues

While Python in theory runs on many different platforms, the application written in Python may fail to run properly or at all without particular changes. Below are Windows code examples that will fail under Unix/Linux/BSD systems due to path handling and native command execution:

```
filename = sys.argv[1] + '\\' + sys.argv[2]

if os.path.isfile(filename):
    print filename, 'exists'
else:
    print filename, 'not found'
```

The code snippet below demonstrates usage of Windows commands for listing a directory, instead of using portable *os.walk()* for example. The below code will fail under most non-Windows systems:

```
for arg in sys.argv[1:]:
    os.system('dir' + ' ' + arg)
```

Mishandling of assets

This is typical example of mishandling application and system assets:

```
def main():
    for arg in sys.argv[1:]:
        if os.path.isfile(arg):
            m = hashlib.md5()
            fin = open(arg, 'r')
            data = fin.read()
            m.update(data)
            print arg, ' ', m.hexdigest()
```

The code above never closes *fin* handle as for loop is missing *fin.close()* statement. The problem is that this code actually works so just running the code will not show the defect. This is a great example of power of static source analysis supported by manual review.

The import/reload problems

Dynamic nature of Python makes is a perfect target for programmers to try to abuse it in one way or another. We've seen some bugs that can be tricky to detect due to importing, rewriting on the fly and reloading modules. Basically if there isn't a need for such action, we consider it to be a bad programming practice.

Misuse of ctypes / problems with modules implemented in C

Ctypes is a powerful mechanism allowing calling C and system libraries within Python code. This brings issues of misusing C/system functions. Furthermore if module is using ctypes to call external API implemented in C/C++ or a system API we reintroduce a whole set of vulnerability classes like buffer / heap overflow, format strings etc. Be careful when you are calling external API, even if you are calling binary module that is part of your project.

Summary

The list of issues presented here is a just a tip of an iceberg. Our objective was not to describe every possible class of defect and insecure Python code fragments in this short paper. Instead, bring potential problems to your attention by demonstrating most common issues we've seen so far. What's not been covered here are bugs directly related to different web and mobile applications due to several reasons. In most cases such applications are implemented using different frameworks and database engines / ORMs and every single solution has own set of possible insecure programming practices. We also didn't touch issues specifically related to Python 3 and conversion from 2.7.x line to 3.

Another set of potential issues brings threading and queuing which often results in extending Python with modules like Twisted, Celery or many others. We also didn't touch the availability and performance issues connected with Global Interpreter Lock (GIL).

Our objective was to provide a reader with set of patterns forming baseline guideline for reviewing his or hers Python source code.

Accessing code examples

Presented code snippets are available from our public github repository at:

https://github.com/avet/InsecureProgramming

You can use those files as a simple test set against your static source code analysis toolkit. Keep in mind that all those files contains vulnerabilities and **mustn't** be run on production systems.

About AVET INS

AVET Information and Network Security is an information security company specialized in delivering security consulting solutions to the most demanding business and public organizations. Our unique expertise, together with a great team of experts and an advanced toolkit base, allows us to secure mission critical systems and applications, effectively defending your organization from cyber security threats.

One of our areas of expertise is the Secure Development Lifecycle (SDL): a process allowing an organization to have secure applications, either developed by 3rd parties or by in-house teams. A crucial element of SDL is source code review. We deliver our customers not only our expertise, but also tools aiding them in deploying and managing SDL, assuring the highest level of security possible.

If you are interested in learning how you can defend your critical application from cyber security threats please contact us.

Contact information

AVET Information and Network Security Sp. z o.o.
Belgijska 11
02-511 Warsaw, Poland
Tel. (+48 22) 88 00 220
Fax. (+48 22) 88 00 726
avet@avet.com.pl
www.avet.com.pl

Further revision of this document

We plan to provide further updates to this best practices guide. You can always find newest publicly available version at http://www.avet.com.pl

AVET INS customers using **AVET SecureCode**, **AVET SecurityOutline** or our risk monitoring services will get access to updated versions before making them publicly available.