Zeste de savoir

# Dropbox a des fuites !

12 août 2019

# Table des matières



Lors de la conférence WOOT'13, les hackers Dhiru Kholia et Przemysław Węgrzyn ont présenté un article décrivant la façon dont ils ont réussi à entrer dans le code de Dropbox[1], réalisant ainsi une prouesse technique qui attira pas mal d'attention sur la sécurité des programmes Python.

Dans cet article, publié initialement sur progdupeu.pl en septembre 2013, nous allons faire un tour d'horizon des techniques de masquage de code-source Python les plus couramment employées, et nous démontrerons comment celles-ci peuvent être contournées en étudiant dans le détail la technique de ces deux hackers. Ce faisant, nous allons découvrir un grand nombre de choses à propos du fonctionnement interne de Python, et, je l'espère, tordre le cou à une poignée d'idées reçues chez les développeurs débutants.

article, à cette adresse . Leur code-source a quant à lui été publié sur un dépôt github ici .

## 1. Généralités sur les applications Python et leur distribution

CPython est la distribution standard de Python. Il s'agit d'un interpréteur écrit en C dont le développement est dirigé depuis le début par le créateur de Python, Guido Van Rossum.

Une application prévue pour tourner sur CPython peut contenir du code sous la forme :

— De modules Python *en clair*, portant l'extension ".py",
— De modules Python pré-compilés, portant l'extension ".pyc",
— De modules en C utilisant l'API C de CPython, et compilés sous la forme de bibliothèques dynamiques. Ces modules portent donc l'extension ".so" sous les systèmes d'exploitation POSIX, ou ".dll" sous Windows.

1. On peut retrouver une vidéo de leur présentation, ainsi que leur

CPython est aussi capable d'exécuter un programme complet compressé dans une archive `zip`, à condition que le fichier constituant son point d'entrée soit nommé `__main__.py` :

```
:::console
% ls
__main__.py  module.py
% cat module.py
def say_hello():
    print "Hello, World!"
% cat __main__.py
import module
module.say_hello()
% zip helloworld module.py __main__.py
  adding: module.py (deflated 5%)
  adding: __main__.py (deflated 9%)
% ls
helloworld.zip  __main__.py  module.py
% python helloworld.zip
Hello, World!
```

En général, les éditeurs d'applications commerciales prennent le parti de "geler" leur code-source dans un exécutable embarquant un interpréteur CPython. Cela leur permet de contrôler l'environnement dans lequel leur logiciel est exécuté, c'est-à-dire *a minima* la présence d'un interpréteur CPython dans une version compatible avec leur soft, et des modules nécessaires à l'exécution de celui-ci. Ces exécutables "gelés" prennent la plupart du temps la forme d'une archive `zip` dont l'en-tête d'auto-extraction (le *SFX stub*) est remplacée par l'interpréteur CPython.

Il existe de nombreuses solutions permettant d'empaqueter une application sous cette forme. On pourra citer à titre d'exemple py2exe , bbfreeze , ou encore cx_Freeze , qui est à ce jour la seule solution compatible avec Python 3.

Contrairement à une idée reçue chez beaucoup de débutants en Python, **le fait de *geler* une application ne constitue absolument pas une protection contre l'accès à son code-source** : il suffit de décompresser cet exécutable au moyen d'*unzip* pour en extraire les modules. C'est le cas par exemple pour *Dropbox*, dont on pourra retrouver l'archive dans le dossier `$HOME/.dropbox-dist/dropbox` sur une installation standard sous GNU-Linux.

```
:::console
% unzip ~/.dropbox-dist/dropbox -d dropbox_modules/ >/dev/null
% ls -lh dropbox_modules |head
_abcoll.pyc
abc.pyc
arch
asynchat.pyc
asyncore.pyc
atexit.pyc
```

```
autogen_explicit_imports.pyc
babel
base64.pyc
bisect.pyc
```

On remarque que les modules Python contenus dans cette archive portent tous l'extension ".pyc". Intéressons-nous maintenant de plus près à ces fichiers.

## 1.1. Les fichiers ".pyc" pré-compilés

Lorsqu'un module Python est parsé par CPython, son code-source est compilé en une séquence de *code objects*, dont on peut considérer qu'il s'agit de morceaux de bytecode directement compréhensibles par l'interpréteur. Il arrive que ces *code objects* soient écrits dans des fichiers intermédiaires, les fameux fichiers ".pyc". C'est notamment le cas, par défaut, lorsque CPython doit charger une dépendance d'un programme qui vient sous la forme d'un module Python : le bytecode résultant de la compilation de ce module est écrit dans un fichier ".pyc" que l'interpréteur pourra recharger lors de la prochaine exécution pour s'éviter de recompiler le code-source. Ces fichiers ".pyc" sont donc initialement prévus pour servir de cache afin d'accélérer le chargement des programmes. Cela présente plusieurs avantages pour les développeurs désireux de masquer leurs sources.

À titre d'illustration, nous allons prendre le script `unpycme.py`, très simple, que voici :

```
:::python
"""
This is an example python script

"""

def print_filename():
    filename = __file__
    print "Current file name is '%s'" % filename


if __name__ == '__main__':
    print_filename()
```

Lorsqu'il est exécuté, ce script affiche le nom du fichier courant.

```
:::console
% python unpycme.py
Current file name is 'unpycme.py'
```

L'exemple suivant montre que l'on peut très simplement générer un fichier `unpycme.pyc` à partir de ce script, lui-même directement exécutable par CPython...

```
:::console
% ls
unpycme.py
% file unpycme.py
unpycme.py: a python script, ASCII text executable
% python -c "import unpycme"
% ls
unpycme.py  unpycme.pyc
% file unpycme.pyc
unpycme.pyc: python 2.7 byte-compiled
% python unpycme.pyc
Current file name is 'unpycme.pyc'
```

... à condition que ce soit **exactement la même version** de CPython que celle avec laquelle il a été généré :

```
:::console
% python3.3 unpycme.pyc
RuntimeError: Bad magic number in .pyc file
```

En interne, un fichier ".pyc" est une structure très simple, composée :

— D'un nombre constant (ou *magic number*) sur deux octets, suivi de la chaîne `0x0d0a` (`"\r\n"`),
— D'un *timestamp* sur 4 octets donnant la date à laquelle celui-ci a été généré,
— D'un code object sérialisé (*marshalled*, dans le vocabulaire Python), décrivant la totalité du module.

Le *magic number* est propre à chaque version de CPython. En fait, il est propre à chaque révision du jeu d'opcodes composant le bytecode Python, qui peut lui-même évoluer au sein d'une même version mineure de CPython, au gré des fonctionnalités ajoutées par les développeurs. Il est donc à considérer comme délibérément instable et non rétrocompatible. Jusqu'à Python 2.7, on pouvait retrouver l'historique de ces *magic numbers* et des versions de Python correspondantes en commentaire dans le fichier source `Python/import.c`[2] de CPython.

Le *timestamp* est utilisé par CPython pour déterminer si un module a été modifié depuis la dernière fois qu'il a été mis en cache. Il ne nous intéresse pas dans cet article. Pas plus que le format précis des *code objects* (du moins pour l'instant).[3]

générés par Python 2.7 à cette adresse .

2. http ://hg.python.org/cpython/file/869d50357a71/Python/import.c
3. On pourra trouver une inspection un peu plus profonde des fichiers ".pyc"

## 1.2. Décompilation du bytecode Python

Il existe de nombreuses solutions permettant d'interagir avec le bytecode Python.

On peut déjà citer le module `dis`[4] de la bibliothèque standard, qui permet de désassembler des *code objects* afin de les présenter sous une forme plus lisible (proche de l'assembleur). Par ailleurs, on peut aussi trouver des décompilateurs plutôt efficaces, tels que *uncompyle2*[5] pour Python 2 ou *unpyc3*[6] pour Python 3.

Ces décompilateurs sont très simples d'utilisation pourvu que les fichiers ".pyc" soient correctement formés. Voici un exemple d'utilisation de *uncompyle2* :

```
:::console
% uncompyle2 unpycme.pyc
# 2013.09.01 22:59:13 CEST
#Embedded file name: unpycme.py
"""
This is an example python script

"""

def print_filename():
    filename = __file__
    print "Current file name is '%s'" % filename


if __name__ == '__main__':
    print_filename()
+++ okay decompyling unpycme.pyc
# decompiled 1 files: 1 okay, 0 failed, 0 verify failed
# 2013.09.01 22:59:13 CEST
```

On remarquera qu'*uncompyle2* est capable de retrouver jusqu'aux noms des variables utilisées dans le code initial. En fait, ceux-ci sont embarqués en clair dans le *code object* :

```
:::console
% strings unpycme.pyc
This is an example python script
Current file name is '%s'(
__file__(
filename(
unpycme.pyt
print_filename
__main__N(
```

4. Documentation du module 'dis'
5. https ://github.com/wibiti/uncompyle2
6. https ://code.google.com/p/unpyc3

```
__doc__R
__name__(
unpycme.pyt
<module>
```

La question qui se pose alors est la suivante : **comment un développeur peut-il protéger ses ".pyc" contre la décompilation ?**

On peut compter trois techniques principales pour cela, impliquant d'empaqueter une version *customisée* de CPython. Comme nous allons le voir dans la suite, ces trois techniques sont utilisées conjointement par Dropbox.

## 1.3. La modification du magic number

La technique la plus simple pour freiner les décompilateurs est de modifier le code d'`import.c` pour que celui-ci utilise un *magic number* différent de la valeur standard. Ceci suffit à rendre impuissant *uncompyle2*, puisqu'il n'a plus de moyen direct de déterminer le jeu d'*opcodes* à utiliser.

Néanmoins, si cette technique suffit à freiner les moins courageux des *script-kiddies*, elle reste facile à contourner. Il suffit de patcher les deux premiers octets du fichier ".pyc" en essayant successivement chaque version du jeu d'*opcodes* jusqu'à tomber sur la bonne version. Sachant qu'il existe à ce jour moins d'une cinquantaine de possibilités, cette solution linéaire est encore complètement envisageable. L'utilisation d'un *magic number* personnalisé n'offre donc pas, à elle seule, un niveau acceptable de sécurité.

### 1.3.1. L'opcode mapping

Une seconde technique, tout aussi simple que la précédente mais beaucoup plus difficile à déjouer, est l'*opcode mapping*. Elle consiste à modifier le jeu d'instructions du *bytecode* Python en permutant les valeurs des *opcodes*. Ceci peut se faire très simplement en modifiant le fichier `Include/opcode.h`[7]. Cette technique est extrêmement efficace contre les décompilateurs puisque même s'ils peuvent lire le fichier ".pyc", le *code object* qu'il contient leur sera complètement incompréhensible et le code Python décompilé (pour peu que la décompilation se termine sans erreur) ne voudra strictement rien dire.

Même si contourner l'*opcode mapping* présente beaucoup plus de difficultés que la technique précédente, cela reste tout à fait possible. Il suffit de récupérer les *bytecodes* d'un même code source générés par un interpréteur standard et par l'interpréteur mappé, de manière à pouvoir en comparer les *opcodes*. C'est entre autres la méthode appliquée par pyREtic[8].

Pour cela, il faut réussir à s'immiscer dans le *runtime* de Python. Dans certains cas, on peut réaliser cela simplement en renommant l'un des modules en ".pyc" de la solution à *reverser* (disons `hiddenmodule.pyc`) en `dummy.pyc`, puis en créant un nouveau module `hiddenmodule.py` contenant un code semblable au suivant :

7. http ://hg.python.org/cpython/file/869d50357a71/Include/opcode.h
8. Présentation de pyREtic .

```
:::python
import dummy

for x in dir(dummy):
    if x[:2] == "__":
        continue

    print "%s mirroring %s.%s" % (x, dummy.__file__, x)

    exec("%s = dummy.%s" % (x, x))

# ... insert your bytecode dumping code here ...
```

Lorsque le programme cible sera lancé et qu'il importera `hiddenmodule`, ce sera le module injecté qui sera chargé et compilé par Python. Ce module injecté pourra ensuite importer tout le contenu du module originel (renommé en `dummy`) et en exposer l'API publique, de manière à ce que le programme puisse continuer à tourner sans être impacté par l'injection. Une fois cette base acquise, nous avons le champ libre pour faire ce que bon nous semble dans l'environnement de l'application : il suffit d'en écrire le code à la suite de `hiddenmodule.py`.

Depuis le runtime Python, il est possible d'accéder au *bytecode* de n'importe quel objet, comme en atteste l'exemple suivant dans un shell Python 2.7 :

```
:::pycon
>>> def add(a, b):
...     return a + b
...
>>> add.__code__.co_code
'|\x00\x00|\x01\x00\x17S'
```

Rien ne nous empêche alors de charger un module contenant des fonctions couvrant la totalité du jeu d'instructions de CPython, et de "*dumper*" (sauvegarder) le bytecode correspondant dans un fichier. En réalisant cette opération à la fois depuis l'interpréteur standard et depuis l'interpréteur modifié, nous obtenons deux jeux d'*opcodes* "synchronisés", desquels on peut déduire une table de traduction de façon triviale. Il ne reste plus qu'à convertir les fichiers ".pyc" obfusqués grâce à cette table pour être enfin capables de les décompiler.

Toutefois, l'interpréteur Python de Dropbox est fortifié contre ce style d'injections : toutes les fonctions permettant de charger un fichier source en ".py" ont été neutralisées, et, comme nous le verrons plus loin, l'attribut `co_code` des *code objects* a été rendu inaccessible depuis le runtime Python, en modifiant la ligne correspondante du tableau `code_memberlist` dans le fichier `Objects/codeobject.c`[9]. Il va donc falloir utiliser un autre vecteur d'attaque.

9. http ://hg.python.org/cpython/file/25211a22228b/Objects/codeobject.c#l203

### 1.4. Le chiffrement du bytecode

Une troisième technique couramment employée est de chiffrer le bytecode dans les ".pyc". Dans le code de Dropbox, un appel à une fonction de déchiffrement a été rajouté au niveau de la fonction `r_object()` (dans le fichier `Python/marshal.c`[10]). On peut par ailleurs constater ce chiffrement en remarquant qu'il est impossible d'extraire la moindre chaîne de caractères valide des fichiers ".pyc" de Dropbox

```
:::console
% strings dropbox_modules/dropsyncore.pyc |head
7Qc8
2l'v]
B_DTh
]1@g
pkT_
l!M=
 ;\_3
ZH'L
Y_gy
/tNr
```

Cette fonction de déchiffrement étant *inlinée* dans le code de la fonction `r_object()` sous GNU-Linux, il est difficilement envisageable de chercher à la réutiliser. Cette sécurité interdit tout espoir de trouver une solution agissant uniquement sur les ".pyc". Le seul moyen de s'en défaire est de réussir une injection dans le *runtime* Python, au sein duquel le bytecode porté par les objets est nécessairement déchiffré pour être interprété par la machine virtuelle.

## 2. L'injection de code

Maintenant qu'il est clairement établi que nous ne pourrons pas nous passer d'une injection de code dans l'interpréteur de Dropbox, voyons ensemble comment celle-ci est possible. Cette injection fonctionne actuellement sur la plupart des programmes embarquant CPython, y compris les distributions standard. Son principe est de nous immiscer dans le *runtime* C de CPython, pour ensuite prendre la main sur le GIL[11] et injecter un code Python sous forme d'une chaîne de caractères dans l'environnement d'exécution courant. Nous nous contenterons ici de la présenter sous GNU-Linux.

### 2.1. LD_PRELOAD et dlsym

Il est possible (et même très facile) d'injecter une fonction dans un programme C sous les systèmes POSIX grâce à la variable d'environnement `LD_PRELOAD`. Prenons par exemple le petit programme suivant :

10. http ://hg.python.org/cpython/file/25211a22228b/Python/marshal.c#l772
11. Global Interpreter Lock

```
:::c
#include <stdio.h>
#include <string.h>

int
main (int argc, char *argv[])
{
    char *s = argv[0];
    size_t len = strlen(s);
    printf("'%s', %zd\n", s, len);
    return 0;
}
```

À l'exécution, il affiche le nom grâce auquel il a été lancé, ainsi que la longueur de la chaîne correspondante :

```
:::console
% gcc -o helloworld helloworld.c -fno-builtin
% ./helloworld
'./helloworld', 12
```

On remarquera que nous utilisons ici l'option `-fno-builtin` de gcc de manière à ce que l'appel à la fonction `strlen` ne soit pas *inliné*[12].

Imaginons maintenant que nous voulions détourner l'appel à `strlen` dans ce code afin d'exécuter la fonction suivante à la place :

```
:::c
#include <stdio.h>

size_t
strlen (const char *s)
{
    size_t len = 0;
    puts("I'm in your strlen, computing your string's length");

    if (!s)
        return 0;

    while (*(s++))
        len++;

    return len;
}
```

12. Ce phénomène a été discuté sur ce thread

Il suffit en réalité de compiler le code de la fonction que nous voulons injecter sous la forme d'une bibliothèque partagée, et d'utiliser `LD_PRELOAD`[13] pour préciser au système d'exploitation que nous voulons charger cette bibliothèque *avant* tous les autres symboles du programme. Durant l'exécution de celui-ci, ce sera notre fonction (et non celle de la bibliothèque standard) qui sera appelée, puisqu'elle sera la première à correspondre au symbole `strlen`.

```
:::console
% gcc -fPIC -shared -o strlen.so strlen.c
% LD_PRELOAD=strlen.so ./helloworld
I'm in your strlen, computing your string's length
'./helloworld', 12
```

Maintenant que nous sommes capables d'exécuter un code arbritraire dans un *runtime* C[14], il nous manque encore l'accès aux fonctions chargées dans celui-ci. Pour cela, nous pouvons utiliser la fonction `dlsym`[15], qui retourne l'adresse de l'objet associé à un symbole donné. Voici comment nous pourrions l'utiliser pour retrouver l'adresse de la fonction `strlen` originale :

```
:::c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

size_t
strlen (const char *s)
{
    size_t (*strlen_) (const char*) = NULL;
    puts("I'm in your strlen, computing your string's length");

    strlen_ = dlsym(RTLD_NEXT, "strlen");
    if (!strlen_) {
        fprintf(stderr, "'strlen' not found!\n");
        exit(EXIT_FAILURE);
    }

    printf("Found 'strlen' at %p\n", strlen_);
    return strlen_(s);
}
```

surchargée... Ceci fait de `strlen` un choix particulièrement judicieux.

L'utilisation de cette fonction requiert la définition de la constante `_GNU_SOURCE` ainsi que de lier notre bibliothèque partagée avec *dl.so* :

13. man (8) ld.so
14. Pour peu que le programme utilise au moins une fois notre fonction
15. man (3) dlsym

```
:::console
% gcc -D_GNU_SOURCE -fPIC -shared -o strlen.so strlen.c -ldl
% LD_PRELOAD=strlen.so ./helloworld
I'm in your strlen, computing your string's length
Found 'strlen' at 0x7fe7a031cbe0
'./helloworld', 12
```

Le flag `RTLD_NEXT` permet de chercher le *prochain* symbole `strlen` après le symbole courant. Dans la suite, nous utiliserons le flag `RTLD_DEFAULT` afin de trouver le *premier* symbole dans l'ordre de recherche par défaut.

Nous avons maintenant toutes les armes en main pour nous immiscer dans le *runtime* de Dropbox. Voyons maintenant comment accéder à son environnement d'exécution Python.

## 2.2. PyRun_SimpleString

Comme on l'a évoqué plus tôt, Dropbox s'est protégé contre les injections de code en neutralisant toutes les fonctions permettant de charger et d'exécuter un fichier au format ".py". Étant donné que ce fait a été établi dans l'article du WOOT'13, nous n'envisagerons même pas cette piste.

Par chance, ils ont omis d'en faire de même pour les fonctions de la famille `PyRun_SimpleString`[16]. Cette fonction de l'API C de CPython permet d'exécuter du code Python en le passant dans une chaîne de caractères. Ce sera notre point d'entrée. Le programme d'injection suivant surcharge la fonction `strlen` de manière à lancer un nouveau thread lors du premier appel. Ce thread commence par s'endormir quelques secondes (pour attendre que l'environnement Python soit correctement initialisé), puis récupère les adresses des fonctions nécessaires à prendre (et rendre) la main sur le GIL, ainsi que la fonction `PyRun_SimpleString`, grâce à laquelle nous exécutons une ligne de code Python pour vérifer que l'injection a fonctionné :

```
:::c
#include <Python.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dlfcn.h>
#include <pthread.h>

PyGILState_STATE (*PyGILState_Ensure_) (void) = NULL;
void (*PyGILState_Release_) (PyGILState_STATE) = NULL;
int (*PyRun_SimpleString_) (const char*) = NULL;


void*
findsym(void* handle, const char *sym)
{
    void *p = dlsym(handle, sym);
```

```
18      if (!p) {
19          fprintf(stderr, "pynject: %s not found!\n", sym);
20          exit(EXIT_FAILURE);
21      }
22      return p;
23  }
24
25
26  void*
27  thread(void* arg)
28  {
29      int ret;
30      PyGILState_STATE gstate;
31
32      puts("pynject: thread started");
33
34      while((ret = sleep(2))){}
35
36      PyGILState_Ensure_  = findsym(RTLD_DEFAULT,
            "PyGILState_Ensure");
37      PyGILState_Release_ = findsym(RTLD_DEFAULT,
            "PyGILState_Release");
38      PyRun_SimpleString_ = findsym(RTLD_DEFAULT,
            "PyRun_SimpleString");
39
40      gstate = PyGILState_Ensure_();
41      PyRun_SimpleString_("print 'pynject: success!'");
42      PyGILState_Release_(gstate);
43
44      puts("pynject: thread stopped");
45      return NULL;
46  }
47
48
49  /**
50   * Injector bootstrap routine.
51   * The standard C strlen function is overriden so that upon first
         call,
52   * it spawns the injector code in a new thread inside the target
         runtime.
53   */
54  pthread_t pthread;
55  size_t (*strlen_) (const char*) = NULL;
56
57  size_t
58  strlen (const char *s)
59  {
60      if (!strlen_) {
61          strlen_ = findsym(RTLD_NEXT, "strlen");
62          pthread_create(&pthread, NULL, thread, NULL);
```

```
63      }
64
65      return strlen_(s);
66  }
```

sont *a priori* pas nécessaires à l'exécution de Dropbox, donc il est extrêmement probable que les versions à venir ne comporteront plus cette faille dont la réparation est triviale.

Après compilation, nous pouvons vérifier le fonctionnement de ce code en tentant directement une injection dans le *runtime* de Dropbox :

```
1  :::console
2  % LD_PRELOAD=pynject.so ~/.dropbox-dist/dropbox
3  pynject: thread started
4  pynject: success!
5  pynject: thread stopped
6  ^C
7  %
```

L'affichage n'est guère impressionnant. Néanmoins, la seconde ligne ayant été générée par le code Python `print 'pynject: success!'`, cela suffit à conclure que nous avons réussi à « mettre un pied dans la porte ». Il ne nous reste plus qu'à nous donner un moyen d'injecter un fichier Python arbitraire plutôt qu'une chaîne de caractères codée en dur.

Remarquons déjà qu'il est possible d'embarquer un programme Python dans le segment `.data` d'un fichier objet au format Elf grâce à `objcopy`[17]. Prenons par exemple le script suivant (que nous agrandirons par la suite) :

```
1  :::python
2  # pynject.py
3  import sys
4
5  sys.stdout.write('pynject: current runtime is Python %s\n' %
       sys.version)
```

Convertissons maintenant ce script en un Elf nommé "pynject.data" :

```
1  :::console
2  % objcopy -I binary -O elf64-x86-64 -B i386:x86-64 pynject.py
       pynject.data
3  % file pynject.data
```

16. Négligence de ses possibilités, ou bien simple oubli ? Ces fonctions ne
17. man (1) objcopy

```
pynject.data: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
% objdump -x pynject.data

pynject.data:     file format elf64-x86-64
pynject.data
architecture: i386:x86-64, flags 0x00000010:
HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .data         00000268  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 g       .data  0000000000000000 _binary_pynject_py_start
0000000000000268 g       .data  0000000000000000 _binary_pynject_py_end
0000000000000268 g       *ABS*  0000000000000000 _binary_pynject_py_size
```

Nous remarquons que le segment .data contient trois symboles :

— `_binary_pynject_py_start` marque sur le début des données du script,
— `_binary_pynject_py_end` marque la fin des données,
— `_binary_pynject_py_size` nous donne la taille de la donnée (268 octets).

En liant ce fichier binaire à notre bibliothèque, nous pourrons accéder directement à ses données dans le code C, donc nous pouvons modifier notre injecteur pour qu'il passe ce script à la fonction `PyRun_SimpleString` :

```
:::c
extern char _binary_pynject_py_start;

void*
thread (void* arg);
{
    /* ... */
    gstate = PyGILState_Ensure_();
    PyRun_SimpleString_(&_binary_pynject_py_start);
    PyGILState_Release_(gstate);
    /* ... */
    return NULL
}
```

Essayons :

```
:::console
% gcc -c -o pynject.o pynject.c -D_GNU_SOURCE -fPIC -Wall -ansi
    -I/usr/include/python2.7 -pthread
% gcc -o pynject.so pynject.o pynject.data -shared -pthread -ldl
% LD_PRELOAD=pynject.so ~/.dropbox-dist/dropbox
pynject: thread started
pynject: current runtime is Python 2.7.3 (default, Apr  5 2013,
    15:07:41)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)]
pynject: thread stopped
```

Le tour est joué. On remarquera au passage que Dropbox a *buildé* son interpréteur CPython avec une version antédiluvienne de GCC, sur une Ubuntu de presque 10 ans d'âge, mais qu'il est basé sur la dernière version stable de CPython 2.

Pour éviter d'avoir à recompiler notre injecteur chaque fois que nous voudrons tester quelque chose, nous allons prendre quelques libertés par rapport à la méthode de Kholia en ajoutant à notre script Python la possibilité de charger et compiler n'importe quel autre fichier ".py", et de le lancer dans l'environnement courant. Profitons-en aussi pour nous affranchir des différences entre Python2 et Python3 en définissant des fonctions d'IO portables sans pour autant surcharger `print` ou `input` dans l'environnement global : rappelez-vous que toutes les modifications que nous apportons à l'environnement sont visibles depuis le reste du *runtime* que nous infiltrons, donc il vaut mieux que nous limitions au maximum les risques d'effets de bord.

```
:::python
# pynject.py
from __future__ import with_statement
import sys
import os

# python3-ready helper functions
# we MUST NOT replace the current 'print' or 'input' global symbols
gets = __builtins__.__dict__.get('raw_input', input)
puts = lambda *args: sys.stdout.write(' '.join(args) + '\n')

# check PYNJECT_FILE environment var for a python filepath to run
py_file = os.getenv('PYNJECT_FILE')
if py_file:
    puts("pynject: loading file '%s'" % py_file)
    with open(py_file) as fh:
        codeobj = compile(fh.read(), py_file, 'exec')
        eval(codeobj)
else:
    puts("pynject: current runtime is Python %s" % sys.version)
```

Tant que nous y sommes, écrivons aussi un Makefile afin de nous épargner la tâche fastidieuse de tout recompiler à la main :

```
:::Makefile
ARCH = $(shell getconf LONG_BIT)

CC = cc
LD = gcc
OBJ = pynject.o pynject.data
CFLAGS = -D_GNU_SOURCE -fPIC -Wall -ansi -I/usr/include/python2.7
    -pthread
LDFLAGS = -shared -pthread -ldl
OFLAGS_32 = -O elf32-i386 -B i386
OFLAGS_64 = -O elf64-x86-64 -B i386:x86-64

.PHONY: clean all

%.data: %.py
    objcopy -I binary $(OFLAGS_$(ARCH)) $< $@

%.o: %.c
    $(CC) -c -o $@ $< $(CFLAGS)

pynject.so: $(OBJ)
    $(LD) -o $@ $^ $(LDFLAGS)

all: pynject.so

clean:
    rm -f *.o *.data *.so
```

Voilà. Nous disposons maintenant d'un injecteur générique, que nous pouvons utiliser pour lancer un script Python arbitraire dans le *runtime* de n'importe quelle solution basée sur CPython (2 ou 3). Essayons-le :

```
:::console
% cat test.py
puts("im in ur dropbox stealin ur opcodz :)")

% PYNJECT_FILE=test.py LD_PRELOAD=pynject.so
    ~/.dropbox-dist/dropbox
pynject: thread started
pynject: loading file 'test.py'
im in ur dropbox stealin ur opcodz :)
pynject: thread stopped
```

Parfait ! Nous sommes maintenant équipés pour nous attaquer à l'*opcode mapping* de Dropbox.

## 3. Les code objects et leur attribut co_code

Afin de poursuivre nos expérimentations, nous allons nous offrir une petite boucle interactive (ou *REPL*, pour *Read Eval Print Loop*) semblable à la console standard Python, que nous pourrons faire tourner dans l'environnement infiltré.

```
:::python
# repl.py
puts("pynject: running interactive loop. [Ctrl+D] to exit")

while True:
    input_str = ''
    try:
        input_str = gets('>>> ').rstrip()
        if input_str.endswith(':'):
            blockline = gets('... ').rstrip()
            while blockline:
                input_str += blockline + '\r\n'
                blockline = gets('... ').rstrip()
    except EOFError:
        break

    try:
        codeobj = compile(input_str, '<REPL>', 'single')
        res = eval(codeobj)
        if res:
            puts(repr(res))
    except Exception as e:
        puts(e.__class__.__name__, str(e))
```

Utilisons-la pour afficher le bytecode d'une fonction basique, comme nous l'avons fait plus haut dans un interpréteur Python 2. Comme vous allez le voir, Dropbox n'a pas encore dit son dernier mot :

```
:::pycon
% PYNJECT_FILE=repl.py LD_PRELOAD=pynject.so
    ~/.dropbox-dist/dropbox
pynject: thread started
pynject: loading file 'repl.py'
pynject: running interactive loop. [Ctrl+D] to exit
>>> def add(a, b):
...     return a + b
...
>>> add
<function add at 0x7f70880359b8>
>>> add.__code__
<code object add at 0x7f7088031f38, file "<REPL>", line 1>
```

```
>>> add.__code__.co_code
AttributeError 'code' object has no attribute 'co_code'
>>> pynject: thread stopped
```

Voilà qui est ennuyeux. Comme on l'a dit plus haut, Dropbox a rendu inaccessible l'attribut `co_code` des *code_objects* dans son *runtime* Python. Il va donc falloir que nous fassions encore quelques efforts avant de crier victoire.

Les *code objects* sont des objets natifs (*builtins*) dans CPython. On pourra trouver leur déclaration dans le fichier `Include/code.h`[18] :

```
:::c
/* Bytecode object */
typedef struct {
    PyObject_HEAD
    int co_argcount;                    /* #arguments, except *args
        */
    int co_nlocals;                       /* #local variables */
    int co_stacksize;                    /* #entries needed for
        evaluation stack */
    int co_flags;                         /* CO_..., see below */
    PyObject *co_code;                     /* instruction opcodes */
    PyObject *co_consts;              /* list (constants used) */
    PyObject *co_names;                     /* list of strings
        (names used) */
    PyObject *co_varnames;              /* tuple of strings (local
        variable names) */
    PyObject *co_freevars;              /* tuple of strings (free
        variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable
        names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;              /* string (where it was
        loaded from) */
    PyObject *co_name;                     /* string (name, for
        reference) */
    int co_firstlineno;                     /* first source line
        number */
    PyObject *co_lnotab;              /* string (encoding
        addr<->lineno mapping) See
                                       Objects/lnotab_notes.txt for
                                          details. */
    void *co_zombieframe;     /* for optimization only (see
        frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code
        objects */
} PyCodeObject;

```

```
25 /* ... */
26
27 /* Public interface */
28 PyAPI_FUNC(PyCodeObject *) PyCode_New(
29     int, int, int, int, PyObject *, PyObject *, PyObject *,
           PyObject *,
30     PyObject *, PyObject *, PyObject *, PyObject *, int, PyObject
           *);
31         /* same as struct above */
```

Bien qu'on ne puisse pas accéder à cet attribut *co_code* depuis Python, il est impensable que Dropbox l'ait purement et simplement supprimé : ils en ont forcément besoin pour exécuter leur code. La solution va donc être d'aller le chercher dans un code en C que l'on rendra accessible depuis Python. En d'autres termes, nous allons créer un module Python en C, ainsi qu'une fonction dont le rôle sera uniquement de retourner le *bytecode* d'un *code object*.

Avant de foncer bille en tête dans son implémentation, gagnons un peu de temps : il est probable que Dropbox ait mélangé l'ordre des attributs dans sa structure `PyCodeObject` de manière à nous compliquer la tâche. C'est en tout cas ce qu'ont cru observer Kholia et Węgrzyn (les commentaires qu'ils ont laissés dans leur code laissent croire qu'ils n'en sont pas certains), et personnellement c'est ce que j'aurais fait à la place de Dropbox. Ainsi, plutôt que de retourner simplement `object->co_code`, nous allons préalablement écrire une fonction dont le but est de déterminer l'*offset* de cet attribut de façon empirique.

Pour ce faire, en instanciant une telle structure grâce au constructeur `PyCode_New`, il nous suffit de la parcourir en comparant chaque adresse avec le `PyObject*` que nous aurons passé au constructeur en guise de `co_code`.

Attention : la fonction suivante n'est pas compatible avec Python 3. Dans l'API C de Python 3, `PyCode_New` prend un argument entier supplémentaire.

```
1  :::c
2  volatile size_t co_code_offset;
3
4  void
5  find_co_code_offset()
6  {
7      PyObject *codestr;
8      PyObject *tuple;
9      PyObject *string;
10     PyCodeObject *code;
11     char *pos;
12     char *end;
13
14     if (co_code_offset)
15         return;
16
```

18. http ://hg.python.org/cpython/file/869d50357a71/Include/code.h

```
    codestr = PyString_FromString("deadbeef");
    string = PyString_FromString("");
    tuple = PyTuple_New(0);


    code = PyCode_New(0, 0, 0, 0, codestr,
                tuple, tuple, tuple, tuple, tuple, string, string,
                    0, string);

    co_code_offset = 0;
    end = (char*) code + sizeof(*code);

    for (pos = (char*) code;
         *((PyObject**) pos) != codestr && pos != end;
         pos++) {}

    if (pos == end) {
        fprintf(stderr, "pynject: Couldn't find co_code
            offset!\n");
    } else {
        co_code_offset = pos - (char*) code;
        printf("pynject: co_code_offset is %zd\n", co_code_offset);
    }
}
```

Notez que l'on n'a pas eu besoin de récupérer les pointeurs sur les fonctions de l'API C de Python : étant donné que nous avons inclus le fichier Python.h, les symboles sont définis à la compilation. Et puisque que nous ne lions pas notre bibliothèque à la libpython, ces symboles seront résolus dans l'environnement courant au moment de l'appel à la fonction `find_co_code_offset`.

Ce n'est certainement pas la fonction la plus propre au monde, mais elle a au moins le mérite de nous indiquer que l'attribut `co_code` se trouve à l'*offset* 96 dans le code de Dropbox alors que dans l'API standard, on irait le chercher à l'*offset* 32.

Nous pouvons maintenant créer notre fonction `co_code` et la rendre accessible en Python :

```
:::c
static PyObject*
pynject_co_code (PyObject *self, PyObject *args)
{
    PyCodeObject *code = NULL;
    PyObject *co_code = NULL;

    find_co_code_offset();

    if (!co_code_offset)
        goto err;
```

```
12
13      if (!PyArg_ParseTuple(args, "O:co_code", &code))
14          goto err;
15
16      if (!code)
17          goto err;
18
19      co_code = (*(PyObject**)(((char*) code) + co_code_offset));
20      Py_XINCREF(co_code);
21
22  err:
23      return co_code;
24  }
25
26
27  static PyMethodDef pynject_methods[] = {
28      {"co_code", pynject_co_code, METH_VARARGS, "Get co_code from
            code object"},
29      {NULL, NULL, 0, NULL}
30  };
31
32  /* ... */
33
34  void*
35  thread (void* arg)
36  {
37      /* ... */
38      gstate = PyGILState_Ensure_();
39
40      Py_InitModule("pynject", pynject_methods);
41
42      PyRun_SimpleString_(&_binary_pynject_py_start);
43      PyGILState_Release_(gstate);
44
45      /* ... */
46  }
```

À présent, étudions l'*opcode mapping* opéré par Dropbox :

```
1  :::pycon
2  % python
3  Python 2.7.3 (default, Apr 10 2013, 06:20:15)
4  [GCC 4.6.3] on linux2
5  Type "help", "copyright", "credits" or "license" for more
       information.
6  >>> add = lambda a, b: a + b
7  >>> map(ord, add.__code__.co_code)
8  [124, 0, 0, 124, 1, 0, 23, 83]
```

```
>>>

% PYNJECT_FILE=repl.py LD_PRELOAD=pynject.so
    ~/.dropbox-dist/dropbox
pynject: thread started
pynject: loading file 'repl.py'
pynject: running interactive loop. [Ctrl+D] to exit
>>> from pynject import co_code
>>> add = lambda a, b: a + b
>>> map(ord, co_code(add.__code__))
pynject: co_code_offset is 96
[102, 0, 0, 102, 1, 0, 52, 66]
>>> pynject: thread stopped
```

Nous pouvons analyser ces deux bytecodes en nous aidant du module standard `dis` pour obtenir la signification des opcodes :

```
:::text
STANDARD        DROPBOX         Signification

124             102             LOAD_FAST
0 (16 bits)     0               (a : args[0])
124             102             LOAD_FAST
1 (16 bits)     1               (b : args[1])
23              52              BINARY_ADD
83              66              RETURN_VALUE
```

On remarque que les seuls octets qui changent dans ce code sont les opcodes. Ce qui a trait aux données reste exactement identique. Nous pouvons donc facilement en déduire que Dropbox a substitué 102 à 124, 52 à 23 et 66 à 83 dans la déclaration des opcodes.

Nous pourrions nous amuser à appliquer la même logique sur un ensemble large de fonctions, couvrant la totalité du vocabulaire du bytecode Python, mais nous allons nous en épargner la peine dans cet article (ça se scripte très facilement), et récupérer directement le résultat :

```
:::python
opcode_map = {
    0: 0,       1: 87,      2: 66,      3: 59,      4: 25,      5:
        27,
    6: 55,      7: 62,      8: 57,      9: 71,      10: 79,     11:
        75,
    12: 21,     13: 4,      14: 72,     15: 1,      16: 30,     17:
        31,
    18: 32,     19: 33,     20: 70,     21: 65,     22: 63,     23:
        78,
```

```
    24: 77,     25: 13,     26: 86,     27: 58,     28: 19,     29:
        56,
    30: 29,     31: 60,     32: 28,     33: 73,     34: 15,     35:
        74,
    36: 20,     37: 81,     38: 12,     39: 68,     40: 80,     41:
        22,
    42: 89,     43: 26,     44: 50,     45: 51,     46: 52,     47:
        53,
    48: 10,     49: 5,      50: 64,     51: 82,     52: 23,     53:
        9,
    54: 11,     55: 24,     56: 84,     57: 67,     58: 76,     59:
        2,
    60: 3,      61: 40,     62: 41,     63: 42,     64: 43,     65:
        85,
    66: 83,     67: 88,     69: 61,     70: 54,     80: 116,    81:
        126,
    82: 100,    83: 94,     84: 120,    85: 122,    86: 132,    87:
        133,
    88: 105,    89: 101,    90: 102,    91: 93,     92: 125,    94:
        95,
    95: 134,    96: 106,    97: 96,     98: 108,    99: 109,
        101: 130,
    102: 124,   103: 92,    104: 91,    105: 90,    106: 119,
        107: 135,
    108: 98,    109: 136,   110: 137,   111: 107,   112: 131,
        113: 113,
    114: 99,    115: 97,    116: 121,   117: 103,   118: 104,
        119: 110,
    120: 111,   121: 115,   122: 112,   123: 114,   133: 140,
        134: 141,
    135: 142,   136: 143,   140: 145,   141: 146,   142: 147
}
```

Il ne nous reste plus maintenant qu'à :

- charger les modules chiffrés qui nous intéressent dans le *runtime* de Dropbox de manière à en récupérer le code déchiffré,
- traduire le bytecode,
- écrire le résultat dans des fichiers ".pyc" compatibles avec l'interpréteur CPython standard.
- Décompiler ces fichiers ".pyc" avec uncompyle2.

## 4. Le marshalling de fichiers ".pyc"

Dans le jargon Python, le *marshalling* désigne l'interaction avec les fichiers ".pyc". Dans la distribution standard, on retrouvera ces fonctionnalités dans le module `marshal`[19] pour une interaction depuis Python, ou bien directement avec les fonctions `PyMarshal_*`[20] depuis l'API

C.

Étant donné que Dropbox a *patché* son interpréteur pour que les ".pyc" soient chiffrés, la première approche est une voie sans issue :

```
:::pycon
pynject: thread started
pynject: loading file 'repl.py'
pynject: running interactive loop. [Ctrl+D] to exit
>>> import marshal
>>> fh = open('../dropbox_modules/__main__dropbox__.pyc')
>>> marshal.load(fh)
ValueError bad marshal data (unknown type code)
>>>
```

Nous devons donc écrire une nouvelle fonction en C, comme pour récupérer l'attribut `co_code`, afin d'être capables de charger les fichiers ".pyc".

```
:::c
long (*PyMarshal_ReadLongFromFile_) (FILE*) = NULL;
PyObject* (*PyMarshal_ReadLastObjectFromFile_) (FILE*) = NULL;

static PyObject*
pynject_load_pyc (PyObject *self, PyObject *args)
{
    const char *pyc_path = NULL;
    FILE *pyc = NULL;
    PyObject *obj = NULL;


    PyMarshal_ReadLongFromFile_ = findsym(RTLD_DEFAULT,
                                      "PyMarshal_ReadLongFromFile");
    PyMarshal_ReadLastObjectFromFile_ = findsym(RTLD_DEFAULT,

                                          "PyMarshal_ReadLastObjectFromFile");


    if (!PyArg_ParseTuple(args, "s:load_pyc", &pyc_path))
        goto err;

    pyc = fopen(pyc_path, "rb");
    if (!pyc) {
        perror("pynject: fopen");
        goto err;
    }
```

19. Documentation du module `marshal`
20. "Data marshalling support"

```
27
28      /* Read magic number and timestamp */
29      PyMarshal_ReadLongFromFile_(pyc);
30      PyMarshal_ReadLongFromFile_(pyc);
31
32      /* Read and decode the toplevel code object */
33      obj = PyMarshal_ReadLastObjectFromFile_(pyc);
34
35      fclose(pyc);
36
37  err:
38      return obj;
39  }
40
41  /* ... */
42
43  static PyMethodDef pynject_methods[] = {
44      {"co_code", pynject_co_code, METH_VARARGS, "Get co_code from
            code object"},
45      {"load_pyc", pynject_load_pyc, METH_VARARGS, "Read pyc and
            return code object"},
46      {NULL, NULL, 0, NULL}
47  };
```

Essayons-la :

```
1  :::pycon
2  pynject: thread started
3  pynject: loading file 'repl.py'
4  pynject: running interactive loop. [Ctrl+D] to exit
5  >>> from pynject import load_pyc
6  >>> code = load_pyc('../dropbox_modules/__main__dropbox__.pyc')
7  >>> code
8  <code object <module> at 0x7f9ce823b2b8, file
       "__main__dropbox__.py", line 6>
```

Bien. Nous arrivons maintenant à la dernière ligne droite : le *remapping* du bytecode et la décompilation finale.

La première étape est plutôt facile. Étant donné que nous avons une table de traduction, il suffit de l'utiliser en faisant attention toutefois à ne pas *remapper* les arguments des opcodes : le bytecode Python n'ayant pas été conçu au hasard, nous savons que tous les opcodes prenant un argument sur 16 bits sont supérieurs à 90.

```
1  :::c
2  /* Include/opcode.h */
```

```
/* ... */

#define END_FINALLY        88
#define BUILD_CLASS        89

#define HAVE_ARGUMENT        90        /* Opcodes from here have an
    argument: */

#define STORE_NAME        90        /* Index in name list */
#define DELETE_NAME        91        /* "" */

/* ... */
```

En dehors de cela, la fonction est triviale :

```
:::python
# remap bytecode string to standard opcodes
def remap(code):
    code = bytearray(code)
    i = 0
    while i < len(code):
        code[i] = opcode_map[code[i]]
        # if opcode is >= 90, skip the arguments
        i += (1 if code[i] < 90 else 3)
    return str(code)
```

Pour écrire le fichier ".pyc", en revanche, il va nous falloir contourner encore l'environnement de Dropbox : ses fonctions de *marshalling* ne nous seront d'aucune aide. C'est en se tournant vers un autre interpréteur Python, PyPy[21], que nous trouverons notre solution. PyPy est un interpréteur Python en Python. Celui-ci propose à peu près les mêmes fonctionnalités que CPython, mais puisque son code-source est écrit en Python, il est aisé de reprendre son implémentation du *marshalling* pour l'injecter dans l'environnement de Dropbox.

On trouvera notre bonheur dans le module `lib_pypy/_marshal.py`[22], avec la classe `_Marshaller`. Nous allons l'emprunter, et corriger quelque peu sa méthode servant à *dumper* les *code objects* de façon à ce qu'elle "remappe" le bytecode au passage :

```
:::python
class _Marshaller:
    # ... same as PyPy's ...


# _Marshaller.dump_code overload
def dump_codeobj(self, x):
```

21. http ://pypy.org/

```
8       self._write(TYPE_CODE)
9       self.w_long(x.co_argcount)
10      self.w_long(x.co_nlocals)
11      self.w_long(x.co_stacksize)
12      self.w_long(x.co_flags)
13      self.dump(remap(pynject.co_code(x)))
14      self.dump(x.co_consts)
15      self.dump(x.co_names)
16      self.dump(x.co_varnames)
17      self.dump(x.co_freevars)
18      self.dump(x.co_cellvars)
19      self.dump(x.co_filename)
20      self.dump(x.co_name)
21      self.w_long(x.co_firstlineno)
22      self.dump(x.co_lnotab)
23
24  _Marshaller.dispatch[types.CodeType] = dump_codeobj
```

Il ne nous reste plus qu'à utiliser ce code pour déchiffrer les ".pyc" de Dropbox :

```
1   :::python
2   def process_pyc(filename, dest=None):
3       if not dest:
4           dest = filename + "_clear"
5
6       obj = pynject.load_pyc(filename)          # Decrypt encrypted
            .pyc
7       puts("Dumping %s" % dest)
8       with open(dest, 'wb') as pyc:
9           pyc.write('\x03\xf3\r\n')             # Python 2.7.3 magic
                number
10          pyc.write('\x00\x00\x00\x00')         # null (useless)
                timestamp
11          _Marshaller(pyc.write).dump(obj)      # Aaaaand... that's
                it.
12
13  pyc_dir = os.getenv('DECRYPT_DIR')
14  if pyc_dir:
15      import os.path
16      py_dir = pyc_dir + "_reversed"
17      try:
18          os.mkdir(py_dir)
19      except OSError:
20          pass # directory already exists
21      for fname in os.listdir(pyc_dir):
22          if not fname.endswith('pyc'):
```

22. https ://github.com/pypy/pypy/blob/master/lib_pypy/_marshal.py

```
            continue
        process_pyc(os.path.join(pyc_dir, fname),
                    os.path.join(py_dir, fname))
```

C'est le moment de récolter le fruit de nos efforts :

```
:::console
% DECRYPT_DIR=dropbox_modules LD_PRELOAD=pynject/pynject.so
    ~/.dropbox-dist/dropbox
pynject: thread started
pynject: current runtime is Python 2.7.3 (default, Apr  5 2013,
    15:07:41)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)]
Dumping dropbox_modules_reversed/cookielib.pyc
pynject: co_code_offset is 96
Dumping dropbox_modules_reversed/__future__.pyc
Dumping dropbox_modules_reversed/posixpath.pyc
Dumping dropbox_modules_reversed/contextlib.pyc
Dumping dropbox_modules_reversed/functools.pyc
Dumping dropbox_modules_reversed/quopri.pyc

[...]

% cd dropbox_modules_reversed
% uncompyle2 --py -o . *.pyc
# 2013.09.08 14:36:18 CEST
+++ okay decompyling _abcoll.pyc
+++ okay decompyling abc.pyc
+++ okay decompyling asynchat.pyc
+++ okay decompyling asyncore.pyc
+++ okay decompyling atexit.pyc

[...]

+++ okay decompyling webbrowser.pyc
+++ okay decompyling zipfile.pyc
# decompiled 118 files: 118 okay, 0 failed, 0 verify failed
# 2013.09.08 14:37:08 CEST
```

Mission accomplie.

## 5. Conclusion

Nous venons d'employer une série de techniques et d'outils relativement simples pour déchiffrer et décompiler le code de l'une des applications Python les plus fortifiées contre le *reversing*. Les intérêts de cette démonstration sont multiples.

Si vous êtes un développeur amateur ou débutant en Python, et que vous comptiez vous lancer dans un projet avec l'idée que vous vous mettrez en sécurité en essayant de masquer votre code-source, vous avez probablement été redirigé sur cet article dans le but de vous détromper. En réalité, les efforts que vous dépenserez à obfusquer votre code seront autant de temps perdu que vous pourriez employer à concevoir un programme plus robuste aux bugs et propice à l'évolution : Python n'est pas du tout un langage adapté à l'obfuscation.

D'une façon plus générale, Kholia et Węgrzyn ont conclu sur un souhait, celui que Dropbox s'ouvre enfin à la création d'un client open source qui n'aurait pas besoin de cacher ses bugs et ses failles "sous le tapis". C'est d'ailleurs l'un des plus grands cas d'emploi de techniques de rétro-ingénierie : permettre d'auditer un logiciel à source fermée, afin de s'assurer que celui-ci ne constitue pas une vulnérabilité pour un système donné.

Nous pourrions aussi ironiser sur le fait que le créateur de Python lui-même, Guido Van Rossum, travaille pour Dropbox depuis début 2013…

En ce qui me concerne, je me suis surtout beaucoup amusé à étudier ce *reversing* et j'espère que la lecture de cet article aura été aussi instructive pour vous que son écriture l'a été pour moi.