

Reversing Obfuscated Python Applications

Breaking the dropbox client on windows

Author: ExtremeCoders © 2014

E-mail: extremecoders@mail.com



According to Wikipedia, *"Dropbox is a file hosting service operated by Dropbox, Inc., headquartered in San Francisco, California, that offers cloud storage, file synchronization, personal cloud, and client software. Dropbox allows users to create a special folder on each of their computers, which Dropbox then synchronizes so that it appears to be the same folder (with the same contents) regardless of which computer is used to view it. Files placed in this folder also are accessible through a website and mobile phone applications"*

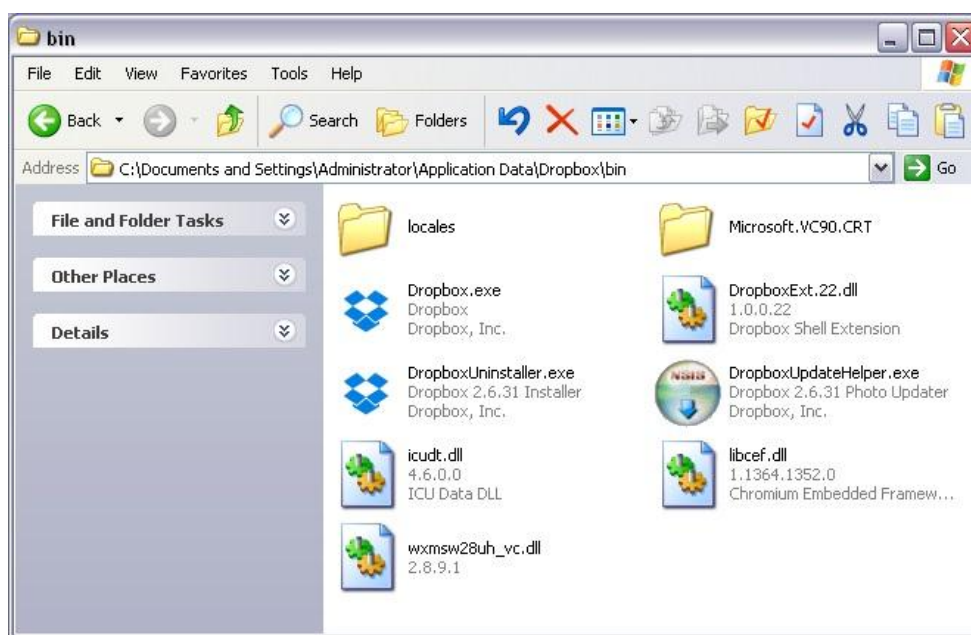
Dropbox provides client software for Microsoft Windows, Mac OS X, Linux, Android, iOS, BlackBerry OS and web browsers, as well as unofficial ports to Symbian, Windows Phone, and MeeGo.

The Dropbox client software is written in python so that a single codebase can be deployed to a wide variety of platform and architectures. Another benefit of using python is ease in coding and reduced time for testing and deployment. However python poses other problem too such as the relative ease in reversing & decompiling as compared to native applications. In case of a closed source application like dropbox this is a serious issue, and something must be done to prevent users from getting access to the source code.

In this regard, the dropbox client on Windows is shipped as an .exe file. The executable is generated using py2exe which serves two purposes - firstly, it becomes a lot easier for the end user to install than fiddling with a bunch of .pyc files and the second and the most important is it prevents over enthusiast users from peeking into the source.

So good luck and bon voyage on this reversing journey.

Introduction



After installing Dropbox (which installs silently without any user intervention) we can navigate to the above folder, to find the main dropbox binary. Since we know it is already written in python and has been packaged by *py2exe* we will not waste any time by running through a PE detection tool like PEiD.

Python code in the file system can reside in two forms. It may be either in plaintext *.py* or compiled *.pyc* form. In the latter case, the file is not directly readable by a text editor but can be run through a python bytecode decompiler in order to obtain the plaintext source code. In the case of dropbox, these *.pyc* files are packaged inside the executable. So our first step is to unpack the executable to get hold of the *.pyc* files.

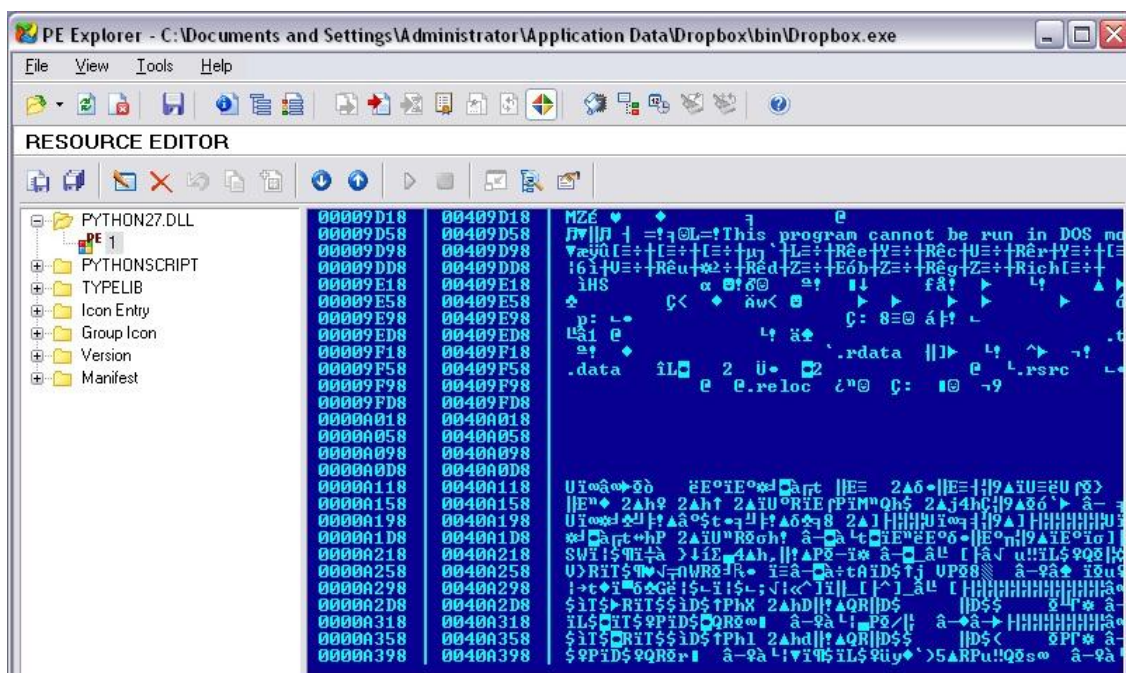
Unpacking the executable

Now let's discuss a bit about *py2exe* and its innards. It is a tool which packages python scripts into a windows executable along with an embedded python interpreter (actually I am oversimplifying here). This executable can then be run on a windows system as a standalone file without the necessity of installing python. All the necessary scripts required for the software to run are packaged within it.

Now the *.pyc* files necessary for the application to run are packaged as a *zip* archive. This archive is just concatenated at the end of the *py2exe* loader stub namely *run_w.exe* or *run.exe* for windows and console applications respectively. The python interpreter on the other hand is packaged as a resource.

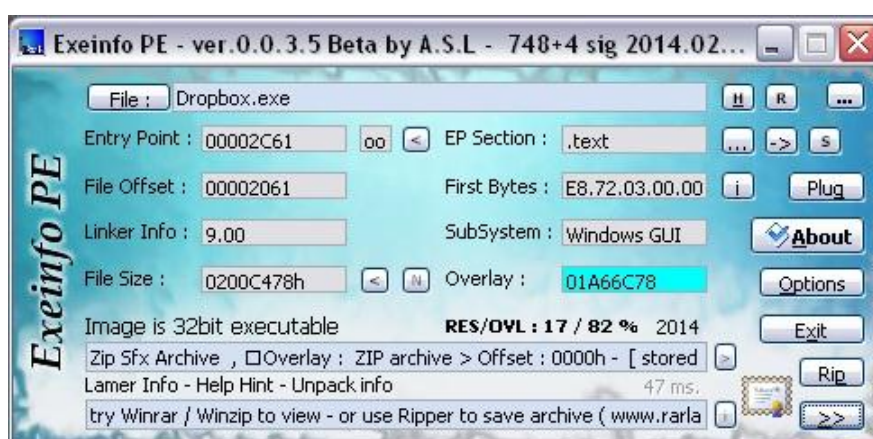
During runtime the executable fires up the embedded python interpreter. This embedded python interpreter is a *.dll* on windows. The dynamically linked library is loaded entirely from memory and since Windows does not allow a PE to be loaded from memory, the tool is provided with its own PE loader. We can look into the file *MemoryModule.c* within *py2exe* source code for the details.

So for unpacking we have to do two things – first, grab the *zip* archive containing the *.pyc* and then extract the python dll embedded as a resource within the executable. For the second objective, we can use any decent resource editor. Here I am using PE Explorer.



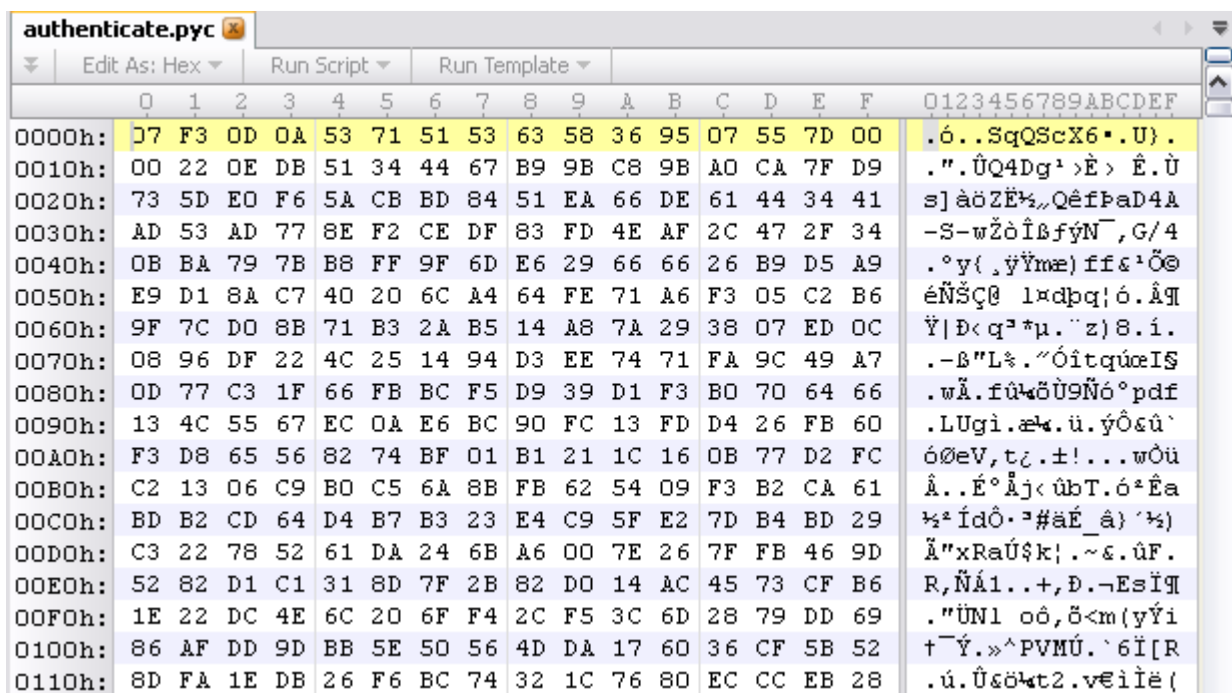
Note that besides the resource *PYTHON27.DLL*, there is another one named *PYTHONSCRIPT*. This contains a set of start-up scripts (actually they are not scripts, since they are compiled and not directly readable by a text editor) which are run before the application is initialized. The purpose of them is to set up some *import hook* which facilitates to load the *pyc* from within the executable. Normally without them, python can only load *.pyc* files from the file system or from a normal zip archive. Since the *pyc* files are packaged in an archive concatenated to the PE, it needs special treatment i.e. *import hooks* to load them. After *import hooks* have been set up, whenever python wants to import a module, the import hooks are called which bypasses the regular import mechanism loading the *pyc* files from the executable. So in short it acts like a proxy. The advantage of this, we do not need to have files residing on the system in order to load them. They can be anywhere!

Okay, now the first objective. For extracting the *zip* archive, we load the file into *exeinfo pe* and dump the overlay. If we want to automate some of the steps, we can use the tool *py2exe dumper*.



Inspecting the pyc files

Once we have extracted the embedded *zip* archive from the executable, we can see that it contains *.pyc* files. Opening any such file in a hex editor reveals that the file is encrypted. There are no readable strings at all. A normal *.pyc* file generally has some readable strings which are missing in this case. Further *.pyc* files begin with a 4 byte magic number followed by another 4 byte timestamp. In this case, it begins with the value *07 F3 0D 0A*, which is different from the normal python 2.7 magic value of *03 F3 0D 0A*. Obviously, we cannot decompile the file as is. We need to decrypt it; else no decompiler will show any interest in reading it.



Figuring the decryption process

Right now we are in the dark about the encryption algorithm used and how to decrypt it. The dropbox application must load these modules, which means internally it must decrypt them before it can work. So if we can grab them from memory after it has decrypted itself, we can get the decryption for free. Let's see if this method is feasible.

We will be coding a program in C which will embed the dropbox python interpreter (python27.dll) we obtained previously. The purpose of the program will be to run any python script in the context of dropbox. In order to compile and link we will use the header files and import libraries provided with the standard python 2.7 distribution. Let's name the output file as *embedder.exe*

```
#include "Python.h"
#include "marshal.h"
#include <windows.h>

void main(int argc, char *argv[]) //The script to run will be provided as an argument
{
    if(argc < 2)
```

```

{
    printf("No script specified to run\n");
    return;
}

HANDLE hand = CreateFileA(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

if(!hand)
{
    printf("Failed to open file %s\n", argv[1]);
    return;
}

char pathname[MAX_PATH];
GetModuleFileNameA(NULL, pathname, sizeof(pathname));
char *c = strrchr(pathname, '\\');
*c=0;

Py_NoSiteFlag = 1; //We do not need to load site modules
Py_SetPythonHome(pathname); //Setting the path to the python libraries
Py_Initialize();

//Allocating a buffer to hold the file's contents
void *buf = calloc(GetFileSize(hand, NULL) + 1, 1);

//Reading the file within the buffer
DWORD dummy;
ReadFile(hand, buf, GetFileSize(hand, NULL), &dummy, NULL);

CloseHandle(hand);

PyGILState_STATE gstate;
gstate = PyGILState_Ensure();
PyRun_SimpleString((char*) buf); //Running the python script located in the buffer
PyGILState_Release(gstate);
Py_Finalize();
free(buf);
}

```

Using the above program, we can run any python script in the context of the dropbox python interpreter. Now we will write a python script which will load those encrypted *pyc* files using the *marshal* module. Hopefully, that will decrypt it too. After that we will dump it back to disk.

```

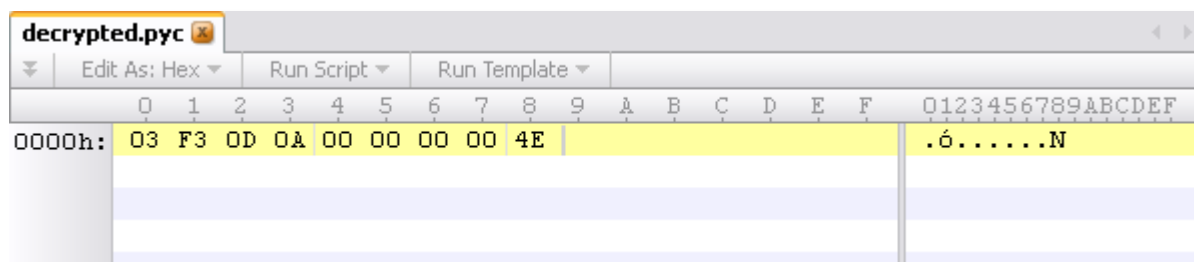
import marshal

infile = open('authenticate.pyc', 'rb')    # The encrypted pyc file
infile.seek(8) # Skip the header, it consists of magic value & timestamp of 4 bytes each
code_obj = marshal.load(infile)           # Unmarshal the file

outfile = open('decrypted.pyc', 'wb')     # The output file
outfile.write('\x03\xf3\x0d\x0a\x00\x00\x00\x00') # Write the header
marshal.dump(code_obj, outfile)           # Dump back to file
outfile.close()
infile.close()

```


So let's run the script using the C program embedding python. We need to pass the name of the script file as an argument. However the results are not encouraging. Indeed it generates a file *decrypted.pyc* but on examining its contents we see this.



The file is basically empty. Dumping to disk has failed. Only it has written the hex byte *4E* in addition to the header. We need to analyse the dropbox python interpreter to see why it has happened.

PyPy to the rescue

CPython provides a function *PyMarshal_WriteObjectToFile*, which dumps a code object to a file on disk. Internally, this calls another function *w_object* which does the actual work. In case of dropbox, *w_object* has been patched to disable marshalling of code objects to disk. So to get around the limitation we need an alternative. We need some code, preferably in python, which will do the marshalling for us.

Luckily there is an implementation of python called PyPy which is written in python itself. We can leverage PyPy's marshalling code to dump the code objects in our case. The marshalling code is in the file *_marshal.py* which can be obtained from PyPy source. Let's save the file as *dropdump.py*.

```
<<< Code from _marshal.py should be copied here as is >>>
```

```
import marshal          # Import the built-in marshal module

infile = open('authenticate.pyc', 'rb')
infile.seek(8)
code_obj = marshal.load(infile)

outfile = open('decrypted.pyc', 'wb')
outfile.write('\x03\xf3\x0d\x0a\x00\x00\x00\x00')
dump(code_obj, outfile)  # Use PyPy's marshalling code
outfile.close()
infile.close()
```

However on running we are greeted with the following ungrateful message.

```
C:\dropbox re>embedder dropdump.py
Traceback (most recent call last):
  File "<string>", line 701, in <module>
  File "<string>", line 673, in dump
  File "<string>", line 52, in dump
  File "<string>", line 206, in dump_code
AttributeError: 'code' object has no attribute 'co_code'
C:\dropbox re>_
```

The mystery of the missing co_code

According to the python documentation “*co_code is a string representing the sequence of bytecode instructions*”. Every python code-object has an array which contains bytecode which will be executed. This array is called *co_code*. This should always be present or otherwise code-objects cannot exist. That means the dropbox python interpreter is hiding that from us. As we cannot access *co_code* from the python layer we need to delve deeper and try to access that from within the native or the assembly layer. We need the services of a debugger. We will use old & faithful Ollydbg, but before that let’s see the structure of a *PyCodeObject*.

```
typedef struct {
    PyObject_HEAD
    int co_argcount;
    int co_nlocals;
    int co_stacksize;
    int co_flags;
    PyObject *co_code; //This is the missing member
    PyObject *co_consts;
    PyObject *co_names;
    PyObject *co_varnames;
    PyObject *co_freevars;
    PyObject *co_cellvars;
    PyObject *co_filename;
    PyObject *co_name;
    int co_firstlineno;
    PyObject *co_lnotab;
    void *co_zombieframe;
    PyObject *co_weakreflist;
} PyCodeObject;
```

The first member in the structure is *PyObject_HEAD*. It is defined as

```
/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD \
    _PyObject_HEAD_EXTRA \
    Py_ssize_t ob_refcnt; \
    struct _typeobject *ob_type;
```

For release builds of python *_PyObject_HEAD_EXTRA* is empty. So essentially it contains two members of 4 bytes each. The first member *ob_refcnt* holds the number of reference counts to this object & the second *ob_type* points to a *PyTypeObject* variable representing the type of this object. For example, if it points to *PyCode_Type*, then this object is a code_object. Just by looking at the second member of a *PyObject* in a debugger, we can know the type of it. This tiny bit of information will turn very useful in our endeavour.

Thus in a standard python distribution *co_code* is located at an offset of 24 byte from the start of *PyCodeObject*. This *co_code* is a pointer to a *PyObject*. It’s in fact a pointer to a *PyStringObject* (which is also a *PyObject*) as mentioned in the documentation. The structure of a *PyStringObject* is as follows.

```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1]; //This array contains the string, its length is in PyObject_VAR_HEAD
} PyStringObject;
```

PyObject_VAR_HEAD is defined as

```
#define PyObject_VAR_HEAD      \
    PyObject_HEAD              \
    Py_ssize_t ob_size; /* Number of items in variable part */
```

Within a *PyStringObject*, *ob_sval* is located at an offset of 20 bytes from the start. We will use these offsets while working in the debugger. These offset values are obtained from a standard python distribution. It is possible that dropbox has changed the structure layout so as to hinder reversing. We will see that shortly.

Exploring the structure of code object

To find whether the dropbox has kept the structure layout intact or has modified it, we will code a small C program. It will generate a code object using the *PyCode_New* CPython function. We will run the program in Ollydbg, and inspect the returned value. We will check whether the offsets are in tandem with what we obtained earlier. If they are different, we will need to find the actual offsets.

```
#include "Python.h"
#include "marshal.h"
#include <windows.h>

void main(int argc, char *argv[])
{
    Py_NoSiteFlag = 1;
    char filename[MAX_PATH];
    GetModuleFileNameA(NULL, filename, sizeof(filename));
    char *c = strrchr(filename, '\\');
    *c = 0;
    Py_SetPythonHome(filename);
    Py_Initialize();

    PyObject *codestring = PyString_FromString("Marker String"); //This marker string
can be used to find out the position of co_code within the code object
    PyObject *tuple = PyTuple_New(0);
    PyObject *string = PyString_FromString("");

    PyCodeObject *codeObject = PyCode_New(0, 0, 0, 0, codestring,
                                           tuple, tuple, tuple, tuple, tuple,
                                           string, string, 0, string);

    Py_Finalize();
}
```


So debug the program in Ollydbg and set a breakpoint after the call to *PyCode_New*. The returned value in *eax* will be the pointer to a *PyCodeObject* which we will be inspecting for anomalies.

0040176E	push eax	python27.Py_SetPythonHome	Registers <FP>
0040176F	call dword ptr [&python27.Py_SetPythonHome]	python27.Py_SetPythonHome	EAX 0094E268
00401775	call dword ptr [&python27.Py_Initialize]	python27.Py_Initialize	ECX 00000001
0040177B	mov esi,dword ptr [&python27.PyString_FromString]	python27.PyString_FromString	EDX 00000000
00401781	push embedder.004020E8	ASCII "Marker String"	EBX 00000000
00401786	call esi	<&python27.PyString_FromString>	ESP 0012FE14
00401788	push ebx		EBP 0012FEF0
00401789	mov dword ptr [ebp-80],eax	python27.PyTuple_New	ESI 1E0B3FA0
0040178C	call dword ptr [&python27.PyTuple_New]	python27.PyTuple_New	EDI 00921038
00401792	push embedder.004020F6		EIP 004017B1
00401797	mov edi,eax	<&python27.PyString_FromString>	C 0 ES 0023
00401799	call esi		P 1 CS 001B
0040179B	push eax		A 0 SS 0023
0040179C	push ebx		Z 0 DS 0023
0040179D	push eax		S 0 FS 003B
0040179E	push eax		T 0 GS 0000
0040179F	push edi		D 0
004017A0	push edi		O 0 LastErr
004017A1	push edi		EFL 00000206
004017A2	push edi		ST0 empty -UN
004017A3	push edi		ST1 empty +UN
004017A4	push dword ptr [ebp-80]		ST2 empty +UN
004017A7	push ebx		ST3 empty 0.0
004017A8	push ebx		ST4 empty 0.0
004017A9	push ebx		ST5 empty 0.0
004017AA	push ebx		ST6 empty 1.0
004017AB	call dword ptr [&python27.PyCode_New]	python27.PyCode_New	ST7 empty 2.2
004017B1	add esp,50		FST 4000 Con
004017B4	call dword ptr [&python27.Py_Finalize]	python27.Py_Finalize	
004017BA	mov ecx,dword ptr [ebp+88]		
004017C0	pop edi		

The returned value in this case is *0x0094E268*. We will follow the value in dump.

0094E268	00000001	
0094E26C	1E34A248	offset python27.PyCode_Type
0094E270	00000000	
0094E274	00000000	
0094E278	00000000	
0094E27C	00000000	
0094E280	00921038	This should be the pointer to co_code if the structure is unmodified
0094E284	00921038	
0094E288	00921038	
0094E28C	00921038	
0094E290	00921038	
0094E294	00924328	
0094E298	00924328	
0094E29C	00000000	
0094E2A0	00953820	
0094E2A4	00924328	
0094E2A8	00000000	
0094E2AC	00000000	

The returned object is indeed a *PyCodeObject* as evident from the second member. Now we need to verify if the value at an offset of 24 is indeed a pointer to *co_code*. So we follow the value at *0x0094E280* in dump to reach here. Remember that *co_code* is actually a *PyStringObject*.

00921038	00000009	
0092103C	1E357BC0	offset python27.PyTuple_Type
00921040	00000000	
00921044	BAADF00D	

Now this is a *PyTupleObject*. We were expecting to find a *PyStringObject* here. This means dropbox has fiddled with the layout of *PyCodeObject* as we suspected earlier. So we need to find the actual offset of *co_code*. From this juncture we can take two paths. We can either follow each member of *PyCodeObject* in dump to see which is a *PyStringObject* containing our marker string or we can write a small C program to do the job. During my reversing session, I took the first path, but now in this tutorial I will demonstrate the second one and then will verify the result in the debugger. We will be modifying the previous program a little to find out the offset of *co_code*.

```

#include "Python.h"
#include "marshal.h"
#include <windows.h>

void main(int argc, char *argv[])
{
    Py_NoSiteFlag = 1;
    char filename[MAX_PATH];
    GetModuleFileNameA(NULL, filename, sizeof(filename));
    char *c = strrchr(filename, '\\');
    *c = 0;
    Py_SetPythonHome(filename);
    Py_Initialize();

    PyObject *codestring = PyString_FromString("Marker String");
    PyObject *tuple = PyTuple_New (0);
    PyObject *string = PyString_FromString("");

    PyCodeObject *codeObject = PyCode_New(0, 0, 0, 0, codestring,
                                           tuple, tuple, tuple, tuple, tuple,
                                           string, string, 0, string);

    char *ptr;

    for(ptr = (char*)codeObject; ptr < (char*)codeObject + sizeof(PyCodeObject); ptr+=4)
        if( *((PyObject**)ptr) == codeString)
            printf("co_code found at offset %d\n", (ptr - (char*)codeObject));

    Py_Finalize();
}

```

Running the program gives the following output.

```

C:\dropbox re>findoffset
co_code found at an offset of 56
C:\dropbox re>

```

To verify the results we will use the debugger. We will be following the value at an offset of 56 (i.e. 0x00953820) from the start of *PyCodeObject*.

00953820	00000002	
00953824	1E357780	offset python27.PyString_Type
00953828	0000000D	
0095382C	FFFFFFFF	

It's indeed a *PyStringObject* as it should be, and further if we change the display to hex we will see our marker string.

00953820	02 00 00 00	80 77 35 1E	0D 00 00 00	FF FF FF FFw5.....
00953830	00 00 00 00	4D 61 72 6B	65 72 20 53	74 72 69 6EMarker Strin
00953840	67 00 AD BA	0D F0 AD BA	00 00 00 00	0D F0 AD BA	g.....
00953850	0D F0 AD BA	0D F0 AD BA	0D F0 AD BA	0D F0 AD BA

So dropbox has modified the structure of *PyCodeObject*. Now *co_code* is located at an offset of 56 instead of 24.

Getting access to co_code

Okay, *co_code* is located at an offset of 56 but we cannot access it in the python layer. We need access to it so that we can marshal the code object to disk, and for this purpose we will code a C extension. The program will contain a function which when fed a *PyCodeObject* will return the *co_code*.

```
#include <Python.h>

static PyObject* getCode(PyObject* self, PyObject* args)
{
    PyObject* code = NULL;
    PyObject* co_code = NULL;

    PyArg_ParseTuple(args, "O", &code);

    _asm
    {
        mov eax, code
        mov eax, dword ptr [eax + 56] //The code object is located at an offset of 56
        mov co_code, eax
    }

    Py_XINCREF(co_code); //Increase the reference count
    return co_code;
}

static PyMethodDef extension_methods[]={
    {"getCode", getCode, METH_VARARGS, "Get Code Object"},
    {NULL, NULL, 0, NULL}};

PyMODINIT_FUNC initedropextension()
{
    //The name of the extension module as seen from python
    //This should be of same name as of the extension file.
    //In this case the file will be named as dropextension.pyd
    Py_InitModule("dropextension", extension_methods);
}
```

We need to modify *dropdump.py* so that it uses our extension for accessing *co_code*. We will only modify the function *dump_code*. Rest will remain same.

```
import dropextension # Import the extension

def dump_code(self, x):
    self._write(TYPE_CODE)
    self.w_long(x.co_argcount)
    self.w_long(x.co_nlocals)
    self.w_long(x.co_stacksize)
    self.w_long(x.co_flags)
    self.dump(dropextension.getCode(x)) # Use our extension to access co_code
    self.dump(x.co_consts)
    self.dump(x.co_names)
    self.dump(x.co_varnames)
    self.dump(x.co_freevars)
    self.dump(x.co_cellvars)
    self.dump(x.co_filename)
    self.dump(x.co_name)
    self.w_long(x.co_firstlineno)
    self.dump(x.co_lnotab)
```

Is that enough?

Lets' now try to marshal the code using the newly coded tools. This time there are no error messages.

```
C:\dropbox re>embedder dropdump.py
C:\dropbox re>
```

We can find the newly created file *decrypted.pyc*. Let's open it in a hex editor.

decrypted.pyc																	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	D3	F3	0D	0A	00	00	00	00	63	00	00	00	00	00	00	00	.ó.....c.....
0010h:	00	03	00	00	00	40	40	00	00	73	08	03	00	00	52	00@@..s....R.
0020h:	00	52	01	00	62	00	00	63	01	00	69	01	00	0F	52	00	.R..b..c..i...R.
0030h:	00	52	02	00	62	02	00	69	02	00	52	00	00	52	02	00	.R..b..i..R..R..
0040h:	62	03	00	69	03	00	52	00	00	52	02	00	62	04	00	69	b..i..R..R..b..i
0050h:	04	00	52	00	00	52	02	00	62	05	00	69	05	00	52	00	..R..R..b..i..R.
0060h:	00	52	02	00	62	06	00	69	06	00	52	00	00	52	02	00	.R..b..i..R..R..
0070h:	62	07	00	69	07	00	52	00	00	52	02	00	62	08	00	69	b..i..R..R..b..i
0080h:	08	00	52	00	00	52	02	00	62	09	00	69	09	00	52	00	..R..R..b..i..R.
0090h:	00	52	02	00	62	0A	00	69	0A	00	52	00	00	52	02	00	.R..b..i..R..R..
00A0h:	62	0B	00	69	0B	00	52	00	00	52	03	00	62	0C	00	63	b..i..R..R..b..c
00B0h:	0D	00	69	0D	00	0F	52	00	00	52	04	00	62	0E	00	63	..i...R..R..b..c
00C0h:	0F	00	69	0F	00	0F	52	00	00	52	05	00	62	10	00	63	..i...R..R..b..c

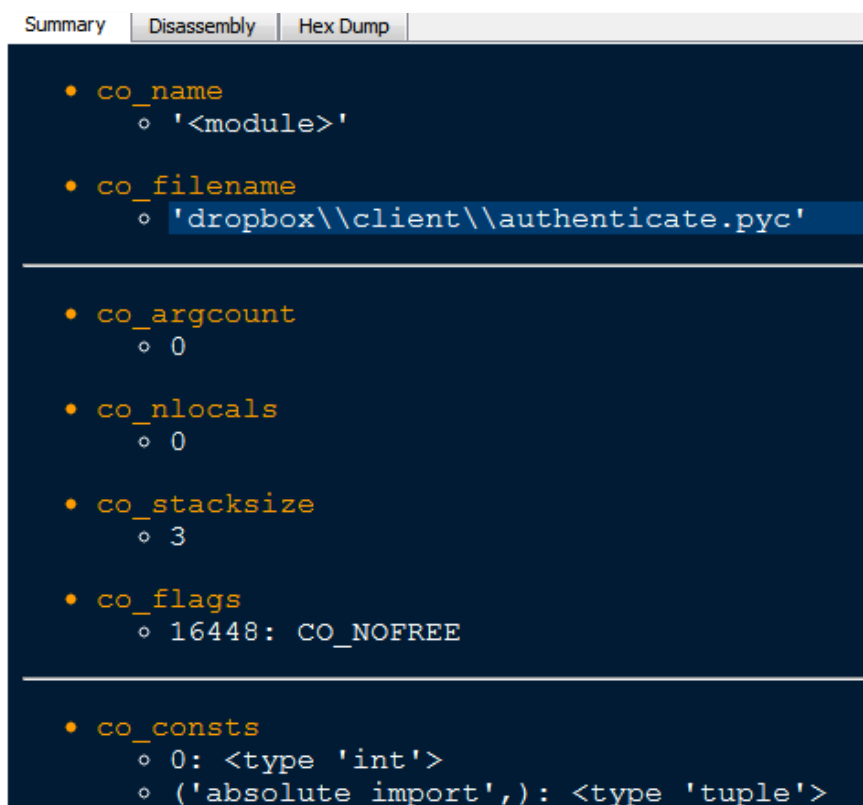
This time the file is not empty. If we scroll down a bit we can find some readable strings. This means the file has been decrypted and marshalling has succeeded. All that is left is to decompile the file.

decrypted.pyc																	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0450h:	72	65	61	64	28	03	00	00	00	73	05	00	00	00	54	52	read(....s....TR
0460h:	41	43	45	73	15	00	00	00	72	65	70	6F	72	74	5F	62	ACEs....report_b
0470h:	61	64	5F	61	73	73	75	6D	70	74	69	6F	6E	73	15	00	ad_assumptions..
0480h:	00	00	75	6E	68	61	6E	64	6C	65	64	5F	65	78	63	5F	..unhandled_exc_
0490h:	68	61	6E	64	6C	65	72	28	01	00	00	00	73	0A	00	00	handler(....s...
04A0h:	00	53	79	6E	63	45	6E	67	69	6E	65	28	01	00	00	00	.SyncEngine(....
04B0h:	73	19	00	00	00	52	65	63	65	6E	74	6C	79	43	68	61	s....RecentlyCha
04C0h:	6E	67	65	64	41	67	67	72	65	67	61	74	6F	72	28	03	ngedAggregator(.
04D0h:	00	00	00	73	16	00	00	00	6C	6F	61	64	5F	64	72	6F	...s....load_dro
04E0h:	70	62	6F	78	5F	66	69	6C	65	63	61	63	68	65	73	1A	pbox_filecaches.
04F0h:	00	00	00	6C	6F	61	64	5F	64	72	6F	70	62	6F	78	5F	...load_dropbox
0500h:	69	64	65	61	6C	5F	74	72	61	63	6B	65	72	73	16	00	ideal_trackers..
0510h:	00	00	72	65	69	6E	69	74	69	61	6C	69	7A	65	5F	64	..reinitialize_d
0520h:	61	74	61	62	61	73	65	73	28	01	00	00	00	73	18	00	atabases(....s..
0530h:	00	00	72	65	70	6F	72	74	5F	75	6E	73	65	6E	74	5F	..report_unsent

We will be using *Easy Python Decompiler*. Let's try to decompile *decrypted.pyc*.



Tough luck! Failure once again. This time it says *Invalid pyc/pyo file*. Although decompilers do fail, but in this case the file is really invalid. We will be using *pychrysanthemum*, a tool for inspecting *pyc* files to verify the results.



From the summary tab everything looks to be normal. Let's have at a look at the Disassembly tab.

```

Summary Disassembly Hex Dump
4 ('dropbox\client\authenticate.pyc')
• 0 52 LOAD_LOCALS
1 00 STOP_CODE
2 00 STOP_CODE
3 52 LOAD_LOCALS
4 01 POP_TOP
5 00 STOP_CODE
6 62 0000 DELETE_GLOBAL 0 ('__future__')
9 63 0100 DUP_TOPX 1
12 69 0100 BUILD_MAP 1
15 0f UNARY_INVERT

6 ('dropbox\client\authenticate.pyc')
16 52 LOAD_LOCALS
17 00 STOP_CODE
18 00 STOP_CODE
19 52 LOAD_LOCALS
20 02 ROT_TWO
21 00 STOP_CODE
22 62 0200 DELETE_GLOBAL 2 ('sys')
25 69 0200 BUILD_MAP 2

7 ('dropbox\client\authenticate.pyc')
28 52 LOAD_LOCALS
29 00 STOP_CODE
30 00 STOP_CODE
31 52 LOAD_LOCALS

```

It definitely shows some disassembled code but if we observe carefully, we will find that all this is junk code. We find many *STOP_CODE*. However the documentation of python says *STOP_CODE* “Indicates end-of-code to the compiler, not used by the interpreter”. So it means we should not find this opcode in a compiled python file. If we scroll down further in the disassembly, we will see other instances strongly suggesting that the code is junk.

```

447 ('dropbox\client\authenticate.pyc')
683 52 LOAD_LOCALS
684 25 <37>
685 00 STOP_CODE
686 59 BUILD_CLASS
687 23 <35>
688 00 STOP_CODE
689 5a 0100 STORE_NAME 1 ('absolute_import')
692 52 LOAD_LOCALS
693 26 <38>
694 00 STOP_CODE
695 56 YIELD_VALUE
696 00 STOP_CODE
697 00 STOP_CODE
698 70 0000 JUMP_IF_TRUE_OR_POP 0 (=> 0)
701 2a STORE_SLICE+2
702 69 5100 BUILD_MAP 81

586 ('dropbox\client\authenticate.pyc')
705 52 LOAD_LOCALS
706 27 <39>

```

In addition to the *STOP_CODE* there are several other opcodes which it could not disassemble. That means our task is not over yet. We need to convert this junk code to something comprehensible to a disassembler & a decompiler.

The use of opcode remapping

Opcode remapping is a technique in which the opcode definition of a Virtual Machine is changed. In case of python it means that the opcodes meaning are different than that of a standard python distribution. Thus if the opcode 23 initially meant *BINARY_ADD*, it may now mean *POP_TOP*. To be able to decompile the file successfully we need to obtain the new opcode mapping. Using that we can definitely decompile the *pyc* file. In the case of dropbox we are confident that it uses this trick, as all other facets of the *pyc* file are perfectly normal. Only we cannot decipher the bytecode instructions.

Now ponder for a moment. Suppose that we compile a python script in this modified python interpreter and compare it with the output generated by compiling the same code but in a standard python 2.7 interpreter there will be some differences. These differences will be due to the fact that the modified interpreter uses different set of opcodes. Rest should be same. So using this we should be able to find out which opcodes were mapped and to what new values.

Generating the file set

So for now we need to generate two sets of *pyc* files. One compiled from dropbox python and the other from standard python 2.7. We need several python script files such that by compiling it we can generate almost all opcodes used by python (python 2.7 has 118 opcodes). To ease our search we can use the *py* files provided in a standard distribution (there are more than a thousand files). That should hopefully generate majority of opcodes if not all. Infact we will see later that even after using more than a thousand files there are about 5 opcodes left. We could ignore them for the sole reason that if we cannot find the usage of those opcodes even after comparing more than a thousand files those opcodes are probably not used normally. An example of such opcode is *EXTENDED_ARG*.

We will use the same code developed earlier in *embedder.exe* & *dropextension.pyd*. For generating the first set of files, we will be linking against the standard python dll. For the second, we need to link with the dropbox python dll along with the C extension. The extension will only be needed in the second case as there is no access to *co_code*. However in both cases the code of *embedder* need not be changed.

For generating the first set of reference files, we will prepare a python script. The name of the script file will be passed as an argument to *embedder* as we have been doing earlier. The purpose of this script will be to load a *py* file from disk, compile it to generate a code object, and then marshal it back to disk using PyPy's marshalling code. We will use PyPy's marshalling code instead of the built-in for consistency.

```
<<< code from _marshal.py should be copied here as is >>>

basedir = os.getcwd()
py_files = os.path.join(basedir, 'py_files') # The source py files will be located here
out_files = os.path.join(basedir, 'org_opcodes') # The output files will go here

for f in os.listdir(py_files):
    txt = open(os.path.join(py_files, f)).read()
    of = open(os.path.join(out_files, f + '.org'), 'wb')
    cobj = compile(txt, '', 'exec') # Compile the code
    dump(cobj, of) # Marshal using PyPy's marshalling code
    of.close()
```

Running the above script using *embedder* generates the first set of reference files.

For generating the second set of reference files we will use the modified code of *_marshal.py* as used in *dropdump.py*. The rest of the code will be as follows.

```
basedir = os.getcwd()
py_files = os.path.join(basedir, 'py_files') # The source py files will be located here
out_files = os.path.join(basedir, 'drop_opcodes') # The output files will go here

for f in os.listdir(py_files):
    txt = open(os.path.join(py_files, f)).read()
    of = open(os.path.join(out_files, f + '.drop'), 'wb')
    cobj = compile(txt, '', 'exec') # Compile the code
    dump(cobj, of) # Marshal using PyPy's marshalling code
    of.close()
```

So the above two code snippets are similar with the difference being the output directory & the extension of the output file. Running the above using *dropbox python dll* and *embedder* yields the second set of reference files.

Finding the opcode mapping

We have the two set of reference files generated from the same source. The difference between these two sets of files should reveal the opcode mapping. We need to code a tool which will find the differences. For simplicity we will be coding the tool in python although we could use any other language here.

```
import os, marshal

opcodes = dict() # Dictionary to store the opcodes

def compare(ocode, ncode):
    orgCodeStr, dropCodeStr = bytearray(ocode.co_code), bytearray(ncode.co_code)

    # Make sure we are comparing strings of same length
    if len(orgCodeStr) == len(dropCodeStr):
        # Compare the code strings bytes
        for o, n in zip(orgCodeStr, dropCodeStr):
            if o != n:
                if o not in opcodes.keys():
                    opcodes[o] = n
                else:
                    if opcodes[o] != n:
                        print 'Two remapped opcodes for a single opcodes, The files are out of sync, skipping'
                        break
            else:
                print 'Code Strings not of same length, skipping...'

    # Recursive scanning for more code objects
    for oconst, nconst in zip(ocode.co_consts, ncode.co_consts):
        if hasattr(oconst, 'co_code') and hasattr(nconst, 'co_code'):
            # both should have co_code
            compare(oconst, nconst)

def main():
    org_files = os.path.join(os.getcwd(), 'org_opcodes')
    new_files = os.path.join(os.getcwd(), 'drop_opcodes')
```

```

for f in os.listdir(org_files):
    # Open the files
    of = open(os.path.join(org_files, f), 'rb')
    nf = open(os.path.join(new_files, f[0:-4] + '.drop'), 'rb')

    # unmarshal & compare opcodes
    compare(marshal.load(of), marshal.load(nf))

    of.close()
    nf.close()
print opcodes

if __name__ == '__main__':
    main()

```

Running the script gives us the following opcode map.

```

{1: 15, 2: 59, 3: 60, 4: 13, 5: 49, 10: 48, 11: 54, 12: 38, 13: 25, 15: 34,
19: 28, 20: 36, 21: 12, 22: 41, 23: 52, 24: 55, 25: 4, 26: 43, 27: 5, 28: 32,
29:30, 30: 16, 31: 17, 32: 18, 33: 19, 40: 61, 41: 62, 42: 63, 43: 64, 50: 44,
51: 45, 52: 46, 53: 47, 54: 70, 55: 6, 56: 29, 57: 8, 58: 27, 59: 3, 60: 31,
61: 69, 62: 7, 63: 22, 64: 50, 65: 21, 66: 2, 67: 57, 68: 39, 71: 9, 72: 14,
73: 33, 74: 35, 75: 11, 76: 58, 77: 24, 78: 23, 79: 10, 80: 40, 81: 37, 82: 51,
83: 66, 84: 56, 85: 65, 86: 26, 87: 1, 88: 67, 89: 42, 90: 105, 91: 104, 92: 103,
93: 91, 94: 83, 95: 94, 96: 97, 97: 115, 98: 108, 99: 114, 100: 82, 101: 89, 102: 90,
103: 117, 104: 118, 105: 88, 106: 96, 107: 111, 108: 98, 109: 99, 110: 119, 111: 120,
112: 122, 114: 123, 115: 121, 116: 80, 119: 106, 120: 84, 121: 116, 122: 85, 124: 102,
125: 92, 126: 81, 130: 101, 131: 112, 132: 86, 133: 87, 134: 95, 135: 107, 136: 109,
137: 110, 140: 133, 141: 134, 142: 135, 143: 136, 146: 141, 147: 142}

```

If we observe carefully, we will find the following opcodes are missing from the map 0, 9, 70, 113, 145. The meaning of the opcodes are 0 -> *STOP_CODE*, 9 -> *NOP*, 70 -> *PRINT_EXPR*, 113 -> *JUMP_ABSOLUTE*, 145 -> *EXTENDED_ARG*.

The opcode 113 can be generated by compiling the following snippet.

```

def foo():
    if bar1:
        if bar2:
            print ' '

```

By following the comparison method devised earlier we can see that opcode 113 is left unchanged. Now among the remaining four opcodes, *STOP_CODE* & *NOP* are never generated in compiled bytecode, so we can ignore them. *PRINT_EXPR* is generated when the interpreter is running in interactive mode so we can ignore it too. The *EXTENDED_ARG* opcode is generated whenever the argument passed to a function is too big to fit in a space of two bytes. It can be generated in cases like passing more than 65,536 parameters to a function. This is also a rare situation, so we can ignore it too.

Now we have recovered the generated the opcode map. We have found which opcodes were changed and to what new values. We need to incorporate this opcode map while marshalling the code object to disk i.e. before dumping we will scan *co_code* and change remapped opcodes to original values, so that it can be disassembled & decompiled.

Opcode unmapping

To incorporate the newly found opcode map, we will reuse the code of *dropdump.py*. The code will be modified as follows. We can name the file as *unmapper.py*.

```
import dropextension, marshal

remap = {0: 0, 113: 113, 145: 145, 20: 9, 30: 70, 15: 1, 59: 2, 60: 3, 13: 4, 49: 5, 48:
10, 54: 11, 38: 12, 25: 13, 34: 15, 28: 19, 36: 20, 12: 21, 41: 22, 52: 23, 55: 24, 4:
25, 43: 26, 5: 27, 32: 28, 16: 30, 17: 31, 18: 32, 19: 33, 61: 40, 62: 41, 63: 42, 64:
43, 44: 50, 45: 51, 46: 52, 47: 53, 70: 54, 6: 55, 29: 56, 8: 57, 27: 58, 3: 59, 31: 60,
69: 61, 7: 62, 22: 63, 50: 64, 21: 65, 2: 66, 57: 67, 39: 68, 9: 71, 14: 72, 33: 73,
35: 74, 11: 75, 58: 76, 24: 77, 23: 78, 10: 79, 40: 80, 37: 81, 51: 82, 66: 83, 56: 84,
65: 85, 26: 86, 1: 87, 67: 88, 42: 89, 105: 90, 104: 91, 103: 92, 91: 93, 83: 94, 94:
95, 97: 96, 115: 97, 108: 98, 114: 99, 82: 100, 89: 101, 90: 102, 117: 103, 118: 104,
88: 105, 96: 106, 111: 107, 98: 108, 99: 109, 119: 110, 120: 111, 122: 112, 123: 114,
121: 115, 80: 116, 106: 119, 84: 120, 116: 121, 85: 122, 102: 124, 92: 125, 81: 126,
101: 130, 112: 131, 86: 132, 87: 133, 95: 134, 107: 135, 109: 136, 110: 137, 133: 140,
134: 141, 135: 142, 136: 143, 141: 146, 142: 147}

def dump_code(self, x):
    self._write(TYPE_CODE)
    self.w_long(x.co_argcount)
    self.w_long(x.co_nlocals)
    self.w_long(x.co_stacksize)
    self.w_long(x.co_flags)
    code = bytearray(dropextension.getCode(x))
    c = 0
    while c < len(code):
        n = remap[code[c]] # Using the opcode map
        code[c] = n
        c+=1 if n < 90 else 3 # Opcodes greater than 89 takes 2 byte parameter

    self.dump(str(code))
    self.dump(x.co_consts)
    self.dump(x.co_names)
    self.dump(x.co_varnames)
    self.dump(x.co_freevars)
    self.dump(x.co_cellvars)
    self.dump(x.co_filename)
    self.dump(x.co_name)
    self.w_long(x.co_firstlineno)
    self.dump(x.co_lnotab)

inf = open('authenticate.pyc', 'rb') # Load the file we wish to decompile
inf.seek(8) # Skip 8 byte header
code = marshal.load(inf) # Unmarshal using built in module
inf.close()

outf = open('decrypted.pyc', 'wb')
outf.write('\x03\xf3\x0d\xa0\x00\x00\x00\x00')
dump(code, outf)
outf.close()
```

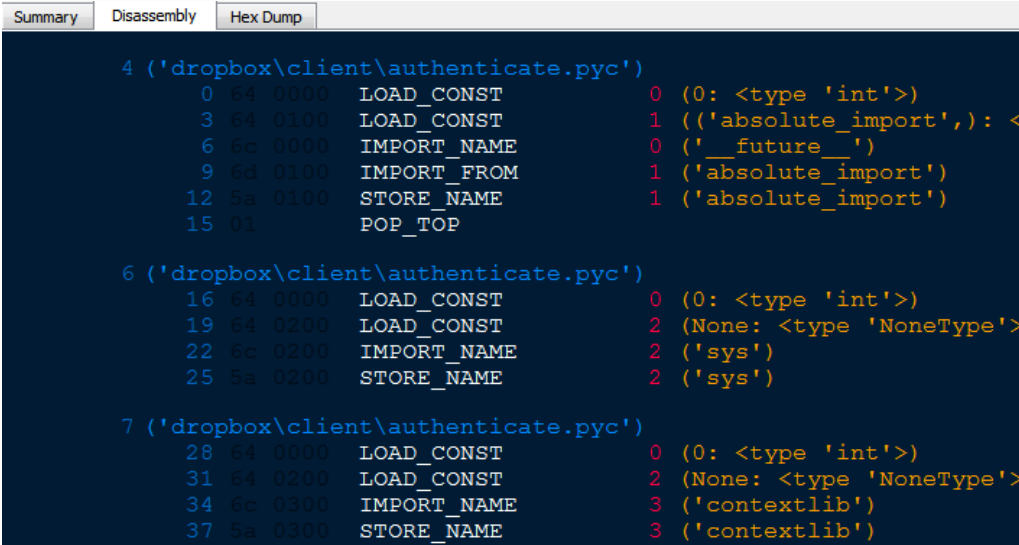
Note that each key value pair in the remap dictionary is reversed. This is due to the fact that now we want to change the new opcode back to the original one. Also note that we have chosen arbitrary values for opcodes 0, 9, 70, 145. We also need to make sure that these chosen arbitrary values do not clash with an existing value. We need to run the script in a similar way using *embedder*. After that it should hopefully be curtains down.

The final results

Running we get the following results. No errors. No messages

```
C:\dropbox re>embedder unmapper.py
C:\dropbox re>
```

A file *decrypted.pyc* is also created. Now time to open in *pychrysanthemum*, before decompiling.



The screenshot shows the Disassembly window of Pychrysanthemum. It displays the disassembly of a Python bytecode file named 'dropbox\client\authenticate.pyc'. The window has three tabs: Summary, Disassembly, and Hex Dump. The Disassembly tab is active, showing a list of instructions with their addresses, opcodes, and disassembled names. The instructions are grouped into three blocks, each starting with a comment indicating the file name.

Address	Opcode	Disassembled Name
4 ('dropbox\client\authenticate.pyc')		
0	64 0000	LOAD_CONST 0 (0: <type 'int'>)
3	64 0100	LOAD_CONST 1 (('absolute_import',): <
6	6c 0000	IMPORT_NAME 0 ('__future__')
9	6d 0100	IMPORT_FROM 1 ('absolute_import')
12	5a 0100	STORE_NAME 1 ('absolute_import')
15	01	POP_TOP
6 ('dropbox\client\authenticate.pyc')		
16	64 0000	LOAD_CONST 0 (0: <type 'int'>)
19	64 0200	LOAD_CONST 2 (None: <type 'NoneType'>)
22	6c 0200	IMPORT_NAME 2 ('sys')
25	5a 0200	STORE_NAME 2 ('sys')
7 ('dropbox\client\authenticate.pyc')		
28	64 0000	LOAD_CONST 0 (0: <type 'int'>)
31	64 0200	LOAD_CONST 2 (None: <type 'NoneType'>)
34	6c 0300	IMPORT_NAME 3 ('contextlib')
37	5a 0300	STORE_NAME 3 ('contextlib')

This time there are no *STOP_CODE* or partially disassembled opcodes. The coast looks clear. We can safely proceed to decompiling. Let's feed the file to *Easy Python Decompiler*.



Decompiling completed without errors. Now time to check out the code and bask in the light of glory and success.

```

def finish_dropbox_boot(self, ret, freshly_linked, wiz_ret, dropbox_folder):
    self.dropbox_app.is_freshly_linked = freshly_linked
    if self.dropbox_app.mbox.is_secondary:
        try:
            self.dropbox_app.mbox.complete_link(self.dropbox_app.config.get('email'),
ret.get('userdisplayname'), ret.get('uid'), self.dropbox_app.mbox.dual_link)
        except AttributeError:
            self.dropbox_app.mbox.callbacks.other_client_exiting()

    if freshly_linked:
        clobber_symlink_at(self.dropbox_app.sync_engine.fs, dropbox_folder)
        self.dropbox_app.safe_makedirs(dropbox_folder, 448, False)
        TRACE('Freshly linked!')
        try:
            if arch.constants.platform == 'win':
                TRACE('Trying to create a shortcut on the Desktop.')
                _folder_name = self.dropbox_app.get_dropbox_folder_name()
                arch.util.add_shortcut_to_desktop(_folder_name,
self.dropbox_app.mbox.is_secondary)
                if self.dropbox_app.mbox.is_primary and _folder_name !=
arch.constants.default_dropbox_folder_name:
                    TRACE('Attempting to remove shortcut named Dropbox since our folder
name is %r', _folder_name)

                arch.util.remove_shortcut_from_desktop(arch.constants.default_dropbox_folder_name)
            except Exception:
                unhandled_exc_handler()

```

The code above is a snippet of the decompiled code. Now we have access to the full source code of dropbox. We can reverse engineer it, look for vulnerabilities or may even code an open source dropbox client. The possibilities are many. With this we come to the end of this tutorial. Hope you liked it and thanks for reading. Some additional information will be presented in the addendum.

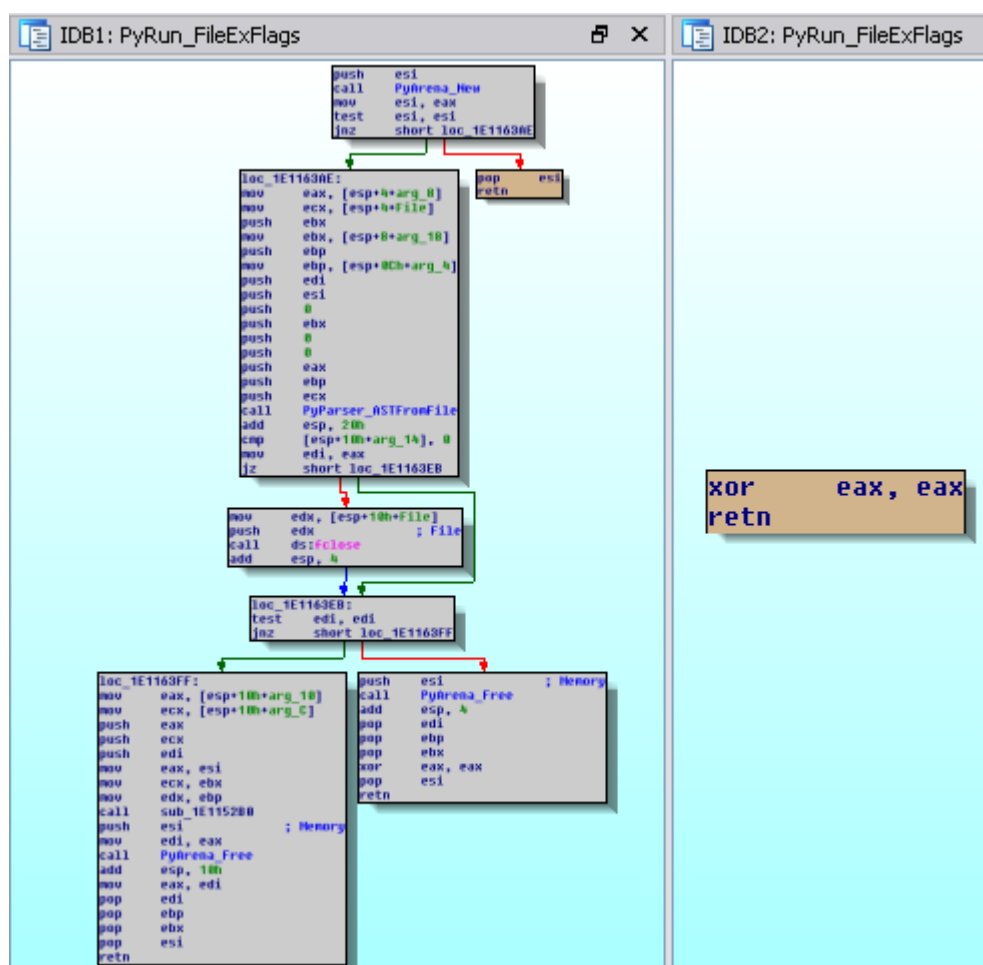
This is *extremecoders* signing off, Ciao!

Addendum

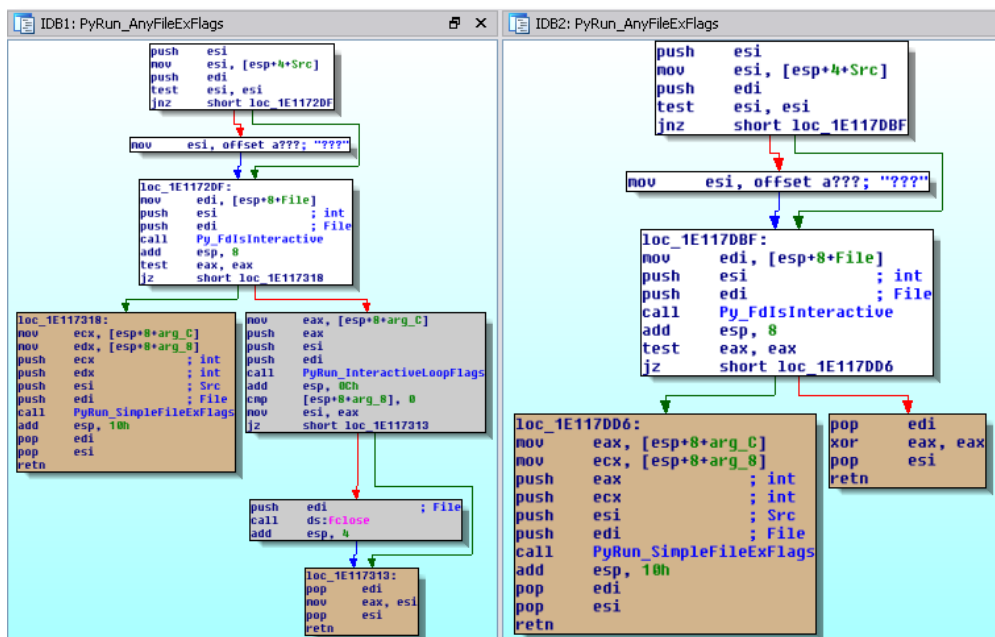
This supplement is provided to discuss some other features of the protection. We will see what steps can be taken to further increase the protection.

Exploring the differences

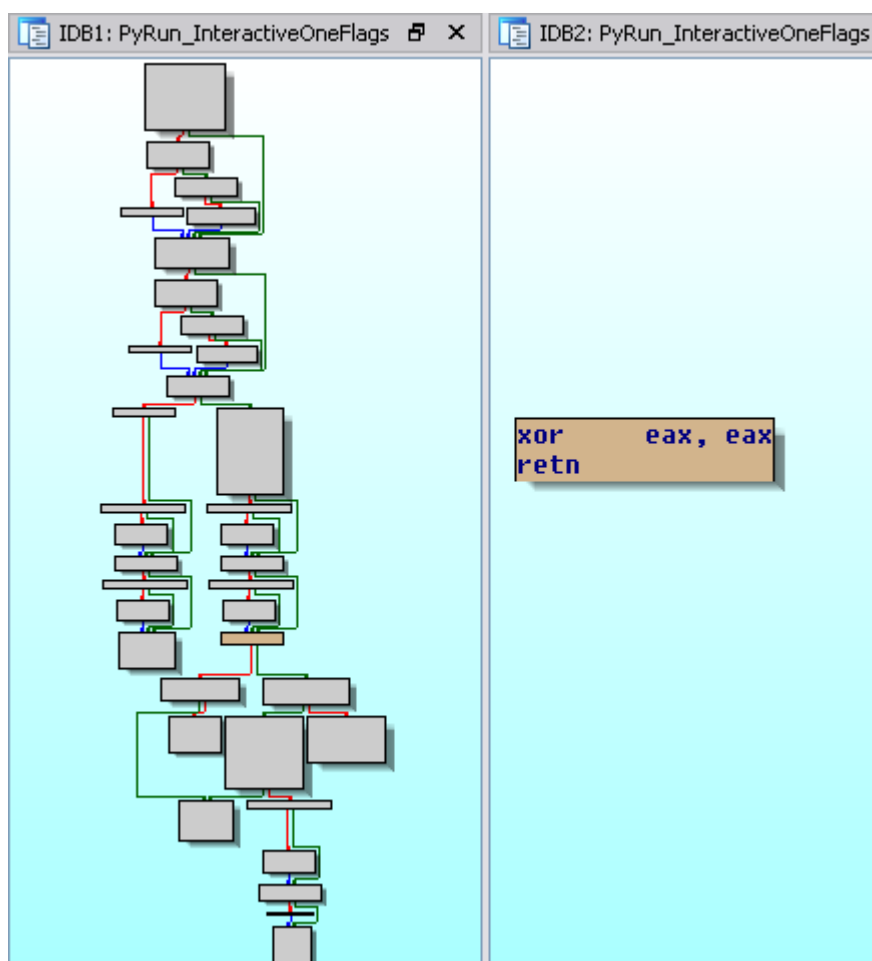
The defacto tool for binary comparison is *BinDiff* from *Zynamics* (now owned by *Google*), but due to its price tag we will be using freely available tools. We will see what other changes have dropbox incorporated to the standard python interpreter. *Patchdiff2* is a great free alternative. It requires *IDA* to run.



The flow graph on the left is from the standard python and the one on the right is from the dropbox python dll. Here we are comparing *PyRun_FileExFlags*. The function is used in CPython to execute a script associated with a file. In case of dropbox it has been patched to do no nothing. We got around this limitation by using *PyRun_SimpleString* which was unpatched. However we had to read the contents of the file ourselves and pass the source code as a string to the function. Similarly other functions such as *PyRun_SimpleFileExFlags*, *PyRun_AnyFileExFlags* have also been patched.



In case of *PyRun_AnyFileExFlags* we see that a block on the right is missing. Dropbox has removed the call to *PyRun_InteractiveLoopFlags*. This function is used to read and execute statements from a file associated with an interactive device until EOF is reached. This is the function executed when we run python in a console window or terminal. The next image shows that *PyRun_InteractiveOneFlags* has also been patched to do nothing. By patching it, we cannot start an interactive python console window (unless we take other drastic measures like implementing our own function).



Refining the protection

Here we will discuss what further could have been done to increase the protection. Firstly, dropbox uses *py2exe* on windows. The source code of *py2exe* is available. The license of *py2exe* allows modification. Dropbox Inc. could have modified the source to develop a custom version, albeit with some more protections like debugger checks, encryption etc. However the disadvantage is the protection needs to be rewritten for other platforms like Linux, MacOSX etc. as *py2exe* is only for windows.

Coming into the CPython part, we used *PyRun_SimpleString* to inject our code. This could have been patched too as it is not needed. This is like a hole in the armour. We used the built-in marshal module to load the encrypted code objects. The code objects were decrypted, immediately after loading. This could be avoided. By modifying *PyEval_EvalFrameEx*, we can make it such that *PyCodeObject* would be decrypted only when it was needed to execute. We can add a new *co_flag* to indicate which code objects were already decrypted in a previous run. By checking the flag we will only decrypt code_objects which do not have the flag set. Another good refinement to the encryption logic would be to decrypt on execution and re-encrypt it back after execution. By using the second logic we will never reach a state in which all code objects are decrypted, preventing a memory dump.

For generating the file set needed for opcode remapping, we used the built-in function *compile*, to compile source code to bytecode. This was also not needed as dropbox already uses compiled *pyc* files in the zip file. Further, by patching *compile*, would result in an exception if we tried to inject some code as python internally always compiles plain text source code to bytecode before execution.

Lastly there could have been improvements to opcode mapping. We could have used multiple opcode maps. That is we create a new *co_flag*. Suppose the flag is named *OPCODE_MAP_FLAG*. If the flag was set to 1 opcode 66 may mean *INPLACE_DIVIDE*, if the flag was set to 2, the same opcode may mean *POP_BLOCK*. The flag would be checked in *PyEval_EvalFrameEx* before execution to know which set of opcodes this currently executing code object uses. This combined with the encryption protection will definitely make dropbox much tougher to crack. The only drawback of increasing the protection is that it may result in degradation in run time performance but optimization is always possible.

Links & References

- **Dropbox** :: <https://www.dropbox.com/>
- **Python** :: <https://www.python.org/>
- **Py2exe** :: <http://www.py2exe.org/>
- **Pe Explorer** :: <http://www.heaventools.com/overview.htm>
- **Exeinfo PE** :: <http://exeinfo.atwebpages.com/>
- **010 Editor** :: <http://www.sweetscape.com/010editor/>
- **PyPy** :: <http://pypy.org/>
- **Ollydbg** :: <http://www.ollydbg.de/version2.html>
- **Easy Python Decompiler** :: <http://sourceforge.net/projects/easypythondecompiler/>
- **Py2Exe Dumper** :: <http://sourceforge.net/projects/py2exedumper/>
- **PyChrysanthemum** :: <https://github.com/monkeycz/pychrysanthemum>
- **Patchdiff2** :: <https://code.google.com/p/patchdiff2/>
- **Security analysis of dropbox** :: <https://www.usenix.org/conference/woot13/workshop-program/presentation/kholia>
- **Dropbox reversing tutorial** :: <http://progdupeu.pl/tutoriels/280/dropbox-a-des-fuites/>