# Reversing a frozen python executable

Extreme Coders

extremecoders@mail.com

## What's inside

Python is a widely used programming language due to its simplicity and ease of development. It presents an exciting prospect to software developers. From the reverser's point of view, reversing such an application presents different challenges which this document is going to explore.

# Reversing a frozen python executable

## Abstract

Python is a general purpose, open source computer programming language. Among other things; Python sports a remarkably simple, readable and maintainable syntax that makes software development simplied. It's main asset is that it can be used to program complex systems without a necessarily complex codebase. It's this feature that makes Python among the top five programming languages used in the world today. Python is deployed in a variety of products. It's current userbase includes the Internet giant *Google, YouTube, Industrial Light & Magic, NASA, BitTorrent, Skype, Dropbox* etc.

There is a growing tendency of software developers to program their software in Python as for the reasons previously described. However Python is a scripted language unlike compiled languages like C or C++. This means that the code is interpreted every time it is run. This presents a major bottleneck in software deployment. If it is necessary to ship the source code with every distribution, then copyrighted software cannot exist. Everything is as good as Open-Source, free for every one to modify and use.

In this context, there has already been some work to protect the software developed in python. Most of these solutions typically take the python source code, compile it to a *.pyc* or *.pyo* file, and then embed these compiled files into a native executable for the target platform. The python runtime along with the necessary libraries is also embeded inside the executable. Whenever the executable is run, a stub (embeded inside) starts the python runtime which in turns begins executing the main program.

For us, the reversers, reverse engineering such an application is different from reversing a compiled application written in C/C++ ,Delphi, Visual Basic or even Assembly language. Since python is an interpreted language, the code runs in a Virtual Machine. If we do not have access to the source code then it is very difficult to reverse such an application from within the VM level. So the primary step in such a reversing endeavour is reconstructing the source code from the executable. Once this step is over, the remaining becomes a trivial task.

# Our Toolset...

- Out target Athtek Digiband from *http://www.athtek.com/digiband.html*
- Python version 2.6 and 2.7 from *http://www.python.org/download/*
- PE analyzers such as peid, exeinfope, Detect It Easy, RDG packer detector from *http://tuts4you.com/download.php?list.37*
- Ollydbg from *http://www.ollydbg.de/version2.html*
- PyInstatller from *http://www.pyinstaller.org/*
- uncompyle2 from *https://github.com/Mysterie/uncompyle2*
- Any Hex Editor like HxD from *http://mh-nexus.de/en/hxd/*

Athtek Digiband is our target which we would study.

Python is required as the reversing session requires us to run scripts which will run only python is installed. Two different versions are required as code intended for one version may not run correctly one the other.

PE Analyzers as usual are plenty in number. This document depicts the use of some but feel free to use another.

Ollydbg - the de-facto tool for reversing on the win32 platform.

The use of pyinstaller and uncompyle2 will become clear as we progress. For now we can download a copy of these two tools.

Lastly any hex editor would do. Here the screenshots are from the popular free HexHeditor HxD.

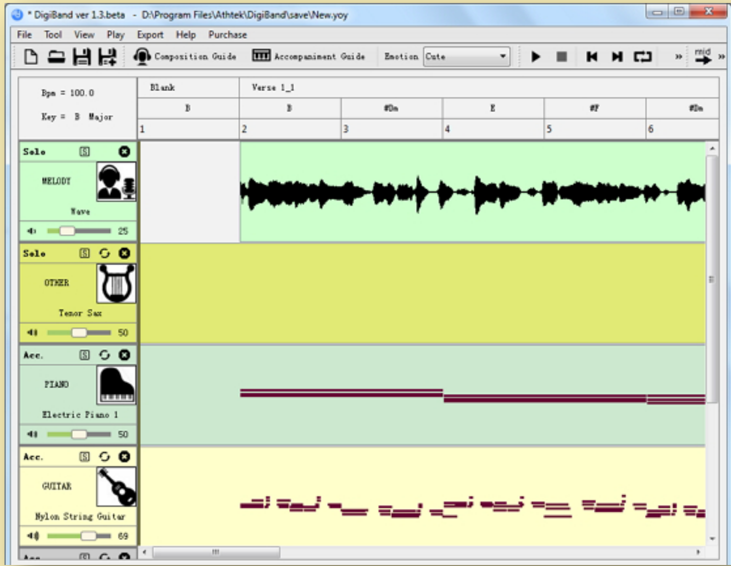
Further, although a pre existing knowledge of programming in python is not necessary but possessing it would definitely help.

# Target

The target intended for our research is Athtek Digiband. It is a software which can automatically generate music. This is what is said about the software on the website

*"AthTek DigiBand is a piece of automatic music composition software for Windows. It can automatically compose music with flexible instruments and melodies. It can also improvise an accompaniment to imported audio files, live computer keyboard playing or humming. Rich instruments and music styles are integrated to this brilliant music software. With AthTek DigiBand, you will enjoy the fun of having a versatile music group on the computer.*

*For people who are not musical, the process of composing and playing music seems almost magical. AthTek DigiBand is a program that aims to bring the joys of musical creation to the tone-deaf masses. With this easy-to-use music software, you can just follow the instructions to compose music or improvise accompaniments. You can even convert your created music into midi notation in AthTek DigiBand."*




Restrictions

| Function | Trial | Pro |
|---|---|---|
| Number of instrument tracks | 6 | 16 |
| Number of preset structure types | 3 | 8 |
| Number of genres | 1 | All |
| Open project | × | √ |
| Update band data | × | √ |
| Export midi | × | √ |
| Export wave | × | √ |
| Customize structure | × | √ |

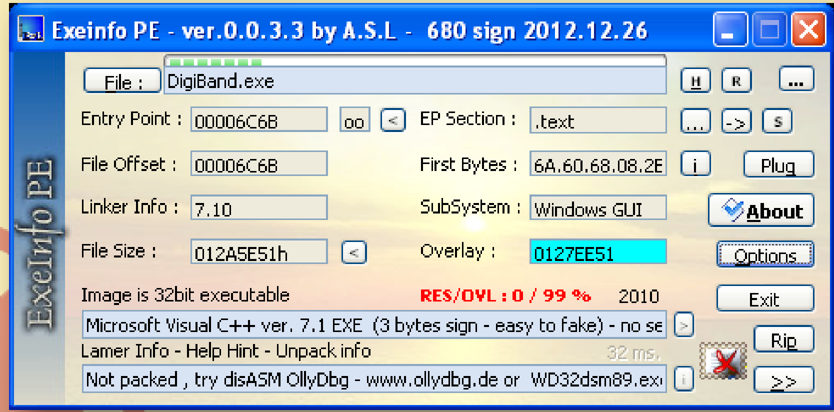
Register

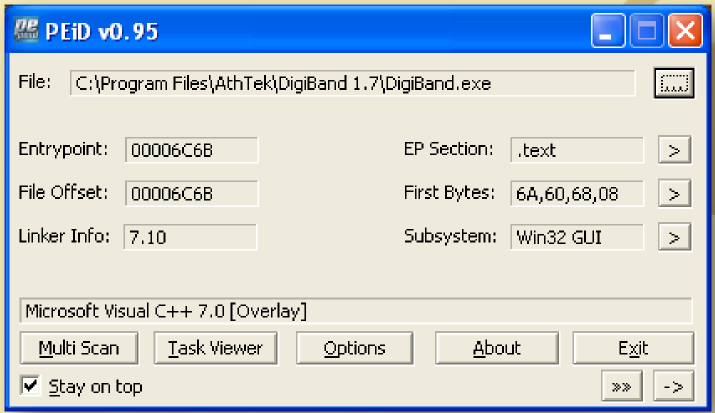
The latest version available at the time of writing is version 1.7. We will be reversing this application to sudy the method to deal with such applications. We will extract the source code from this application but we will NOT bypass the registration for reasons evident. In fact by following this tutorial we can even recompile the application.

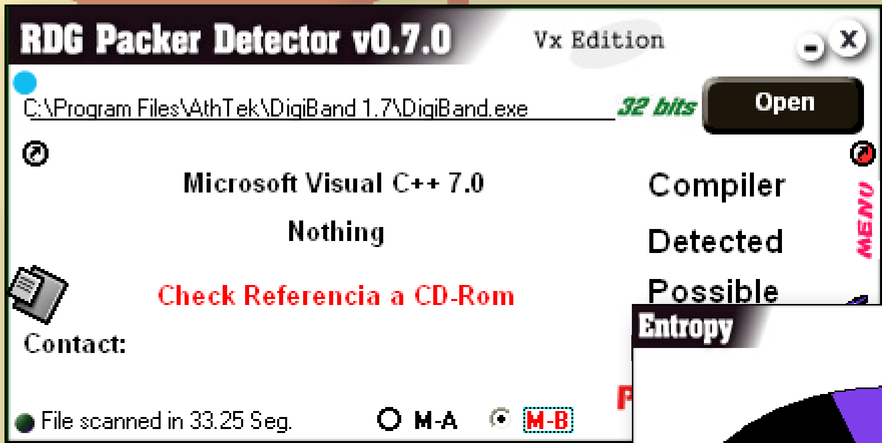
The trial version has many limitations as per the graphic. It is infact possible to remove this limitations but as said before, that is not the purpose of this document. Moreover doing so would harm the developers.

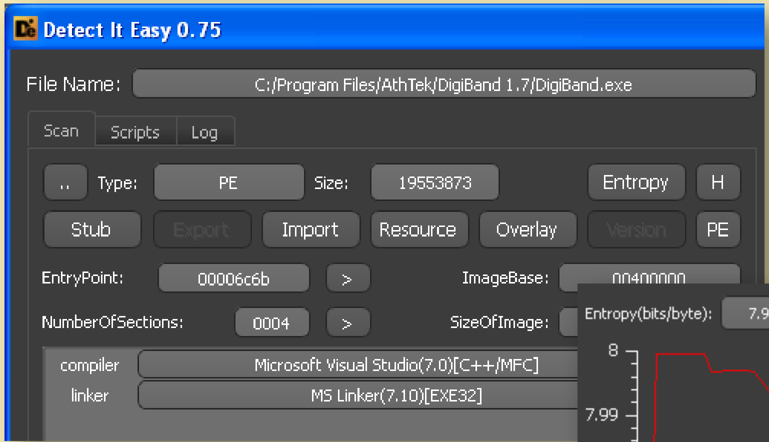
# Studying the binary

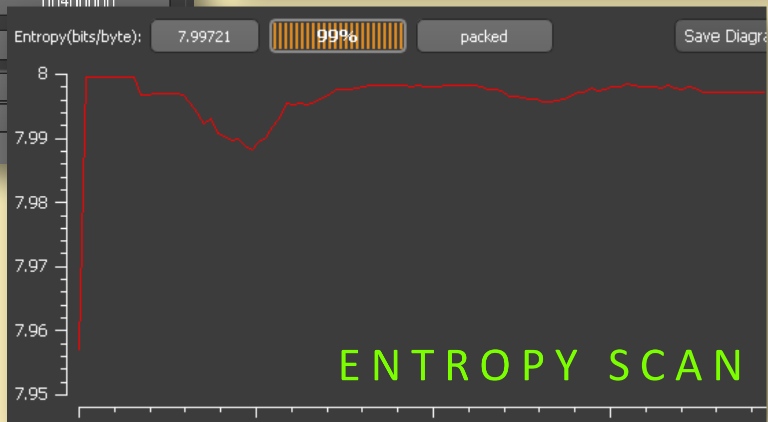
The first step in reversing as usual is a detailed study of the binary. So let's load the binary into some of analyzers and see the results.

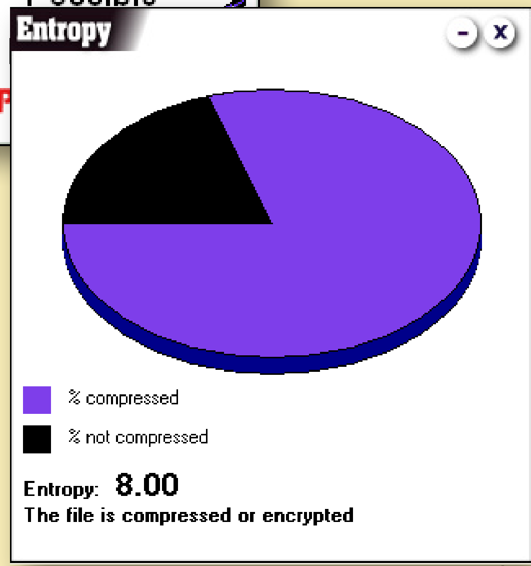



**Results :**

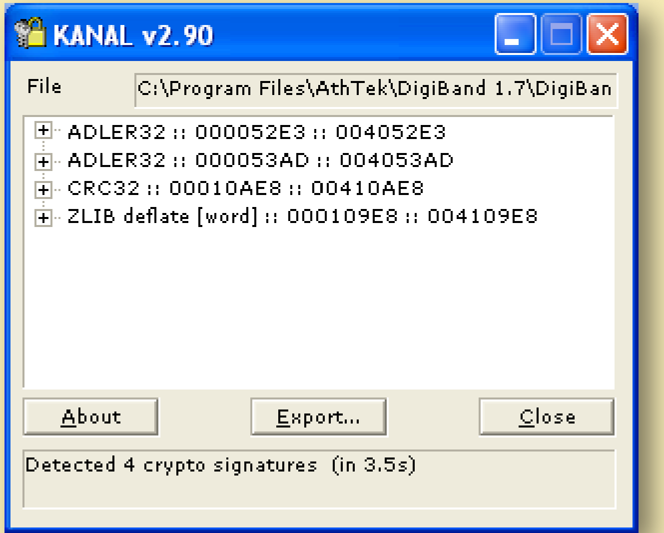
Compiler: Visual C++

Entropy scan: File is packed or encrypted

# Analyzing the results

The results we got so far is not encouraging. All the tools used for analysis reported Microsoft Visual C++ 7.0 as the compiler. No packer or protector or detected. However entropy scanning reports that the file is packed or encrypted. So either the file is protected by some advanced protectors which the world has not yet heard of (like a private protector specifically developed by the company which is improbable) or it is too simple that it evades common methods of detection.

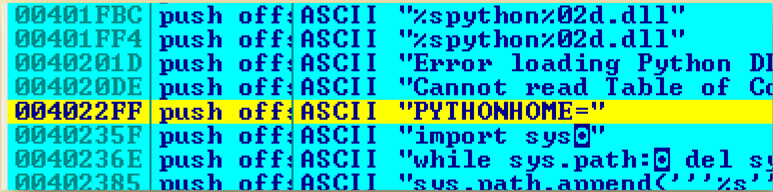
In such a confusing situation a useful tool comes to our rescue. It is the Cryptographic Analyzer plugin aka Kanal for Peid. So lets load it in the tool and see what it has to say.



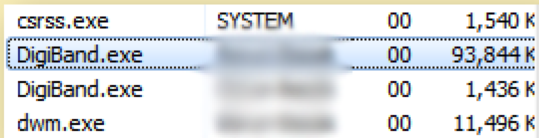

Kanal reports 4 crypto signatures. Among them the one that is specifically interesting is the ZLIB signature. ZLIB is a deflate compression algorithm commonly used to compress random length data blocks and it is efficient in doing so.

So it looks like we have found our packer. The file in all probability contains data compressed by ZLIB.

Debugging the application in Ollydbg reveals other useful of information. Doing a string search reveals the presence of the *python* string implying the application was developed in Python. But the file we are running is not a *py* file, it is an *exe* file. So the python script was actually converted to an exe file which we ran.




Another feature is the parent process starts up a child process which in turn loads the main GUI. The parent process meanwhile holds a *mutex* waiting for the child to terminate, and once the child terminates, the parent also terminate. This characteristics can be easily detected if we use the latest version of Ollydbg (v 2.01) and in Options turn on *Debug Child Process.*




With these features in mind let us search popular *py to exe* tools on the Internet. The results found are *py2exe, pyinstaller, cx-freeze, bbfreeze* etc.

It is now time to read the documentation on these tools, to find out which of the tool has the similar set of features as we found out. This is a monotonous task which all reversers try to avoid, but in our case we have no other option. So lets do the unavoidable.

# Finding the python to exe tool

At this point (after reading the necessary documentations) we can zero in on *pyinstaller* as the tool used to generate the exe. The relevant documentation (which strongly supports our assumption) copied from the *pyinstaller* manual is presented below :

## One Pass Execution

In a single directory deployment (`--onedir`, which is the default), all of the binaries are already in the file system. In that case, the embedding app:

- opens the archive
- starts Python (on Windows, this is done with dynamic loading so one embedding app binary can be used with any Python version)
- imports all the modules which are at the top level of the archive (basically, bootstraps the import hooks)
- mounts the `ZlibArchive(s)` in the outer archive
- runs all the scripts which are at the top level of the archive
- finalizes Python

## Two Pass Execution

There are a couple situations which require two passes:

- a `--onefile` deployment (on Windows, the files can't be cleaned up afterwards because Python does not call `FreeLibrary`; on other platforms extracted in the same process that uses them)
- `LD_LIBRARY_PATH` needs to be set to find the binaries (not extension modules, but modules the extensions are linked to).

The first pass:

- opens the archive
- extracts all the binaries in the archive (in PyInstaller 2.0, this is always to a temporary directory).
- sets a magic environment variable
- sets `LD_LIBRARY_PATH` (non-Windows)
- executes itself as a child process (letting the child use his stdin, stdout and stderr)
- waits for the child to exit (on *nix, the child actually replaces the parent)
- cleans up the extracted binaries (so on *nix, this is done by the child)

# Unpacking the executable

As per the documentation it is mentioned that the executable contains an embedded ZLIB compressed archive. This ZLIB archive contains the python code which we are after. Further it is mentioned, that execution is done in two steps, the first step decompresses the archive to a temporary directory; and in the second this decompressed code is executed by a child process. When the child terminates the parent cleans up the extracted files in the temporary directory. So this definition nicely fits in the behaviour we previously obtained. However one thing we are still not sure is whether the decompressed ZLIB archive contains the actual python code in human readable form.

So we have to find a way to grab the ZLIB archive and decompress it. We are in luck as *pyinstaller* itself ships with a tool (actually a python script) which can be used to inspect the contents of an executable produced by it. The script is *ArchiveViewer.py* and can be found in the *utils* directory within the *pyinstaller* distribution. So lets run the script on the executable and await the results.

```
python utils/ArchiveViewer.py <archivefile>
```

ArchiveViewer lets you examine the contents of any archive build with *PyInstalle*
executable (PYZ, PKG or exe). Invoke it with the target as the first arg (It has b
as a Send-To so it shows on the context menu in Explorer). The archive can be
using these commands:

**O <nm>**
Open the embedded archive <nm> (will prompt if omitted).

**U**
Go up one level (go back to viewing the embedding archive).

**X <nm>**
Extract nm (will prompt if omitted). Prompts for output filename. If nor
extracted to stdout.

**Q**
Quit.

USAGE

In the script output we can find references to files such as *python26.dll, PyQt.pyd* etc. This implies python version 2.6 was used with Qt being the User Interface frameworks.

Another thing worth investigating is the *outPYZ1.pyz* file. Looking in the documentation reveals that the *pyz* file is a ZLIB compressed archive which contains the compiled version (i.e. a *.pyc* or *.pyo*) of the python source code (*.py* file).

Moreover the provided script *ArchiveViewer.py* can be used to view and extract such archive. The usage of the script is shown below and is also documented in the manual.

C:\WINDOWS\system32\cmd.exe - python utils\ArchiveViewer.py DigiBand.e

```
C:\pyinstaller-2.0>python utils\ArchiveViewer.py DigiBand.exe
 pos, length, uncompressed, iscompressed, type, name
[(0, 3441258, 3441258, 0, 'z', 'outPYZ1.pyz'),
 (3441258, 8234, 20581, 1, 'm', 'iu'),
 (3449492, 146, 194, 1, 'm', 'struct'),
 (3449638, 5926, 14452, 1, 'm', 'archive'),
 (3455564, 1157, 2272, 1, 's', '_mountzlib'),
 (3456721, 81, 76, 1, 's', 'useUnicode'),
 (3456802, 382, 595, 1, 's', 'pyi_rth_qt4plugins'),
 (3457184, 1021, 2084, 1, 's', 'versioneddll'),
 (3458205, 857, 1710, 1, 's', 'win32comgenpy'),
 (3459062, 1738, 2916, 1, 's', 'main'),
 (3460800, 39799, 213504, 1, 'b', 'pygame.surface.pyd'),
 (3500599, 4806, 9728, 1, 'b', 'pygame._numericsndarray.pyd'),
 (3505405, 11446, 25088, 1, 'b', 'win32wnet.pyd'),
 (3516851, 22074, 43008, 1, 'b', 'pygame.transform.pyd'),
 (3538925, 130122, 354304, 1, 'b', 'pythoncom26.dll'),
 (3669047, 22657, 52736, 1, 'b', 'pygame.pypm.pyd'),
 (3691704, 10632, 24576, 1, 'b', 'pygame.rect.pyd'),
 (3702336, 4566, 10240, 1, 'b', 'pygame.key.pyd'),
 (3706902, 666750, 2382071, 1, 'b', 'numpy.linalg.lapack_lite.
 (4373652, 39509, 98816, 1, 'b', 'win32api.pyd'),
 (4413161, 19230, 40960, 1, 'b', '_socket.pyd'),
 (4432391, 4996, 11264, 1, 'b', 'pygame.rwobject.pyd'),
```

# Peering inside the *PYZ*

Extracting the *pyz* file and then running *ArchiveViewer.py* on the file results in

```
C:\WINDOWS\system32\cmd.exe
C:\pyinstaller-2.0>python utils\ArchiveViewer.py outPYZ1.py
Warning: pyz is from a different Python version
 Name: (ispkg, pos, len)
{'AUDIO': (False, 1993277L, 1396),
 'BaseHTTPServer': (False, 1220005L, 8468),
 'ChordSelectGUI': (False, 2430403L, 3361),
 'ConfigParser': (False, 705233L, 8129),
 'FixTk': (False, 1544298L, 565),
 'HomePanel': (False, 1364899L, 4824),
 'HummingGUI': (False, 363239L, 6757),
 'ImportMidiGUI': (False, 1573997L, 1912),
 'ImportVsqxGUI': (False, 2883614L, 1148),
 'KeyKeyboardWidget': (False, 1704485L, 2136),
 'MainWindow': (False, 208896L, 3917),
 'MidiInputGUI': (False, 566637L, 7786),
 'MusicInfoDialog': (False, 2949681L, 1649),
 'PyQt4': (True, 889927L, 117),
 'Queue': (False, 3047952L, 3202),
 'SocketServer': (False, 15088L, 7383),
 'StringIO': (False, 1831243L, 4584),
 'TEXTS': (False, 2291429L, 5355),
 'TEXTSCN': (False, 2187650L, 17679),
 'TEXTSEN': (False, 718479L, 15923),
 'TEXTSJP': (False, 22471L, 14739),
```

Now things looks promising, it seems the files which were inside the *pyz* conatins the program code. However this should not be human readble, since as per the docs the *pyz* file contains compiled python code(i.e. *.pyc* or *.pyo*). We have to find a way to decompile these files back to original python source code (*.py* file).

So lets extract any such file within the archive such as the *register* file present at an offset of 1949910L inside the archive and see what's inside.

So lets load the file *register* in a Hex-Editor to see its contents.

```
register
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  63 00 00 00 00 00 00 00 00 36 00 00 00 40 00 00
00000010  00 73 58 0B 00 00 64 00 00 5A 00 00 64 01 00 64
00000020  02 00 6B 01 00 6C 02 00 5A 02 00 6C 03 00 5A 03
00000030  00 6C 04 00 5A 04 00 6C 05 00 5A 05 00 6C 06 00
00000040  5A 06 00 6C 07 00 5A 07 00 01 64 01 00 64 03 00
00000050  6B 08 00 6C 09 00 5A 09 00 6C 0A 00 5A 0A 00 6C
00000060  0B 00 5A 0B 00 6C 0C 00 5A 0C 00 6C 0D 00 5A 0D
00000070  00 6C 0E 00 5A 0E 00 6C 0F 00 5A 0F 00 01 64 01
00000080  00 64 04 00 6B 10 00 6C 11 00 5A 11 00 01 64 01
00000090  00 64 05 00 6B 12 00 6C 13 00 5A 13 00 01 64 01
000000A0  00 64 06 00 6B 14 00 5A 14 00 64 01 00 64 06 00
000000B0  6B 15 00 5A 15 00 64 01 00 64 06 00 6B 16 00 5A
```

The file starts with 0x63. Now this file must be a compiled python file. However, a compiled python file (*.pyc*) file starts with a diferrent magic header. If we open a .pyc file from python version 2.6 we will see that the header is of the following format.

*struct header*
*{*
*DWORD magic;*
*DWORD timestamp;*
*};*

For version 2.6 the magic is hardcoded to be *D1 F2 0D 0A*.
The timestamp is the time on which this *pyc* file was created. The python compiler uses this value to check if the *pyc* is older than the corresponding *py* source and accordingly recompile.

# Decompiling the *PYC*

So in order to make python recognize the *pyc* file we have to insert the header bytes in the file before we can proceed. So load the file in a hex editor and append the following bytes (*D1 F2 0D 0A 00 00 00 00*) at the beginning of the file. Here the first 4 bytes as already said is the hardcoded magic value, and the last 4 is the timestamp which can be anything. After insertion we save and re-name the file as a *.pyc* file.

```
register

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  D1 F2 0D 0A 00 00 00 00 63 00 00 00 00 00 00 00
00000010  00 36 00 00 00 40 00 00 00 73 58 0B 00 00 64 00
00000020  00 5A 00 00 64 01 00 64 02 00 6B 01 00 6C 02 00
00000030  5A 02 00 6C 03 00 5A 03 00 6C 04 00 5A 04 00 6C
00000040  05 00 5A 05 00 6C 06 00 5A 06 00 6C 07 00 5A 07
00000050  00 01 64 01 00 64 03 00 6B 08 00 6C 09 00 5A 09
00000060  00 6C 0A 00 5A 0A 00 6C 0B 00 5A 0B 00 6C 0C 00
```

Now if we run the file, then python will recognize it but ONLY python version 2.6 will recognize. Any other version will re-port as *Invalid Value of Magic header.*

Now the final step in this reversing endeveour is decompil-ing this *pyc* file back to its *py* source. Again we resort to the power of world wide web. Searching for python 2.6 decompilers, I came across quite a few of them. I tested each but was unsuccessful in majority of them. So in this document, I am only mentioning the one which I was sucessful with. It's the the *uncompyle2* decompiler. It can decompile python version 2.5, 2.6 & 2.7. However it will run ONLY on version 2.7.

The usage of the decompiler is well documented in the *readme* file that comes with it. So let's install it and run on the *pyc.*

```
C:\WINDOWS\system32\cmd.exe

C:\Python27\Scripts>python uncompyle2 -o register.py register.pyc
# 2014.01.09 09:27:12 India Standard Time
decompiled 1 files: 1 okay, 0 failed, 0 verify failed
# decompiled 1 files: 1 okay, 0 failed, 0 verify failed
# 2014.01.09 09:27:15 India Standard Time
```

Now, let's open the produced *py* file with our fingers crossed. The script says that everything was sucessful. The results are shown on the page after.

# And the final results

```
# Embedded file name: F:\project\youband_ver1.6\build\pyi.win32\digi\outPYZ1.pyz/register
"""
Module implementing RegisterDialog.
"""
from PyQt4.QtGui import QDialog, QMessageBox, QColor, QPixmap, QGridLayout, QTextEdit
from PyQt4.QtCore import pyqtSignature, QObject, SIGNAL, Qt, QSize, QRect, QCoreApplication
from Ui_RegisterGUI import Ui_Dialog
from Ui_VerdiffGUI import Ui_VerdiffDialog
import TEXTS
import wmi
import time
import base64
import traceback
import urllib, os, socket
SUPERKEY = ('SD130901GOTDZ4W3P1D', 'SD13TESTGOTDZ4W3P1D')

class register(QObject):

    def __init__(self, parent):
        super(register, self).__init__(parent)
        self.father = parent
        if TEXTS.SOFT_VERSION == TEXTS.TRAIL_VER:
            self.machine_code_list = ['112233445566']
        else:
            self.machine_code_list = self.getMachineCodeList()
        if TEXTS.LANGUAGE == TEXTS.CN_VER:
            self.forbidden_web = 'http://yuyinniaoniao.com/register/youband/check_forbid.php?code=%s' %
        elif TEXTS.LANGUAGE == TEXTS.JP_VER:
            self.forbidden_web = 'http://yuyinniaoniao.com/jp/register/youband/check_forbid.php?code=%s'
```

And it's SUCCESS!. The tool has indeed done a great job and in addition it has also generated the documentation strings which makes the reverser's job easier. We have succeeded to extract the original source code from the executable. We can even recompile the application from the generated sources, but as already said it is not the purpose of this document. With this, we come to the end of this mini tutorial. Hope you use your newly gained knowledge in a positive way.

This is *extremecoders* signing off, Ciao!