SANS | GIAC CERTIFICATIONS

WHITE PAPER

# Grow Your Own Forensic Tools: A Taxonomy of Python Libraries Helpful for Forensic Analysis

T.J. OConnor

**Copyright SANS Institute 2021. Author Retains Full Rights.**

This paper was published by SANS Institute. Reposting is not permitted without express written permission.

# Grow Your Own Forensic Tools: *A Taxonomy of Python Libraries Helpful for Forensic Analysis*

*GIAC (GCFA) Gold Certification*

Author: T.J. OConnor, terrence.oconnor@usma.edu
Advisor: Don Weber

Accepted: April 1st, 2010

Abstract

Python, a high-level language, provides an outstanding interface for forensic analysts to write scripts to examine evidence. Python is the driving language for several current open-source forensic analysis projects from Volatility, for memory analysis to libPST for abstracting the process of examining email. This paper provides a taxonomy of the different forensics libraries and examples of code that a forensic analyst can quickly generate using Python to further examine evidence.

© 2010 SANS Institute, Author retains full rights.

@ 2021 SANS Institute Author Retains Full Rights

# 1. Introduction

Forensics tools exist in abundance on the Web. Want to find a tool to dump the Windows SAM database out of volatile memory? Google and you will quickly find out that it exists. Want to mount and examine the contents of an iPhone backup? A tool exists to solve that problem as well. But what happens when a tool does not already exist? Anyone who has recently performed a forensic investigation knows that you are often left with a sense of frustration, knowing data existed only you had a tool that could access it.

In response, we present this paper on a taxonomy of Python libraries to support forensic analysis. In the following sections, we examine how you can grow your own tools in-house to solve specific problems. Want to search for Cisco VPN Configuration files and crack them? Want to plot imagery metadata geo-location information on a Google map? What about your own custom Windows Registry analysis tool? Specific analysis of malware or network dumps? In the following sections, we will write tools to accomplish all of this.

The high-level language Python has a omnipotence of modules and libraries, several of which can help us develop forensic tools. In this paper, we will examine how we can quickly put together tools for specific forensic investigations. In the following sections, we will take a look at using Python when working with encrypted files, extracting metadata, examining windows artifacts, tracking Web and email usage, foot-printing applications, carving artifacts from volatile memory, carving file systems, and analyzing network traffic.

# 2. Python for Forensic Analysis

## 2.1 Introduction to Python Modules

The Python programming language is a high-level, general-purpose language with clear syntax and a comprehensive standard library. Often referred to as a *scripting language*, security experts have singled out Python as a language to develop information security toolkits. The modular design, human-readable code, and fully developed suite of libraries provide a starting point for security researchers and experts to build tools.

TJ OConnor, terrence.oconnor@usma.edu
© 2010 SANS Institute, Author retains full rights.

By default, Python comes with an extensive standard library which provides everything from built-in modules providing access to simple I/O to platform-specific API calls. The beauty of Python is user-contributed individual modules, packages, and frameworks. Several of these user-default and user-contributed libraries, modules, packages, and frameworks can assist forensic analysts with building tools to accomplish interesting tasks. In the following sections, we will look at how a forensic analyst can use these tools to aid in investigations.

## 2.2 Crypto, Hash, and Conversion Functions

Let's begin by solving a very simple problem. Occasionally in the course of an investigation we run into encryption, where a target attempts to hide information. ROT-13 provides a very simple method a target may use to hide information. Yes, our targets may use much more complex algorithms to encrypt information, but Python also provides libraries for those algorithms, as you will see shortly. We will first use ROT-13, though, so we can get an understanding of how Python works.

The ROT-13 cipher is a simple cipher that substitutes each alphabetic character with the letter thirteen places further along, wrapping back if necessary, so the letter O becomes B, the letter P becomes C, etc. Encoding and decoding a ROT-13 cipher is relatively easy. The Python string library contains a function called *maketrans* that substitutes one character for another. Thus, we can declare a new function, ROT13 that performs ROT-13 cipher, as shown in Figure 1.

```
ROT13 = string.maketrans('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
'nopqrstuvwxyzabcdefghijklmNOPQRSTUVWXYZABCDEFGHIJKLM')
```

Figure 1. Function for encoding/decoding ROT-13.

In our script, we open all the files with the extension .txt on a target computer and run ROT-13 against each line of those files. After translating each line's ROT-13 encoding, we can check to see if the line now contains any legitimate words from a file of dictionary words. If it does, we will print a message that we found an ROT-13-encoded message. Figure 2 depicts this in a Python script.

```
import sys, os, string

ROT13 =
string.maketrans('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ','nopqrstuvwxyz
```



@ 2021 SANS Institute Author Retains Full Rights

© 2010 SANS Institute, Author retains full rights.

```
abcdefghijklmNOPQRSTUVWXYZABCDEFGHIJKLM')

dictionaryFile = open("dictionary")
dictionary = dictionaryFile.readlines()

for root, dir, files in os.walk(str(sys.argv[1])):
	for file in files:
		if ".txt" in str(file):
			foundWord = 0
			notFound = 0
			lines = open(file).readlines()
			for line in lines:
				translatedLine=line.translate(ROT13)
				translatedWords=translatedLine.split()
				for eachWord in translatedWords:
					if (eachWord+'\n') in dictionary:
						foundWord=foundWord+1
					else:
						notFound = notFound+1
			if (foundWord > notFound):
				print file+" may contain ROT-13 encoding."
```

Figure 2. ROT-13 detection script.

Yes, the likelihood that you will encounter a ROT-13 cipher on an actual forensic investigation is minimal. However, we used the last example to demonstrate a basic function of Python. It is also important to understand ROT-13 when decoding elements like Windows Registry keys, as we will see later. Let's examine some of the other encryptions and encodings. First, we will build a small Python program to search a target machine for Cisco PCF files, which contain the configuration settings for a Cisco IPSEC VPN Tunnel.

These configuration files also contain an interesting line, beginning with either **enc_GroupPwd**= or **enc_UserPassword=**. This line contains the encrypted group or user password. Unfortunately, this password is hashed using a relatively weak encryption algorithm and can be converted back to the original plaintext password. Other security researchers have demonstrated this previously; however, we built a Python library that can be imported into your program. LibCiscoCrack provides a function to reverse the encrypted passwords back to their plaintext encodings. Having the capability to enumerate a file system, looking for PCF files, and the ability to decode passwords, we can build a VPN password cracker in Python in less than five minutes, as shown in Figure 3.



© 2010 SANS Institute, Author retains full rights.

@ 2021 SANS Institute Author Retains Full Rights

```
Import os, sys, libCiscoCrack

for root, dir, files in os.walk(str(sys.argv[1])):
        for file in files:
            if ".pcf" in file:
                lines = open(file).readlines()
                for line in lines:
                    if "password" in line:
                        plainTextPass = libCiscoCrack.crack(line)
                    print plainTextPass
```

Figure 3. Cisco VPN password detection and cracking script.

Finding PCF files on the file system and cracking them is not where it ends. As you'll see throughout the following sections, we can write Python programs to search the unallocated file space—the volatile memory. Python provides libraries and modules for decoding base64, symmetric key encryption, and hex encoding as well. Converting a base64 encoded string to its plaintext equivalent takes three lines of code (see Figure 4).

```
import base64

msg = raw_input("Enter Base64 Message:")
print "Decoded Val:"+base64.decodestring(msg)
```

Figure 4. Example of Base64 decoding.

Next, imagine a scenario where we have several keys for a symmetric key encryption algorithm like DES, and a message we suspect of being encrypted using one of those keys. We'll store each of those candidate keys in a file called "keys.txt." As you see in Figure 5, we can import the pyCrypto library and write a script to exhaustively search those keys, decoding our ciphertext message (ciph).[1] After decoding the file with a different key each time, we'll count the number of alphanumeric characters to see if there is a possible message. If the decoded text contains mostly alphanumeric characters, we will print out a message showing the candidate key and what it decodes the message to.

[1] pyCrypto is a separate module, which must be installed after installing the Python standard libraries. For more on installing additional modules or libraries, see the documentation on the Python Web site: http://docs.python.org/install/

TJ OConnor, terrence.oconnor@usma.edu

© 2010 SANS Institute, Author retains full rights.

```
import base64, string
from Crypto.Cipher import DES
THRESH = 0.9

keyFile = open("keys.txt")
keys = keyFile.readlines()
ciph=base64.decodestring("cG0okyHpOAADuNLv8bRpxpyZeU8kMA2kWV2zoV+YUos=")

for key in keys:
   obj=DES.new(key[0:8], DES.MODE_ECB)
   decodedStr=obj.decrypt(ciph)
   foundLetters = 0
   for eachChar in decodedStr:
      if eachChar.isalpha() or eachChar.isdigit()\
      or eachChar.isspace():
         foundLetters = foundLetters+1
   if (float(foundLetters)/float(len(decodedStr)) > THRESH):
      print "DES(ciphertext,"+key[0:8]+")="+decodedStr
```

Figure 5. Example of symmetric key decryption.

Now we have our own DES brute-force cracking engine in fewer than 20 lines of code. Sure, some might argue that Python runs rather slow, as it is interpreted: Psyco might help in that situation (Psyco, 2010). Psyco is a Python extension that significantly speeds up the execution of code. Relatively easy to implement, a coder can simply call import psyco and then psyco.full() in the first lines of code. This will force the following Python code to use the psyco module and (*hopefully*) run much faster.

One tool forensic investigators use to sort through large amounts of data to find suspect data for further investigation is a cryptographic hash. Both MD5 and SHA256 are great choices for implementing a hash. Inputting an executable picture, or media file into a hash algorithm results in a relatively shorter string that uniquely identifies that file. Databases of hashes exist for known benign software, known malicious programs, emerging threats, and child pornography. Let's look at two of those repositories to build a homebrew virus-scanning program in Python.

First, the NIST National Software reference library maintains a list of known software. If you are trying to explain whether or not a file should exist on a system under investigation, you can query the NIST NSRL to determine if it belongs there. Second, Team Cymru maintains a registry of known malware hashes. If you suspect a file contains spyware, a Trojan, or malicious code, you can check to see if it is in the malware hash registry. The SANS Internet Storm Center provides an interface for querying both

TJ OConnor, terrence.oconnor@usma.edu

© 2010 SANS Institute, Author retains full rights.

repositories over the DNS protocol. Using Python, we can write a script to scan a file system, submitting files to the database for comparison. For those files that hit on the Malware Hash Registry, we'll print the information. Figure 6 shows this exact program, written by a student of mine, Kevin Cullberg (2010), in less than 30 lines of code.

```
import os, hashlib, sys, socket, string

for root, dir, files in os.walk(str(sys.argv[1])):
        for fp in files:
                try:
                        # open a file and calculate the md5 hash
                        fn = root+fp
                        infile = open(fn, "rb")
                        content = infile.read()
                        infile.close()
                        m = hashlib.md5()
                        m.update(content)
                        hash = m.hexdigest()
                        # send the md5 hash the Team Cmuru for inspection
                        mhr = socket.socket(socket.AF_INET,\
             socket.SOCK_STREAM)
                        mhr.connect(("hash.cymru.com", 43))
                        mhr.send(str(hash + "\r\n"))
                        response = ''
                        # wait for the response from Team Cymru
                        while True:
                                d = mhr.recv(4096)
                                response += d
                                if d == '':
                                        break
                        # if the response is malware - print filename
                        if "NO_DATA" not in response:
                                print "<INFECTED>:"+str(fn)
                except:
                        pass
```

Figure 6. A homebrew virus scanner in less than 30 lines of code.

After detecting known malware, a forensic investigator has knowledge of what is definitely malicious on a system. But what if an investigator wants to look further at benign files? As you'll see in the next section, there are several libraries that can assist us in looking into file metadata, or data that describes data.

## 2.3 File Metadata Extraction

File metadata can prove very useful for an analyst during a forensic investigation: it can provide information on who created the file, when it was created, and with what

TJ OConnor, terrence.oconnor@usma.edu

© 2010 SANS Institute, Author retains full rights.

@ 2021 SANS Institute Author Retains Full Rights

tool; it can even provide information such as where the file was created. Imagine the following scenario: we begin an investigation and find thousands of pictures taken with

```
import string,sys,os
from PIL import Image
from PIL.ExifTags import TAGS

for root, dir, files in os.walk(str(sys.argv[1])):
        for fp in files:
          if ".JPG" in fp.upper():
         # open a file and extract exif
         fn = root+fp
         try:
           i = Image.open(fn)
           info = i._getexif()
           exif={}
           for tag, value in info.items():
              decoded = TAGS.get(tag, tag)
              exif[decoded]=value
           # from the exif data, extract gps
           exifGPS = exif['GPSInfo']
           latData = exifGPS[2]
           lonData = exifGPS[4]
           # calculate the lat / long
           latDeg = latData[0][0] / float(latData[0][1])
           latMin = latData[1][0] / float(latData[1][1])
           latSec = latData[2][0] / float(latData[2][1])
           lonDeg = lonData[0][0] / float(lonData[0][1])
           lonMin = lonData[1][0] / float(lonData[1][1])
           lonSec = lonData[2][0] / float(lonData[2][1])
           # correct the lat/lon based on N/E/W/S
           Lat = (latDeg + (latMin + latSec/60.0)/60.0)
           if exifGPS[1] == 'S': Lat = Lat * -1
           Lon = (lonDeg + (lonMin + lonSec/60.0)/60.0)
           if exifGPS[3] == 'W': Lon = Lon * -1
           # print file
           msg=fn+" located at "+str(Lat)+","+str(Lon)
           print msg
         except:
           pass
```

Figure 7. Script to extract geo-location information from images.

an iPhone. A Python script can allow us to quickly iterate through those files, plotting each location on a Google map.

To build such a script, let's start first by importing the Python Image Library (PIL). PIL allows us to extract the Exchangeable Image file format (EXIF) data from images (Python Imaging Library, n.d.). Among other metadata, EXIF contains the latitude and longitude of image files. As you see in Figure 7, we can write a script to extract the latitude and longitude for pictures containing geo-location data.

© 2010 SANS Institute, Author retains full rights.



@ 2021 SANS Institute Author Retains Full Rights

Next, we'll use the extracted latitude and longitude coordinates to plot out a Google map. Google Maps use the KML file format to display geographic data (Apiolaza, 2009). To display our images, we'll build a KML file that we can subsequently import into Google Maps. Figure 8 builds a KML document with the place marks of the geo-locations of the images we recently found.

```
kmlheader = '<?xml version="1.0" encoding="UTF-8"?>\n'\
'<kml xmlns="http://www.opengis.net/kml/2.2">\n'

kml = (
  '<Placemark>\n'
  '<name>%s</name>\n'
  '<Point>\n'
  '<coordinates>%6f,%6f</coordinates>\n'
  '</Point>\n'
  '</Placemark>\n'
  '</kml>'
  ) %(fp,longitude, latitude)

kmldoc = kmlheader + kml
print kmldoc
```

Figure 8. Script to build KML to import into Google Maps.

Running our new script, *images2KML.py*, against a directory of images results in producing a Google map (Figure 9) with plotted points for each image.

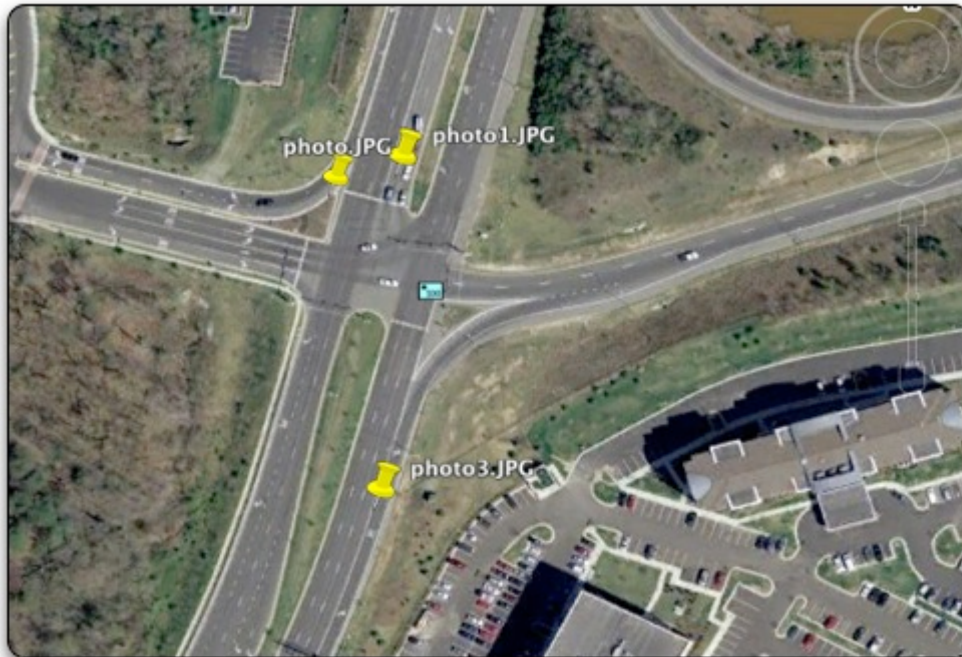


Figure 9. Image locations plotted on Google Maps.

TJ OConnor, terrence.oconnor@usma.edu

© 2021 SANS Institute Author retains full rights.

After decoding information about the location of files, we may want to look at the authors of other file types. pyPDF provides a Python library capable of extracting such document information, document merging, splitting, cropping and encrypting and decrypting PDF files (Fenniak, 2010). The Document Information Class of PDF can return information about the author, creator, producer, subject, or title of a PDF File.

Looking for files created by a particular author or length? Let's write a small program that will look for PDF files longer than five pages, created by "Dr Evil". You'll notice that Figure 10 shows how we can import pyPDF in and define a PdfFileReader object that reads in each file, checking the title, author, and number of pages. If we pass our comparison test, we print the file name and length to the screen. You may also notice we suppress warnings and also use a try/except scheme to catch errors. If you try to parse through all the documents on a given file system, you will run into plenty of errors. If you want to know more about which files cause errors, you could add a print statement in the except scheme instead of just passing by; for now, our script helps by automating the process of finding some low-hanging forensics fruit.

```
import warnings,sys,os,string
from pyPdf import PdfFileWriter, PdfFileReader
warnings.filterwarnings("ignore")

for root, dir, files in os.walk(str(sys.argv[1])):
        for fp in files:
      if ".pdf" in fp:
        fn = root+"/"+fp
        try:
          pdfFile = PdfFileReader(file(fn,"rb"))
          title = pdfFile.getDocumentInfo().title.upper()
          author = pdfFile.getDocumentInfo().author.upper()
          pages = pdfFile.getNumPages()

          if (pages > 5) and ("DR EVIL" in author):
            resultStr = "Matched:"+str(fp)+"-"+str(pages)
```

Figure 10. Metadata extraction from PDF file types.

In addition to multimedia and PDF types, an investigator can find a rich source of metadata information inside of Microsoft Office documents. Microsoft Office stores information inside Word (DOC), Excel Workbook (XLS), and PowerPoint (PPT) presentations in binary format using a basic container structure called OLE2, which can contain information about everything from the author to the embedded pictures inside the



© 2010 SANS Institute, Author retains full rights.
@ 2021 SANS Institute Author Retains Full Rights

document. Now, let's see how we can use the OLE2 format to help a forensic investigation.

Imagine a scenario where we suspect a user has downloaded a Microsoft Office document that has embedded VBA macros. The macros caused damage to the compromised system, so we want to identify the document that caused the problem and reverse-engineer it. We can use the OleFileIO_PL library to read the Microsoft OLE streams from files, detecting those that have "macro/vba" in their OLE data (see Figure 11). Notice that we use the except/pass to walk right over files that raise an error for either not being an OLE2 file type or that contain no OLE2 streams. This allows us to find Microsoft OLE files missing a Microsoft Office extension.

```
import OleFileIO_PL,os,sys,string

for root, dir, files in os.walk(str(sys.argv[1])):
        for fp in files:
         fn = root+fp
         try:
            ole = OleFileIO_PL.OleFileIO(fn)
            if ole.exists('macros/vba'):
               print fn+":"+" contains vba macros."
         except:
            pass
```

Figure 11. Metadata extraction from OLE2 (MS Office) file types.

After a thorough examination of metadata contained in files, we may want to look deeper inside the Operating System internals of our target. In the next section, we will provide tools for examining the configuration and settings inside the Windows Operating System.

## 2.4 Examining Microsoft Artifacts

The Windows Registry stores configuration settings and options in a hierarchical database on the Microsoft Windows Operating System. The Registry provides a forensically rich environment, containing everything from the wireless keys for WPA-PSK to AutoComplete Passwords in Internet Explorer to a list of recently opened documents and programs (Carvey, 2007). Specific applications stored their own registry keys as well, making endless possibilities of locations to check.

Depending upon the environment, an analyst can access the Registry via a series of built-in tools such as REGEDIT or REGEDIT32. The Windows Powershell also


© 2010 SANS Institute, Author retains full rights.
@ 2021 SANS Institute Author Retains Full Rights

provides a powerful interface for grabbing registry keys. But what if you want to access a Windows Registry during offline analysis inside a forensic distribution such as SIFT, DEFT, or HELIX?

The RunMRU keu provides an interesting key inside the Windows Registry. It shows all the recently run commands from the start menu. Figure 12 shows how to pull the values out of this key and display them on the screen. Notice that we first open a winreg key, then enumerate through the list of values for the key. Each value is printed to the screen.

```
import _winreg

RunMRUKey = _winreg.OpenKey(
   _winreg.HKEY_CURRENT_USER,
   "Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\RunMRU")

try:
   i = 0
   while 1:
      name, value, type = _winreg.EnumValue(RunMRUKey, i)
      print name+":"+value
      i += 1
except WindowsError:
   print
```

Figure 12. Examining windows registry keys using Python.

In addition to the Registry, the Windows operating system family has several unique forensic artifacts. Several python based-tools exist to analyze these artifacts. An example of a tool constructed entirely in Python to solve a Windows forensic problem is Vinetto (Monniez, 2010). Vinetto is a forensic tool to examine a Thumbs.db file from the command line: this can be useful for a forensic investigator performing an investigation. Vinetto can display metadata contained within a Thumbs.db file, extract the related thumbnails and place them in a directory, or produce an html report. Figure 13 depicts the results of running Vinetto against a Thumbs.db file to produce images.

© 2010 SANS Institute, Author retains full rights.



@ 2021 SANS Institute Author Retains Full Rights

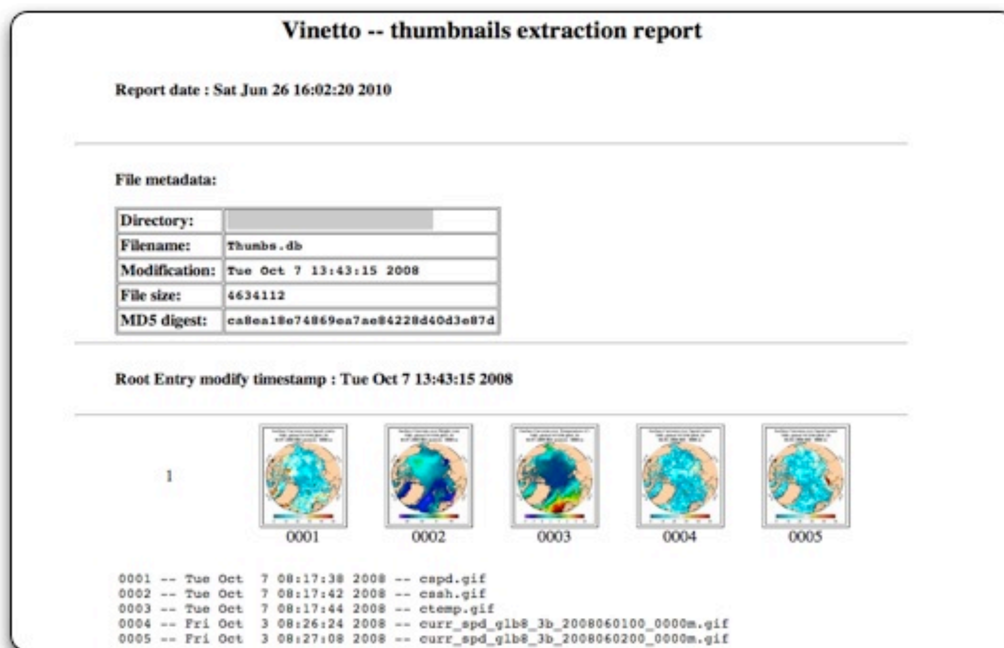


Figure 13. Sample output from Vinetto.

## 2.5 *NIX Artifacts

PyUTMP module provides a Python-oriented interface to the utmp file on the Unix operating system (Clapper, 2010). The utmp file provides information about which users are currently logged onto a system. However, the utmp structure is binary and cannot be read by a simple text editor: rather, each Operating System (Solaris, Unix, Linux) provides a series of tools for reading the binary structure.

Image a scenario where you must perform forensics on a live Linux system. You suspect that the standard binaries for displaying utmp information have been compromised by application-level root-kits and you want to read information in the utmp binary using a script. Further forensic analysis leads you to believe that the application-level root-kit was installed the week of June 20th, 2010. Figure 14 shows how we can detect the users who logged into our system the week of June 20th, bypassing using standard binaries that may have been compromised in the initial attack.

TJ OConnor, terrence.oconnor@usma.edu

@ 2021 SANS Institute Author Retains Full Rights

© 2010 SANS Institute, Author retains full rights.

```
import time
from pyutmp import UtmpFile

start = time.mktime(time.strptime("12:00-20.06.2010","%H:%M-%d.%m.%Y"))
end = time.mktime(time.strptime("12:00-27.06.2010","%H:%M-%d.%m.%Y"))

for utmp in UtmpFile():
        checkTime = time.ctime(utmp.ut_time)
        if start < utmp.ut_time < end and\
                utmp.ut_user_process:
                        print '%s logged in at %s' % (utmp.ut_user,\
                time.ctime(utmp.ut_time))
```

Figure 14. Example script to read the UTMP binary on a Unix system.

## 2.6 Tracking Email and Web Client Usage

Interfacing with Web and email clients can assist in a forensic investigation. A case might exist where you find the credentials for a pop email account and wish to investigate further. The account contains thousands of messages that must be filtered against a massive set of keywords already established in your investigation.

```
import poplib
from email import parser

keyWordFile = open("keywords.txt")
keyWords = keyWordFile.readlines()

pop_conn = poplib.POP3_SSL('pop.gmail.com')
pop_conn.user('username')
pop_conn.pass_('password')
messages = [pop_conn.retr(i) for i in range(1, len(pop_conn.list()[1]) + 1)]
messages = ["\n".join(mssg[1]) for mssg in messages]
messages = [parser.Parser().parsestr(mssg) for mssg in messages]
pop_conn.quit()

for message in messages:
   for keyWord in keyWords:
        if keyWord in message['subject']:
          print message['subject']
```

Figure 15. Email/web example.

Python presents a library capable of both connecting to the pop server using poplib and parsing the messages using the parser package from the email module. In the

© 2010 SANS Institute, Author retains full rights.



@ 2021 SANS Institute Author Retains Full Rights

example above (Figure 15),[2] we demonstrate how to connect to a pop server, dump the messages in their entirety and parse them against a list of keywords from the file *keywords.txt.*

## 2.7 Footprinting Applications

Windows uses the Portable Executable (PE) file format for executable code object code and Dynamic Link Libraries (DLLS). Using a combination of two libraries, PEFile (Carrera, 2006a) and PYDASM (Carrera, 2006b), we can build a small disassembler. This can help a forensic analyst to determine if a program loaded the address of a particular system call, for example the syscall, to bind a network socket (Seitz, 2009).

A few years back the SPYLOCKED Trojan caused quite a bit of damage. SPYLOCKD downloaded a Trojan DLL and Executable to a computer and then notified the user that virus protection had expired and needed to be upgraded. If the user clicked and *upgraded* the virus protection, he/she further infected the box with malware. Almost immediately after the attack, anti-virus vendors created a signature for the malicious DLL that the application hooked to. Finding the DLL and deleting it from the system corrected the issue.

But what if you wanted to know which pieces of new executable code on your system hooked to that DLL? Enter PEFile. We can write a quick Python script using the PEFile library to scan all executables in Figure 16, looking at which DLLs they are hooked to and printing out any executables that hook to "SPLOCKD.DLL." We will also print out the other DLLS and IMPORTS used by the malicious executable for further analysis.

```
import pefile,sys,os,sys

for root, dir, files in os.walk(str(sys.argv[1])):
        for fp in files:
      if ".exe" in fp:
        try:
          pe = pefile.PE(root+"/"+fp)
          for entry in pe.DIRECTORY_ENTRY_IMPORT:
            if "SPYLOCKD.dll" in entry.dll:
              print fp+" hooked to SPYLOCKD."
```

---

[2] This example is based an anonymous post on the Stack Overflow Web site: http://stackoverflow.com/questions/1225586/checking-email-with-python

TJ OConnor, terrence.oconnor@usma.edu

© 2010 SANS Institute, Author retains full rights.

```
          except:
            #no linked DLLs - passing
                pass
```

Figure 16. Displaying DLLs linked from a Windows portable executable.

## 2.8 Cracking Encryption and Steganography

So far we've used some forensic-tailored Python libraries. However, our creation of forensic analysis tools is not limited to forensic libraries only. We can use standard Python modules and libraries to perform our analysis. Consider the example of a system where we find several password-encrypted zip files that we want to try to crack against a

```
import zipfile

zFile = open(sys.argv[1],"r")
passFile = open(sys.argv[2],"r")
passwords = passFile.readlines()

for password in passwords:
     try:
          for info in zfile.infolist():
               fname = info.filename
               print "trying..."+str(password)
               data = zfile.read(fname,str(password))
               print "password found:"+str(password))
               break
     except Exception, e:
          print e
               if ('Bad password') in e: pass
```

Figure 17. Brute-force cracking zip files using Python.

custom word list of passwords. We can use the default module for handling zip files.

For encrypted or password-protected zip files, the zip-file module provides a function call to extract files. When called with the wrong password, the function raises an error. That certainly helps us, as we can enumerate down a list, continuing on if we raise an error and only exiting when our script passes the correct password to decode. Figure 17 shows an example for a custom zip file password cracker.

If we wanted to get fancy, we could combine this with other tools, such as dumping all the ASCII or UNICODE strings in volatile memory, hoping that memory might have the contents of a recently used password on the system. Volatile memory, as you'll see in the next section, can prove very useful in a forensic investigation.



@ 2021 SANS Institute Author Retains Full Rights

© 2010 SANS Institute, Author retains full rights.

## 2.9 Carving Volatile Memory: *Volatility*

Analyzing the contents of volatile memory (RAM) to find digital artifacts can give an investigator insight into the current state of a system. Volatile memory analysis can enable an investigator to discover open network connections, recently used passwords, deleted files, the process table, or even the contents of the Windows registry. However, carving those artifacts out of memory can be a rather challenging task. Enter Volatility.

Volatility is currently one of the largest open-source projects for digital forensics, with a growing repository of code samples (Walters, 2010). The project provides a framework of Python libraries for extracting digital artifacts from volatile memory. Because the toolkit acts as a framework, it abstracts away the underlying operating system. This enables an investigator to build independent Operating System tools to examine the contents of the memory.

The structure of Volatility allows investigators to develop modules for extracting specific information out of RAM. By default, the framework comes with plug-in modules to:

- Print the list of open connections and scan for connection objects
- Print a list of loaded dlls for each process
- Dump crash dump information and convert it to a raw dump
- Show the files open for each process and dump processes to executable
- Identify image properties, including data, time and location
- Print a list of registry keys for each process found in the process table

The authors of Volatility created the code to be extensible: thus, several investigators have extended the Volatility framework by building third-party plug-in modules. Some of the more interesting plug-in modules include:

- CryptoScan – Finds TrueCrypt passphrases.
- Suspicious – Finds “suspicious” processes.
- Keyboardbuffer – Extracts keyboard buffer used by the BIOS.
- Getsids – Get information about what user (SID) started processes.

Extending Volatility for your own purposes is relatively easy. To start, download Volatility from the distribution site and create a new file in the *memory_plugins*

TJ OConnor, terrence.oconnor@usma.edu

© 2010 SANS Institute, Author retains full rights.

@ 2021 SANS Institute Author Retains Full Rights

directory. You can call the file whatever you wish. In our example, we will extract the running processes from a memory dump. This example is an abbreviated form of the examples included by the author, Aaron Walters (2010).

In Figure 18, you will notice we have to declare a class for our new plug-in and define both *help(self) and execute(self)* methods. In our execute method, we use the built-in function *process_list() to* dump the process list, we then iterate through the process list, printing the process name and process id for each process found in volatile memory.

```
from vutils import *
from forensics.win32.tasks import *

class getPids(forensics.commands.command):

   def help(self):
      return  "Print list running processes"

   def execute(self):

         (addr_space, symtab, types) = \
     load_and_identify_image(self.op, self.opts)
      all_tasks = process_list(addr_space,types,symtab)
      print "%-20s %-6s" %('Name','Pid')

      for task in all_tasks:
         if not addr_space.is_valid_address(task): continue
         image_file_name = \
           process_imagename(addr_space, types, task)
          process_id = process_pid(addr_space,types,task)
          print "%-20s %-6d" % (image_file_name, process_id)
```

Figure 18. Volatility example.

## 2.10 Analyzing Network Traffic: Scapy

The Scapy toolkit is a powerful packet-manipulation program, capable of decoding a wide variety of protocols (Biondi, 2010). The supported protocols include IP, TCP, UDP, ICMP, 802.11, and even Bluetooth. Scapy differs from standard tools by providing an investigator the ability to write small Python scripts that can investigate network traffic. An investigator can write Scapy scripts to investigate either real-time traffic by sniffing a promiscuous network interface, or load previously captured pcap files. Furthermore, the Scapy developers included the ability to perform deep-packet dissection, passive OS fingerprinting, or plotting via third-party tools like GnuPlot.

© 2010 SANS Institute, Author retains full rights.

What is really exciting is that an investigator can write four lines of a Scapy program to do what used to take hundreds of lines of C code or multiple-command switches in tcpdump. The capabilities for an investigator to investigate traffic are endless: *make a graph of network destinations, count the number of 802.11 deauth packets in a time period to detect an attack, or make a table of the IP IDENT fields to detect covert TCP packets.*

Let's examine a situation where an investigator may want to record statistics about the geo-location of the IP address source and destination. After importing Scapy, we'll call the function *sniff* in our main() procedure. We can use a filter to detect only IP packets and then pass those packets with an IP layer to a subsequent function we define, called *prnPkt*. The subsequent function looks up the geo-location information for the source and destination based on the IP address fields extracted from each packet.

```
import scapy, GeoIP
from scapy import *

gi = GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)

def prnPkt(pkt):
        src=pkt.getlayer(IP).src
        dst=pkt.getlayer(IP).dst
        srcCo = gi.country_code_by_addr(src)
        dstCo = gi.country_code_by_addr(dst)
        print srcCo+">>"+dstCo

try:
        while True:
            sniff(filter="ip",prn=prnPkt,store=0)
except KeyboardInterrupt:
        print "\nExiting.\n"
```

Figure 19. Packet-sniffing using Scapy.

## 2.11 Stand-Alone Python Tools

AnalyzeMFT is a stand-alone Python script designed to fully parse the Master File Table (MFT) from an NTFS file system and present the results in human-readable format (Kovar, 2010). AnalyzeMFT is constructed entirely in Python and for each MFT record can record if the entry is valid, type of record, parent folder record and sequence, standard information attributes, file name records, object IDs, birth Volume ID, Domain, flags and notes. AnalyzeMFT differs from some of the other Python libraries and modules that we have reviewed in the fact that it is a stand-alone script. However, it



© 2010 SANS Institute, Author retains full rights.

@ 2021 SANS Institute Author Retains Full Rights

provides an excellent example of how an investigator can use the Python programming language to build a comprehensive toolkit to solve a forensic problem.

Another toolkit created entirely in Python to solve forensics problems is the pyFlag (Forensics and Log Analysis GUI) project. pyFlag provides an advanced forensic tool for the analysis of large volumes of log files for forensic investigators (Cohen, n.d.). The current version of pyFlag includes the following features:

- Network Forensics – analyzes network captures in TCPDump format.
- Log Analysis – capable of reading many log formats and multiple methods for querying log data.
- Disk Forensics – supports a large number of file formats, carving techniques, and hard disk drive analysis.
- Memory Forensics – based on the Volatility Framework, can perform limited analysis of volatile memory.

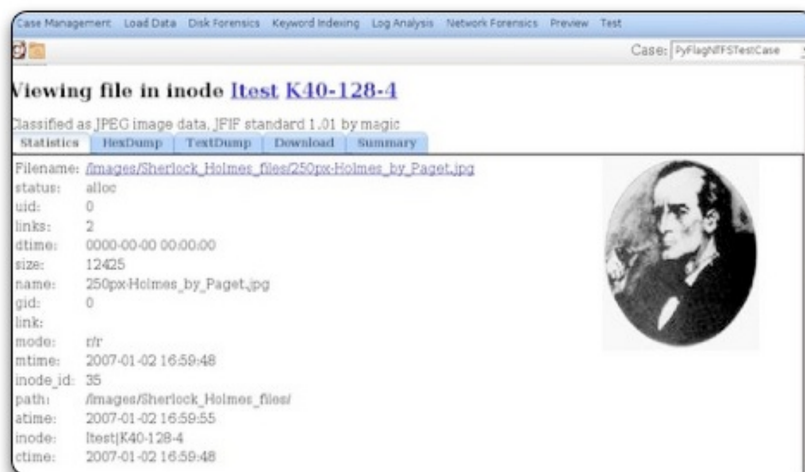


Figure 20. Screenshot of pyFlag forensic tool.

# 3 Conclusions

In this paper we examined using Python to perform a variety of forensic collection and analysis tasks. We demonstrated using Python to work with encrypted files, to extract metadata, to examine windows artifacts, to track Web and email usage, to foot print applications, to carve artifacts from volatile memory, to carve file systems, and



@ 2021 SANS Institute Author Retains Full Rights

© 2010 SANS Institute, Author retains full rights.

to analyze network traffic. Additionally, this paper introduced some stand-alone toolkits built entirely in Python. Python proves an excellent language for creating tools for forensic analysis because of its easy-to-understand pseudo-code like syntax and abundance of standard libraries and thirty-party libraries and modules.

In the preceding sections we presented an abundance of these third-party libraries and modules used to create tools to analyze specific forensic problems. Although this paper has addressed and introduced several libraries for forensic analysis, it is no way conclusive. Appendix A records some of the tools we have discussed in the course of this paper as well as other useful libraries and modules for writing forensic scripts. However, everyday new libraries are added to the Python standard distribution and third-party libraries continue to grow in abundance as well. Several libraries such as libForensics and Volatility have begun implementing Python frameworks for development of further tools for specific niches of forensic analysis. Learning to import these libraries and write code for future forensic investigation can prove a rather useful tool in the overall development of a forensic investigator.

We have considered several of the advantages of writing custom Python tools for forensic analysis. Primarily, it gives investigators the ability to solve novel problems. Additionally, it gives the investigator the confidence to clearly articulate how the tool has manipulated working or actual copies of the data during the course of an investigation. Next, it can reduce the actual amount of time required to perform analysis by restricting the tool to the specific data we are attempting to extract, interpret, and analyze instead of relying on the full functionality of an all-encompassing forensics suite.

In conclusion, Python proves a rather useful tool in a forensic analyst's arsenal considering the relative ease it takes to create simple Python scripts, the abundance of libraries, and the constant necessity for forensic investigators to solve novel problems to acquire, interpret and analyze data. If you are not writing scripts to assist in your forensic investigations already, we definitely encourage you to begin now.

© 2010 SANS Institute, Author retains full rights.


@ 2021 SANS Institute Author Retains Full Rights

# 4. References

Apiolaza, L. (2009, February 1). Generating dynamic Google maps with Python. Retrieved from http://quantum.uncronopio.org/

Biondi, P. (2010). Scapy – Powerful Packet Manipulation Program. Retrieved from http://www.secdev.org/projects/scapy/

Carrera, E. (2006a). PEFILE – a Python module to read and work with PE (Portable Executable) files. Retrieved from http://www.pythonware.com/products/pil/

Carrera, E. (2006b). Pydasm: Introduction. Retrieved from http://dkbza.org/pydasm.html

Carvey, H. (2007). Windows forensic analysis. Burlington, MA: Syngress Publishing, Inc.

Clapper, B. (2010). PyUTMP—Python interface to Unix utmp. Retrieved from http://bmc.github.com/pyutmp/

Cohen, M. (n.d.) PyFlag – Python Forensic Log Analysis GUI. Retrieved from http://www.pyflag.net

Cullberg, Kevin (2010). Homebrew virus scanner. CS485F Course Web site. West Point, NY. Retrieved from http://www-internal.eecs.usma.edu/CS485F

Fenniak, M. (2010). PyPdf. Retrieved from http://pybrary.net/pyPdf/

Kovar, D. (2010). AnalyzeMFT — A Python tool to deconstruct the Windows NTFS $MFT file. Retrieved from http://www.integriography.com/

Monniez, Christophe (2010). Vinetto. Retrieved from http://sourceforge.net/projects/vinetto/

Python Imaging Library (PIL). (n.d.). Retrieved from http://wwwc.Pythonware.com/products/pil/

Psyco [Web site]. (2010). Retrieved from http://psyco.sourceforge.net/

Seitz, J. (2009). Gray Hat Python. San Francisco, CA: No Starch Press.

Walters, A. (2010). Volatility – Memory Forensics. Retrieved from https://www.volatilesystems.com

© 2010 SANS Institute, Author retains full rights.

TJ OConnor, terrence.oconnor@usma.edu

@ 2021 SANS Institute Author Retains Full Rights

## Appendix A – Useful Libraries and Modules For Analysis

| | |
|---|---|
| AnalyzeMFT | A Python Tool to deconstruct the Windows NTFS $MFT file. Available at http://www.integriography.com/. |
| Atlasutils | Useful python utilities and scripts. Available at http://atlas.r4780y.com/resources/atlasutils-current.tgz. |
| Disass-3.0 | A Python Library for disassembling executables for anlaysis. Available at http://atlas.r4780y.com/resources/disass-3.0-080424.tgz. |
| GrokEVT | A collection of scripts built for reading Windows® NT/2K/XP/2K3 event log files. Available at http://projects.sentinelchicken.org/grokevt/. |
| Hashlib | A standard Python library responsible for calculating secure message digests and hashes. |
| LibDisassemble | A Python Library for disassembling executables, required for PEFile. Available at http://atlas.r4780y.com/resources/libdisassemble-2.5-080424.tgz. |
| LibForensics | A Python library for developing digital forensic applications. Requires Python 3.1. Available at http://code.google.com/p/libforensics/. |
| Mount_EWF.PY | Allows mounting storage media data in EWF Files. Available at http://www.forensicswiki.org/wiki/Libewf. |
| PEFile | A multi-platform Python module to read and work with Portable Executable (aka PE) files. Available at http://code.google.com/p/pefile/. |
| PIL | Python Imaging Library adds image processing capability to Python interpreter. Available at http://www.Pythonware.com/products/pil/. |
| Psyco | Python extension which can be applied to speed up execution of any Python code. Available at http://psyco.sourceforge.net.. |

© 2010 SANS Institute, Author retains full rights.


@ 2021 SANS Institute Author Retains Full Rights

| | |
|---|---|
| PyCrypto | The Python Cryptography Toolkit. Available at http://www.pycrypto.org. |
| PyDasm | A Python interface to the libdasm, x86 dissembler. Available at http://dkbza.org/pydasm.html. |
| PyFlag | An advanced forensic tool, written in Python, capable of analysis of large volumes of log files. Available at http://www.pyflag.net. |
| PyPDF | A Python Toolkit capable of extracting, splitting, merging, cropping, and reading metadata from PDF Files. Available at http://pybrary.net/pyPdf/. |
| PySimReader | A Python Toolkit for extracting information from SIM Cards. Available at http://www.ladyada.net/media/simreader/pySimReader-Serial-src-v2.zip. |
| PyUTMP | A Python interface to the Unix UTMP file. Available at http://bmc.github.com/pyutmp/. |
| Scapy | A powerful packet manipulation library, capable of decoding several different protocols. Available at http://www.secdev.org/projects/scapy/. |
| String | A Python module from the standard library capable of performing a variety of tasks with strings. |
| Vinnetto | A Python tool capable of extracting thumbnail images and their metadata. Available at http://sourceforge.net/projects/vinetto/. |
| Volatility | A complete collection of tools for analysis of artifacts from volatile memory. Available at https://www.volatilesystems.com/default/volatility. |
| ZipFile | A standard Python library capable of reading, uncompressing, and creating zip compressed files. |

© 2010 SANS Institute, Author retains full rights.


@ 2021 SANS Institute Author Retains Full Rights