tutorialspoint

SIMPLY EASY LEARNING

About the Tutorial

Digital forensics is the branch of forensic science that analyzes, examines, identifies as well as recovers the digital evidences from electronic devices. It is commonly used in criminal law and private investigation. This tutorial will make you comfortable with performing Digital Forensics in Python on Windows operated digital devices. In this tutorial, you will learn various concepts and coding for carrying out digital forensics in Python.

Audience

This tutorial will be useful for graduates, post graduates, and research students who either have an interest in this subject or have this subject as a part of their curriculum. Any reader who is enthusiastic about gaining knowledge digital forensics using Python programming language can also pick up this tutorial.

Prerequisites

This tutorial is designed by making an assumption that the reader has a basic knowledge about operating system and computer networks. You are expected to have a basic knowledge of Python programming.

If you are novice to any of these subjects or concepts, we strongly suggest you go through tutorials based on these, before you start your journey with this tutorial.

Copyright & Disclaimer

Table of Contents

1. Python Digital Forensics — Introduction

This chapter will give you an introduction to what digital forensics is all about, and its historical review. You will also understand where you can apply digital forensics in real life and its limitations.

What is Digital Forensics?

Digital forensics may be defined as the branch of forensic science that analyzes, examines, identifies and recovers the digital evidences residing on electronic devices. It is commonly used for criminal law and private investigations.

For example, you can rely on digital forensics extract evidences in case somebody steals some data on an electronic device.

Brief Historical Review of Digital Forensics

The history of computer crimes and the historical review of digital forensics is explained in this section as given below:

1970s-1980s: First Computer Crime

Prior to this decade, no computer crime has been recognized. However, if it is supposed to happen, the then existing laws dealt with them. Later, in 1978 the first computer crime was recognized in Florida Computer Crime Act, which included legislation against unauthorized modification or deletion of data on a computer system. But over the time, due to the advancement of technology, the range of computer crimes being committed also increased. To deal with crimes related to copyright, privacy and child pornography, various other laws were passed.

1980s-1990s: Development Decade

This decade was the development decade for digital forensics, all because of the first ever investigation (1986) in which Cliff Stoll tracked the hacker named Markus Hess. During this period, two kind of digital forensics disciplines developed – first was with the help of ad-hoc tools and techniques developed by practitioners who took it as a hobby, while the second being developed by scientific community. In 1992, the term "**Computer Forensics**" was used in academic literature.

2000s-2010s: Decade of Standardization

After the development of digital forensics to a certain level, there was a need of making some specific standards that can be followed while performing investigations. Accordingly, various scientific agencies and bodies have published guidelines for digital forensics. In 2002, Scientific Working Group on Digital Evidence (SWGDE) published a paper named "Best practices for Computer Forensics”. Another feather in the cap was a European led international treaty namely **“The Convention on Cybercrime”** was signed by 43 nations and ratified by 16 nations. Even after such standards, still there is a need to resolve some issues which has been identified by researchers.

Process of Digital Forensics

Since first ever computer crime in 1978, there is a huge increment in digital criminal activities. Due to this increment, there is a need for structured manner to deal with them. In 1984, a formalized process has been introduced and after that a great number of new and improved computer forensics investigation processes have been developed.

A computer forensics investigation process involves three major phases as explained below:

Phase 1: Acquisition or Imaging of Exhibits

The first phase of digital forensics involves saving the state of the digital system so that it can be analyzed later. It is very much similar to taking photographs, blood samples etc. from a crime scene. For example, it involves capturing an image of allocated and unallocated areas of a hard disk or RAM.

Phase 2: Analysis

The input of this phase is the data acquired in the acquisition phase. Here, this data was examined to identify evidences. This phase gives three kinds of evidences as follows:

- Inculpatory evidences: These evidences support a given history.
- Exculpatory evidences: These evidences contradict a given history.
- Evidence of tampering: These evidences show that the system was tempered to avoid identification. It includes examining the files and directory content for recovering the deleted files.

Phase 3: Presentation or Reporting

As the name suggests, this phase presents the conclusion and corresponding evidences from the investigation.

Applications of Digital Forensics

Digital forensics deals with gathering, analyzing and preserving the evidences that are contained in any digital device. The use of digital forensics depends on the application. As mentioned earlier, it is used mainly in the following two applications:

Criminal Law

In criminal law, the evidence is collected to support or oppose a hypothesis in the court. Forensics procedures are very much similar to those used in criminal investigations but with different legal requirements and limitations.

Private Investigation

Mainly corporate world uses digital forensics for private investigation. It is used when companies are suspicious that employees may be performing an illegal activity on their computers that is against company policy. Digital forensics provides one of the best routes for company or person to take when investigating someone for digital misconduct.

Branches of Digital Forensics

The digital crime is not restricted to computers alone, however hackers and criminals are using small digital devices such as tablets, smart-phones etc. at a very large scale too. Some of the devices have volatile memory, while others have non-volatile memory. Hence depending upon type of devices, digital forensics has the following branches:

Computer Forensics

This branch of digital forensics deals with computers, embedded systems and static memories such as USB drives. Wide range of information from logs to actual files on drive can be investigated in computer forensics.

Mobile Forensics

This deals with investigation of data from mobile devices. This branch is different from computer forensics in the sense that mobile devices have an inbuilt communication system which is useful for providing useful information related to location.

Network Forensics

This deals with the monitoring and analysis of computer network traffic, both local and WAN(wide area network) for the purposes of information gathering, evidence collection, or intrusion detection.

Database Forensics

This branch of digital forensics deals with forensics study of databases and their metadata.

Skills Required for Digital Forensics Investigation

Digital forensics examiners help to track hackers, recover stolen data, follow computer attacks back to their source, and aid in other types of investigations involving computers. Some of the key skills required to become digital forensics examiner as discussed below:

Outstanding Thinking Capabilities

A digital forensics investigator must be an outstanding thinker and should be capable of applying different tools and methodologies on a particular assignment for obtaining the output. He/she must be able to find different patterns and make correlations among them.

Technical Skills

A digital forensics examiner must have good technological skills because this field requires the knowledge of network, how digital system interacts.

Passionate about Cyber Security

Because the field of digital forensics is all about solving cyber-crimes and this is a tedious task, it needs lot of passion for someone to become an ace digital forensic investigator.

Communication Skills

Good communication skills are a must to coordinate with various teams and to extract any missing data or information.

Skillful in Report Making

After successful implementation of acquisition and analysis, a digital forensic examiner must mention all the findings the final report and presentation. Hence he/she must have good skills of report making and an attention to detail.

Limitations

Digital forensic investigation offers certain limitations as discussed here:

Need to produce convincing evidences

One of the major setbacks of digital forensics investigation is that the examiner must have to comply with standards that are required for the evidence in the court of law, as the data can be easily tampered. On the other hand, computer forensic investigator must have complete knowledge of legal requirements, evidence handling and documentation procedures to present convincing evidences in the court of law.

Investigating Tools

The effectiveness of digital investigation entirely lies on the expertise of digital forensics examiner and the selection of proper investigation tool. If the tool used is not according to specified standards then in the court of law, the evidences can be denied by the judge.

Lack of technical knowledge among the audience

Another limitation is that some individuals are not completely familiar with computer forensics; therefore, many people do not understand this field. Investigators have to be sure to communicate their findings with the courts in such a way to help everyone understand the results**.**

Cost

Producing digital evidences and preserving them is very costly. Hence this process may not be chosen by many people who cannot afford the cost.

2. Python Digital Forensics – Getting Started with Python

In the previous chapter, we learnt the basics of digital forensics, its advantages and limitations. This chapter will make you comfortable with Python, the essential tool that we are using in this digital forensics investigation.

Why Python for Digital Forensics?

Python is a popular programming language and is used as tool for cyber security, penetration testing as well as digital forensic investigations. When you choose Python as your tool for digital forensics, you do not need any other third party software for completing the task.

Some of the unique features of Python programming language that makes it a good fit for digital forensics projects are given below:

- **Simplicity of Syntax**: Python's syntax is simple compared to other languages, that makes it easier for one to learn and put into use for digital forensics.

- **Comprehensive inbuilt modules**: Python's comprehensive inbuilt modules are an excellent aid for performing a complete digital forensic investigation.

- **Help and Support**: Being an open source programming language, Python enjoys excellent support from the developer's and users' community.

Features of Python

Python, being a high-level, interpreted, interactive and object-oriented scripting language, provides the following features:

- **Easy to Learn**: Python is a developer friendly and easy to learn language, because it has fewer keywords and simplest structure.

- **Expressive and Easy to read:** Python language is expressive in nature; hence its code is more understandable and readable.

- **Cross-platform Compatible**: Python is a cross-platform compatible language which means it can run efficiently on various platforms such as UNIX, Windows, and Macintosh.

- **Interactive Mode Programming**: We can do interactive testing and debugging of code because Python supports an interactive mode for programming.

- **Provides Various Modules and Functions**: Python has large standard library which allows us to use rich set of modules and functions for our script.

- **Supports Dynamic Type Checking**: Python supports dynamic type checking and provides very high-level dynamic data types.

- **GUI Programming**: Python supports GUI programming to develop Graphical user interfaces.

- **Integration with other programming languages**: Python can be easily integrated with other programming languages like C, C++, JAVA etc.

Installing Python

Python distribution is available for various platforms such as Windows, UNIX, Linux, and Mac. We only need to download the binary code as per our platform. In case if the binary code for any platform is not available, we must have a C compiler so that source code can be compiled manually.

This section will make you familiar with installation of Python on various platforms:

Python Installation on Unix and Linux

You can follow following the steps shown below to install Python on Unix/Linux machine.

Step1: Open a Web browser. Type and enter **https://www.python.org/downloads/**.

Step2: Download zipped source code available for Unix/Linux.

Step3: Extract the downloaded zipped files.

Step4: If you wish to customize some options, you can edit the **Modules/Setup** file.

Step5: Use the following commands for completing the installation:

```
run ./configure script

make

make install
```

Once you have successfully completed the steps given above, Python will be installed at its standard location **/usr/local/bin** and its libraries at **/usr/local/lib/pythonXX** where XX is the version of Python.

Python Installation on Windows

We can follow following simple steps to install Python on Windows machine.

Step1: Open a web browser. Type and enter **https://www.python.org/downloads/**.

Step2: Download the Windows installer **python-XYZ.msi** file, where XYZ is the version we need to install.

Step3: Now run that MSI file after saving the installer file to your local machine.

Step4: Run the downloaded file which will bring up the Python installation wizard.

Python Installation on Macintosh

For installing Python 3 on Mac OS X, we must use a package installer named **Homebrew.**

You can use the following command to install Homebrew, incase you do not have it on your system:

```
$ ruby -e "$(curl -fsSL

https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

If you need to update the package manager, then it can be done with the help of following command:

```
$ brew update
```

Now, use the following command to install Python3 on your system:

```
$ brew install python3
```

Setting the PATH

We need to set the path for Python installation and this differs with platforms such as UNIX, WINDOWS, or MAC.

Path setting at Unix/Linux

You can use the following options to set the path on Unix/Linux:

- **If using csh shell**- Type **setenv PATH "$PATH:/usr/local/bin/python"** and then press Enter.

- **If using bash shell (Linux)** – Type **export ATH="$PATH:/usr/local/bin/python"** and then press Enter.

- **If using sh or ksh shell** – Type **PATH="$PATH:/usr/local/bin/python"** and then press Enter.

Path Setting at Windows

Type **path %path%;C:\Python** at the command prompt and then press Enter.

Running Python

You can choose any of the following three methods to start the Python interpreter:

Method 1: Using Interactive Interpreter

A system that provides a command-line interpreter or shell can easily be used for starting Python. For example, Unix, DOS etc. You can follow the steps given below to start coding in interactive interpreter:

Step1: Enter **python** at the command line.

Step2: Start coding right away in the interactive interpreter using the commands shown below:

```
$python # Unix/Linux

or

python% # Unix/Linux

or

C:> python # Windows/DOS
```

Method 2: Using Script from the Command-line

We can also execute a Python script at command line by invoking the interpreter on our application. You can use commands shown below:

```
$python script.py # Unix/Linux

or

python% script.py # Unix/Linux

or

C: >python script.py # Windows/DOS
```

Method 3. Integrated Development Environment

If a system has GUI application that supports Python, then Python can be run from that GUI environment. Some of the IDE for various platforms are given below:

- **Unix IDE**: UNIX has IDLE IDE for Python.

- **Windows IDE:** Windows has PythonWin, the first Windows interface for Python along with GUI.

- **Macintosh IDE:** Macintosh has IDLE IDE which is available from the main website, downloadable as either MacBinary or BinHex'd files.

3. Python Digital Forensics – Artifact Report

Now that you are comfortable with installation and running Python commands on your local system, let us move into the concepts of forensics in detail. This chapter will explain various concepts involved in dealing with artifacts in Python digital forensics.

Need of Report Creation

The process of digital forensics includes reporting as the third phase. This is one of the most important parts of digital forensic process. Report creation is necessary due to the following reasons:

- It is the document in which digital forensic examiner outlines the investigation process and its findings.
- A good digital forensic report can be referenced by another examiner to achieve same result by given same repositories.
- It is a technical and scientific document that contains facts found within the 1s and 0s of digital evidence.

General Guidelines for Report Creation

The reports are written to provide information to the reader and must start with a solid foundation. investigators can face difficulties in efficiently presenting their findings if the report is prepared without some general guidelines or standards. Some general guidelines which must be followed while creating digital forensic reports are given below:

- **Summary:** The report must contain the brief summary of information so that the reader can ascertain the report's purpose.
- **Tools used:** We must mention the tools which have been used for carrying the process of digital forensics, including their purpose.
- **Repository:** Suppose, we investigated someone's computer then the summary of evidence and analysis of relevant material like email, internal search history etc., then they must be included in the report so that the case may be clearly presented.
- **Recommendations for counsel:** The report must have the recommendations for counsel to continue or cease investigation based on the findings in report.

Creating Different Type of Reports

In the above section, we came to know about the importance of report in digital forensics along with the guidelines for creating the same. Some of the formats in Python for creating different kind of reports are discussed below:

CSV Reports

One of the most common output formats of reports is a CSV spreadsheet report. You can create a CSV to create a report of processed data using the Python code as shown below:

First, import useful libraries for writing the spreadsheet:

```
from __future__ import print_function

import csv

import os

import sys
```

Now, call the following method:

```
Write_csv(TEST_DATA_LIST, ["Name", "Age", "City", "Job description"], os.getcwd())
```

We are using the following global variable to represent sample data types:

```
TEST_DATA_LIST = [["Ram", 32, Bhopal, Manager], ["Raman", 42, Indore,
Engg.],["Mohan", 25, Chandigarh, HR], ["Parkash", 45, Delhi, IT]]
```

Next, let us define the method to proceed for further operations. We open the file in the "**w**" mode and set the newline keyword argument to an empty string.

```
def Write_csv(data, header, output_directory, name=None):

     if name is None:

            name = "report1.csv"

     print("[+] Writing {} to {}".format(name, output_directory))

     with open(os.path.join(output_directory, name), "w", newline="") as \
csvfile:

            writer = csv.writer(csvfile)

            writer.writerow(header)

            writer.writerow(data)
```

If you run the above script, you will get the following details stored in report1.csv file.

Name	Age	City	Designation
Ram	32	Bhopal	Manager
Raman	42	Indore	Engg.
Mohan	25	Chandigarh	HR
Parkash	45	Delhi	IT

Excel Reports

Another common output format of reports is Excel (.xlsx) spreadsheet report. We can create table and also plot the graph by using Excel. We can create report of processed data in Excel format using Python code as shown below:

First, import XlsxWriter module for creating spreadsheet:

```
import xlsxwriter
```

Now, create a workbook object. For this, we need to use Workbook() constructor.

```
workbook = xlsxwriter.Workbook('report2.xlsx')
```

Now, create a new worksheet by using add_worksheet() module.

```
worksheet = workbook.add_worksheet()
```

Next, write the following data into the worksheet:

```
report2 = (['Ram', 32, ‘Bhopal’],['Mohan',25, ‘Chandigarh’] ,['Parkash',45,
‘Delhi’])


row = 0
col = 0
```

You can iterate over this data and write it as follows:

```
for item, cost in (a):
    worksheet.write(row, col, item)
    worksheet.write(row, col+1, cost)
    row +=1
```

Now, let us close this Excel file by using close() method.

```
workbook.close()
```

The above script will create an Excel file named report2.xlsx having the following data:

Ram	32	Bhopal
Mohan	25	Chandigarh
Parkash	45	Delhi

Investigation Acquisition Media

It is important for an investigator to have the detailed investigative notes to accurately recall the findings or put together all the pieces of investigation. A screenshot is very useful to keep track of the steps taken for a particular investigation. With the help of the following Python code, we can take the screenshot and save it on hard disk for future use.

First, install Python module named pyscreenshot by using following command:

```
Pip install pyscreenshot
```

Now, import the necessary modules as shown:

```
import pyscreenshot as ImageGrab
```

Use the following line of code to get the screenshot:

```
image=ImageGrab.grab()
```

Use the following line of code to save the screenshot to the given location:

```
image.save('d:/image123.png')
```

Now, if you want to pop up the screenshot as a graph, you can use the following Python code:

```
import numpy as np

import matplotlib.pyplot as plt

import pyscreenshot as ImageGrab

imageg = ImageGrab.grab()

plt.imshow(image, cmap='gray', interpolation='bilinear')

plt.show()
```

4. Python Digital Forensics – Mobile Device Forensics

This chapter will explain Python digital forensics on mobile devices and the concepts involved.

Introduction

Mobile device forensics is that branch of digital forensics which deals with the acquisition and analysis of mobile devices to recover digital evidences of investigative interest. This branch is different from computer forensics because mobile devices have an inbuilt communication system which is useful for providing useful information related to location.

Though the use of smartphones is increasing in digital forensics day-by-day, still it is considered to be non-standard due to its heterogeneity. On the other hand, computer hardware, such as hard disk, is considered to be standard and developed as a stable discipline too. In digital forensic industry, there is a lot of debate on the techniques used for non-standards devices, having transient evidences, such as smartphones.

Artifacts Extractible from Mobile Devices

Modern mobile devices possess lot of digital information in comparison with the older phones having only a call log or SMS messages. Thus, mobile devices can supply investigators with lots of insights about its user. Some artifacts that can be extracted from mobile devices are as mentioned below:

- **Messages**: These are the useful artifacts which can reveal the state of mind of the owner and can even give some previous unknown information to the investigator.

- **Location History**: The location history data is a useful artifact which can be used by investigators to validate about the particular location of a person.

- **Applications Installed**: By accessing the kind of applications installed, investigator get some insight into the habits and thinking of the mobile user.

Evidence Sources and Processing in Python

Smartphones have SQLite databases and PLIST files as the major sources of evidences. In this section we are going to process the sources of evidences in python.

Analyzing PLIST files

A PLIST (Property List) is a flexible and convenient format for storing application data especially on iPhone devices. It uses the extension **.plist**. Such kind of files used to store information about bundles and applications. It can be in two formats: **XML** and **binary**. The following Python code will open and read PLIST file. Note that before proceeding into this, we must create our own **Info.plist** file.

First, install a third party library named **biplist** by the following command:

```
Pip install biplist
```

Now, import some useful libraries to process plist files:

```
import biplist

import os

import sys
```

Now, use the following command under main method can be used to read plist file into a variable:

```
def main(plist):

     try:

       data = biplist.readPlist(plist)

     except (biplist.InvalidPlistException,biplist.NotBinaryPlistException) as e:

print("[-] Invalid PLIST file - unable to be opened by biplist")

sys.exit(1)
```

Now, we can either read the data on the console or directly print it, from this variable.

SQLite Databases

SQLite serves as the primary data repository on mobile devices. SQLite an in-process library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. It is a database, which is zero-configured, you need not configure it in your system, unlike other databases.

If you are a novice or unfamiliar with SQLite databases, you can follow the link https://www.tutorialspoint.com/sqlite/index.htm. Additionally, you can follow the link https://www.tutorialspoint.com/sqlite/sqlite_python.htm in case you want to get into detail of SQLite with Python.

During mobile forensics, we can interact with the **sms.db** file of a mobile device and can extract valuable information from **message** table. Python has a built in library named **sqlite3** for connecting with SQLite database. You can import the same with the following command:

```
 import sqlite3
```

Now, with the help of following command, we can connect with the database, say **sms.db** in case of mobile devices:

```
Conn = sqlite3.connect('sms.db')

C = conn.cursor()
```

Here, C is the cursor object with the help of which we can interact with the database.

Now, suppose if we want to execute a particular command, say to get the details from the **abc table**, it can be done with the help of following command:

```
c.execute("Select * from abc")

c.close()
```

The result of the above command would be stored in the **cursor** object. Similarly we can use **fetchall()** method to dump the result into a variable we can manipulate.

We can use the following command to get column names data of message table in **sms.db:**

```
c.execute("pragma table_info(message)")

table_data = c.fetchall()

columns = [x[1] for x in table_data
```

Observe that here we are using SQLite PRAGMA command which is special command to be used to control various environmental variables and state flags within SQLite environment. In the above command, the **fetchall()** method returns a tuple of results. Each column's name is stored in the first index of each tuple.

Now, with the help of following command we can query the table for all of its data and store it in the variable named **data_msg**:

```
c.execute("Select * from message")

data_msg = c.fetchall()
```

The above command will store the data in the variable and further we can also write the above data in CSV file by using **csv.writer()** method.

iTunes Backups

iPhone mobile forensics can be performed on the backups made by iTunes. Forensic examiners rely on analyzing the iPhone logical backups acquired through iTunes. AFC (Apple file connection) protocol is used by iTunes to take the backup. Besides, the backup process does not modify anything on the iPhone except the escrow key records.

Now, the question arises that why it is important for a digital forensic expert to understand the techniques on iTunes backups? It is important in case we get access to the suspect's computer instead of iPhone directly because when a computer is used to sync with iPhone, then most of the information on iPhone is likely to be backed up on the computer.

Process of Backup and its Location

Whenever an Apple product is backed up to the computer, it is in sync with iTunes and there will be a specific folder with device's unique ID. In the latest backup format, the files are stored in subfolders containing the first two hexadecimal characters of the file name. From these back up files, there are some files like info.plist which are useful along with the database named Manifest.db. The following table shows the backup locations, that vary with operating systems of iTunes backups:

OS	Backup Location
Win7	C:\Users\[username]\AppData\Roaming\AppleComputer\MobileSync\Backup\
MAC OS X	~/Library/Application Suport/MobileSync/Backup/

For processing the iTunes backup with Python, we need to first identify all the backups in backup location as per our operating system. Then we will iterate through each backup and read the database Manifest.db.

Now, with the help of following Python code we can do the same:

First, import the necessary libraries as follows:

```
from __future__ import print_function
import argparse
import logging
import os
from shutil import copyfile
import sqlite3
import sys
logger = logging.getLogger(__name__)
```

Now, provide two positional arguments namely INPUT_DIR and OUTPUT_DIR which is representing iTunes backup and desired output folder:

```
if __name__ == "__main__":
     parser.add_argument("INPUT_DIR",help="Location of folder containing iOS
backups, ""e.g. ~\Library\Application Support\MobileSync\Backup folder")
     parser.add_argument("OUTPUT_DIR", help="Output Directory")
     parser.add_argument("-l", help="Log file path",default=__file__[:-2] + "log")
```

```
    parser.add_argument("-v", help="Increase verbosity",action="store_true")
    args = parser.parse_args()
```

Now, setup the log as follows:

```
 if args.v:
              logger.setLevel(logging.DEBUG)
     else:
              logger.setLevel(logging.INFO)
```

Now, setup the message format for this log as follows:

```
 msg_fmt = logging.Formatter("%(asctime)-15s %(funcName)-13s""%(levelname)-8s
%(message)s")
strhndl = logging.StreamHandler(sys.stderr)
strhndl.setFormatter(fmt=msg_fmt)
fhndl = logging.FileHandler(args.l, mode='a')
fhndl.setFormatter(fmt=msg_fmt)
logger.addHandler(strhndl)
logger.addHandler(fhndl)
logger.info("Starting iBackup Visualizer")
logger.debug("Supplied arguments: {}".format(" ".join(sys.argv[1:])))
logger.debug("System: " + sys.platform)
logger.debug("Python Version: " + sys.version)
```

The following line of code will create necessary folders for the desired output directory by using **os.makedirs()** function:

```
 if not os.path.exists(args.OUTPUT_DIR):
     os.makedirs(args.OUTPUT_DIR)
```

Now, pass the supplied input and output directories to the main() function as follows:

```
if os.path.exists(args.INPUT_DIR) and os.path.isdir(args.INPUT_DIR):
     main(args.INPUT_DIR, args.OUTPUT_DIR)
else:
     logger.error("Supplied input directory does not exist or is not ""a
directory")
     sys.exit(1)
```

Now, write **main()** function which will further call **backup_summary()** function to identify all the backups present in input folder:

```
def main(in_dir, out_dir):
     backups = backup_summary(in_dir)
def backup_summary(in_dir):
     logger.info("Identifying all iOS backups in {}".format(in_dir))
     root = os.listdir(in_dir)
     backups = {}
     for x in root:
           temp_dir = os.path.join(in_dir, x)
           if os.path.isdir(temp_dir) and len(x) == 40:
                 num_files = 0
                 size = 0
                 for root, subdir, files in os.walk(temp_dir):
                       num_files += len(files)
                       size += sum(os.path.getsize(os.path.join(root, name))
                                         for name in files)
                 backups[x] = [temp_dir, num_files, size]
     return backups
```

Now, print the summary of each backup to the console as follows:

```
     print("Backup Summary")
     print("=" * 20)
     if len(backups) > 0:
           for i, b in enumerate(backups):
                 print("Backup No.: {} \n""Backup Dev. Name: {} \n""# Files: {} 
\n""Backup Size (Bytes): {}\n".format(i, b, backups[b][1], backups[b][2]))
```

Now, dump the contents of the Manifest.db file to the variable named db_items.

```
     try:
 db_items = process_manifest(backups[b][0])
     except IOError:
           logger.warn("Non-iOS 10 backup encountered or " "invalid backup. 
Continuing to next backup.")
continue
```

Now, let us define a function that will take the directory path of the backup:

```
def process_manifest(backup):
    manifest = os.path.join(backup, "Manifest.db")
    if not os.path.exists(manifest):
        logger.error("Manifest DB not found in {}".format(manifest))
        raise IOError
```

Now, using SQLite3 we will connect to the database by cursor named **c**:

```
    c = conn.cursor()
    items = {}
    for row in c.execute("SELECT * from Files;"):
        items[row[0]] = [row[2], row[1], row[3]]
    return items
create_files(in_dir, out_dir, b, db_items)
        print("=" * 20)
    else:
        logger.warning("No valid backups found. The input directory should be
" "the parent-directory immediately above the SHA-1 hash " "iOS device backups")
    sys.exit(2)
```

Now, define the **create_files()** method as follows:

```
def create_files(in_dir, out_dir, b, db_items):
    msg = "Copying Files for backup {} to {}".format(b, os.path.join(out_dir, b))
    logger.info(msg)
```

Now, iterate through each key in the **db_items** dictionary:

```
        for x, key in enumerate(db_items):
        if db_items[key][0] is None or db_items[key][0] == "":
            continue
        else:
            dirpath = os.path.join(out_dir, b,
os.path.dirname(db_items[key][0]))
            filepath = os.path.join(out_dir, b, db_items[key][0])
            if not os.path.exists(dirpath):
                os.makedirs(dirpath)
```

```
                    original_dir = b + "/" + key[0:2] + "/" + key
                    path = os.path.join(in_dir, original_dir)
                    if os.path.exists(filepath):
                    filepath = filepath + "_{}".format(x)
```

Now, use **shutil.copyfile()** method to copy the backed-up file as follows:

```
                    try:
                          copyfile(path, filepath)
                    except IOError:
                          logger.debug("File not found in backup: {}".format(path))
                          files_not_found += 1
      if files_not_found > 0:
            logger.warning("{} files listed in the Manifest.db not" "found in
backup".format(files_not_found))
      copyfile(os.path.join(in_dir, b, "Info.plist"), os.path.join(out_dir, b,
"Info.plist"))
      copyfile(os.path.join(in_dir, b, "Manifest.db"), os.path.join(out_dir, b,
"Manifest.db"))
      copyfile(os.path.join(in_dir, b, "Manifest.plist"), os.path.join(out_dir, b,
"Manifest.plist"))
      copyfile(os.path.join(in_dir, b, "Status.plist"),os.path.join(out_dir, b,
"Status.plist"))
```

With the above Python script, we can get the updated back up file structure in our output folder. We can use **pycrypto** python library to decrypt the backups.

Wi-Fi

Mobile devices can be used to connect to the outside world by connecting through Wi-Fi networks which are available everywhere. Sometimes the device gets connected to these open networks automatically.

In case of iPhone, the list of open Wi-Fi connections with which the device has got connected is stored in a PLIST file named **com.apple.wifi.plist**. This file will contain the Wi-Fi SSID, BSSID and connection time.

We need to extract Wi-Fi details from standard Cellebrite XML report using Python. For this, we need to use API from Wireless Geographic Logging Engine (WIGLE), a popular platform which can be used for finding the location of a device using the names of Wi-Fi networks.

We can use Python library named **requests** to access the API from WIGLE. It can be installed as follows:

```
pip install requests
```

API from WIGLE

We need to register on WIGLE's website https://wigle.net/account to get a free API from WIGLE. The Python script for getting the information about user device and its connection through WIGEL's API is discussed below:

First, import the following libraries for handling different things:

```
from __future__ import print_function

import argparse

import csv

import os

import sys

import xml.etree.ElementTree as ET

import requests
```

Now, provide two positional arguments namely **INPUT_FILE** and **OUTPUT_CSV** which will represent the input file with Wi-Fi MAC address and the desired output CSV file respectively:

```
if __name__ == "__main__":

     parser.add_argument("INPUT_FILE", help="INPUT FILE with MAC Addresses")

     parser.add_argument("OUTPUT_CSV", help="Output CSV File")

     parser.add_argument("-t", help="Input type: Cellebrite XML report or TXT
file",choices=('xml', 'txt'), default="xml")

     parser.add_argument('--api', help="Path to API key
     file",default=os.path.expanduser("~/.wigle_api"),

     type=argparse.FileType('r'))

     args = parser.parse_args()
```

Now following lines of code will check if the input file exists and is a file. If not, it exits the script:

```
if not os.path.exists(args.INPUT_FILE) or \ not os.path.isfile(args.INPUT_FILE):

             print("[-] {} does not exist or is not a
file".format(args.INPUT_FILE))

             sys.exit(1)

directory = os.path.dirname(args.OUTPUT_CSV)

if directory != '' and not os.path.exists(directory):

     os.makedirs(directory)
```

```
api_key = args.api.readline().strip().split(":")
```

Now, pass the argument to main as follows:

```
main(args.INPUT_FILE, args.OUTPUT_CSV, args.t, api_key)
def main(in_file, out_csv, type, api_key):
    if type == 'xml':
        wifi = parse_xml(in_file)
    else:
        wifi = parse_txt(in_file)
    query_wigle(wifi, out_csv, api_key)
```

Now, we will parse the XML file as follows:

```
def parse_xml(xml_file):
    wifi = {}
    xmlns = "{http://pa.cellebrite.com/report/2.0}"
    print("[+] Opening {} report".format(xml_file))
    xml_tree = ET.parse(xml_file)
    print("[+] Parsing report for all connected WiFi addresses")
    root = xml_tree.getroot()
```

Now, iterate through the child element of the root as follows:

```
    for child in root.iter():
        if child.tag == xmlns + "model":
            if child.get("type") == "Location":
                for field in child.findall(xmlns + "field"):
                    if field.get("name") == "TimeStamp":
                        ts_value = field.find(xmlns + "value")
                        try:
                        ts = ts_value.text
                        except AttributeError:
    continue
```

Now, we will check that 'ssid' string is present in the value's text or not:

```
            if "SSID" in value.text:
                bssid, ssid = value.text.split("\t")
```

```
                    bssid = bssid[7:]
                    ssid = ssid[6:]
```

Now, we need to add BSSID, SSID and timestamp to the wifi dictionary as follows:

```
               if bssid in wifi.keys():

               wifi[bssid]["Timestamps"].append(ts)
                    wifi[bssid]["SSID"].append(ssid)
               else:
                    wifi[bssid] = {"Timestamps": [ts], "SSID":
[ssid],"Wigle": {}}
return wifi
```

The text parser which is much simpler that XML parser is shown below:

```
def parse_txt(txt_file):
     wifi = {}
     print("[+] Extracting MAC addresses from {}".format(txt_file))
     with open(txt_file) as mac_file:
          for line in mac_file:
               wifi[line.strip()] = {"Timestamps": ["N/A"], "SSID":
["N/A"],"Wigle": {}}
return wifi
```

Now, let us use requests module to make **WIGLE API** calls and need to move on to the **query_wigle()** method:

```
def query_wigle(wifi_dictionary, out_csv, api_key):
     print("[+] Querying Wigle.net through Python API for {} "
"APs".format(len(wifi_dictionary)))
     for mac in wifi_dictionary:

          wigle_results = query_mac_addr(mac, api_key)
def query_mac_addr(mac_addr, api_key):

     query_url = "https://api.wigle.net/api/v2/network/search?" \
"onlymine=false&freenet=false&paynet=false" \ "&netid={}".format(mac_addr)
     req = requests.get(query_url, auth=(api_key[0], api_key[1]))
```

```
    return req.json()
```

Actually there is a limit per day for WIGLE API calls, if that limit exceeds then it must show an error as follows:

```
    try:
        if wigle_results["resultCount"] == 0:
            wifi_dictionary[mac]["Wigle"]["results"] = []
            continue
        else:
            wifi_dictionary[mac]["Wigle"] = wigle_results
    except KeyError:
        if wigle_results["error"] == "too many queries today":
        print("[-] Wigle daily query limit exceeded")
        wifi_dictionary[mac]["Wigle"]["results"] = []
        continue
    else:
        print("[-] Other error encountered for " "address {}: {}".format(mac,
wigle_results['error']))
    wifi_dictionary[mac]["Wigle"]["results"] = []
    continue
prep_output(out_csv, wifi_dictionary)
```

Now, we will use **prep_output()** method to flattens the dictionary into easily writable chunks:

```
def prep_output(output, data):
    csv_data = {}
    google_map = https://www.google.com/maps/search/
```

Now, access all the data we have collected so far as follows:

```
for x, mac in enumerate(data):
        for y, ts in enumerate(data[mac]["Timestamps"]):
            for z, result in enumerate(data[mac]["Wigle"]["results"]):
                shortres = data[mac]["Wigle"]["results"][z]
```

```
                          g_map_url = "{}{},{}".format(
                                 google_map, shortres["trilat"],
shortres["trilong"])
```

Now, we can write the output in CSV file as we have done in earlier scripts in this chapter by using **write_csv()** function.

5. Python Digital Forensics – Investigating Embedded Metadata

In this chapter, we will learn in detail about investigating embedded metadata using Python digital forensics.

Introduction

Embedded metadata is the information about data stored in the same file which is having the object described by that data. In other words, it is the information about a digital asset stored in the digital file itself. It is always associated with the file and can never be separated.

In case of digital forensics, we cannot extract all the information about a particular file. On the other side, embedded metadata can provide us information critical to the investigation. For example, a text file's metadata may contain information about the author, its length, written date and even a short summary about that document. A digital image may include the metadata such as the length of the image, the shutter speed etc.

Artifacts Containing Metadata Attributes and their Extraction

In this section, we will learn about various artifacts containing metadata attributes and their extraction process using Python.

Audio and Video

These are the two very common artifacts which have the embedded metadata. This metadata can be extracted for the purpose of investigation.

You can use the following Python script to extract common attributes or metadata from audio or MP3 file and a video or a MP4 file.

Note that for this script, we need to install a third party python library named **mutagen** which allows us to extract metadata from audio and video files. It can be installed with the help of the following command:

```
pip install mutagen
```

Some of the useful libraries we need to import for this Python script are as follows:

```
from __future__ import print_function
import argparse
import json
import mutagen
```

The command line handler will take one argument which represents the path to the MP3 or MP4 files. Then, we will use **mutagen.file()** method to open a handle to the file as follows:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser('Python Metadata Extractor')
    parser.add_argument("AV_FILE", help="File to extract metadata from")
    args = parser.parse_args()
    av_file = mutagen.File(args.AV_FILE)
    file_ext = args.AV_FILE.rsplit('.', 1)[-1]
    if file_ext.lower() == 'mp3':
        handle_id3(av_file)
    elif file_ext.lower() == 'mp4':
        handle_mp4(av_file)
```

Now, we need to use two handles, one to extract the data from MP3 and one to extract data from MP4 file. We can define these handles as follows:

```
def handle_id3(id3_file):
    id3_frames = {'TIT2': 'Title', 'TPE1': 'Artist', 'TALB': 'Album','TXXX':
'Custom', 'TCON': 'Content Type', 'TDRL': 'Date released','COMM': 'Comments',
'TDRC': 'Recording Date'}
    print("{:15} | {:15} | {:38} | {}".format("Frame", "Description","Text",
"Value"))
    print("-" * 85)
    for frames in id3_file.tags.values():
        frame_name = id3_frames.get(frames.FrameID, frames.FrameID)
        desc = getattr(frames, 'desc', "N/A")
        text = getattr(frames, 'text', ["N/A"])[0]
        value = getattr(frames, 'value', "N/A")
        if "date" in frame_name.lower():
            text = str(text)
        print("{:15} | {:15} | {:38} | {}".format(
            frame_name, desc, text, value))
def handle_mp4(mp4_file):
    cp_sym = u"\u00A9"
    qt_tag = {
        cp_sym + 'nam': 'Title', cp_sym + 'art': 'Artist',
        cp_sym + 'alb': 'Album', cp_sym + 'gen': 'Genre',
            'cpil': 'Compilation', cp_sym + 'day': 'Creation Date',
            'cnID': 'Apple Store Content ID', 'atID': 'Album Title ID',
            'plID': 'Playlist ID', 'geID': 'Genre ID', 'pcst': 'Podcast',
            'purl': 'Podcast URL', 'egid': 'Episode Global ID',
            'cmID': 'Camera ID', 'sfID': 'Apple Store Country',
            'desc': 'Description', 'ldes': 'Long Description'}
    genre_ids = json.load(open('apple_genres.json'))
```

Now, we need to iterate through this MP4 file as follows:

```
    print("{:22} | {}".format('Name', 'Value'))
    print("-" * 40)
    for name, value in mp4_file.tags.items():
            tag_name = qt_tag.get(name, name)
            if isinstance(value, list):
                    value = "; ".join([str(x) for x in value])
            if name == 'geID':
                    value = "{}: {}".format(
                            value, genre_ids[str(value)].replace("|", " - "))
            print("{:22} | {}".format(tag_name, value))
```

The above script will give us additional information about MP3 as well as MP4 files.

Images

Images may contain different kind of metadata depending upon its file format. However, most of the images embed GPS information. We can extract this GPS information by using third party Python libraries. You can use the following Python script can be used to do the same:

First, download third party python library named **Python Imaging Library (PIL)** as follows:

```
pip install pillow
```

This will help us to extract metadata from images.

We can also write the GPS details embedded in images to KML file, but for this we need to download third party Python library named **simplekml** as follows:

```
pip install simplekml
```

In this script, first we need to import the following libraries:

```
from __future__ import print_function
import argparse
from PIL import Image
from PIL.ExifTags import TAGS
import simplekml
import sys
```

Now, the command line handler will accept one positional argument which basically represents the file path of the photos.

```
parser = argparse.ArgumentParser('Metadata from images')
parser.add_argument('PICTURE_FILE', help="Path to picture")
args = parser.parse_args()
```

Now, we need to specify the URLs that will populate the coordinate information. The URLs are **gmaps** and **open_maps**. We also need a function to convert the degree minute seconds (DMS) tuple coordinate, provided by PIL library, into decimal. It can be done as follows:

```
 gmaps = "https://www.google.com/maps?q={},{}"
open_maps = "http://www.openstreetmap.org/?mlat={}&mlon={}"
def process_coords(coord):
     coord_deg = 0
     for count, values in enumerate(coord):
            coord_deg += (float(values[0]) / values[1]) / 60**count
     return coord_deg
```

Now, we will use **image.open()** function to open the file as PIL object.

```
img_file = Image.open(args.PICTURE_FILE)
exif_data = img_file._getexif()
if exif_data is None:
     print("No EXIF data found")
     sys.exit()
for name, value in exif_data.items():
     gps_tag = TAGS.get(name, name)
     if gps_tag is not 'GPSInfo':
            continue
```

After finding the **GPSInfo** tag, we will store the GPS reference and process the coordinates with the **process_coords()** method.

```
    lat_ref = value[1] == u'N'
    lat = process_coords(value[2])
    if not lat_ref:
        lat = lat * -1
    lon_ref = value[3] == u'E'
    lon = process_coords(value[4])
    if not lon_ref:
        lon = lon * -1
```

Now, initiate **kml** object from **simplekml** library as follows:

```
    kml = simplekml.Kml()
    kml.newpoint(name=args.PICTURE_FILE, coords=[(lon, lat)])
    kml.save(args.PICTURE_FILE + ".kml")
```

We can now print the coordinates from processed information as follows:

```
    print("GPS Coordinates: {}, {}".format(lat, lon))
    print("Google Maps URL: {}".format(gmaps.format(lat, lon)))
    print("OpenStreetMap URL: {}".format(open_maps.format(lat, lon)))
    print("KML File {} created".format(args.PICTURE_FILE + ".kml"))
```

PDF Documents

PDF documents have a wide variety of media including images, text, forms etc. When we extract embedded metadata in PDF documents, we may get the resultant data in the format called Extensible Metadata Platform (XMP). We can extract metadata with the help of the following Python code:

First, install a third party Python library named **PyPDF2** to read metadata stored in XMP format. It can be installed as follows:

```
pip install PyPDF2
```

Now, import the following libraries for extracting the metadata from PDF files:

```
from __future__ import print_function
from argparse import ArgumentParser, FileType
import datetime
from PyPDF2 import PdfFileReader
import sys
```

Now, the command line handler will accept one positional argument which basically represents the file path of the PDF file.

```
parser = argparse.ArgumentParser('Metadata from PDF')
parser.add_argument('PDF_FILE', help='Path to PDF file',type=FileType('rb'))
args = parser.parse_args()
```

Now we can use **getXmpMetadata()** method to provide an object containing the available metadata as follows:

```
pdf_file = PdfFileReader(args.PDF_FILE)
xmpm = pdf_file.getXmpMetadata()
if xmpm is None:
     print("No XMP metadata found in document.")
     sys.exit()
```

We can use **custom_print()** method to extract and print the relevant values like title, creator, contributor etc. as follows:

```
custom_print("Title: {}", xmpm.dc_title)
custom_print("Creator(s): {}", xmpm.dc_creator)
custom_print("Contributors: {}", xmpm.dc_contributor)
custom_print("Subject: {}", xmpm.dc_subject)
custom_print("Description: {}", xmpm.dc_description)
custom_print("Created: {}", xmpm.xmp_createDate)
custom_print("Modified: {}", xmpm.xmp_modifyDate)
custom_print("Event Dates: {}", xmpm.dc_date)
```

We can also define **custom_print()** method in case if PDF is created using multiple software as follows:

```
def custom_print(fmt_str, value):
    if isinstance(value, list):
        print(fmt_str.format(", ".join(value)))
    elif isinstance(value, dict):
        fmt_value = [":".join((k, v)) for k, v in value.items()]
        print(fmt_str.format(", ".join(value)))
    elif isinstance(value, str) or isinstance(value, bool):
        print(fmt_str.format(value))
    elif isinstance(value, bytes):
        print(fmt_str.format(value.decode()))
    elif isinstance(value, datetime.datetime):
        print(fmt_str.format(value.isoformat()))
    elif value is None:
        print(fmt_str.format("N/A"))
    else:
        print("warn: unhandled type {} found".format(type(value)))
```

We can also extract any other custom property saved by the software as follows:

```
if xmpm.custom_properties:
    print("Custom Properties:")
    for k, v in xmpm.custom_properties.items():
        print("\t{}: {}".format(k, v))
```

The above script will read the PDF document and will print the metadata stored in XMP format including some custom properties stored by the software with the help of which that PDF has been made.

Windows Executables Files

Sometimes we may encounter a suspicious or unauthorized executable file. But for the purpose of investigation it may be useful because of the embedded metadata. We can get the information such as its location, its purpose and other attributes such as the manufacturer, compilation date etc. With the help of following Python script we can get the compilation date, useful data from headers and imported as well as exported symbols.

For this purpose, first install the third party Python library **pefile**. It can be done as follows:

```
pip install pefile
```

Once you successfully install this, import the following libraries as follows:

```
from __future__ import print_function
import argparse
from datetime import datetime
from pefile import PE
```

Now, the command line handler will accept one positional argument which basically represents the file path of the executable file. You can also choose the style of output, whether you need it in detailed and verbose way or in a simplified manner. For this you need to give an optional argument as shown below:

```
parser = argparse.ArgumentParser('Metadata from executable file')
parser.add_argument("EXE_FILE", help="Path to exe file")
parser.add_argument("-v", "--verbose", help="Increase verbosity of output",
action='store_true', default=False)
args = parser.parse_args()
```

Now, we will load the input executable file by using **PE** class. We will also dump the executable data to a dictionary object by using **dump_dict()** method.

```
pe = PE(args.EXE_FILE)
ped = pe.dump_dict()
```

We can extract basic file metadata such as embedded authorship, version and compilation time using the code shown below:

```
 file_info = {}
for structure in pe.FileInfo:
     if structure.Key == b'StringFileInfo':
            for s_table in structure.StringTable:
                   for key, value in s_table.entries.items():
                          if value is None or len(value) == 0:
                                 value = "Unknown"
                          file_info[key] = value
print("File Information: ")
print("==================")
for k, v in file_info.items():
    if isinstance(k, bytes):
        k = k.decode()
    if isinstance(v, bytes):
        v = v.decode()
    print("{}: {}".format(k, v))
comp_time = ped['FILE_HEADER']['TimeDateStamp']['Value']
comp_time = comp_time.split("[")[-1].strip("]")
time_stamp, timezone = comp_time.rsplit(" ", 1)
comp_time = datetime.strptime(time_stamp, "%a %b %d %H:%M:%S %Y")
print("Compiled on {} {}".format(comp_time, timezone.strip()))
```

We can extract the useful data from headers as follows:

```
for section in ped['PE Sections']:
    print("Section '{}' at {}: {}/{} {}".format(
        section['Name']['Value'], hex(section['VirtualAddress']['Value']),
        section['Misc_VirtualSize']['Value'],
        section['SizeOfRawData']['Value'], section['MD5'])
    )
```

Now, extract the listing of imports and exports from executable files as shown below:

```
if hasattr(pe, 'DIRECTORY_ENTRY_IMPORT'):
    print("\nImports: ")
    print("=========")
    for dir_entry in pe.DIRECTORY_ENTRY_IMPORT:
        dll = dir_entry.dll
        if not args.verbose:
            print(dll.decode(), end=", ")
            continue
        name_list = []
        for impts in dir_entry.imports:
            if getattr(impts, "name", b"Unknown") is None:
                name = b"Unknown"
            else:
                name = getattr(impts, "name", b"Unknown")
            name_list.append([name.decode(), hex(impts.address)])
        name_fmt = ["{} ({})".format(x[0], x[1]) for x in name_list]
        print('- {}: {}'.format(dll.decode(), ", ".join(name_fmt)))
    if not args.verbose:
        print()
```

Now, print **exports**, **names** and **addresses** using the code as shown below:

```
  if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
      print("\nExports: ")
      print("=========")
      for sym in pe.DIRECTORY_ENTRY_EXPORT.symbols:
          print('- {}: {}'.format(sym.name.decode(), hex(sym.address)))
```

The above script will extract the basic metadata, information from headers from windows executable files.

Office Document Metadata

Most of the work in computer is done in three applications of MS Office – Word, PowerPoint and Excel. These files possess huge metadata, which can expose interesting information about their authorship and history.

Note that metadata from 2007 format of word (.docx), excel (.xlsx) and powerpoint (.pptx) is stored in a XML file. We can process these XML files in Python with the help of following Python script shown below:

First, import the required libraries as shown below:

```
from __future__ import print_function
from argparse import ArgumentParser
from datetime import datetime as dt
from xml.etree import ElementTree as etree
import zipfile
parser = argparse.ArgumentParser('Office Document Metadata’)
parser.add_argument("Office_File", help="Path to office file to read")
args = parser.parse_args()
```

Now, check if the file is a ZIP file. Else, raise an error. Now, open the file and extract the key elements for processing using the following code:

```
zipfile.is_zipfile(args.Office_File)
zfile = zipfile.ZipFile(args.Office_File)
core_xml = etree.fromstring(zfile.read('docProps/core.xml'))
app_xml = etree.fromstring(zfile.read('docProps/app.xml'))
```

Now, create a dictionary for initiating the extraction of the metadata:

```
core_mapping = {
     'title': 'Title',
     'subject': 'Subject',
     'creator': 'Author(s)',
     'keywords': 'Keywords',
     'description': 'Description',
     'lastModifiedBy': 'Last Modified By',
     'modified': 'Modified Date',
     'created': 'Created Date',
     'category': 'Category',
     'contentStatus': 'Status',
     'revision': 'Revision'
}
```

Use **iterchildren()** method to access each of the tags within the XML file:

```
for element in core_xml.getchildren():
     for key, title in core_mapping.items():
          if key in element.tag:
               if 'date' in title.lower():
                    text = dt.strptime(element.text, "%Y-%m-%dT%H:%M:%SZ")
               else:
                    text = element.text
               print("{}: {}".format(title, text))
```

Similarly, do this for app.xml file which contains statistical information about the contents of the document:

```
app_mapping = {
      'TotalTime': 'Edit Time (minutes)',
      'Pages': 'Page Count',
      'Words': 'Word Count',
      'Characters': 'Character Count',
      'Lines': 'Line Count',
      'Paragraphs': 'Paragraph Count',
      'Company': 'Company',
      'HyperlinkBase': 'Hyperlink Base',
      'Slides': 'Slide count',
      'Notes': 'Note Count',
      'HiddenSlides': 'Hidden Slide Count',
}
for element in app_xml.getchildren():
      for key, title in app_mapping.items():
            if key in element.tag:
                  if 'date' in title.lower():
                        text = dt.strptime(element.text, "%Y-%m-%dT%H:%M:%SZ")
                  else:
                        text = element.text
                  print("{}: {}".format(title, text))
```

Now after running the above script, we can get the different details about the particular document. Note that we can apply this script on Office 2007 or later version documents only.

6. Python Digital Forensics – Network Forensics-I

This chapter will explain the fundamentals involved in performing network forensics using Python.

Understanding Network Forensics

Network forensics is a branch of digital forensics that deals with the monitoring and analysis of computer network traffic, both local and WAN(wide area network), for the purposes of information gathering, evidence collection, or intrusion detection. Network forensics play a critical role in investigating digital crimes such as theft of intellectual property or leakage of information. A picture of network communications helps an investigator to solve some crucial questions as follows:

- What websites has been accessed?
- What kind of content has been uploaded on our network?
- What kind of content has been downloaded from our network?
- What servers are being accessed?
- Is somebody sending sensitive information outside of company firewalls?

Internet Evidence Finder (IEF)

IEF is a digital forensic tool to find, analyze and present digital evidence found on different digital media like computer, smartphones, tablets etc. It is very popular and used by thousands of forensics professionals.

Use of IEF

Due to its popularity, IEF is used by forensics professionals to a great extent. Some of the uses of IEF are as follows:

- Due to its powerful search capabilities, it is used to search multiple files or data media simultaneously.
- It is also used to recover deleted data from the unallocated space of RAM through new carving techniques.
- If investigators want to rebuild web pages in their original format on the date they were opened, then they can use IEF.
- It is also used to search logical or physical disk volumes.

Dumping Reports from IEF to CSV using Python

IEF stores data in a SQLite database and following Python script will dynamically identify result tables within the IEF database and dump them to respective CSV files.

This process is done in the steps shown below:

- First, generate IEF result database which will be a SQLite database file ending with .db extension.
- Then, query that database to identify all the tables.
- Lastly, write this result tables to an individual CSV file.

Python Code

Let us see how to use Python code for this purpose:

For Python script, import the necessary libraries as follows:

```
from __future__ import print_function
import argparse
import csv
import os
import sqlite3
import sys
```

Now, we need to provide the path to IEF database file:

```
if __name__ == '__main__':
     parser = argparse.ArgumentParser('IEF to CSV')
     parser.add_argument("IEF_DATABASE", help="Input IEF database")
     parser.add_argument("OUTPUT_DIR", help="Output DIR")
     args = parser.parse_args()
```

Now, we will confirm the existence of IEF database as follows:

```
     if not os.path.exists(args.OUTPUT_DIR):
            os.makedirs(args.OUTPUT_DIR)
     if os.path.exists(args.IEF_DATABASE) and \
            os.path.isfile(args.IEF_DATABASE):
            main(args.IEF_DATABASE, args.OUTPUT_DIR)
     else:
            print("[-] Supplied input file {} does not exist or is not a "
"file".format(args.IEF_DATABASE))

            sys.exit(1)
```

Now, as we did in earlier scripts, make the connection with SQLite database as follows to execute the queries through cursor:

```
def main(database, out_directory):

     print("[+] Connecting to SQLite database")

     conn = sqlite3.connect(database)

     c = conn.cursor()
```

The following lines of code will fetch the names of the tables from the database:

```
     print("List of all tables to extract")

     c.execute("select * from sqlite_master where type='table'")

     tables = [x[2] for x in c.fetchall() if not x[2].startswith('_') and not
x[2].endswith('_DATA')]
```

Now, we will select all the data from the table and by using **fetchall()** method on the cursor object we will store the list of tuples containing the table's data in its entirety in a variable:

```
            print("Dumping {} tables to CSV files in {}".format(len(tables),
out_directory))

     for table in tables:

            c.execute("pragma table_info('{}')".format(table))

            table_columns = [x[1] for x in c.fetchall()]

            c.execute("select * from '{}'".format(table))

            table_data = c.fetchall()
```

Now, by using **CSV_Writer()** method we will write the content in CSV file:

```
     csv_name = table + '.csv'

     csv_path = os.path.join(out_directory, csv_name)

     print('[+] Writing {} table to {} CSV file'.format(table,csv_name))

     with open(csv_path, "w", newline="") as csvfile:

            csv_writer = csv.writer(csvfile)

            csv_writer.writerow(table_columns)

            csv_writer.writerows(table_data)
```

The above script will fetch all the data from tables of IEF database and write the contents to the CSV file of our choice.

Working with Cached Data

From IEF result database, we can fetch more information that is not necessarily supported by IEF itself. We can fetch the cached data, a bi product for information, from email service provider like Yahoo, Google etc. by using IEF result database.

The following is the Python script for accessing the cached data information from Yahoo mail, accessed on Google Chrome, by using IEF database. Note that the steps would be more or less same as followed in the last Python script.

First, import the necessary libraries for Python as follows:

```
from __future__ import print_function
import argparse
import csv
import os
import sqlite3
import sys
import json
```

Now, provide the path to IEF database file along with two positional arguments accepts by command-line handler as done in the last script:

```
 if __name__ == '__main__':
      parser = argparse.ArgumentParser('IEF to CSV')
      parser.add_argument("IEF_DATABASE", help="Input IEF database")
      parser.add_argument("OUTPUT_DIR", help="Output DIR")
      args = parser.parse_args()
```

Now, confirm the existence of IEF database as follows:

```
     directory = os.path.dirname(args.OUTPUT_CSV)
     if not os.path.exists(directory):os.makedirs(directory)
     if os.path.exists(args.IEF_DATABASE) and \ os.path.isfile(args.IEF_DATABASE):
             main(args.IEF_DATABASE, args.OUTPUT_CSV)
else:              print("Supplied input file {} does not exist or is not a "
"file".format(args.IEF_DATABASE))
             sys.exit(1)
```

Now, make the connection with SQLite database as follows to execute the queries through cursor:

```
 def main(database, out_csv):
     print("[+] Connecting to SQLite database")
     conn = sqlite3.connect(database)
     c = conn.cursor()
```

You can use the following lines of code to fetch the instances of Yahoo Mail contact cache record:

```
     print("Querying IEF database for Yahoo Contact Fragments from " "the Chrome
Cache Records Table")
     try:
             c.execute("select * from 'Chrome Cache Records' where URL like "
"'https://data.mail.yahoo.com" "/classicab/v2/contacts/?format=json%'")
     except sqlite3.OperationalError:
             print("Received an error querying the database -- database may be"
"corrupt or not have a Chrome Cache Records table")
             sys.exit(2)
```

Now, the list of tuples returned from above query to be saved into a variable as follows:

```
             contact_cache = c.fetchall()
             contact_data = process_contacts(contact_cache)
             write_csv(contact_data, out_csv)
```

Note that here we will use two methods namely **process_contacts()** for setting up the result list as well as iterating through each contact cache record and **json.loads()** to store the JSON data extracted from the table into a variable for further manipulation:

```
def process_contacts(contact_cache):
     print("[+] Processing {} cache files matching Yahoo contact cache " "
data".format(len(contact_cache)))
     results = []
     for contact in contact_cache:
             url = contact[0]
             first_visit = contact[1]
             last_visit = contact[2]
             last_sync = contact[3]
             loc = contact[8]
```

```
            contact_json = json.loads(contact[7].decode())
            total_contacts = contact_json["total"]
            total_count = contact_json["count"]
            if "contacts" not in contact_json:
                  continue
            for c in contact_json["contacts"]:
              name, anni, bday, emails, phones, links = ("", "", "", "", "", "")
                  if "name" in c:
                    name = c["name"]["givenName"] + " " + \
c["name"]["middleName"] + " " + c["name"]["familyName"]
                  if "anniversary" in c:
                    anni = c["anniversary"]["month"] + \"/" +
c["anniversary"]["day"] + "/" + \c["anniversary"]["year"]
                  if "birthday" in c:
                    bday = c["birthday"]["month"] + "/" + \c["birthday"]["day"] +
"/" + c["birthday"]["year"]
                  if "emails" in c:
                    emails = ', '.join([x["ep"] for x in c["emails"]])
                  if "phones" in c:
                    phones = ', '.join([x["ep"] for x in c["phones"]])
                  if "links" in c:
                    links = ', '.join([x["ep"] for x in c["links"]])
```

Now for company, title and notes, the **get** method is used as shown below:

```
                  company = c.get("company", "")
                  title = c.get("jobTitle", "")
                  notes = c.get("notes", "")
```

Now, let us append the list of metadata and extracted data elements to the **`result`** list as follows:

```
                  results.append([url, first_visit, last_visit, last_sync, loc,
name, bday,anni, emails, phones, links, company, title, notes,total_contacts,
total_count])
      return results
```

Now, by using **CSV_Writer()** method, we will write the content in CSV file:

```
def write_csv(data, output):
     print("[+] Writing {} contacts to {}".format(len(data), output))
     with open(output, "w", newline="") as csvfile:
            csv_writer = csv.writer(csvfile)
            csv_writer.writerow([
                   "URL", "First Visit (UTC)", "Last Visit (UTC)",
                   "Last Sync (UTC)", "Location", "Contact Name", "Bday",
                   "Anniversary", "Emails", "Phones", "Links", "Company", "Title",
                   "Notes", "Total Contacts", "Count of Contacts in Cache"])
            csv_writer.writerows(data)
```

With the help of above script, we can process the cached data from Yahoo mail by using IEF database.

7. Python Digital Forensics – Network Forensics-II

The previous chapter dealt with some of the concepts of network forensics using Python. In this chapter, let us understand network forensics using Python at a deeper level.

Web Page Preservation with Beautiful Soup

The World Wide Web (WWW) is a unique resource of information. However, its legacy is at high risk due to the loss of content at an alarming rate. A number of cultural heritage and academic institutions, non-profit organizations and private businesses have explored the issues involved and contributed to the development of technical solutions for web archiving.

Web page preservation or web archiving is the process of gathering the data from World Wide Web, ensuring that the data is preserved in an archive and making it available for future researchers, historians and the public. Before proceeding further into the web page preservation, let us discuss some important issues related to web page preservation as given below:

- **Change in Web Resources**: Web resources keep changing everyday which is a challenge for web page preservation.

- **Large Quantity of Resources**: Another issue related to web page preservation is the large quantity of resources which is to be preserved.

- **Integrity**: Web pages must be protected from unauthorized amendments, deletion or removal to protect its integrity.

- **Dealing with multimedia data:** While preserving web pages we need to deal with multimedia data also, and these might cause issues while doing so.

- **Providing access:** Besides preserving, the issue of providing access to web resources and dealing with issues of ownership needs to be solved too.

In this chapter, we are going to use Python library named **Beautiful Soup** for web page preservation.

What is Beautiful Soup?

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It can be used with **urlib** because it needs an input (document or url) to create a soup object, as it cannot fetch web page itself. You can learn in detail about this at https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Note that before using it, we must install a third party library using the following command:

```
pip install bs4
```

Next, using Anaconda package manager, we can install Beautiful Soup as follows:

```
conda install -c anaconda beautifulsoup4
```

Python Script for Preserving Web Pages

The Python script for preserving web pages by using third party library called Beautiful Soup is discussed here:

First, import the required libraries as follows:

```
from __future__ import print_function
import argparse
from bs4 import BeautifulSoup, SoupStrainer
from datetime import datetime
import hashlib
import logging
import os
import ssl
import sys
from urllib.request import urlopen
import urllib.error
logger = logging.getLogger(__name__)
```

Note that this script will take two positional arguments, one is URL which is to be preserved and other is the desired output directory as shown below:

```
if __name__ == "__main__":
     parser = argparse.ArgumentParser('Web Page preservation')
     parser.add_argument("DOMAIN", help="Website Domain")
     parser.add_argument("OUTPUT_DIR", help="Preservation Output Directory")
     parser.add_argument("-l", help="Log file path",
     default=__file__[:-3] + ".log")
     args = parser.parse_args()
```

Now, setup the logging for the script by specifying a file and stream handler for being in loop and document the acquisition process as shown:

```
    logger.setLevel(logging.DEBUG)
    msg_fmt = logging.Formatter("%(asctime)-15s %(funcName)-10s""%(levelname)-8s
%(message)s")
    strhndl = logging.StreamHandler(sys.stderr)
    strhndl.setFormatter(fmt=msg_fmt)
    fhndl = logging.FileHandler(args.l, mode='a')
    fhndl.setFormatter(fmt=msg_fmt)
    logger.addHandler(strhndl)
    logger.addHandler(fhndl)
    logger.info("Starting BS Preservation")
    logger.debug("Supplied arguments: {}".format(sys.argv[1:]))
    logger.debug("System " + sys.platform)
    logger.debug("Version " + sys.version)
```

Now, let us do the input validation on the desired output directory as follows:

```
    if not os.path.exists(args.OUTPUT_DIR):
        os.makedirs(args.OUTPUT_DIR)
    main(args.DOMAIN, args.OUTPUT_DIR)
```

Now, we will define the **main()** function which will extract the base name of the website by removing the unnecessary elements before the actual name along with additional validation on the input URL as follows:

```
def main(website, output_dir):
    base_name = website.replace("https://", "").replace("http://",
"").replace("www.", "")
    link_queue = set()
    if "http://" not in website and "https://" not in website:
        logger.error("Exiting preservation - invalid user input:
{}".format(website))
        sys.exit(1)
    logger.info("Accessing {} webpage".format(website))
    context = ssl._create_unverified_context()
```

Now, we need to open a connection with the URL by using **urlopen()** method. Let us use try-except block as follows:

```
    try:
        index = urlopen(website, context=context).read().decode("utf-8")
    except urllib.error.HTTPError as e:
        logger.error("Exiting preservation - unable to access page:
{}".format(website))
        sys.exit(2)
    logger.debug("Successfully accessed {}".format(website))
```

The next lines of code include three function as explained below:

- **write_output()** to write the first web page to the output directory
- **find_links()** function to identify the links on this web page
- **recurse_pages()** function to iterate through and discover all links on the web page.

```
    write_output(website, index, output_dir)
    link_queue = find_links(base_name, index, link_queue)
    logger.info("Found {} initial links on webpage".format(len(link_queue)))
    recurse_pages(website, link_queue, context, output_dir)
    logger.info("Completed preservation of {}".format(website))
```

Now, let us define **write_output()** method as follows:

```
def write_output(name, data, output_dir, counter=0):
    name = name.replace("http://", "").replace("https://", "").rstrip("//")
    directory = os.path.join(output_dir, os.path.dirname(name))
    if not os.path.exists(directory) and os.path.dirname(name) != "":
        os.makedirs(directory)
```

We need to log some details about the web page and then we log the hash of the data by using **hash_data()** method as follows:

```
    logger.debug("Writing {} to {}".format(name, output_dir))
    logger.debug("Data Hash: {}".format(hash_data(data)))
    path = os.path.join(output_dir, name)
    path = path + "_" + str(counter)
    with open(path, "w") as outfile:
        outfile.write(data)
```

```
     logger.debug("Output File Hash: {}".format(hash_file(path)))
```

Now, define **hash_data()** method with the help of which we read the **UTF-8** encoded data and then generate the **SHA-256** hash of it as follows:

```
def hash_data(data):
     sha256 = hashlib.sha256()
     sha256.update(data.encode("utf-8"))
     return sha256.hexdigest()
def hash_file(file):
     sha256 = hashlib.sha256()
     with open(file, "rb") as in_file:
            sha256.update(in_file.read())
     return sha256.hexdigest()
```

Now, let us create a **Beautifulsoup** object out of the web page data under **find_links()** method as follows:

```
def find_links(website, page, queue):
     for link in BeautifulSoup(page, "html.parser",parse_only=SoupStrainer("a", 
href=True)):
            if website in link.get("href"):
                   if not os.path.basename(link.get("href")).startswith("#"):
                          queue.add(link.get("href"))
     return queue
```

Now, we need to define **recurse_pages()** method by providing it the inputs of the website URL, current link queue, the unverified SSL context and the output directory as follows:

```
def recurse_pages(website, queue, context, output_dir):
     processed = []
     counter = 0
     while True:
            counter += 1
            if len(processed) == len(queue):
            break

            for link in queue.copy():             if link in processed:
                          continue
```

```
                    processed.append(link)
                    try:
                      page = urlopen(link, context=context).read().decode("utf-8")
                    except urllib.error.HTTPError as e:
                            msg = "Error accessing webpage: {}".format(link)
                            logger.error(msg)
                            continue
```

Now, write the output of each web page accessed in a file by passing the link name, page data, output directory and the counter as follows:

```
                            write_output(link, page, output_dir, counter)
                            queue = find_links(website, page, queue)
            logger.info("Identified {} links throughout website".format(
                    len(queue)))
```

Now, when we run this script by providing the URL of the website, the output directory and a path to the log file, we will get the details about that web page that can be used for future use.

Virus Hunting

Have you ever wondered how forensic analysts, security researchers, and incident respondents can understand the difference between useful software and malware? The answer lies in the question itself, because without studying about the malware, rapidly generating by hackers, it is quite impossible for researchers and specialists to tell the difference between useful software and malware. In this section, let us discuss about **VirusShare**, a tool to accomplish this task.

Understanding VirusShare

VirusShare is the largest privately owned collection of malware samples to provide security researchers, incident responders, and forensic analysts the samples of live malicious code. It contains over 30 million samples.

The benefit of VirusShare is the list of malware hashes that is freely available. Anybody can use these hashes to create a very comprehensive hash set and use that to identify potentially malicious files. But before using VirusShare, we suggest you to visit https://virussshare.com for more details.

Creating Newline-Delimited Hash List from VirusShare using Python

A hash list from VirusShare can be used by various forensic tools such as X-ways and EnCase. In the script discussed below, we are going to automate downloading lists of hashes from VirusShare to create a newline-delimited hash list.

For this script, we need a third party Python library **tqdm** which can be downloaded as follows:

```
pip install tqdm
```

Note that in this script, first we will read the VirusShare hashes page and dynamically identify the most recent hash list. Then we will initialize the progress bar and download the hash list in the desired range.

First, import the following libraries:

```
from __future__ import print_function
import argparse
import os
import ssl
import sys
import tqdm
from urllib.request import urlopen
import urllib.error
```

This script will take one positional argument, which would be the desired path for the hash set:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser('Hash set from VirusShare')
    parser.add_argument("OUTPUT_HASH", help="Output Hashset")
    parser.add_argument("--start", type=int,
    help="Optional starting location")
    args = parser.parse_args()
```

Now, we will perform the standard input validation as follows:

```
    directory = os.path.dirname(args.OUTPUT_HASH)
    if not os.path.exists(directory):
        os.makedirs(directory)
    if args.start:
        main(args.OUTPUT_HASH, start=args.start)
        else:
                main(args.OUTPUT_HASH)
```

Now we need to define **main()** function with ****kwargs** as an argument because this will create a dictionary we can refer to support supplied key arguments as shown below:

```
def main(hashset, **kwargs):
        url = "https://virusshare.com/hashes.4n6"
        print("[+] Identifying hash set range from {}".format(url))
        context = ssl._create_unverified_context()
```

Now, we need to open VirusShare hashes page by using **urlib.request.urlopen()** method. We will use try-except block as follows:

```
        try:
                index = urlopen(url, context=context).read().decode("utf-8")
        except urllib.error.HTTPError as e:
                print("[-] Error accessing webpage - exiting..")
                sys.exit(1)
```

Now, identify latest hash list from downloaded pages. You can do this by finding the last instance of the HTML **href** tag to VirusShare hash list. It can be done with the following lines of code:

```
                tag = index.rfind(r'<a href="hashes/VirusShare_')
                stop = int(index[tag + 27: tag + 27 + 5].lstrip("0"))
                if "start" not in kwargs:
                        start = 0
                else:
                        start = kwargs["start"]
                if start < 0 or start > stop:
                        print("[-] Supplied start argument must be greater than or equal
""to zero but less than the latest hash list, ""currently: {}".format(stop))
                        sys.exit(2)
                print("[+] Creating a hashset from hash lists {} to {}".format(start,
stop))
                hashes_downloaded = 0
```

Now, we will use **tqdm.trange()** method to create a loop and progress bar as follows:

```
            for x in tqdm.trange(start, stop + 1,
unit_scale=True,desc="Progress"):
                   url_hash =
"https://virusshare.com/hashes/VirusShare_"\"{}.md5".format(str(x).zfill(5))
                   try:
                          hashes = urlopen(url_hash,
context=context).read().decode("utf-8")
                          hashes_list = hashes.split("\n")
                   except urllib.error.HTTPError as e:
                          print("[-] Error accessing webpage for hash list {}"" -
continuing..".format(x))
                          continue
```

After performing the above steps succefully, we will open the hash set text file in **a+** mode to append to the bottom of text file.

```
                          with open(hashset, "a+") as hashfile:
                                 for line in hashes_list:
                                        if not line.startswith("#") and line != "":
                                               hashes_downloaded += 1
                                               hashfile.write(line + '\n')
            print("[+] Finished downloading {} hashes into {}".format(
                   hashes_downloaded, hashset))
```

After running the above script, you will get the latest hash list containing MD5 hash values in text format.

8. Python Digital Forensics – Investigation using Emails

The previous chapters discussed about the importance and the process of network forensics and the concepts involved. In this chapter, let us learn about the role of emails in digital forensics and their investigation using Python.

Role of Email in Investigation

Emails play a very important role in business communications and have emerged as one of the most important applications on internet. They are a convenient mode for sending messages as well as documents, not only from computers but also from other electronic gadgets such as mobile phones and tablets.

The negative side of emails is that criminals may leak important information about their company. Hence, the role of emails in digital forensics has been increased in recent years. In digital forensics, emails are considered as crucial evidences and Email Header Analysis has become important to collect evidence during forensic process.

An investigator has the following goals while performing email forensics:

- To identify the main criminal
- To collect necessary evidences
- To presenting the findings
- To build the case

Challenges in Email Forensics

Email forensics play a very important role in investigation as most of the communication in present era relies on emails. However, an email forensic investigator may face the following challenges during the investigation:

Fake Emails

The biggest challenge in email forensics is the use of fake e-mails that are created by manipulating and scripting headers etc. In this category criminals also use temporary email which is a service that allows a registered user to receive email at a temporary address that expires after a certain time period.

Spoofing

Another challenge in email forensics is spoofing in which criminals used to present an email as someone else's. In this case the machine will receive both fake as well as original IP address.

Anonymous Re-emailing

Here, the Email server strips identifying information from the email message before forwarding it further. This leads to another big challenge for email investigations.

Techniques Used in Email Forensic Investigation

Email forensics is the study of source and content of email as evidence to identify the actual sender and recipient of a message along with some other information such as date/time of transmission and intention of sender. It involves investigating metadata, port scanning as well as keyword searching.

Some of the common techniques which can be used for email forensic investigation are:

- Header Analysis
- Server investigation
- Network Device Investigation
- Sender Mailer Fingerprints
- Software Embedded Identifiers

In the following sections, we are going to learn how to fetch information using Python for the purpose of email investigation.

Extraction of Information from EML files

EML files are basically emails in file format which are widely used for storing email messages. They are structured text files that are compatible across multiple email clients such as Microsoft Outlook, Outlook Express, and Windows Live Mail.

An EML file stores email headers, body content, attachment data as plain text. It uses **base64** to encode binary data and Quoted-Printable (QP) encoding to store content information. The Python script that can be used to extract information from EML file is given below:

First, import the following Python libraries as shown below:

```
from __future__ import print_function

from argparse import ArgumentParser, FileType

from email import message_from_file

import os

import quopri

import base64
```

In the above libraries, **quopri** is used to decode the QP encoded values from EML files. Any base64 encoded data can be decoded with the help of **base64** library.

Next, let us provide argument for command-line handler. Note that here it will accept only one argument which would be the path to EML file as shown below:

```
if __name__ == '__main__':

     parser = ArgumentParser('Extracting information from EML file')

     parser.add_argument("EML_FILE",help="Path to EML File", type=FileType('r'))

     args = parser.parse_args()

     main(args.EML_FILE)
```

Now, we need to define **main()** function in which we will use the method named **message_from_file()** from email library to read the file like object. Here we will access the headers, body content, attachments and other payload information by using resulting variable named **emlfile** as shown in the code given below:

```
def main(input_file):

     emlfile = message_from_file(input_file)

     for key, value in emlfile._headers:

             print("{}: {}".format(key, value))

print("\nBody\n")

if emlfile.is_multipart():

     for part in emlfile.get_payload():

             process_payload(part)

else:

     process_payload(emlfile[1])
```

Now, we need to define **process_payload()** method in which we will extract message body content by using **get_payload()** method. We will decode QP encoded data by using **quopri.decodestring()** function. We will also check the content MIME type so that it can handle the storage of the email properly. Observe the code given below:

```
def process_payload(payload):

     print(payload.get_content_type() + "\n" + "=" *
len(payload.get_content_type()))

     body = quopri.decodestring(payload.get_payload())

     if payload.get_charset():

             body = body.decode(payload.get_charset())

     else:

             try:
                     body = body.decode()
             except UnicodeDecodeError:
                     body = body.decode('cp1252')
     if payload.get_content_type() == "text/html":
             outfile = os.path.basename(args.EML_FILE.name) + ".html"
             open(outfile, 'w').write(body)
     elif payload.get_content_type().startswith('application'):
             outfile = open(payload.get_filename(), 'wb')
             body = base64.b64decode(payload.get_payload())
             outfile.write(body)
             outfile.close()
             print("Exported: {}\n".format(outfile.name))
     else:
             print(body)
```

After executing the above script, we will get the header information along with various payloads on the console.

Analyzing MSG Files using Python

Email messages come in many different formats. MSG is one such kind of format used by Microsoft Outlook and Exchange. Files with MSG extension may contain plain ASCII text for the headers and the main message body as well as hyperlinks and attachments.

In this section, we will learn how to extract information from MSG file using Outlook API. Note that the following Python script will work only on Windows. For this, we need to install third party Python library named **pywin32** as follows:

```
pip install pywin32
```

Now, import the following libraries using the commands shown:

```
from __future__ import print_function
from argparse import ArgumentParser
import os
import win32com.client
import pywintypes
```

Now, let us provide an argument for command-line handler. Here it will accept two arguments one would be the path to MSG file and other would be the desired output folder as follows:

```
if __name__ == '__main__':
     parser = ArgumentParser('Extracting information from MSG file')
     parser.add_argument("MSG_FILE", help="Path to MSG file")
     parser.add_argument("OUTPUT_DIR", help="Path to output folder")
     args = parser.parse_args()
     out_dir = args.OUTPUT_DIR
     if not os.path.exists(out_dir):
            os.makedirs(out_dir)
     main(args.MSG_FILE, args.OUTPUT_DIR)
```

Now, we need to define **main()** function in which we will call **win32com** library for setting up **Outlook API** which further allows access to the **MAPI** namespace.

```
def main(msg_file, output_dir):
     mapi = win32com.client.Dispatch("Outlook.Application").GetNamespace("MAPI")
     msg = mapi.OpenSharedItem(os.path.abspath(args.MSG_FILE))
     display_msg_attribs(msg)
     display_msg_recipients(msg)
     extract_msg_body(msg, output_dir)
     extract_attachments(msg, output_dir)
```

Now, define different functions which we are using in this script. The code given below shows defining the **display_msg_attribs()** function that allow us to display various attributes of a message like subject, to , BCC, CC, Size, SenderName, sent, etc.

```
def display_msg_attribs(msg):
     attribs = [
            'Application', 'AutoForwarded', 'BCC', 'CC', 'Class',
            'ConversationID', 'ConversationTopic', 'CreationTime',
            'ExpiryTime', 'Importance', 'InternetCodePage', 'IsMarkedAsTask',
            'LastModificationTime', 'Links','ReceivedTime', 'ReminderSet',
            'ReminderTime', 'ReplyRecipientNames', 'Saved', 'Sender',
            'SenderEmailAddress', 'SenderEmailType', 'SenderName', 'Sent',
            'SentOn', 'SentOnBehalfOfName', 'Size', 'Subject',
            'TaskCompletedDate', 'TaskDueDate', 'To', 'UnRead'
```

```
        ]
        print("\nMessage Attributes")
        for entry in attribs:
                print("{}: {}".format(entry, getattr(msg, entry, 'N/A')))
```

Now, define the **display_msg_recipeints()** function that iterates through the messages and displays the recipient details.

```
def display_msg_recipients(msg):
        recipient_attrib = [
                'Address', 'AutoResponse', 'Name', 'Resolved', 'Sendable'
        ]
        i = 1
        while True:
        try:
                recipient = msg.Recipients(i)
        except pywintypes.com_error:
                break
        print("\nRecipient {}".format(i))
        print("=" * 15)
        for entry in recipient_attrib:
                print("{}: {}".format(entry, getattr(recipient, entry, 'N/A')))
        i += 1
```

Next, we define **extract_msg_body()** function that extracts the body content, HTML as well as Plain text, from the message.

```
def extract_msg_body(msg, out_dir):
        html_data = msg.HTMLBody.encode('cp1252')
        outfile = os.path.join(out_dir, os.path.basename(args.MSG_FILE))
        open(outfile + ".body.html", 'wb').write(html_data)
        print("Exported: {}".format(outfile + ".body.html"))
        body_data = msg.Body.encode('cp1252')
        open(outfile + ".body.txt", 'wb').write(body_data)
        print("Exported: {}".format(outfile + ".body.txt"))
```

Next, we shall define the **extract_attachments()** function that exports attachment data into desired output directory.

```
def extract_attachments(msg, out_dir):
      attachment_attribs = [
              'DisplayName', 'FileName', 'PathName', 'Position', 'Size'
      ]
      i = 1 # Attachments start at 1
      while True:
              try:
                      attachment = msg.Attachments(i)
              except pywintypes.com_error:
                      break
```

Once all the functions are defined, we will print all the attributes to the console with the following line of codes:

```
              print("\nAttachment {}".format(i))
              print("=" * 15)
              for entry in attachment_attribs:
                      print('{}: {}'.format(entry, getattr(attachment, entry,"N/A")))
              outfile =
os.path.join(os.path.abspath(out_dir),os.path.split(args.MSG_FILE)[-1])
              if not os.path.exists(outfile):
                      os.makedirs(outfile)
              outfile = os.path.join(outfile, attachment.FileName)
              attachment.SaveAsFile(outfile)
              print("Exported: {}".format(outfile))
              i += 1
```

After running the above script, we will get the attributes of message and its attachments in the console window along with several files in the output directory.

Structuring MBOX files from Google Takeout using Python

MBOX files are text files with special formatting that split messages stored within. They are often found in association with UNIX systems, Thunderbolt, and Google Takeouts.

In this section, you will see a Python script, where we will be structuring MBOX files got from Google Takeouts. But before that we must know that how we can generate these MBOX files by using our Google account or Gmail account.

Acquiring Google Account Mailbox into MBX Format

Acquiring of Google account mailbox implies taking backup of our Gmail account. Backup can be taken for various personal or professional reasons. Note that Google provides backing up of Gmail data. To acquire our Google account mailbox into MBOX format, you need to follow the steps given below:

- Open **My account** dashboard.
- Go to Personal info & privacy section and select Control your content link.
- You can create a new archive or can manage existing one. If we click, **CREATE ARCHIVE** link, then we will get some check boxes for each Google product we wish to include.
- After selecting the products, we will get the freedom to choose file type and maximum size for our archive along with the delivery method to select from list.
- Finally, we will get this backup in MBOX format.

Python Code

Now, the MBOX file discussed above can be structured using Python as shown below:

First, need to import Python libraries as follows:

```
from __future__ import print_function
from argparse import ArgumentParser
import mailbox
import os
import time
import csv
from tqdm import tqdm
import base64
```

All the libraries have been used and explained in earlier scripts, except the **mailbox** library which is used to parse MBOX files.

Now, provide an argument for command-line handler. Here it will accept two arguments: one would be the path to MBOX file, and the other would be the desired output folder.

```
if __name__ == '__main__':
      parser = ArgumentParser('Parsing MBOX files')
      parser.add_argument("MBOX", help="Path to mbox file")
      parser.add_argument("OUTPUT_DIR",help="Path to output directory to write
report ""and exported content")
      args = parser.parse_args()
      main(args.MBOX, args.OUTPUT_DIR)
```

Now, will define **main()** function and call **mbox** class of mailbox library with the help of which we can parse a MBOX file by providing its path:

```
def main(mbox_file, output_dir):
      print("Reading mbox file")
      mbox = mailbox.mbox(mbox_file, factory=custom_reader)
      print("{} messages to parse".format(len(mbox)))
```

Now, define a reader method for **mailbox** library as follows:

```
def custom_reader(data_stream):
      data = data_stream.read()
      try:
            content = data.decode("ascii")
      except (UnicodeDecodeError, UnicodeEncodeError) as e:
            content = data.decode("cp1252", errors="replace")
      return mailbox.mboxMessage(content)
```

Now, create some variables for further processing as follows:

```
      parsed_data = []
      attachments_dir = os.path.join(output_dir, "attachments")
      if not os.path.exists(attachments_dir):
            os.makedirs(attachments_dir)
      columns = ["Date", "From", "To", "Subject", "X-Gmail-Labels", "Return-Path",
"Received", "Content-Type", "Message-ID","X-GM-THRID", "num_attachments_exported",
"export_path"]
```

Next, use **tqdm** to generate a progress bar and to track the iteration process as follows:

```
    for message in tqdm(mbox):
        msg_data = dict()
        header_data = dict(message._headers)
    for hdr in columns:
        msg_data[hdr] = header_data.get(hdr, "N/A")
```

Now, check weather message is having payloads or not. If it is having then we will define **write_payload()** method as follows:

```
    if len(message.get_payload()):
        export_path = write_payload(message, attachments_dir)
        msg_data['num_attachments_exported'] = len(export_path)
        msg_data['export_path'] = ", ".join(export_path)
```

Now, data need to be appended. Then we will call **create_report()** method as follows:

```
        parsed_data.append(msg_data)
    create_report(
    parsed_data, os.path.join(output_dir, "mbox_report.csv"), columns)
def write_payload(msg, out_dir):
    pyld = msg.get_payload()
    export_path = []
    if msg.is_multipart():
        for entry in pyld:
            export_path += write_payload(entry, out_dir)
    else:
        content_type = msg.get_content_type()
        if "application/" in content_type.lower():
            content = base64.b64decode(msg.get_payload())
            export_path.append(export_content(msg, out_dir, content))
        elif "image/" in content_type.lower():
            content = base64.b64decode(msg.get_payload())
            export_path.append(export_content(msg, out_dir, content))

        elif "video/" in content_type.lower():
            content = base64.b64decode(msg.get_payload())
            export_path.append(export_content(msg, out_dir, content))
        elif "audio/" in content_type.lower():
            content = base64.b64decode(msg.get_payload())
            export_path.append(export_content(msg, out_dir, content))
        elif "text/csv" in content_type.lower():
            content = base64.b64decode(msg.get_payload())
            export_path.append(export_content(msg, out_dir, content))
        elif "info/" in content_type.lower():
            export_path.append(export_content(msg, out_dir,
            msg.get_payload()))
        elif "text/calendar" in content_type.lower():
            export_path.append(export_content(msg, out_dir,
            msg.get_payload()))
        elif "text/rtf" in content_type.lower():
            export_path.append(export_content(msg, out_dir,
            msg.get_payload()))
        else:
            if "name=" in msg.get('Content-Disposition', "N/A"):
                content = base64.b64decode(msg.get_payload())
                export_path.append(export_content(msg, out_dir, content))
            elif "name=" in msg.get('Content-Type', "N/A"):
                content = base64.b64decode(msg.get_payload())
                export_path.append(export_content(msg, out_dir, content))
    return export_path
```

Observe that the above if-else statements are easy to understand. Now, we need to define a method that will extract the filename from the **msg** object as follows:

```
def export_content(msg, out_dir, content_data):
    file_name = get_filename(msg)
    file_ext = "FILE"
    if "." in file_name:            file_ext = file_name.rsplit(".", 1)[-1]
    file_name = "{}_{:.4f}.{}".format(file_name.rsplit(".", 1)[0], time.time(),
file_ext)
```

```
    file_name = os.path.join(out_dir, file_name)
```

Now, with the help of following lines of code, you can actually export the file:

```
    if isinstance(content_data, str):
        open(file_name, 'w').write(content_data)
    else:
        open(file_name, 'wb').write(content_data)
    return file_name
```

Now, let us define a function to extract filenames from the **message** to accurately represent the names of these files as follows:

```
def get_filename(msg):
    if 'name=' in msg.get("Content-Disposition", "N/A"):
        fname_data = msg["Content-Disposition"].replace("\r\n", " ")
        fname = [x for x in fname_data.split("; ") if 'name=' in x]
        file_name = fname[0].split("=", 1)[-1]
    elif 'name=' in msg.get("Content-Type", "N/A"):
        fname_data = msg["Content-Type"].replace("\r\n", " ")
        fname = [x for x in fname_data.split("; ") if 'name=' in x]
        file_name = fname[0].split("=", 1)[-1]
    else:
        file_name = "NO_FILENAME"
    fchars = [x for x in file_name if x.isalnum() or x.isspace() or x == "."]
    return "".join(fchars)
```

Now, we can write a CSV file by defining the **create_report()** function as follows:

```
def create_report(output_data, output_file, columns):
    with open(output_file, 'w', newline="") as outfile:
        csvfile = csv.DictWriter(outfile, columns)
        csvfile.writeheader()
        csvfile.writerows(output_data)
```

Once you run the script given above, we will get the CSV report and directory full of attachments.

9. Python Digital Forensics – Important Artifacts in Windows-I

This chapter will explain various concepts involved in Microsoft Windows forensics and the important artifacts that an investigator can obtain from the investigation process.

Introduction

Artifacts are the objects or areas within a computer system that have important information related to the activities performed by the computer user. The type and location of this information depends upon the operating system. During forensic analysis, these artifacts play a very important role in approving or disapproving the investigator's observation.

Importance of Windows Artifacts for Forensics

Windows artifacts assume significance due to the following reasons:

- Around 90% of the traffic in world comes from the computers using Windows as their operating system. That is why for digital forensics examiners Windows artifacts are very essentials.
- The Windows operating system stores different types of evidences related to the user activity on computer system. This is another reason which shows the importance of Windows artifacts for digital forensics.
- Many times the investigator revolves the investigation around old and traditional areas like user crated data. Windows artifacts can lead the investigation towards non-traditional areas like system created data or the artifacts.
- Great abundance of artifacts is provided by Windows which are helpful for investigators as well as for companies and individuals performing informal investigations.
- Increase in cyber-crime in recent years is another reason that Windows artifacts are important.

Windows Artifacts and their Python Scripts

In this section, we are going to discuss about some Windows artifacts and Python scripts to fetch information from them.

Recycle Bin

It is one of the important Windows artifacts for forensic investigation. Windows recycle bin contains the files that have been deleted by the user, but not physically removed by the system yet. Even if the user completely removes the file from system, it serves as an important source of investigation. This is because the examiner can extract valuable information, like original file path as well as time that it was sent to Recycle Bin, from the deleted files.

Note that the storage of Recycle Bin evidence depends upon the version of Windows. In the following Python script, we are going to deal with Windows 7 where it creates two files: **$R** file that contains the actual content of the recycled file and **$I** file that contains original file name, path, file size when file was deleted.

For Python script we need to install third party modules namely **pytsk3, pyewf** and **unicodecsv**. We can use **pip** to install them. We can follow the following steps to extract information from Recycle Bin:

- First, we need to use recursive method to scan through the **$Recycle.bin** folder and select all the files starting with **$I**.
- Next, we will read the contents of the files and parse the available metadata structures.
- Now, we will search for the associated $R file.
- At last, we will write the results into CSV file for review.

Let us see how to use Python code for this purpose:

First, we need to import the following Python libraries:

```
from __future__ import print_function

from argparse import ArgumentParser

import datetime

import os

import struct

from utility.pytskutil import TSKUtil

import unicodecsv as csv
```

Next, we need to provide argument for command-line handler. Note that here it will accept three arguments – first is the path to evidence file, second is the type of evidence file and third is the desired output path to the CSV report, as shown below:

```
if __name__ == '__main__':

     parser = argparse.ArgumentParser('Recycle Bin evidences')

     parser.add_argument('EVIDENCE_FILE', help="Path to evidence file")

     parser.add_argument('IMAGE_TYPE', help="Evidence file format",

     choices=('ewf', 'raw'))

     parser.add_argument('CSV_REPORT', help="Path to CSV report")

     args = parser.parse_args()

     main(args.EVIDENCE_FILE, args.IMAGE_TYPE, args.CSV_REPORT)
```

Now, define the **main()** function that will handle all the processing. It will search for **$I** file as follows:

```
def main(evidence, image_type, report_file):
    tsk_util = TSKUtil(evidence, image_type)
    dollar_i_files = tsk_util.recurse_files("$I",
path='/$Recycle.bin',logic="startswith")
    if dollar_i_files is not None:
        processed_files = process_dollar_i(tsk_util, dollar_i_files)
        write_csv(report_file,['file_path', 'file_size',
'deleted_time','dollar_i_file', 'dollar_r_file', 'is_directory'],processed_files)
    else:
        print("No $I files found")
```

Now, if we found **$I** file, then it must be sent to **process_dollar_i()** function which will accept the **tsk_util** object as well as the list of $I files, as shown below:

```
 def process_dollar_i(tsk_util, dollar_i_files):
    processed_files = []
    for dollar_i in dollar_i_files:
        file_attribs = read_dollar_i(dollar_i[2])
        if file_attribs is None:
            continue
        file_attribs['dollar_i_file'] = os.path.join('/$Recycle.bin',
dollar_i[1][1:])
```

Now, search for **$R** files as follows:

```


        recycle_file_path =
os.path.join('/$Recycle.bin',dollar_i[1].rsplit("/", 1)[0][1:])
        dollar_r_files = tsk_util.recurse_files("$R" +
dollar_i[0][2:],path=recycle_file_path, logic="startswith")
        if dollar_r_files is None:
            dollar_r_dir = os.path.join(recycle_file_path,"$R" +
dollar_i[0][2:])
            dollar_r_dirs = tsk_util.query_directory(dollar_r_dir)
        if dollar_r_dirs is None:
            file_attribs['dollar_r_file'] = "Not Found"
            file_attribs['is_directory'] = 'Unknown'
        else:
            file_attribs['dollar_r_file'] = dollar_r_dir
            file_attribs['is_directory'] = True
        else:
            dollar_r = [os.path.join(recycle_file_path, r[1][1:])for r in 
dollar_r_files]
            file_attribs['dollar_r_file'] = ";".join(dollar_r)
            file_attribs['is_directory'] = False
            processed_files.append(file_attribs)
return processed_files
```

Now, define **read_dollar_i()** method to read the **$I** files, in other words, parse the metadata. We will use **read_random()** method to read the signature's first eight bytes. This will return none if signature does not match. After that, we will have to read and unpack the values from **$I** file if that is a valid file.

```
 def read_dollar_i(file_obj):
      if file_obj.read_random(0, 8) != '\x01\x00\x00\x00\x00\x00\x00\x00':
            return None
      raw_file_size = struct.unpack('<q', file_obj.read_random(8, 8))
      raw_deleted_time = struct.unpack('<q', file_obj.read_random(16, 8))
      raw_file_path = file_obj.read_random(24, 520)
```

Now, after extracting these files we need to interpret the integers into human-readable values by using **sizeof_fmt()** function as shown below:

```
      file_size = sizeof_fmt(raw_file_size[0])
      deleted_time = parse_windows_filetime(raw_deleted_time[0])
      file_path = raw_file_path.decode("utf16").strip("\x00")
      return {'file_size': file_size, 'file_path': file_path,'deleted_time': 
deleted_time}
```

Now, we need to define **sizeof_fmt()** function as follows:

```
def sizeof_fmt(num, suffix='B'):
    for unit in ['', 'Ki', 'Mi', 'Gi', 'Ti', 'Pi', 'Ei', 'Zi']:
        if abs(num) < 1024.0:
            return "%3.1f%s%s" % (num, unit, suffix)
        num /= 1024.0
    return "%.1f%s%s" % (num, 'Yi', suffix)
```

Now, define a function for interpreted integers into formatted date and time as follows:

```
def parse_windows_filetime(date_value):
    microseconds = float(date_value) / 10
    ts = datetime.datetime(1601, 1, 1) + datetime.timedelta(
        microseconds=microseconds)
    return ts.strftime('%Y-%m-%d %H:%M:%S.%f')
```

Now, we will define **write_csv()** method to write the processed results into a CSV file as follows:

```
def write_csv(outfile, fieldnames, data):
    with open(outfile, 'wb') as open_outfile:
        csvfile = csv.DictWriter(open_outfile, fieldnames)
        csvfile.writeheader()
        csvfile.writerows(data)
```

When you run the above script, we will get the data from $I and $R file.

Sticky Notes

Windows Sticky Notes replaces the real world habit of writing with pen and paper. These notes used to float on the desktop with different options for colors, fonts etc. In Windows 7 the Sticky Notes file is stored as an OLE file hence in the following Python script we will investigate this OLE file to extract metadata from Sticky Notes.

For this Python script, we need to install third party modules namely **olefile, pytsk3, pyewf** and **unicodecsv**. We can use the command **pip** to install them.

We can follow the steps discussed below for extracting the information from Sticky note file namely **StickyNote.snt:**

- Firstly, open the evidence file and find all the StickyNote.snt files.
- Then, parse the metadata and content from the OLE stream and write the RTF content to files.
- Lastly, create CSV report of this metadata.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
from argparse import ArgumentParser
import unicodecsv as csv
import os
import StringIO
from utility.pytskutil import TSKUtil
import olefile
```

Next, define a global variable which will be used across this script:

```
REPORT_COLS = ['note_id', 'created', 'modified', 'note_text', 'note_file']
```

Next, we need to provide argument for command-line handler. Note that here it will accept three arguments – first is the path to evidence file, second is the type of evidence file and third is the desired output path as follows:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser('Evidence from Sticky Notes')
    parser.add_argument('EVIDENCE_FILE', help="Path to evidence file")
    parser.add_argument('IMAGE_TYPE', help="Evidence file format",choices=('ewf',
'raw'))
    parser.add_argument('REPORT_FOLDER', help="Path to report folder")
    args = parser.parse_args()
    main(args.EVIDENCE_FILE, args.IMAGE_TYPE, args.REPORT_FOLDER)
```

Now, we will define **main()** function which will be similar to the previous script as shown below:

```
def main(evidence, image_type, report_folder):
    tsk_util = TSKUtil(evidence, image_type)
    note_files = tsk_util.recurse_files('StickyNotes.snt', '/Users','equals')
```

Now, let us iterate through the resulting files. Then we will call **parse_snt_file()** function to process the file and then we will write RTF file with the **write_note_rtf()** method as follows:

```
    report_details = []
    for note_file in note_files:
        user_dir = note_file[1].split("/")[1]
        file_like_obj = create_file_like_obj(note_file[2])
        note_data = parse_snt_file(file_like_obj)
        if note_data is None:
            continue
        write_note_rtf(note_data, os.path.join(report_folder, user_dir))
        report_details += prep_note_report(note_data, REPORT_COLS,"/Users" + 
note_file[1])
    write_csv(os.path.join(report_folder, 'sticky_notes.csv'), 
REPORT_COLS,report_details)
```

Next, we need to define various functions used in this script.

First of all we will define **create_file_like_obj()** function for reading the size of the file by taking **pytsk** file object. Then we will define **parse_snt_file()** function that will accept the file-like object as its input and is used to read and interpret the sticky note file.

```
 def parse_snt_file(snt_file):
    if not olefile.isOleFile(snt_file):
        print("This is not an OLE file")
        return None
    ole = olefile.OleFileIO(snt_file)
    note = {}
    for stream in ole.listdir():
        if stream[0].count("-") == 3:
            if stream[0] not in note:
                note[stream[0]] = {"created": 
ole.getctime(stream[0]),"modified": ole.getmtime(stream[0])}
            content = None
            if stream[1] == '0':
                content = ole.openstream(stream).read()
            elif stream[1] == '3':
                content = ole.openstream(stream).read().decode("utf-16")
            if content:
                note[stream[0]][stream[1]] = content
    return note
```

Now, create a RTF file by defining **write_note_rtf()** function as follows:

```
def write_note_rtf(note_data, report_folder):
    if not os.path.exists(report_folder):
        os.makedirs(report_folder)
    for note_id, stream_data in note_data.items():
        fname = os.path.join(report_folder, note_id + ".rtf")
        with open(fname, 'w') as open_file:
            open_file.write(stream_data['0'])
```

Now, we will translate the nested dictionary into a flat list of dictionaries that are more appropriate for a CSV spreadsheet. It will be done by defining **prep_note_report()** function. Lastly, we will define **write_csv()** function.

```
def prep_note_report(note_data, report_cols, note_file):
    report_details = []
    for note_id, stream_data in note_data.items():
        report_details.append({
            "note_id": note_id,
            "created": stream_data['created'],
            "modified": stream_data['modified'],
            "note_text": stream_data['3'].strip("\x00"),
            "note_file": note_file
        })
    return report_details
def write_csv(outfile, fieldnames, data):
    with open(outfile, 'wb') as open_outfile:
        csvfile = csv.DictWriter(open_outfile, fieldnames)
        csvfile.writeheader()
        csvfile.writerows(data)
```

After running the above script, we will get the metadata from Sticky Notes file.

Registry Files

Windows registry files contain many important details which are like a treasure trove of information for a forensic analyst. It is a hierarchical database that contains details related to operating system configuration, user activity, software installation etc. In the following Python script we are going to access common baseline information from the **SYSTEM** and **SOFTWARE** hives.

For this Python script, we need to install third party modules namely **pytsk3, pyewf** and **registry**. We can use **pip** to install them.

We can follow the steps given below for extracting the information from Windows registry:

- First, find registry hives to process by its name as well as by path.
- Then we to open these files by using **`StringIO`** and **`Registry`** modules.
- At last we need to process each and every hive and print the parsed values to the console for interpretation.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
from argparse import ArgumentParser
import datetime
import StringIO
import struct
from utility.pytskutil import TSKUtil
from Registry import Registry
```

Now, provide argument for the command-line handler. Here it will accept two arguments - first is the path to the evidence file, second is the type of evidence file, as shown below:

```
if __name__ == '__main__':
     parser = argparse.ArgumentParser('Evidence from Windows Registry')
     parser.add_argument('EVIDENCE_FILE', help="Path to evidence file")
     parser.add_argument('IMAGE_TYPE', help="Evidence file format",
     choices=('ewf', 'raw'))
     args = parser.parse_args()
     main(args.EVIDENCE_FILE, args.IMAGE_TYPE)
```

Now we will define **main()** function for searching **SYSTEM** and **SOFTWARE** hives within **/Windows/System32/config** folder as follows:

```
def main(evidence, image_type):
      tsk_util = TSKUtil(evidence, image_type)
      tsk_system_hive = tsk_util.recurse_files('system',
'/Windows/system32/config', 'equals')
      tsk_software_hive = tsk_util.recurse_files('software',
'/Windows/system32/config', 'equals')
      system_hive = open_file_as_reg(tsk_system_hive[0][2])
      software_hive = open_file_as_reg(tsk_software_hive[0][2])
      process_system_hive(system_hive)
      process_software_hive(software_hive)
```

Now, define the function for opening the registry file. For this purpose, we need to gather the size of file from **pytsk** metadata as follows:

```
def open_file_as_reg(reg_file):
      file_size = reg_file.info.meta.size
      file_content = reg_file.read_random(0, file_size)
      file_like_obj = StringIO.StringIO(file_content)
      return Registry.Registry(file_like_obj)
```

Now, with the help of following method, we can process **SYSTEM** hive:

```
def process_system_hive(hive):
      root = hive.root()
      current_control_set = root.find_key("Select").value("Current").value()
      control_set = root.find_key("ControlSet{:03d}".format(current_control_set))
      raw_shutdown_time = struct.unpack('<Q',
control_set.find_key("Control").find_key("Windows").value("ShutdownTime").value())
      shutdown_time = parse_windows_filetime(raw_shutdown_time[0])
      print("Last Shutdown Time: {}".format(shutdown_time))
      time_zone =
control_set.find_key("Control").find_key("TimeZoneInformation").value("TimeZoneKey
Name").value()
      print("Machine Time Zone: {}".format(time_zone))
      computer_name =
control_set.find_key("Control").find_key("ComputerName").find_key("ComputerName").
value("ComputerName").value()
      print("Machine Name: {}".format(computer_name))
```

```
     last_access =
control_set.find_key("Control").find_key("FileSystem").value("NtfsDisableLastAcces
sUpdate").value()

     last_access = "Disabled" if last_access == 1 else "enabled"

     print("Last Access Updates: {}".format(last_access))
```

Now, we need to define a function for interpreted integers into formatted date and time as follows:

```
def parse_windows_filetime(date_value):

     microseconds = float(date_value) / 10

     ts = datetime.datetime(1601, 1, 1) + datetime.timedelta(

            microseconds=microseconds)

     return ts.strftime('%Y-%m-%d %H:%M:%S.%f')

def parse_unix_epoch(date_value):

     ts = datetime.datetime.fromtimestamp(date_value)

     return ts.strftime('%Y-%m-%d %H:%M:%S.%f')
```

Now with the help of following method we can process **SOFTWARE** hive:

```
 def process_software_hive(hive):

     root = hive.root()

     nt_curr_ver = root.find_key("Microsoft").find_key("Windows
NT").find_key("CurrentVersion")

     print("Product name: {}".format(nt_curr_ver.value("ProductName").value()))

     print("CSD Version: {}".format(nt_curr_ver.value("CSDVersion").value()))

     print("Current Build: {}".format(nt_curr_ver.value("CurrentBuild").value()))

     print("Registered Owner:
{}".format(nt_curr_ver.value("RegisteredOwner").value()))

     print("Registered Org:
{}".format(nt_curr_ver.value("RegisteredOrganization").value()))

     raw_install_date = nt_curr_ver.value("InstallDate").value()

     install_date = parse_unix_epoch(raw_install_date)

     print("Installation Date: {}".format(install_date))
```

After running the above script, we will get the metadata stored in Windows Registry files.

10. Python Digital Forensics – Important Artifacts in Windows-II

This chapter talks about some more important artifacts in Windows and their extraction method using Python.

User Activities

Windows having **NTUSER.DAT** file for storing various user activities. Every user profile is having hive like **NTUSER.DAT,** which stores the information and configurations related to that user specifically. Hence, it is highly useful for the purpose of investigation by forensic analysts.

The following Python script will parse some of the keys of **NTUSER.DAT** for exploring the actions of a user on the system. Before proceeding further, for Python script, we need to install third party modules namely **Registry, pytsk3**, **pyewf** and **Jinja2**. We can use **pip** to install them.

We can follow the following steps to extract information from **NTUSER.DAT** file:

- First, search for all **NTUSER.DAT** files in the system.

- Then parse the **WordWheelQuery, TypePath and RunMRU** key for each **NTUSER.DAT** file.

- At last we will write these artifacts, already processed, to an HTML report by using **Jinja2** module.

Python Code

Let us see how to use Python code for this purpose:

First of all, we need to import the following Python modules:

```
from __future__ import print_function

from argparse import ArgumentParser

import os

import StringIO

import struct

from utility.pytskutil import TSKUtil

from Registry import Registry

import jinja2
```

Now, provide argument for command-line handler. Here it will accept three arguments - first is the path to evidence file, second is the type of evidence file and third is the desired output path to the HTML report, as shown below:

```
if __name__ == '__main__':
     parser = argparse.ArgumentParser('Information from user activities')
     parser.add_argument('EVIDENCE_FILE',help="Path to evidence file")
     parser.add_argument('IMAGE_TYPE',help="Evidence file format",choices=('ewf',
'raw'))
     parser.add_argument('REPORT',help="Path to report file")
     args = parser.parse_args()
     main(args.EVIDENCE_FILE, args.IMAGE_TYPE, args.REPORT)
```

Now, let us define **main()** function for searching all **NTUSER.DAT** files, as shown:

```
def main(evidence, image_type, report):
     tsk_util = TSKUtil(evidence, image_type)
     tsk_ntuser_hives = tsk_util.recurse_files('ntuser.dat','/Users', 'equals')
     nt_rec = {
             'wordwheel': {'data': [], 'title': 'WordWheel Query'},
             'typed_path': {'data': [], 'title': 'Typed Paths'},
             'run_mru': {'data': [], 'title': 'Run MRU'}
     }
```

Now, we will try to find the key in **NTUSER.DAT** file and once you find it, define the user processing functions as shown below:

```
     for ntuser in tsk_ntuser_hives:
             uname = ntuser[1].split("/")


     open_ntuser = open_file_as_reg(ntuser[2])
             try:
                     explorer_key =
open_ntuser.root().find_key("Software").find_key("Microsoft").find_key("Windows").
find_key("CurrentVersion").find_key("Explorer")
             except Registry.RegistryKeyNotFoundException:
                     continue
             nt_rec['wordwheel']['data'] += parse_wordwheel(explorer_key, uname)
             nt_rec['typed_path']['data'] += parse_typed_paths(explorer_key, uname)
```

```
            nt_rec['run_mru']['data'] += parse_run_mru(explorer_key, uname)


            nt_rec['wordwheel']['headers'] = \
                  nt_rec['wordwheel']['data'][0].keys()
            nt_rec['typed_path']['headers'] = \
                  nt_rec['typed_path']['data'][0].keys()
            nt_rec['run_mru']['headers'] = \
                  nt_rec['run_mru']['data'][0].keys()
```

Now, pass the dictionary object and its path to **write_html()** method as follows:

```
            write_html(report, nt_rec)
```

Now, define a method, that takes **pytsk** file handle and read it into the Registry class via the **StringIO** class.

```
def open_file_as_reg(reg_file):
      file_size = reg_file.info.meta.size
      file_content = reg_file.read_random(0, file_size)
      file_like_obj = StringIO.StringIO(file_content)
      return Registry.Registry(file_like_obj)
```

Now, we will define the function that will parse and handles **WordWheelQuery** key from **NTUSER.DAT** file as follows:

```
def parse_wordwheel(explorer_key, username):
      try:
            wwq = explorer_key.find_key("WordWheelQuery")
      except Registry.RegistryKeyNotFoundException:
            return []
      mru_list = wwq.value("MRUListEx").value()
      mru_order = []
      for i in xrange(0, len(mru_list), 2):
            order_val = struct.unpack('h', mru_list[i:i + 2])[0]
            if order_val in mru_order and order_val in (0, -1):
                  break
            else:
                  mru_order.append(order_val)
      search_list = []
      for count, val in enumerate(mru_order):
             ts = "N/A"
             if count == 0:
                    ts = wwq.timestamp()
             search_list.append({
                    'timestamp': ts,
                    'username': username,
                    'order': count,
                    'value_name': str(val),
                    'search': wwq.value(str(val)).value().decode("UTF-
16").strip("\x00")
             })
      return search_list
```

Now, we will define the function that will parse and handles **TypedPaths** key from **NTUSER.DAT** file as follows:

```
def parse_typed_paths(explorer_key, username):
      try:
             typed_paths = explorer_key.find_key("TypedPaths")
      except Registry.RegistryKeyNotFoundException:
             return []
      typed_path_details = []
      for val in typed_paths.values():
             typed_path_details.append({
                    "username": username,
                    "value_name": val.name(),
                    "path": val.value()
             })
      return typed_path_details
```

Now, we will define the function that will parse and handles **RunMRU** key from **NTUSER.DAT** file as follows:

```
def parse_run_mru(explorer_key, username):
    try:
        run_mru = explorer_key.find_key("RunMRU")
    except Registry.RegistryKeyNotFoundException:
        return []
    if len(run_mru.values()) == 0:
        return []
    mru_list = run_mru.value("MRUList").value()
    mru_order = []
    for i in mru_list:
        mru_order.append(i)
    mru_details = []
    for count, val in enumerate(mru_order):
        ts = "N/A"
        if count == 0:
            ts = run_mru.timestamp()
        mru_details.append({
            "username": username,
            "timestamp": ts,
            "order": count,
            "value_name": val,
            "run_statement": run_mru.value(val).value()
        })
    return mru_details
```

Now, the following function will handle the creation of HTML report:

```
def write_html(outfile, data_dict):
    cwd = os.path.dirname(os.path.abspath(__file__))
    env = jinja2.Environment(loader=jinja2.FileSystemLoader(cwd))
    template = env.get_template("user_activity.html")
    rendering = template.render(nt_data=data_dict)
    with open(outfile, 'w') as open_outfile:
        open_outfile.write(rendering)
```

At last we can write HTML document for report. After running the above script, we will get the information from NTUSER.DAT file in HTML document format.

LINK files

Shortcuts files are created when a user or the operating system creates shortcut files for the files which are frequently used, double clicked or accessed from system drives such as attached storage. Such kinds of shortcut files are called link files. By accessing these link files, an investigator can find the activity of window such as the time and location from where these files have been accessed.

Let us discuss the Python script that we can use to get the information from these Windows LINK files.

For Python script, install third party modules namely **pylnk, pytsk3, pyewf**. We can follow the following steps to extract information from **lnk** files:

- First, search for **lnk** files within the system.
- Then, extract the information from that file by iterating through them.
- Now, at last we need to this information to a CSV report.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
from argparse import ArgumentParser
import csv
import StringIO
from utility.pytskutil import TSKUtil
import pylnk
```

Now, provide the argument for command-line handler. Here it will accept three arguments – first is the path to evidence file, second is the type of evidence file and third is the desired output path to the CSV report, as shown below:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser('Parsing LNK files')
    parser.add_argument('EVIDENCE_FILE', help="Path to evidence file")
```

```
     parser.add_argument('IMAGE_TYPE', help="Evidence file format",choices=('ewf',
'raw'))
     parser.add_argument('CSV_REPORT', help="Path to CSV report")
     args = parser.parse_args()
     main(args.EVIDENCE_FILE, args.IMAGE_TYPE, args.CSV_REPORT)
```

Now, interpret the evidence file by creating an object of **TSKUtil** and iterate through the file system to find files ending with **lnk**. It can be done by defining **main()** function as follows:

```
def main(evidence, image_type, report):
     tsk_util = TSKUtil(evidence, image_type)
     lnk_files = tsk_util.recurse_files("lnk", path="/", logic="endswith")
     if lnk_files is None:
            print("No lnk files found")
            exit(0)
     columns = [
            'command_line_arguments', 'description', 'drive_serial_number',
            'drive_type', 'file_access_time', 'file_attribute_flags',
            'file_creation_time', 'file_modification_time', 'file_size',
            'environmental_variables_location', 'volume_label',
            'machine_identifier', 'local_path', 'network_path',
            'relative_path', 'working_directory'
     ]
```

Now with the help of following code, we will iterate through **lnk** files by creating a function as follows:

```
     parsed_lnks = []
     for entry in lnk_files:
            lnk = open_file_as_lnk(entry[2])
            lnk_data = {'lnk_path': entry[1], 'lnk_name': entry[0]}
            for col in columns:
                   lnk_data[col] = getattr(lnk, col, "N/A")
            lnk.close()
            parsed_lnks.append(lnk_data)
     write_csv(report, columns + ['lnk_path', 'lnk_name'], parsed_lnks)
```

Now we need to define two functions, one will open the **pytsk** file object and other will be used for writing CSV report as shown below:

```
def open_file_as_lnk(lnk_file):
      file_size = lnk_file.info.meta.size
      file_content = lnk_file.read_random(0, file_size)
      file_like_obj = StringIO.StringIO(file_content)
      lnk = pylnk.file()
      lnk.open_file_object(file_like_obj)
      return lnk
def write_csv(outfile, fieldnames, data):
      with open(outfile, 'wb') as open_outfile:
             csvfile = csv.DictWriter(open_outfile, fieldnames)
             csvfile.writeheader()
             csvfile.writerows(data)
```

After running the above script, we will get the information from the discovered **lnk** files in a CSV report.

Prefetch Files

Whenever an application is running for the first time from a specific location, Windows creates **prefetch files.** These are used to speed up the application startup process. The extension for these files is **.PF** and these are stored in the **"\Root\Windows\Prefetch"** folder.

Digital forensic experts can reveal the evidence of program execution from a specified location along with the details of the user. Prefetch files are useful artifacts for the examiner because their entry remains even after the program has been deleted or un-installed.

Let us discuss the Python script that will fetch information from Windows prefetch files as given below:

For Python script, install third party modules namely **pylnk, pytsk3** and **unicodecsv**. Recall that we have already worked with these libraries in the Python scripts that we have discussed in the previous chapters.

We have to follow steps given below to extract information from **prefetch** files:

- First, scan for **.pf** extension files or the prefetch files.
- Now, perform the signature verification to eliminate false positives.
- Next, parse the Windows prefetch file format. This differs with the Windows version. For example, for Windows XP it is 17, for Windows Vista and Windows 7 it is 23, 26 for Windows 8.1 and 30 for Windows 10.
- Lastly, we will write the parsed result in a CSV file.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
import argparse
from datetime import datetime, timedelta
import os
import pytsk3
import pyewf
import struct
import sys
import unicodecsv as csv
from utility.pytskutil import TSKUtil
```

Now, provide an argument for command-line handler. Here it will accept two arguments, first would be the path to evidence file and second would be the type of evidence file. It also accepts an optional argument for specifying the path to scan for prefetch files:

```
if __name__ == "__main__":
     parser = argparse.ArgumentParser('Parsing Prefetch files')
     parser.add_argument("EVIDENCE_FILE", help="Evidence file path")
     parser.add_argument("TYPE", help="Type of Evidence",choices=("raw", "ewf"))
     parser.add_argument("OUTPUT_CSV", help="Path to write output csv")
     parser.add_argument("-d", help="Prefetch directory to
scan",default="/WINDOWS/PREFETCH")
     args = parser.parse_args()
     if os.path.exists(args.EVIDENCE_FILE) and \
                 os.path.isfile(args.EVIDENCE_FILE):
           main(args.EVIDENCE_FILE, args.TYPE, args.OUTPUT_CSV, args.d)
     else:
           print("[-] Supplied input file {} does not exist or is not a
""file".format(args.EVIDENCE_FILE))
           sys.exit(1)
```

Now, interpret the evidence file by creating an object of **TSKUtil** and iterate through the file system to find files ending with **.pf**. It can be done by defining **main()** function as follows:

```
def main(evidence, image_type, output_csv, path):
    tsk_util = TSKUtil(evidence, image_type)
    prefetch_dir = tsk_util.query_directory(path)
    prefetch_files = None
    if prefetch_dir is not None:
        prefetch_files = tsk_util.recurse_files(".pf", path=path, 
logic="endswith")
    if prefetch_files is None:
        print("[-] No .pf files found")
        sys.exit(2)
    print("[+] Identified {} potential prefetch 
files".format(len(prefetch_files)))
    prefetch_data = []
    for hit in prefetch_files:
        prefetch_file = hit[2]
        pf_version = check_signature(prefetch_file)
```

Now, define a method that will do the validation of signatures as shown below:

```
def check_signature(prefetch_file):
    version, signature = struct.unpack("<2i", prefetch_file.read_random(0, 8))
    if signature == 1094927187:
        return version
    else:
        return None
    if pf_version is None:
        continue
    pf_name = hit[0]
    if pf_version == 17:
        parsed_data = parse_pf_17(prefetch_file, pf_name)
        parsed_data.append(os.path.join(path, hit[1].lstrip("//")))
        prefetch_data.append(parsed_data)
```

Now, start processing Windows prefetch files. Here we are taking the example of Windows XP prefetch files:

```
def parse_pf_17(prefetch_file, pf_name):
     create = convert_unix(prefetch_file.info.meta.crtime)
     modify = convert_unix(prefetch_file.info.meta.mtime)
def convert_unix(ts):
     if int(ts) == 0:
            return ""
     return datetime.utcfromtimestamp(ts)
def convert_filetime(ts):
     if int(ts) == 0:
            return ""
     return datetime(1601, 1, 1) + timedelta(microseconds=ts / 10)
```

Now, extract the data embedded within the prefetched files by using **`struct`** as follows:

```
     pf_size, name, vol_info, vol_entries, vol_size, filetime, \
            count = struct.unpack("<i60s32x3iq16xi",prefetch_file.read_random(12, 
136))
     name = name.decode("utf-16", "ignore").strip("/x00").split("/x00")[0]
     vol_name_offset, vol_name_length, vol_create, \
            vol_serial = struct.unpack("<2iqi",prefetch_file.read_random(vol_info, 
20))
     vol_serial = hex(vol_serial).lstrip("0x")
     vol_serial = vol_serial[:4] + "-" + vol_serial[4:]
     vol_name = struct.unpack(
            "<{}s".format(2 * vol_name_length),
            prefetch_file.read_random(vol_info + vol_name_offset,vol_name_length * 
2))[0]
     vol_name = vol_name.decode("utf-16", "ignore").strip("/x00").split("/x00")[0]
     return [
            pf_name, name, pf_size, create,
            modify, convert_filetime(filetime), count, vol_name,
            convert_filetime(vol_create), vol_serial      ]
```

As we have provided the prefetch version for Windows XP but what if it will encounter prefetch versions for other Windows. Then it must have to display an error message as follows:

```
            elif pf_version == 23:
                print("[-] Windows Vista / 7 PF file {} -- unsupported".format(
                        pf_name))
                continue
            elif pf_version == 26:
                print("[-] Windows 8 PF file {} -- unsupported".format(
                        pf_name))
            continue
            elif pf_version == 30:
                print("[-] Windows 10 PF file {} -- unsupported".format(
                        pf_name))
            continue
    else:
            print("[-] Signature mismatch - Name: {}\nPath: {}".format(hit[0], 
hit[1]))
            continue
write_output(prefetch_data, output_csv)
```

Now, define the method for writing result into CSV report as follows:

```
def write_output(data, output_csv):
    print("[+] Writing csv report")
    with open(output_csv, "wb") as outfile:
            writer = csv.writer(outfile)
            writer.writerow([
                "File Name", "Prefetch Name", "File Size (bytes)",
                "File Create Date (UTC)", "File Modify Date (UTC)",
                "Prefetch Last Execution Date (UTC)",
                "Prefetch Execution Count", "Volume", "Volume Create Date",
                "Volume Serial", "File Path"           ])
            writer.writerows(data)
```

After running the above script, we will get the information from prefetch files of Windows XP version into a spreadsheet.

11. Python Digital Forensics – Important Artifacts in Windows-III

This chapter will explain about further artifacts that an investigator can obtain during forensic analysis on Windows.

Event Logs

Windows event log files, as name –suggests, are special files that stores significant events like when user logs on the computer, when program encounter an error, about system changes, RDP access, application specific events etc. Cyber investigators are always interested in event log information because it provides lots of useful historical information about the access of system. In the following Python script we are going to process both legacy and current Windows event log formats.

For Python script, we need to install third party modules namely **pytsk3, pyewf, unicodecsv, pyevt and pyevtx**. We can follow the steps given below to extract information from event logs:

- First, search for all the event logs that match the input argument.
- Then, perform file signature verification.
- Now, process each event log found with the appropriate library.
- Lastly, write the output to spreadsheet.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
import argparse
import unicodecsv as csv
import os
import pytsk3
import pyewf
import pyevt
import pyevtx
import sys
from utility.pytskutil import TSKUtil
```

Now, provide the arguments for command-line handler. Note that here it will accept three arguments – first is the path to evidence file, second is the type of evidence file and third is the name of the event log to process.

```
if __name__ == "__main__":
      parser = argparse.ArgumentParser('Information from Event Logs')
      parser.add_argument("EVIDENCE_FILE", help="Evidence file path")
      parser.add_argument("TYPE", help="Type of Evidence",choices=("raw", "ewf"))
      parser.add_argument("LOG_NAME",help="Event Log Name (SecEvent.Evt,
SysEvent.Evt, ""etc.)")
      parser.add_argument("-d", help="Event log directory to
scan",default="/WINDOWS/SYSTEM32/WINEVT")
      parser.add_argument("-f", help="Enable fuzzy search for either evt or"" evtx
extension", action="store_true")
      args = parser.parse_args()
      if os.path.exists(args.EVIDENCE_FILE) and \
                  os.path.isfile(args.EVIDENCE_FILE):
            main(args.EVIDENCE_FILE, args.TYPE, args.LOG_NAME, args.d, args.f)
      else:
            print("[-] Supplied input file {} does not exist or is not a
""file".format(args.EVIDENCE_FILE))
      sys.exit(1)
```

Now, interact with event logs toquery the existence of the user supplied path by creating our **TSKUtil** object. It can be done with the help of **main()** method as follows:

```
def main(evidence, image_type, log, win_event, fuzzy):
      tsk_util = TSKUtil(evidence, image_type)
      event_dir = tsk_util.query_directory(win_event)
      if event_dir is not None:
            if fuzzy is True:
                  event_log = tsk_util.recurse_files(log, path=win_event)
      else:
                  event_log = tsk_util.recurse_files(log, path=win_event,
logic="equal")
      if event_log is not None:
            event_data = []
            for hit in event_log:
                    event_file = hit[2]
                    temp_evt = write_file(event_file)
```

Now, we need to perform signature verification followed by defining a method that will write the entire content to the current directory:

```
def write_file(event_file):
    with open(event_file.info.name.name, "w") as outfile:
        outfile.write(event_file.read_random(0, event_file.info.meta.size))
    return event_file.info.name.name
        if pyevt.check_file_signature(temp_evt):
            evt_log = pyevt.open(temp_evt)
            print("[+] Identified {} records in
{}".format(evt_log.number_of_records, temp_evt))
            for i, record in enumerate(evt_log.records):
                strings = ""
                for s in record.strings:
                    if s is not None:
                        strings += s + "\n"
                event_data.append([
                    i, hit[0], record.computer_name,
                    record.user_security_identifier,
                    record.creation_time, record.written_time,
                    record.event_category, record.source_name,
                    record.event_identifier, record.event_type,
                    strings, "",
                    os.path.join(win_event, hit[1].lstrip("//"))
                ])
        elif pyevtx.check_file_signature(temp_evt):
            evtx_log = pyevtx.open(temp_evt)
            print("[+] Identified {} records in {}".format(
                evtx_log.number_of_records, temp_evt))
            for i, record in enumerate(evtx_log.records):
                strings = ""
                for s in record.strings:
                    if s is not None:
                        strings += s + "\n"
                event_data.append([
                    i, hit[0], record.computer_name,
                    record.user_security_identifier, "",
                    record.written_time, record.event_level,
                    record.source_name, record.event_identifier,
                    "", strings, record.xml_string,
                    os.path.join(win_event, hit[1].lstrip("//"))
                ])
                else:
                    print("[-] {} not a valid event log. Removing temp
"
                            "file...".format(temp_evt))
                    os.remove(temp_evt)
                    continue
            write_output(event_data)
        else:
            print("[-] {} Event log not found in {} directory".format(log,
win_event))
            sys.exit(3)
    else:
        print("[-] Win XP Event Log Directory {} not found".format(win_event))
        sys.exit(2)
```

Lastly, define a method for writing the output to spreadsheet as follows:

```
def write_output(data):
    output_name = "parsed_event_logs.csv"
    print("[+] Writing {} to current working directory: {}".format(
            output_name, os.getcwd()))
    with open(output_name, "wb") as outfile:
        writer = csv.writer(outfile)
        writer.writerow([
            "Index", "File name", "Computer Name", "SID",
            "Event Create Date", "Event Written Date",
            "Event Category/Level", "Event Source", "Event ID",
            "Event Type", "Data", "XML Data", "File Path"
    ])
        writer.writerows(data)
```

Once you successfully run the above script, we will get the information of events log in spreadsheet.

Internet History

Internet history is very much useful for forensic analysts; as most cyber-crimes happen over the internet only. Let us see how to extract internet history from the Internet Explorer, as we discussing about Windows forensics, and Internet Explorer comes by default with Windows.

On Internet Explorer, the internet history is saved in **index.dat** file. Let us look into a Python script, which will extract the information from **index.dat** file.

For Python script we need to install third party modules namely **pylnk, pytsk3, pymsiecf** and **unicodecsv**.

We can follow the steps given below to extract information from **index.dat** files:

- First, search for **index.dat** files within the system.
- Then, extract the information from that file by iterating through them.
- Now, write all this information to a CSV report.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
import argparse
from datetime import datetime, timedelta
import os
import pytsk3
import pyewf
import pymsiecf
import sys
import unicodecsv as csv
from utility.pytskutil import TSKUtil
```

Now, provide arguments for command-line handler. Note that here it will accept two arguments – first would be the path to evidence file and second would be the type of evidence file.

```
if __name__ == "__main__":
parser = argparse.ArgumentParser('getting information from internet history')
     parser.add_argument("EVIDENCE_FILE", help="Evidence file path")
     parser.add_argument("TYPE", help="Type of Evidence",choices=("raw", "ewf"))
     parser.add_argument("-d", help="Index.dat directory to
scan",default="/USERS")
     args = parser.parse_args()
     if os.path.exists(args.EVIDENCE_FILE) and os.path.isfile(args.EVIDENCE_FILE):
            main(args.EVIDENCE_FILE, args.TYPE, args.d)
     else:
            print("[-] Supplied input file {} does not exist or is not a
""file".format(args.EVIDENCE_FILE))
            sys.exit(1)
```

Now, interpret the evidence file by creating an object of **TSKUtil** and iterate through the file system to find **index.dat** files. It can be done by defining the **main()** function as follows:

```
def main(evidence, image_type, path):
     tsk_util = TSKUtil(evidence, image_type)
     index_dir = tsk_util.query_directory(path)
     if index_dir is not None:
            index_files = tsk_util.recurse_files("index.dat",
path=path,logic="equal")
            if index_files is not None:
                   print("[+] Identified {} potential index.dat
files".format(len(index_files)))
                   index_data = []
                   for hit in index_files:
                          index_file = hit[2]
                          temp_index = write_file(index_file)
```

Now, define a function with the help of which we can copy the information of index.dat file to the current working directory and later on they can be processed by a third party module:

```
  def write_file(index_file):
     with open(index_file.info.name.name, "w") as outfile:
            outfile.write(index_file.read_random(0, index_file.info.meta.size))
      return index_file.info.name.name
```

Now, use the following code to perform the signature validation with the help of the built-in function namely **check_file_signature()**:

```
                        if pymsiecf.check_file_signature(temp_index):
                              index_dat = pymsiecf.open(temp_index)
                              print("[+] Identified {} records in {}".format(
                                    index_dat.number_of_items, temp_index))
                              for i, record in enumerate(index_dat.items):
                                    try:
                                          data = record.data
                                          if data is not None:
                                          data = data.rstrip("\x00")


                                    except AttributeError:
                                          if isinstance(record,
pymsiecf.redirected):
                                          index_data.append([i, temp_index, "",
"", "", "", "",record.location, "", "", record.offset,os.path.join(path,
hit[1].lstrip("//"))])
                                          elif isinstance(record,
pymsiecf.leak):
                                          index_data.append([i, temp_index,
record.filename, "","", "", "", "", "", "", record.offset,os.path.join(path,
hit[1].lstrip("//"))])
                                          continue
                                    index_data.append([
                                          i, temp_index, record.filename,
                                          record.type, record.primary_time,
                                          record.secondary_time,
                                          record.last_checked_time,
record.location,
                                    record.number_of_hits, data, record.offset,
                                          os.path.join(path,
hit[1].lstrip("//"))
                                    ])
                        else:
                              print("[-] {} not a valid index.dat file. Removing
"
                                          "temp file..".format(temp_index))
                              os.remove("index.dat")
                              continue
                  os.remove("index.dat")
                  write_output(index_data)
            else:
                  print("[-] Index.dat files not found in {}
directory".format(path))
                  sys.exit(3)
      else:
            print("[-] Directory {} not found".format(win_event))
            sys.exit(2)
```

Now, define a method that will print the output in CSV file, as shown below:

```
 def write_output(data):
      output_name = "Internet_Indexdat_Summary_Report.csv"
      print("[+] Writing {} with {} parsed index.dat files to current "
                  "working directory: {}".format(output_name,
len(data),os.getcwd()))
      with open(output_name, "wb") as outfile:
            writer = csv.writer(outfile)
            writer.writerow(["Index", "File Name", "Record Name",
                                    "Record Type", "Primary Date", "Secondary
Date",
                                    "Last Checked Date", "Location", "No. of
Hits",
                                    "Record Data", "Record Offset", "File
Path"])
            writer.writerows(data)
```

After running above script we will get the information from index.dat file in CSV file.

Volume Shadow Copies

A shadow copy is the technology included in Windows for taking backup copies or snapshots of computer files manually or automatically. It is also called volume snapshot service or volume shadow service(VSS).

With the help of these VSS files, forensic experts can have some historical information about how the system changed over time and what files existed on the computer. Shadow copy technology requires the file system to be NTFS for creating and storing shadow copies.

In this section, we are going to see a Python script, which helps in accessing any volume of shadow copies present in the forensic image.

For Python script we need to install third party modules namely **pytsk3, pyewf, unicodecsv, pyvshadow** and **vss**. We can follow the steps given below to extract information from VSS files:

- First, access the volume of raw image and identify all the NTFS partitions.
- Then, extract the information from that shadow copies by iterating through them.
- Now, at last we need to create a file listing of data within the snapshots.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python libraries:

```
from __future__ import print_function
import argparse
from datetime import datetime, timedelta
import os
import pytsk3
import pyewf
import pyvshadow
import sys
import unicodecsv as csv
from utility import vss
from utility.pytskutil import TSKUtil
from utility import pytskutil
```

Now, provide arguments for command-line handler. Here it will accept two arguments – first is the path to evidence file and second is the output file.

```
if __name__ == "__main__":
     parser = argparse.ArgumentParser('Parsing Shadow Copies')
     parser.add_argument("EVIDENCE_FILE", help="Evidence file path")
     parser.add_argument("OUTPUT_CSV",
     help="Output CSV with VSS file listing")
     args = parser.parse_args()
```

Now, validate the input file path's existence and also separate the directory from output file.

```
     directory = os.path.dirname(args.OUTPUT_CSV)
     if not os.path.exists(directory) and directory != "":
            os.makedirs(directory)
     if os.path.exists(args.EVIDENCE_FILE) and \
                   os.path.isfile(args.EVIDENCE_FILE):
            main(args.EVIDENCE_FILE, args.OUTPUT_CSV)
     else:
            print("[-] Supplied input file {} does not exist or is not a "
                          "file".format(args.EVIDENCE_FILE))
            sys.exit(1)
```

Now, interact with evidence file's volume by creating the **TSKUtil** object. It can be done with the help of **main()** method as follows:

```
 def main(evidence, output):
     tsk_util = TSKUtil(evidence, "raw")
     img_vol = tsk_util.return_vol()
     if img_vol is not None:
            for part in img_vol:
                   if tsk_util.detect_ntfs(img_vol, part):
                   print("Exploring NTFS Partition for VSS")


                   explore_vss(evidence, part.start *
img_vol.info.block_size,output)
     else:
            print("[-] Must be a physical preservation to be compatible ""with
this script")
            sys.exit(2)
```

Now, define a method for exploring the parsed volume shadow file as follows:

```
def explore_vss(evidence, part_offset, output):
    vss_volume = pyvshadow.volume()
    vss_handle = vss.VShadowVolume(evidence, part_offset)
    vss_count = vss.GetVssStoreCount(evidence, part_offset)
    if vss_count > 0:
        vss_volume.open_file_object(vss_handle)
        vss_data = []
        for x in range(vss_count):
            print("Gathering data for VSC {} of {}".format(x, vss_count))
            vss_store = vss_volume.get_store(x)
            image = vss.VShadowImgInfo(vss_store)
            vss_data.append(pytskutil.openVSSFS(image, x))
        write_csv(vss_data, output)
```

Lastly, define the method for writing the result in spreadsheet as follows:

```
def write_csv(data, output):
    if data == []:
        print("[-] No output results to write")
        sys.exit(3)
    print("[+] Writing output to {}".format(output))
    if os.path.exists(output):
        append = True
    with open(output, "ab") as csvfile:
        csv_writer = csv.writer(csvfile)
        headers = ["VSS", "File", "File Ext", "File Type", "Create Date",
                   "Modify Date", "Change Date", "Size", "File Path"]
        if not append:
            csv_writer.writerow(headers)
        for result_list in data:
            csv_writer.writerows(result_list)
```

Once you successfully run this Python script, we will get the information residing in VSS into a spreadsheet.

12. Python Digital Forensics – Investigation of Log Based Artifacts

Till now, we have seen how to obtain artifacts in Windows using Python. In this chapter, let us learn about investigation of log based artifacts using Python.

Introduction

Log-based artifacts are the treasure trove of information that can be very useful for a digital forensic expert. Though we have various monitoring software for collecting the information, the main issue for parsing useful information from them is that we need lot of data.

Various Log-based Artifacts and Investigating in Python

In this section, let us discuss various log based artifacts and their investigation in Python:

Timestamps

Timestamp conveys the data and time of the activity in the log. It is one of the important elements of any log file. Note that these data and time values can come in various formats.

The Python script shown below will take the raw date-time as input and provides a formatted timestamp as its output.

For this script, we need to follow the following steps:

- First, set up the arguments that will take the raw data value along with source of data and the data type.

- Now, provide a class for providing common interface for data across different date formats.

Python Code

Let us see how to use Python code for this purpose:

First, import the following Python modules:

```
from __future__ import print_function
from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter
from datetime import datetime as dt
from datetime import timedelta
```

Now as usual we need to provide argument for command-line handler. Here it will accept three arguments, first would be the date value to be processed, second would be the source of that date value and third would be its type.

```
if __name__ == '__main__':
     parser = ArgumentParser('Timestamp Log-based artifact')
     parser.add_argument("date_value", help="Raw date value to parse")
     parser.add_argument("source", help="Source format of
date",choices=ParseDate.get_supported_formats())
     parser.add_argument("type", help="Data type of input
value",choices=('number', 'hex'), default='int')
     args = parser.parse_args()
     date_parser = ParseDate(args.date_value, args.source, args.type)
     date_parser.run()
     print(date_parser.timestamp)
```

Now, we need to define a class which will accept the arguments for date value, date source, and the value type.

```
 class ParseDate(object):
     def __init__(self, date_value, source, data_type):
            self.date_value = date_value
            self.source = source
            self.data_type = data_type
            self.timestamp = None
```

Now we will define a method that will act like a controller just like the **`main()`** method:

```
     def run(self):
            if self.source == 'unix-epoch':
                   self.parse_unix_epoch()
            elif self.source == 'unix-epoch-ms':
                   self.parse_unix_epoch(True)
            elif self.source == 'windows-filetime':
                   self.parse_windows_filetime()
     @classmethod
     def get_supported_formats(cls):
            return ['unix-epoch', 'unix-epoch-ms', 'windows-filetime']
```

Now, we need to define two methods which will process Unix epoch time and **FILETIME** respectively:

```
    def parse_unix_epoch(self, milliseconds=False):
        if self.data_type == 'hex':
            conv_value = int(self.date_value)
            if milliseconds:
                conv_value = conv_value / 1000.0
        elif self.data_type == 'number':
            conv_value = float(self.date_value)
            if milliseconds:
                conv_value = conv_value / 1000.0
        else:
            print("Unsupported data type '{}'
provided".format(self.data_type))
            sys.exit('1')
        ts = dt.fromtimestamp(conv_value)
        self.timestamp = ts.strftime('%Y-%m-%d %H:%M:%S.%f')
    def parse_windows_filetime(self):
        if self.data_type == 'hex':
            microseconds = int(self.date_value, 16) / 10.0
        elif self.data_type == 'number':
            microseconds = float(self.date_value) / 10
        else:
            print("Unsupported data type '{}'
provided".format(self.data_type))
            sys.exit('1')
        ts = dt(1601, 1, 1) + timedelta(microseconds=microseconds)
        self.timestamp = ts.strftime('%Y-%m-%d %H:%M:%S.%f')
```

After running the above script, by providing a timestamp we can get the converted value in easy-to-read format.

Web Server Logs

From the point of view of digital forensic expert, web server logs are another important artifact because they can get useful user statistics along with information about the user and geographical locations. Following is the Python script that will create a spreadsheet, after processing the web server logs, for easy analysis of the information.

First of all we need to import the following Python modules:

```
from __future__ import print_function
from argparse import ArgumentParser, FileType
import re
import shlex
import logging
import sys
import csv
logger = logging.getLogger(__file__)
```

Now, we need to define the patterns that will be parsed from the logs:

```
iis_log_format = [
     ("date", re.compile(r"\d{4}-\d{2}-\d{2}")),
     ("time", re.compile(r"\d\d:\d\d:\d\d")),
     ("s-ip", re.compile(
            r"((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}")),
     ("cs-method", re.compile(
            r"(GET)|(POST)|(PUT)|(DELETE)|(OPTIONS)|(HEAD)|(CONNECT)")),
     ("cs-uri-stem", re.compile(r"([A-Za-z0-1/\.-]*)")),
     ("cs-uri-query", re.compile(r"([A-Za-z0-1/\.-]*)")),
     ("s-port", re.compile(r"\d*")),
     ("cs-username", re.compile(r"([A-Za-z0-1/\.-]*)")),
     ("c-ip", re.compile(
            r"((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}")),
     ("cs(User-Agent)", re.compile(r".*")),
     ("sc-status", re.compile(r"\d*")),
     ("sc-substatus", re.compile(r"\d*")),
     ("sc-win32-status", re.compile(r"\d*")),
     ("time-taken", re.compile(r"\d*"))]
```

Now, provide an argument for command-line handler. Here it will accept two arguments, first would be the IIS log to be processed, second would be the desired CSV file path.

```
if __name__ == '__main__':
     parser = ArgumentParser('Parsing Server Based Logs')
     parser.add_argument('iis_log', help="Path to IIS Log",type=FileType('r'))
     parser.add_argument('csv_report', help="Path to CSV report")
     parser.add_argument('-l', help="Path to processing log",default=__name__ +
'.log')
     args = parser.parse_args()
     logger.setLevel(logging.DEBUG)
     msg_fmt = logging.Formatter("%(asctime)-15s %(funcName)-10s ""%(levelname)-8s
%(message)s")
     strhndl = logging.StreamHandler(sys.stdout)
     strhndl.setFormatter(fmt=msg_fmt)
     fhndl = logging.FileHandler(args.log, mode='a')
     fhndl.setFormatter(fmt=msg_fmt)
     logger.addHandler(strhndl)
     logger.addHandler(fhndl)
     logger.info("Starting IIS Parsing ")
     logger.debug("Supplied arguments: {}".format(", ".join(sys.argv[1:])))
     logger.debug("System " + sys.platform)
     logger.debug("Version " + sys.version)
     main(args.iis_log, args.csv_report, logger)
     iologger.info("IIS Parsing Complete")
```

Now we need to define **`main()`** method that will handle the script for bulk log information:

```
def main(iis_log, report_file, logger):
     parsed_logs = []
          for raw_line in iis_log:
               line = raw_line.strip()
               log_entry = {}
               if line.startswith("#") or len(line) == 0:
                    continue
               if '\"' in line:
                    line_iter = shlex.shlex(line_iter)
```

```
            else:
                line_iter = line.split(" ")
            for count, split_entry in enumerate(line_iter):
                col_name, col_pattern = iis_log_format[count]
                if col_pattern.match(split_entry):
                    log_entry[col_name] = split_entry
                else:
                    logger.error("Unknown column pattern discovered. "
                                "Line preserved in full below")
                    logger.error("Unparsed Line: {}".format(line))
            parsed_logs.append(log_entry)
    logger.info("Parsed {} lines".format(len(parsed_logs)))
    cols = [x[0] for x in iis_log_format]
    logger.info("Creating report file: {}".format(report_file))
    write_csv(report_file, cols, parsed_logs)
    logger.info("Report created")
```

Lastly, we need to define a method that will write the output to spreadsheet:

```
def write_csv(outfile, fieldnames, data):
    with open(outfile, 'w', newline="") as open_outfile:
        csvfile = csv.DictWriter(open_outfile, fieldnames)
        csvfile.writeheader()
        csvfile.writerows(data)
```

After running the above script we will get the web server based logs in a spreadsheet.

Scanning Important Files using YARA

YARA(Yet Another Recursive Algorithm) is a pattern matching utility designed for malware identification and incident response. We will use YARA for scanning the files. In the following Python script, we will use YARA.

We can install YARA with the help of following command:

```
pip install YARA
```

We can follow the steps given below for using YARA rules to scan files:

- First, set up and compile YARA rules
- Then, scan a single file and then iterate through the directories to process individual files.
- Lastly, we will export the result to CSV.

Python Code

Let us see how to use Python code for this purpose:

First, we need to import the following Python modules:

```
from __future__ import print_function

from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter

import os

import csv

import yara
```

Next, provide argument for command-line handler. Note that here it will accept two arguments – first is the path to YARA rules, second is the file to be scanned.

```
if __name__ == '__main__':

     parser = ArgumentParser('Scanning files by YARA')

     parser.add_argument('yara_rules',help="Path to Yara rule to scan with. May be
file or folder path.")

     parser.add_argument('path_to_scan',help="Path to file or folder to scan")

     parser.add_argument('--output',help="Path to output a CSV report of scan
results")

     args = parser.parse_args()

     main(args.yara_rules, args.path_to_scan, args.output)
```

Now we will define the **`main()`** function that will accept the path to the **`yara`** rules and file to be scanned:

```
def main(yara_rules, path_to_scan, output):
    if os.path.isdir(yara_rules):
        yrules = yara.compile(yara_rules)
    else:
        yrules = yara.compile(filepath=yara_rules)
    if os.path.isdir(path_to_scan):
        match_info = process_directory(yrules, path_to_scan)
    else:
        match_info = process_file(yrules, path_to_scan)
    columns = ['rule_name', 'hit_value', 'hit_offset', 'file_name',
                        'rule_string', 'rule_tag']
    if output is None:
        write_stdout(columns, match_info)
    else:
        write_csv(output, columns, match_info)
```

Now, define a method that will iterate through the directory and passes the result to another method for further processing:

```
def process_directory(yrules, folder_path):
    match_info = []
    for root, _, files in os.walk(folder_path):
        for entry in files:
            file_entry = os.path.join(root, entry)
            match_info += process_file(yrules, file_entry)
    return match_info
```

Next, define two functions. Note that first we will use **match()** method to **yrules** object and another will report that match information to the console if the user does not specify any output file. Observe the code shown below:

```
def process_file(yrules, file_path):
    match = yrules.match(file_path)
    match_info = []
    for rule_set in match:
        for hit in rule_set.strings:
            match_info.append({
```

```
                          'file_name': file_path,
                          'rule_name': rule_set.rule,
                          'rule_tag': ",".join(rule_set.tags),
                          'hit_offset': hit[0],
                          'rule_string': hit[1],
                          'hit_value': hit[2]
                    })
     return match_info
def write_stdout(columns, match_info):
     for entry in match_info:
           for col in columns:
                 print("{}: {}".format(col, entry[col]))
           print("=" * 30)
```

Lastly, we will define a method that will write the output to CSV file, as shown below:

```
def write_csv(outfile, fieldnames, data):
     with open(outfile, 'w', newline="") as open_outfile:
           csvfile = csv.DictWriter(open_outfile, fieldnames)
           csvfile.writeheader()
           csvfile.writerows(data)
```

Once you run the above script successfully, we can provide appropriate arguments at the command-line and can generate a CSV report.