Forensics Python

- Forensic File Carving
- Processing Data with Struct
- Python`s Image Library
- Database Operations
  - Structured Query Language

Forensic with Python

- Forensic is one of the most rapidly changing fields
  - New artifacts discovered in old operating systems
  - New artifacts created by new applications and operating system features
- If you want to make your mark as a new tool developer,there is a HUGE opportunity in Forensics
- SANS FOR500-Windows Forensics and FOR508-Advanced Digital Forensics are great to learn of new techniques and artifacts that need tools written
- The Cuckoo's Egg

Forensics Artifact Carving

- Data Stream Carving
  - Text from chat sessions in Sqlite3 database
  - Commands typed at CMD.EXE prompts from memory
  - Passwords, session-negotiated encryption keys from disk swap space/page files
- File Carving
  - Images such as JPG,GIF,and so on
  - Documents such as DOC,DOCX,XLS,and so on
  - Media such as MP3,MOV,WMV,and so on

Carving Forensic Artifacts

- At a high level, the steps for doing this are the same for all types of data sources
- We will cover the following four steps:

1) Getting read access to the data
2) Understand the “Metadata” structures that organizes/breaks up your target data and extracts your data
    - Hard drives:Directory structures containing files such as MFTM,Block headers, etc.
    - Memory:Paging system, OS Data Structures,etc.
    - Network:PCAP headers, Frame headers, etc.
    - Unknown Structures:Cover Channels,Malware,etc.
3) Extract relevant parts with a regular expression
4) Analyze the data

Forensic File Carving

- On both Linux and Windows, the hard drives can be treated as a (very large) file and read using standard file I/0
- Linux
  - “/dev/sda”:First physical drive
  - “/dev/sda1”:First logical drive on first physical drive

```
>>> fh = open("/dev/sda1")
>>> fh.read(80)
'\xebX\x90mkfs.fat\x00\x02\x08 \x00\x02\x00\x00\x00\x00\xf8\x00\x00?\x00\xff\x00
\x00\x08\x00\x00\x00\x00\x10\x00\x00\x04\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00
\x01\x00\x06\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x01)ua\x91\
x82NO NAME   '
```

- Windows
  - \\.\PhysicalDriveo:First physical drive
  - \\.\C:Contents of logical drive C:
-

Live Memory Carving

- You can carve artifacts from live memory
  - On windows, you use Winpmem, which is part of Rekall
  - Winpmem.exe creates a file called \\.\pmem that will give you access to live memory
- https://isc.sans.edu/diary/Searching+live+memory+on+a+running+machine+with+winpmem/17063
- https://www.dshield.org/diary/%22In+the+end+it+is+all+PEEKS+and+POKES.%22/17069
- On Linux Kernels 2.6 and later, /dev/mem less than reliable but can be used
- Installing third-party kernel extension FMEM will work modern Linux systems

Linux Live Network Capture(sniffing)

- Linux “raw sockets” enable you to sniff everything promiscuously
- The socket options determine what types of packets you see
  - **Capture EVERYTHING**

    socket.socket(socket.AF_PACKET, socket.SOCK_RAW,socket.ntohs(0x0003))
  - **Capture TCP**

    socket.socket(socket.AF_INET, socket.SOCK_RAW,socket.IPPROTO_TCP)
  - **Capture UDP**
  - socket.socket(socket.AF_INET, socket.SOCK_RAW,socket.IPPROTO_UDP)
  - **Capture ICMP**
  - socket.socket(socket.AF_INET, socket.SOCK_RAW,socket.IPPROTO_ICMP)

Linux Live Network Capture(sniffing)

- For example:

```
>>> import socket
>>> s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,socket.ntohs(0x0003))
>>> while True:
...  print(s.recv(65535))
...
b'4\xe1-\xe9\x8d8\x00\x1d\xaa\x9f\xd1\xbc\x08\x00E\x00\x004\x1aP\x00\x00x\x06\xc
5\x8c\xac\xd9\xa0n\xac\x10i\x8f\x01\xbb\x8a\x92\xa7\x8b\x90\xd8\xd3\x982g\x80\x1
0\x04\x1a\x92\xb7\x00\x00\x01\x01\x08\nx4 c\x06X\x13c'
b'4\xe1-\xe9\x8d8\x00\x1d\xaa\x9f\xd1\xbc\x08\x00E\x00\x00q\xc9\xcc\x00\x00k\x06
\xa1\xacl\xb1a\xbd\xac\x10i\x8f\x01\xbb\xa3\x80\x8c\xd7\xf3y\x1a)\xc6\xd6\x80\x1
8\x02&\xd33\x00\x00\x01\x01\x08\n@\x0f\x02\xd2\x1er\xecF\x17\x03\x03\x008\x00\x0
0\x00\x00\x00\x00\x01g\x83G\xb4\xac\\\xb4\xfd\xac\x07i\xf1\x1d\xb2\xf3\x13\xb4uw
Cc\xda\xb4\xf0\xbf\x03\xbbx\xc5\xca\xd62*\xb4\xc3\x0c\xac\x19\xa0|\xae\x1c\x12u\
```

Understanding the Structure

- You can parse a PCAP by just using standard file I/O

```
network_packets = open(“test.pcap”).read()
```

- But, you have to understand the pcap file data structure and where the data is scattered across the file`s data structures
-

PCAP HEADER	Ethernet HEADER	IP HEADER	TCP HEADER	MY DATA PART1	Ethernet HEADER	ICMP HEADER	OTHER DATA	Ethernet HEADER	IP HEADER	TCP HEADER	MY DATA PART2

- It is easier for you if you use a library like scapy that has rdpcap(),tcp() and so on, which already understands that structure!

- The same is true when dealing with all the previous LIVE captures OR a static forensics artifact such as a disk dump or a memory capture

FILE SYSTEM	Directory Informatio n	Sector HEADER	Cluster HEADER	MY DATA PART1	Cluster HEADER	OTHER DATA	Cluster HEADER	MY DATA PART2

Third-Party Modules

- There are third-party modules that understand this
  - Hard Drives:Plaso, GRR,AnalyzeMFT
  - Memory:Rekall, Volatility
  - Network:DPKT,Scapy
  - Document:pyPDF,zipfile
- If these modules exist for your data structure, then you should use them!
- But, what do you do when they don`t exist?
- Parse the data structures yourself as a plug-in to one of those modules or as a standalone carver

Alert on Unknown Unknowns

- Consider this example of parsing network communications

- Bad!!

```
for eachpacket in listofpackets:
    if eachpacket[proto] ==’icmp’:
        do important analysis
    if eachpacket[proto] ==’tcp’:
        do important analysis
```

- Good!

```
for eachpacket in listofpackets:
    if eachpacket[proto] ==’icmp’:
        do important analysis
    elif eachpacket[proto] ==’tcp’:
        do important analysis
    else:
        print(“WARNING: UNKNOWN PROTOCOL”)
```

The STRUCT Module

- The struct module is used for interpreting binary data
- Converts a string of binary to a tuple of integers and strings
- struct.unpack(): Similar to regex, you create a string that says how you want to extract the data
- That string is created to match the format of the data you are trying to extract
- !BBBB = 4 個 1byte INTs
- The struct format string:
  - First character indicates little or big endian
    - ! or > indicates to interpret data as BIG ENDIAN
    - < indicates to interpret data as LITTLE ENDIAN
    - = or @ indicates to interpret data based on the system it script is running on (I.e.sys.byteorder)
  - REMAINING characters indicate how to interpret data

```
>>> import struct
>>> struct.unpack('!BBBB',"\xc0\xa8\x80\xc2")
(192, 168, 128, 194)
```

```
>>> import sys
>>> sys.byteorder
'little'
```

Struct Format Characters

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	bytes of length 1	1	
b	signed char	integer	1	(1),(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
n	ssize_t	integer		(4)
N	size_t	integer		(4)
e	(7)	float	2	(5)
f	float	float	4	(5)
d	double	float	8	(5)
s	char[]	bytes		
p	char[]	bytes		
P	void *	integer		(6)

https://docs.python.org/3/library/struct.html

Struct Unpack(I)

```
>>> import struct
>>> struct.unpack(">BB","\xff\x00")
(255, 0)
>>> struct.unpack("<BB","\xff\x00")
(255, 0)
>>> struct.unpack("<bB","\xff\x00")
(-1, 0)
>>> struct.unpack("<H","\xff\x00")
(255,)
>>> struct.unpack(">H","\xff\x00")
(65280,)
>>> struct.unpack(">h","\xff\x00")
(-256,)
>>> struct.unpack("2s","\xff\x00")
('\xff\x00',)
```

Struct Unpack(II)

```
>>> struct.unpack("<cccc","\x01\x41\x42\x43")
('\x01', 'A', 'B', 'C')
>>> struct.unpack("<4c","\x01\x41\x42\x43")
('\x01', 'A', 'B', 'C')
>>> struct.unpack("<4B","\x01\x41\x42\x43")
(1, 65, 66, 67)
>>> struct.unpack("<BxxB","\x01\x02\x03\x04")
(1, 4)
>>> struct.unpack("<B2xB","\x01\x02\x03\x04")
(1, 4)
>>> struct.unpack("<I","\x01\x02\x03\x04")
(67305985,)
>>> struct.unpack("<5c","\x48\x45\x4c\x4c\x4f")
('H', 'E', 'L', 'L', 'O')
>>> struct.unpack("<5s","\x48\x45\x4c\x4c\x4f")
('HELLO',)
>>> struct.unpack("@17p","\x07\x48\x65\x6c\x6c\x6f\x20\x68\x6f\x77\x20\x61\x72\x
65\x20\x79\x6f")
('Hello h',)
```

Pascal String “Hello how are yo”

Unpacking Bits as Flags

- A binary bit is often used as flag. For example, the TCP SYN flag is represented by the 2nd bit in byte 13 of the TCP header
- itertools.compress() will convert SET bits to words

```
>>> import itertools
>>> list(itertools.compress(["BIT0",'BIT1','BIT2'],[1,0,1]))
['BIT0', 'BIT2']
```

- You need bits as a list of integers

```
>>> format(147,"08b")
'10010011'
>>> list(map(int,format(147,"08b")))
[1, 0, 0, 1, 0, 0, 1, 1]
```

- Combine these two techniques to convert byte flags to words

```
>>> def tcp_flags_as_str(flag):
...     tcp_flags=['CWR','ECE','URG','ACK','PSH','RST','SYN','FIN']
...     return "|".join(list(itertools.compress(tcp_flags,map(int,format(flag,"08b")))))
```

Ether Header Struct

- Let`s make a struct string to capture the Ethernet header
- All network traffic is BIG ENDIAN, so it will start with a!

Destination MAC(6 bytes)	
DST MAC continued	SRC MAC(6 bytes)
Source MAC continued	
Ether Type(2 bytes)	

```
>>> import socket,struct,codecs
>>> s=socket.socket(socket.AF_PACKET, socket.SOCK_RAW,socket.ntohs(0x0003))
>>> while True:
...  data = s.recv(65535)
...  try:
...    eth_src, eth_dst,eth_type = struct.unpack("!6s6sH",data[:14])
...    print("ETH: SRC:{0} DST:{1} TYPE:{2}".format(codecs.encode(eth_src,"hex"),codecs.encode(eth_d
st,"hex"),hex(eth_type)))
...  except:
...    print("error")
...
ETH: SRC:b'001daa9fd1bc' DST:b'34e12de98d38' TYPE:0x800
ETH: SRC:b'34e12de98d38' DST:b'001daa9fd1bc' TYPE:0x800
```

= !6s6sH

IP Header Struct

<table>
<tr><td>Vers</td><td>Hlen</td><td>SVC(1 byte)</td><td colspan="4">Total Length(2 bytes)</td></tr>
<tr><td colspan="3">Identification(2 bytes)</td><td>unused</td><td>DF</td><td>MF</td><td>Fragment Offset</td></tr>
<tr><td>TTL(1)</td><td colspan="2">Protocol(1)</td><td colspan="4">Header Checksum(2 bytes)</td></tr>
<tr><td colspan="7">Source IP Address(4 bytes)</td></tr>
<tr><td colspan="7">Destination IP Address (4 bytes)</td></tr>
<tr><td colspan="7">IP Options(if any 4 byte boundaries)</td></tr>
</table>

```
>>> import socket,struct
>>> s=socket.socket(socket.AF_PACKET, socket.SOCK_RAW,socket.ntohs(0x0003))
>>> while True:
...  data = s.recv(65535)
...  iph = struct.unpack('!BBHHHBBHLL',data[14:34])
...  srcip = socket.inet_ntoa(struct.pack('!L',iph[8]))
...  dstip = socket.inet_ntoa(struct.pack('!L',iph[9]))
...  print("IP: SRC:{0} DST:{1}-{2}".format(srcip,dstip,iph))
...
IP: SRC:172.16.105.131 DST:172.16.105.255-(69, 0, 78, 10983, 0, 128, 17, 58388, 2886756739, 2886756863)
IP: SRC:172.16.105.131 DST:172.16.105.255-(69, 0, 78, 10984, 0, 128, 17, 58387, 2886756739, 2886756863)
IP: SRC:172.16.105.131 DST:172.16.105.255-(69, 0, 78, 10985, 0, 128, 17, 58386, 2886756739, 2886756863)
```

= !BBHHHBBHLL

TCP Header Struct

Source Port(2 bytes)			Destination Port(2 bytes)
Sequence Number(4 bytes)			
Acknowledgment Number(4 bytes)			
Hlen	Rsvd	Flags(1 byte)	Window(2 bytes)
Checksum(2 bytes)		Urgent Pointer(2 bytes)	
TCP Options(if any 4-byte boundaries)			

```
>>> import socket,struct
>>> s=socket.socket(socket.AF_INET, socket.SOCK_RAW,socket.IPPR
>>> while True:
...   data = s.recv(65535)
...   tcp=struct.unpack("!HHIIBBHHH",data[:20])
...   print("TCP: ",tcp)
...
('TCP: ', (17664, 1500, 851378176, 2064049094, 202, 39, 17359, 44048, 27023))
('TCP: ', (17664, 1500, 851443712, 2064049093, 202, 39, 17359, 44048, 27023))
('TCP: ', (17664, 1500, 851509248, 2064049092, 202, 39, 17359, 44048, 27023))
```

= !HHIIBBHHH

UDP Header Struct

Source Port(2 bytes)	Destination Port(2 bytes)
Total Length(2 bytes)	UDP Checksum(2 bytes)

= !HHHH

(sport,dport,len,chksum) = struct.unpack(“!HHHH”,ip_payload_data[:8])

- The UDP header has four uniform 2-byte components and is trivially easy to unpack

```
>>> import socket,struct
>>> s=socket.socket(socket.AF_INET, socket.SOCK_RAW,socket.IPPROTO_UDP)
>>> while True:
...  ip_payload_data = s.recv(65535)
...  print(struct.unpack('!HHHH',ip_payload_data[:8]),ip_payload_data[8:])
...
(17664, 56, 55834, 0) b'\x01\x11\xe9\x07\xac\x10i\x87\xe0\x00\x00\xfb\x14\xe9\x14\xe9\x00$\xca\x8e\x00\x
00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x04wpad\x05local\x00\x00\x01\x00\x01'
(17664, 56, 55836, 0) b'\x01\x11\xe9\x05\xac\x10i\x87\xe0\x00\x00\xfb\x14\xe9\x14\xe9\x00$\xca\x8e\x00\x
00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x04wpad\x05local\x00\x00\x01\x00\x01'
```

ICMP Header Struct

Type(1 byte)	Code(1 byte)	Checksum(2 bytes)
ICMP Data(4 or more bytes)		

- ICMP is not a very complex structure either
  - Type and Code are each 1 byte
  - Checksum is 2 bytes
  - Data is 4 bytes or more
- The TYPE code indicates what kind of ICMP traffic
  - If type ==8,then it is a “**PING REQUEST**”
  - If type ==0,then it is a “**PING REPLY**”
  - If type ==3,then it is a “**Unreachable**” and the contents of the code field say why it can`t be reached

= ?????

Use Regex to Get Data

- JPEG(**Joint Photographic Experts Group**) image is well-defined file format. It has markers throughout the file to indicate blocks of data
- Wikipedia has good reference on the JPEG file standard
  - https://en.wikipedia.org/wiki/JPEG#The_JPEG_standard
- JPEG images start with a hex magic value of \xff\xd8 and end of \xff\xd9
- We can use regex to GREEDILY match from \xff\xd8 until \xff\xd9

```
Def string2jpg(rawstring):
    if not '\xff\xd8' in rawstring or not '\xff\xd9' in rawstring:
        print("ERROR : Invalid or corrupt image! ",rawstring[:100])
        return None
    jpg = re.findall(r'\xff\xd8 .* \xff\xd9',rawstring,reDOTALL)[0]
    return jpg
```

Analyzing the Data

- You can write a parser and manually interact with the document/artidact
- You can use a third-party module to analyze it
  - ZIP:pyzip
  - PDF:pypdf,pdf-parder.py, PDFMiner
  - Office Doc: PyWin32 and COM
  - Office Docx:Extract zip and XML
  - Media(JPG,MOV):PIL,PyMedia,OpenCV,pySWF
  - EXE,DLL:pefile
- Find them with PIP!

Installing the PILLLOW Image Package

- PILLOW is current maintained fork, It is 100% backward compatible and has new features.
- PILLOW is not included with the standard libraries. It must be downloaded and installed.
- Free download here:https://python-pillow.org/

```
pip3 install pillow
```

- Features include:
  - READ and WRITE images from disk
  - Crop,resize,rotate and otherwise manipulate the images
  - Read/write image metadata
  - Support multiple image formats include JPG,BMP,TGA, more

Opening Images with PILLOW

- You create a PILLOW image object with the Image.open() method

```
>>> from PIL import Image
>>> imagedata = Image.open("./IMG_0007.JPG")
>>> imagedata.show()
```

- But, if your images aren`t on a disk? What if they are stores in a variable?
- You have two easy options:
  - Option1: write the variable to disk and then open it from disk with the PILLOW
  - Option2: use the cStringIO.StringIO(Python2) or io.BytesIO and io.StringIO(Python3) libraries to treat the bytes of string as an open file object
- Option 2 will be much faster because it doesn`t require any disk IO

Listing Metadata(1)

- Image._getexif() will return a dictionary full of the Exif information that was extracted from the image
- The KEYS in the dictionary are EXIF tags
- Exif TAGS are standardized integers that are assigned to specific data types
- Exif TAG 271 contains the MAKE of the camera
- Exif TAG 272 contains the MODEL of the camera
- Exif TAG 34853 contains GPS information about the photo!

```
>>> from PIL import Image
>>> imagedata = Image.open("./IMG_0007.JPG")
>>> info=imagedata._getexif()
>>> print(info[272])
iPhone 6 Plus
>>> print(info[271])
Apple
```

Listing Metadata(2)

```
>>> from PIL.ExifTags import TAGS
>>> for name,data in info.items():
...     tagname=TAGS.get(name,"unknown-tag")
...     print("TAG:%s (%s)" % (name,tagname) )
...
TAG:36864 (ExifVersion)
TAG:37121 (ComponentsConfiguration)
TAG:37378 (ApertureValue)
TAG:36867 (DateTimeOriginal)
TAG:36868 (DateTimeDigitized)
TAG:41989 (FocalLengthIn35mmFilm)
TAG:40960 (FlashPixVersion)
TAG:37383 (MeteringMode)
TAG:37385 (Flash)
TAG:37386 (FocalLength)
TAG:40962 (ExifImageWidth)
TAG:271 (Make)
TAG:272 (Model)
TAG:37521 (SubsecTimeOriginal)
```

Listing Metadata(2)

```
from PIL import Image
def getGPS(imobject):
    info=imobject._getexif()
    latDegrees =  info[34853][2][0][0]/float(info[34853][2][0][1])
    latDegrees += info[34853][2][1][0]/float(info[34853][2][1][1])/60
    latDegrees += info[34853][2][2][0]/float(info[34853][2][2][1])/3600
    lonDegrees =  info[34853][4][0][0]/float(info[34853][4][0][1])
    lonDegrees += info[34853][4][1][0]/float(info[34853][4][1][1])/60
    lonDegrees += info[34853][4][2][0]/float(info[34853][4][2][1])/3600
    if(info[34853][1]=='S'):
         latDegrees *= -1
    if(info[34853][4]=='W'):
         lonDegrees *= -1
    return latDegrees,lonDegrees

imobject=Image.open("./IMG_0007.JPG")
lat,lon=getGPS(imobject)

print(lon,lat)
print("http://maps.google.com/maps?q=%.9f,%.9f&z=15" %(lat,lon))
```

Python SQL Database Modules

- Python modules exist for most common SQL formats; they enable you to log in and select data directly from the tables in the database
  - MySQL; Many including mysql-connector-python, pyMysql,MySQL-Python, and more
  - MSSQL:pymssql,pytds
  - SQLITE:sqlite3 is built into Python
  - Oracle: sqlpython,cx_Oracle

Sqlite3 Connect and Retrieve Table and Column Names

- .connect(<path>):Open a file from local drive

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
```

- .execute(): Run an SQL statement!
  - Sqlite3 keeps its schema with table names in sqlite_master

```
>>> list(db.execute("select name from sqlite_master where type='table';"))
[('meta',), ('urls',), ('sqlite_sequence',), ('visits',), ('visit_source',), ('keyword_search_terms',),
('downloads',), ('downloads_url_chains',), ('downloads_slices',), ('segments',), ('segment_usage',), ('t
yped_url_sync_metadata',)]
```

- The SQL field has the statement that created the table with the column names

```
>>> list(db.execute("select sql from sqlite_master where name='urls';"))
[('CREATE TABLE urls(id INTEGER PRIMARY KEY AUTOINCREMENT,url LONGVARCHAR,title LONGVARCHAR,visit_count
INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,last_visit_time INTEGER NOT NULL,hidde
n INTEGER DEFAULT 0 NOT NULL)',)]
```

Sqlite3 Query the Record from the Database

- Use a for loop to iterate through rows

```
>>> import sqlite3
>>> db = sqlite3.connect("./History")
>>> for eachrow in db.execute("SELECT urls.id, urls.url,urls.title,urls.visit_count,urls.typed_count,urls.last_visit_time,urls.hidden FROM urls,vi
sits WHERE urls.id=visits.url;"):
...  print(eachrow)
...
```

```
(8113, 'http://giantdorks.org/alain/export-chrome-or-chromium-browsing-history-on-linux/', 'Alain Kelder | Export Chrome (or Chromium) browsing hi
story on Linux', 1, 0, 13198737425511091, 0)
(8114, 'https://superuser.com/questions/261998/which-folder-chrome-stores-my-history-in-linux', 'ubuntu - Which folder Chrome stores my history in
 Linux? - Super User', 1, 0, 13198737607315308, 0)
>>>
```

Thank you for your attention