Open source intelligence techniques & commentary

Vacuuming Image Metadata from The Wayback Machine

Written by **Justin,** December 6th, 2016

Not long ago I was intrigued by the Oct282011.com Internet mystery (if you haven't heard of it check out this podcast). Friends of the Hunchly mailing list and myself embarked on a brief journey to see if we could root out any additional clues or, of course, solve the mystery. One of the major sources of information for the investigation was The Wayback Machine, which is a popular resource for lots of investigations.

For this particular investigation there were a lot weird images strewn around as clues, and I wondered if it would be possible to retrieve those photos from the Wayback Machine and then examine them for EXIF data to see if we could find authorship details or other tasty nuggets of information. Of course I was not going to do this manually, so I thought it was a perfect opportunity to build out a new tool to do it for me.

We are going to leverage a couple of great tools to make this magic happen. The first is a Python module written by Jeremy Singer-Vine called waybackpack. While you can use waybackpack on the commandline as a standalone tool, in this blog post we are going to simply import it and leverage pieces of it to interact with the Wayback Machine. The second tool is ExifTool, by Phil Harvey. This little beauty is the gold standard when it comes to extracting EXIF information from photos and is trusted the world over.

The goal is for us to pull down all images for a particular URL on the Wayback Machine, extract any EXIF data and then output all of the information into a spreadsheet that we can then go and review.

Let's get rocking.

Prerequisites

This post involves a few moving parts, so let's get this boring stuff out of the way first.

Installing Exiftool

On Ubuntu based Linux you can do the following:

sudo apt-get install exiftool

Mac OSX users can use Phil's installer here.

For you folks on Windows you will have to do the following:

- Download the ExifTool binary from here. Save it to your C:\Python27 directory (you DO have Python installed right?)

- Rename it to exiftool.exe
- Make sure that C:\Python27 is in your Path. Don't know how to do this? Google will help. Or just email me.

Installing The Necessary Python Libraries

Now we are ready to install the various Python libraries that we need:

pip install bs4 requests pandas pyexifinfo waybackpack

Alright let's get down to it shall we?

Coding It Up

Now crack open a new Python file, call it ***waybackimages.py*** (download the source here) and start pounding out (use both hands) the following:

```
1  import bs4
2  import hashlib
3  import json
4  import os
5  import pandas
6  import pyexifinfo
7  import requests
8  import sys
9  import urlparse
10 import waybackpack
11
12 # ensure to place the trailing / for base domains
13 url = "http://www.oct282011.com/"
14
15 reload(sys)
16 sys.setdefaultencoding("utf-8")
17
18 if not os.path.exists("waybackimages"):
19     os.mkdir("waybackimages")
```

Nothing too surprising here. We are just importing all of the required modules, we set our target URL and then create a directory for all of our images to be stored.

Let's now implement the first function that will be responsible for querying the Wayback Machine for all unique snapshots of our target URL:

```
20 #
21 # Searches the Wayback machine for the provided URL
22 #
23 def search_archive(url):
24
25     # search for all unique captures for the URL
26     results = waybackpack.search(url,uniques_only=True)
27
28     timestamps = []
29
30     # build a list of timestamps for captures
31     for snapshot in results:
32         timestamps.append(snapshot['timestamp'])
33
34     # request a list of archives for each timestamp
35     packed_results = waybackpack.Pack(url,timestamps=timestamps)
36
37     return packed_results
```

- **Line 24:** we define our **search_archive** function to take the **url** parameter which represents the URL that we want to search the Wayback Machine for.
- **Line 27:** we leverage the **search** function provided by ***waybackpack*** to search for our URL and also we specify that we only want unique captures so that we aren't having to examine a bunch of duplicate captures.
- **Lines 29-33:** we create an empty list of timestamps (29) and then begin walking through the results of our search (32) and add the timestamp that corresponds to a particular capture in the Wayback Machine (33).
- **Lines 36-38:** we pass in the original URL and the list of timestamps to create a **Pack** object (36). A **Pack** object assembles the timestamps and the URL into a Wayback Machine friendly format. We then return this object from our function (38).

Now that our search function is implemented, we need to process the results, retrieve each captured page and then extract all image paths stored in the HTML. Let's do this now.

```
39 #
40 # Retrieve the archived page and extract the images from it.
41 #
42 def get_image_paths(packed_results):
43
44     images          = []
45     count           = 1
46
47     for asset in packed_results.assets:
48
49         # get the location of the archived URL
50         archive_url = asset.get_archive_url()
51
52         print "[*] Retrieving %s (%d of %d)" % (archive_url,count,len(packed_results.assets))
53
54         # grab the HTML from the Wayback machine
55         result = asset.fetch()
56
57         # parse out all image tags
58         soup = bs4.BeautifulSoup(result)
59         image_list = soup.findAll("img")
60
61         # loop over the images and build full URLs out of them
62         if len(image_list):
63
64             for image in image_list:
65
66                 if not image.attrs['src'].startswith("http"):
67                     image_path =  urlparse.urljoin(archive_url,image.attrs['src'])
68                 else:
69                     image_path = image.attrs['src']
70
71                 if image_path not in images:
72                     print "[+] Adding new image: %s" % image_path
73                     images.append(image_path)
74
75         count += 1
76
77     return images
```

- **Line 43:** we setup our **get_image_paths** function to receive the **Pack** object.
- **Lines 48-56:** we walk through the list of assets (48), and then use the **get_archive_url** function (51) to hand us a useable URL. We print out a little helper message (53) and then we retrieve the HTML using the **fetch** function (56).

- **Lines 59-60:** now that we have the HTML we hand it off to ***BeautifulSoup*** (59) so that we can begin parsing the HTML for image tags. The parsing is handled by using the **findAll** function (60) and passing in the ***img*** tag. This will produce a list of all IMG tags discovered in the HTML.
- **Lines 63-70:** we walk over the list of IMG tags found (64) and we build URLs (67-70) to the images that we can use later to retrieve the images themselves.
- **Lines 72-74:** if we don't already have the image URL (72) we print out a message (73) and then add the image URL to our list of all images found (74).

Alright! Now that we have extracted all of the image URLs that we can, we need to download them and process them for EXIF data. Let's implement this now.

```
79  #
80  # Download the images and extract the EXIF data.
81  #
82  def download_images(image_list,url):
83
84      image_results = []
85      image_hashes  = []
86
87      for image in image_list:
88
89          # this filters out images not from our target domain
90          if url not in image:
91              continue
92
93          try:
94              print "[v] Downloading %s" % image
95              response = requests.get(image)
96          except:
97              print "[!] Failed to download: %s" % image
98              continue
99
100         if "image" in response.headers['content-type']:
101
102             sha1 = hashlib.sha1(response.content).hexdigest()
103
104             if sha1 not in image_hashes:
105
106                 image_hashes.append(sha1)
107
108                 image_path = "waybackimages/%s-%s" % (sha1,image.split("/")[-1])
```

```
109
110                 with open(image_path,"wb") as fd:
111                     fd.write(response.content)
112
113                 print "[*] Saved %s" % image
114
115                 info = pyexifinfo.get_json(image_path)
116
117                 info[0]['ImageHash'] = sha1
118
119                 image_results.append(info[0])
120
121     return image_results
```

Let's pick this code apart a little bit:

- **Line 83:** we define our **download_images** function that takes in our big list of image URLs and the original URL we are interested in.
- **Lines 85-86:** we create our **image_results** variable (85) to hold all of our EXIF data results and the **image_hashes** variable (86) to keep track of all of the unique hashes for the images we download. More on this shortly.
- **Lines 88-98:** we walk through the list of image URLs (88) and if our base URL is not in the image path (91) then we ignore it. We then download the image (96) so that we can perform our analysis.
- **Lines 101-103:** if we have successfully downloaded the image (101), we then SHA-1 hash the image so that we can track unique images. This allows us to have multiple images at with the same file name but if the contents of the image are different (even by one byte) then we will track them separately. This also prevents us from having multiple copies of exact duplicate images.
- **Lines 105-114:** if it is a new unique image (105) we add the hash to our list of image hashes (107) and then we write the image out to disk (111).
- **Line 116:** here we are calling the ***pyexifinfo*** function **get_json**. This function extracts the EXIF data and then returns the results as a Python dictionary.
- **Lines 118-120:** we add our own key to the **info** dictionary that contains the SHA-1 hash of the image (118) and then we add the dictionary to our master list of results (120).

We are almost finished. Now we just need to tie all of these functions together and get some output in CSV format so that we can easily review all of the EXIF data that we have discovered. Time to put the finishing touches on this script!

```
1 results = search_archive(url)
2
3 print "[*] Retrieved %d possible stored URLs" % len(results.assets)
```

```
4
5  image_paths = get_image_paths(results)
6
7  print "[*] Retrieved %d image paths." % len(image_paths)
8
9  image_results = download_images(image_paths,url)
10
11 # return to JSON and have pandas build a csv
12 image_results_json = json.dumps(image_results)
13
14 data_frame = pandas.read_json(image_results_json)
15
16 csv = data_frame.to_csv("results.csv")
17
18 print "[*] Finished writing CSV to results.csv"
```

Let's break down this last bit of code:

- **Lines 124-132:** we call all of our functions starting by performing the Wayback Machine search (124), extracting the image paths (128) and then downloading and processing all of the images (132).
- **Lines 135-139:** we convert the returned dictionary to a JSON string (135), and then pass that JSON into the ***pandas* read_json** function (137) that will create a dataframe in ***pandas.*** We then leverage a wonderful function **to_csv** that converts that data frame to a CSV file, complete with automatic headers. This saves us from having to code up a complicated CSV creation routine. The CSV file is stored in ***results.csv*** in the same directory as the script.

Let It Rip!

Ok now for the fun part. Set the URL you are interested in, and just run the script from the command line or from your favourite Python IDE. You should see some output like the following:

---

[*] Retrieved 41 possible stored URLs
[*] Retrieving https://web.archive.org/web/20110823161411/http://www.oct282011.com/ (1 of 41)

```
[*] Retrieving https://web.archive.org/web/20110830211214/http://www.oct282011.com/ (2 of 41)
[+] Adding new image: https://web.archive.org/web/20110830211214/http://www.oct282011.com/st.jpg

...

[*] Saved https://web.archive.org/web/20111016032412/http://www.oct282011.com/material_same_habits.png
[v] Downloading https://web.archive.org/web/20111018162204/http://www.oct282011.com/ignoring.png
[v] Downloading https://web.archive.org/web/20111018162204/http://www.oct282011.com/material_same_habits.png
[v] Downloading https://web.archive.org/web/20111023153511/http://www.oct282011.com/ignoring.png
[v] Downloading https://web.archive.org/web/20111023153511/http://www.oct282011.com/material_same_habits.png
[v] Downloading https://web.archive.org/web/20111024101059/http://www.oct282011.com/ignoring.png
[v] Downloading https://web.archive.org/web/20111024101059/http://www.oct282011.com/material_same_habits.png
[*] Finished writing CSV to results.csv
```

START NOW $49.99

Online Python Course

Want more Python and OSINT?

Join my mailing list now and don't miss out!

Your email...

SUBSCRIBE

RECENT POSTS

Follow the Bitcoin With Python, BlockExplorer and Webhose.io
New! Automatically Discover Website Connections Through Tracking Codes
Building a Keyword Monitoring Pipeline with Python, Pastebin and Searx
Vacuuming Image Metadata from The Wayback Machine
Dark Web OSINT Part Four: Using Scikit-Learn to Find Hidden Service Clones

RECENT COMMENTS

Justin on Automatically Discover Website Connections Through Tracking Codes
Justin on Gaming Meets OSINT: Using Python to Help Solve Her Story
OSINTDude on Automatically Discover Website Connections Through Tracking Codes
shinrahunter on Gaming Meets OSINT: Using Python to Help Solve Her Story
Harvey on Automatically Discover Website Connections Through Tracking Codes

CATEGORIES

API (11)
Bitcoin (1)
Dark Web (5)
Facebook (1)
Forensics (1)
Geolocation (7)
Gephi (4)
Google Maps API (1)
Imagga (1)
Imagga API (1)
OpenCorporates API (1)
OSINT (28)
Pastebin API (1)
Photography (6)
Python (24)
Shodan (1)
Spyonweb API (1)
Text Analysis (10)
TinEye API (2)
Twitter API (1)
Uncategorized (3)
Vimeo API (1)
Wayback Machine (1)

Web Scraping (3)
Webhose.io API (1)
Wikimapia API (1)
YouTube API (1)