Open source intelligence techniques & commentary

Expanding Skype Forensics with OSINT: Email Accounts

Written by **Justin,** May 2nd, 2016

I will be the first to tell you that I know little about forensics compared to most law enforcement or private forensic examiners. One thing that I always found amazing was looking at the result of a forensic acquisition and seeing all of that magical data flowing out from it. Email addresses, phone numbers, usernames, social media, images, the list goes on and on. This always struck me as a place where OSINT could be applied as a follow-on to try to expand your knowledge of the acquired device and

the owner. So I reached out to a few forensics gurus (thanks Shafik Punja and Jacques Boucher) to ask them where there is a good source of forensic information on a hard drive that I could use to begin querying online services for additional information. Jacques was kind enough to point out that the Skype database is in SQLite format and was a veritable treasure trove of information.

So what we are going to do in this post is twofold: we will build a Python script that can extract emails from **any** SQLite database and we will utilize the Full Contact API to perform lookups on the email accounts that we find. The final output of our adventure will be a spreadsheet that contains all of the social media profiles that were discovered. Let's get started.

Python and SQLite

SQLite is a file based database system that has many of the great features that most server based database systems have, all compacted down into a nice tiny little file. The wonderful thing is that Python has built-in support for SQLite so this will make it very easy for us to interface to any SQLite file that we choose. This script will be designed such that no matter what database is passed in, we will systematically walk through each discovered table, and search through each column looking for email addresses, which should allow you to move beyond just Skype for exploration.

SQLite has a table that describes the schema for the database called SQLITE_MASTER. We will execute a query against this table to pull out all of the tables that are contained in the database. From there we will walk through each table and do a SQL SELECT statement to select all records from each table. When results come back, they are broken down into columns and we will walk through each column one by one and attempt to extract email addresses. Let's get started by creating a new Python script named ***sqlite_parser.py*** and entering the following code (download full source here):

```
1 import argparse
2 import re
3 import sqlite3
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-d","--database",    required=True,help="Path to the SQLite database you wish to analyze.")
7 args = vars(ap.parse_args())
8
9 match_list  = []
```

```
10 regex_match = re.compile('[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+')
```

- **Lines 1-3:** we import the necessary libraries we are going to use to parse the database.
- **Lines 8-10:** here we setup a command line argument parser to handle passing in the location of the database file we wish to parse.
- **Line 9:** we initialize an empty list that will hold all of the matches that we extract from the database.
- **Line 10:** here we are setting up a regular expression to match email addresses. This is not an exhaustive pattern but it does the job. The great thing is that you could modify this for IP addresses (which we are going to do in Part 2) or any other pattern that you wish to extract from a SQLite database.

Now let's get hooked up to the SQLite database and find all of the tables that are available:

```
12 # connect to the database
13 db     = sqlite3.connect(args['database'])
14 cursor = db.cursor()
15
16 cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
17
18 tables = cursor.fetchall()
```

Let's break this down a bit:

- **Line 13:** we connect to the SQLite database file by passing in the path to the database file. The path comes from our command line argument that we passed in.
- **Line 14:** once we have connected, we create a SQLite cursor object that we will use to issue SQL statements to the database.
- **Line 16:** we execute our query to extract all tables from the **sqlite_master** table which will give us the list of tables we can then walk through.
- **Line 18:** we use the **fetchall()** function to retrieve the query results from our previous query on line 16. This will return a list that we can then loop over and perform subsequent queries against.

Now we need to loop over each table, query it for all of its data, and then walk through each column, applying our regular expression to try to extract email addresses. This will be a longer chunk of code, so let's get to it:

```
20 for table in tables:
21
22     print "[*] Scanning table...%s" % table
23
24     # now do a broad select for all records
```

```
25      cursor.execute("SELECT * FROM %s" % table[0])
26
27      rows = cursor.fetchall()
28
29      for row in rows:
30
31          for column in row:
32
33              try:
34                  matches = regex_match.findall(column)
35              except:
36                  continue
37
38              for match in matches:
39
40                  if match not in match_list:
41
42                      match_list.append(match)
```

- **Lines 20-27:** we begin looping (20) over the list of tables, print out a helpful message (22) and then we do a SELECT statement on the table (25) to retrieve all of the records from the table. We use the **fetchall()** function (27) again to retrieve all records from our SELECT query.
- **Lines 29-31:** we start looping over each row in the result (29) and then for each row we have we start looping over each column (31).
- **Lines 33-36:** we attempt to apply our email regular expression (34) and if we are unsuccessful (36) we head back to the top of the loop on line 33 to start looking in the next column.
- **Lines 38-42:** we walk through the list of matches from our regular expression (38) and if we encounter a new match that we haven't already encountered (40) we add the match to our global list of matches (42) before carrying on.

Now let's put the final touches on the script to close our database resources and print out all of the matches we have encountered. Add the following lines of code to your script:

```
43  cursor.close()
44  db.close()
45
46  print "[*] Discovered %d matches." % len(match_list)
47
48  for match in match_list:
49      print match
```

- **Lines 43-44:** we close the database cursor (43) and the connection to the SQLite file (44).

- **Lines 48-49:** we loop through all of our matches and print out each one.

Well done! Now it is tip to give it a spin to make sure it works. As was described by Jacques to me, you can locate the Skype SQLite files like so:

For Mac OSX:

/Users/<your_mac_username>/Library/Application\ Support/Skype/<your_skype_username>/main.db

For Windows:

%appdata%\Skype\main.db

Let it Rip!

So let's give this little unit a run and see some output. In this case I have the main.db file in the same file location as my Python script:

Justins-MacBook-Pro:Desktop justin$ **python sqlite_osint.py -d main.db**

[*] Scanning table...DbMeta
[*] Scanning table...AppSchemaVersion
[*] Scanning table...Contacts
[*] Scanning table...LegacyMessages
[*] Scanning table...Calls
[*] Scanning table...Accounts
[*] Scanning table...Transfers

...more tables scanned here

[*] Discovered 68 matches.

justin.seitz@gmail.com

... 67 other email addresses here.

If you get output like the above then you know everything is working! Now let's integrate the Full Contact API so we can do some OSINT on the email addresses that we have extracted.

Integrating FullContact

We can now extract email accounts from any SQLite we please, now of course to leverage this information we want to try to attribute those email accounts to social media or some other online presence that we can utilize to gain additional intelligence. We will utilize FullContact (a data aggregator) to do some additional lookups on the email accounts that we discover. As we discover accounts we will add them to a CSV file so that we can explore the data using Excel or Google Fusion Tables, this will also allow us to easily bring this data into other tools. The first step is to sign up for a FullContact API key. Once you have done that, save your ***sqlite_parser.py*** script as ***sqlite_fullcontact.py*** (download from here) and let's make some modifications. Right after your **import sqlite3** line add the following:

```
import argparse
import re
import sqlite3
import requests
import time
import csv

full_contact_api_key = "YOURAPIKEY"
```

Perfect, we are just adding some additional modules and a variable to hold your FullContact API key. If you don't have the **requests** library you can use pip to install it (view videos here for help). Now we are going to start adding code to the

bottom of your script at line 53. Punch out the following:

```
53 # setup for csv writing
54 fd         = open("%s-social-media.csv" % args['database'],"wb")
55 fieldnames = ["Email Address","Network","User ID","Username","Profile URL"]
56 writer     = csv.DictWriter(fd,fieldnames=fieldnames)
57 writer.writeheader()
```

- **Line 54:** we open a file handle for our CSV file where we will store the retrieved information.
- **Line 55:** we setup a list of field names which will become the columns in our spreadsheet.
- **Line 56:** we use the **csv** module's DictWriter class, passing in the file handle we created on line 54 and the list of field names created on line 55. The DictWriter class will allow us to write rows in our spreadsheet by passing in a dictionary.
- **Line 57:** we write out the top row (the header) of our spreadsheet that will give us column names like any good spreadsheet should have.

Next we need to walk through the list of matches from our SQLite parsing, and pass each email address of to the FullContact API to try to retrieve any results they may have. Let's implement the code to do this now:

```
59 while match_list:
60
61     # build the request up for Full Contact
62     headers = {}
63     headers['X-FullContact-APIKey'] = full_contact_api_key
64
65     match   = match_list.pop()
66
67     print "[*] Trying %s" % match
68     url = "https://api.fullcontact.com/v2/person.json?email=%s" % match
69
70     response = requests.get(url,headers=headers)
```

- **Line 59:** this block of code will continue executing as long as there are email matches to process.
- **Lines 62-63:** in order to authenticate to the FullContact API we have to pass in the **X-FullContact-APIKey** HTTP header. We do this by setting up a **headers** dictionary (62) and then set the required header and the value of our API key (63).
- **Lines 65-68:** we grab an email address from our list (65) print it out (67) and then we build a URL to pass in the email address to the FullContact API endpoint (68).
- **Lines 70-72:** we send off the request to the FullContact API, passing in our HTTP headers in the **headers** variable (70) and then sleep for two seconds (72) to obey the rate-limiting imposed by the FullContact servers.

We have our request sent off and now it is time to test the results, and if there are good matches we need to store them in our CSV file. Let's implement the code to do so:

```
74      if response.status_code == 200:
75
76          contact_object = response.json()
77
78          if contact_object.has_key('socialProfiles'):
79
80              for profile in contact_object['socialProfiles']:
81
82                  record = {}
83                  record['Email Address'] = match
84                  record['Network']       = profile.get("type","N/A")
85                  record['User ID']       = profile.get("id","N/A")
86                  record['Username']      = profile.get("username","N/A")
87                  record['Profile URL']   = profile.get("url","N/A")
88
89                  writer.writerow(record)
90
91                  # print some output to the screen
92                  print "Network: %s"  % profile.get("type","N/A")
93                  print "Username: %s" % profile.get("username","N/A")
94                  print "URL: %s"      % profile.get("url","N/A")
95                  print "ID: %s"       % profile.get("id","N/A")
96                  print
```

- **Line 74:** if we receive a good response back from the FullContact API we are ready to test for matches.
- **Line 76:** we let the requests library parse the JSON response from the server and store the result in the **contact_object** variable.
- **Line 78:** we test the **contact_object** (which is now a dictionary) for the ***socialProfiles*** key which will indicate to us that there is a hit for a social media profile.
- **Line 80:** a single email account can have multiple social media profiles, and they are stored in list in the ***socialProfiles*** key. We begin walking through each social media profile and store each one in the **profile** variable.
- **Lines 82-87:** we initialize an empty dictionary called **record** (82) and begin populating it with all of the data retrieved from the FullContact API. The **get()** function will attempt to retrieve each of the values you see (type, id, username, url) and if there is no value present it will automatically return "N/A" as shown.
- **Line 89:** we write the **record** dictionary to our CSV file as a new row.
- **Lines 91-96:** we simply print out the information that we have found so we can monitor the output as the script is running.

We now need to implement a separate check to see if the FullContact API is wanting us to wait before retrieving results. This can happen randomly, and to be honest I am not sure why. We are going to test for the HTTP response code 202 instead of 200 and then re-add the email address that was tested back to our **matches** list to make sure that we aren't dropping email addresses. Be mindful of indentation here, it should be indented as far as our status code 200 check:

```
    elif response.status_code == 202:

        print "[*] Sleeping for a bit."

        # push this item back onto the list and sleep
        match_list.append(match)
        time.sleep(30)

fd.close()
```

Just like that! We test for the 202 code, and if we encounter it we push the email address back into our list of email addresses to check and then we sleep for 30 seconds. The last line of code simply closes the file handle associated to our CSV file. Now it is time to give it a run!

Let It Rip!

You are going to run this script in the exact same way that you ran the previous one but the output should be different, assuming you get good results from the FullContact API:

Justins-MacBook-Pro:Desktop justin$ **python sqlite_osint_fullcontact.py -d main.db**

[*] Scanning table...DbMeta
[*] Scanning table...AppSchemaVersion
[*] Scanning table...Contacts

[*] Trying justin.seitz@gmail.com

Network: twitter
Username: jms_dot_py
URL: https://twitter.com/jms_dot_py
ID: 817668451

Of course there were many more hits than just my email address but you can see how this can greatly expand your investigations once you have a SQLite database in your hands. In our next instalment we'll take a look at how to build a map out of the IP addresses discovered to demonstrate how flexible our SQLite parsing is and how easy it is to adapt our scripts. Do you have a cool SQLite database that yields a pile of information like Skype does? Shoot me an email and let me know, because I would love to test it out! Thanks for reading, I'm looking forward to hearing from you.

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)