Classifying malware using network traffic analysis.

Or how to learn Redis, git, tshark and Python in 4 hours.

CIRCL
Computer Incident
Response Center
Luxembourg

Alexandre Dulaunoy

February 2, 2018

Problem Statement

- We have more 5000 pcap files generated per day for each malware execution in a sandbox
- We need to classify[1] the malware into various sets
- The project needs to be done in less than a day and the code shared to another team via GitHub

[1] Classification parameters are defined by the analyst

File Format and Filename

```
...
0580c82f6f90b75fcf81fd3ac779ae84.pcap
05a0f4f7a72f04bda62e3a6c92970f6e.pcap
05b4a945e5f1f7675c19b74748fd30d1.pcap
05b57374486ce8a5ce33d3b7d6c9ba48.pcap
05bbddc8edac3615754f93139cf11674.pcap
05bf1ff78685b5de06b0417da01443a9.pcap
05c3bccc1abab5c698efa0dfec2fd3a4.pcap
...
<MD5 hash of the malware).pcap
```

MD5 values[2] of malware samples are used by A/V vendors, security researchers and analyst.

[2] https://www.virustotal.com/ as an example

MapReduce and Network Forensic

- MapReduce is an old concept in computer science
  - The **map** stage to perform isolated computation on independent problems
  - The **reduce** stage to combine the computation results
- Network forensic computations can easily be expressed in map and reduce steps:
  - parsing, filtering, counting, sorting, aggregating, anonymizing, shuffling...

Processing and Reading pcap files

```
ls -1  | parallel --gnu 'tcpdump -s0 -A -n -r {1}'
```

- Nice for processing the files but...
- How do we combine the results?
- How do we extract the classification parameters? (e.g. sed, awk, regexp?)

tshark

```
tshark -G fields
```

- Wireshark is supporting a wide range of dissectors
- tshark allows to use the dissectors from the command line

```
tshark -E header=yes -E separator=, -Tfields\
-e ip.dst -r mycap.cap
```

Concurrent Network Forensic Processing

- To allow concurrent processing, a non-blocking data store is required
- To allow flexibility, a schema-free data store is required
- To allow fast processing, you need to scale horizontally and to know the cost of querying the data store
- To allow streaming processing, write/cost versus read/cost should be equivalent

Redis: a key-value/tuple store

- Redis is key store written in C with an extended set of data types like lists, sets, ranked sets, hashes, queues
- Redis is usually in memory with persistence achieved by regularly saving on disk
- Redis API is simple (telnet-like) and supported by a multitude of programming languages
- `http://www.redis.io/`

Redis: installation

- Download Redis 4.0.8 (stable version)
- tar xvfz redis-4.0.8.tar.gz
- cd redis-4.0.8
- make

Keys

- Keys are free text values (up to 2^{31} bytes) - newline not allowed
- Short keys are usually better (to save memory)
- Naming convention are used like keys separated by colon

Value and data types

- binary-safe strings
- lists of binary-safe strings
- sets of binary-safe strings
- hashes (dictionary-like)
- pubsub channels

Running redis and talking to redis...

- screen
- cd ./src/ && ./redis-server
- new screen session (crtl-a c)
- redis-cli
- DBSIZE

Commands available on all keys

Those commands are available on all keys regardless of their type

- TYPE [key] $\rightarrow$ gives you the type of key (from string to hash)
- EXISTS [key] $\rightarrow$ does the key exist in the current database
- RENAME [old new]
- RENAMENX [old new]
- DEL [key]
- RANDOMKEY $\rightarrow$ returns a random key
- TTL [key] $\rightarrow$ returns the number of sec before expiration
- EXPIRE [key ttl] or EXPIRE [key ts]
- KEYS [pattern] $\rightarrow$ returns all keys matching a pattern (!to use with care)

Commands available for strings type

- SET [key] [value]
- GET [key]
- MGET [key1] [key2] [key3]
- MSET [key1] [valueofkey1] ...
- INCR [key] — INCRBY [key] [value] $\rightarrow$! string interpreted as integer
- DECR [key] — INCRBY [key] [value] $\rightarrow$! string interpreted as integer
- APPEND [key] [value]

Commands available for sets type

- SADD [key] [member] $\rightarrow$ adds a member to a set named key
- SMEMBERS [key] $\rightarrow$ return the member of a set
- SREM [key] [member] $\rightarrow$ removes a member to a set named key
- SCARD [key] $\rightarrow$ returns the cardinality of a set
- SUNION [key ...] $\rightarrow$ returns the union of all the sets
- SINTER [key ...] $\rightarrow$ returns the intersection of all the sets
- SDIFF [key ...] $\rightarrow$ returns the difference of all the sets
- S....STORE [destkey key ...] $\rightarrow$ same as before but stores the result

Commands available for list type

- RPUSH - LPUSH [key] [value]
- LLEN [key]
- LRANGE [key] [start] [end]
- LTRIM [key] [start] [end]
- LSET [key] [index] [value]
- LREM [key] [count] [value]

Geospatial type

- GEOADD [key] [longitude] [latitude] [member]
- GEODIST [key] [longitude] [latitude] [radius] m—km—ft—mi
- GEORADIUS [key] [longitude] [latitude] [radius] m—km—ft—mi
- GEOPOS [key] [member]

Commands availabled for sorted set type

- ZADD [key] [score] [member]
- ZCARD [key]
- ZSCORE [key] [member]
- ZRANK [key] [member] $\rightarrow$ get the rank of a member from bottom
- ZREVRANK [key] [member] $\rightarrow$ get the rank of a member from top

Atomic commands

- GETSET [key] [newvalue] $\rightarrow$ sets newvalue and return previous value
- (M)SETNX [key] [newvalue] $\rightarrow$ sets newvalue except if key exists (useful for locking)

MSETNX is very useful to update a large set of objects without race condition.

Database commands

- SELECT [0-15] $\rightarrow$ selects a database (default is 0)
- MOVE [key] [db] $\rightarrow$ move key to another database
- FLUSHDB $\rightarrow$ delete all the keys in the current database
- FLUSHALL $\rightarrow$ delete all the keys in all the databases
- SAVE - BGSAVE $\rightarrow$ save database on disk (directly or in background)
- DBSIZE
- MONITOR $\rightarrow$ what's going on against your redis datastore (check also redis-stat)

Redis from shell?

```
ret=$(redis-cli SADD dns:${md5} ${rdata})
num=$(redis-cli SCARD dns:${md5})
```

- Why not Python?

```
import redis
r = redis.StrictRedis(host='localhost', port=6379, db=0)
r.set('foo', 'bar')
r.get('foo')
```

How do you integrate it?

```
ls -1 ./pcap/*.pcap | parallel --gnu "cat {1} |
tshark -E header=yes -E separator=, -Tfields -e http.server
python import.py -f {1} "
```

- Code need to be shared?

Structure and Sharing

- As you start to work on a project and you are willing to share it
- One approach is to start with a common directory structure for the software to be shared

```
/PcapClassifier/bin/ -> where to store the code
                /etc/ -> where to store configuration
                /data/
                README.md
```

import.py (first iteration)

```
# Need to catch the MD5 filename of the malware (from the pcap filename)
import argparse
import sys
argParser = argparse.ArgumentParser(description='Pcap classifier')
argParser.add_argument('-f', action='append', help='Filename')
args = argParser.parse_args()
if args.f is not None:
    filename = args.f
    for line in sys.stdin:
        print (line.split(",")[0])
else:
    argParser.print_help()
```

Storing the code and its revision

Initialize a git repository (one time)

```
cd PcapClassifier
git init
```

Add a file to the index (one time)

```
git add ./bin/import.py
```

Commit a file to the index (when you do changes)

```
git commit ./bin/import.py
```

import.py (second iteration)

```
# Need to know and store the MD5 filename processed
import argparse
import sys
import redis

argParser = argparse.ArgumentParser(description='Pcap classifier')
argParser.add_argument('-f', action='append', help='Filename')
args = argParser.parse_args()

r = redis.StrictRedis(host='localhost', port=6379, db=0)

if args.f is not None:
    md5 = args.f[0].split(".")[0]
    r.sadd('processed', md5)
    for line in sys.stdin:
        print (line.split(",")[0])
else:
    argParser.print_help()
```

Redis datastructure

- Now we have a set of the file processed (including the MD5 hash of the malware):

$\{processed\}=\{abcdef\ldots, deadbeef\ldots, 0101aba\ldots.\}$

- A type set to store the field type decoded from the pcap (e.g. http.user_agent)

$\{type\}=\{"http.user\_agent", "http.server"\}$

- A unique set to store all the values seen per type

$\{e : http.user\_agent\}=\{"Mozilla\ldots", "Curl\ldots"\}$

import.py (third iteration)

```
import argparse
import sys
import redis

argParser = argparse.ArgumentParser(description='Pcap classifier')
argParser.add_argument('-f', action='append', help='Filename')
args = argParser.parse_args()

r = redis.StrictRedis(host='localhost', port=6379, db=0)

if args.f is not None:
    md5 = args.f[0].split(".")[0]
    r.sadd('processed', md5)
    lnumber=0
    fields=None
    for line in sys.stdin:
        if lnumber == 0:
            fields = line.rstrip().split(",")
            for field in fields:
                r.sadd('type', field)
        else:
            elements = line.rstrip().split(",")
            i=0
            for element in elements:
                try:
                    r.sadd('e:'+fields[i], element)
                except IndexError:
                    print("Empty fields")
                i=i+1

        lnumber = lnumber + 1
else:
    argParser.print_help()
```

How to link a type to a MD5 malware hash?

- Now, we have set of value per type

$\{e : http.user\_agent\}=\{"Mozilla...","Curl..."\}$

- Using MD5 to hash type value and create a set of type value for each malware

$\{v : MD5hex("Mozilla...")\}=\{"MD5\ of\ malware","...."\}$

import.py (fourth iteration)

```
import argparse
import sys
import redis
import hashlib

argParser = argparse.ArgumentParser(description='Pcap classifier')
argParser.add_argument('-f', action='append', help='Filename')
args = argParser.parse_args()

r = redis.StrictRedis(host='localhost', port=6379, db=0)

if args.f is not None:
    md5 = args.f[0].split(".")[0]
    r.sadd('processed', md5)
    lnumber=0
    fields=None
    for line in sys.stdin:
        if lnumber == 0:
            #print (line.split(",")[0])
            fields = line.rstrip().split(",")
            for field in fields:
                r.sadd('type', field)
        else:
            elements = line.rstrip().split(",")
            i=0
            for element in elements:
                try:
                    r.sadd('e:'+fields[i], element)
                    ehash = hashlib.md5()
                    ehash.update(element.encode('utf-8'))
                    ehhex = ehash.hexdigest()
                    if element is not "":
                        r.sadd('v:'+ehhex, md5)
                except IndexError:
                    print("Empty fields")
                i=i+1

        lnumber = lnumber + 1
else:
    argParser.print_help()
```

graph.py - exporting graph from Redis

```
import redis
import networkx as nx
import hashlib

r = redis.StrictRedis(host='localhost', port=6379, db=0)

g = nx.Graph()

for malware in r.smembers('processed'):
    g.add_node(malware)

for fieldtype in r.smembers('type'):
    g.add_node(fieldtype)
    for v in r.smembers('e:'+fieldtype.decode('utf-8')):
        g.add_node(v)

        ehash = hashlib.md5()
        ehash.update(v)
        ehhex = ehash.hexdigest()
        for m in r.smembers('v:'+ehhex):
            print (m)
            g.add_edge(v,m)

nx.write_gexf(g,"/tmp/out.gexf")
```

graph.py - exporting graph from Redis