Intrusion Detection Through Relationship Analysis

GIAC GCIA Gold Certification

Author: Patrick Neise, patrick.neise@gmail.com

Advisor: Angel Alonso Parrizas

Accepted: 22 October 2016

Abstract

With the average time to detection of a network intrusion in enterprise networks assessed to be 6-8 months, network defenders require additional tools and techniques to shorten detection time. Perimeter, endpoint, and network traffic detection methods today are mainly focused on detecting individual incidents while security incident and event management (SIEM) products are then used to correlate the isolated events. Although proven to be able to detect network intrusions, these methods can be resource intensive in both time and personnel. Through the use of network flows and graph database technologies, analysts can rapidly gain insight into which hosts are communicating with each other and identify abnormal behavior such as a single client machine communicating with other clients via Server Message Block (SMB). Combining the power of tools such as Bro, a network analysis framework, and neo4j, a native graph database that is built to examine data and its relationships, rapid detection of anomalous behavior within the network becomes possible. This paper will identify the tools and techniques necessary to extract relevant network information, create the data model within a graph database, and query the resulting data to identify potential malicious activity.

1. Introduction

Based on the most recent Verizon Data Breach Investigations Report (DBIR) the number of incidents and confirmed breaches continues to rise year over year. Additionally, attacks occur across the globe and throughout all industry verticals with "no locale, industry or organization is bulletproof when it comes to the compromise of data." (Verizon, 2016). Examination of the breach report from the DBIR consists of data from over 100,000 incidents and 3,141 confirmed breaches.

The DBIR identifies that the overwhelming majority of breaches are the result of external threat actors, with internal threats occurring at about one-quarter of the time in comparison. Additionally, the top two reasons for attacks year over year are financial motivations and espionage, with the former occurring nearly four times as often. At a minimum, these figures should demonstrate that current network defensive measures and techniques are inadequate for today's threat environment.

An interesting trend highlighted in DBIR demonstrates the continued growth in user devices and people being the target of attacks while the occurrence of threats targeting servers has declined. The increased targeting of endpoints and users coupled with the overall decline in time to compromise to the order of minutes required defenders to be in the position to quickly respond to attacks. Additionally, the overall time to exfiltrate compromised information is on the order of days, resulting in an extremely complex defensive posture for medium to large enterprises. Coupling the time to compromise and exfiltrate data with the projected time to detection of comprise being around six to eight months demonstrates the severe disadvantage defenders currently face.

It should be apparent that current methods of detection are woefully inadequate in the face of today's threats. The pattern of reliance on signature-based detection mechanisms coupled with data aggregators and SIEMs does not appear to allow rapid detection of attacker activity. Additionally, typical flat network installations make detection of lateral movement challenging at best.

To improve on detection and response times, new techniques and tools are required to assist defenders in locating potentially malicious behavior. Rather than relying on signatures for intrusion detection systems (IDS) or antivirus to signify that an attack has occurred, defenders can add the analysis of relationships determined from network traffic to identify possible malicious activity.

Through the combination of tools currently being used by most network defenders and emerging technology and techniques, defenders will be able to identify potential malicious behavior. As a powerful IDS, Bro will be leveraged to provide session level data to neo4j, a graph database that treats relationships as a first-class entities. By transforming session data into nodes and relationships in a graph database, potentially malicious activity such as a client machine attempting to communicate via Server Message Block (SMB) to all other clients in the network, will be identifiable with a single query.

2. The Tools

To support rapid concept development, integration with existing tools, and minimal cost of concept implementation the tools used are completely free and open source or offer open source alternatives to their for-pay offerings. Additionally, the individual tools are recognized as being extremely flexible and useful for their specific goals and functionality. The implementation of creating relationships for analysis from network traffic will be accomplished with the Bro Network Security Monitor, the graph database neo4j, the containerization platform Docker, and the Python programming language. Finally, development and implementation are conducted using Ubuntu as a base operating system.

Versions of each tool used in the project include Bro 2.4.1, neo4j 3.0.4, Python 3.5.2, Docker 1.12.1, and Ubuntu 16.04. While the included source code will be compatible with some earlier versions of the tools, reproduction of the results may require modification to the code or techniques demonstrated.

2.1. Bro

As an open-source network security framework, the Bro Network Security Monitor is an extremely flexible and powerful framework for network traffic analysis and network security monitoring. As a passive traffic analyzer, Bro provides the ability to monitor traffic in real-time as well as evaluate recorded network traffic captures.

While Bro supports real-time monitoring and large-scale deployment, the implementation discussed focuses on the offline analysis of captured network traffic.

Specifically, the comprehensive logging of processed network traffic is used to develop the nodes and relationships entered into the graph database for analysis. Bro's built-in support for the most popular application-layer protocols and analysis of the file content exchanged over those protocols provides a significant amount of information to populate the graph database.

2.2. neo4j

To provide new insight into the robust logs generated by Bro, the graph database neo4j allows the analyst to evaluate the data produced. Additionally, neo4j allows the analyst to create and analyze the relationships generated by the connections between the data.

As a graph database, neo4j is built to provide a rapid and intuitive development of the underlying data model. The ability to change the data model without the need to completely re-architect the application allows for shorter development cycles and an agile response to new information and concepts as they are discovered.

Fundamental to the goals of analyzing relationships amongst the data generated by Bro logs is the ability to query the graph database to extract new and relevant information. Neo4j provides Cyber as a native graph query language which allows for a simple and expressive manner to describe relationship queries.

Additionally, neo4j provides a built-in web browser that allows for querying, visualizing, and interacting with the data contained in the database. The neo4j browser will be primarily used in the application as a data visualization tool. Python will be used to interact with the database for data creation and modification.

2.3. Python

As an interpreted and object-oriented programming language, Python provides an ideal platform from which to build the processing and integration needed to bring the individual tools and techniques together.

In addition to the standard library included with Python, a third party library is used to interact with the neo4j database. The pandas library provides an interface to easily manipulate and interact with structured data such as the log files produced by Bro and the comma separated value (CSV) files imported by neo4j.

Python will also provide for the relative seamless integration of the processes needed to parse the original PCAP and the created Bro logs, importing the required information into neo4j, and orchestration of the necessary Docker containers.

2.4. Docker

Although not a requirement to implement the functionality discussed within this technique, Docker allows for rapid development and deployment of the previously discussed tools.

Docker images for Bro and neo4j are used in order to remove the requirement to install either of those tools on the analysis platform. Specifically, the neo4j image provided by the official neo4j repository is located on Docker Hub, and the Bro image is provided by the user blacktop on Docker Hub. Although the Bro Docker image used in this procedure is provided by a third party, the Dockerfile is available for review on github and the Docker Hub. The Python code will execute from within a third container, specifically a container running the lightweight Alpine Linux distribution and Python 3.5.2.

The use of Docker containers greatly reduces the technical overhead required to implement this analysis technique while also providing a means to rapidly modify and redeploy the necessary tools. Additionally, the use of containers provides an added layer of secure when executing and evaluating packet captures or network traffic that may contain malicious packets or executables.

3. The Technique

From a technical standpoint, the implementation of the analysis technique is not overly complicated from the viewpoint of a practiced network security monitoring analyst. As discussed earlier, although Bro allows for the analysis of real-time traffic the techniques discussed will center on previously gathered packet captures.

With an existing packet capture available, the Bro Docker container from blacktop will be used to process the PCAP file to generate the necessary Bro log files. While Bro also allows for the creation of custom scripts to extract additional information and create custom log files, the efforts contained herein will focus on the included core Bro functionality.

The next step in the process is the extraction and transformation of the Bro log data for loading into neo4j. The standard Python libraries in concert with pandas will be used to read the generated Bro logs, extrapolate required information, and create files needed for the batch import of nodes and relationships within the graph database.

With all of the necessary information from the PCAP now transformed and represented as CSV files containing nodes and relationships, the information can be imported in neo4j for analysis. Due to the potential for a large number of nodes, on the order of millions, that may result from analyzing a large capture file, the neo4j-import tool will be run from within the Docker container in order to expedite import. With the database populated, the analyst can now query the database for interesting and relevant information. A few queries will be highlighted to demonstrate the syntax and capability of neo4j to assist the analyst, but a full listing of the possible outcomes is beyond the scope of this research and left to future analysis.

3.1. PCAP to Bro logs

The first step in the process in to run the selected PCAP through Bro in order to obtain the logs output by Bro. The example PCAP used throughout this process was obtained from the NETRESEC website in order to provide a relatively large file with expected malicious traffic. Specifically, the PCAP used throughout the example is the maccdc2012_00000.pcap from the 2012 Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC). Once extracted, the maccdc2012_00000.pcap is slightly over 1GB in size.

The reading of the PCAP into Bro and creation of the needed logs is a relatively straightforward process. From the documentation for the blacktop/bro Docker container, the following command will read the PCAP into Bro for processing:

```
$ docker run –rm -v /path/to/pcap:/pcap:rw blacktop/bro -r
my.pcap local
```

Breaking down each component of the above 'docker run' statement:

'—rm' – Remove the specified container after running. The above statement will process the PCAP and exit, then Docker will remove the container.
'-v /path/to/pcap:/pcap:rw' – Map the local folder '/path/to/pcap' to the '/pcap' folder in the Docker container as read/write. The Bro instance running in the container can now read the PCAP and write the logs to the local folder for further processing, even after removing the container.
'blacktop/bro' – A Docker image created by blacktop that contains the necessary files and dependencies to run Bro 2.4 within the container.
'-r my.pcap local' – Read the PCAP file specified by 'my.pcap' into Bro and run a common set of loaded scripts.

After the processing the PCAP through Bro, the logs associated with the analysis will be located in the same local directory as specified when running the container. There are logs for each observed connection, DNS traffic, HTTP traffic, DHCP, etc. The analysis included below will focus on the connections (conn.log), DNS traffic (dns.log), and HTTP traffic (http.log).

Using the default Bro settings, the logs will be written in a tab separated file. The log file includes multiple lines of descriptive information in the header and a single line footer. Using Python to process the log files of concern, the excess lines in the header and footer will be removed, leaving only the single line header that contains the column headers.

The Python source code for reading the pcap and cleaning the logs of concern is included as Appendix A below. Of note, to start and interact with the Docker containers, the subprocess module from the Python Standard Library is used to open a new process (Bordage, 2016).

3.2. Bro Logs to neo4j

In order import the converted Bro logs into neo4j, a consistent data model must be applied across the logs to create the desired nodes and relationships. The creation of an effective data model provides for repeatable and consistent ingest of logged information while creating a structure that supports extraction of desired information via specific, targeted queries of the database.

3.2.1. The Data Model

Through evaluation of the information contained within the conn.log, dns.log, and http.log files created by Bro and the potential types of questions to be answered by database queries, the types of nodes and relationships in the data model can be determined. In a general sense, the nodes can be thought of as the entities, the 'things' in the database, and the relationships describe how those entities depend on and interact with each other. Additionally, the nodes and relationships can each have specific properties that further define information about the specific entries.

As illustrated in Figure 1 below, the nodes created from the Bro logs consist of a Connection node, DNS_Record node, HTTP_Record node, and an IP node. Each of the nodes is labeled according to the names identified. By labeling the nodes, the speed of queries is improved by limiting the search to only nodes of the desired label type.

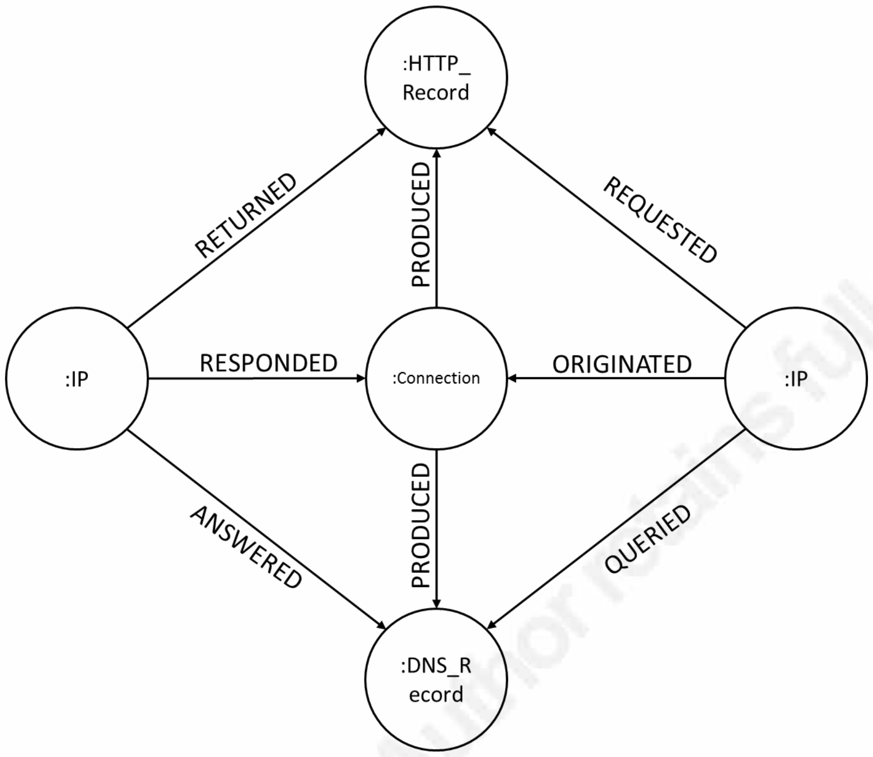

Figure 1: neo4j Data Model

The Connection node represents each connection entry within the conn.log created from the PCAP. The properties associated with each Connection node include all of the information contained within the individual log entry associated with the node, placing the Connection node at the center of the information located within the model.

The IP node is representative of every individual IP address located within the conn.log file. At the node level, there is no differentiation of the host being represented by the node as either the originator or responder for the individual connection entries. The goal is to provide a node for each IP address to identify how each address is related to the other through the connections, DNS entries, and HTTP entries located within the Bro logs.

The DNS_Record and HTTP_Record node types are similar in that they represent individual entries from their associated log files. The goal for these node types is to represent and capture the information associated with both sides of a DNS transaction or HTTP request. All of the data captured in the dns.log and http.log is identified as properties within the associated node for the given log entry.

The nodes by themselves are essentially just representations of individual log entries and therefore provide only as much utility as the log files themselves. The real power in this process comes from the identification and creation of the relationships between the different nodes.

The dns.log and http.log files contain amplifying information associated with their related entries within the conn.log file. To capture the relationship between the Connections and the individual record entries, The Connection node is represented as having PRODUCED the record node. The PRODUCED relationship allows numerous records associated with a single collection to be associated with each other for rapid identification and grouping of information for analytical queries. Additionally, the Connections nodes are related to the two IP nodes associated with the log entry through either an ORIGINATED or RESPONDED relationship to identify both ends of the connection.

The remaining relationships identify how the DNS_Record and HTTP_Record nodes are related to their associated Connection and IP nodes. Similar to the ORIGINATED/RESPONDED relationships the QUERIED/ANSWERED and REQUESTED/RETURNED relationship types connect the two IP nodes involved in the transmission of DNS or HTTP data respectively.

With the components of the data model identified, namely the nodes and relationships, the individual log files can be manipulated into the required layout to support import into neo4j. Due to the potentially large volume of imported information, the neo4j-import tool will be used to create the nodes and relationships from the Bro logs. In the case of the example MACCDC pcap used throughout this process, the pcap file itself is just over 1GB in size and contains over 4 million connections. This built-in tool requires the imported files to be in a specific format with requirements for how to label nodes as well as how to connect those nodes via the identified relationships.

3.2.2. Parsing the Logs

With the elements of the data model determined the Bro logs can be parsed and placed in the necessary format for import into the neo4j database (B.3. Use the Import tool, 2016). The import tool requires nodes and relationships to be placed into separate files for import. The nodes can be written into a single file or split into multiple files, passing each file into the import tool as a command line argument. The relationships can be treated in a similar manner.

Since Bro logs are output in a tab separated data structure, the pandas module for Python is used to manipulate the data structure to produce the CSV files. Pandas easily supports operations across columns and rows of large datasets, the conn.log in the example PCAP results in nearly 4 million rows of data. The process of cleaning and formatting the log files requires insertion and deletion of columns, removal of rows with missing data, indexing of rows, and removal of duplicate data.

For this process, files for each of the node types present in the data model and individual files are created for the relationship pairs, i.e. ORIGINATED/RESPONDED.

The resulting files are simply CSV files with a header line that identifies the column contents. For the node files, each column is added to the node as a property unless the column header contains a special string recognized by the import tool. For example, the ':LABEL' column heading specifies that the contents of that column are the Labels to apply to that particular node (i.e. Connection, DNS_Record, HTTP_Record, etc.) An additional column identifier used throughout this process is the ':ID' identifier. The ':ID' column is used by the import tool in the creation of the relationships between nodes as the lookup mechanism used to match two nodes to a relationship.

The relationship files are created in a similar manner as CSV files. However, they use a few different special column headers. The ':START_ID' and ':END_ID' column headers are used to signify which nodes are being connected by the relationship, and reference the previously discussed ':ID' column in the node CSV files. The ':TYPE' column is similar to the node label in that it signifies what type of relationship (i.e. PRODUCED, RESPONDED, QUERIED, etc.) connects the two nodes. Finally, just like the node CSV files, any remaining columns are created as properties of the relationship.

Cleaning the Bro logs, extracting the needed information, and creating the CSV files for import is completed using Python modules from the standard library. Source code for review is included as Appendix B – logs_to_csv.py.

3.2.3. Database Import

With the CSV files created, the next step in the process is to import the nodes and relationships into a neo4j Docker container to support queries and analysis. Operations with the neo4j container are handled in the same manner as the Bro container.

The Python script detailed in Appendix C performs the three major functions necessary to establish the database running with the imported data:

1) starting the neo4j container with a shared host volume and ports

2) deleting existing database and importing the new data via shell script, and

3) restarting the neo4j container

3.3. Putting It All Together

Figure 2 below provides a graphical representation of the overall process of taking a PCAP through Bro, cleaning and processing the logs into CSV files, and importing the data into neo4j using Docker containers for each phase of the process.

To pull the individual Python modules, a single Python script imports the functionality of the previously described portions of the process. The bro_to_neo.py script as shown in Appendix D – bro_to_neo.py, provides the framework to execute the individual components.

The overall script takes the identified PCAP file through the entire process to end with a running neo4j instance containing the nodes and relationships as extracted from the PCAP. The analyst is now able to use Cypher, the query language of neo4j, to extract relevant data from the network traffic.

For exploratory purposes, neo4j provides a built-in browser that allows for database interaction with the database and running Cypher queries. To access the browser after the neo4j Docker container has been started, access http://localhost:7474 with a browser. Upon initial access, the default password of 'neo4j' will have to be changed.

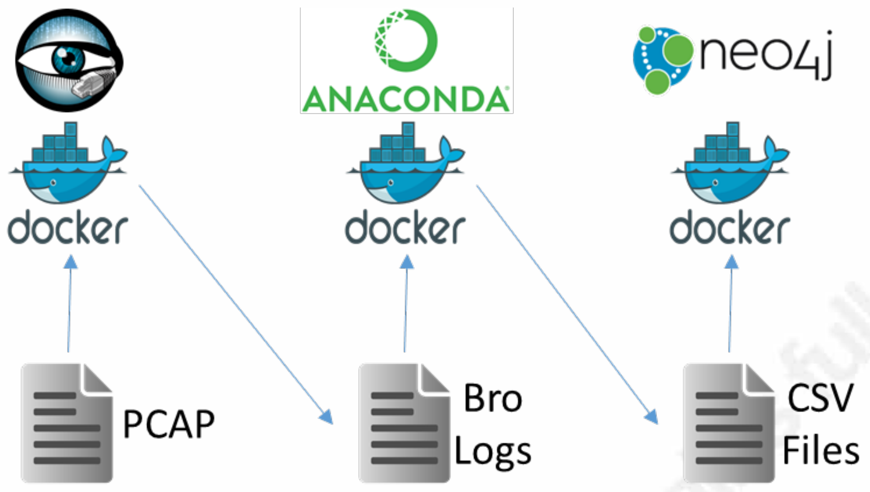

Figure 2: Overall Process Flow

4. Getting Results

With the database now populated with all of the nodes and relationships that were created from the input PCAP file, an analyst can now begin to extract information regarding the nature of the traffic. Neo4j includes a SQL-like query language, Cypher (Neo4j: The World's Leading Graph Database, 2016). Although this discussion is not intended to be a full Cypher tutorial, the language and concepts should not prove difficult for an intrusion analyst or network security professional.

Based on the example PCAP file, the database has 4,062,012 nodes and 16,420,388 relationships. Finding the node and relationship counts are relatively simple queries. To find the number of nodes in the database, the following query locates nodes of all types and returns the count.

```
MATCH(n) RETURN COUNT(n);
```

And to find the number of relationships:

```
MATCH()-[r]-() RETURN COUNT(r);
```

Patrick Neise, patrick.neise@gmail.com

The queries above queries demonstrate two of the most basic query patterns, looking for a node and looking for nodes connected by a relationship. The query format approximates a graphical representation of two nodes, represented by the parenthesis () and the relationship between the nodes represented by the brackets [].

4.1. Noisy Hosts

As an introduction to the ease of use and power of the Cypher query language, the first example seeks to identify the host making the highest number of outbound connections. The query below demonstrates the general flow of a Cypher query, find a match on nodes and relationships and return the results in the desired format.

```
MATCH (ip.ip:IP)-[rel:ORIGINATED]-(c:Connection)
RETURN ip, COUNT(rel) AS connections
ORDER BY connections DESC;
```

The first line of the query finds all nodes with the label ':IP' that have an ':ORIGINATED' relationship to nodes with a ':Connection' label. The second line counts up the relationships, represented by the variable 'rel' and assigns that value to the variable connections and then returns those results group by IP address. The final line sorts the results by the relationships counts in descending order.

The results for the above query completed in just under 2.5 seconds and returned the following results (truncated to the top five results for brevity).

Table 1: Connections count by IP address

ip.ip	connections
192.168.202.110	1315638
192.168.202.83	1260866

192.168.204.45	631382
192.168.202.79	539051
192.168.202.101	78626

While the above query only returns the number of outbound connections, with a slight modification as demonstrated below the query can return the number of bytes downloaded by each host.

```
MATCH (ip:IP)-[rel:ORIGINATED]-(c:Connection)
RETURN ip.ip, SUM(toInt(c.resp_ip_bytes)) AS
downloaded_bytes
ORDER BY downloaded_bytes DESC;
```

The following results were returned in just over 35 seconds, again limited to the top five results for brevity.

Table 2: Total downloaded bytes by IP address

ip.ip	**downloaded_bytes**
192.168.202.110	176243022
192.168.202.83	50912804
192.168.202.76	41323570
192.168.202.79	28226086
192.168.204.45	26309412

A brief analysis of the results of the two queries demonstrates that the number of outbound connections does not necessarily correlate with the largest amount of content downloaded.

4.2. DNS

Analysis of DNS logs can prove to be very informative to the intrusion analyst in identifying potentially malicious network traffic. Similar to the previously discussed query for outbound connections, the query below will return the count of DNS queried name sorted in descending order.

```
MATCH (dns:DNS_Record)
WHERE dns['id.resp_p'] = '53'
RETURN dns.query, COUNT(dns.query) AS total
ORDER BY total DESC;
```

This query introduces the 'WHERE' clause, which limits the results returned to those that are queries to port 53 in order to ensure results are for queries to DNS servers. The below results were returned in only 59 msec.

Table 3: Total queries by domain name

dns.query	total
44.206.168.192.in-addr.arpa	1257
www.apple.com	913
version.bind	522
creativecommons.org	367
www.dokuwiki.org	349

4.3. HTTP

Transitioning to the analysis of web traffic from the http.log file create by Bro, the below query returns the sorted counts of User Agent strings seen in HTTP requests. The query continues with general structure and content of the previous queries with a focus on web traffic analysis.

```
MATCH (web:HTTP_Record)
RETURN web.user_agent, COUNT(web.user_agent) AS total
ORDER BY total DESC;
```

The User Agent query returned the following results in 212 msec. Of note to an analyst may be the appearance of the NMAP User Agent string 6204 times in the example PCAP, indications of high volumes of scanning within the network.

Table 4: User Agent count

web.user_agent	**total**
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)	60035
Mozilla/5.0 (compatible; Nmap Scripting Engine; http://nmap.org/book/nse.html)	6204
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0)	6140
-	3194
Mozilla/5.0 (X11; Linux i686; rv:5.0.1) Gecko/20100101 Firefox/5.0.1	640

5. Conclusion

Rapid and effective analysis of large amounts of captured network traffic can greatly increase the likelihood of detection of malicious activity by an intrusion analyst. The combination of existing analyst techniques with emerging tools can provide new processes and opportunities for the analyst to identify issues faster while potentially detecting indicators unidentified by previous techniques.

The use of Docker containers to process a PCAP through the Bro Network Security Monitor, process the Bro logs using Python and pandas, ingest the logs into neo4j, and conduct queries on the data provides the analyst with new viewpoints on the data. The queries discussed throughout the process only seek to highlight the capabilities and potential of using a graph database to analyze network traffic.

6. Future Work

Future research areas to build on this process could be used to conduct near real-time analysis of network traffic as it is passed through Bro, add additional analytics for identifications of malicious traffic, and automation of graph queries and reporting. Through leveraging Bro's ability to analyze network traffic at 'wire speed' this process could be further developed to provide near real time results of queries. Additionally, the queries described above only scratch the surface of the possible information that could be extracted by analyzing the relationships associated with network traffic.

7. References

Bejtlich, R. (2013). *The Practice of Network Security Monitoring.* San Francisco: No Starch Press, Inc.

blacktop/bro. (2016, 8 24). Retrieved from Docker Hub: https://hub.docker.com/r/blacktop/bro/

Docker - Build, Ship, and Run Any App Anywhere. (2016, 8 25). Retrieved from docker: https://www.docker.com

Gallagher, S. (2016). *What you need to know about Docker.* Birmingham: Pack Publishing Ltd.

neo4j. (2016, 8 24). Retrieved from Docker Hub: https://hub.docker.com/_/neo4j/

Neo4j: The World's Leading Graph Database. (2016, 8 25). Retrieved from Neo4j: https://neo4j.com

Robinson, I., Webber, J., & Eirfrem, E. (2015). *Graph Databases - New Opportunities for Connected Data.* Sebastpol: O'Reilly Media, Inc.

Sanders, C., & Smith, J. (2014). *Applied Network Security Monitoring.* Waltham: Syngress.

Schilling, M. A. (2016). *Strategic Management of Technological Innovation.* New York: McGraw-Hill Education.

The Bro Network Security Monitor. (2016, 08 26). Retrieved from The Bro Network Security Monitor: https://www.bro.org

The py2neo v3 Handbook. (2016, 8 24). Retrieved from The py2neo v3 Handbook: py2neo.org/v3/

Van Bruggen, R. (2014). *Learning Neo4j.* Birminham: Packt Publishing Ltd.

Verizon. (2016). *2016 Data Breach Investigations Report.* Verizon.

Welcome to Python.org. (2016, 8 25). Retrieved from python: https://www.python.org

Patrick Neise, patrick.neise@gmail.com

Appendix A

pcap_to_log.py

```python
from subprocess import Popen, PIPE
import os
import re


def generate_logs(pcap_dir, pcap):
    """
    Process PCAP through a Bro container
    :param pcap_dir: Full directory path to location of PCAP
    :param pcap: PCAP filename
    :return: 0
    """
    volume = os.path.join(pcap_dir, ':/pcap:rw')
    container_name = 'bro'

    print("Starting bro container to process {}".format(pcap))

    p = Popen(['docker', 'run', '--rm', '--name', container_name,
               '-v', volume, 'blacktop/bro', '-r', pcap,
               'local'],
              stdout=PIPE)
    out = p.stdout.read()

    print("\n")
    print("{} processed.".format(pcap))
    print("\n")

    return 0


def clean_log(log_dir, filename):
    """
    Removes excess header and footer information from Bro
    logs to support import as CSV
    :param log_dir: Path to log files
    :param filename: Filename of log file
    :return: 0
    """
    print("Cleaning {}".format(filename))
    log_file = os.path.join(log_dir, filename)
    clean_file = os.path.join(log_dir, filename + '.clean')

    with open(log_file) as original:
        lines = original.readlines()

    with open(clean_file, 'w') as cleaned:
        cleaned.writelines(lines[6][8:])
        cleaned.writelines(lines[9:-1])


    # this is to fix errors in http log due to random "
    in some of the User Agent strings
    if filename == "http.log":
        with open(clean_file, "r") as sources:
            lines = sources.readlines()
        with open(clean_file, "w") as sources:
            for line in lines:
                sources.write(re.sub(r'\t"', '\t', line))

    return 0

def main():
    base_dir = os.getcwd()
    data_dir = os.path.join(base_dir, "data")
    pcap = 'maccdc2012_00000.pcap'
    logs = ['conn.log', 'dns.log', 'http.log']

    # convert pcap into bro logs
    pcap_to_log(data_dir, pcap)

    # remove excess header and footer from each bro log
    for log in logs:
        clean_log(data_dir, log)

if __name__ == '__main__':
    main()
```

Appendix B

logs_to_csv.py

```
from subprocess import Popen, PIPE
import os
import re


def generate_logs(pcap_dir, pcap):
    """
    Process PCAP through a Bro container
    :param pcap_dir: Full directory path to location of PCAP
    :param pcap: PCAP filename
    :return: 0
    """
    volume = os.path.join(pcap_dir, ':/pcap:rw')
    container_name = 'bro'

    print("Starting bro container to process {}".format(pcap))
    print("NOTE: This can take some time depending on size of
pcap")
    print("NOTE: bro will display Errors in screen")
    print("\n")


    p = Popen(['docker', 'run', '--rm', '--name',
container_name,
               '-v', volume, 'blacktop/bro', '-r', pcap,
               'local'],
              stdout=PIPE)
    out = p.stdout.read()

    print("\n")
    print("{} processed.".format(pcap))
    print("\n")

    return 0


def clean_log(log_dir, filename):
    """
    Removes excess header and footer information from Bro logs
to support import as CSV
    :param log_dir: Path to log files
    :param filename: Filename of log file
    :return: 0
    """
    print("Cleaning {}".format(filename))
    log_file = os.path.join(log_dir, filename)
    clean_file = os.path.join(log_dir, filename + '.clean')

    with open(log_file) as original:
        lines = original.readlines()

    with open(clean_file, 'w') as cleaned:
        cleaned.writelines(lines[6][8:])
        cleaned.writelines(lines[9:-1])


    # this is to fix errors in http log due to random " in some
of the User Agent strings
    if filename == "http.log":
        with open(clean_file, "r") as sources:
            lines = sources.readlines()
        with open(clean_file, "w") as sources:
            for line in lines:
                sources.write(re.sub(r'\t"', '\t', line))

    return 0

def main():
    base_dir = os.getcwd()
    data_dir = os.path.join(base_dir, "data")
    pcap = 'maccdc2012_00000.pcap'
    logs = ['conn.log', 'dns.log', 'http.log']

    # convert pcap into bro logs
    pcap_to_log(data_dir, pcap)

    # remove excess header and footer from each bro log
    for log in logs:
        clean_log(data_dir, log)

if __name__ == '__main__':
    main()
```

Patrick Neise, patrick.neise@gmail.com

Appendix C

csv_to_neo.py

```
from subprocess import Popen, PIPE
import os
import time


def start_neo4j(data_dir):
    print("Starting neo4j container...")
    volume = os.path.join(data_dir, ':/var/lib/neo4j/import')

    p = Popen(['docker', 'run', '--publish=7474:7474', '--
publish=7687:7687',
               '-v', volume, '--name', 'neo4bro', '-d',
'neo4j'], stdout=PIPE)
    out = p.stdout.read()
    print("neo4j container running")
    time.sleep(5)

    return 0


def run_import():
    print("Importing nodes and relationships into neo4j...")
    p = Popen(['docker', 'exec', 'neo4bro', '/bin/bash',
'/var/lib/neo4j/import/import_connection.sh'],
              stdout=PIPE)
    out = p.stdout.read()
    print("Import complete")

    return 0

def restart_neo4j():
    print("Restarting neo4j container...")
    p = Popen(['docker', 'restart', 'neo4bro'], stdout=PIPE)
    out = p.stdout.read()
    print("neo4j container restarted")


def main():
    base_dir = os.getcwd()
    data_dir = os.path.join(base_dir, "data")

    start_neo4j(data_dir)
    run_import()
    restart_neo4j()


if __name__ == "__main__":
    main()
```

Appendix D

bro_to_neo.py

```
import pcap_to_log
import logs_to_csv
import csv_to_neo4j
import os


def pcap_to_clean_logs(data_dir, pcap):
    """
    Read in pcap that is located in data_dir and process
through Bro container to crete Bro logs.
    Process Bro logs into clean csv files for import in noe4j.
    :param data_dir: Directory where pacp is located, relative
to location of script.
    :param pcap: Filename of the pcap to run through Bro.
    :return: 0
    """
    # convert pcap into bro logs
    pcap_to_log.generate_logs(data_dir, pcap)

    # remove excess header and footer from each bro log
    logs = ["conn.log", "dns.log", "http.log"]
    for log in logs:
        pcap_to_log.clean_log(data_dir, log)

    return 0


def import_and_run_neo4j(data_dir):
    """
    Start the neo4j container, execute shell script on the
container to import nodes and relationships created from
    Bro logs and then restart the neo4j container.
    :param data_dir: Directory that contains node and
relationship csv's for import in neo4j
    :return: 0
    """
    csv_to_neo4j.start_neo4j(data_dir)
    csv_to_neo4j.run_import()
```

Patrick Neise, patrick.neise@gmail.com

```
    csv_to_neo4j.restart_neo4j()

    return 0

def main():
    base_dir = os.getcwd()
    data_dir = os.path.join(base_dir, "data")
    pcap = "maccdc2012_00000.pcap"

    pcap_to_clean_logs(data_dir, pcap)

    logs_to_csv.create_nodes_relationships(data_dir)

    import_and_run_neo4j(data_dir)


if __name__ == '__main__':
    main()
```