**Intrusion Detection through Graph Analytics**

Harsh Khandelwal and Milind

*Department of Computer Science*

*University of Illinois, Chicago*

**I. ABSTRACT**

Graph based analysis for finding out application vulnerabilities and intrusion detection is a popular approach in the field of computer security analysis. We propose a new framework for using graph analytics for intrusion detection that relies on querying the application data, stored in the form of a graph database and using visualization techniques for intrusion detection.

**II. INTRODUCTION**

Application data analysis and security analysis are the areas which continue to grow in terms of complexity, scale and maturity. During the past few years, the use of better and faster technologies has resulted in the ease of development of new system applications and web applications which has resulted in increased data feeds, number of users, number of computers, and, consecutively, a prodigious growth in the number of threats that security and information technology professionals must evaluate, prioritize, and mitigate. With this ever growing increase in the need of system security, a lot of attack detection and prevention techniques that use data analytics techniques, have been proposed to pacify this need. Most of the existing analytics techniques used for intrusion detection are too inflexible to adapt to new behaviors or attacks that happen on a daily basis.

We propose a system framework which specifies a data model for storing large amounts of information logged by applications and implement analytics techniques for supporting intrusion detection. The system implements graph based analytics using Neo4J graph database as it not only provides an efficient way for storing logged information, it also provided a more efficient way of visualization through queries. The system works both for system applications and web applications the system takes input data in the form of a csv file. For system applications, this file contains information about system calls such as call times, process names, process ids, system arguments and such. For web applications, information about requests, responses, databases and servers and such are stored in the file. We then map graph nodes, properties and edges with the data in the file according to the specifications of the data model that we use. The data file is then loaded into in to the system and according to the specified mappings, the data is stored in the form a graph in Neo4J. The system then provides a way to query the data and gather relevant information for raising an Intrusion Detection alert. A detailed approach is given in section IV.

The rest of the report is organized as follows: Section III highlights the related work. Section IV and V talk about the system implementation and a detailed overview of the techniques used for Intrusion Detection. Experimental evaluation results are presented in Section VI. Section VII concludes the report. A list of figures has been put in the Appendix I for reference.

**III. BACKGROUND & RELATED WORK**

In this section, we present several works that have been done on using graph based analysis for intrusion detection. For instance, [1] presented a new scheme for logging system calls for binary programs. [2] and [3] present system call logging techniques that treat processes as subjects, and files, sockets and other passive entities as objects. [4] talks about using taint propagation analysis as a way to implement security policies for intrusion detection. Taking inferences from the central ideas of these papers helped us to define properties of the graph nodes used in our system model and map relationships between them. Also, we worked on removing the limitations of the above mentioned techniques and tried to combine these techniques to arrive at a practical, scalable approach to intrusion detection using graph based analysis.

**IV. APPROACH**

We divide this section into sub sections with the first describes about the Common Data Model we followed and give an introduction to Graph databases. In the subsequent sections, we provide a detailed overview of the approach used for our system.

**IV.I Common Data Model**

To start we take a reference of a common data model. This model gives a general description of all the components a general web or desktop application can have. From the data model, our approach is concentrated over using three components i.e. Subjects, Objects and Events.

**Subjects:** Subjects are active entities that execute actions. This category includes the browser in the client (or components of a web page), web-server, app-server, script-interpreter, query-engine in the database, and so on. Each subject is associated with the following information:

i. *Principal* which started the *subject*.

ii. *Executable*. If we consider the portions of a web page, e.g., iframes, forms, as subjects (If we are tracking at that level of granularity), the executable is the script or the iframe source.

**Objects**: Objects are entities that execute no actions. They include database-tables/columns, files, database -connections, request-socket, response-socket, included PHP files, URI resources.

Each object is associated with the following information:

i. Location. This depends on the type of the object. For files, it is the path, for database tables the names of the database and of the table, for the components of a web page, the URI or path of the component.

ii. The permissions associated with the object. Every permission is a relationship of a certain type (r-w-x, update, insert and select) with a principal.

iii. Tag. The object tag contains information about the integrity and confidentiality of the object. In addition, it contains pointers to the objects that the current object was derived from.

**IV.II** **Graph Databases and Neo4J**

Graph databases are databases that use graph structures for semantic queries. From a technical point of view, semantic queries are precise relational-type operations much like a database query. They work on structured data and therefore have the possibility to utilize comprehensive features like operators (e.g. >, < and =), namespaces, pattern matching, subclassing, transitive relations, semantic rules and contextual full text search.

These are NoSQL type databases in nature which use nodes, edges and properties to represent and store data. Different nodes represents the different categories of data in the database. Edges represent the how the different nodes are connected. Properties are the additional characteristics possessed by both the nodes and the edges.

Essentially, Graph databases are key-value databases with an additional concept of **relationship**. Relationships allow value in the store to be related to each other in a free form way. They allow for complex hierarchies to be quickly traversed, addressing one of the more common performance problems found in traditional key-value stores.

The graph database we use for our research is Neo4j. It is the most popularly used graph database and provide a good interface to carry out our approach.

**Neo4j** is an ACID-compliant transactional database system that provides native graph storage and processing. The storing and fetching values from database and graph processing via a special native query language Cypher.

Our selection criteria for choosing Neo4j was its relatively well documentation as opposed to its counterparts and a very active stack overflow community. They helped a lot in understanding the tricks that are involved in Neo4j.

**Cypher** is a declarative graph query language native to this database. This simple yet very powerful language allows expressive and efficient querying and updating of the graph store. Complicated queries can be easily expressed via Cypher. Cypher supports a variety of SQL like clauses: MATCH, WHERE, CREATE, DELETE are some of them. In Neo4j, cypher denotes a relation in the following form:

(NODE) - [:RELATIONSHIP] 🡪 (NODE)

Where the arrow head indication direction is optional.

**IV.III Overview**

In our approach, we take system calls data from a wide variety of web and desktop applications including web servers, music players, etc. This data is of the form of a CSV file. Figure 1 shows a sanitized view of an input CSV file which has been cleaned to make it easier to load to Neo4j. More so the extraneous characters have been removed. From this input file we classify the columns as follows:

1. Subjects include process name, process ids (pid), thread ids (tid), etc.

2. Objects include path locations and memory addresses. Columns arg1 and arg2 are the objects

3. Events include system calls (syscall), return values (ret\_val), return time (ret\_val) and call time (call\_time).

Once we have the classification ready, our next step is to identify the node mappings and properties that would exist in the graph in accordance with the common data model. From the data model we define three major relationships:

1. (Event)-[**:IS\_GENERATED\_BY**]🡪(Subject)

2. (Event)**-**[:**AFFECTS**]🡪(Object)

3. (Event)-[:**AFFECTS**] 🡪(Subject)

The next step in our approach would be to make use of a native Neo4j feature of loading the csv using a file type URL. Since such a functionality is inherent to Neo4j, it makes it easier on the effort to not write a compatible customized import API. Although, for our work we did needn’t write the API, it is useful to have one in case of dealing with a large dataset (order of hundreds of thousands) as the load csv from file functionality is considerably slow in comparison. The API with its own caveats, its discussion is not important here.

Once we have the data ready in our database, we can query based on various criteria’s to understand the points in the graph which can be a potential source of attack. The cases we tested out are discussed next.

**V. TECHNICAL DETAILS**

This section gives an insight on what all queries we ran and what was the idea in deciding upon using them in the effort to understand whether these can be used to detect attack patterns and also find anomalies within applications.

As previously described, the input data was first loaded to the Neo4j database. We used the query in Figure 2 to load the csv data into the database. The input set contained of about 200 rows of data. We took only a subset from the dataset to make sure we would not have to wait for long duration of time to load the complete data sets each of which had about 500,000 records. Since the version we used for our run was the Neo4j community edition, this provides a single node only which makes it practically not feasible for dealing with such huge datasets. Also a scaling feature of the graph is a limitation. Hence it was wiser for us to stick to a small subset of data for visualization of our analysis. Figure 3 shows the graph that is rendered for this particular dataset load. The MATCH query below:

**Match (n:Event)-[r]->(m) return n, r, m**  
  
matches all the nodes that meet the connection criteria as per what is described in the query. It is important to return the result of any condition to have it rendered on the user screen.

Using the cypher query structures, we were able to come up with a generalized set of queries. These queries can be used at any graph data structure to understand the connections between nodes through labels (edges). Few of those queries were:

i. Finding the depth of graph

ii. Finding the neighbours of a particular node

iii. Finding Paths from one node to another

iv. Degrees of separation between different nodes

v. Graph diameter

vi. In-degree and out-degree of a particular node

vii. Construct an adjacency matrix of the graph for detailed analysis

**Attack Detection via Visualization**

Using these queries we were able to find out a class of attack where a process had spawned a new thread when being the victim which was not the case in the benign run of the application while training. One set of query which helped in finding out whether a new thread had spawned for a process checked whether the thread id is different from a process id. If yes, that node would require a detailed evaluation by looking towards its’ immediate object and event nodes to find out whether any sensitive operation was run. In our case we compared a benign and a malicious instance of a process. In the malicious run a fork system call was made at a point in the program. We used the following query to find out the subject node where this behavior was seen first:

**Match (s:Subject) where s.Pid <> s.tid  
return (s)**

Once this subject was returned we found out all neighbors of this node using the following query

**match (n:Event)-[r]->(s:Subject) , (n:Event)-[r1]->(o:Object)**   
**where s.Pid <> s.tid return n, r, s, o**

The output of the above two queries are shown in Figure 3 and Figure 4. Figure 3 shows a ‘python’ node where a tid is not equal to pid. Figure 4 shows the neighbours of the python node. On further investigating, we find that this node was related to an ‘open’ system call event that called open towards an object node that had an argument property of the password file. We also mark such subject as tainted equivalent to true by adding the property through the same query.

When we compared the complete graph of the benign trace we saw only one subject node named python in the graph but in this case where the application is compromised, we found two in the graph. This way graph analytics helped us on determining an attack.

**VI. EVALUATION**

The proposal is presented more as a proof of concept than an actual system. However, we did some query performance evaluation for Neo4J. The tests imply the performance of the query runs on a system running on Intel i7 chip with a 2.8GHz processor using different types of data gathered from multiple applications. We were able to record the query response timings for an input set of 1000 system calls. The query for loading the data into Neo4J ran for about 2 seconds. With a licensed version of Neo4J, this runtime can be reduced to *<1s*. All other queries running on the graph had a run time between 200 – 600 milliseconds depending on the query complexity and the data returned by those queries.

**VII. CONCLUSION**

Through our work, we just presented a basic framework implementation of the idea of using graph analytics for intrusion detection. The proposed system works well for both web and desktop applications with the only human involvement of mapping the nodes and relationships among them according to the common data model described before. The framework is very flexible and new queries can just be added according to new intrusion detection policies at any time and added to the pre-existing model. The proposal is not very practical for implementation in a live setting as a lot of effort is still required to improve the query performance for loading very large datasets into the database. Overall, the system provides a very good visualization tool for understanding and preventing a plethora of different attacks.

**VIII. REFERENCES**

[1] Kyu Hyung Lee, Xiangyu Zhang, Dongyan Xu: High Accuracy Attack Provenance via Binary-based Execution Partition

[2] S. T. King and P. M. Chen: Backtracking Intrusions

[3] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M Whang. Provenance-aware tracing of worm break-in and contaminations: A process-coloring approach.

[4] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A practical approach to defeat a wide range of attacks

[5] Neo4J Manual v2.3.3

[6] Neo4J Lynda Tutorial

[7] Neo4J with Cypher: Coursera Tutorial

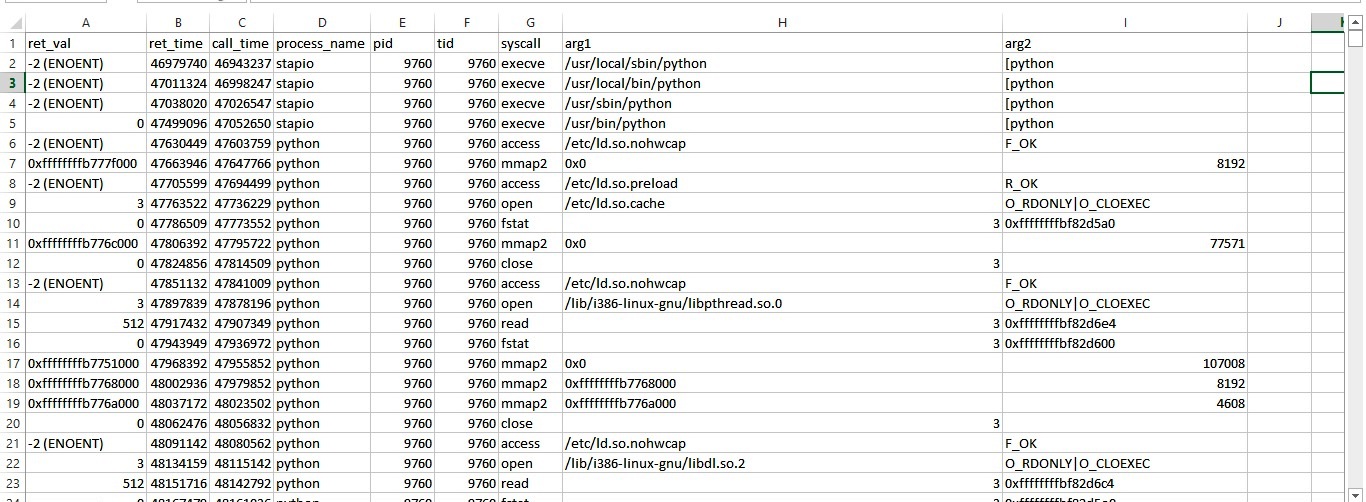
**IX. APPENDICES**

Appendix I: List of Figures

Appendix II: Individual Contributions

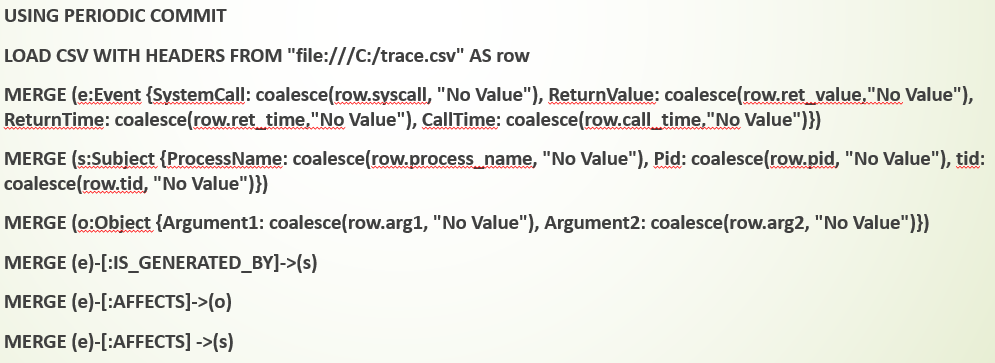
**Appendix I: List of Figures**

**Figure 1:**

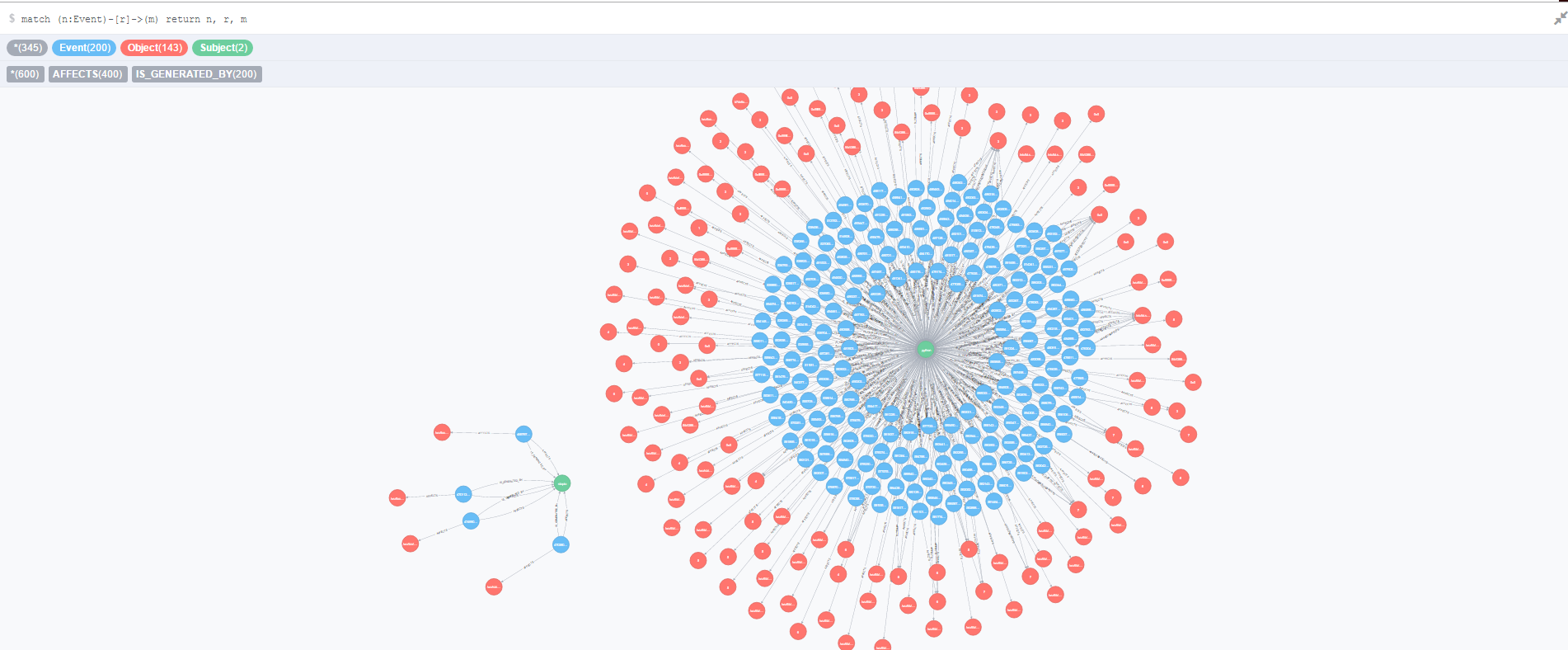


**Figure 1: Input data Sample after cleaning to make it compatible with Neo4J. We implemented the cleaning manually but another easy way is through pre-defined power shell script functions for editing files.**

**Figure 2:**

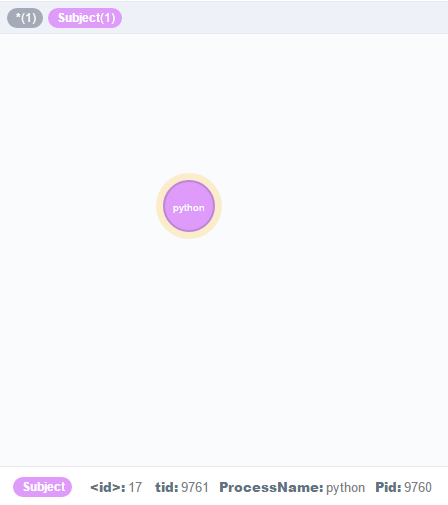


**Figure2a: Query to Load the data in csv file to the graph db**

****

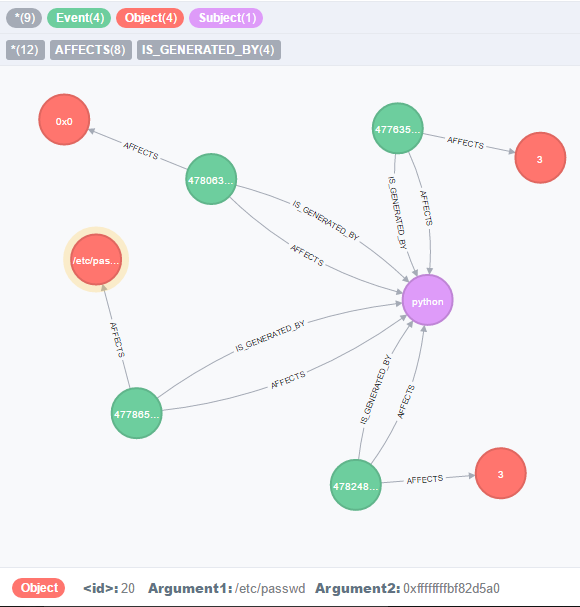
**Figure 2b: Output of the query in fig. 2a visualized in Neo4J**

**Figure 3:**

****

**Figure 3: Output of the query run Match (s:Subject) where s.Pid <> s.tid return (s)**

**Figure 4:**

****

**Figure 4: Output of the query for finding the neighbours of subject node under attack**

**Appendix II: Individual Contributions**

First, we thank **Prof. Venkat**, **Birhanu Eshete** and **Rigel Gjomemo** for their valuable guidance and support throughout the semester.

The individual contributions of the team members are:

**Milind**

* Set up the Neo4J framework on the test systems
* Defining nodes, properties and relationships for the data captured for web applications
* Writing general queries for the graph, referencing the online tutorial

**Harsh Khandelwal**

* Defining the graph components for system/desktop applications
* Defining query criteria for attack detection
* Writing queries for attack detection