5-2019

Using Graph Databases to Address Network Complexity Problems that can Hinder Security Incident Response

Andrew Erickson
eran1005@stcloudstate.edu

Using Graph Databases to Address Network Complexity Problems That can Hinder Security Incident Response

by

Andrew Erickson

A Thesis

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Information Assurance

June, 2019

Thesis Committee:
Dennis Guster, Chairperson
Jim Chen
Changsoo Sohn

Abstract

The network complexity problem within computer security incident response is an issue pertaining to the complexity of a computer network as it grows in both size and scale. The larger the computer network grows, the more difficult reconnaissance becomes, which is necessary to execute correction and prevention measures that address issues that arise during security incident response. Leveraging graph databases can help solve problems present in relational databases with large, tree-like structures, like those present in computer networks, and along with solving those problems adds flexibility that is needed due to the mutability of computer networks. This paper focuses on using graph databases to discover the blast radius of day zero vulnerabilities on the fly by using the properties of graph databases to find intuitive infection vectors that may be present during a day zero vulnerability. Additionally, options for visualizing security data in ways that make the data more actionable will be explored.

Acknowledgments

I would like to thank everyone who helps make the BCRL possible at St. Cloud State University. It's a wonderful teaching tool that has taught me many invaluable lessons about computers and computer networks.

Table of Contents

Chapter Page

List of Tables

List of Figures

Figure Page

Chapter 1: Introduction

Introduction

Computer security incident response (SIR or CSIR) is a must for any large organization that touches modern technology; in today's day and age, that is almost all of them. The larger an organization grows, the bigger its networking and computing overhead becomes, and the harder it gets to perform security incident response tasks. The main idea behind this is that, in general, more complicated the computer network results in more difficulty in performing security incident response tasks due to the highly unique relationship between different computer networks within a business.

Solving the complexity issue with this problem begins to be apparent when looking at how a security incident response is handled in a general sense. If a CSIR event occurs, the first step is often tracking down the computer, server, or system affected by the event before determining how the device was infected. The resulting question of who within the network is also potentially vulnerable to this CSIR event and might be a potential risk for the event spreading to another computer or internal network is a much larger and more complicated question to answer than the first. Much like the facial or object recognition problem in computer science, the problem of predicting the impact and reach of a CSIR event is a particularly challenging problem.

One of the best real-world examples of the problems faced by large networks and CSIR events is in the case of ransomware. The initial threat vector may be identified easily, but where the ransomware may be able to spread once within the network is much harder to work out. Examples of this threat can be seen in the

WannaCry (2017) and Petya (S., 2017) attacks, where large numbers of computers both inside and outside of large organizations were compromised. Remedying this challenge relies on having a solution in place that can make determining the blast radius of these events quick and painless. This way, the organization's defenses can be modified to protect against the threat and prevent proliferation to other segments of the network.

Solving this problem can be broken into two main parts: constructing and maintaining a data pool and visualizing the data so that it is actionable. Constructing and maintaining the data pool consists of gathering and maintaining security data relevant to that organization's network (e.g. IP addresses, domain information, DNS information, and other relevant data). Accessing and using that data is a large part of addressing CSIR events, even though it often does not paint a readable, easy-to-understand picture of the affected devices and the problem itself. Visualizing that data is another problem, as there is a multitude of methods for visualizing the data so that it is understandable as well as actionable.

Problem Statement

Complex computer networks lead to large amounts of reconnaissance when a CSIR event occurs. This often requires the use of multiple skilled personnel ranging from server teams, Windows and Linux specialized teams, networking, firewall, and more depending on the type of CSIR event. Each of these teams will spend time working on different aspects of the problem presented by the CSIR event requiring their expertise. The time spent doing reconnaissance results in slower response times, which

are critical during times when zero-day vulnerabilities are found exposed within the network.

Nature and Significance of the Problem

Addressing the computer networking problem is a large task requiring the integration of cross-discipline data into an easy-to-access structure that can be leveraged by CSIR teams during CSIR events. This structure must also be actionable and understood by other teams that may be recruited to help solve CSIR events. Leveraging the properties of graph databases allows for quick analysis of potentially vulnerable network segments and can be used to reduce the reconnaissance time needed during a zero-day vulnerability disclosure. Along with helping to reduce reconnaissance times, graph databases have a plethora of visualization integrations available to help produce easily interpreted visuals that condense security data into an understandable format.

Objective of the Study

The objective of this study is to create a method that leverages an existing framework for mapping network topology, relationships, and security vulnerability data, as well as extending the method to help reduce reconnaissance times during zero-day vulnerability response. Additionally, an exploration of visualization options available for graph data and available frameworks to create those visualizations will also be explored.

Hypotheses

Given a graph database's ability to select nodes and edges based on a specific set of criterion, it should be possible to select and modify specific nodes that would be affected by a zero-day vulnerability, modify their relevant information within the graph database, and run queries that elucidate the potential infection vectors possible from the given zero-day.

Limitations of the Study

One major limitation of this study is that sufficient real-world network data is difficult to programmatically plug into a graph database and access. As a result, a mock architecture was used and some of the nuances present in real-world networks may be abstracted. One such example can be seen in the bloat from legacy software and other compounding decisions that build on themselves. In the test set up used in this study, there is no legacy software or architecture choices that affect the network and subsequently the graph.

Another limitation is the scope of zero-day vulnerabilities to choose from and test against in the test network. Each vulnerability needs its corresponding vulnerable node to be tagged with the relevant vulnerability information. This is not the case when reading into a graph database from existing sources, as the information is already stored within the existing data source and is simply copied over. In this study, one zero-day is simulated due to the overhead required for adding node data and other relevant information unique to each zero-day.

The visualization options available will be analyzed from a potential use standpoint with the goal being to outline the available options and their use cases, the methods for populating those visualization options, their particular dependencies, and their available integrations with the graph database chosen in this paper, Neo4j.

Definition of Terms

CSIR–computer security incident response–Generally, the response team reacts to security issues relating most often to malware, viruses, trojans, ransomware, or other methods that could lead to the compromise of a company's data, systems, or personnel. The CSIR team is responsible for addressing, mitigating, reporting, and repairing the damage caused by the aforementioned security incidents.

CVE–common vulnerabilities and exposures–Refers to a known weakness in existing software, hardware, or computer device that could potentially be used to circumvent intended operating methods. CVE utilizes a scale ranging from innocuous to extremely dangerous.

Blast Radius–Used to describe how far reaching an infection or attack is on the computer network.

The Network Problem–The larger a computer network gets, the more difficult it is to narrow down the problems and connections present within that network. An easy to understand example is looking at virtualization. Many web servers hosted on virtual machines may be tied back to and hosted on one physical server. If one of those virtual machines is compromised, it may be necessary to include the physical server itself as a potential victim when considering the blast radius of a CSIR incident.

Summary

In this section, the network problem that is faced by CSIR teams is introduced and the hurdles met by CSIR teams in addressing that problem are made apparent. In the next chapter, we review literature pertaining to some of the key components involved in solving the network problem, mainly focusing on graph databases and other works addressing parts of the network problem and using graph databases to solve it.

Chapter 2: Background and Literature Review

Introduction

In this section, we look to provide a brief overview of graph databases followed by an analysis of the overhead present in attacking the networking problem using relational databases compared with their graph counterparts. From there, we will look at how the network problem in a CSIR lens is addressed in a paper presented by Noel, Harley, Tam, and Gyor (2014) before introducing data visualization and the various methods of doing aforementioned data visualization. From there, it is possible to move forward to creating a method for addressing zero-day vulnerabilities using the framework provided by the aforementioned authors.

Background Related to the Problem

In-depth coverage of the specific kinds of graph databases available for use and their differences has been covered at length by Angles (2012). Similarly, the differences between those available graph databases from a design perspective and outlines for specific use cases are defined by Buerli (2012). In their paper Buerli (2012) does an excellent job explaining the structural differences present in the various graph databases in the following paragraph:

> Rather, each graph database is optimized for a specific set of task or queries. The problem resides in the multiple divisions of graph databases. Graph databases can focus on graph algorithms like shortest path queries and subgraph matching which require the whole graph to reside in memory and make distributed systems very difficult. On the other side of the spectrum, a graph

database can focus on handling large graphs by scaling horizontally. This however makes many graph algorithms extremely inefficient or even impossible. Graphs can also focus on either online querying where low latency is required, or offline querying where larger data is handled. (p. 3)

The above paragraph shows that the choice of graph database should match the use case required of that graph database. The framework relating to addressing the network problem designed and outlined by Noel et al. (2014) uses Neo4j. This choice is a good fit for their framework as Neo4j is "very efficient in graph traversals" (Buerli, 2012). Along with efficient traversals, Neo4j has a few other important features needed for their framework, which can be seen in Angles' Table 1 (2012) below:

Table 1

Comparison of Graph Database Features

	Graphs				Nodes		Edges		
Graph Database	Simple graphs	Hypergraphs	Nested graphs	Attributed graphs	Node labeled	Node attribution	Directed	Edge labeled	Edge attribution
AllegroGraph	•				•		•	•	
DEX				•	•	•	•	•	•
Filament	•				•		•	•	
G-Store	•				•		•	•	
HyperGraphDB		•			•		•	•	
InfiniteGraph				•	•	•	•	•	•
Neo4j				•	•	•	•	•	•
Sones		•		•	•	•	•	•	•
vertexDB	•				•		•	•	

Note: Reprinted from *A comparison of current graph database models,* by Angles. Retrieved from 2012 IEEE 28th International Conference on Data Engineering Workshops (pp. 171-177) Copyright 2012 by IEEE.

In this table, we can see Neo4j is an attributed graph and most importantly is capable of labeling nodes, attributes, edges, and edge attributes. All of these can be used to better outline the network being defined within the database. The key advantage Neo4j has over similarly attributed graphs like DEX, InfinitGraph, and Sones is that Neo4j is transactional and thus lends itself well to larger applications and organizations where multiple users may be utilizing it simultaneously.

The network problem as briefly described above will be analyzed further in the following paragraphs. To again briefly summarize the problem, the initial infection vector or cause of the CSIR event needs to be investigated when a CSIR event occurs. Investigating this problem can prove to be problematic when a network is of sufficient size and complexity. Following off the example of the network problem within the definition of terms, we can extend this example to help elucidate further hidden problems presented by the network problem. In the above example, a web server hosted on a virtual machine was compromised and the physical hardware hosting that virtual machine was found. Most organizations will have a relational database or a drawn network diagram that keeps track of the hierarchy of ownership so that tracking down what infrastructure is used by a specific application is fairly easy to find.

The problem, however, becomes apparent when asking “What other applications similar to the compromised host live in a space that could potentially lead to their compromise?” is necessary. The structure of the relational database is effective for cases where there is a need to find one instance of a hardware server owning a virtual machine, or the virtual machine owning a web server. When we add in more complex

questions like needing to know what hardware owns web servers or virtual machines and has a valid network connection to the compromised system that was first identified there is a massive failure due to the complexity of the question. Even more so the problems become more apparent when the levels of the ownership tree above vary in depth. In the above example of hardware → virtual machine → web server, we only have a depth two search required, but many situations call for much more depth in the search such as in a case where: network hardware → physical host hardware → virtual machine host → virtual machine → docker container. Going past a parent-child depth relationship of four leads to nearly unresolvable search times in relational databases as can be seen in Table 2 from Robinson, Webber, and Emil (2015) below.

Table 2

Comparison of Relational and Graph Database Execution Times

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Note: Reprinted from *Graph Databases* by Robinson, Webber, & Emil, Retrieved from the book *Graph Databases* Copyright 2015 O'Reilly & Associates.

This failure is the first issue that crops up in the network problem and is a difficult problem for relational databases to solve. This problem is difficult due to the number of table joins that are needed to be performed along with the number of tree traversals required to answer the aforementioned question. It can be seen above that just due to

the nature of the tree traversals required relational databases have a difficult time addressing just the parent-child relationships present in the hardware finding portion of the question above. To then take that information and cross-reference it with a network diagram or network devices table constructed within a relational database is another slow task that may require tree traversal down a hierarchy of networking devices. Again, here we hit a depth problem and need to be careful about asking questions that require tree traversals that exceed a depth of four. As we can see the questions asked by CSIR teams can possibly be answered by a relational database, but there are glaring problems present when the depth exceeds four. This happens to be the case in most large organizations and as such makes leveraging a relational database for CSIR events difficult.

A potential solution to the problems presented above is the graph database. Graphs are defined as: “Formally, a graph is just a collection of vertices and edges—or, in less intimidating language, a set of nodes and the relationships that connect them.” (Robinson et al., 2015) An example of a simple graph structure present in a social media situation can be seen below in Figure 1:

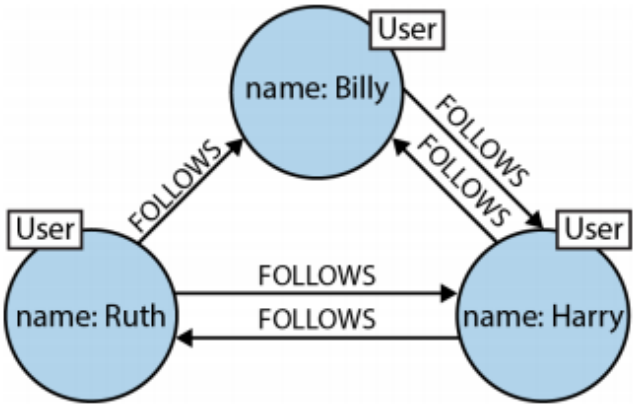


Figure 1. Simple example of nodes, edges, & relationships (Robinson et al., 2015).

This simple example above can be a useful tool for understanding one of the most important features of a method used by many graph databases to perform faster searches. The tool those particular graph databases use is something called index-free adjacency. The basic idea is that each node has a pointer stored within it that points to nodes it is related to. From the above graph, we can understand there would be pointers from Harry to Ruth and from Ruth to Billy, but there would be no pointer in the reverse case from Billy to Ruth. In order to resolve that relationship, we would need to resolve the pointer from Billy to Harry and then from Harry to Ruth. Since the information of who is connected to who is tied directly to the node, no index is required to gather that information and instead it is based solely off of information that is gathered based on adjacency. Since indexing is not used traversing depths becomes much easier and faster when compared with a relational database. As such, it can be seen how a graph database addresses one of the key concerns for solving the network problem as depths over four do not take infeasible amounts of time since graph databases use index-free adjacency.

Another problem potentially addressed by graph databases is converting a network or application diagram into a format useable by a graph database. When using a relational database this can be a complicated process involving table creation and normalization. We can avoid this entirely with a graph database and take our whiteboard drawing of the network or application and import it into the graph database directly. A sample of an application diagram from Robinson et al. (2015) can be seen below:

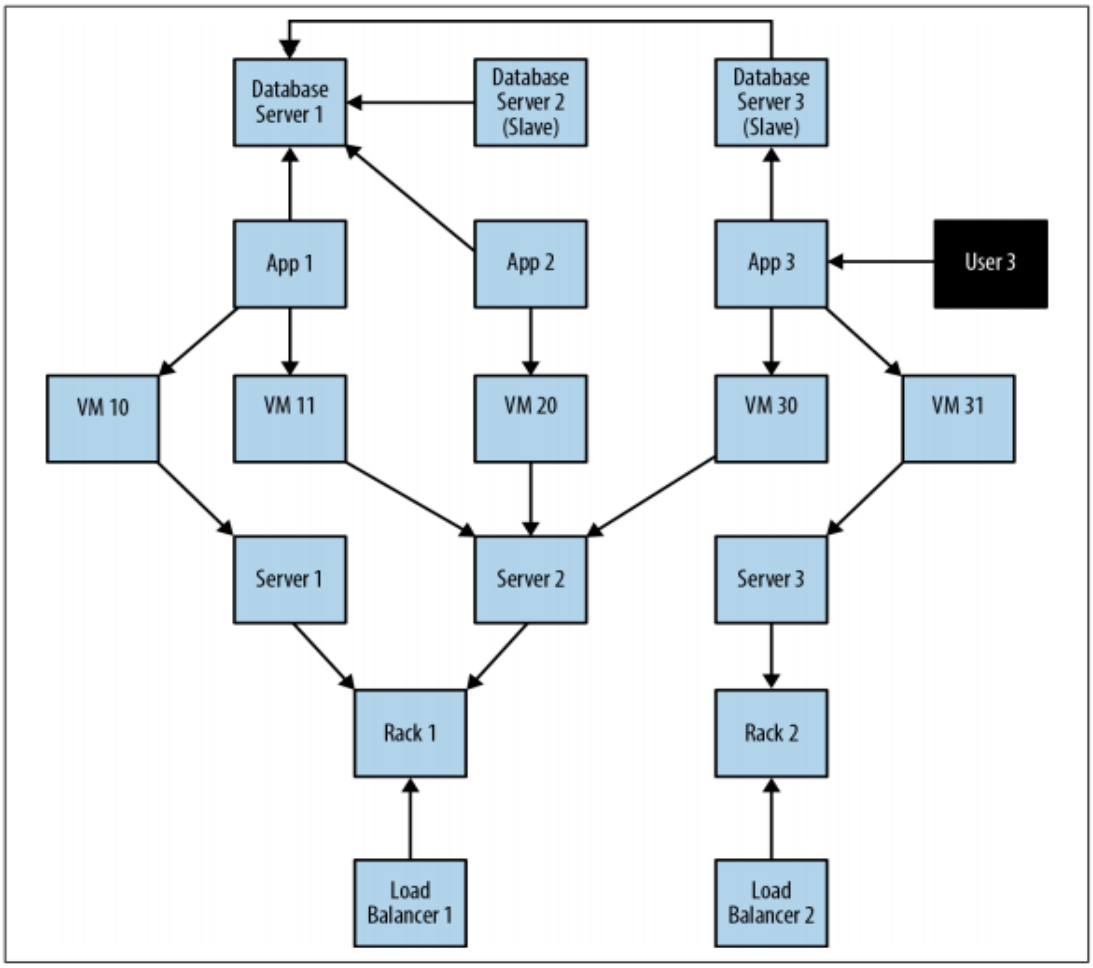


Figure 2. Example application diagram and supporting architecture (Robinson et al., 2015).

Taking this whiteboard sketch for the application and transforming it into a graph database would simply result in a nearly identical layout but populated with metadata pertaining to the application. This idea from Robinson et al. (2015) can be seen below.

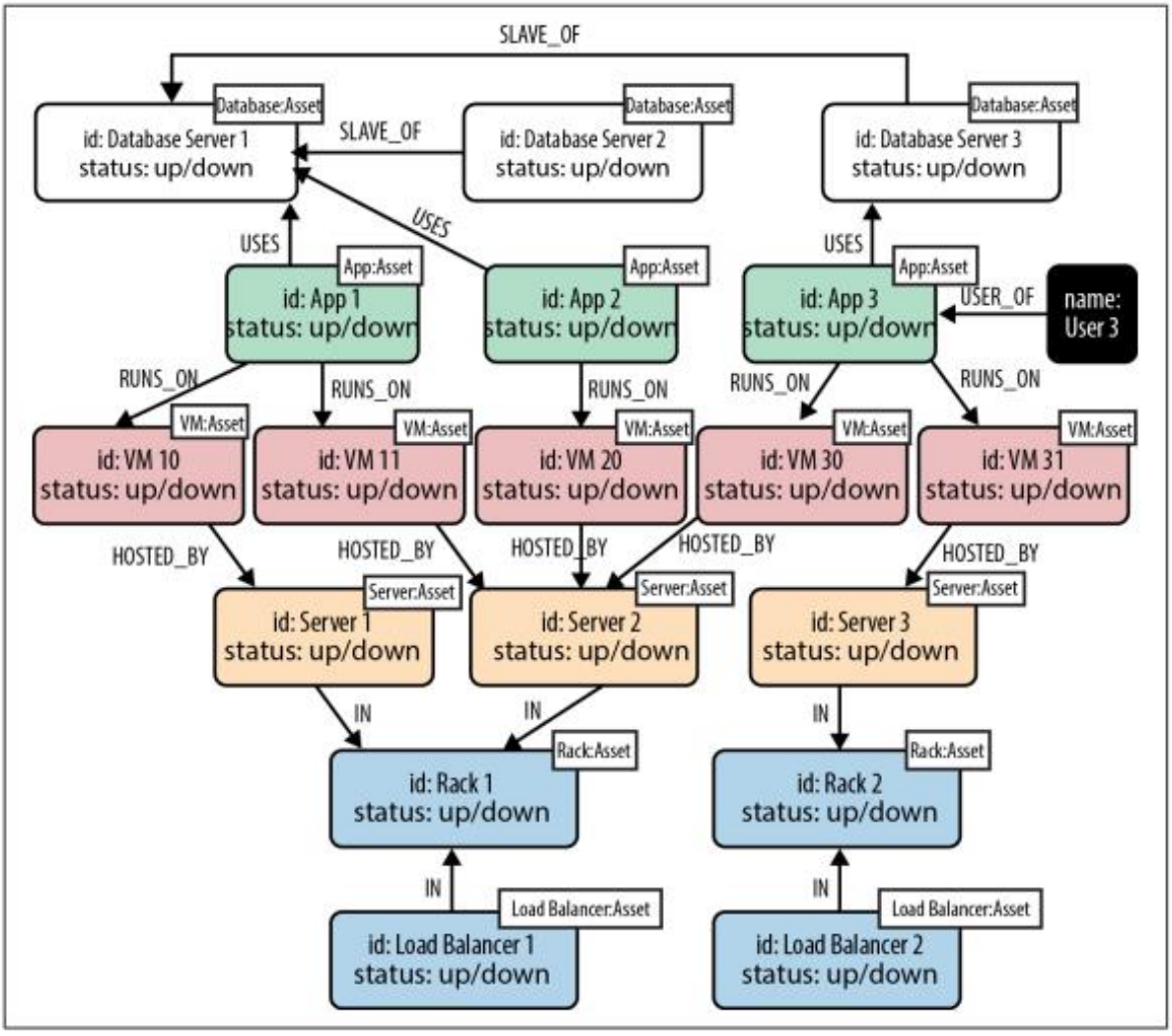


Figure 3. Post transformation to graph database application diagram (Robinson et al., 2015).

The beauty of the graph database is that the application owner knows where things are, but it can easily be queried and manipulated by someone seeking information who isn't the app owner. Similar data centers can be added to the graph

and connected to each other nodes and edges based on their relationships to the others. In this way, it is possible to see the relations between different deployments and makes their corresponding dependencies clearer.

The full evolution of solving this problem in a CSIR context can be seen in the framework presented by Noel et al. (2014). In figures 4, 5, and 6 from their paper shown below:

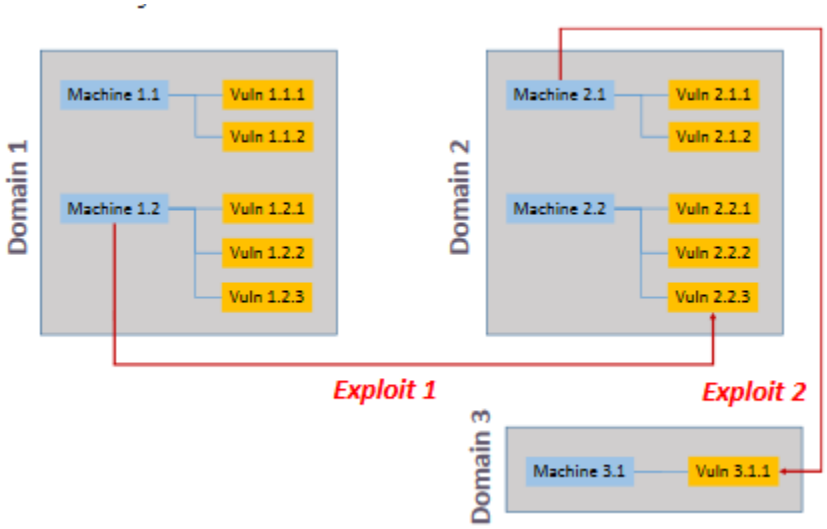


Figure 4. Cross domain exploit chains leveraging existing vulnerabilities (Noel et al., 2014).

We can see a simple domain layout, the computers on that domain and their respective vulnerabilities. Figure 4 is then inserted into Neo4j and is shown in Noel et al. (2014) Figure 5 below:

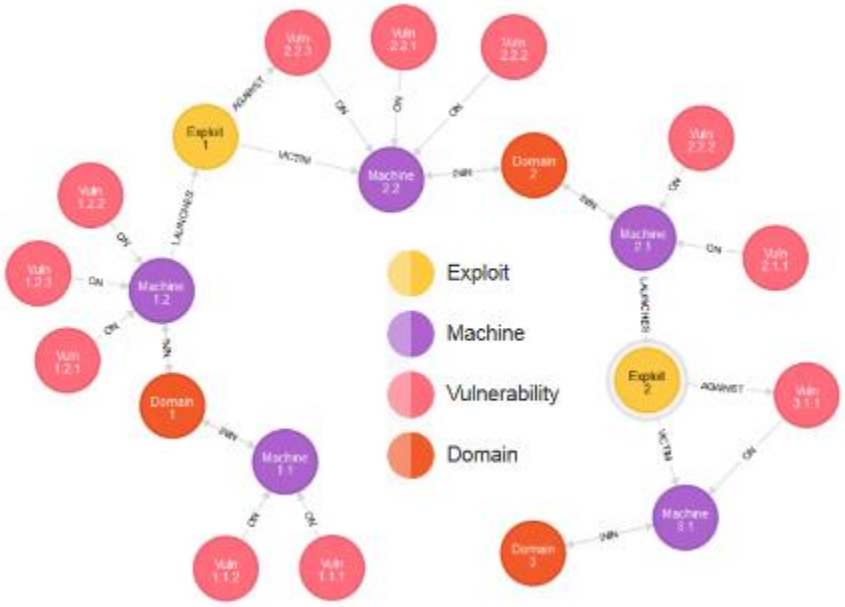


Figure 5. Post transformation to graph database format (Noel et al., 2014)

From this database, it is possible to run queries based on relationships within the graph. The following query is used by Noel et al. (2014) to find “all paths of exploitable vulnerabilities between a particular pair of machines: “

```
MATCH path=
    (start:Machine{name:'Machine 1.1'})
    -[r:LAUNCHES|VICTIM|IN*]→
    (end:Machine{name:'Machine 3.1'})
RETURN path
```

Resulting in the following Figure 6 depicting any existing paths between machine 1.1 and machine 3.1:

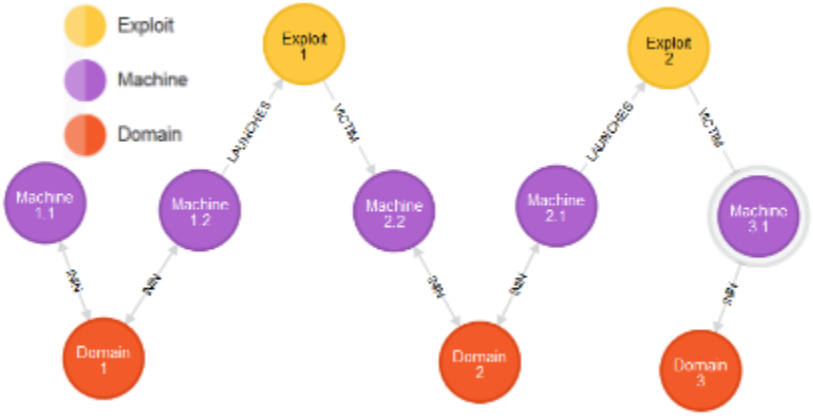


Figure 6. Subgraph generated via query of example cross domain exploit chain (Noel et al., 2014).

From this generated graph we can see how it is possible for machine 1.1 to reach machine 3.1 through use of various exploits. Below is a walk-through of the various exploits used to eventually reach machine 3.1:

1.1 → 1.2 (same domain can exploit without issue)

1.2 → 2.2 (exploit nodes are holes in the firewall allowing cross-domain exploitation)

2.2 → 2.1 (same domain exploit)

2.1 → 3.1 (exploit to domain 3 across a hole in firewall).

With the basics of the framework provided by Noel et al. (2014) outlined it can be used as the basis for a methodology to discover the blast radius of a zero-day vulnerability given enough information about the infection vector.

Data visualization can be seen above in various forms in Figures four, five, and six. These visualizations are excellent options for their particular data sets small size but will soon fail when scaled up in size due to the lack of data condensation. In domains with hundreds or thousands of machines, it's possible for many of those machines to be

infected during a zero-day disclosure and as such the visualizations in Figure four, five, and six quickly begin to fail as large sets of data appear and overwhelm the visualization capabilities of those visualizations. This necessitates different kinds of graph visualizations for these scenarios as well as provides a good place for introducing different concepts and categories that pertain to data visualization. Some of the main benefits of data visualization are covered in *Data Visualizations (*2011) written by Noah Iliinsky and Julie Steele. In this book they introduce some of the core benefits of data visualization below in the following bulleted list:

- Visualization leverages the incredible capabilities and bandwidth of the visual system to move a huge amount of information into the brain very quickly.
- Visualization takes advantage of our brains' built-in "software" to identify patterns and communicate relationships and meaning.
- Visualization can inspire new questions and further exploration.
- Visualization helps to identify sub-problems.
- Visualization is excellent for identifying trends and outliers, discovering or searching for interesting or specific data points in a larger field, etc. (Iliinsky & Steele, 2011).

Each point within the above list highlights and applies to the CSIR problem in a fairly straight forward and intuitive way. The first bullet shows how the size problem that can occur in larger organizations can be addressed quite well by a proper visualization of CSIR data to help quickly arm CSIR first responders with data pertaining to a newly disclosed zero-day vulnerability. The second bullet highlights how the brain can often

digest and interpret a visualization better and more quickly than it can raw data that would be provided by an SQL query. The third bullet hints to uses for visualizations that might lead to questions assisting in the prevention and remediation of a potentially similar zero-day. The fourth bullet is quite relevant to CSIR as many problems have sub-problems that are hard to identify and using visualizations to potentially identify those sub-problems is quite the boon. The final bullet point is perfect for CSIR incidents as often the key requirements are identifying trends and searching for specific data points in a larger field.

With the basics of data visualization introduced as well as the benefits of various options available for integration with Neo4j can then be shown. The list of options from the Neo4j website in their graph visualization tools section is as follows:

Directly embeddable libraries.

- Neovis.js
- Popoto.js

Embeddable libraries without direct connection.

- D3.js
- Vis.js
- Sigma.js
- Vivagraph.js
- Cytoscape.js

Standalone tools.

- GraphXR

- yFiles
- Linkurious
- Graphistry
- Perspectives
- Keylines

(Neo4j, 2018)

Each of these tools has advantages and disadvantages that will be briefly introduced and covered.

Directly embeddable libraries.

Neovis.js. The brief description from Neo4j is as follows,

> “Customizing and coloring styles based on labels, properties, nodes, and relationships is defined in a single configuration object. Neovis.js can be used without writing Cypher and with minimal JavaScript for integrating into your project.” (Neo4j, 2018)

Neovis integrates with Neo4j and creates a customized and colored graph using a configuration object that takes data defined on those nodes and their relationships to create the corresponding visualization. The example visualization on the Neo4j site using Neovis.js can be seen below:

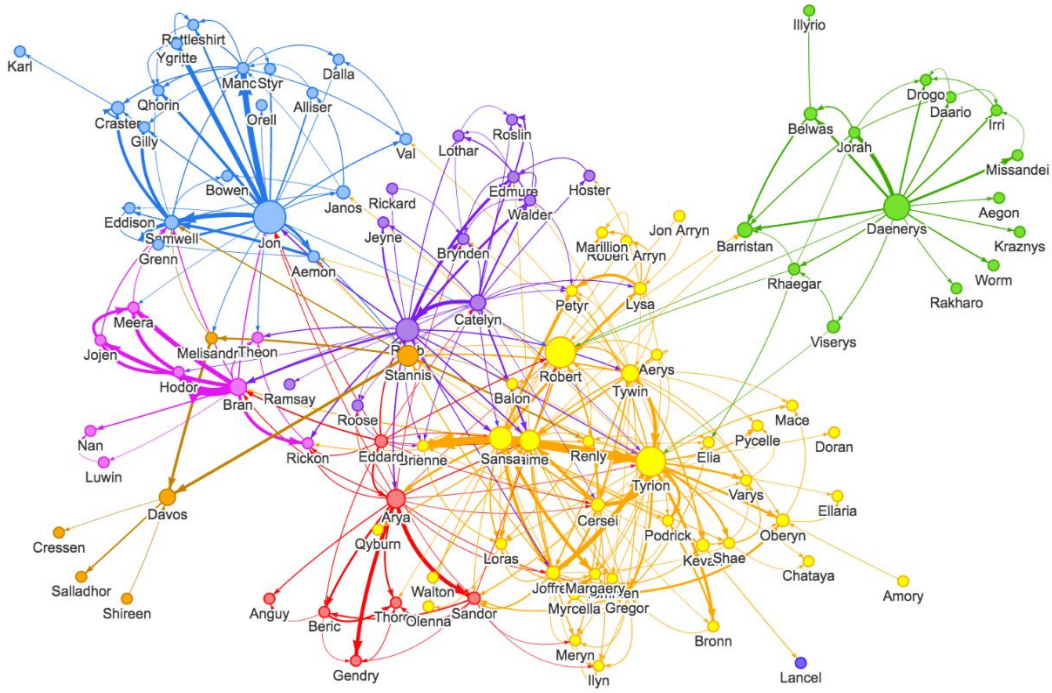


Figure 7. Example visualization method based on game of thrones relationships (Lyon, 2016; Neo4J, 2018).

The above figure helps provide a better understanding between the various related nodes, the size of their relationships, and the domains they belong to as identified by the walktrap method. In this case, the example is game of thrones characters and their relationships. The following quote explicitly defines the various properties and their meaning within the visualization:

> “The graph of thrones. Node size is proportional to betweenness centrality, colors indicate the cluster of the node as determined by the walktrap method, and the edge thickness is proportional to the number of interactions between two characters.” (Lyon, 2016)

Applying the above graph representation to a network and computer domain would result in a similar data visualization to the game of thrones visualization above, though it may suffer from the vast number of devices present on a large network if not condensed before representation using Neovis.js.

Popoto.js. This library is based on the D3.js library and serves as a tool for assisting in query generation within Neo4j. As far as visualizations are concerned it does not provide anything revolutionary but does serve as a great tool for introducing users to the Cypher query language. Acting as a simple visual query builder an example can be seen below from the Popoto.js website:

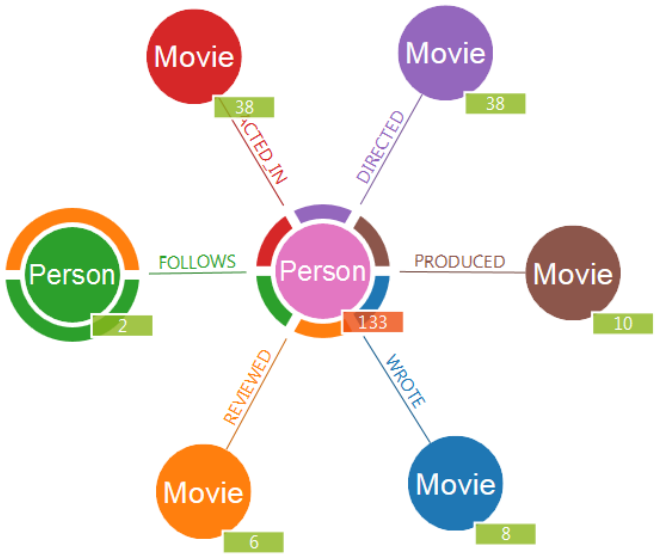


Figure 8. Example visual query builder for graph data (Popoto.js, 2018)

D3.js. The brief introduction from Neo4j introduces D3.js as the following:

> “As the first line on D3’s website states ‘D3.js is a JavaScript library for manipulating documents based on data.’ You can bind different kinds of data to a DOM and then execute different kinds of functions on it. One of those functions

includes generating an SVG, canvas, or HTML visualization from the data in the DOM.” (Neo4j, 2018)

Some of the examples on the Neo4j site leverage D3.js as a tool for producing their visualizations. Since D3.js is not Neo4j integrated it requires the data set to be exported in a format compatible with the D3.js visualization being used. Most often this is done by exporting the nodes and their relationships into either JSON objects or the object format required of the visualization being used. D3.js has a huge plethora of options when it comes to potential visualizations and sifting through them for visualizations that match a particular use case is required. A few different visualizations will be introduced along with their use cases. The first example from the D3.js website is a packed circle graph which can be seen below:

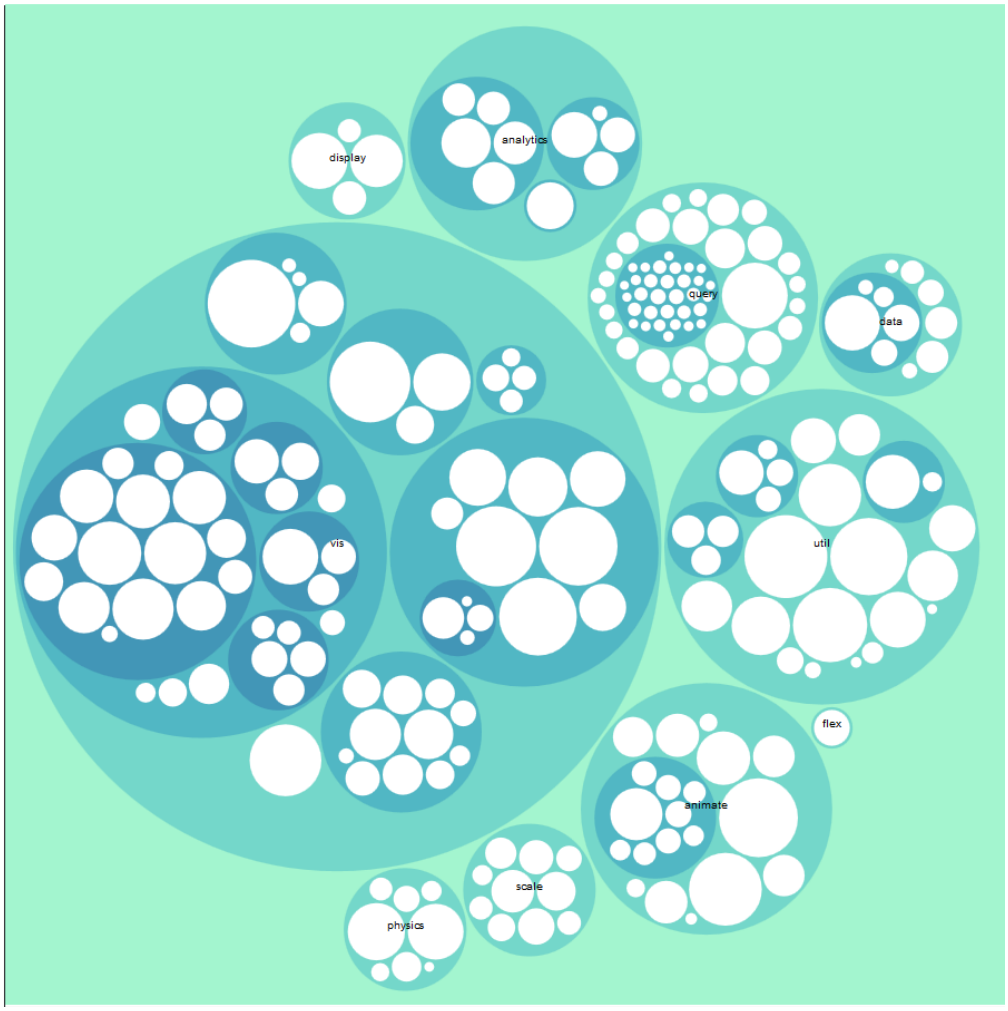


Figure 9. Example packed circle graph for graph data visualization (Bostock, 2019).

The above visualization is an excellent way to condense data into smaller subsections that can be traversed interactively in a web browser. It addresses one of the main problems with the other presented visualizations by containing the data set to an easy to digest set of subsections. The downside is each of these subsections needs to be designed and created to match specific use cases that users of the visualization may desire. As such if a packed circle graph was to be designed to contain a list of

CVE’s and their subsequent operating systems and domains it would need to be outlined and designed to meet the requirements of the users.

The next visualization is the radial tidy tree and can be seen below:

Figure 10. Example radial tidy tree for graph data visualization (Sontrop, 2018).

The radial tree is a bit more explicit in its labeling and helps paint a better picture of what object parents have what object children. As such it is a great way to visualize a

simple understandable network hierarchy to show what is owned under a specific domain. Again, it would need to be configured to meet the user's requirements.

Up next is a collapsible force layout, this is a layout similar to the one built into Neo4j but has the benefit of being more modifiable than the one present within Neo4j. It can be seen in Figure 11 below:

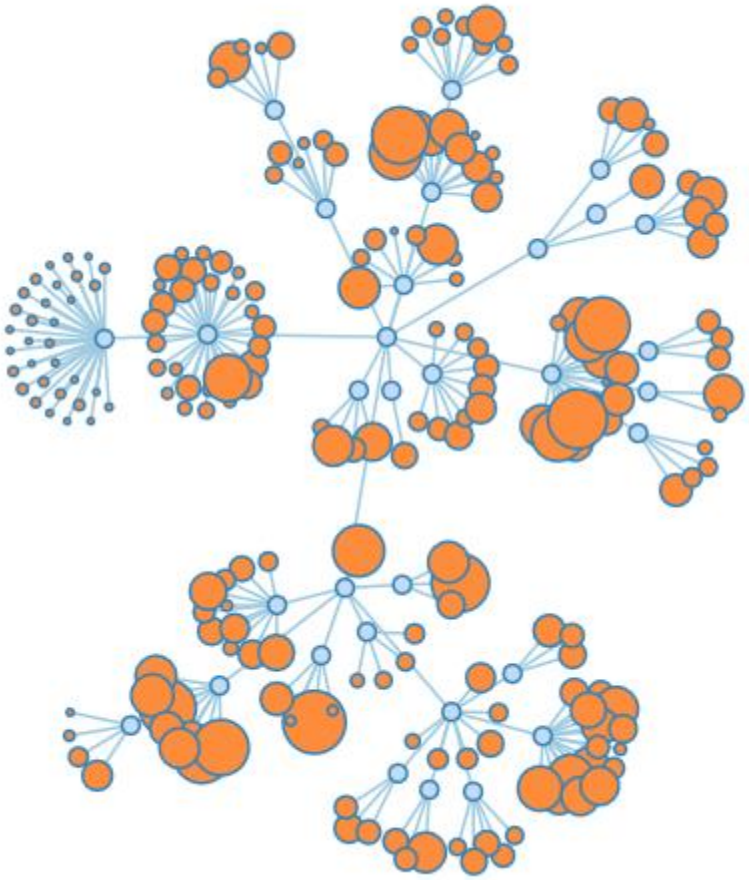


Figure 11. Example collapsible force layout for graph data visualization (Bostock, 2019).

The advantages of the D3.js graphs is their being modifiable by a developer to better fit an end users' needs along with the ease of putting them into a web browser for ease of access.

Vis.js. This visualization framework offers customization options for a multitude of different visualization style sets. These stylizations are fairly straightforward and do well for showing simple data sets visually. The following are examples from the framework:

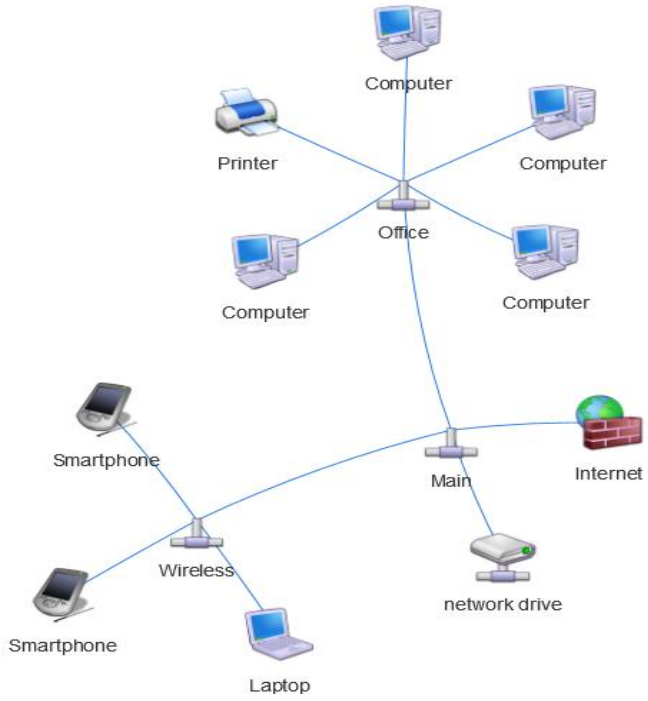


Figure 12. Simple Vis.js graph relationship visualization option (Visjs, 2019).

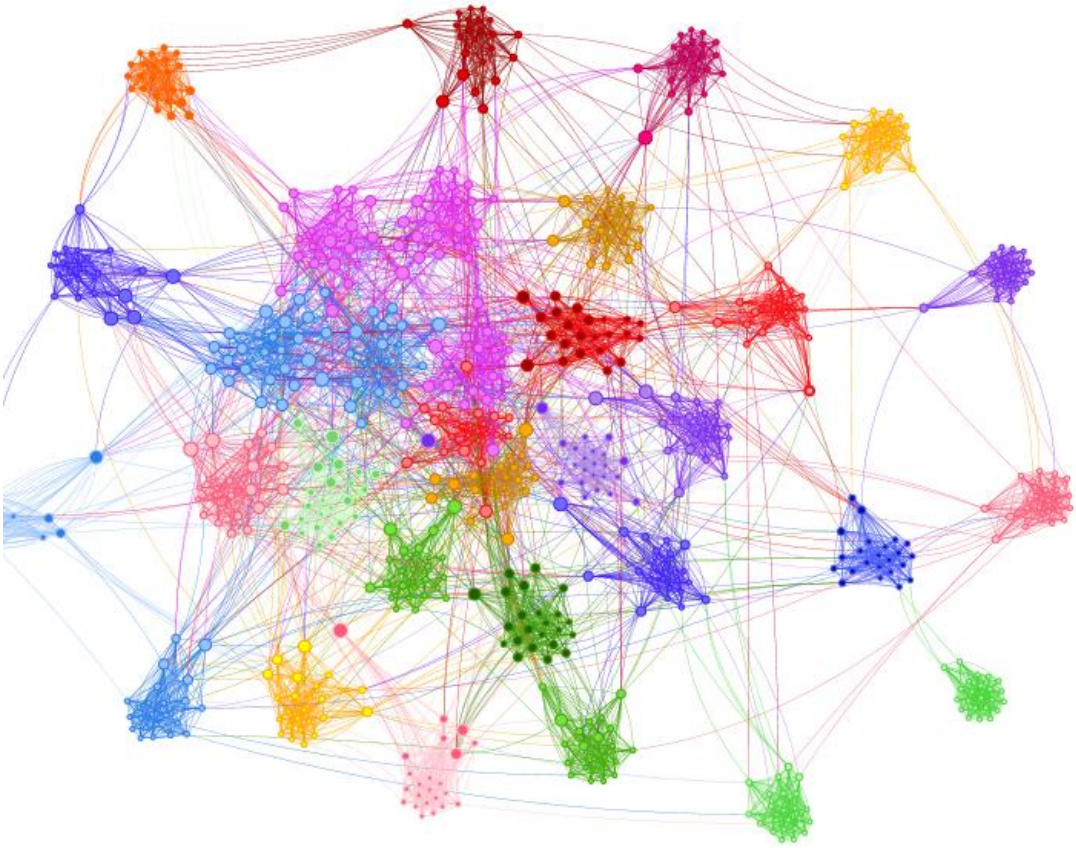


Figure 13. Complex Vis.js graph relationship visualization option (Visjs, 2019).

The above figures from the Vis.js website offer some simple as well as in-depth solutions for visualization that differ slightly from the other options mentioned previously. The first example in Figure 12 introduces an easy to understand framework that uses pictures instead of simple circles for nodes. For computers and network graphs, this provides a simple picture of the network that is easy to understand and interpret. A large amount of complexity is lost due to the nature of the graph but the benefits gained are the ease of digesting the graph at a glance for experts as well as lay persons. Figure 13 works similarly to the one presented in Figure 7 but has more options for nodes and edge customization. Such as the curvature present on the edges in the visualization as

well as the node color and size. Vis.js has a multitude of other visualization options available that can be perused to find one that fits well for a particular use case.

Sigma.js. The basic description from Neo4j is as follows: “Sigma is a highly-extensible library meant for modification to meet the requirements necessary of it. It takes JSON and GEXF formats as inputs and is highly modifiable to meet varying requirements.“ (Neo4j, 2018) The example from the Sigma website does an excellent job showcasing its ability to be modified as it shows a walk through with more and more extensibility added at each step. Starting with the basic graph import, a hover text feature is added to show the node name on hovering the node. Next an on click function that trims the node to highlight only its relevant relations is added. The two can be seen in the screenshots from the Sigma site below:

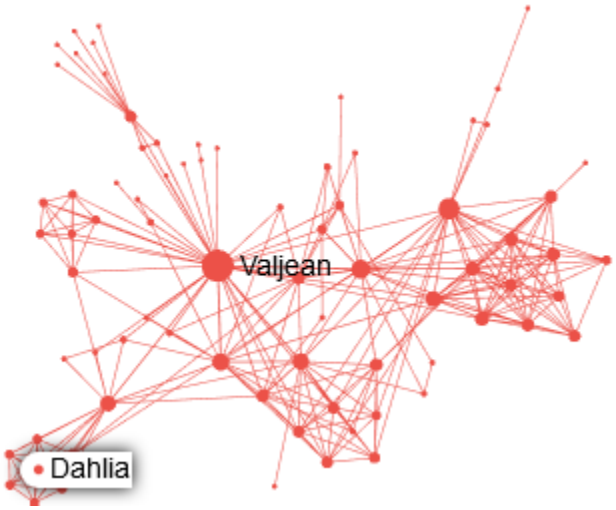


Figure 14. Full Sigma.js library example (Jacomy, 2019).

Figure 15. Drill down enhancement for Sigma.js library example (Jacomy, 2019).

In Figure 14 the on hover functionality can be seen and in Figure 15 the on click functionality to limit nodes to only directly related ones can be seen. Both of these functionalities can be useful from a data exploration perspective to increase the effectiveness of the visualization and assist in helping specialists asks the right questions during a zero-day vulnerability announcement and remediation.

Vivagraph.js. Offers a few interesting customizations options built into it that are similar to the other offerings but has something new to offer as well. Similar to Vis.js, Vivagraph can use alternate images to represent nodes instead of just simple circles; in the case of Vivagraph, it can use custom images with little configuration which is a great feature to have. Along with that feature, Vivagraph has one more feature that sets it apart from the others and that is its layout customization options. For example, in the below screenshots from their GitHub that feature can be seen in action:

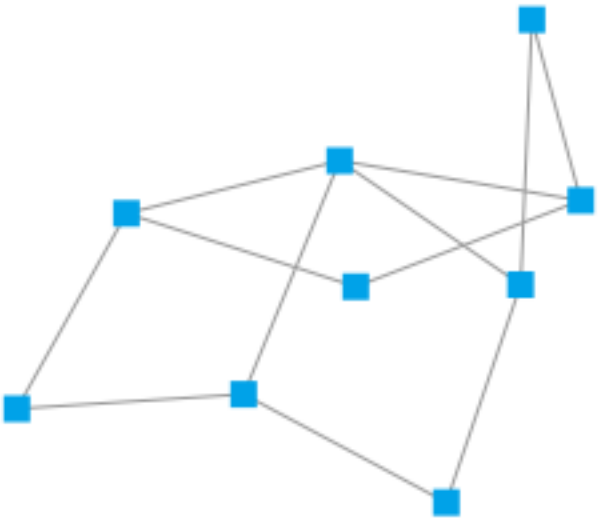


Figure 16. No layout applied Vivagraph example (Anvaka, 2019).

Figure 17. Layout applied Vivagraph example (Anvaka, 2019).

The two node sets in Figure 16 and 17 above are the same set of nodes just without a layout applied to them. Vivagraph gives the ability to apply and customize a layout to the set of nodes and create a cleaned up layout like the one in Figure fifteen. This is particularly helpful in CSIR due to the nature of computers generally being in groups and clusters. Sets of servers, sets of user computers, sets of mobile devices and so forth. Being able to customize the organization of those node sets allows for the resulting visualization to be more understandable at a glance.

Cytoscape.js. This visualization framework offers another differentiating feature that separates it from the others in having touch screen compatibility. Outside of

touchscreen capability, it has similar offerings to the previous frameworks. As such its main usage niche is if particular requirements pertaining to touch screen compatibility are required.

Standalone product tools. Along with the frameworks presented above, there are standalone products that can be purchased that provide visualization options and custom implementations. The brief introduction for these tools is as follows:

> “Certain tools and products are designed as standalone applications that can connect to Neo4j and interact with the stored data without involving any code. These applications are built with non-developers in mind–for business analysts, data scientists, managers, and other users to interact with Neo4j in a node-graph format.“ (Neo4J, 2018)

GraphXR. Touted as a “start-to-finish web-based visualization platform for interactive analytics” (Neo4J, 2018). GraphXR offers data collection and presentation through built-in Neo4J Desktop connections. Their range of applications includes law enforcement, medical research, and knowledge management. An example of one implementation of the GraphXR suite can be seen below:

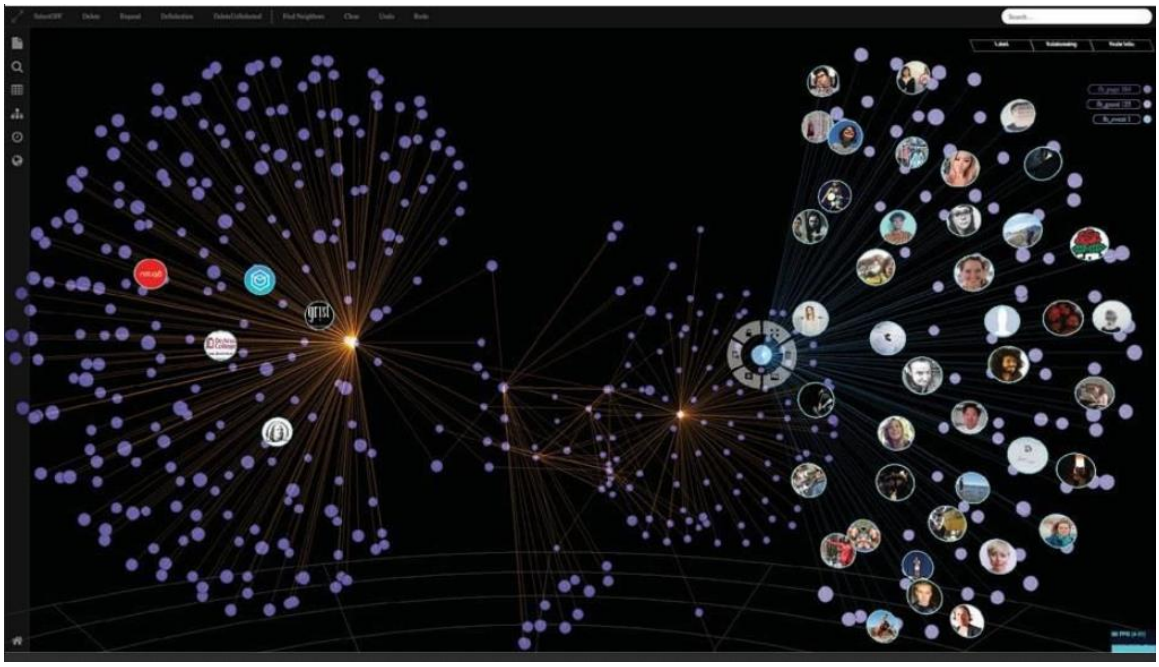


Figure 18. Example GraphXR implementation *(*Neo4J, 2018).

yFiles. This suite offers a wide range of potential visualization options and integration with Neo4J Desktop. The yFiles website site introduces yFiles with the following:

> “Turn your data into clear diagrams with the help of unequaled automatic diagram layouts, use rich visualizations for your diagram elements, and give your users an intuitive interface for smooth interaction. With yFiles diagramming components you will get this out-of-the-box for your applications. On nearly any platform or technology.“ (yFiles, 2019)

Their suite of visualizations is amalgamated in the below screenshot:

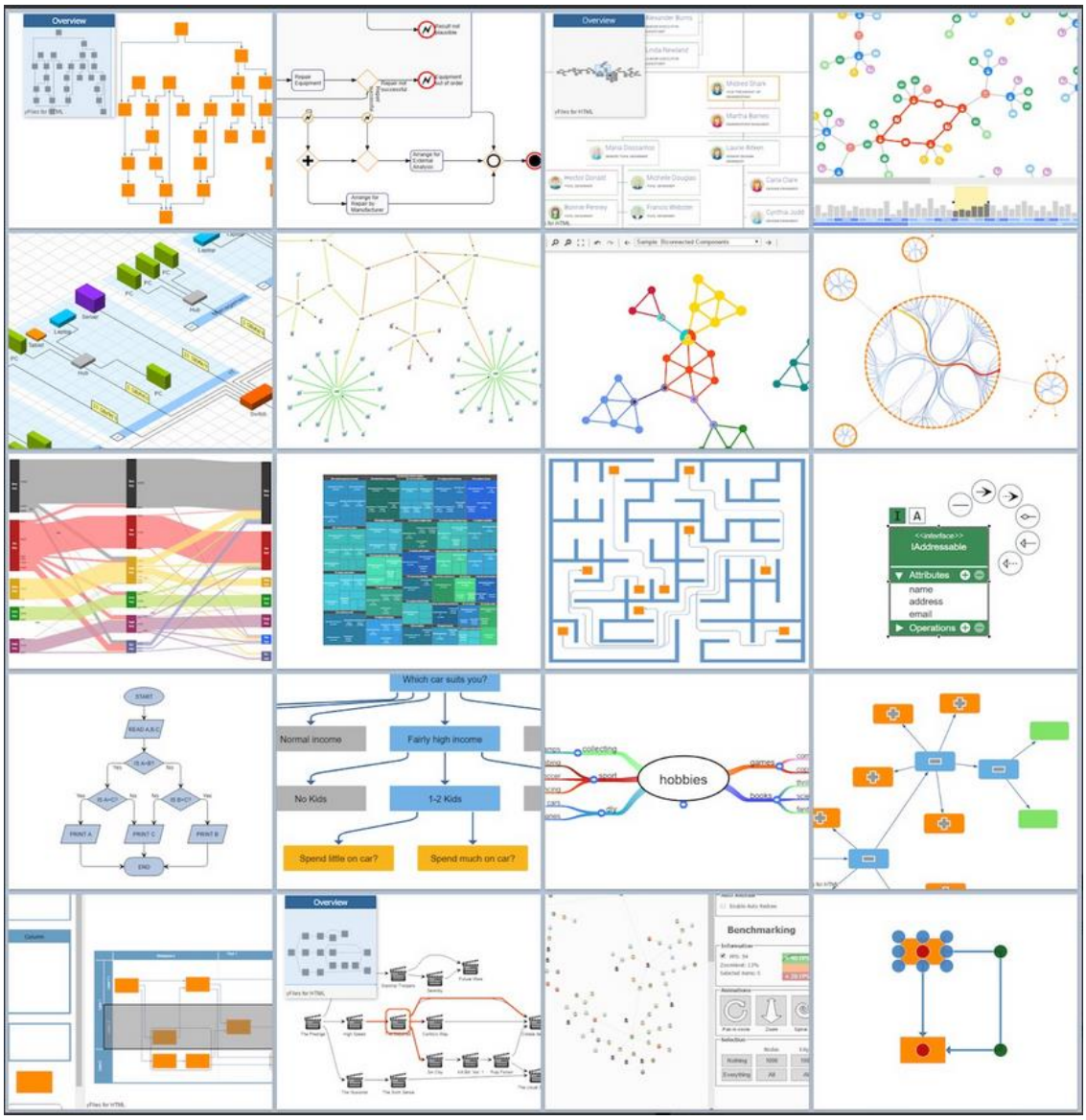


Figure 19. yFiles Visalization options aggregation (Neo4J, 2018).

Linkurious Enterprise. Designed for use in assisting analysts with detecting and analyzing threats in large pools of connected data. Linkurious is used to fight “financial crime, terror networks or cyber threats.” (Neo4J, 2018) In an example from their site on detecting credit card fraud with Neo4J some sample visualization can be seen below:

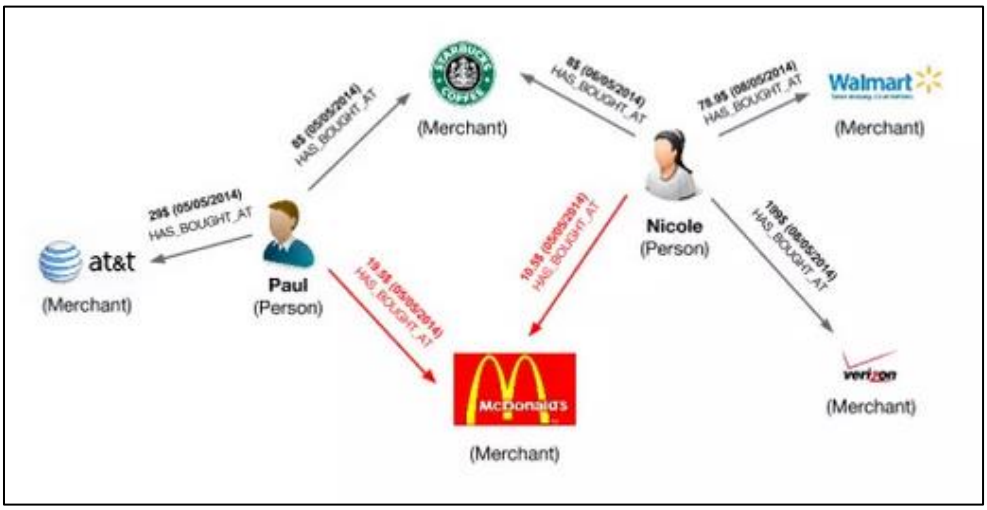


Figure 20. Small-scale Linkurio graph model for detecting credit card fraud (Linkurio, 2018).

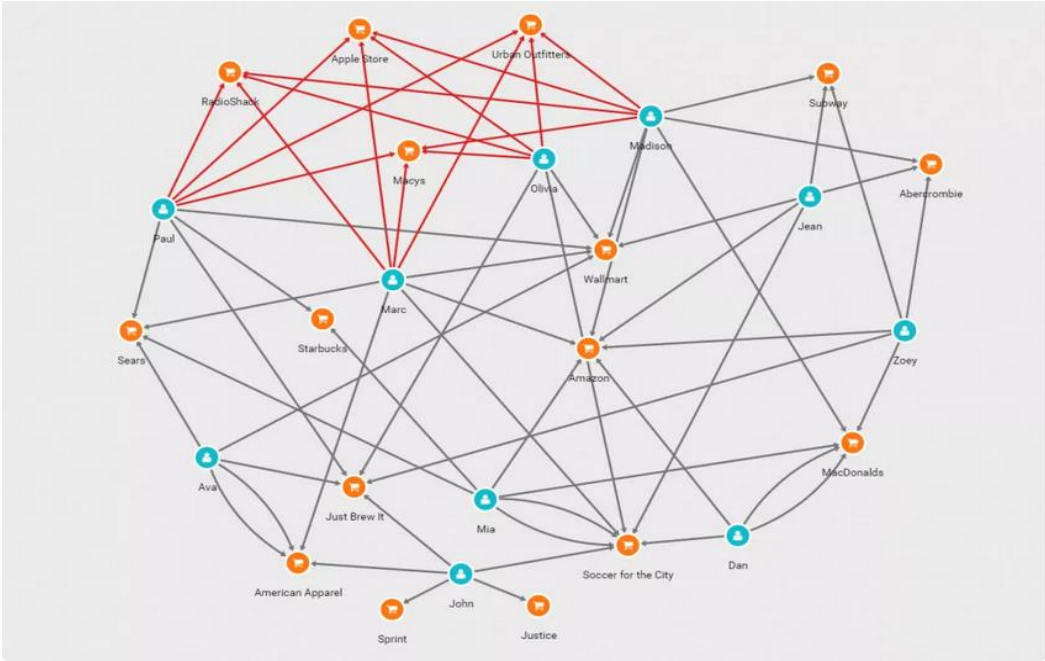


Figure 21. Medium-scale Linkurio graph model for detecting credit card fraud (Linkurio, 2018).

Graphistry. This suite abstracts queries and wrangling with data away from the user and automatically handles transforming the data into visual investigation maps built for the needs of analysts. An example of visualization can be seen below:

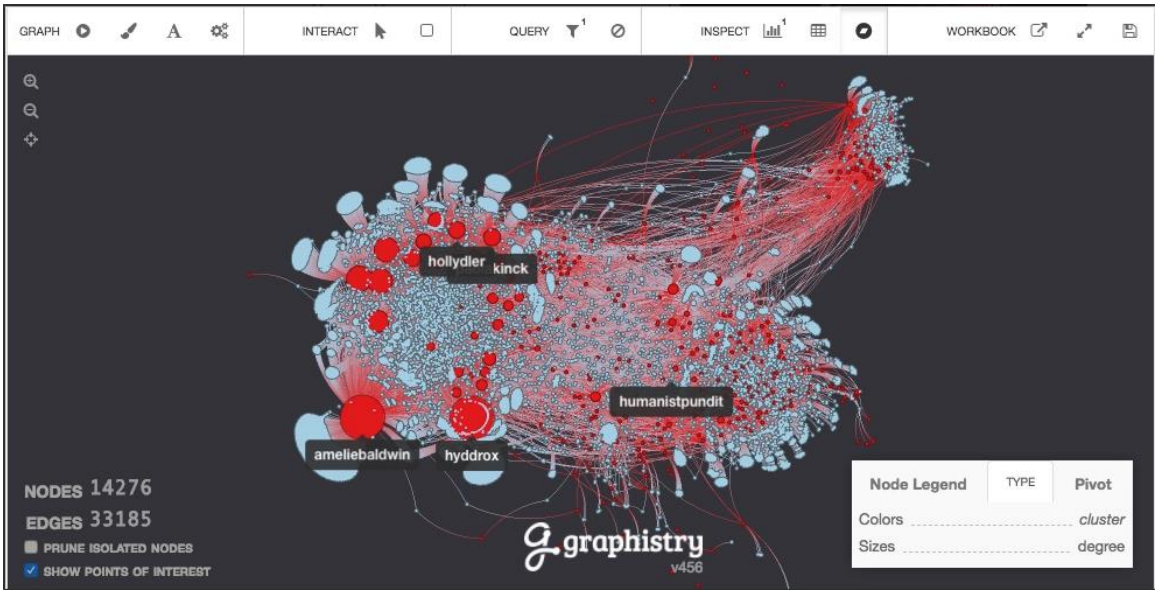


Figure 22. Visual investigation map using graphistry *(Neo4J, 2018)*

Perspectives. This solution offers a wide range of options for data visualization with a graph database browser option, a network topology option, and more. Below the network topology visualization is shown:

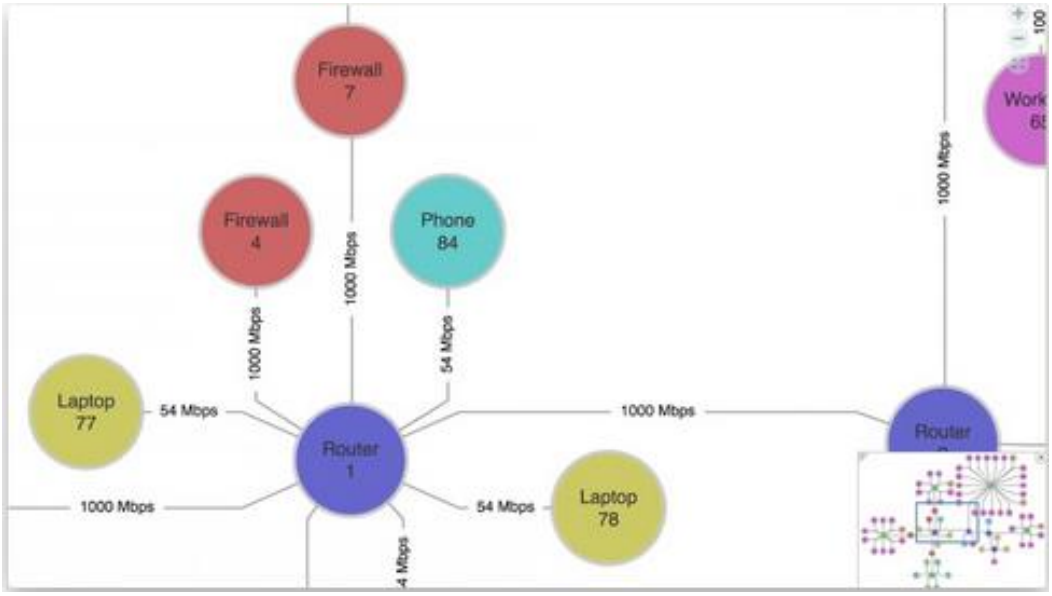


Figure 23. Tom Sawyer perspectives network visualization option (Tom Sawyer, 2019).

Keylines. This suite offers a variety of options for visualization much like the other frameworks, the key difference is Keylines touts the ability to work on any device and on any common browser. A Keylines visualization sample can be seen below:

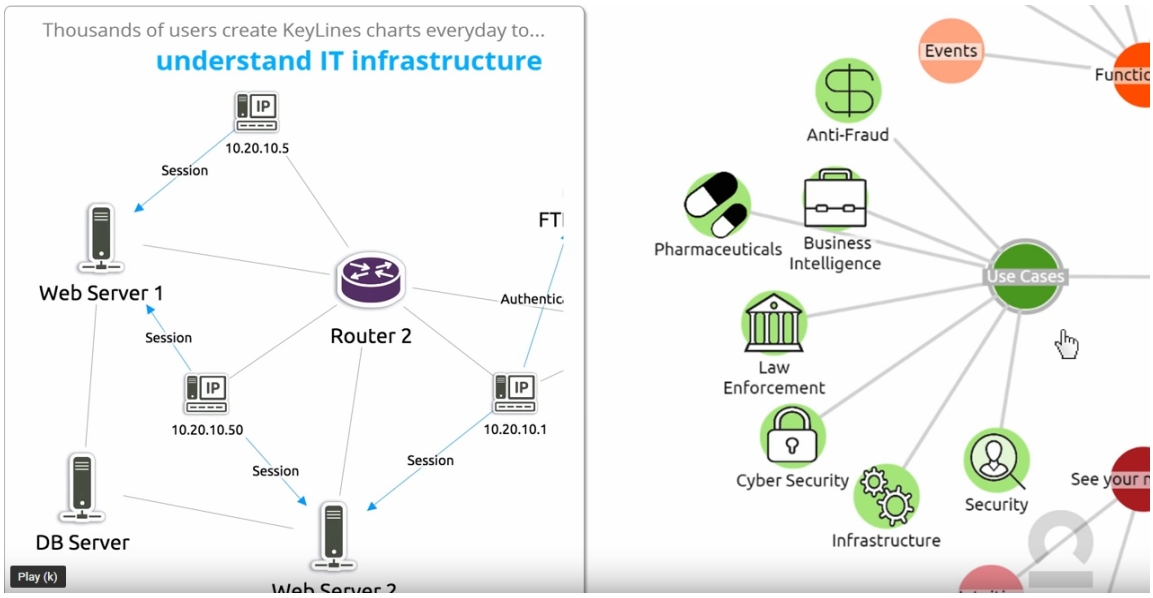


Figure 24. Keylines network and business organization visualzation examples (Keylines, 2019).

With the various options for visualization of Neo4J data covered it is possible to move forward into the literature review.

Literature Related to the Problem

There is a plethora of information relating to graph databases as can be seen in the studies by Angles (2012) and Buerli (2012). These two studies looked at different aspects of graph databases and their differences. Buerli did a more generalized overview of different graph databases and their technical differences. Angles focused more on documenting the feature set, operations, and data structures that were leveraged by each graph database solution. Along with these general overviews, there are also many optimization problems that need to be solved with graph databases, such as querying subgraphs of large sets of data (Hong, Zou, Lian, & Yu, 2015) for the varying design architectures shown in the write-up by Buerli. The subgraph querying process is highlighted well in a paper by Venkatesh (2014). These aforementioned papers help define most of the necessary background as far as graph databases are concerned. Covering the relevant problems as well as introducing concepts that differ from relational databases.

The history of graph databases naturally starts with the history of databases. This history is reviewed in a paper by Berg, Seymour, and Goel where they detail databases throughout the decades starting with 1960s up through the early 21st century. The 1960s mark a period in time where it became more cost effective for companies to increase the sizes of their data stores. The two models used were the network model CODASYL (Conference on Data System Language) and the hierarchical model IMS

(Information Management System). A summary of that first generation from the Berg et al. paper is as follows:

> "The first generation of database systems was navigational. Applications typically accessed data by following pointers from one record to another. Storage details depended on the type of data to be stored. Thus, adding an extra field to database required rewriting the underlying access/modification scheme. Emphasis was on records to be processed, not overall structure of the system. A user would need to know the physical structure of the database in order to query for information. One database that proved to be a commercial success was the SABRE system that was used by IBM to help American Airlines manage its reservations data (Bercich, 2002). This system is still utilized by the major travel services for their reservation systems." (Berg et al., 2012)

The 1970s brought further improvements, changes, and new designs to the database environment. The biggest of those being the creation of the relational database model. The relational model suggested that applications search for content rather than by using links. The key of this model was having the local organization disconnected from the physical information storage leading up to this becoming the standard for database systems. Two major relational databases were created at this time INGRES and System R. Each lead to a multitude of offshoots that are well known today such as MS SQL, POSTGRES, Oracle, DB2, and more. Around this time the Entity-Relationship model was proposed and allowed for designers to focus on applications of data rather than the logical table structure.

During the 1980s commercialization of these systems from the 1970s began to rapidly proliferate and some from the 1960s began to fall out of use. Some such systems that began to fall out of use were network and hierarchical models which are used in some legacy systems but sparsely used outside that. During this period the creation of the object-oriented database system occurred. These databases are designed for integration with various programming languages with the main feature being the support of modeling and creation of data as objects. The main benefits and drawbacks being highlighted in the following from Berg et al.,

> "OODBMS could efficiently manage a large number of different data types. Objects with complex behaviors were easy to handle using inheritance and polymorphism. This also helped in reducing the large number of relations by creating objects. The biggest problem with OODBMS was switching an existing database to OODBMS, as the transition requires an entire change from scratch and it is typically tied to a specific programming language and an API (Application Programming Interface) which reduces the flexibility of the database. To overcome the problems of OODBMS and take full advantage of the relational model and object-oriented model, the Object Relational Database Model was developed in the early 1990s." (Berg et al., 2012)

The 1990s resulted in the death of a fair amount of database providers and resulted in the remaining companies to offer more complex products at higher prices. Most developments focused on client tools for building applications. Toward the end of the 1990s, Extensible Markup Language (XML) was introduced which helped solve

long-standing database problems. During this time period, NoSQL was also introduced by Carlo Strozzi. These NoSQL datastores are described and introduced in the following quote from Berg et al:

> "Often, NoSQL databases are categorized according to the way they store the data and they fall under categories such as key-value stores, BigTable implementations, document store databases, and graph databases. NoSQL database systems rose alongside major internet companies, such as Google, Amazon, Twitter, and Facebook, which had significantly different challenges in dealing with data that the traditional RDBMS solutions could not cope. NoSQL databases are often highly optimized for retrieve and append operations and often offer little functionality beyond record storage. The reduced run time flexibility, compared to full SQL systems, is compensated by significant gains in scalability and performance for certain data models." (Berg et al., 2012)

The above section from Berg et al. introduces NoSQL and as such the basic infrastructure needed for graph databases. NoSQL has many applications outside of graphs, though for the purpose of this literature review the graph database applications of NoSQL and its history will be the focus of the next section.

Graph databases originated from graph theory, graph theory and the basic idea of graphs were introduced by the Swiss mathematician Leonhard Euler in the 18th century to solve the Königsberg bridge problem. The problem is outlined and briefly detailed in a paper by Victor Adamchik (2005) below:

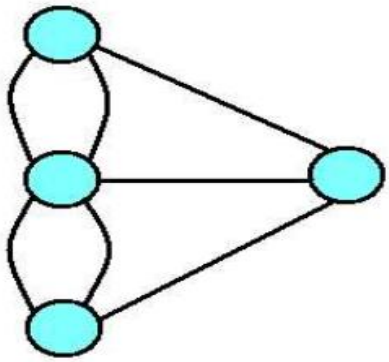


Figure 25. Visual representation of Königsberg bridge problem (Adamchik, 2005).

> "German city of Königsberg (now it is Russian Kaliningrad) was situated on the river Pre-gel. It had a park situated on the banks of the river and two islands. Mainland and islands were joined by seven bridges. A problem was whether it was possible to take a walk through the town in such a way as to cross over every bridge once, and only once." (Adamchik, 2005)

The graph of the problem is below:

Figure 26. Graph representation of Königsberg bridge problem (Adamchik, 2005).

The subsequent graph makes solving the Königsberg bridge problem much easier to manage. There are a few other problems addressed by graph theory such as the traveling salesman problem and the four coloring problem that can be addressed with graphs. Graph theory is the basis for graph databases and using them to solve problems that occur within graphs. Graph databases themselves are based on Graph theory and NoSQL to solve problems present in classical database schema which often fail to scale horizontally and instead can only scale vertically. Silvan Weber expands on some of the various NoSQL databases defined in Berg et al. giving examples for each. The brief description of graphing databases is as follows:

> “Nowadays, for example, graph databases can be used in location-based services (LBS), to find common friends on social networks or to establish the shortest paths through the daily traffic (with standard algorithms like Dijkstra) or, in a more general manner, for the efficient querying of data in a network.“ (Weber)

With graph theory and graph databases introduced and the background given in the previous section, a road map can be seen for how the current marketplace of graph databases has come to be.

Zero-day vulnerabilities are covered in a few pieces of literature with the best coverage by Bilge and Dumitras (2012). Their paper gives a great summary of the zero-day problem with the following quote:

> “A zero-day attack is a cyber attack exploiting a vulnerability that has not been disclosed publicly. There is almost no defense against a zero-day attack: while

the vulnerability remains unknown, the software affected cannot be patched and anti-virus products cannot detect the attack through signature-based scanning."

When a zero-day is discovered and publicly disclosed there is often a huge spike in malware and other attacks that leverage that zero-day vulnerability as can be seen from Bilge's et al. (2012) work and the following results in respect to malware and other exploits leveraging publicly disclosed zero-day vulnerabilities: "183–85,000 more variants are detected each day" after public disclosure. The massive uptick in the usage of the exploit creates a situation where rapid infection of unpatched systems can occur as can be seen in the Petya (S. 2017), WannaCry (2017), Meltdown (Lipp, Schwarz, Gross, Prescher, Haas, Fogh, & Hamburg, 2019) and Spectre (Lipp et al., 2019) vulnerabilities. In many cases, due to the nature of zero-day vulnerabilities, it is also necessary to patch existing systems that were already infected with a zero-day vulnerability that had not yet been exposed. Once a now disclosed zero-day is discovered within the network, determining the blast radius is of the utmost importance to the removal of the exploit and re-securing the system.

Literature related directly to the problem specifically is rather sparse due to expertise in both security incident response being needed along with an understanding of graph databases. The one paper that directly addresses parts of those problems in depth is the paper by Noel et al. (2014). In this paper, they create a framework for using graph databases to discover obvious and less obvious security vulnerabilities that might be possible within a system for the purposes of threat and vulnerability management and remediation. In the case of the paper, Noel et al. (2014) use their framework as a

detection and prevention platform for finding and remediating potentially missed exploit chains.

Summary

This section introduced graph databases, why Neo4j was a good choice and used in Noel et al. (2014), introduced the framework set up by Noel et al. (2014), and set up the groundwork for the methodology to be used to address finding the blast radius of newly disclosed zero-days leveraging the framework presented in Noel et al. (2014). Background information was given on a specific sub-problem relating to data visualization in larger data sets as well as potential frameworks that can be used to address visualization problems that may present themselves within those data sets.

Chapter 3: Methodology

Introduction

In this chapter, there will be an exposition into the design of an environment to test finding the blast radius of a newly released zero-day vulnerability using the framework presented by Noel et al. (2014). Three tests with various numbers of vulnerable servers will be performed. As the network grows so will the network requirements and structures needed to support it. Though the architecture grows and exceeds a depth of four in certain situations it will not come close to the size of a massive enterprise network. As such there are considerations that must be made with visualization and the data set in general moving forward.

Design of the Study

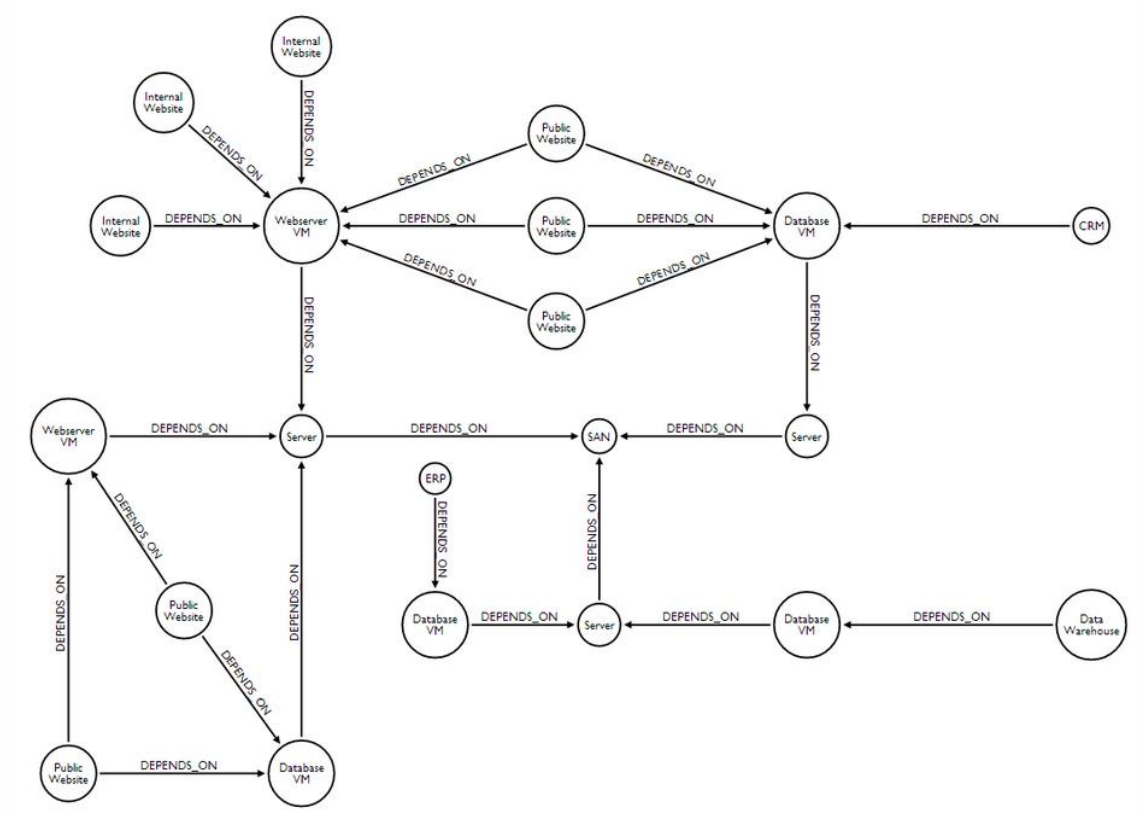
A qualitative approach will be used during this study as the goal is to unearth a methodology that can be used to dynamically enumerate the blast radius of a zero-day attack. The difficulty becomes apparent due to the nature of zero-day attacks leveraging potentially unknown attack vectors. If the graph database used for detecting the attack has no way to create a subgraph of the potentially affected machines due to a lack of data it will be impossible to create a depiction of the blast radius for that particular zero-day vulnerability. The larger the pool of ancillary data the easier it is to create an accurate subgraph and subsequent map of the blast radius for a given zero-day. The key here being that enough ancillary data must be included to detect the majority of zero-day attacks so that it is possible to create subgraphs and subsequently a blast radius of the zero-day event.

The nature of zero-day vulnerabilities, however, do not lend themselves to a situation where they are completely detectable as there will be cases where a database will be missing information necessary to create a subgraph that can be fitted to find the blast radius of a zero-day. Due to this taking stock of data that is needed to detect a specific zero-day can help in the discovery of new zero-days that leverage different but similar vulnerabilities. As such looking at the infection vectors of specific zero-days is a good start for creating a methodology for detecting newly discovered zero-day vulnerabilities.

The set of tests is designed to be done over an iterative scale for a general proof of concept. On a small-scale up through a large-scale where more complex networking architecture and hosts are added as the tests progress. The first set of tests leverages a small-scale data center and works as a proof of concept on a small-scale. The second set of tests adds a few more variables to better simulate a real-world network and zero-day scenario. The third test adds more clones of the data center to act as a representation of multiple data centers that a larger organization would have and tests detection across multiple differing disconnected domains.

Data Collection–Small-scale

The first test requires working with small data set to create subgraphs for a specific zero-day. In this case, a sample small data center graph is built in Neo4j using one of the GraphGist modules created by Bastani (2019). In that article the following hand created architecture diagram can be seen:



F*igure 27.* GraphGist small data center diagram (Bastani, 2019).

After being imported into Neo4j using the queries provided in the GraphGist the following graph which mirrors the above diagram becomes apparent:

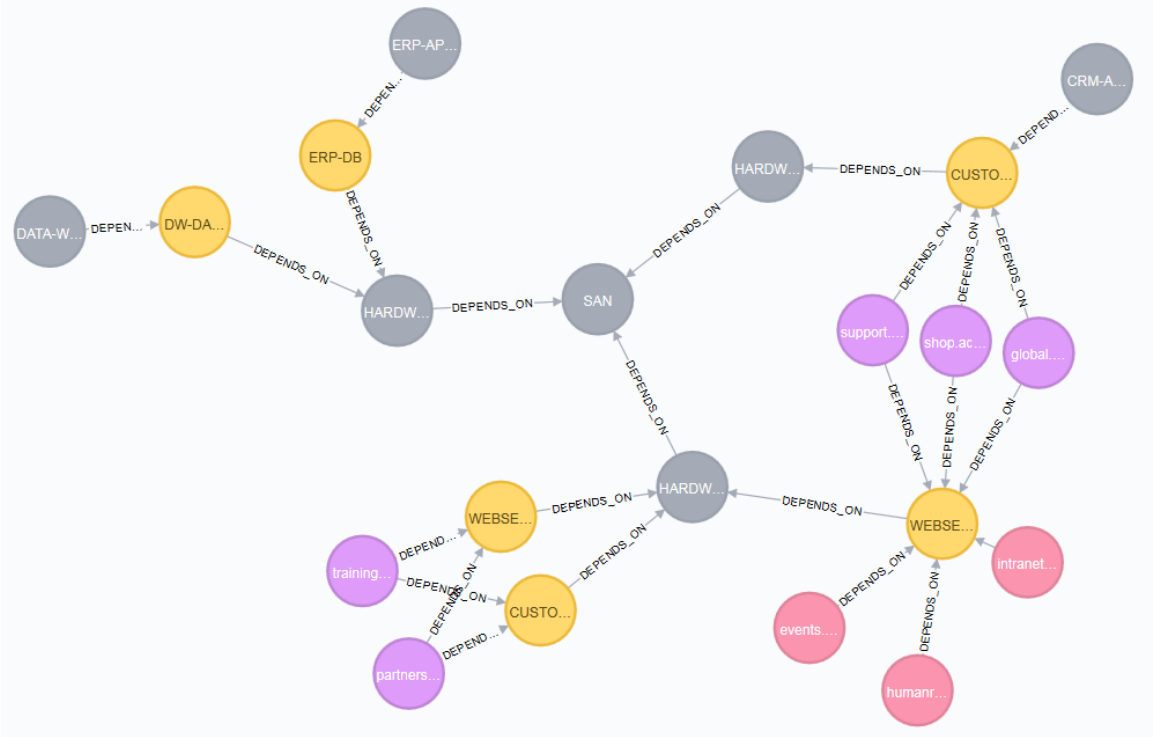


Figure 28. Diagram represented within graphing database Neo4J

Using this small-scale data center, it is possible to see the various hardware and software and its individual dependencies. The next step in this small-scale test is to introduce a zero-day vulnerability node and exploit node similar to the framework provided by Noel et al. (2014). In this initial test, the goal is to show it is possible to detect a released or discovered zero-day given the data to do so exists within our graph. The test zero-day for this case is the Apache Struts vulnerability present in Apache Struts versions 2.2.0 and lower. The two web servers present in the small-scale data center have been modified to have the following attribute to represent having their versioning properly documented within the Neo4j database.

```
// Create Webserver VM 1
CREATE (webservervm1:VirtualMachine {
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
        version: "Apache Struts 2.2.0" })
```

In bold above the node is tagged with a version documenting it leveraging Apache Struts 2.2.0 and subsequently being vulnerable to the Apache Struts vulnerability being tested for. Next, the vulnerability test case is added:

```
CREATE (vuln1:Vulnerability {
        name: "Vuln 1.1",
        CVE_number: "CVE-2010-1870",
        description: "Apache Struts Remote Command Execution",
        port_requirements: "80",
        software_requirements:"Apache Struts <2.2.0"
})
```

The above code snippet documents a basic vulnerability node identical in concept to the ones presented in Noel et al. With that node created and added to the graph it is possible to move onto the next section. Next, it is necessary to make a subgraph of the entire graph leveraging the struts vulnerability defined above to the nodes vulnerable to them:

MATCH (n), (m) WHERE n.CVE_number="CVE-2010-1870" AND

m.version="Apache Struts 2.2.0" CREATE (n)-[:ON]→(m)

Using the statement above variables *n* and *m* are gathered into a list of nodes that correspond to their particular information set within the query. Ergo in the case of n, nodes that correspond to the tag CVE_number and match the value “CVE-2010-1870” will be gathered and stored within n. The same will occur for *m* with version and “Apache Struts 2.2.0”. From that point, a relationship is added between the gather *n* and *m* nodes in this case defined as “ON”. Thus, the written-out query tells us to add a relationship to show the vulnerability node exists on web server nodes matching the requirements defined in the query. From there an exploit node is added based on the newly created zero-day:

CREATE (exploit1:Exploit {

name: "Exploit 1",

description: "Apache Struts Remote Command Execution",

software_requirements:"Apache Struts <2.2.0" })

With this now created the public facing websites that can be used to exploit the Apache Struts vulnerability can be connected to the exploit node, and subsequently the exploit node can be connected to the vulnerable web server.

MATCH (n), (m) WHERE n.system="INTERNET" AND m.name="Exploit 1"

CREATE (n)-[:EXPLOITS]->(m)

MATCH (n), (m) WHERE n.name="Exploit 1" AND m.version="Apache

Struts 2.2.0" CREATE (n)-[:VICTIM]→(m)

Below is the current graph with its new exploit connections:

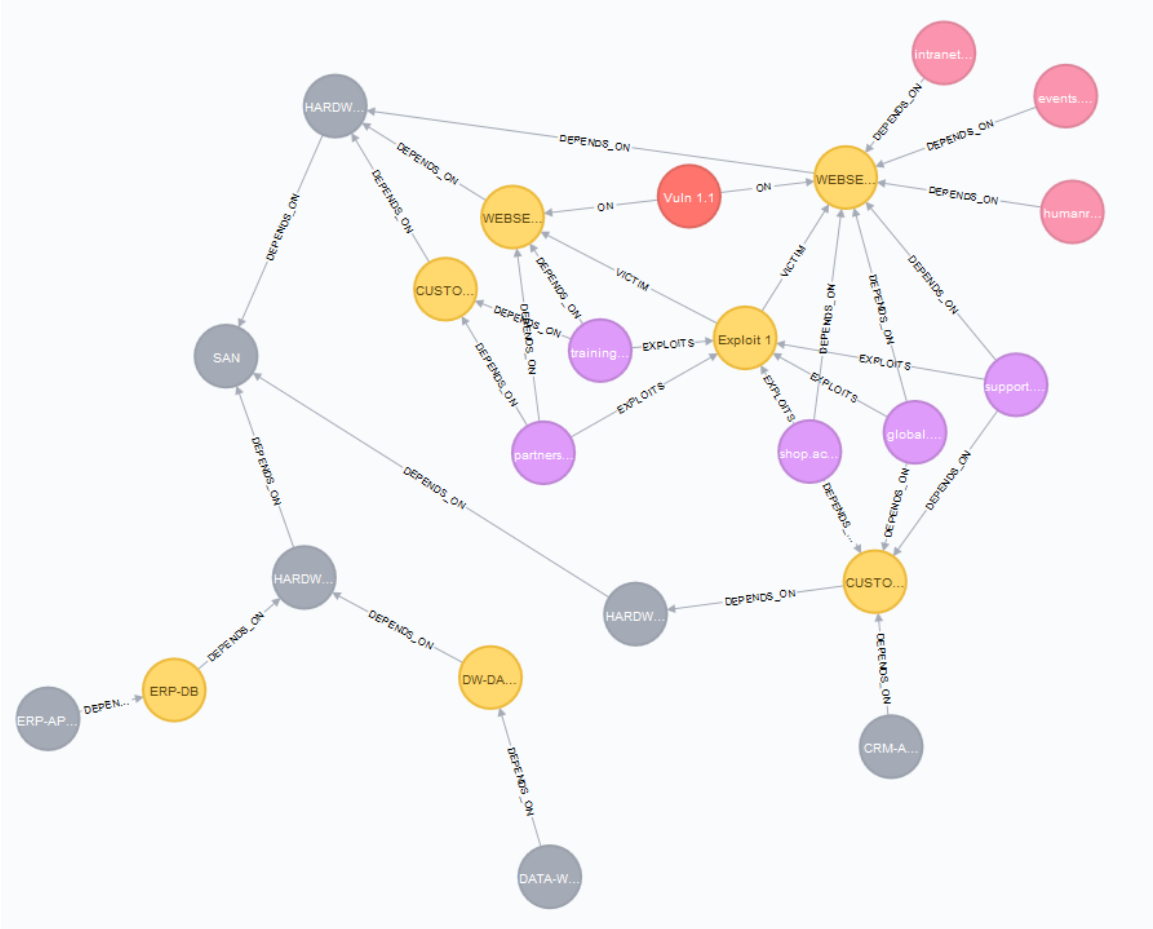


Figure 28.1. Graph after adding vulnerability node to affected machines.

Doing a selection from this graph for the relevant sections results in the following where the public facing internet websites can exploit the struts vulnerability on the web server hosting it.

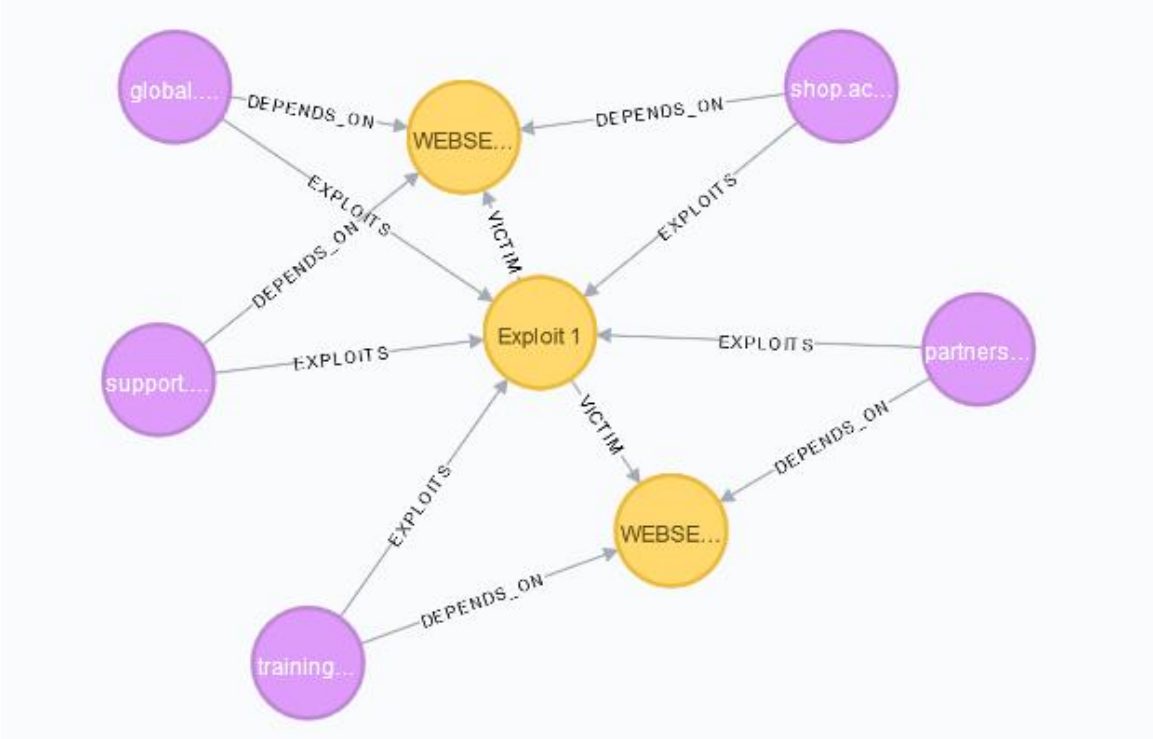


Figure 28.2. Subgraph showing only machines vulnerable to Apache struts.

The small-scale test shows that the graph database was able to dynamically add hosted websites and connect it to the potentially vulnerable web server. This section showed how to implement the framework presented by Noel et al. to identify the potentially exploitable nodes. A few more nodes and network devices will be added when moving up to a larger system, causing the relations to become more complicated. Each subsequent test increases the number of node traversals required to find the affected nodes.

Data Collection–Medium Scale

Taking the previous data center and modifying it to contain multiple data centers with various connections requires the addition of a few networking structures such as firewalls and the connections between those various firewalls and data centers. In the

following figure two switches have been added as well as an internal firewall between the switches acting as a routing device:

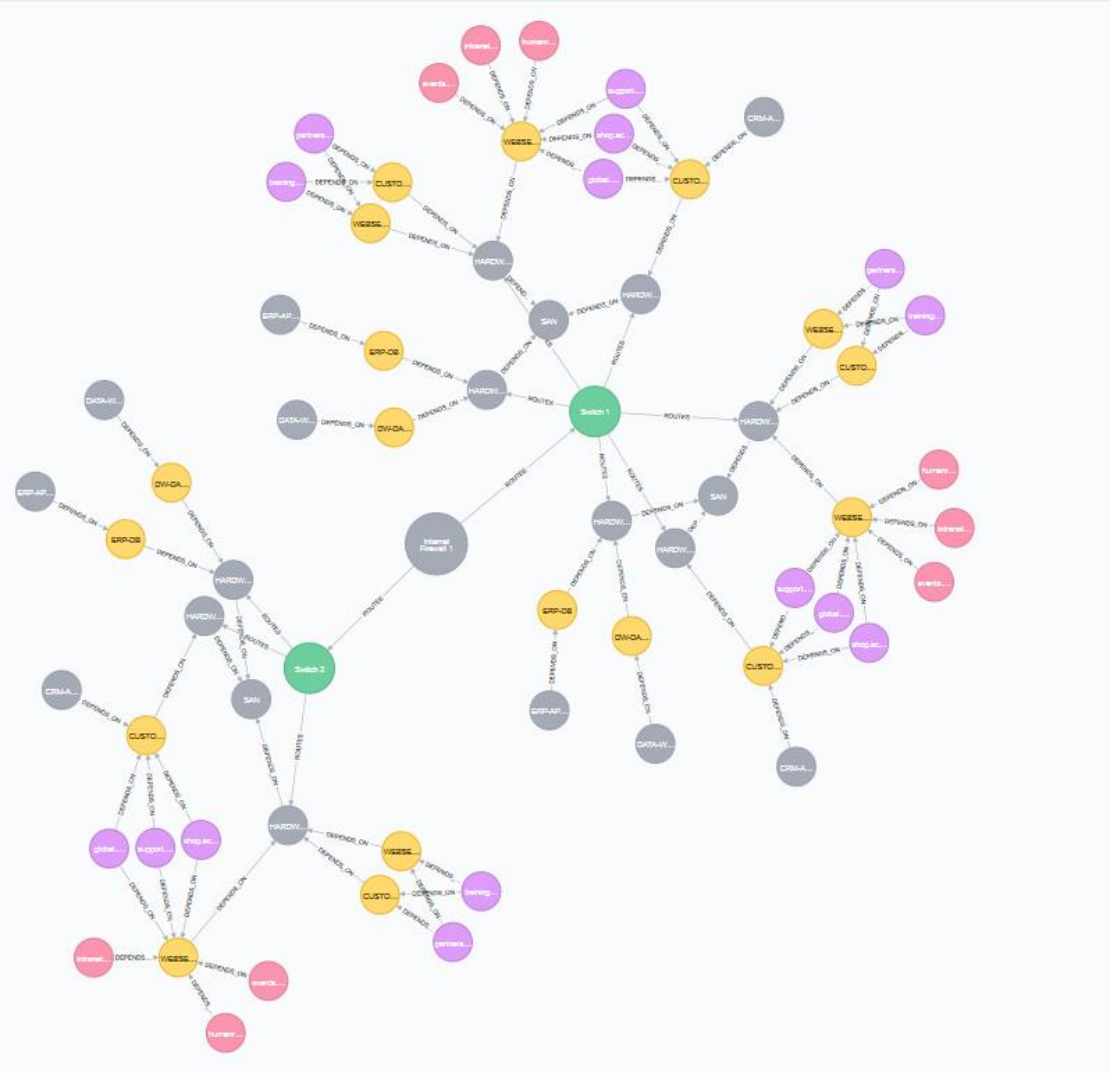


Figure 29. Medium-scale datacenter represented within Neo4J.

The newly added data center is a mirror of the previous nodes with the switch and firewall nodes added with the code snippet below:

```
CREATE (switch1:Switch {
        name: "Switch 1",
```

```
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })
CREATE (switch2_1:Switch {
        name: "Switch 2",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })
CREATE (firewall1:Firewall {
        name: "Internal Firewall 1",
                type: "FIREWALL",
        system: "PHYSICAL FIREWALL"
    })
```

The above snippets are fairly straight forward and create nodes with a name, type, and system value. The following section creates relationships between those various nodes:

```
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-SERVER-1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-SERVER-3" CREATE (n)-[:ROUTES]->(m)
```

The above block of code assigns relationships between hardware servers and their corresponding switch, the same is done for the second switch. The below snippet creates the relationship between the internal firewall and the newly added switches.

```
MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 1" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 2" CREATE (n)-[:ROUTES]→(m)
```

Both newly added topologies are the same as the first structurally just using different node names. The exploit node and its relationships are re-added using the same query from the small-scale example. The vulnerability node is omitted since each node is tagged with vulnerability data directly so it is unneeded for this data set. Running the query from earlier shows a similar output from before but with the new web servers and the respective exploits ↔ victim relationship:

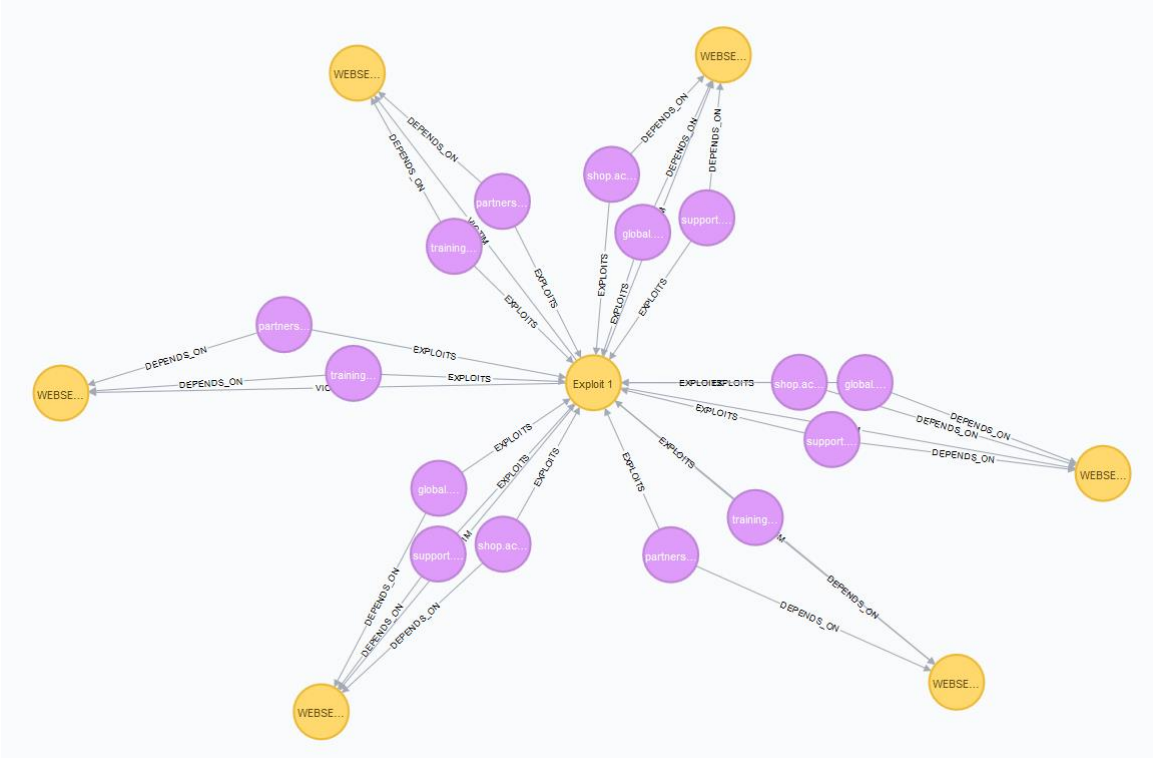


Figure 29.1. Subgraph showing medium-scale machines vulnerable to Apache struts.

Data Collection–Large-scale

In the large-scale test, an additional data center was added connected to another switch and internal firewall. In this way, the separate graphs can be seen to be on a separate domain, though they can also be linked through a DMZ or internet node and queried separately from there as well depending on the architecture being mapped within the graph database. The new graph can be seen below:

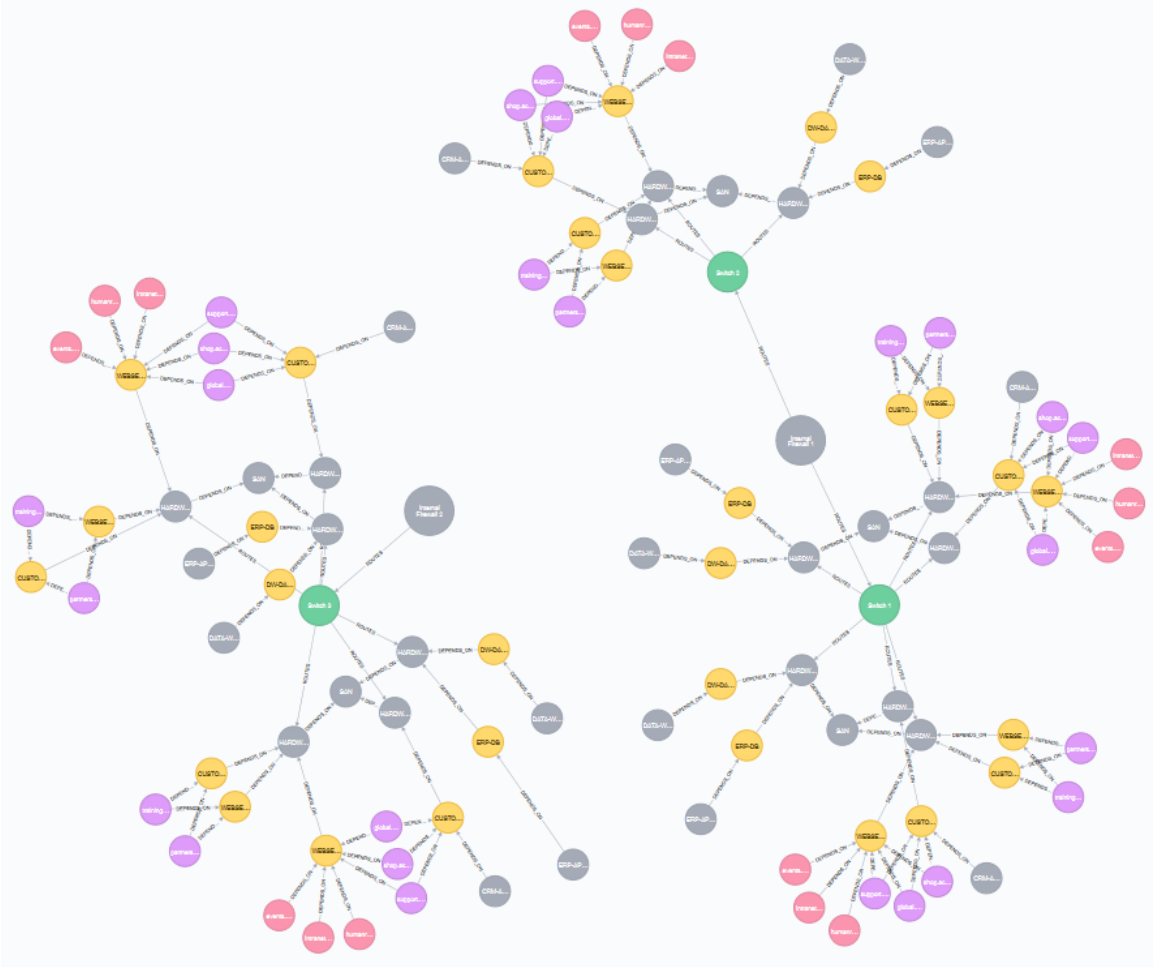


Figure 30. Large-scale datacenter with split domains.

Again the exploit node and its subsequent connections are created resulting in the following graph:

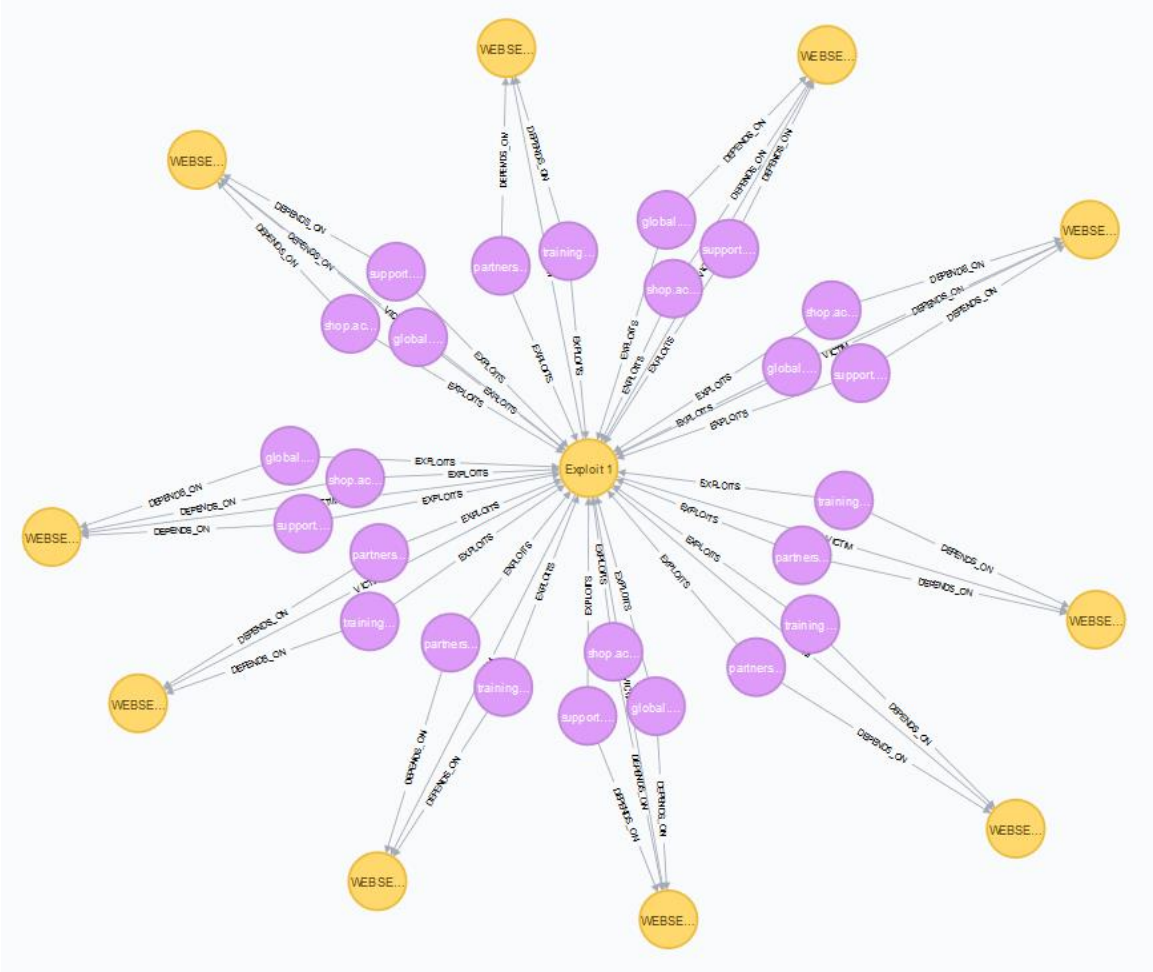


Figure 30.1. Subgraph showing large-scale machines vulnerable to Apache struts.

In a situation like above, the search is not domain agnostic and in some cases that could prove problematic. If only one domain is suspected of being infected there would be a large number of false positives created by querying the entire dataset. The query can be modified to the following to query a subgraph of a subgraph to return domain-specific results:

```
MATCH (switchvar:Switch { name: "Switch 3" })←[:DEPENDS_ON| ROUTES]-
>(vulnmachine:Hardware)
WITH vulnmachine
```

```
MATCH (vulnmachine)←[:DEPENDS_ON]-(webserver)←[VICTIM|EXPLOITS]-
>(exploiter)
WHERE webserver.version = "Apache Struts 2.2.0"
RETURN *
```

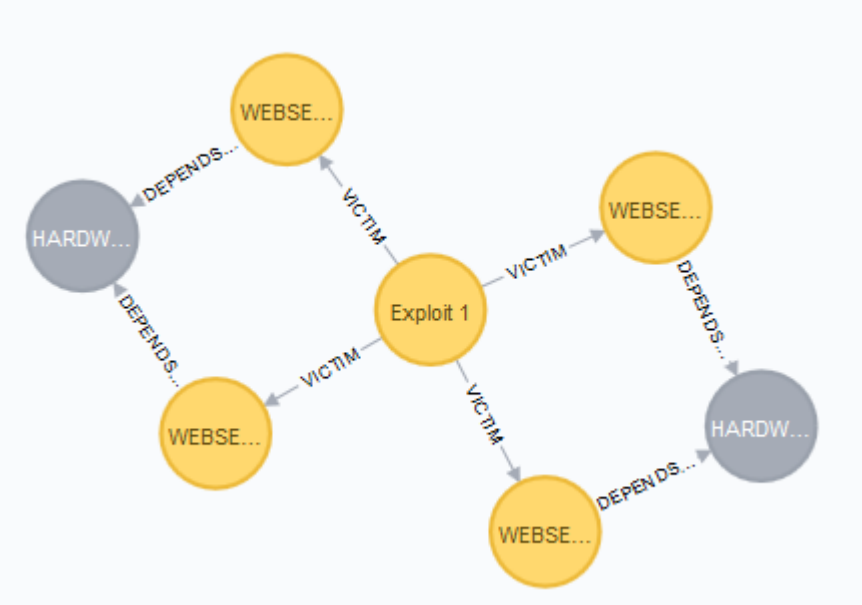


Figure 30.2. Subgraph showing machines on network Switch 3 vulnerable to Apache struts.

The above results show the vulnerable web servers and hardware on the new domain under the newly added “Switch 3”. The same can be done for the other switches and results in similar graphs displaying the correct vulnerable web servers on their respective domains.

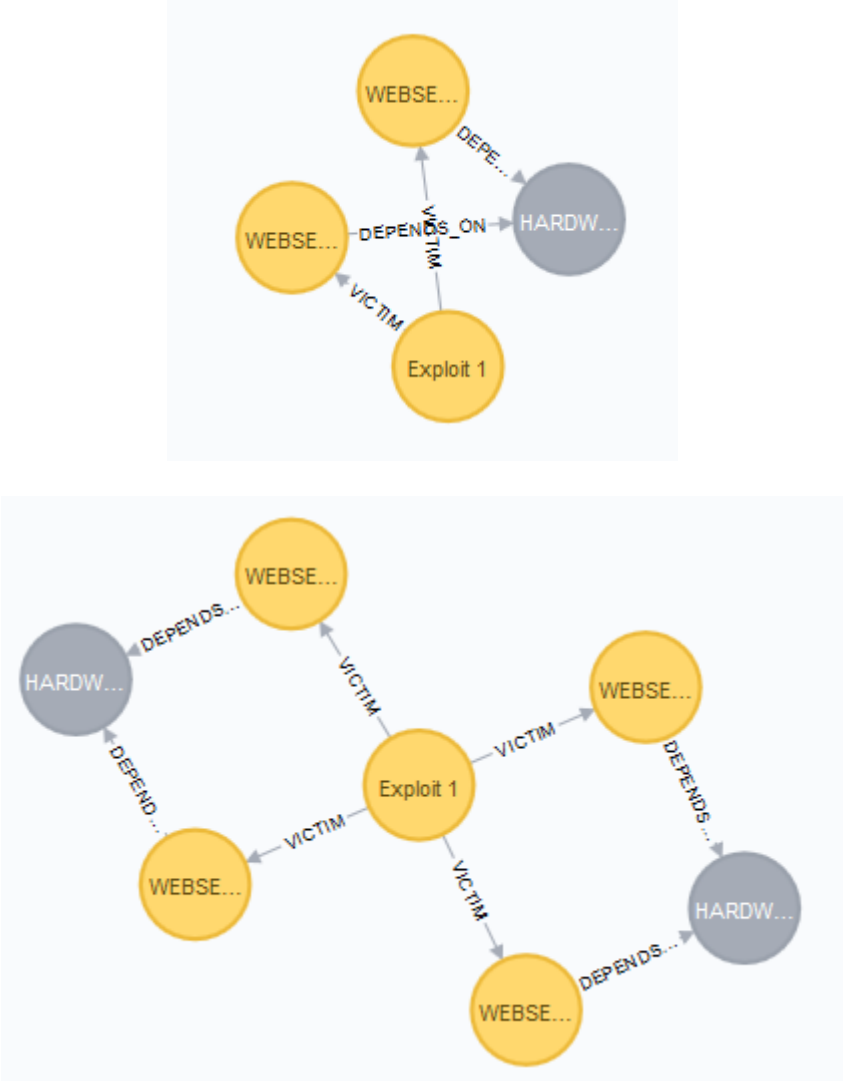


Figure 30.3. Subgraphs showing machines on network Switch 2 & 1 vulnerable to Apache struts.

First is “Switch 2” and its vulnerable hardware, second is “Switch 1” and its vulnerable hardware. All of these queries completed in less than one second even though the needed number of traversals to find the nodes exceeds four.

Summary

In this section, small, medium, and large-scale tests were performed to test on the fly additions of exploit nodes and the ability to create an accurate blast radius

summary from that data. Along with that diving into a specific section of a subgraph of a subgraph was done in order to show graph databases ability to address the network problem and still return a blast radius for a specific network segment.

Chapter 4: Data Presentation and Analysis

Introduction

In this chapter, there will be an analysis of the small, medium, and large-scale tests and their results. With each set of results, the expected outputs versus the actual outputs will be compared.

Data Presentation–Small-scale

In this first experiment, the basic framework format by Noel et al. (2014) was implemented into a test data center architecture. The data center was modified to contain information about machines that could be used to find the Apache Struts vulnerability within the graph. This first set of tests was designed to show a basic proof of concept for how a potential zero-day vulnerability could be correlated and found by taking a subgraph of the existing graph. In this test set, there were only two vulnerable web servers and a total of five external websites capable of exploiting those web servers.

Table 3

Small-scale Expected vs Generated Results

	Expected	Results
Web servers	2	2
Exploit sites	5	5

Data Presentation–Medium-scale

In this set of tests, more data centers were added to the existing one from the small-scale test and some networking equipment was added between them. This set of

tests shows that with more nodes and complexity added the graph is able to return the affected nodes and the potential tangentially related attack nodes. The affected nodes are returned and are related to their vulnerable web server directly in the resulting subgraph. Returning the potentially vulnerable web sites tied to their requisite web server allows for fast remediation and prioritization based on web portal size or importance to the organization.

Table 4

Medium-scale Expected vs Generated Results

	Expected	Results
Web servers	6	6
Exploit sites	15	15

Data Presentation–Large-scale

This set of tests again added more potentially vulnerable web servers and a different domain that is disconnected from the main graph. In running this set of tests, the same test query was used to generate the following results:

Table 5

Large-scale Expected vs Generated Results

	Expected	Results
Web servers	10	10
Exploit sites	25	25

As can be seen in the subgraph having a disconnected graph had no bearing on the results due to the nature of implementing the exploit nodes relationships based on data tagged to the existing nodes. This section's data collection also showcased the ability to take a subgraph of a subgraph in order to gather the affected web server nodes from the new domain added for the large-scale tests.

Data Analysis

The data collected works as a proof of concept to show that on various scales with differing levels of network complexity it is possible to add and detect zero-day attacks given the attack vector and version of software or hardware being exploited exists within the graph database. This is shown with the results given in the small, medium, and large-scale tests leveraging the framework provided in Noel et al. (2014). Along with being able to add exploit nodes on the fly using the framework presented by Noel et al. (2014) it points towards a possible method to explore zero-day exploits that may proliferate across domains should a zero-day leverage multiple different attack vectors.

Summary

In this chapter, the expected results were compared with the results gathered from querying the graph database. Manually doing an analysis of the graph nodes showed that it correlated correctly with the results procedurally generated based on node data. Along with that analysis showing the potential usage of querying a subgraph of a subgraph as a possibility for narrowing the search scope assists in performing reconnaissance on specific sections of the network.

Chapter 5: Results, Conclusions, and Recommendations

Introduction

In this section, the importance and implications of the gathered data are explored and analyzed. The tests are first analyzed for their success as a proof of concept; next, the potential exceptions are outlined and detailed. Moreover, the potential for use of the framework presented by Noel et al. (2014) being leveraged for more than just threat and vulnerability management. Second, problems potentially present when visualizing data with Neo4j's built-in tools is elucidated and outlined for potential use in future works.

Results

The key question posed in the methodology is if it is possible to use a graph database to detect and address zero-day vulnerabilities given data exists to identify potentially at-risk systems within the graph database. The results show that in the case of the Apache Struts vulnerability which was used as the test case it is possible to detect vulnerable websites and web servers given the criterion mentioned earlier is met. This may not hold true in all cases however as certain zero-day vulnerabilities may leverage unknown attack vectors that cannot easily be stored within a graph database ahead of time. One example of this would be something such as Van Eck Phreaking that involves measuring side-band electromagnetic emissions. Vulnerability to such an attack vector would generally not be logged anywhere and would be difficult if not impossible to quickly create a query for so that an exploit node and relationships could be created. In other cases, however, such as Petya or Spectre, where a hardware vulnerability or a particular version of the software is exploited, the vulnerable machines

can be identified fairly easily based on existing versioning or hardware information. As such exploit nodes can be created dynamically and the resulting blast radius can be determined. This holds true even in the face of the network problem due to the nature of graph databases and their usage of index-free adjacency. In the case of Petya, it is possible to go a step further as its infection vector is known along with its intended victim. If the nodes within the graph contain information on the infection vector, in the case of Petya it's the allowance of the SMB protocol between machines, as well as the operating system of the machine that's vulnerable in this case windows. Then it is possible to use the cross-domain vulnerability chains presented by Noel et al. (2014) to find potential cross domain exploits using the method for dynamically generating exploit nodes presented in this paper.

Moving forward to visualizations, Neo4j has a built-in visualization tool that makes understanding basic graphs or subsections of a graph easy. A problem arises when the need to visualize large pools of nodes arises. Neo4j can customize the number of displayed nodes in each section or subsection of the graph to accommodate this problem; however, it does not always result in being the best fit for visualizing that particular data set. The best visualization framework for a particular situation is requirements dependent and as such relies on stakeholder or situations needs. As such the basic visualization provided by Neo4J was sufficient for the visualization needs of this paper and as such was used to display the various results.

The focus of this paper was on the creation of exploit nodes on the fly to detect zero-day vulnerabilities and not analyzing best-fit visualizations for that data set. As

such the groundwork has been laid for future work relating to visualization methods for security data and finding the pros and cons of the various methods of visualizing that data.

Conclusion

The goal of this study was to provide a proof of concept for determining the blast radius of a newly published zero-day vulnerability given the data to determine what software or hardware is potentially at risk is known and contained within the graph database. In the test cases presented the tested zero-day vulnerability was properly detected and displayed the correct relationships when queried. In these test cases, it was a simple relation that website --exploits→ web server, in other cases this relationship may extend farther due to other vulnerabilities existing on the system. This can also be detected using the framework presented by Noel et al. (2014) after the new zero-day vulnerability exploit nodes are added. Along with the proof of concept querying subgraphs of subgraphs was also covered to show the power of graph databases in addressing the network problem to generate a more specific blast radius for specific sections of the network.

Future Work

Studying different zero-day vulnerabilities and the infection vectors leveraged by those vulnerabilities can assist in building upon the framework presented by Noel et al. (2014) as well as show potential holes present in the overall graph's construction that could show where there might be missing information needed to detect outlier zero-day vulnerabilities. As an example, information pertaining to SMB access across domains

would need to be documented to detect the Petya vulnerability. Furthermore, study into zero-day vulnerabilities that leverage multiple vectors of attack such as the first example in Figure 4 of Noel et al. (2014) and how to exploit nodes for them can be added and correctly linked would help protect against more advanced persistent threats more likely to leverage something beyond simple malware.

Future work pertaining to various visualization methods and the benefits as well as weaknesses of those methods is an open area of research. Using the resources provided by Neo4j in their developer section various integrations with graphing solutions are outlined and explored. Taking these solutions and performing an analysis on their uses when integrated with security data as presented in this paper to find the strengths and weaknesses of a multitude of different visualization methods is an open area that requires more research.

References

Angles, R. (2012). A comparison of current graph database models. *2012 IEEE 28th International Conference on Data Engineering Workshops*, pp. 171-177. doi:10.1109/icdew.2012.31

Anvaka. (2019, March 03). *Anvaka/VivaGraphJS*. Retrieved April 24, 2019, from https://github.com/anvaka/VivaGraphJS

Bastani, K. (n.d.). *Network dependency graph*. Retrieved January 15, 2019, from https://neo4j.com/graphgist/network-dependency-graph

Berg, K. L., Seymour, T., & Goel, R. (2012). History of databases. *International Journal of Management & Information Systems (IJMIS)*, *17*(1), 29-36. doi:10.19030/ijmis.v17i1.7587

Bilge, L., & Dumitras, T. (2012). Before we knew it: An empirical study of zero-day attacks in the real world. *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS 12*, 833-844. doi:10.1145/2382196.2382284

Bostock, M. (2019, April 12). *Collapsible force layout*. Retrieved April 22, 2019, from https://bl.ocks.org/mbostock/1062288

Bostock, M. (2019, March 01). *Zoomable circle packing*. Retrieved April 22, 2019, from https://observablehq.com/@d3/zoomable-circle-packing

Buerli, M. (2012, December). *The current state of graph databases*. Cal Poly, San Luis Obispo.

Hong, L., Zou, L., Lian, X., & Yu, P. S. (2015). Subgraph matching with set similarity in a large graph database*. IEEE Transactions on Knowledge and Data Engineering, 27*(9), 2507-2521. doi:10.1109/tkde.2015.2391125

Iliinsky, N., & Steele, J. (2011). *Designing data visualizations.* Sebastopol: OReilly. doi:http://courses.ischool.utexas.edu/unmil/files/Designing_Data_Visualizations.pdf

J. (2018, December 18). *Stolen credit cards and fraud detection with Neo4j*. Retrieved April 29, 2019, from https://linkurio.us/blog/stolen-credit-cards-and-fraud-detection-with-neo4j/

Jacomy, A., & Plique, G. (n.d.). *Sigmajs*. Retrieved April 24, 2019, from http://sigmajs.org/

KeyLines Network Visualization Software. (2019). Retrieved April 29, 2019, from https://cambridge-intelligence.com/keylines/

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. *USENIX Security Symposium.*

Lyon, W. (2016, June 26). Analyzing the graph of thrones - network analysis with Neo4j. Retrieved April 22, 2019, from https://www.lyonwj.com/2016/06/26/graph-of-thrones-neo4j-social- network-analysis/

Neo4j Graph Platform–The Leader in Graph Databases. (n.d.). Retrieved November 7, 2019, from https://neo4j.com/

Noel, S., Harley, E., Tam, K.H., & Gyor, G. (2014). Big-data architecture for cyber attack graphs representing security relationships in NoSQL graph databases.

Popoto.js. (2018). Retrieved from http://www.popotojs.com/

Ransom.Wannacry. (2017, May 12). Retrieved January 30, 2019, from https://www.symantec.com/security-center/writeup/2017-051310-3522-99

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). Sebastopol, CA: OReilly & Associates. doi:https://pdfs.semanticscholar.org/f511/7084ca43e888fb3e17ab0f0e684cced0f8fd.pdf

S. (2017). R/Malware—all Petya Ransomware write-up. Retrieved February 28, 2019, from https://www.reddit.com/r/Malware/comments/

Sontrop, H. (2018, October 4). *Radial tidy tree*. Retrieved from https://bl.ocks.org/FrissAnalytics/974dc299c5bc79cc5fd7ee9fa1b0b366

Tom Sawyer Perspectives. (2019). Retrieved April 29, 2019, from https://www.tomsawyer.com/perspectives/

Venkatesh, S. (2014). *Subgraph pattern matching for graph databases* (Doctoral dissertation), University of Georgia. doi:https://getd.libs.uga.edu/pdfs/venkatesh_sumana_201412_ms.pdf

Visjs. Retrieved April 22, 2019, from http://visjs.org/examples/network/edgeStyles/smoothWorldCup.html

Weber, S. (n.d.). *NoSQL databases*. Retrieved April 29, 2019, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.468.7089&rep=rep1&type=pdf

YFiles Product Family. (2019). Retrieved April 29, 2019, from https://www.yworks.com/products/yfiles

Appendix

```
// Create CRM
CREATE (crm1:Application {
        ip:'10.10.32.1',
        host:'CRM-APPLICATION',
        type: 'APPLICATION',
        system: 'CRM'
    })

// Create ERP
CREATE (erp1:Application {
        ip:'10.10.33.1',
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })

// Create Data Warehouse
CREATE (datawarehouse1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
```

```
        system: 'DW'
    })

// Create Public Website 1
CREATE (Internet1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 2
CREATE (Internet2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet3:Internet {
        ip:'10.10.35.3',
```

```
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 4
CREATE (Internet4:Internet {
        ip:'10.10.35.4',
        host:'training.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 5
CREATE (Internet5:Internet {
        ip:'10.10.35.1',
        host:'partners.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Internal Website 1
```

```
CREATE (Intranet1:Intranet {
        ip:'10.10.35.2',
        host:'events.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet2:Intranet {
        ip:'10.10.35.3',
        host:'intranet.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 3
CREATE (Intranet3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })
```

```
// Create Webserver VM 1
CREATE (webservervm1:VirtualMachine {
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Webserver VM 2
CREATE (webservervm2:VirtualMachine {
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase1:VirtualMachine {
        ip:'10.10.35.7',
```

```
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 2
CREATE (customerdatabase2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 3
CREATE (databasevm3:VirtualMachine {
        ip:'10.10.35.9',
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
```

```
CREATE (dwdatabase:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware1:Hardware {
        ip:'10.10.35.11',
        host:'HARDWARE-SERVER-1',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
CREATE (hardware2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })
```

```
// Create Hardware 3
CREATE (hardware3:Hardware {
        ip:'10.10.35.13',
        host:'HARDWARE-SERVER-3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
CREATE (san1:Hardware {
        ip:'10.10.35.14',
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Connect CRM to Database VM 1
CREATE (crm1)-[:DEPENDS_ON]->(customerdatabase1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet1)-[:DEPENDS_ON]->(customerdatabase1),
```

```
        (Internet2)-[:DEPENDS_ON]->(customerdatabase1),
        (Internet3)-[:DEPENDS_ON]->(customerdatabase1)

// Connect Database VM 1 to Hardware 1
CREATE  (customerdatabase1)-[:DEPENDS_ON]->(hardware1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware1)-[:DEPENDS_ON]->(san1)

// Connect Public Websites 1-3 to Webserver VM 1
CREATE  (webservervm1)<-[:DEPENDS_ON]-(Internet1),
        (webservervm1)<-[:DEPENDS_ON]-(Internet2),
        (webservervm1)<-[:DEPENDS_ON]-(Internet3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm1)<-[:DEPENDS_ON]-(Intranet1),
        (webservervm1)<-[:DEPENDS_ON]-(Intranet2),
        (webservervm1)<-[:DEPENDS_ON]-(Intranet3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm1)-[:DEPENDS_ON]->(hardware2)
```

```
// Connect Hardware 2 to SAN 1
CREATE  (hardware2)-[:DEPENDS_ON]->(san1)

// Connect Webserver VM 2 to Hardware 2
CREATE  (webservervm2)-[:DEPENDS_ON]->(hardware2)

// Connect Public Websites 4-6 to Webserver VM 2
CREATE  (webservervm2)<-[:DEPENDS_ON]-(Internet4),
        (webservervm2)<-[:DEPENDS_ON]-(Internet5)

// Connect Database VM 2 to Hardware 2
CREATE  (hardware2)<-[:DEPENDS_ON]-(customerdatabase2)

// Connect Public Websites 4-5 to Database VM 2
CREATE  (Internet4)-[:DEPENDS_ON]->(customerdatabase2),
        (Internet5)-[:DEPENDS_ON]->(customerdatabase2)

// Connect Hardware 3 to SAN 1
CREATE  (hardware3)-[:DEPENDS_ON]->(san1)

// Connect Database VM 3 to Hardware 3
CREATE  (hardware3)<-[:DEPENDS_ON]-(databasevm3)
```

```
// Connect ERP 1 to Database VM 3
CREATE  (erp1)-[:DEPENDS_ON]->(databasevm3)

// Connect Database VM 4 to Hardware 3
CREATE  (hardware3)<-[:DEPENDS_ON]-(dwdatabase)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse1)-[:DEPENDS_ON]->(dwdatabase)

RETURN *

CREATE (vuln1:Vulnerability {
        name: "Vuln 1.1",
                CVE_number: "CVE-2010-1870",
                description: "Apache Struts Remote Command Execution",
                port_requirements: "80",
                software_requirements:"Apache Struts <2.2.0"
    })

//MATCH (n), (m) WHERE n.version="Apache Struts 2.2.0" AND
m.CVE_number="CVE-2010-1870" CREATE (n)-[:ON]->(m)
```

```
MATCH (n), (m) WHERE n.CVE_number="CVE-2010-1870" AND
m.version="Apache Struts 2.2.0" CREATE (n)-[:ON]->(m)

CREATE (exploit1:Exploit {
        name: "Exploit 1",
                description: "Apache Struts Remote Command Execution",
                software_requirements:"Apache Struts <2.2.0"
    })

MATCH (n), (m) WHERE n.system="INTERNET" AND m.name="Exploit 1" CREATE
(n)-[:EXPLOITS]->(m)

MATCH (n), (m) WHERE n.name="Exploit 1" AND m.version="Apache Struts 2.2.0"
CREATE (n)-[:VICTIM]->(m)

MATCH (n) WHERE n.version="Apache Struts 2.2.0" AND n.name="Exploit 1" AND
n.system="INTERNET" RETURN n

MATCH (exploit1:Exploit { name: "Exploit 1" })<-[:VICTIM|EXPLOITS]-
>(vulnmachine) RETURN *

//------------------------------------------------
```

```
CREATE (switch1:Switch {
        name: "Switch 1",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })

CREATE (switch2_1:Switch {
        name: "Switch 2",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })


CREATE (firewall1:Firewall {
        name: "Internal Firewall 1",
                type: "FIREWALL",
        system: "PHYSICAL FIREWALL"
    })
```

```
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 1"
CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 2"
CREATE (n)-[:ROUTES]->(m)



CREATE (vuln1:Vulnerability {
        name: "Vuln 1.1",
                CVE_number: "CVE-2010-1870",
                description: "Apache Struts Remote Command Execution",
```

```
                port_requirements: "80",
                software_requirements:"Apache Struts <2.2.0"
        })



MATCH (n) where id(n) = 44 DETACH DELETE n



------------------------------MEDIUM

// Create CRM
CREATE (crm1:Application {
        ip:'10.10.32.1',
        host:'CRM-APPLICATION',
        type: 'APPLICATION',
        system: 'CRM'
    })

// Create ERP
CREATE (erp1:Application {
        ip:'10.10.33.1',
```

```
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })

// Create Data Warehouse
CREATE (datawarehouse1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
        system: 'DW'
    })

// Create Public Website 1
CREATE (Internet1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 2
```

```
CREATE (Internet2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet3:Internet {
        ip:'10.10.35.3',
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 4
CREATE (Internet4:Internet {
        ip:'10.10.35.4',
        host:'training.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })
```

```
// Create Public Website 5
CREATE (Internet5:Internet {
        ip:'10.10.35.1',
        host:'partners.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Internal Website 1
CREATE (Intranet1:Intranet {
        ip:'10.10.35.2',
        host:'events.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet2:Intranet {
        ip:'10.10.35.3',
        host:'intranet.acme.net',
        type: "APPLICATION",
```

```
        system: "INTRANET"
    })

// Create Internal Website 3
CREATE (Intranet3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Webserver VM 1
CREATE (webservervm1:VirtualMachine {
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Webserver VM 2
CREATE (webservervm2:VirtualMachine {
```

```
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase1:VirtualMachine {
        ip:'10.10.35.7',
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 2
CREATE (customerdatabase2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })
```

```
// Create Database VM 3
CREATE (databasevm3:VirtualMachine {
        ip:'10.10.35.9',
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
CREATE (dwdatabase:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware1:Hardware {
        ip:'10.10.35.11',
        host:'HARDWARE-SERVER-1',
        type: "HARDWARE SERVER",
```

```
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
CREATE (hardware2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 3
CREATE (hardware3:Hardware {
        ip:'10.10.35.13',
        host:'HARDWARE-SERVER-3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
CREATE (san1:Hardware {
        ip:'10.10.35.14',
```

```
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Connect CRM to Database VM 1
CREATE (crm1)-[:DEPENDS_ON]->(customerdatabase1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet1)-[:DEPENDS_ON]->(customerdatabase1),
    (Internet2)-[:DEPENDS_ON]->(customerdatabase1),
    (Internet3)-[:DEPENDS_ON]->(customerdatabase1)

// Connect Database VM 1 to Hardware 1
CREATE  (customerdatabase1)-[:DEPENDS_ON]->(hardware1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware1)-[:DEPENDS_ON]->(san1)

// Connect Public Websites 1-3 to Webserver VM 1
CREATE  (webservervm1)<-[:DEPENDS_ON]-(Internet1),
    (webservervm1)<-[:DEPENDS_ON]-(Internet2),
```

```
        (webservervm1)<-[:DEPENDS_ON]-(Internet3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm1)<-[:DEPENDS_ON]-(Intranet1),
        (webservervm1)<-[:DEPENDS_ON]-(Intranet2),
        (webservervm1)<-[:DEPENDS_ON]-(Intranet3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm1)-[:DEPENDS_ON]->(hardware2)

// Connect Hardware 2 to SAN 1
CREATE  (hardware2)-[:DEPENDS_ON]->(san1)

// Connect Webserver VM 2 to Hardware 2
CREATE  (webservervm2)-[:DEPENDS_ON]->(hardware2)

// Connect Public Websites 4-6 to Webserver VM 2
CREATE  (webservervm2)<-[:DEPENDS_ON]-(Internet4),
        (webservervm2)<-[:DEPENDS_ON]-(Internet5)

// Connect Database VM 2 to Hardware 2
CREATE  (hardware2)<-[:DEPENDS_ON]-(customerdatabase2)
```

```
// Connect Public Websites 4-5 to Database VM 2
CREATE  (Internet4)-[:DEPENDS_ON]->(customerdatabase2),
      (Internet5)-[:DEPENDS_ON]->(customerdatabase2)

// Connect Hardware 3 to SAN 1
CREATE  (hardware3)-[:DEPENDS_ON]->(san1)

// Connect Database VM 3 to Hardware 3
CREATE  (hardware3)<-[:DEPENDS_ON]-(databasevm3)

// Connect ERP 1 to Database VM 3
CREATE  (erp1)-[:DEPENDS_ON]->(databasevm3)

// Connect Database VM 4 to Hardware 3
CREATE  (hardware3)<-[:DEPENDS_ON]-(dwdatabase)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse1)-[:DEPENDS_ON]->(dwdatabase)

// Create CRM
CREATE (crm2_1:Application {
```

```
        ip:'10.10.32.1',
        host:'CRM-APPLICATION',
        type: 'APPLICATION',
        system: 'CRM'
    })

// Create ERP
CREATE (erp2_1:Application {
        ip:'10.10.33.1',
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })

// Create Data Warehouse
CREATE (datawarehouse2_1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
        system: 'DW'
    })
```

```
// Create Public Website 1
CREATE (Internet2_1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 2
CREATE (Internet2_2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet2_3:Internet {
        ip:'10.10.35.3',
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
```

```
        })

// Create Public Website 4
CREATE (Internet2_4:Internet {
            ip:'10.10.35.4',
            host:'training.acme.com',
            type: "APPLICATION",
            system: "INTERNET"
        })

// Create Public Website 5
CREATE (Internet2_5:Internet {
            ip:'10.10.35.1',
            host:'partners.acme.com',
            type: "APPLICATION",
            system: "INTERNET"
        })

// Create Internal Website 1
CREATE (Intranet2_1:Intranet {
            ip:'10.10.35.2',
            host:'events.acme.net',
```

```
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet2_2:Intranet {
        ip:'10.10.35.3',
        host:'intranet.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 3
CREATE (Intranet2_3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Webserver VM 1
CREATE (webservervm2_1:VirtualMachine {
```

```
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Webserver VM 2
CREATE (webservervm2_2:VirtualMachine {
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase2_1:VirtualMachine {
        ip:'10.10.35.7',
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
```

```
    })

// Create Database VM 2
CREATE (customerdatabase2_2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 3
CREATE (databasevm2_3:VirtualMachine {
        ip:'10.10.35.9',
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
CREATE (dwdatabase2:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
```

```
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware2_1:Hardware {
        ip:'10.10.35.11',
        host:'HARDWARE-SERVER-1',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
CREATE (hardware2_2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 3
CREATE (hardware2_3:Hardware {
```

```
        ip:'10.10.35.13',
        host:'HARDWARE-SERVER-3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
CREATE (san2_1:Hardware {
        ip:'10.10.35.14',
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Connect CRM to Database VM 1
CREATE (crm2_1)-[:DEPENDS_ON]->(customerdatabase2_1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet2_1)-[:DEPENDS_ON]->(customerdatabase2_1),
    (Internet2_2)-[:DEPENDS_ON]->(customerdatabase2_1),
    (Internet2_3)-[:DEPENDS_ON]->(customerdatabase2_1)
```

```
// Connect Database VM 1 to Hardware 1
CREATE  (customerdatabase2_1)-[:DEPENDS_ON]->(hardware2_1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware2_1)-[:DEPENDS_ON]->(san2_1)

// Connect Public Websites 1-3 to Webserver VM 1
CREATE  (webservervm2_1)<-[:DEPENDS_ON]-(Internet2_1),
        (webservervm2_1)<-[:DEPENDS_ON]-(Internet2_2),
        (webservervm2_1)<-[:DEPENDS_ON]-(Internet2_3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm2_1)<-[:DEPENDS_ON]-(Intranet2_1),
        (webservervm2_1)<-[:DEPENDS_ON]-(Intranet2_2),
        (webservervm2_1)<-[:DEPENDS_ON]-(Intranet2_3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm2_1)-[:DEPENDS_ON]->(hardware2_2)

// Connect Hardware 2 to SAN 1
CREATE  (hardware2_2)-[:DEPENDS_ON]->(san2_1)
```

```
// Connect Webserver VM 2 to Hardware 2

CREATE  (webservervm2_2)-[:DEPENDS_ON]->(hardware2_2)


// Connect Public Websites 4-6 to Webserver VM 2

CREATE  (webservervm2_2)<-[:DEPENDS_ON]-(Internet2_4),

        (webservervm2_2)<-[:DEPENDS_ON]-(Internet2_5)


// Connect Database VM 2 to Hardware 2

CREATE  (hardware2_2)<-[:DEPENDS_ON]-(customerdatabase2_2)


// Connect Public Websites 4-5 to Database VM 2

CREATE  (Internet2_4)-[:DEPENDS_ON]->(customerdatabase2_2),

        (Internet2_5)-[:DEPENDS_ON]->(customerdatabase2_2)


// Connect Hardware 3 to SAN 1

CREATE  (hardware2_3)-[:DEPENDS_ON]->(san2_1)


// Connect Database VM 3 to Hardware 3

CREATE  (hardware2_3)<-[:DEPENDS_ON]-(databasevm2_3)


// Connect ERP 1 to Database VM 3

CREATE  (erp2_1)-[:DEPENDS_ON]->(databasevm2_3)
```

```
// Connect Database VM 4 to Hardware 3
CREATE  (hardware2_3)<-[:DEPENDS_ON]-(dwdatabase2)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse2_1)-[:DEPENDS_ON]->(dwdatabase2)

RETURN *
//RETURN *


--------------------------------------------------------------------------

// Create CRM
CREATE (crm3_1:Application {
            ip:'10.10.32.1',
            host:'CRM-APPLICATION',
            type: 'APPLICATION',
            system: 'CRM'
        })

// Create ERP
```

```
CREATE (erp3_1:Application {
        ip:'10.10.33.1',
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })

// Create Data Warehouse
CREATE (datawarehouse3_1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
        system: 'DW'
    })

// Create Public Website 1
CREATE (Internet3_1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })
```

```
// Create Public Website 2
CREATE (Internet3_2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet3_3:Internet {
        ip:'10.10.35.3',
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 4
CREATE (Internet3_4:Internet {
        ip:'10.10.35.4',
        host:'training.acme.com',
        type: "APPLICATION",
```

```
        system: "INTERNET"
    })

// Create Public Website 5
CREATE (Internet3_5:Internet {
        ip:'10.10.35.1',
        host:'partners.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Internal Website 1
CREATE (Intranet3_1:Intranet {
        ip:'10.10.35.2',
        host:'events.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet3_2:Intranet {
        ip:'10.10.35.3',
```

```
        host:'intranet.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 3
CREATE (Intranet3_3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Webserver VM 1
CREATE (webservervm3_1:VirtualMachine {
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })
```

```
// Create Webserver VM 2
CREATE (webservervm3_2:VirtualMachine {
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase3_1:VirtualMachine {
        ip:'10.10.35.7',
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 2
CREATE (customerdatabase3_2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
```

```
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 3
CREATE (databasevm3_3:VirtualMachine {
        ip:'10.10.35.9',
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
CREATE (dwdatabase3:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware3_1:Hardware {
        ip:'10.10.35.11',
```

```
        host:'HARDWARE-SERVER-3_1',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
CREATE (hardware3_2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-3_2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 3
CREATE (hardware3_3:Hardware {
        ip:'10.10.35.13',
        host:'HARDWARE-SERVER-3_3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
```

```
CREATE (san3_1:Hardware {
        ip:'10.10.35.14',
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Connect CRM to Database VM 1
CREATE (crm3_1)-[:DEPENDS_ON]->(customerdatabase3_1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet3_1)-[:DEPENDS_ON]->(customerdatabase3_1),
    (Internet3_2)-[:DEPENDS_ON]->(customerdatabase3_1),
    (Internet3_3)-[:DEPENDS_ON]->(customerdatabase3_1)

// Connect Database VM 1 to Hardware 1
CREATE  (customerdatabase3_1)-[:DEPENDS_ON]->(hardware3_1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware3_1)-[:DEPENDS_ON]->(san3_1)

// Connect Public Websites 1-3 to Webserver VM 1
```

```
CREATE  (webservervm3_1)<-[:DEPENDS_ON]-(Internet3_1),
        (webservervm3_1)<-[:DEPENDS_ON]-(Internet3_2),
        (webservervm3_1)<-[:DEPENDS_ON]-(Internet3_3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm3_1)<-[:DEPENDS_ON]-(Intranet3_1),
        (webservervm3_1)<-[:DEPENDS_ON]-(Intranet3_2),
        (webservervm3_1)<-[:DEPENDS_ON]-(Intranet3_3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm3_1)-[:DEPENDS_ON]->(hardware3_2)

// Connect Hardware 2 to SAN 1
CREATE  (hardware3_2)-[:DEPENDS_ON]->(san3_1)

// Connect Webserver VM 2 to Hardware 2
CREATE  (webservervm3_2)-[:DEPENDS_ON]->(hardware3_2)

// Connect Public Websites 4-6 to Webserver VM 2
CREATE  (webservervm3_2)<-[:DEPENDS_ON]-(Internet3_4),
        (webservervm3_2)<-[:DEPENDS_ON]-(Internet3_5)
```

```
// Connect Database VM 2 to Hardware 2
CREATE  (hardware3_2)<-[:DEPENDS_ON]-(customerdatabase3_2)

// Connect Public Websites 4-5 to Database VM 2
CREATE  (Internet3_4)-[:DEPENDS_ON]->(customerdatabase3_2),
      (Internet3_5)-[:DEPENDS_ON]->(customerdatabase3_2)

// Connect Hardware 3 to SAN 1
CREATE  (hardware3_3)-[:DEPENDS_ON]->(san3_1)

// Connect Database VM 3 to Hardware 3
CREATE  (hardware3_3)<-[:DEPENDS_ON]-(databasevm3_3)

// Connect ERP 1 to Database VM 3
CREATE  (erp3_1)-[:DEPENDS_ON]->(databasevm3_3)

// Connect Database VM 4 to Hardware 3
CREATE  (hardware3_3)<-[:DEPENDS_ON]-(dwdatabase3)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse3_1)-[:DEPENDS_ON]->(dwdatabase3)
```

```
RETURN *


MATCH (n), (m) WHERE n.name = "Switch 2" AND m.host = "HARDWARE-
SERVER-3_1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 2" AND m.host = "HARDWARE-
SERVER-3_2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 2" AND m.host = "HARDWARE-
SERVER-3_3" CREATE (n)-[:ROUTES]->(m)



CREATE (vuln1:Vulnerability {
        name: "Vuln 1.1",
                CVE_number: "CVE-2010-1870",
                description: "Apache Struts Remote Command Execution",
                port_requirements: "80",
                software_requirements:"Apache Struts <2.2.0"
    })
```

```
//MATCH (n), (m) WHERE n.version="Apache Struts 2.2.0" AND
m.CVE_number="CVE-2010-1870" CREATE (n)-[:ON]->(m)
MATCH (n), (m) WHERE n.CVE_number="CVE-2010-1870" AND
m.version="Apache Struts 2.2.0" CREATE (n)-[:ON]->(m)

CREATE (exploit1:Exploit {
        name: "Exploit 1",
                description: "Apache Struts Remote Command Execution",
                software_requirements:"Apache Struts <2.2.0"
    })

MATCH (n), (m) WHERE n.system="INTERNET" AND m.name="Exploit 1" CREATE
(n)-[:EXPLOITS]->(m)

MATCH (n), (m) WHERE n.name="Exploit 1" AND m.version="Apache Struts 2.2.0"
CREATE (n)-[:VICTIM]->(m)

MATCH (n) WHERE n.version="Apache Struts 2.2.0" AND n.name="Exploit 1" AND
n.system="INTERNET" RETURN n

MATCH (exploit1:Exploit { name: "Exploit 1" })<-[:VICTIM|EXPLOITS]-
>(vulnmachine) RETURN *
```

```
// Create CRM
CREATE (crm4_1:Application {
        ip:'10.10.32.1',
        host:'CRM-APPLICATION',
        type: 'APPLICATION',
        system: 'CRM'
    })

// Create ERP
CREATE (erp4_1:Application {
        ip:'10.10.33.1',
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })
```

```
// Create Data Warehouse
CREATE (datawarehouse4_1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
        system: 'DW'
    })

// Create Public Website 1
CREATE (Internet4_1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 2
CREATE (Internet4_2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
```

```
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet4_3:Internet {
        ip:'10.10.35.3',
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 4
CREATE (Internet4_4:Internet {
        ip:'10.10.35.4',
        host:'training.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 5
CREATE (Internet4_5:Internet {
        ip:'10.10.35.1',
```

```
        host:'partners.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Internal Website 1
CREATE (Intranet4_1:Intranet {
        ip:'10.10.35.2',
        host:'events.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet4_2:Intranet {
        ip:'10.10.35.3',
        host:'intranet.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 3
```

```
CREATE (Intranet4_3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Webserver VM 1
CREATE (webservervm4_1:VirtualMachine {
        ip:'10.10.35.5',
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Webserver VM 2
CREATE (webservervm4_2:VirtualMachine {
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
```

```
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase4_1:VirtualMachine {
        ip:'10.10.35.7',
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 2
CREATE (customerdatabase4_2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 3
CREATE (databasevm4_3:VirtualMachine {
        ip:'10.10.35.9',
```

```
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
CREATE (dwdatabase4:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware4_1:Hardware {
        ip:'10.10.35.11',
        host:'HARDWARE-SERVER-4_1',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
```

```
CREATE (hardware4_2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-4_2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 3
CREATE (hardware4_3:Hardware {
        ip:'10.10.35.13',
        host:'HARDWARE-SERVER-4_3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
CREATE (san4_1:Hardware {
        ip:'10.10.35.14',
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })
```

```
// Connect CRM to Database VM 1
CREATE (crm4_1)-[:DEPENDS_ON]->(customerdatabase4_1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet4_1)-[:DEPENDS_ON]->(customerdatabase4_1),
        (Internet4_2)-[:DEPENDS_ON]->(customerdatabase4_1),
        (Internet4_3)-[:DEPENDS_ON]->(customerdatabase4_1)

// Connect Database VM 1 to Hardware 1
CREATE  (customerdatabase4_1)-[:DEPENDS_ON]->(hardware4_1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware4_1)-[:DEPENDS_ON]->(san4_1)

// Connect Public Websites 1-3 to Webserver VM 1
CREATE  (webservervm4_1)<-[:DEPENDS_ON]-(Internet4_1),
        (webservervm4_1)<-[:DEPENDS_ON]-(Internet4_2),
        (webservervm4_1)<-[:DEPENDS_ON]-(Internet4_3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm4_1)<-[:DEPENDS_ON]-(Intranet4_1),
```

```
        (webservervm4_1)<-[:DEPENDS_ON]-(Intranet4_2),
        (webservervm4_1)<-[:DEPENDS_ON]-(Intranet4_3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm4_1)-[:DEPENDS_ON]->(hardware4_2)

// Connect Hardware 2 to SAN 1
CREATE  (hardware4_2)-[:DEPENDS_ON]->(san4_1)

// Connect Webserver VM 2 to Hardware 2
CREATE  (webservervm4_2)-[:DEPENDS_ON]->(hardware4_2)

// Connect Public Websites 4-6 to Webserver VM 2
CREATE  (webservervm4_2)<-[:DEPENDS_ON]-(Internet4_4),
        (webservervm4_2)<-[:DEPENDS_ON]-(Internet4_5)

// Connect Database VM 2 to Hardware 2
CREATE  (hardware4_2)<-[:DEPENDS_ON]-(customerdatabase4_2)

// Connect Public Websites 4-5 to Database VM 2
CREATE  (Internet4_4)-[:DEPENDS_ON]->(customerdatabase4_2),
        (Internet4_5)-[:DEPENDS_ON]->(customerdatabase4_2)
```

```
// Connect Hardware 3 to SAN 1
CREATE  (hardware4_3)-[:DEPENDS_ON]->(san4_1)

// Connect Database VM 3 to Hardware 3
CREATE  (hardware4_3)<-[:DEPENDS_ON]-(databasevm4_3)

// Connect ERP 1 to Database VM 3
CREATE  (erp4_1)-[:DEPENDS_ON]->(databasevm4_3)

// Connect Database VM 4 to Hardware 3
CREATE  (hardware4_3)<-[:DEPENDS_ON]-(dwdatabase4)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse4_1)-[:DEPENDS_ON]->(dwdatabase4)

RETURN *


// Create CRM
CREATE (crm5_1:Application {
        ip:'10.10.32.1',
```

```
        host:'CRM-APPLICATION',
        type: 'APPLICATION',
        system: 'CRM'
    })

// Create ERP
CREATE (erp5_1:Application {
        ip:'10.10.33.1',
        host:'ERP-APPLICATION',
        type: 'APPLICATION',
        system: 'ERP'
    })

// Create Data Warehouse
CREATE (datawarehouse5_1:Application {
        ip:'10.10.34.1',
        host:'DATA-WAREHOUSE',
        type: 'DATABASE',
        system: 'DW'
    })

// Create Public Website 1
```

```
CREATE (Internet5_1:Internet {
        ip:'10.10.35.1',
        host:'global.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 2
CREATE (Internet5_2:Internet {
        ip:'10.10.35.2',
        host:'support.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 3
CREATE (Internet5_3:Internet {
        ip:'10.10.35.3',
        host:'shop.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })
```

```
// Create Public Website 4
CREATE (Internet5_4:Internet {
        ip:'10.10.35.4',
        host:'training.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Public Website 5
CREATE (Internet5_5:Internet {
        ip:'10.10.35.1',
        host:'partners.acme.com',
        type: "APPLICATION",
        system: "INTERNET"
    })

// Create Internal Website 1
CREATE (Intranet5_1:Intranet {
        ip:'10.10.35.2',
        host:'events.acme.net',
        type: "APPLICATION",
```

```
        system: "INTRANET"
    })

// Create Internal Website 2
CREATE (Intranet5_2:Intranet {
        ip:'10.10.35.3',
        host:'intranet.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Internal Website 3
CREATE (Intranet5_3:Intranet {
        ip:'10.10.35.4',
        host:'humanresources.acme.net',
        type: "APPLICATION",
        system: "INTRANET"
    })

// Create Webserver VM 1
CREATE (webservervm5_1:VirtualMachine {
        ip:'10.10.35.5',
```

```
        host:'WEBSERVER-1',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Webserver VM 2
CREATE (webservervm5_2:VirtualMachine {
        ip:'10.10.35.6',
        host:'WEBSERVER-2',
        type: "WEB SERVER",
        system: "VIRTUAL MACHINE",
                version: "Apache Struts 2.2.0"
    })

// Create Database VM 1
CREATE (customerdatabase5_1:VirtualMachine {
        ip:'10.10.35.7',
        host:'CUSTOMER-DB-1',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })
```

```
// Create Database VM 2
CREATE (customerdatabase5_2:VirtualMachine {
        ip:'10.10.35.8',
        host:'CUSTOMER-DB-2',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 3
CREATE (databasevm5_3:VirtualMachine {
        ip:'10.10.35.9',
        host:'ERP-DB',
        type: "DATABASE SERVER",
        system: "VIRTUAL MACHINE"
    })

// Create Database VM 4
CREATE (dwdatabase5:VirtualMachine {
        ip:'10.10.35.10',
        host:'DW-DATABASE',
        type: "DATABASE SERVER",
```

```
        system: "VIRTUAL MACHINE"
    })

// Create Hardware 1
CREATE (hardware5_1:Hardware {
        ip:'10.10.35.11',
        host:'HARDWARE-SERVER-5_1',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 2
CREATE (hardware5_2:Hardware {
        ip:'10.10.35.12',
        host:'HARDWARE-SERVER-5_2',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create Hardware 3
CREATE (hardware5_3:Hardware {
        ip:'10.10.35.13',
```

```
        host:'HARDWARE-SERVER-5_3',
        type: "HARDWARE SERVER",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Create SAN 1
CREATE (san5_1:Hardware {
        ip:'10.10.35.14',
        host:'SAN',
        type: "STORAGE AREA NETWORK",
        system: "PHYSICAL INFRASTRUCTURE"
    })

// Connect CRM to Database VM 1
CREATE (crm5_1)-[:DEPENDS_ON]->(customerdatabase5_1)

// Connect Public Websites 1-3 to Database VM 1
CREATE  (Internet5_1)-[:DEPENDS_ON]->(customerdatabase5_1),
    (Internet5_2)-[:DEPENDS_ON]->(customerdatabase5_1),
    (Internet5_3)-[:DEPENDS_ON]->(customerdatabase5_1)

// Connect Database VM 1 to Hardware 1
```

```
CREATE  (customerdatabase5_1)-[:DEPENDS_ON]->(hardware5_1)

// Connect Hardware 1 to SAN 1
CREATE  (hardware5_1)-[:DEPENDS_ON]->(san5_1)

// Connect Public Websites 1-3 to Webserver VM 1
CREATE  (webservervm5_1)<-[:DEPENDS_ON]-(Internet5_1),
        (webservervm5_1)<-[:DEPENDS_ON]-(Internet5_2),
        (webservervm5_1)<-[:DEPENDS_ON]-(Internet5_3)

// Connect Internal Websites 1-3 to Webserver VM 1
CREATE  (webservervm5_1)<-[:DEPENDS_ON]-(Intranet5_1),
        (webservervm5_1)<-[:DEPENDS_ON]-(Intranet5_2),
        (webservervm5_1)<-[:DEPENDS_ON]-(Intranet5_3)

// Connect Webserver VM 1 to Hardware 2
CREATE  (webservervm5_1)-[:DEPENDS_ON]->(hardware5_2)

// Connect Hardware 2 to SAN 1
CREATE  (hardware5_2)-[:DEPENDS_ON]->(san5_1)

// Connect Webserver VM 2 to Hardware 2
```

```
CREATE  (webservervm5_2)-[:DEPENDS_ON]->(hardware5_2)

// Connect Public Websites 4-6 to Webserver VM 2
CREATE  (webservervm5_2)<-[:DEPENDS_ON]-(Internet5_4),
        (webservervm5_2)<-[:DEPENDS_ON]-(Internet5_5)

// Connect Database VM 2 to Hardware 2
CREATE  (hardware5_2)<-[:DEPENDS_ON]-(customerdatabase5_2)

// Connect Public Websites 4-5 to Database VM 2
CREATE  (Internet5_4)-[:DEPENDS_ON]->(customerdatabase5_2),
        (Internet5_5)-[:DEPENDS_ON]->(customerdatabase5_2)

// Connect Hardware 3 to SAN 1
CREATE  (hardware5_3)-[:DEPENDS_ON]->(san5_1)

// Connect Database VM 3 to Hardware 3
CREATE  (hardware5_3)<-[:DEPENDS_ON]-(databasevm5_3)

// Connect ERP 1 to Database VM 3
CREATE  (erp5_1)-[:DEPENDS_ON]->(databasevm5_3)
```

```
// Connect Database VM 4 to Hardware 3
CREATE  (hardware5_3)<-[:DEPENDS_ON]-(dwdatabase5)

// Connect Data Warehouse 1 to Database VM 4
CREATE  (datawarehouse5_1)-[:DEPENDS_ON]->(dwdatabase5)

RETURN *



CREATE (switch1:Switch {
        name: "Switch 1",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })

CREATE (switch2_1:Switch {
        name: "Switch 2",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })
```

```
CREATE (firewall1:Firewall {
        name: "Internal Firewall 1",
                type: "FIREWALL",
        system: "PHYSICAL FIREWALL"
    })
```

```
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 1" AND m.host = "HARDWARE-
SERVER-3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 1"
CREATE (n)-[:ROUTES]->(m)
```

```
MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 2"
CREATE (n)-[:ROUTES]->(m)



CREATE (vuln1:Vulnerability {
        name: "Vuln 1.1",
                CVE_number: "CVE-2010-1870",
                description: "Apache Struts Remote Command Execution",
                port_requirements: "80",
                software_requirements:"Apache Struts <2.2.0"
    })

//MATCH (n), (m) WHERE n.version="Apache Struts 2.2.0" AND
m.CVE_number="CVE-2010-1870" CREATE (n)-[:ON]->(m)
MATCH (n), (m) WHERE n.CVE_number="CVE-2010-1870" AND
m.version="Apache Struts 2.2.0" CREATE (n)-[:ON]->(m)

CREATE (exploit1:Exploit {
        name: "Exploit 1",
                description: "Apache Struts Remote Command Execution",
                software_requirements:"Apache Struts <2.2.0"
    })
```

```
MATCH (n), (m) WHERE n.system="INTERNET" AND m.name="Exploit 1" CREATE
(n)-[:EXPLOITS]->(m)

MATCH (n), (m) WHERE n.name="Exploit 1" AND m.version="Apache Struts 2.2.0"
CREATE (n)-[:VICTIM]->(m)

MATCH (n) WHERE n.version="Apache Struts 2.2.0" AND n.name="Exploit 1" AND
n.system="INTERNET" RETURN n

MATCH (exploit1:Exploit { name: "Exploit 1" })<-[:VICTIM|EXPLOITS]-
>(vulnmachine) RETURN *

MATCH (switchvar:Switch { name: "Switch 3" })<-[:DEPENDS_ON]->(vulnmachine)
WITH vulnmachine
MATCH p=(vulnmachine) WHERE n.type = "HARDWARE SERVER" RETURN (p)

MATCH (switchvar:Switch { name: "Switch 3" })<-[:DEPENDS_ON|ROUTES]-
>(vulnmachine:Hardware)
WITH vulnmachine
MATCH (vulnmachine)<-[:DEPENDS_ON]-(webserver)<-[VICTIM|EXPLOITS]-
>(exploiter)
```

```
WHERE webserver.version = "Apache Struts 2.2.0"
RETURN *

//------------------------------------------------


CREATE (switch1:Switch {
        name: "Switch 1",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })

CREATE (switch2_1:Switch {
        name: "Switch 2",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
    })

CREATE (switch3_1:Switch {
        name: "Switch 3",
                type: "PHYSICAL SWITCH",
        system: "PHYSICAL INFRASTRUCTURE"
```

```
    })


CREATE (firewall2:Firewall {
        name: "Internal Firewall 2",
                type: "FIREWALL",
        system: "PHYSICAL FIREWALL"
    })

MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-4_1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-4_2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-4_3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-5_1" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-5_2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Switch 3" AND m.host = "HARDWARE-
SERVER-5_3" CREATE (n)-[:ROUTES]->(m)
```

```
MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-2" CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.type = "PHYSICAL SWITCH" AND m.host =
"HARDWARE-SERVER-3" CREATE (n)-[:ROUTES]->(m)

MATCH (n), (m) WHERE n.name = "Internal Firewall " AND m.name = "Switch 3"
CREATE (n)-[:ROUTES]->(m)
MATCH (n), (m) WHERE n.name = "Internal Firewall 1" AND m.name = "Switch 2"
CREATE (n)-[:ROUTES]->(m)

MATCH (n) where id(n) = 44 DETACH DELETE n
```