View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

VIRTUAL PATCHING: FIGHTING BRUTE FORCE ATTACKS IN A SOFTWARE DEFINED NETWORK ENVIRONMENT

An Undergraduate Research Scholars Thesis

by

BLAKE NATHANIEL NELSON

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Guofei Gu

May 2018

Major: Computer Engineering

TABLE OF CONTENTS

Page

ABSTRACT 1

ACKNOWLEDGMENTS 2

NOMENCLATURE 3

CHAPTER

I. INTRODUCTION 4

Motivation 4
Software Defined Network 5
Virtual Patching 6

II. METHODS 7

Identifying and Mitigating Suspicious Activity 8
Algorithmic Solution 9

III. RESULTS 12

Dictionary Attack 12
Attack from Suspicious Location 16

IV. CONCLUSION 18

Potential Issues 19
Areas for Further Research 20

REFERENCES 21

ABSTRACT

Virtual Patching: Fighting Brute Force Attacks in a Software Defined Network Environment

Blake Nathaniel Nelson
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Guofei Gu
Department of Computer Science and Engineering
Texas A&M University

A new design for virtual patching applications is presented for software defined network environments. Based on OpenFlow implementation, a software defined network can be programmed to intelligently detect threats and handle them accordingly. By implementing a virtual patching solution with the Floodlight OpenFlow API, these networks can detect malicious traffic before it reaches the vulnerable device, based on common signs like packet size or destinations of open but unused ports. A controller hosts an Intrusion Detection Service (IDS) on the network would track signs of malicious data, and scan incoming traffic for any of those signs. If a packet is reasonably suspicious, it is not allowed to continue on it's path, while all other traffic continues as normal. Because software defined networks are inherently programmable, a general solution can be put in place that network administrators can use to create virtual patching rules on the fly. This allows for vast flexibility and efficiency, which is critical when dealing with a live exploitation on the network. Experimental results for both the attack specific solution and the general, programmable solution have not yet been obtained.

ACKNOWLEDGEMENTS

I would like to thank Dr. Gu for allowing me to work within his lab, and learn more about research in the cyber security field. I am grateful for the opportunity to be exposed to many smart minds and topics in this area, and will walk away with plenty of new knowledge to take with me as I pursue the next step in my career as a computer engineer and scientist.

I also thank Lei Xu and the rest of the graduate students in Dr. Gu's SUCCESS lab. Lei provided me with resources and expertise that helped advance my research and explore the avenues that would allow me to succeed. All of the graduate students gave me support and motivation to continue my work and learn more about the topic.

Lastly, I am thankful to my parents who gave me the tools and opportunities to succeed at this great university, as well as love and support as I continued my studies. I also thank the friends that I have surrounded my self with, who have encouraged me throughout my career, but also challenged me to be a better student every day.

NOMENCLATURE

SDN	Software Defined Network
IoT	Internet of Things

CHAPTER I

INTRODUCTION

In modern domestic network environments, Internet of Things devices are becoming increasingly common. Consumers find them a convenient way to automate their daily tasks. However, these devices are commonly a source of security vulnerabilities. They provide another attack vector into the lives of individual consumers, or easy access to large amounts of cumulative processing power. They are able to gain access to these devices through brute force attacks on weak passwords, or default passwords that were never changed. Patching these vulnerabilities can be difficult for the manufacturer, so software defined networks can take on a role in keeping these devices secure and not allowing for things like devastating botnets to be created.

Motivation

Botnets are becoming a problem

In 2016, many home devices such as IP cameras and home routers were found to be infected with a hidden piece of malware, dubbed "Mirai." This malware would work its way through networks, installing itself and waiting for commands to attack. It was used in high profile attacks such as the denial of service attack on Dyn, the popular DNS hosting platform.

Mirai was able to get onto so many IoT devices for a simple reason – many are left on default settings. While there were many clever things about Mirai, it took advantage of the idea that consumers were not conscious about their security. IoT devices are becoming increasingly popular due to convenience to the user. However, hackers will leverage their weak security to create powerful tools intended to hurt business.

Patching is not always easy

Defending these devices or upgrading their security after new discoveries is not as easy as it is with a home computer. Many devices are not able to run anti virus software or any of the other tools that keep a normal desktop computer infection free [8]. These devices may be more critical than a home router as well – internet connected heart monitors or industrial control systems cannot afford the downtime required to update, or the risk of a bug in the new version. With these problems at hand, a simple update or local intrusion detection system is not the right answer.

Software Defined Network

In general, network architecture is divided into layers, each with their own function. Without going into detail about each layer, there are two that are the most relevant to the Software Defined Network model – the network layer, and the transport layer. The network layer "handles the addressing and routing of the the data," while the transport layer deals with the data as packets, and their delivery [5].

A software defined network abstracts the two layers described previously, into a single programmable control layer. Instead of single direction, static rules for the flow of traffic, SDN architecture allows for the dynamic creation and definition of these flow rules. This allows for administrators to make quick responses to necessary changes in the network environment. These could range from new business units to emergency security issues. It also centralizes the network, so that any adjustments from the wide and complex to the remote and device specific can be made convenient [4]. An SDN control server would pick up data from it's data layer, determines it's destination based on the installed flow logic, and passes it to the appropriate

application. A full diagram of an SDN's architecture layers can be seen below in Figure 1, as provided by the Open Network Foundation [4].

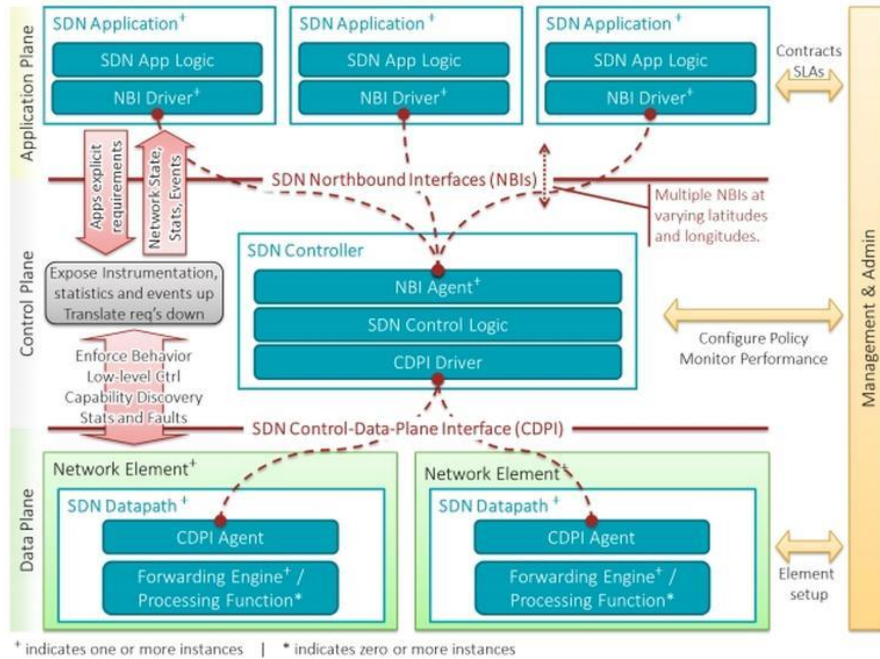

Figure 1: High level diagram of SDN layers

Virtual Patching

Putting it simply, virtual patching is solving the problem, outside of the problem. Since IoT devices are typically hard to patch on the physical system, a solution could be applied to the layer above it, ensuring that anything dangerous does not reach its target. At the network level, rules are put in place to filter out potentially hazardous data. When a virtual patch is applied, the physical system is not changed, however the rules on what that system is allowed to do is changed. Virtual patches would be deployed to prevent applications crashing or losing data due to a bug, while the true software patch is being developed and rolled out. They could also be deployed as a security measure, if a business is concerned about known malicious traffic.

CHAPTER II

METHODS

Botnets can come in a variety of packages, and can find their way onto devices in a number of ways. This paper also does not include an exhaustive list of preventative measures when dealing with securing devices on an SDN. However, it does cover some of the simple and most basic ones, as well as establish the ideology of how to quickly implement any new solutions to new vulnerabilities as they are discovered. Persistent threats will always be at the forefront of cyber security, but with a flexible and general platform to implement defenses, responses can be made faster and data loss will be diminished. This paper focuses mainly on password attacks, which are the easiest and fastest way that hackers are able to compromise a machine.

Continuing with the example given in the introduction, a common vector of attack in IoT devices is their weak passwords. Whether this means leaving the device on the default password, or having something short and computationally easy to guess, a software defined network can be leveraged to mitigate these risks to the consumer. By implementing a virtual patch on an SDN controller, several methods can be designed to make password attacks very hard for the attacker to be successful. The patch would be centrally controlled, but deployed and applied on a wide basis for maximum effectiveness. The main techniques of this specific SDN virtual patch will utilize buffered delays and packet source location checking to determine what is a possible attack.

Identifying and Mitigating Suspicious Activity

Exponential Delay on Login Attempts

Brute force password attacks require high volumes of password attempts on a machine, and when a password is not very complex, a brute force attack can be successful very quickly if it only has to attempt ~10,000 combinations of characters before getting to the result "password" - not a difficult task for a computer with a lot of time on it's hands. However, delaying these password attempts makes attacks take exponentially longer to perform, while still preserving a user friendly experience. With a simple delay of 2N milliseconds per packet, every packet that reaches the SDN controller, the longer the delay becomes, until even a simple password has taken 210,000 milliseconds; impossibly long. Making these login attempts so difficult means that automating these tasks starts to lose it's value, while two or three successive failed attempts by the actual user is negligible in terms of delay. These buffers are also separated into buckets by location, so that a legitimate request from a user isn't stuck behind 10,000 bad requests from an attacker.

Mapping Location to Level of Suspicion

Another defense mechanism is location checking, and this is a little more difficult to navigate. In most cases, a single login request from Russia amid hundreds of prior logins from Kansas would look suspicious. But what if the user was on business in Russia, and wanted to check is online baby monitor because he missed his family? Multiple factors went into the location checking algorithm for this paper. The first is location history. In the previous example, the login attempt from Russia would trigger the SDN controller, but it would not be flagged immediately as a threat. The controller would look at previous attempts, and compare the times

at which the last attempt and suspicious attempt was made. If these were recent (in the last two days), the attempt would be blocked due to reasonable suspicion.

Algorithmic Solution

This paper proposes a unique algorithm that can be deployed in a software defined network, to mitigate the attacks that plague Internet of Things devices and create the botnets that so often disrupt consumer industry. First, any packet received by the SDN controller will be inspected, and determined to be a password request. Every password attempt immediately following a failed password attempt will be automatically deemed as suspicious, and handled using escalating tiers of concern. At the most basic level, each suspicious packet will be intercepted and delayed. In general, each packet will be delayed to it's destination by the following equation:

$$D = c \cdot 2^N \qquad (1)$$

where D is the delay (in milliseconds) that the current packet will be sent, following the time that the last packet was sent (or the time the packet arrived, in the case of the first packet following a failed log in attempt). The exponent N is the number of packets that have been received since the last failed log in attempt. Finally, c is a constant that amplifies the delay of the packet, based on it's level of suspicion, which will be determined further based on factors such as the location the packet was sent from, in comparison to recent packet reception history. For a regular login attempt, the constant would simply be 1, since this may be a normal user, who would be bothered if their third log in attempt after fat-fingering their password twice was stuck for several seconds because of this defense mechanism.

Each packet's source location is important – a log in attempt coming from Eastern Europe doesn't make sense on a machine in the Midwest United States. It would be easy to

automatically mark these as reasonably suspicious and move on. However, humans tend to be more complicated. For example, a father may be traveling to Eastern Europe but want to check in on his baby daughter through his IP camera. On a larger scale, a business person may be traveling to Eastern Europe, but require information stored on a machine back at his office. These examples would create an unsatisfactory user experience at best, and a business disaster at worst.

When dealing with packet source, the first step is storing the MAC address and location that the machine is logged in from on a percentage basis. This allows the SDN controller to understand where to expect a log in request from (such as the users home desktop), or more generally, what machine will be logging in (if the user logs in from their mobile device often). It is likely that these locations and addresses will have heavy percentages (75-90%) on one value, such as their home IP address or their most used machine's MAC address. The time of the last log in will also be stored for comparison. With these in place, when a packet arrives to the SDN controller, the source address and location are compared to the percentage tables. If the location and/or address does not match the table with a reasonable amount of confidence, then the constant c from the above formula would be increased to 5.

A full diagram of the process can be seen below in Figure 2. Each step in the process has valid reasoning. Every log in attempt must be delayed due to brute force attacks that may not fall into the other categories that the algorithm defines as suspicious in the following steps. An attacker may spoof their location or MAC address to mimic that of the user, but still require multiple attempts to guess the correct password. These spoofing brute force attacks would still be mitigated when the magnitude of password attempts are given a default delay.

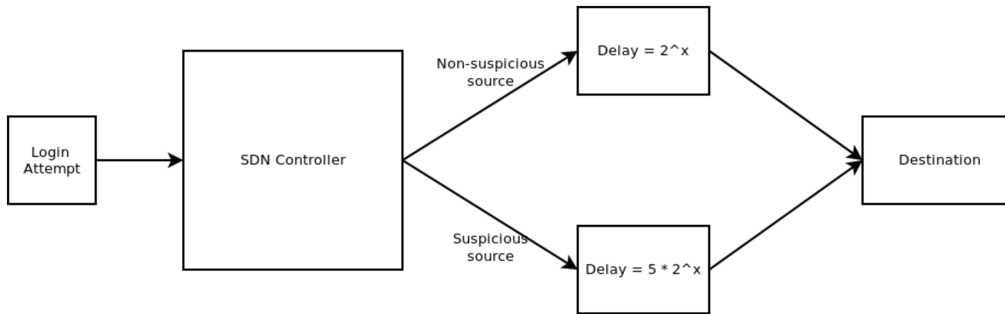

Figure 2: Diagram of the attack delay process

The constant in the function described above allows for multiple levels of security measures to be applied to suspicious packets. Some packets may be less suspicious than others, and the ability to account for the difference in severity allows for increased security without hindering innocent actions by the appropriate user. In this paper, the constant is applied when a packet is received from an unusual location or machine; another indicator that something malicious is occurring.

All together, these steps combine to build an algorithm that mitigates against the brute force attacks that have allowed hackers to gain access to IoT devices and execute their malicious botnets. The algorithm will obtain login frequency and location from all incoming packets and apply a delay formula to them. By exponentially delaying each packet from reaching its destination, a small number of mistakes, such as a normal user might make, will go relatively unhindered, while a serious attack will be rendered computationally infeasible.

CHAPTER III

RESULTS

This paper describes two experiments that demonstrate the successful performance of this algorithm. There are variations and extensions of the brute force attack, such as a rainbow table attack, which uses hash chains to resolve passwords very quickly. These experiments exemplify two different scenarios: i) a regular dictionary attack, where an attacker reads tens of thousands of pre-computed potential character combinations from a file and injects them into a remote login attempt and ii) an attacker with an obviously suspicious location, as determined by the SDN controller, attempts to remotely access an internet connected device.

Dictionary Attack

Implementation and calculations

The attack application is designed as a dictionary attack on a victim's device. A single unsuccessful SSH log in attempt using Python was recorded to take on average 0.42 seconds. The initial attack application was not multithreaded, so each successive login happened consecutively instead of in parallel. Using this design, a dictionary attack of 4000 character combinations took an average of 1683.6 seconds to perform, or 28 minutes. This was the worst case, assuming the correct password was at the end of the dictionary file. By extrapolating this data, a standard dictionary attack of $1 \cdot 10^6$character combinations would then take $4.2 \cdot 10^5$ seconds, or 4.18 days. 4 days is a long time, if this attack is run on $1 \cdot 10^6$ systems at once, then it is well worth the time taken in order to gain control to all of those systems. By applying the algorithm in this paper, each attack is increased by 2^Nmilliseconds. With a shortened character combination set of 12, the total attack time was 4112.87 seconds, or 1.14 hours. While this

character combination is fairly small relative to what a realistic attack would be, it was more realistic to what this experiment would be able to handle. In this case, the constant c is assumed to be 1 for the most basic case. Extrapolating this data once again, for a character combination set of $1 \cdot 10^6$ combinations, the attack would take at least $4.2 \cdot 10^5$ seconds, which is roughly $3.138 \cdot 10^{301022}$ years – computationally infeasible.

Assuming the attacker's application is not multi-threaded is not a safe assumption since adding threads to the program will dramatically increase efficiency, especially as passwords get longer with more characters and possibilities. Therefore, the attack application was rewritten to include multi-threaded capabilities. Using the same character combination set of 4000 from the previous experiment, the attack application was implemented with 30 threads. This shortens the attack time significantly, down to 76.16 seconds, or a mere 1 minute. The attack was repeated with the implemented delay, this time with a character combination set of 360. This was increased so that each thread had 12 requests to perform, similar to the previous experiment. Once again, the total execution time was roughly 4165.34 seconds, or 1.16 hours.

Extrapolating this data to a larger and more realistic system further exemplifies the success of the method. A CentOS Linux system with approximately 94 GB of available RAM has the capability of running over 770,000 processes at once. Assuming 70,000 processes are being performed for regular system services, this leaves 700,000 threads that can be running at once, significantly reducing the total time it takes to attempt a large amount of character combinations. Suppose a password has 8 possible alphabet characters, all lower case. With 700,000 concurrent threads, this would take roughly 298,234 attempts per thread, which at 0.42 seconds per attempt, would take almost 35 hours to compute. While this is almost 3 days, it is still relatively feasible with an attacker with infinite time and resources. However, with the

algorithm applied to each of these threads, the amount of time for all requests goes up to $125296 + 2^{377801998336}$ seconds, which is approximately $2^{377801998328}$, and reasonable to conclude as computationally infeasible.

In practice, this would take even longer than the computations above. Most modern login portals have password requirements when the user creates an account. These passwords must be at least 8 characters but could be larger and include capital letters, numbers, and special characters. Therefore, the 26^8 combinations used in the calculations above would be increased to 26^{41} combinations. This would require up to an infeasible amount of time for even $1{,}000$ systems running $700{,}000$ threads, and would be drastically increased further by applying the algorithm from this paper, as seen in the previous calculations.

Graphs and Figures

The figure below provides a graph comparing the two approaches: a normal brute force dictionary attack and an attack subject to the algorithmic defense described in this paper. The first graph (Figure 3) displays a function of time taken versus number of requests, for 10,000 total requests. The shape of the graph shows that this is a linear function that can be represented as:

$$T = 0.42 \cdot x \tag{2}$$

where T is the total time in seconds that the application has ran, and x is the number of requests that have been performed. The second graph (Figure 4) displays the function of the time taken versus number of requests when the algorithm of this paper has been performed. It should be noted that the x axis of this graph was truncated significantly from the 10,000 data points collected in the previous graph. This is to give a better visualization of the shape of the function, since as the exponential function increases, the shape would become more and more vertical,

thus making the graph difficult to read. This shape can be represented as the exponential function written as:

$$T = 0.42 + 2^x \tag{3}$$

where T is still the total time in seconds that the application has ran, and x is the number of performed login attempts. When comparing the figures and their ranges in proportion to the available domains, it is clear to prove that applying the algorithm to log in attempts significantly increases security in large scale dictionary attacks, while remaining relatively non-intrusive for small numbers of user errors.

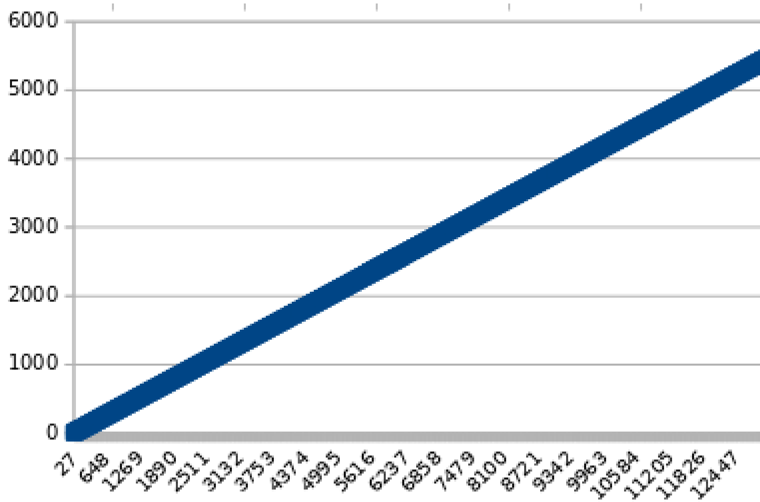

Figure 3: Time versus number of requests for a standard dictionary attack.

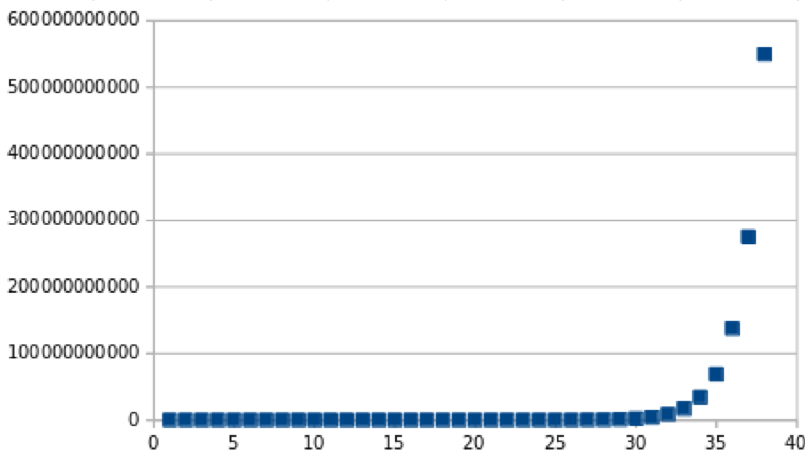

Figure 4: Time versus number of requests for a standard dictionary attack, when the delay algorithm is applied.

Attack from Suspicious Location

Calculations

In an experiment testing packets from a suspicious location, the attack was designed as a mutli-threaded application, sending packets from a foreign location. It used a character combination set of 360, with 30 threads. With the constant now included in the delay calculation, the time of execution for a dictionary attack from an unusual or suspicious location was 20646.28 seconds, or 5.735 hours. Notice that this is much longer than an attack that is not delayed, with a larger character combination set.

Once again, this data is extrapolated so that a more real world case can be defined. This situation also assumes the same multi-threaded system as the previous experiment, where 700,000 threads would be run together. The password in question is also assumed to be 8 characters long, all lower case. As computed in the previous experiment, this type of system would be able to guess the correct password to SSH into a machine in approximately 35 hours. However, in this experiment, the attacker is assumed to have a reasonably suspicious location or MAC address, based on the comparative frequency of the current values versus values received in the past. Therefore, when applying the delaying algorithm, the constant c has been changed from 1 to 5, as described in the Methods section of this paper. While this value is active, the amplitude of the delay is heavily magnified. Because the level of suspicion is increased by a mysterious, foreign address, the delay must be increased to meet that. Using numbers from the previous experiment, the time delay would be amplified to $5 \cdot 2^{377801998328}$. While this amplification would not alter the overall conclusion much – the function was computationally infeasible to begin with – even earlier login attempts would be significantly increased in response time. In the event that the attacker is lucky enough to have the correct password early in

it’s dictionary, it would have to be in the first fifth of the dictionary to take less time than a non-suspicious location attacker. Even then, the password must be found much earlier than that in order for it to be considered computationally feasible.

Graphs and Figures

Borrowing again from the previous experiment, a non-suspicious, non-delayed dictionary attack could be expressed as a linear function, the equation and graph of which can be seen in the previous subsection, as well as Figure 3. However, the graph of the time delay versus time when the algorithm is applied with the suspicious location coefficient set, and the graph is slightly modified from the graph in Figure 4. Seen below in Figure 5, the shape of the graph is very similar to it’s counterpart in the previous subsection. However, one would notice the range of data is heavily scaled in comparison. The exponential delay is vertically stretched thanks to the suspicious location coefficient. This function can be expressed by the following equation:

$$T = 0.42 + 5 \cdot 2^x \tag{4}$$

where T is still the time delay, and x is still the number of requests. This equation is similar to (3), except the constant c from equation (1) is now set to 5 for the suspicious address.

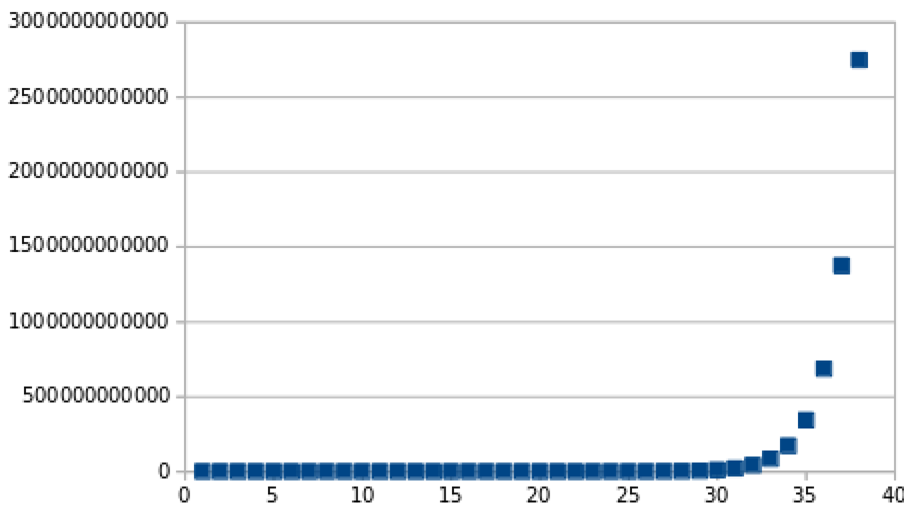

Figure 5: Time versus number of login attempts, from a suspicious location

CHAPTER IV

CONCLUSION

Explicitly, the evidence given by the calculations, graphs, and experimental measurements concludes that this algorithm applied on a software defined network will exponentially delay log in attempts on a machine following the first unsuccessful attempt. This is especially effective against a large number of attempts, which make it a successful attempt in defending against brute force dictionary attacks that plague IoT devices and lead to the creation of harmful botnets. The table below (Table 1) organizes the results found in this paper. The dictionary attack was applied in the three different ways described at the top, and the total execution time in seconds can be seen in each column. By comparing these three results, it is clear that this method is a successful defense against targeted dictionary password attacks. Extrapolating this data to larger, more realistic data sets shows that it would only become more successful with the more possibilities.

Table 1: Comparative results

Time to execute (seconds)

Attack	Control Set	Delayed	Unusual location
Dictionary Attack	76.16	4165.34	20646.28

A more broad conclusion could be stated about the success of virtual patching on a software defined network. If a defense such as this buffered delay can be applied over a large network of internet connected devices in critical areas such as a business or hospital, then so could a solution to any new security issue that is discovered. A wide range of systems that

currently have to battle between the down sides of upgrading their systems or the risk of infection, will now have a fast and simple solution. This is also critical because cyber defense is a moving target; there are always new vulnerabilities being discovered, and new ways to protect against them. If a virtual patch can be put in place before the entire system is updated, then the system can stay secure without risking down time for an upgrade. Overall, the flexibility of a software defined network is it's greatest strength in the world of cyber defense.

A virtual patch on a software defined network is a better solution than current intrusion detection systems or firewall implementations. Virtual patching draws similarities to firewalls; however a virtual patch is much smarter. It does not simply drop packets that meet a level of suspicion, but instead allows for escalating levels of suspicion and defense. It is also more flexible than legacy solutions. Rules can be added or altered to adapt to changing defense requirements, or new hosts being added to the infrastructure.

Potential Issues

Some of these changes leave the possibility for the user to be negatively affected. Their log in attempt may be throttled heavily if occurring during a massive brute force attack by a foreign agent. In this case, a recommended solution would be to notify the user that a potential attack is taking place. The attacker would not see this information, or otherwise wouldn't care. However, it at least makes the user aware that something dangerous is going on, and leaves them the option to be patient, or take necessary action with their device.

The location frequency table could be abused by a foreign attack, if the packets from a foreign location contribute to the location table. This could quickly alter the table to making the most common location somewhere far away from the device. In this case, two solutions could be implemented. The first is to not allow high packet volume to contribute to the frequency table.

However, this could raise issues with user convenience in the event a person spends an extended amount of time from their device. The other solution is to add another level to the location coefficient, which accounts for the distance of the log in attempt source location from the device itself. This would allow more flexibility on the magnitude of the delay.

Areas for Further Research

An argument could be made for further research to be done on weighting foreign locations. For example, Russia might be considered more dangerous than Dallas to somebody in Oklahoma City. A user is more likely to give their credentials to a family member a state or two away then they are the other side of the globe. This could also be reflected in the table of common locations. A threshold could be applied to the location frequency, and if a location far away from the device and below the threshold makes a login attempt, then the delay could be applied.

The algorithm could be extended to completely block default password attempts - “password,” “root,” and any other common default password, without knowing what the real default password is – from any location outside of the recognized locations. This would rely completely on the location frequency table, and could apply a threshold as well. If a location does not constitute above 98% of the usual access attempts, then it is blocked automatically. This could also be vulnerable to the same hijacking discussed in the potential issues section above, so implementations of this extension should have those solutions in mind.

REFERENCES

[1] Burnett, Mark. “Blocking Brute Force Attacks.” *System Administration Database*, UVA Computer Science, 2007. http://www.cs.virginia.edu/~csadmin/gen_support /brute_force.php

[2] Izard, Ryan. “How to Process a Packet-In Message.” *Project Floodlight*, Atlassian, 12 Jul. 2016, https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/9142279/How+to+Process+a+Packet-In+Message

[3] Izard, Ryan. “How to Write a Module.” *Project Floodlight*, Atlassian, 26 Apr 2016, https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343513/How+to+Write+a+Module.

[4] Kassner, Michael. “How software-defined networking will benefit IT and organizations.” Tech Republic, 19 May 2014, https://www.techrepublic.com/article/how-software-defined-networking-implementations-will-benefit-it-and-organizations/.

[5] Rouse, Margaret. “OSI reference model (Open Systems Interconnection.” Search Networking, TechTarget, Aug. 2014, https://searchnetworking.techtarget.com/definition/OSI.

[6] Shin, Seugwon et al. FRESCO: Modular Composable Security Services for Software-Defined Networks. *The Network and Distributed System Security Symposium,* 23 April 2013, San Diego, California. 2017. Print.

[7] Wang, Helen J. et al. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. *Special Interest Group on Data Communication*, 2004, Portland, Oregon. 2017. Print.

[8] Zou, Xu, “IoT devices are hard to patch: Here’s why – and how to deal with security.” Tech Beacon, https://techbeacon.com/iot-devices-are-hard-patch-heres-why-how-deal-security.