THE TRIPLE POT AND TECHNIQUES IN DISTRIBUTED SYSTEM CALL INTRUSION DETECTION

BY

JONATHAN MING-GUY CHU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

ABSTRACT

In cyber security, engineers need to devise ways to protect their systems from hackers. One of the ways that they do this is through intrusion detection. Host based intrusion detection systems reside on the computer and perform internal diagnostics of a computer to detect malware and misuse. These HIDS use a variety of methods to detect and prevent attacks such as file integrity verification, log monitoring, file access patterns and etc. In this thesis, we look at the method of analyzing system calls for anomalous behavior.

Programs use system calls to gain access to functions from an operating systems kernel. Therefore, it is theoretically possible to detect when a hacker may be exploiting a program by analyzing system call patterns of an application. However, despite previous work in this area, there remain many challenges to accurately detecting malicious exploits and intruders through system call analysis which have prevented it from being used in real systems.

To help bridge the gap and address the challenges in making system call analysis a reality, we introduce a new method of system call analysis that we call the Triple Pot method. Our method utilizes three computers running concurrently on the same network to check for anomalous behavior of an application. The key idea is that by setting up a staged, fake network of computers we can get the hacker to identify their exploit for us. We will show how our method can be used to automatically identify zero day attacks that could not previously have been detected using previous system call analysis methods.

In addition, we also introduce a method to aggregate and analyze system calls from distributed machines to use information from multiple computers to detect zero day attacks. We do this by creating a probabilistic model of the networked computer systems to determine the likelihood that an application is exhibiting anomalous behavior that is caused by a malicious hacker. Our methods can accurately locate malicious behavior with low false positives.

To my parents and brothers, for teaching me that impossible is nothing.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Roy Campbell for his support and mentorship during my graduate studies. Also, I would like to thank Alexander Hadiwijaya and Aaron Kawer for their helpful work and support on the projects. Moreover, I would like to thank the other members of the Systems Research Group, such as Mirko Montanari, Abhishek Verma, Mayank Pundir, Alejandro Guiterrez, and Reza Farivar for their advice and wisdom. Finally, I would like to thank my family and friends for their continuing support of my endeavours. My graduate studies have been supported through funding from TCIPG and Boeing.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Today's hackers continually explore and develop new methods to perform cyber attacks on systems. With the rise of the personal computer in almost every home in the world, cyber attacks can be done by anyone from script kiddies to large rival enterprise corporations. An entire economy has been developed on the black market in which people may discover and sell new computer exploits to the highest bidder[1]. Bidders for cyber warfare technology encourage increased cyber activity that has the potential to be good or bad. Corporations may even try to procure these exploits to glean data from rival competitors which helps them obtain corporate strategies and technical products.

A cyber attack usually occurs in two stages: exploit stage and payload stage. An exploit in an application may incorporate the use of a backdoor password, a buffer overflow attack, a SQL injection attack, and many other methods. The exploit is used by hackers to attempt to gain access to a remote computer. If the exploit stage is successful, a hacker may then move onto the payload stage. In this stage, hackers take advantage of the access given to them by the exploit to try to steal information from the computer or damage the system for the user. An example would be the Meterpreter module on the Metasploit software, which allows people to send a variety of commands to a remote computer to do malicious activity.

One of the most dangerous threats to corporations is the Advanced Persistent Threat[2]. This is a threat where an entity continually targets a specific corporation for cyber hacking. An example may be a rival corporation trying to learn secrets from its competitor. This threat is different from more common cyber threats because most cyber threats do not target a specific individual, but rather spread across the web trying to acquire whatever it can get such as botnets trying to harness cpu power. APT is

different because it is done by some competitive rival trying to gain an unfair advantage. These are the types of entities that employ zero day attack methods so that they can undermine their opponent's defenses with an unknown exploit. The vast majority of cyber defense is signature based. Companies are dependent on attacks being identified and then pushed into the database for detection. However, entities that try to persistently attack another entity will want to evade detection from signature scanners. Thus, they will use zero day attacks that nobody knows about yet to try to get into their enemies systems. These are the attacks that our system tries to counter against.

Most exploits do happen at the application layer, either through some sort of a buffer overflow attack or some other unusual program exploit. Consequentially, traces of the attack can be recorded through the system calls that the application sends to the operating system kernel. Applications make heavy use of system calls and thus we can build a profile of an application and record the system calls used in everyday operation of the system. A profile of an application consists of the normal, safe operations that an app can perform on a regular basis.

The system that we propose to use to analyze system calls is a distributed system meant to simulate the conditions of a real enterprise server running on a network. These servers are used to communicate with the rest of the internet and are assumed to have network connectivity. We call our model, The Triple Pot model because it involves the simultaneous use of three computers that may be either virtual or physical machines. For our purposes, we use three virtual machines to simulate the three pots. One pot is the real server doing real work with the outside world, while the other two pots are there to record suspicious activity for security purposes. By creating three different systems, we can better identify suspicious system call behavior on the applications running on the machines. We do penetration testing on our system model by throwing exploits at it to test its detection performance. We find that our system is able to reliably detect new attacks with low false positives, making it more practical for real systems than previous approaches that only analyzed system calls in isolation to one machine. Even without a database known signatures, our system is able to identify a malicious attack as it occurs without prior knowledge of how the attack works.

Furthermore, our method does this while preserving the privacy of each machine activities.

Secondly, we also look at the case when multiple computers may collaborate to share security details. We notice that for the case of advanced persistent threats, computers on the same network are likely to be compromised together. Moreover, computers on a completely different network are unlikely to be compromised as the targeted entity. This is because attackers do not want to draw too much attention to their exploit because they want to be able to use it to only target specific entities without being discovered. Therefore, networks that are not on the hit list are unlikely to exhibit the same anomalous system call patterns that a particular zero day exploit may cause. As such, we develop a probabilistic model to calculate the likelihood that a computer has been infected with a zero day exploit by examining its system calls and the system calls of other computers running the same application.

CHAPTER 2

RELATED WORK

Host based intrusion detection has been a field with potential that hasn't quite yet bridged the gulf between research and practice. It is far more common to see companies in industry install network based intrusion detection systems or virus scanners instead. However, despite HIDS' status as a niche application in the field of computer security, there has been progress in research for developing ways to have practical systems employ HIDS monitoring.

2.1 System Log Approach to Host-Based Intrusion Detection

Many of the proposed approaches to HIDS involve monitoring things like CPU usage, system logs, access patterns. For example, Lichdodzijewski et al looked at using self-organizing maps to detect anomalous logins. SOM is an unsupervised learning technique for data analysis that uses artificial neural networks[3]. In another instance Kola et al explored the use of SOM for analyzing cpu usage and disk usage[4]. They claimed that SOM was more stable than clustering algorithms like k-means because it required less predefined user variables that could change greatly depending on the situation. In these cases, they looked at the security of linux operating systems.

For Windows, researchers like Topallar et al looked at using SOM to analyze patterns in windows registry access[5]. In addition Stolfo et all explored the effectiveness of two different anomaly detection algorithms PAD and SVM on Windows registry behavior[6]. They claimed that the PAD method could outperform traditional SVM methods at a lower complexity cost.

People have also looked at the Basic Security Model audit records which used to be created by Sun. These BSM records provided improved security features over standard UNIX operating systems. Koral et al explored the use of state transition analysis to record and detect penetrations by modeling them as a series of state changes from an initial secure state to a target compromised state[7]. They claimed that their method was more flexible in detecting penetration attacks by considering how actions on the computer change the state rather than just looking for duplicate actions like in signature scanning based methods. Peng et al used similar techniques to create workflow diagrams to identify signatures of attacks[8]. Later on, Anup et al also examined how to monitor BSM logs for anomalies and they proposed building a profile of the system using back propagation neural networks or elman networks to learn the past behavior[9]. If a future behavior deviated too much from the past, it would be considered potentially malicious.

The issue with these prior methods is that they mainly work on monitoring systems at an abstract level. They view the logs of the system which do not contain fine grained application activity. Also, they cannot isolate how or which application was exploited. Instead these methods mainly look at detecting computers that are being misused after exploitation. They do not detect the exploit itself. Instead they take a more lightweight approach and check logs which only get updated infrequently. Security professionals usually want to know how their computer was hacked and not just that it was hacked. By the time these methods detect malicious behavior, the hackers would have already done their damage by moving files or corrupting the computer. In some of these papers, researchers stated that they wanted to keep the workload light so that the CPU would not be strained running a security program in the background. However, we are interested in enterprise level security and believe that computers are starting to get faster with more and more cores. We believe that increased processor performance can lead to more insightful security applications.

Our goal is to expand the scope of the monitoring system and scan the actual application activity. More importantly, we want to be able to detect an exploit as soon as possible, so the security monitor can be notified of any possible intrusions that take place to mitigate the damage before anything happens.

2.2 System Call Approach to Host-Based Intrusion Detection

System calls are used by applications to interact with the operating system and the files on it. They may be used to open up a file or send data through a port. By monitoring system calls, one can obtain a fine grained model of application activity.

Hofmeyr, Forrest, and Somayaji looked at detecting intrusions through system call analysis[10]. To do this, they calculated the hamming distance on system call commands and used a sliding window approach to gather short sequences of system calls. They build up a profile of normal behavior from the program of interest and then measure the “Hamming distance” from a new trace to the traces in the profile. If a trace is too different than it is considered anomalous and potentially malicious. Gaurav also looked at using machine learning to analyze system calls. He used not just the system call method, but the argument to the method when logging the activity of the application[11].

Liu et al also explored system call based intrusion detection by looking at it with protocol context[12]. They combined both the control flow of the program activity with the data flow of the program. The control flow refers to the system calls while the data refers to the system arguments. They call their approach, the semantic flow model, and they tried to create unique traces of a program that they called semantic units where a semantic unit describes one aspect of a program’s behavior.

Gideon and Hu similarly investigated using the semantic approach for using contiguous and discontiguous system call patterns[13]. They analyzed their traces using an extreme learning machine to create an artificial neural network to detect anomalous traces.

Sekar et al proposed to use a finite-state automation mechanism to detect malicious behavior from applications[14]. They believed that their method avoided the more machine learning heavy computations of other approaches by using an efficient FSA method that required less space and constant time per system call during the learning and detection phase.

Most of these methods assumed a window size of size 6 to handle the straces, but this may not be optimal. Eskin, Lee, and Stolfo explored using entropy based methods to find optimal window sizes to gather system call sequences [15].

Wagner and Soto looked at how to counter system call analysis by developing what they call a mimicry attack which takes advantage of the fact that some of the proposed system call analysis relies heavily on matching sequences of calls[16]. They noticed that it was possible to create a program which mimicked the actual system calls of an application and thus mask their malicious app.

So far, most of the papers on system call analysis that we surveyed used a single computer to perform the analysis. They do not take into account how multiple computers may be used to enhance anomaly detection. Our approach is different in that we leverage multiple computers to enhance anomaly detection in a way that a single computer could not possibly do. While there has been some previous work in dealing with distributed network systems, they mostly dealt with network data[17][18][19]. They used peer to peer systems to aggregate data and applied tactics such as ant colony optimization algorithms or information theoretic approaches to crunch the data. Our methods instead look at the fine grained system calls of each application to determine how similar or dissimilar they are with applications running on other machines. We believe that utilizing a distributed system where computers share specific information with each other can improve detection of anomalous behavior and reduce false positives in a manner not possible with standard single computer HIDS.

CHAPTER 3

SYSTEM CALL TRACING OVERVIEW

Applications may use system calls from everything from reading the time to opening a file. In addition processes may spawn child processes that also have system calls. When someone tries to exploit an application to create a new terminal, that new terminal is still considered a child process of the original application, which means we can still capture a malicious hacker's activity and trace it back to its source. We use the strace program found in Ubuntu to capture system call traces.

Previous research on system calls used metrics such as Hamming distance or SVM to compare traces. In our method we use Levenschtein distance which takes into account the edit operations of insertion, deletion, and substitution. We implement it using the Wagner-Fischer method which is a dynamic programming algorithm. With this method, we are able to calculate the edit distance between sequences of system calls. We define a sequence of systems calls to be 7 consecutive system calls. We also define a trace to be multiple consecutive sequences that may represent an action such an exploit. In our experiments we capture the system calls by using the program id of the application. We also capture any child processes that are spawned from the original process.

Many of the previous system call metrics tried to detect anomalies by believing that traces from exploits were significantly different than regular traces. They thought that there would be a big edit distance difference. This is why machine learning approaches were used to try to detect the relationship between system calls to determine the likelihood that a sequence was malicious or not.

However, we believe that this assumption is not valid because hackers can develop exploits that are very similar to normal sequences of system calls. An exploit

does not need to have any special properties for a malicious hacker to use. Therefore, any supervised learning approach to detect exploits may not be very effective as there is little in common between exploits of different applications. There are so many ways an exploit can be created, that it is not really feasible to be able to read a sequence and identify it as a malicious with no other data to support it. Also, if there are about 100 possible potential system call commands and seven spots for a system call, then there are over trillions of possible system call sequences. This means that different applications often have very different profiles from one another. One cannot train on one application and then use that behavior to extrapolate results from other applications. This creates problems in that every application needs to build their own behavioral profile. Currently, application developers do not provide these behavioral profiles themselves. This is one major reason, why system call anomaly detection methods have not seen wide use in the industry yet. Much of what we will see later on is how to leverage additional data besides just the sequence of system calls to determine whether a trace is malicious or not.

The methods we explore in this paper are based more on creating dictionaries of sequences and then finding matches among different dictionaries to leverage the additional information of multiple computers.

CHAPTER 4

THE TRIPLE POT METHOD

4.1 Assumptions and Design Methodology

The Triple Pot method utilizes three computers that are on the same network. It makes certain assumptions about the scenario in which this method is utilized. These assumptions are built off the premise that we are interested in the situation where an enterprise server that communicates online with users may be subject to an Advanced Persistent Threat from another entity. Such an adversary would probably be considered to have a high level of resources and professional aptitude. As such, they would be utilizing a zero day exploit that they either purchase on the black market or develop in house.

For our scenario, we do not handle the case of the more common hacker who would utilize cybercrime exploits that are well known. Hackers such as script kiddies usually take exploits from common malware libraries and attempt to hack into outdated computers that do not have patches to protect against such exploits. We assume that our enterprise level server has the requisite patches against known attacks and also has a scanner which can detect exploits from a database of previous exploits.

Instead, we focus on identifying zero-day attacks that use exploits that nobody is aware of yet. It is believed that these types of exploits may be used for months to penetrate computer defenses before anybody becomes aware. For corporations, these zero-day attacks are the toughest to guard against as most malware detection relies on scanning for signatures of known attacks. Since a zero-day attack is not known, traditional cyber defenses cannot reliably protect against these exploits.

Host-based intrusion detection systems that utilize novel anomaly based detection methods may have a chance of detecting when an attack is occurring. However, their

usefulness in real life situations has been hampered for a variety of reasons. First, these systems usually rely on a robust and complete profile of the application's behavior to be built. These profiles are needed to determine when an application's new behavior does not match the past history of the application. A major issue with this is that it is extremely difficult to generate these profiles for an application because an application may have millions of different paths that it could take. All one can really hope for is a profile that covers as many paths as possible, but it will usually be difficult to cover all the paths. As such, it is not practically feasible for prior methods that utilize application profiles as baselines to distinguish anomalous behavior to be effective. These methods to detect anomalies suffer from the problem of proclaiming all new activity to be malicious.

Our system is different in that while it still uses a behavioral profile of the application to detect anomalies, it has an improved mechanism for distinguishing malicious anomalous behavior from benign anomalous behavior. By harnessing 2 other computers running the same application, we can examine their behavior to determine if our own computer is running malicious software. This is possible because we take advantage of our knowledge on the attack patterns of malicious hackers. Frankly, we rely on the key assumption that an adversary acts with little knowledge of the internal networking of an organization's servers and the organization of the data within the organization. As such, they will adopt a strategy of port scanning all computers on the internal network that they can reach and then executing an attack on ones that have the specified weakness.

Assumption: Adversary scans network and attempts to attack all vulnerable computers

By setting up two additional fake computers next to our real computer, we can set up a trap that can give us additional information about hacks from competitors. Instead of one instance of the exploit, we will have 2 instances of the exploit and 2 sequences of illegitimate sequences of system calls to inspect. Since we already know for sure that the fake computers aren't supposed to be running any new behaviors because we set them up ourselves, we know that any anomalous behavior from our

fake computer must be from an outside malicious entity. By comparing traces from the fake computer to the real computer, we can tell if a sequence of system calls is malicious or not. Anomalous malicious system calls would be the same on both the real computer and the fake computer. Anomalous benign system calls would appear on the real computer, but not the fake computer.

The reason why we utilize a third computer is to establish a third computer that doesn't have access to the network to be remotely exploited. This computer could not possibly display any malicious behavior because it is not even connected. We assume that the hacker does not have physical access to the servers and is trying to attack remotely over the network. The role of the third computer is to act as a profile of the normal background processes that an application may execute. Applications may have background processes running that may be deemed unusual. Furthermore, many applications have built in schedules to perform some action at a certain time. This may be behavior such as a server defragmenting its hard drive or cleaning up memory space. It may also be periodically sending out notifications of system events. We have also seen applications constantly attempting to poll network queues for data. However, if our third computer only creates benign processes, then we have an additional dictionary of safe sequences to compare to the other sequences on the other machines. If a sequence from the other real machines matches a sequence from the dictionary created from the benign machine, then we know to classify that sequence as safe.

In our model, we have three computers and one monitoring center. The first computer is called the ACC(Active Computer Connected to network) and this computer is the real computer being monitored that does actual processing for the enterprise. The second computer is called the PCC(Passive Computer Connected) and this computer is a fake computer meant to serve as a trap to catch malicious behavior. The third computer is called the PCD(Passive Computer Disconnected) and this computer is a fake computer that is not connected to the internet. It is meant to check processes from the ACC and PCC to classify healthy sequences. These three computers, ACC, PCC, and PCD are connected to a monitoring center that can collect information from the three computers to make a decision on whether the active computer has a malicious sequence or not. By creating a third monitoring center, we can offload some of the

heavy processing from the ACC to another computer to make the CPU processing more efficient on the computer of interest.

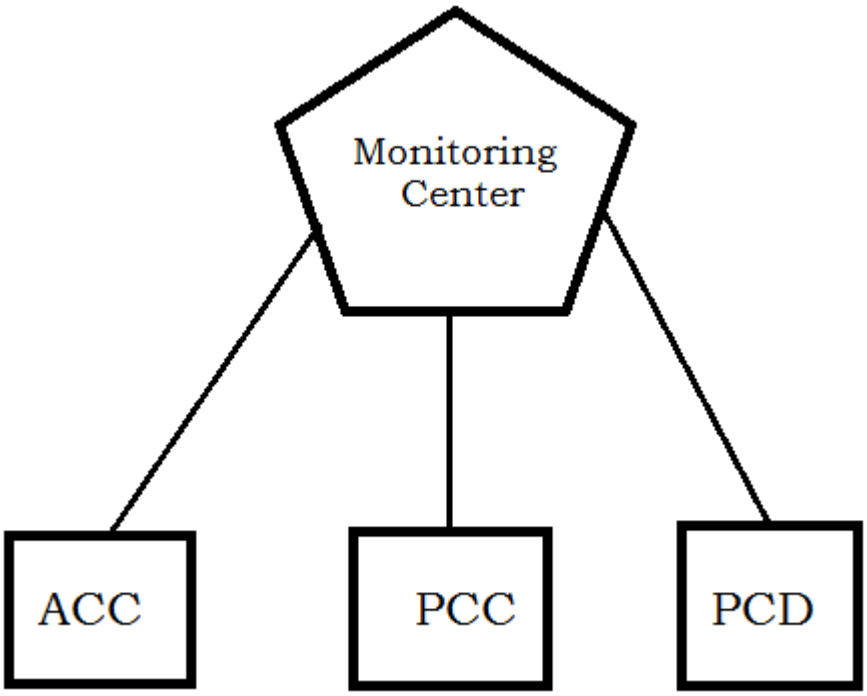

Figure 4.1: Triple Pot Monitoring

4.2 Algorithm

The algorithm to that we utilize to combine system calls from the ACC, PCC, and PCD is efficient. First, it involves creating five types of logs. These logs are used to compare and contrast different traces found in the Triple Pot system. These logs are system call traces from Ubuntu that we acquired by setting up virtual machines. The system calls can be recorded using the strace application.

The first log is the normal behavioral profile of the ACC. This log can be built from running the system in an isolated and safe network and then recording system call traces of the computer. This is considered to be the safe log and any future traces that match up with traces in the normal behavioral profile are considered to be safe. We can

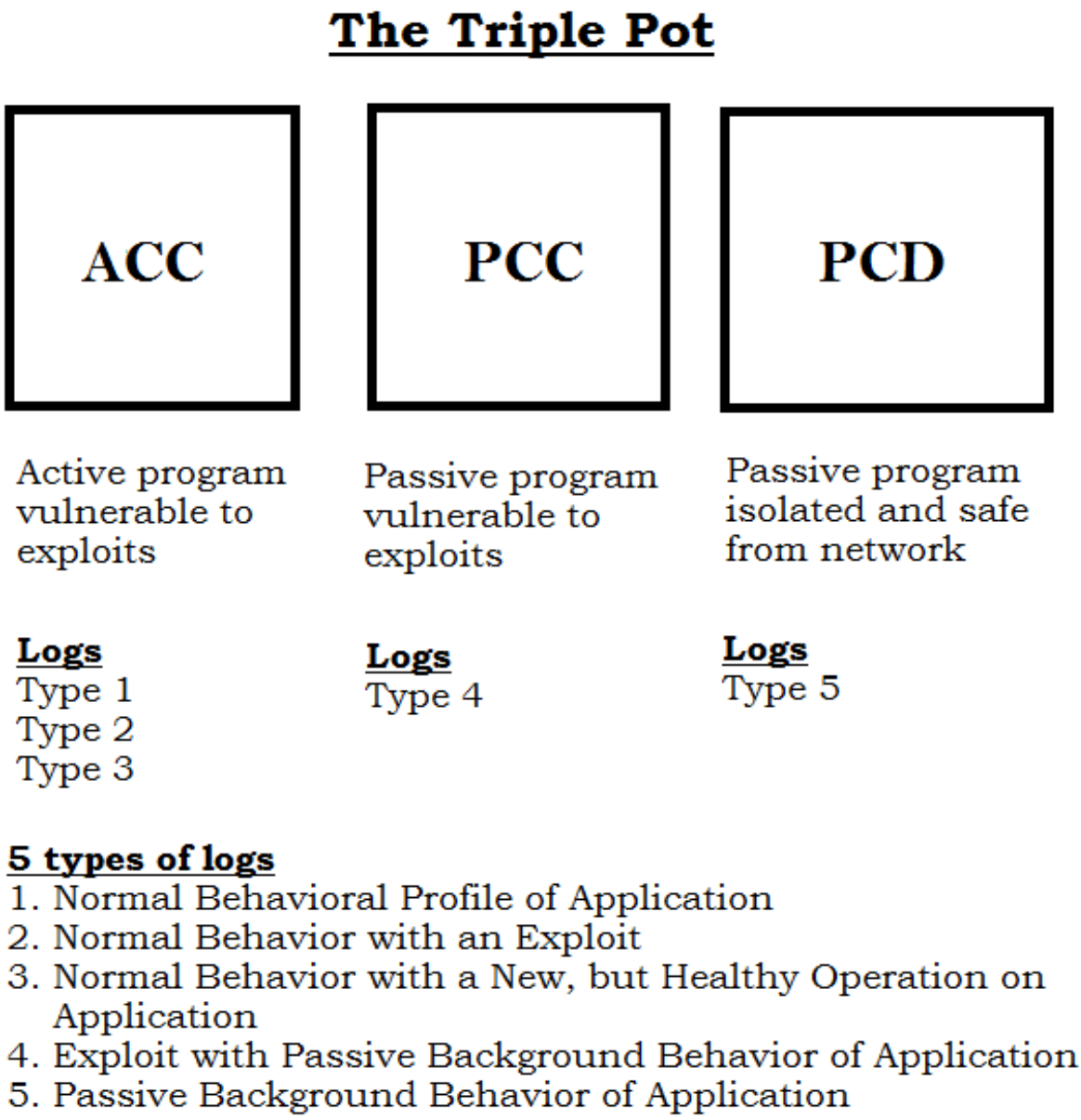

Figure 4.2: The Triple Pot

create what we call a dictionary of safe traces from this log by using a sliding window of size W. This size W can be modified or changed depending on the circumstances. For our purposes, we use a window size of 7.

The next log that we will discuss is the Type 4 log. This log is of the system call traces of the PCC. The system calls on the PCC should be almost nonexistent because it is a fake computer not running any actual user activity. If someone from the outside tries to access it, then we assume that they are a malicious hacker trying to get into any

computer it can find. If we capture any suspicious traces from Type 4 logs, we can compare it to the logs from the ACC to see if there are any matches, in which case we can flag those as malicious too.

The Type 5 log is generated from the PCD, and it is a log of all the background processes that may spawn from an application. This is to take into account any periodic updating that a system may try to do. It also tells us what processes should be running on the PCC that are not normally run in the background.

For testing purposes, we also have a Type 2 and Type 3 which are logs of the PCC in normal use. The Type 2 logs contain exploits of the application under consideration. The Type 3 logs contain the operation under normal use, but also having new operations not featured in the profile. This is to show how our system can distinguish anomalous traces that come from exploits and anomalous traces that are just new behaviors not featured in the normal behavioral profile.

Our algorithm is divided into 3 main stages P1, P2, and P3. P1 can be done entirely in the ACC. P2 and P3 can be done by the central monitoring center. In these stages, we divide an strace of system logs by using a sliding window of size 7. So each sequence is made up of 7 system calls. Also, since we cannot know when a sequence for a command begins or ends, we create a dictionary of all possible sequences. So the window slides by 1 every time, instead of by 7. In this way if there are 70 system calls, a dictionary can be formed with size 63 of the total possible patterns, which is N(num of system calls) – W(window size).

In P1, we find anomalous traces of system calls on the active computer that are not found in the profile dictionary. In P2, we find anomalous traces of system calls on the fake computer that are not found in the profile dictionary or background processes dictionary. In P3, we combine the results of P1 and P2 to see if there are any matches between them. If there are a high number of matches then we would consider there to be an exploit.

To help detect Mimicry attacks, we would also check in P2, whether there are an unusually high number of anomalous traces in the PCC that are not in the background processes, but are matched up with the profile of the normal behavior of the ACC. These traces would be considered suspicious even if they match up with the normal

behavior of the ACC because the PCC shouldn't have any behavior running on it at all because it is a fake dummy computer. Only a hacker would be running a normal process on it. Mimicry attacks are when a hacker tries to imitate the normal system call behavior of an app while still performing malicious acts. Our system can thus detect Mimicry attacks in a way that previous systems could not.

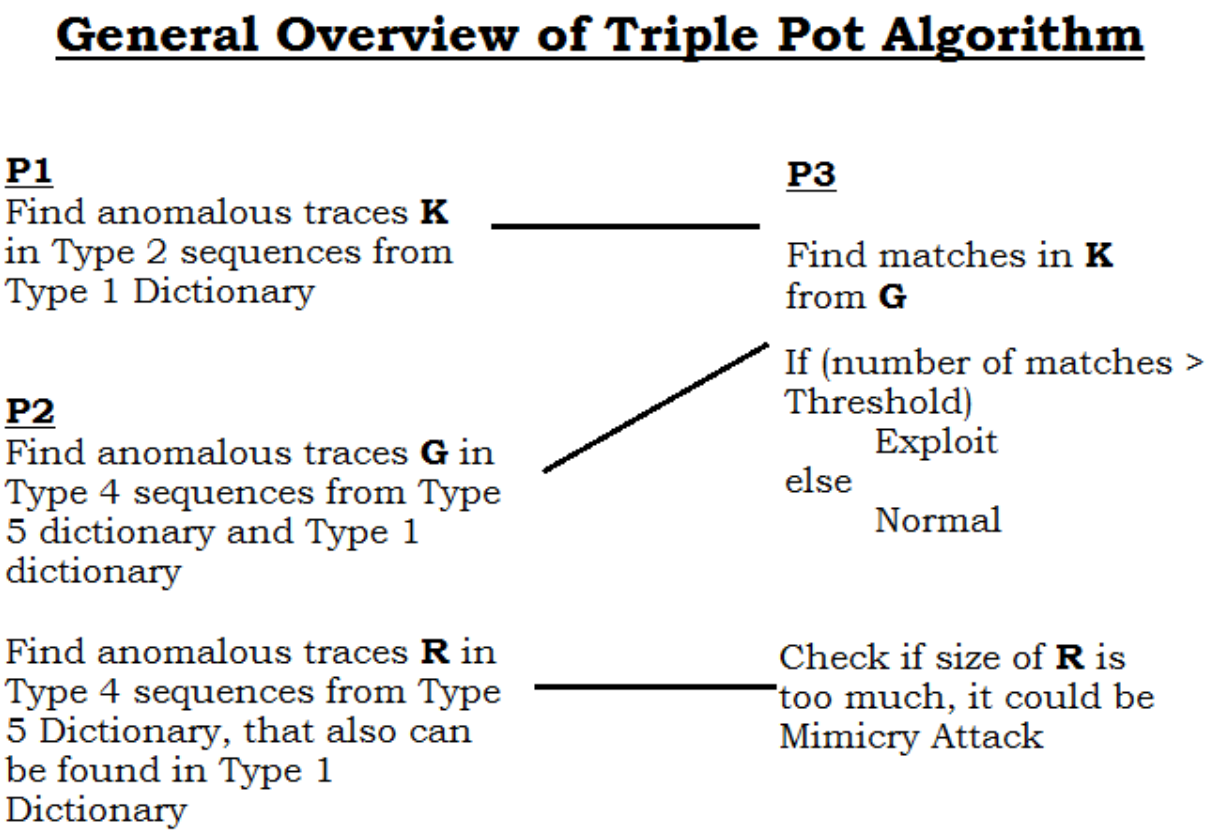

Figure 4.3: Triple Pot Algorithm

P1 ALGORITHM PSEUDOCODE

```
static ArrayList<String> P1(String file1, String file2){
            try {
                  int threshold = 1;

                  ArrayList<String> lines = lines from Type 1 log;

                  //create dictionary of patterns for Type 1 log
                  ArrayList<String> patterns = new ArrayList<String>();
                  HashSet<String> unset = new HashSet<String>();
                  Sequence curp

                  for(int i=0; i < lines.size()-wsize; i++){
                        for(int j=i; j < i+ wsize; j++){
                              curp.add(lines.get(j));
                        }
                        if(unset.add(curp)){
                              patterns.add(curp);
                        }
                        curp.clear();
                  }

                  HashMap<String, Integer> tc = new HashMap<String,
Integer>();
                  ArrayList<String> lines2 = lines from Type 2 log;

                  //create list of patterns of Type 2 log
                  ArrayList<String> patterns2 = new ArrayList<String>();

                  for(int i=0; i < lines2.size()-wsize; i = i + wsize){
                        for(int j=i; j < i+ wsize; j++){
                              curp.add(lines.get(j));
                        }
                        patterns2.add(curp);
                        curp.clear()
                  }

                  int min = 0;
                  ArrayList<String> p1 = new ArrayList<String>();

               for(int i=0; i < patterns2.size(); i++){
                  String[] foo = patterns2.get(i);
                  min = 1000;
                  for(int j =0; j < patterns.size(); j++){
                        String[] bar = patterns.get(j);
                        int ret = EditDistance(foo, bar);
                        if(ret < min){
                              min = ret;
                        }
                  }

                  if(min >= threshold){
```

```
                        p1.add(patterns2.get(i));
                    }
                }

                return p1;
            }
            catch(Exception e){
                e.printStackTrace();
            }

            return null;
      }
```

P2 ALGORITHM PSEUDOCODE

```
static ArrayList<String> t4vt5t1(String file4, String file5,String file1){
            try {
                  int threshold = 1;

                  ArrayList<String> lines = lines from Type 5 log

                  //create dictionary of patterns for Type 5
                  ArrayList<String> patterns5 = new ArrayList<String>();
                  HashSet<String> unset = new HashSet<String>();

                  for(int i=0; i < lines.size()-wsize; i++){
                        for(int j=i; j < i+ wsize; j++){
                              curp.add(lines.get(j));
                        }

                        if(unset.add(curp)){
                              patterns5.add(curp);
                        }
                        curp.clear();
                  }

                  ArrayList<String> lines2 = lines from Type 4 log

                  //create dictionary of patterns for Type 4
                  ArrayList<String> patterns4 = new ArrayList<String>();
                  HashSet<String> unset2 = new HashSet<String>();
                  HashMap<String, Integer> tc = new HashMap<String,
Integer>();

                  for(int i=0; i < lines2.size()-wsize; i++){
                        for(int j=i; j < i+ wsize; j++){
                              curp.add(lines2.get(j));
                        }

                        if(unset2.add(curp)){
                              patterns4.add(curp);
                        }
```

```
                    curp.clear();
                }

                ArrayList<String> lines3 = lines from Type 3 log

                //create dictionary of patterns for Type 3
                ArrayList<String> patterns1 = new ArrayList<String>();
                HashSet<String> unset3 = new HashSet<String>();

                for(int i=0; i < lines3.size()-wsize; i++){
                    for(int j=i; j < i+ wsize; j++){
                        curp.add(lines3.get(j));
                    }

                    if(unset3.add(curp)){
                        patterns1.add(curp);
                    }

                    curp.clear();
                }

                //compare Type 4 dictionary with Type 5 and Type 1 to
find p2 dictionary
                ArrayList<String> qdict = new ArrayList<String>();
                ArrayList<String> rdict = new ArrayList<String>();
                int min = 0;
            for(int i=0; i < patterns4.size(); i++){
                String[] foo = patterns4.get(i);
                min = 1000;
                for(int j =0; j < patterns5.size(); j++){
                    String[] bar = patterns5.get(j);
                    int ret = EditDistance(foo, bar);
                    if(ret < min){
                        min = ret;
                    }
                }

                if(min >= threshold){
                    int min2 = 1000;
                    for(int k=0; k < patterns1.size(); k++){
                        String[] bar = patterns1.get(k);
                        int ret = EditDistance(foo, bar);
                        if(ret < min2){
                            min2 = ret;
                        }
                    }

                    if(min2 < threshold){
                        rdict.add(patterns4.get(i));
                    }
                    else{
                        qdict.add(patterns4.get(i));
                    }
                }
```

```
            }


            return qdict;
          }
          catch(Exception e){
                e.printStackTrace();
          }
          return null;
    }
```

P3 ALGORITHM PSEUDOCODE

```
public static void P3() {

      Initialize files
      String file1
      String file2
      String file4
      String file5

      ArrayList<String> p1 = t1vt2(file1, file2);

      ArrayList<String> p2 = t4vt5t1(file4, file5,file1);

      int min = 0;
      int threshold = 0;
      int dups = 0;
      ArrayList<String> attackmatches = new ArrayList<String>();

        for(int i=0; i < p1.size(); i++){
           String[] foo = p1.get(i);
           min = 1000;
           for(int j =0; j < p2.size(); j++){
                 String[] bar = p2.get(j);
                 int ret = EditDistance(foo, bar);
                 if(ret < min){
                       min = ret;
                 }
           }

           if(min <= threshold){
                 attackmatches.add(p1.get(i));
           }
        }
}
```

4.3 Experimental Trials

For our experiments, we looked at using data from five applications that we exploited. We used exploits in Unreal, Samba, Vsftpd, Tomcat server, and MySql. We wanted to see if our program would be able to detect attacks on these applications. To conduct our simulations we set up virtual machines and ran a virtual network. Then we recorded the system calls of the applications to create profiles during normal use. After that, we began running exploits on the applications to see if our anomaly detection methods would discover them. We found that it was always able to detect the attacks that we conducted on it with low false positive rate. It seems exploits will frequently cause system calls to be anomalous in a manner that can be detected. Exploits on applications like Tomcat could cause 796 anomalous sequences of size 7 to be detected. However, the range in terms of the size of anomalous traces caused by exploits could vary greatly. For example, in VSFTPD, it only took 40 anomalous traces for an exploit to succeed. This shows that predetermining a threshold to indicate whether a run of system calls is malicious or not may be difficult because an exploit may succeed with many system calls or few system calls. However, it also seems that very few system calls will be declared as anomalously malicious in our system if they are indeed benign. We ran multiple cases of new behavior to determine if our algorithm would correctly determine that the new behavior was healthy or not. New safe behavior that was not seen in the original behavioral profile of ACC, would be classified as healthy in our system due to the comparison checks with the PCC that our method employs. Therefore, we do indeed see that our system does a good job in distinguishing new, healthy behavior from new malicious behavior. Even with an incomplete behavioral profile of an application, we could still determine whether a sequence of system calls was malicious which is something that prior methods could not do.

Our virtual computers that we used to simulate the ACC, PCC, and PCD were using the Ubuntu 8.04 versoin running previous versions of these exploits. They were running on Intel processors with core i5 processors using 2GB memory for each VM and regular hard drive space.

Table 4.1: Discovering Exploits in Applications

Application	Anomalous matches form exploit	Anomalous matches from new behavior Case #1	Anomalous matches from new behavior Case #2	Anomalous matches from new behavior Case #3	Anomalous matches from new behavior Case #4	Anomalous matches from new behavior Case #5
Unreal	179	0	0	0	0	0
Samba	46	8	0	6	6	6
Vsftpd	40	0	0	0	0	0
Tomcat	796	3	4	127	8	8
MySql	172	4	0	14	5	na

4.4 Privacy

Our method was designed to also preserve the privacy of an application. Only the computer running the application knows the system calls that the application runs because only anomalous system calls are sent to the monitoring center. Therefore, the privacy of the ACC is not compromised. We explored other methods of analysis, mainly sending all files to the central monitoring service, but we thought that might hinder privacy of the computer by leaking details of its operation to the network. If we can avoid this issue when possible, it would be ideal. However, if one does not care about privacy, they could potentially offload all the files and computations to the central monitor. This would alleviate the CPU burden on the server of running the anomaly detection scans of the normal behavioral profile. A server may be running multiple applications so a different anomaly scanner would need to be created for each application. We noticed that some applications create thousands of system calls a second and handling all these logs could be unwieldy. If an enterprise server handles a large amount of

requests from online users that could overload the anomaly detection algorithm in P1, then an alternative might be to send the logs to another computer, either hosted on the same rack to avoid leaking information to the network or possibly sent to the network if one does not have sensitive data and is just trying to prevent its computers from being misused for botnets.

4.5 Summary

In this chapter, we have seen The Triple Pot model and discussed how it works, the assumptions made, and the cases when it can be applied. We showed that our experiments verified the effectiveness of The Triple Pot model on real exploits running on real applications. We were able to achieve strong performance despite not using a complete behavioral profile of an application. We could detect anomalous behavior and then determine whether it was malicious or benign. In this way, we could greatly reduce the number of false positives that would normally swarm a security operator running system call analysis using previous methods.

CHAPTER 5

DISTRIBUTED SYSTEM CALL ANALYSIS

5.1 Assumptions and Use Case

This section covers a probabilistic model of how a network of computers running system call monitoring software could aggregate their data to be utilized to find zero day exploits. This would involve multiple networks from different organizations sharing their data together. This model is not related to The Triple Pot model above. It is a separate model dealing with the case where an organization may have multiple computers on a network. There may be other organizations running the same program in a different location. For security purposes, they may want to share their system call logs with a secure third party monitoring center. Anyone who has used computers has probably encountered messages when an application crashes, asking whether they want to send logs to the application's company to help debug the problem. This is a common scenario used by companies developing applications. In our case, instead of sending logs, we want to send the system call activity. The system call activity is obviously more extensive and fine grained than a regular error log. In this way, it can potentially be a bigger drain on CPU usage and network bandwidth. However, for system that require high assurance of security, there may be a necessary need to undertake distributed system call analysis.

In order for it to be possible to aggregate data from multiple organizations, there needs to exist a trusted third party that people will not mind sending their data to. This third party would have to agree to keep all system call activity confidential and not to reveal anything to anybody. We believe that if multiple organizations share their system call activity, that it would be possible to enhance security by making it easier to identify

zero day exploits. The third party could compare traces and see which ones match and which ones did not match with anybody else's traces.

Although different organizations may have different data used for each application, the application's function still remain the same. The system calls record the functions of the application and it is not necessary for us to rely computers sharing the same data. Computers just need to share the same application and perform similar functions to get the same system traces. For example, the save button on an app should produce the same sequence of system calls regardless of what data actually gets saved. System calls have two main attributes, the call itself and the data that the call handles. Different organizations should share the same call functions, but just with different data handled in the argument list for those calls. We mainly focus on the system call functions and not the system call arguments used.

5.2 Internal Network Analysis

The first thing we may want to consider is the internal network of computers in an organization. An organization may have dozens of servers running identical programs. We assume that there is a dedicated central monitoring application for each network. This dedicated central monitoring center would be able to collect the relevant data from all the computers in the internal network to see which applications run on which computers. All the computers running an instance of the same application would be categorized together. This model does not use any dummy computers, but rather focuses on the use case where a company wants to see if any information can be gleaned from the system call activity of just the real computers alone. Once again, we make the assumption that any malicious hacker would scan a network for all vulnerable computers. Then, they would try to hack into as many computers as possible. If we know what the hacker is trying to do, then we can build a model around it.

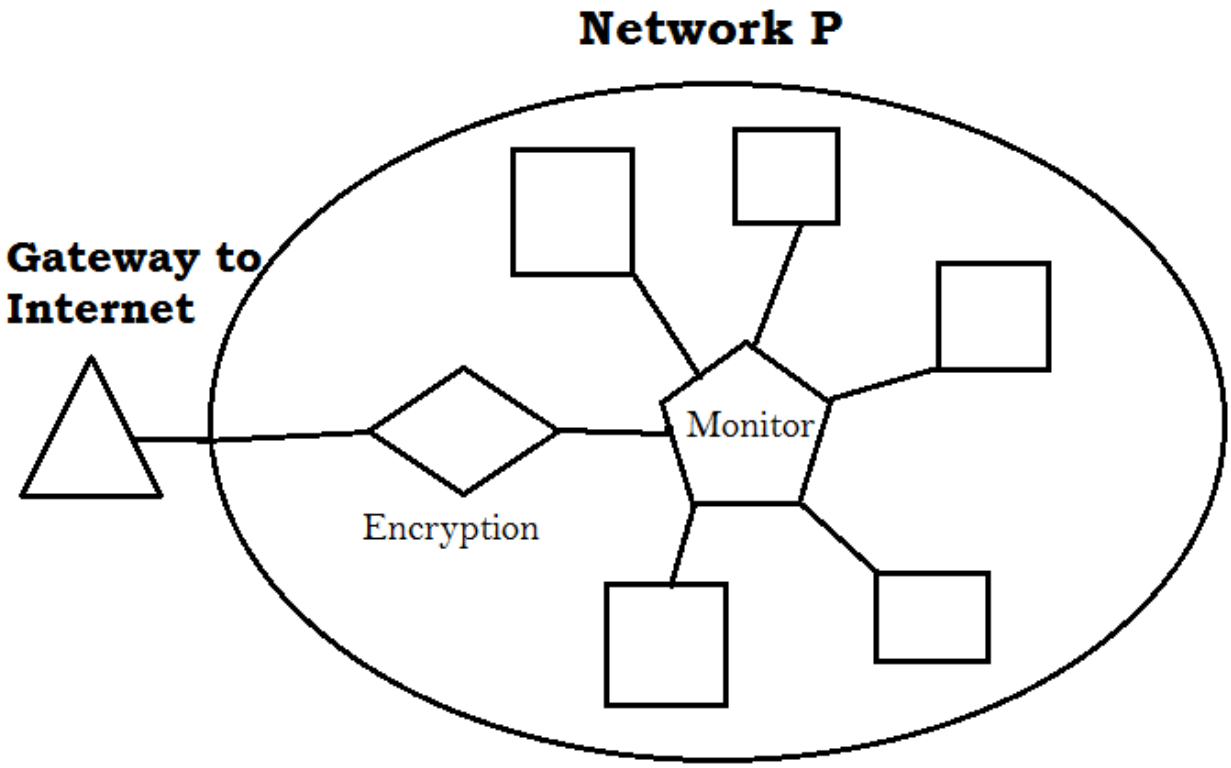

Figure 5.1: Internal Network Model

First, we want to believe if a hacker has a list of vulnerable computers, there are two ways he could go about attacking them. First, he could manually attack them one by one by himself. Second, he could write a script that automatically attacks them. If an organization had a vulnerable application that could be exploited by this malicious hacker and this vulnerable application was installed on multiple computers, then we assume that an anomalous trace that only appeared once would not be malicious. Sometimes, applications may just go on an errant behavior either caused by a bug in the application or something else. These one time errant behaviors could cause an anomalous system call sequence to be thrown, however, it would not be a malicious sequence caused by a hacker. We want to be able to capture this reasoning in our mathematical model to generate a score that would increase if there is a higher likelihood of a malicious exploit and decrease if it is not a malicious exploit. However, we also want to have our score to increase high for the initial few matches and then to increase less later on as more matches are found and there is diminishing returns. For

an enterprise data center that may have 100 or so server running the same program, we assume that either very few of the computers will be exploited or almost all the computers will be exploited. We don't think we will see many cases where there are a medium number of exploits on potential targets. This is because a hacker will either do it manually and thus get very few targets since he has to do it by hand, or he will have a script and thus be able to attack all of them quickly. We believe this type of growth in the scoring model can be represented by a logarithmic graph. These graphs have a curve that first grows quickly with the first few numbers and then plateaus out near the end of the curve. By varying the base of the logarithmic function, we can change how steep the scoring curve should be on our heuristic. We chose a log base of 2 because it seemed to be a reasonable number for most enterprises with a couple dozen server computers. Larger companies with hundreds of computers may want to change their base to be a higher number like 10.

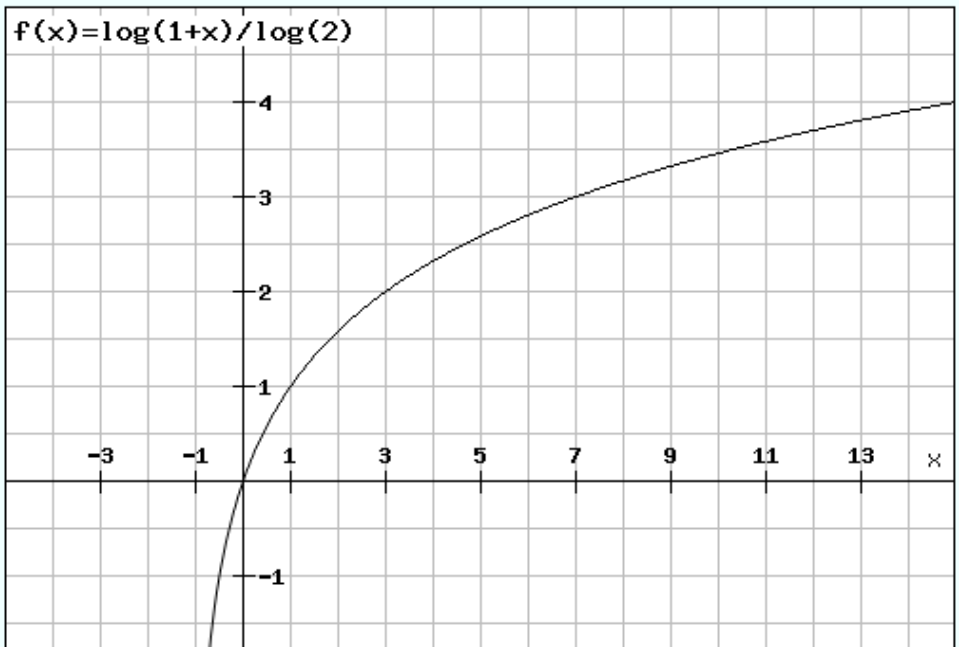

Figure 5.2: Log Plot Graph

Let H(P, Z) represent the score that a network P has for a trace Z to determine whether it is anomalous or not

$$\mathrm{H}(\mathrm{P},\mathrm{Z}) = \log_2(1+i) \tag{5.1}$$

i represents the number of internal computers that share a trace Z

As the number of computers that contain an anomalous trace increases, there is greater likelihood that it is an exploit. In our algorithm we add 1 to the inner logarithm because we want to shift the curve to the left by one. In this way, if there is only 1 computer possessing an anomalous trace then it will be log(2) which is 1. We thought that the number 1 would be a good starting point for the equation.

5.3 External Network Analysis

When we think about the external network, we consider computers in other organizations that may be running the same application. For example, MySql is a commonly used application that can be deployed on multiple machines in many companies. If we again assume the advanced persistent threat model then we assume that a zero day exploit being used at one company is unlikely to be used at another. Therefore, by comparing traces from one company to another, we can gain a better understanding on whether a system call sequence is malicious or not. If only one company has experienced a system call sequence, but not any of the other companies, then we claim that increases the probability that the sequence is malicious. To express the relationship among the companies mathematically, we adopt a probabilistic model.

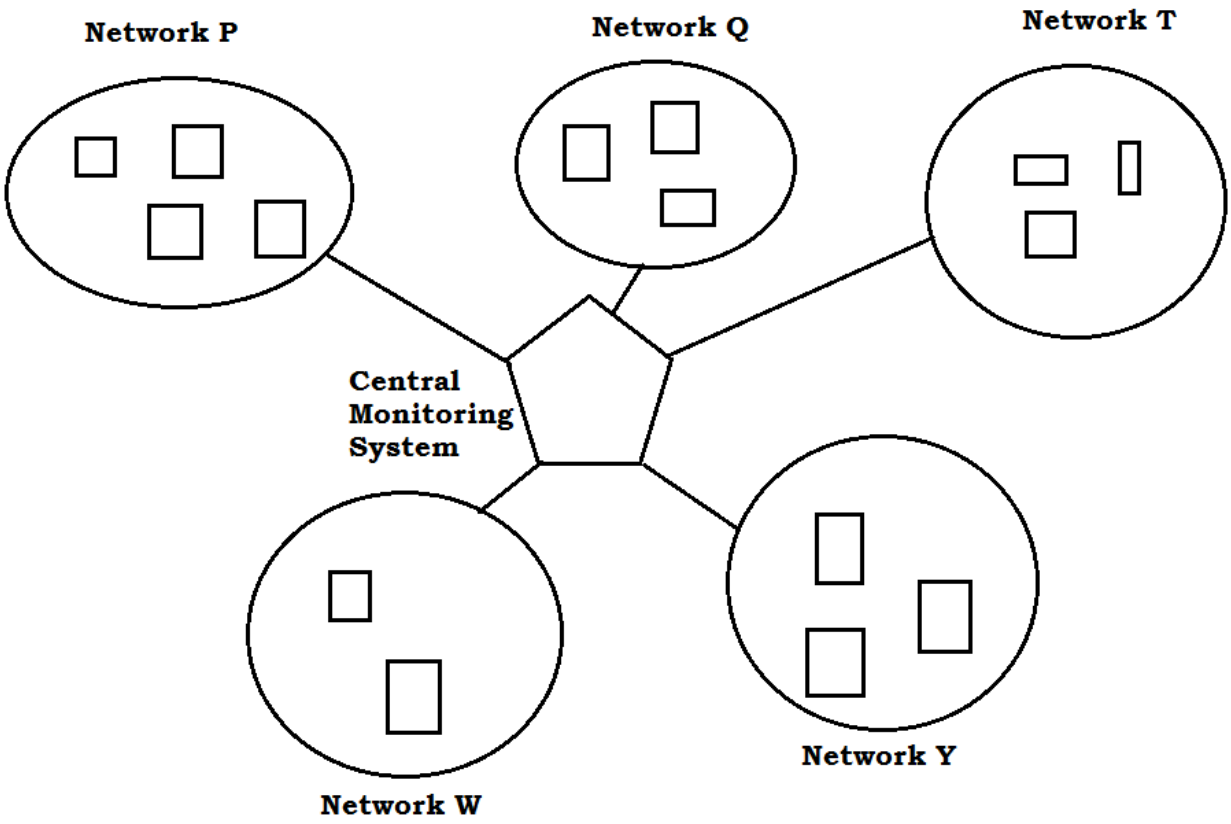

Figure 5.3: External Network Model

First, we assume that the computers on the networks are sending their system call logs to the central monitoring system. This monitoring system should then be able to handle the processing of the logs. It compares traces from two computers by calculating a similarity score between their logs. If two computers have logs that are similar than they are more likely to share a random trace between the two of them. Different computers running an application may use the application for different purposes. By comparing the probability that a trace from one computer will be found in another computer we can help identify whether computers are running the same types of workloads. This probability score is calculated by counting the number of matches in computer A that are also in computer B divided by the total number of sequences in A. This will give us a future probability that any randomly anomalous sequence in A will be in B. As such we call this probability J(P, Q) with P and Q being computers that we are comparing to. For an internal network of computers we perform the summation of all the

computers running on the internal network running the same program. Then we take the J(P, Q) from all the external networks being compared and multiplicatively combine them together to get a total score Q(P, Z) to determine whether a trace is malicious or not. We use the information from all networks to help facilitate this calculation so that we have an optimal amount of information to make our judgments.

J(P, Q) = Probability that P has a trace in Q

Calculate J(P,Q) by looking at the number of traces that P has in common with Q to calculate a similarity score

$$J(P,Q) = \frac{|P \cap Q|}{|P|} \quad (5.2)$$

Q(P, Z) = Probability that P has a trace Z which is malicious, given n networks

$$Q(P,Z) = \prod_{i=1}^{n}(1 - Ji) \quad (5.3)$$

We make a distinction between computers running on the same internal network and computers running on an external network by utilizing different techniques for each one and generating two different scoring values. In this way a security monitor can then compare the two values: Q(P, Z) and H(P, Z) to determine whether an anomalous trace Z is malicious or not. We consider a trace to be a sliding window of 1,000 system calls. If more than 30% of the system calls within that trace match up with another trace from another network then we consider that to be a match for the trace.

5.4 Experimental Results

To test our theory, we wanted to examine 3 relationships between traces to see how similar they are. The first relationship we test is whether the same application on multiple machines exhibits identical behavior that can be recorded to be similar. One potential issue is that applications have background noise that can interfere with the sequencing of the system calls. Some applications may have a lot of background noise to deal with while other applications may have less to deal with. If there is a lot of background noise, it may be difficult to generate a profile or to compare applications from different machines even if they are identical. We tested Tomcat server, Vsftpd server, and Unreal IRC server. We found that Tomcat server had some background noise, but that in general background noise did not deter the system call profiling much. Identical applications exhibited identical behavior that could be captured in a profile to be used for comparison.

The second relationship we tested was whether exploits exhibited the same system call traces when being run against applications. Again, we found that we could profile an application's exploit consistently from machine to machine. It was again Tomcat's background processes that could sometimes interfere to hinder the profiling, but only by a little

The third relationship we tested was how different exploits were from the normal behavioral profile of an application. For this we noticed that exploits could be created to be very similar to a behavioral profile or very different. Some exploits reproduced many valid system calls with just minor adjustments and were still able to succeed. Other exploits reproduced few valid system calls and produced many anomalous traces and were easier to detect. This goes to show that deciding upon a threshold for how many anomalous sequences in a trace should constitute a declaration of that trace as an exploit may be difficult for single host based anomaly detection systems. That is why our system that leverages multiple machines is more effective than a single machine running anomaly detection.

APPLICATION BEHAVIOR TO APPLICATION BEHAVIOR COMPARISON

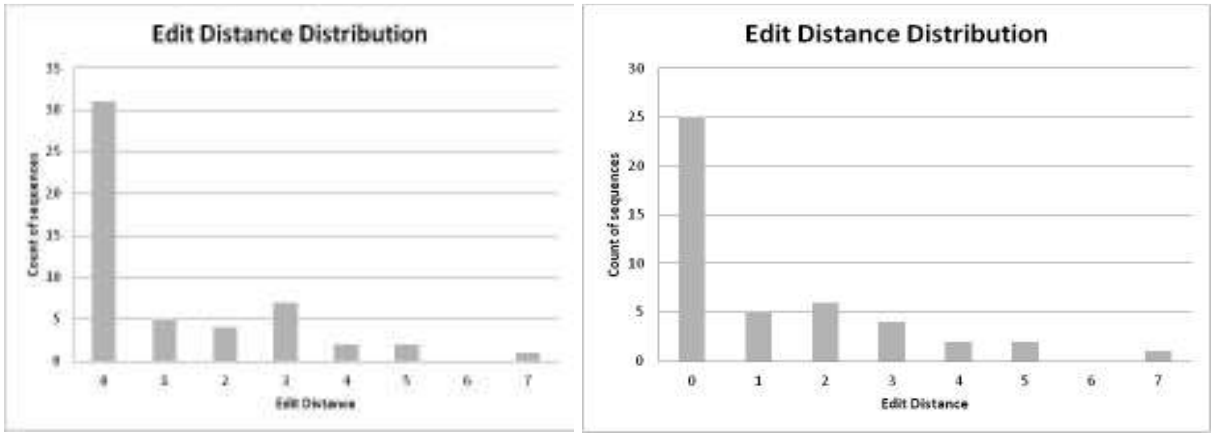

Figure 5.4: Comparison of a Tomcat trace to 2 other Tomcat Traces performing the same actions

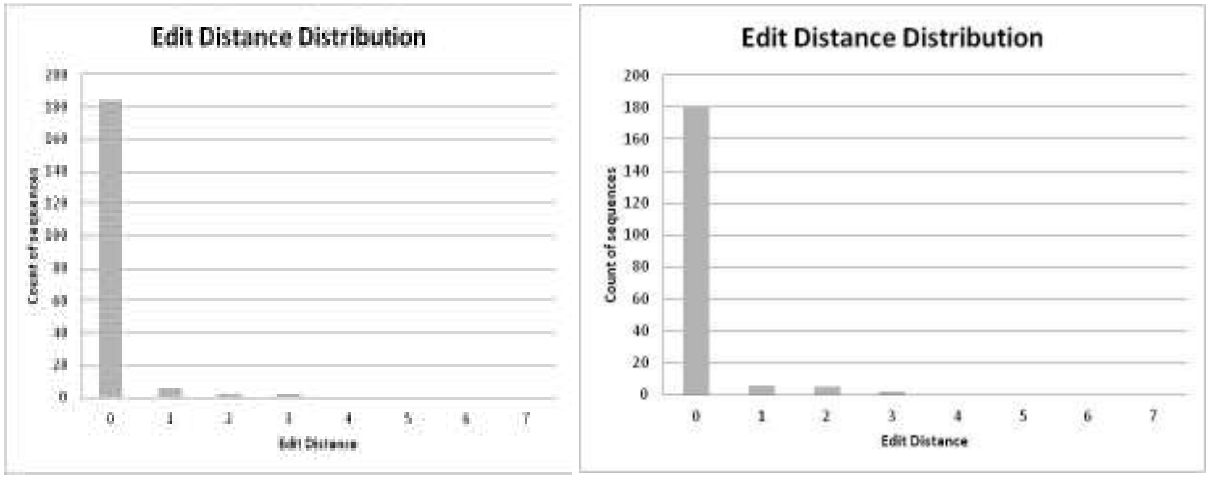

Figure 5.5: Comparison of a Vsftpd trace to 2 other Vsftpd Traces performing the same actions

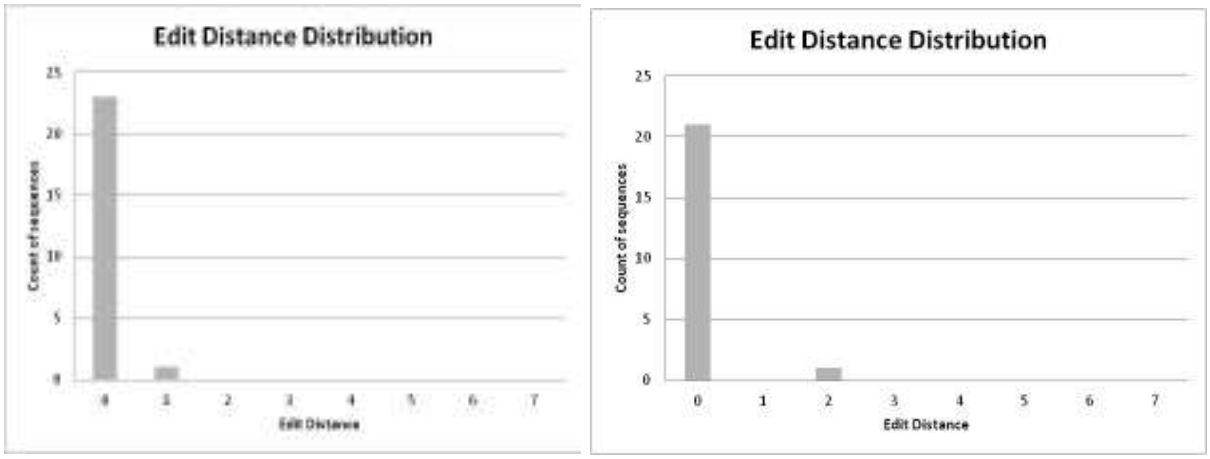

Figure 5.6: Comparison of an Unreal trace to 2 other Unreal Traces performing the same actions

APPLICATION EXPLOIT TO APPLICATION EXPLOIT COMPARISON

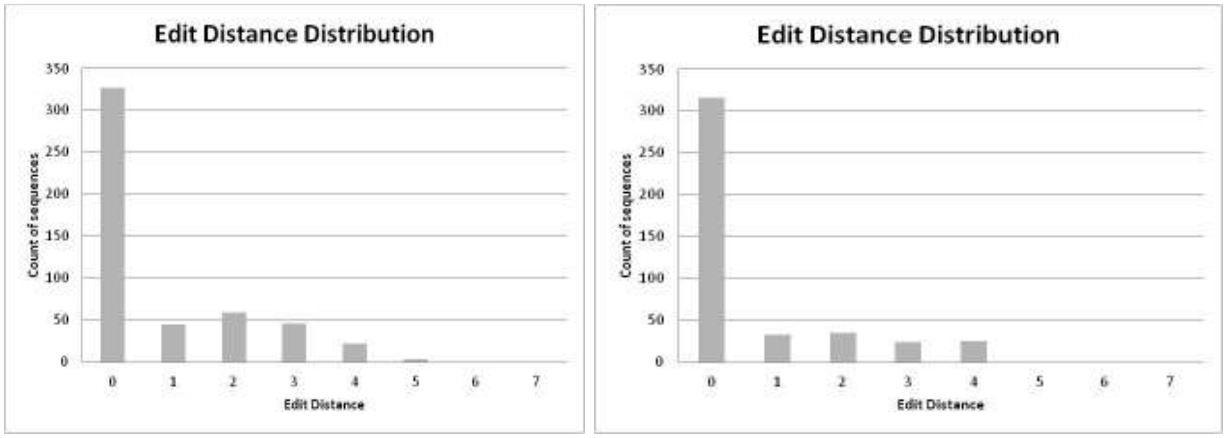

Figure 5.7: Comparison of a Tomcat trace to 2 other Tomcat traces performing the same exploit

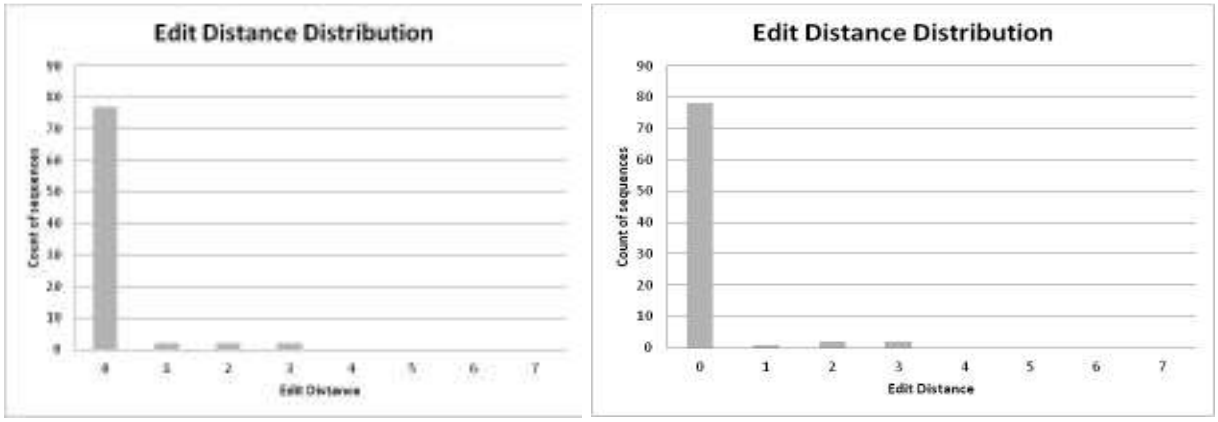

Figure 5.8: Comparison of a Vsftpd trace to 2 other Vsftpd traces performing the same exploit

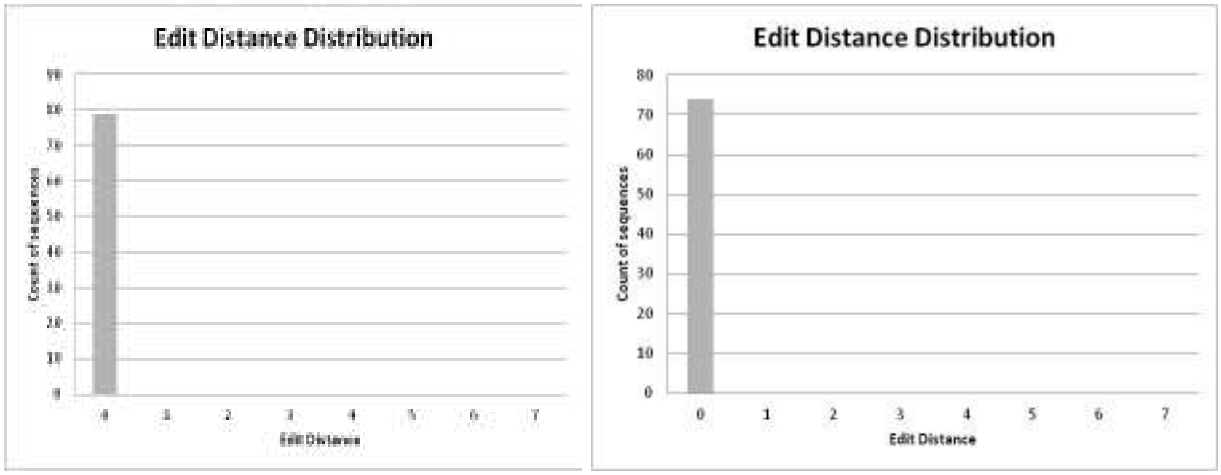

Figure 5.9: Comparison of an Unreal trace to 2 other Unreal traces performing the same exploit

APPLICATION EXPLOIT TO APPLICATION PROFILE COMPARISON

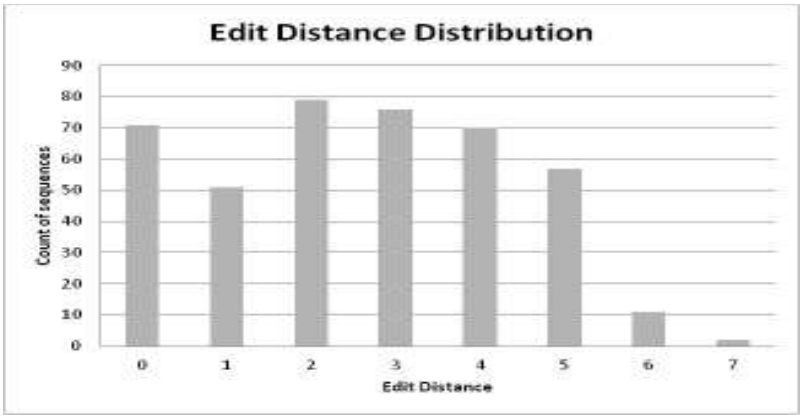

Figure 5.10: Comparison of a Tomcat exploit to the Tomcat normal behavioral profile

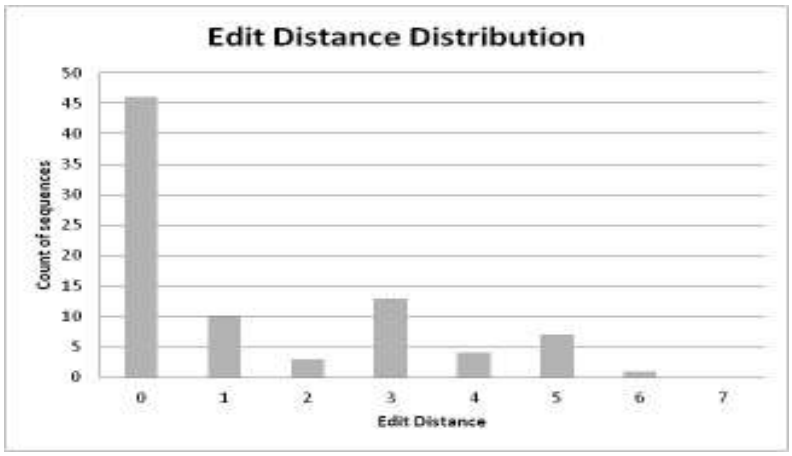

Figure 5.11: Comparison of a Vsftpd exploit to the Vsftpd normal behavioral profile

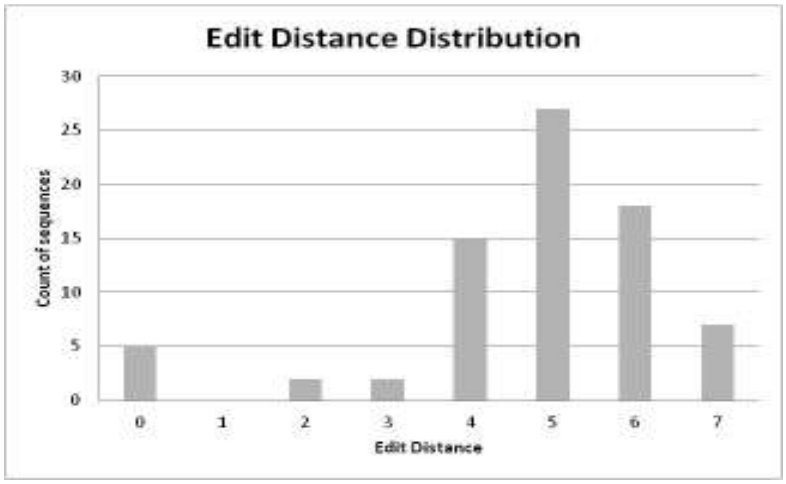

Figure 5.12: Comparison of an Unreal exploit to the Unreal normal behavioral profile

CHAPTER 6

CONCLUSION

In summary, we have seen how system call tracing can be an additional tool in the security professional's bag to examine their computing infrastructure for bugs. We have introduced The Triple Pot method for when one wants to set up a secure hosting environment for their enterprise server. Our method differs from prior approaches by analyzing multiple streams of system call data simultaneously. This method helps identify anomalous traces that are malicious and is able to classify benign traces as safe. In this way, we are able to achieve low rates of false positives so that computer administrators do not see an unreasonable number of reports of suspicious activity. We looked at multiple real life exploits and set up a test environment to verify and improve our initial model. By making certain assumptions about the hacker's methodology, we can create a better defense against the Advanced Persistent Threat model.

In addition, we also looked at a separate model where multiple organizations may collaborate together to share information about their system call methods. Our method introduced a mathematical formula for indicating the likelihood that an anomalous trace was really a malicious exploit or just an errant run of the program. We also looked at how to compare traces from multiple organizations to generate a score for how likely a trace is an exploit. These two metrics can be used in conjunction with one another to help analyze computers for possible security violations. Our experimental results on the similarity of applications and exploits showed that it was possible for background processes to add noise to the profiling and tracing process that could obscure traces, however, this noise was often minimal and it seemed that valid, correct traces of program execution could be created.

Host based intrusion detection is a topic that can still be greatly explored and the continuous streams of reports of new zero-day exploits make finding new methods to discover them more important than before.

REFERENCES

[1] Frei, Stefan. “The Known Unknowns” NSS Labs, December 2013 (https://www.nsslabs.com/reports/known-unknowns-0)

[2] Walder, Bob. “The Targeted Persistent Attack(TPA) - When the Thing That Goes Bump in the Night Really is the Bogeyman” NSS Labs, August 2012 (https://www.nsslabs.com/reports/targeted-persistent -attack-tpa-misunderstood-security-threat-every-enterprise-faces)

[3] Lichodzijewski, Peter, A. Nur Zincir-Heywood, and Malcolm I. Heywood. "Host-based intrusion detection using self-organizing maps." *IEEE international joint conference on neural networks*. 2002.

[4] Sujatha, P. Kola, et al. "A behavior based approach to host-level intrusion detection using self-organizing maps." *Emerging Trends in Engineering and Technology, 2008. ICETET'08. First International Conference on*. IEEE, 2008.

[5] Topallar, M., et al. "Host-based intrusion detection by monitoring Windows registry accesses." *Signal Processing and Communications Applications Conference, 2004. Proceedings of the IEEE 12th*. IEEE, 2004.

[6] Stolfo, Salvatore J., et al. "A comparative evaluation of two algorithms for windows registry anomaly detection." *Journal of Computer Security* 13.4 (2005): 659-693.

[7] Ilgun, Koral, Richard A. Kemmerer, and Phillip A. Porras. "State transition analysis: A rule-based intrusion detection approach." *Software Engineering, IEEE Transactions on* 21.3 (1995): 181-199.

[8] Ning, Peng, Sushil Jajodia, and Xiaoyang Sean Wang. "Abstraction-based intrusion detection in distributed environments." *ACM Transactions on Information and System Security (TISSEC)* 4.4 (2001): 407-452.

[9] Ghosh, Anup K., Aaron Schwartzbard, and Michael Schatz. "Learning Program Behavior Profiles for Intrusion Detection." *Workshop on Intrusion Detection and Network Monitoring*. Vol. 51462. 1999.

[10] Hofmeyr, Steven A., Stephanie Forrest, and Anil Somayaji. "Intrusion detection using sequences of system calls." *Journal of computer security* 6.3 (1998): 151-180.

[11] Tandon, Gaurav. *Machine learning for host-based anomaly detection*. Diss. Florida Institute of Technology, 2008.

[12] Liu, Anyi, et al. "Enhancing System-Called-Based Intrusion Detection with Protocol Context." *SECURWARE 2011, The Fifth International Conference on Emerging Security Information, Systems and Technologies*. 2011.

[13] Creech, Gideon, and Jiankun Hu. "A Semantic Approach to Host-based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns." (2013): 1-1.

[14] Sekar, R., et al. "A fast automaton-based method for detecting anomalous program behaviors." *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001.

[15] Eskin, Eleazar, Wenke Lee, and Salvatore J. Stolfo. "Modeling system calls for intrusion detection with dynamic window sizes." *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*. Vol. 1. IEEE, 2001.

[16] Wagner, David, and Paolo Soto. "Mimicry attacks on host-based intrusion detection systems." *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002.

[17] Janakiraman, S., and V. Vasudevan. "ACO based Distributed Intrusion Detection System." *JDCTA* 3.1 (2009): 66-72.

[18] Yegneswaran, Vinod, Paul Barford, and Somesh Jha. "Global Intrusion Detection in the DOMINO Overlay System." *NDSS*. 2004.

[19] Janakiraman, Ramaprabhu, Marcel Waldvogel, and Qi Zhang. "Indra: A peer-to-peer approach to network intrusion detection and prevention." *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*. IEEE, 2003.