ABSTRACT

Mahalati, Jaideep. Facilitating Alert Correlation Using Resource Trees. (Under the direction of Dr. Peng Ning).

With the steady increase in the number of attacks against networks and hosts, security systems such as intrusion detection systems are widely deployed into networks. Intrusion detection systems may flag large numbers of alerts, where false alerts are mixed with true ones. To understand the security threats and take appropriate actions, it is necessary to perform alert correlation. One class of alert correlation methods is the prerequisite and consequence based approach, where the prerequisite of an attack is the necessary condition to launch the attack, and the consequence of an attack is the possible outcome if the attack succeeds. Through matching the consequence of earlier attacks with the prerequisites of later ones, attack scenarios can be discovered. However, one limitation of these approaches is that the specification of prerequisites and consequences for different alert types usually is time-consuming and error-prone. To address this limitation, this thesis proposes a resource tree based method to facilitate the specification of prerequisites and consequences. Attacks can be viewed from the perspective of resources. Example resources include various network services and privileges. This thesis further organizes resources into trees, where the nodes in the trees are labelled with conditions (represented by predicates). To specify the prerequisite and consequence of an attack, it is required to look for the desirable resource trees related to the attack's prerequisite and consequence, then traverse the trees to find the appropriate nodes, and finally select the suitable predicates to put into the prerequisite and consequence. This approach is simple and less expert-dependent. The usability study and comprehensiveness study (with more than 3000 alert types) demonstrate the effectiveness of this approach. Correlation results with different datasets further show that prerequisites and consequences defined using our methodology can be effectively used for alert correlation.

Facilitating Alert Correlation Using Resource Trees

by

Jaideep Mahalati

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh

2005

Approved By:

Dr. Ting Yu	Dr. Douglas S. Reeves

Dr. Peng Ning
Chair of Advisory Committee

To my parents...

Biography

Jaideep Mahalati was born in Indore, India, in 1981. He received his Bachelor's degree in Computer Science from Devi Ahilya University, Indore, India in 2003. Since 2003, he has been a Masters student in the Department of Computer Science at North Carolina State University. In summer of 2004 he worked at Citrix Systems Inc., Fort Lauderdale, Florida, as a software development engineer intern.

Acknowledgements

I sincerely acknowledge the efforts of Dr. Peng Ning, my thesis advisor, in providing me apt direction and guidance whenever required. He helped me remain focused and persistent all the way through. Working with him has been a pleasure and an honor.

I am grateful to my committee members, Dr. Douglas S. Reeves and Dr. Ting Yu, for devoting time and offering valuable inputs.

I also appreciate the help I received from my colleague Dingbang Xu. I received valuable suggestions from him regarding my work.

This work would not have been possible without the selfless help and support I received from Ratna.

I also acknowledge the fact that my roommates– Pratik, Manav and Saurabh, supported me throughout and let me skip my cooking turns in times of need.

Finally, I would also like to thank my family for always being there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increased usage of computer networks and the Internet, the number of network attacks have also increased. To defend against these attacks, intrusion detection systems (IDSs) are widely deployed into networks. The main goal of the intrusion detection system is to identify intrusive activities and raise alerts in such an event.

Intrusion detection can be broadly classified into two types: *Anomaly Detection* and *Misuse detection*. *Anomaly Detection*: In this type of intrusion detection a "normal activity profile" is defined for a system and any activity that deviates from this profile, by a predefined threshold margin, is considered intrusive. There are certain drawback of this technique. 1) Anomalous behaviour that is not intrusive is considered as an intrusion (false positive). 2) Also intrusive activities that are not anomalous result in false negatives.

Therefore, the key issue here is to select the threshold levels to reduce both false positives and false negatives. Anomaly detection systems are computationally expensive because it requires maintaining several system profile metrics.

Misuse Detection: Another alternative intrusion detection technique is based on the principle of representing attacks in the form of patterns or *signatures*. In this approach an intrusion is detected by matching the predefined signatures for these intrusive activities. This technique is efficient in identifying the different variants of the same attack. The main issue here is that it cannot identify new attacks for which signatures are not defined.

It is well-known that IDSs may flag a huge number of alerts, among which false alerts are mixed with real ones. To learn the security threats represented by these alerts,

usually it is necessary to perform alert correlation.

Alert correlation can be defined as combination and analysis of intrusion alerts for the purpose of intrusion alert refinement and scenario building. Alert correlation reduces the number of alerts making it easier for the system administrator to comprehend the attacks and take appropriate actions.

Existing alert correlation approaches can be roughly divided into four categories: (1) approaches based on the similarity between alert attributes (e.g., [30, 28, 16, 6]), (2) approaches based on pre-defined attack scenarios (e.g., [10, 18, 9]), (3) approaches based on the prerequisites (pre-conditions) and consequences (post-conditions) of attacks (e.g., [29, 7, 21]), and (4) approaches integrating multiple security systems such as IDSs and vulnerability scanners (e.g., [24, 19, 35, 36]). These approaches can perform alert clustering, or build step-by-step attack scenarios to help security officers understand security threats.

We are particularly interested in the prerequisites and consequences based correlation approaches, which target at discovering attack scenarios. An attack scenario is a sequence of steps taken by attackers to accomplish their goal. To build attack scenarios, prerequisites and consequences based approaches model each attack through specifying its prerequisite and consequence, where the prerequisite is the necessary condition to launch the attack successfully, and the consequence is the possible outcome if the attack succeeds. For example, the prerequisite of *FTP_AIX_Overflow* attack is AIX ftpd services running on the victim hosts, and its consequence is to gain root privilege on the victim hosts. Through matching the consequence of an earlier attack with the prerequisite of a later one, causal relations between individual attacks are identified and attack scenarios are built by integrating multiple causal relations. These approaches are quite promising as they do not require any prior knowledge about the attack scenarios or the huge amount of correlation rules to be specified.

We notice that specifying the prerequisite and consequence for each type of attacks is crucial for the prerequisites and consequences based approaches. Current methods for specifying prerequisites and consequences [29, 7, 21] usually employ a manual, ad hoc process, which is time-consuming and error-prone even for security experts. Prerequisites and consequences are usually modeled by predicates [7, 21]. For example, predicate *SadmindService(VictimIP)* denotes an Sadmind service running on host *VictimIP*, and predicate *RootAccess(VictimIP)* represents the root privilege on host *VictimIP*. It is quite possible that different users may use different predicates to denote the same condition, or use the

same predicate to represent different conditions. Users may neglect some conditions that are essential to the attacks, or give some unnecessary conditions in the specification, which may lead to the failure of the correlation methods.

Motivated by these limitations associated with the manual specification, we propose a resource tree based approach to facilitate the systematic specification of prerequisites and consequences.

We observe that each attack can be viewed from the perspective of *resources*. For example, given an attack of *Sadmind_Amslverify_Overflow*, we notice that the necessary requirement to launch this attack is the *Sadmind* service running on a *Solaris* machine, and the possible outcome if the attack succeeds is that the attacker may gain root privilege. From the *resource* point of view, the prerequisite is *Solaris* operating system and *Sadmind* service, and the consequence is the root privilege. In addition, we notice that some resources may be further split. For example, privileges can be classified into root access and local access. Thus, it is desirable to organize resources into tree structures, which we call *resource trees*.

We use resources trees to facilitate the specification of prerequisites and consequences of attacks. Specifically, we label nodes in the resource tree with pre-defined predicates, where these predicates represent the conditions the corresponding nodes satisfy. For example, we may mark the node of "root access" in a "privilege" resource tree with *RootAccess(VictimIP)*, denoting a root privilege on host *VictimIP*. Based on the resource trees with predicate labeling, the specification of prerequisites and consequences is straightforward. For each type of attack, we find desirable resource trees for both prerequisites and consequences, go through the paths in the trees to find the desirable nodes, and put the appropriate predicates associated with the nodes into the prerequisites and consequences, respectively.

Our approach makes the specification of prerequisites and consequences efficient and less expert-dependent. We conducted a set of usability experiments. Our results demonstrate that the specification process is simple and efficient. We asked users from different backgrounds (security and non-security fields) to apply our method. The results show that there is no significant difference between these users in terms of correctness (97.5% for security students and 96.9% for non-security students). We also performed a comprehensiveness study for our resource trees with more than 3000 attack types. In addition to these experiments we also used TIAA[22], a correlation tool, to validate our set

of prerequisites and consequences. The results of the experiment and the type of datasets used are discussed in section 4.3.

We notice that Xu and Ning [34] recently proposed a resource based approach for alert correlation. In [34], each attack is modelled through input and output resources, where input resources are the necessary resources to launch the attack successfully, and the output resources are the resources the attack can provide if it succeeds. [34] focuses on the method to identify the causal relations between individual attacks through matching the output resources of earlier attacks with the input resources of later ones. However, [34] does not provide a systematic way to specify input and output resources. We organize resources into resource trees, and we provide a systematic way to specify resources related to prerequisites and consequences. We propose a method to utilize these resource trees and identify the predicates for defining prerequisites and consequences of different alert types.

1.1 Summary of Contribution

The major contribution of this thesis is to provide a systematic approach for specification of prerequisite and consequence for intrusion alert correlation. This resource tree based approach is extensible in nature, making it possible to model different types of attacks. Our methodology makes this process less expert-dependent and more efficient. Defining the prerequisites and consequences is the core of correlation techniques like [29, 7, 21, 22, 34]. The results of the experiments with different datasets, section 4.3 show that our approach works efficiently for different alert types. The comprehensiveness study, section 4.2 and the usability analysis, section 4.1 further show the completeness and ease of use of our approach.

Another contribution of this thesis is a complete set of prerequisites and consequences for all the snort alert types. I have defined prerequisites and consequences for more than 3000 *Snort* alert types. I have build the knowledge base [21], which is the collection of all the hyper-alert types with the set of all the predicates used and the implication relationships between the predicates, for all these snort alert types. This is a very big asset for anyone using the TIAA [22] for correlating alerts. The user of TIAA only needs to use the predefined knowledge base built by me to perform correlation for different Snort alert types with the actual alert data. Any alert type that is not defined in the knowledge base can be

defined using our resource tree based approach. The knowledge base acts as the dictionary of attacks for the prerequisite and consequence based correlation techniques.

1.2 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives an overview of the correlation method based on prerequisites and consequences [21] and discusses some related research in alert correlation. Chapter 3 describes our resource tree based approach to specify prerequisites and consequences for different alert types. Chapter 4 presents the evaluation of our approach and the experimental results. Chapter 5 concludes this thesis and points out some open issues.

Chapter 2

Background Study

2.1 Prerequisite and Consequence Based Alert Correlation

The work in this thesis is to extend a previous correlation method based on prerequisites and consequences of attacks [21]. Our goal is to facilitate the specification of prerequisites and consequences, which makes the correlation approach much easier to apply. In this section, we give an overview of this correlation approach with slight modification, which simplifies our discussion without losing the essence of the method. For the sake of presentation, we call this correlation approach the *causal correlation approach* since it identifies causal relationships between alerts.

The causal correlation approach [21] models each type of detected attacks (i.e., alerts) through specifying its prerequisite, consequence as well as attributes. Intuitively, the prerequisite of an attack is the necessary condition to launch the attack successfully (e.g., a vulnerable ftp service), the consequence of an attack is the possible outcome if the attack succeeds (e.g., gaining the root privilege), and the attributes are the related information about the attack (e.g., the source and destination IP addresses of the attack). Prerequisites and consequences are specified by predicates. Formally, an *alert type* is a triple (*fact, prerequisite, consequence*), where *fact* is a set of alert attribute names, *prerequisite* is a logical formula whose free variables are in *fact*, and *consequence* is a set of logical formulas where all the free variables are in *fact*. For example, we can model an alert type

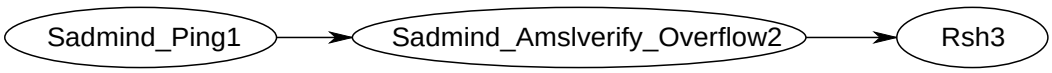


Figure 2.1: An example of alert correlation graphs

FTPBufferOverflow as { {SrcIP, SrcPort, DestIP, DestPort}, FTPService(DestIP, DestPort), {RootAccess(DestIP)}}, denoting that it has four attributes, its prerequisite is a ftp service running on host *DestIP* at port *DestPort*, and its consequence is that the attacker may gain root privilege at host *DestIP*. Given an alert type T = (*fact, prerequisite, consequence*), a type T alert t is a tuple on *fact*, where an interval-based timestamp [*begin_time, end_time*] is associated with this tuple. For example, an *FTPBufferOverflow* alert can be {SrcIP=10.100.1.1, SrcPort=3456, DestIP=10.100.1.20, DestPort=21} with a timestamp [5-15-2005 10:10:10, 5-15-2005 10:10:11].

Given a type T alert t, the *prerequisite set* (or *consequence set*, respectively) of alert t is a set of instantiated predicates through replacing the attribute names in T's *prerequisite* (or *consequence*, respectively) with their corresponding values in t. These prerequisite and consequence sets for alerts can help us identify causal relations (which are formalized as *prepare-for* relations) between individual alerts. Formally, given two alerts t_1 and t_2, if one of the instantiated predicates in t_1's consequence set implies one of the instantiated predicate in t_2's prerequisite set, and $t_1.end\_time < t_2.begin\_time$, then we say t_1 *prepares for* t_2. Intuitively, if an earlier attack can contribute to the prerequisite of a later one, these two attacks can be correlated.

An attack scenario is a sequence of prepare-for relations that link individual attacks together, which is represented as an alert correlation graph. An *alert correlation graph* $CG = (N, E)$ is a directed acyclic graph, where any node $n \in N$ is an alert, and any edge $(n_1, n_2) \in E$ represents that n_1 prepares for n_2. Figure 2.1 shows an example of alert correlation graphs. In Figure 2.1, attackers first probe the *Sadmind* service (*Sadmind_Ping*), then launch buffer overflow attacks against this service (*Sadmind_Amslverify_Overfow*) to gain root privileges on the victim host, and finally use remote shell (*Rsh*) to control the victim host.

Our method to facilitate the specification of prerequisites and consequences is based on the causal correlation approach in [21]. However, our method can also be extended

to accommodate other prerequisite and consequence based correlation approaches such as [7].

2.2 Related Work

In recent years, several alert correlation approaches have been proposed. We roughly divide these approaches into four categories. (1) The similarity based approaches [30, 28, 16, 6, 15, 25]. These approaches essentially perform alert clustering through computing the similarity between alert attributes. These techniques fail to identify causal relationships between related alerts. (2) The pre-defined attack scenario based approaches [10, 18, 9]). These approaches build attack scenarios through matching alerts to the predefined attack templates. Thus, their ability highly depend on the known scenario templates. The attack scenarios must be manually specified by the human users or can be learned through training datasets. (3) The prerequisites (pre-conditions) and consequences (post-conditions) based approaches [29, 7, 21, 22, 34]. These approaches build attack scenarios with the prior knowledge of individual attack types. However, they do not require knowledge about scenario templates, and they can discover novel attack scenarios. (4) Multiple security systems based approaches [24, 19, 35, 36]. These approaches correlate alerts from different security tools such as IDSs and vulnerability scanners.

The EMERALD correlation component uses a well defined list of alert attributes, which Valdes and Skinner term as alert features [30]. The EMERALD correlation component includes the following features: 1) identification of the sensor in terms of its identifier, location and name, 2) the alert thread that allows the sensor to make a decision on correlation itself while overriding all other matching criteria, 3) the incident class, 4) the source and the target IP lists, 5) the target TCP/UDP port lists, 6) the source user id, 7) the target user id, and 8) the time. Additionally, sensors report specific incident signatures that display a lot of variance between heterogeneous sensors and as a result do not hold much relevance. The alert attributes have similarity values ranging from 0 to 1 based on the probabilistic alert correlation. Various methods are employed to determine similarity value for specific attributes [30]. For instance, the similarity value can be determined from a predefined similarity matrix (e.g. similarity of different attack classes). The similarity value can be also calculated as the numerical distance (e.g. time) or as the degree of overlap

of lists (e.g. lists of IP-addresses). However, it is not possible for all sensors to produce alerts with all of the similarity criteria. For this reason, the available alert attributes are employed in the similarity calculation. The overall similarity of two alerts, as calculated in a probabilistic alert correlation system, is the weighted average of the similarities of the overlapping alert attributes. This similarity calculation uses the predefined expectation of similarity as weights whose value ranges between 0 and 1. The correlation component may also assign minimum similarity values to alert attributes specific to a situation. Several different views of the incidents may be obtained by using different values for expectation of similarity and minimum similarity. This approach is incapable of capturing the precise correlation between events and falls in the first category of correlation techniques.

Another correlation system, the Mission Impact Intrusion Report Correlation System (M-Correlator), has been developed to extend the EMERALD system developed at SRI International by adding more information. Besides using the IDS alert information the M-Correlator System also makes use of the information external to it [24]. Alert classes that are insignificant are filtered out. All alerts are assigned a relevance score and a priority. Alert relevance is calculated based on the information about prerequisites of an attack and the information regarding the target system environment. Alert priority is assigned based on the level of significant impact of the alert class on the target and on the level of importance of that target. The M-Correlator system then employs "incident ranking" that ranks alerts based on their calculated relevance scores, assigned priority and the probability-of-success value in the alert messages. Incident ranking is performed with the help of an implementation of the Bayes networks. Henceforth, a clustering algorithm based on alert attributes is employed to combine related alerts. The Aggregation and Correlation Component (ACC) was built by Debar and Wepsi on top of the Tivoli Enterprise Console (TEC). TEC is an event-handling product of IBM/Tivoli Systems in their RAID paper from year 2001 [10]. Debar and Wepsi focussed on four major issues in the existing IDS that proved to be weaknesses: alert flooding, insensitivity to context, generation of false alerts and limited scalability. The intrusion-detection architecture presented by Debar and Wepsi consists of ACCs and "probes" in hierarchical distribution for providing scalability although there is little explanation provided on how this hierarchical structure of ACCs is achieved. The probes are IDS that generate alerts in the common TEC-event format. In some cases, the probes can be gateway probes that are used in conjunction with the proprietary systems to transform the alerts to the common TEC-event format.

The algorithm used for correlation in the ACCs is a hybrid algorithm that takes into account both the predefined attack scenarios as well as the comparison between available alert attributes. The algorithm takes into account only three alert attributes namely- the source, the target, and the attack class. Only selected attributes are compared to form different views of the incident, therby utilizing a very basic version of the minimum similarity and expected similarity concept. Further, the algorithm only considers perfect matches of the alert attributes. Due to the absence of any correlation language the correlation rules are either entered manually or generated from the configuration files. Two kinds of preprogrammed correlation rules used by ACCs are "duplicates" and "consequences". The Duplicate rules are employed to combine alerts representing same attack but that are generated from separate sources. The duplicate rule defines the various attack classes (specific names of known attacks) that are to be considered as duplicates and the specific attributes of the alerts that should match. The Consequence rules are employed to combine alerts that are known to occur in pairs in a multi-step attack. The consequence rule defines two attack classes, two probe ids, and a wait time representing the maximum time between the two alerts under consideration for correlation. It is possible, therefore, to program predefined attack scenarios into the system but it will still be very tedious and will fail to be generic.

Quicksand adopts a distributed approach for alert correlation [5]. Quicksand bases its distributed structure of correlation scheme on the numerous Event Correlation Units (ECU) that reside on the machines on the network. Typically, ECU resides on the same machine as the one on which the probe it is communicating with resides and there is one ECU per probe (IDS). Each ECU plays a part in the process of correlation by communicating directly to a probe to receive alerts and implement the distributed pattern matching algorithm. The algorithm employed for correlation in Quicksand is also based on predefined attack scenarios. However, the novelty of the algorithm used in Quicksand lies in its distributed structure. Unlike ACCs, Quicksand uses the Attack Scenario Language (ASL) to describe the predefined attack scenarios. All events have unique identifiers based on event types and situation-specific constraints. Each ECU has its local "pattern" that is defined as a correlating sets of events. Each pattern has the capability to send events collected locally in it for further processing in other ECUs. Initialization of patterns on each ECU still needs to be done by a central control unit in accordance with the ASL definitions. In this manner, the complete attack scenario is distributed over several ECUs in the form of local patterns. The advantages of this system are threefold: 1) the network traffic is

reduced as the ECUs know what alerts are relevant to other nodes in the network, 2) use of distributed architecture increases fault tolerance, and 3) performance of the correlation component is enhanced.

A number of attack specification languages have been developed for describing the attack scenarios. Moring and Debar use the chronicles formalism [19] to describe the attack scenarios based on time information using a very high level language. A chronicle is a set of events, interlinked by time constraints, and whose occurrence may depend on the context. Krgel et al. present another specification language, the Attack Scenario Language (ASL), for attack scenarios in conjunction with their distributed pattern matching algorithm [5]. The Mirador-project utilizes the LAMBDA language [8] which is very similar to ASL.

STATL [11] is another attack specification language for attack scenario modelling. STATL has a well defined syntax and a parser. STATL is used to model attacks in both the network and host domains. In STATL attacks are viewed with the perspective of *states* and *transitions*. The attack is modelled as the different states between the initial safe state and the final compromised state and the various transitions between these states. These languages are use to specify pre-defined attack scenarios.

Templeton et al. use JIGSAW [29], another attack specification language. JIGSAW is used to model an attack in terms of *capabilities* and *concepts*. Capabilities refer to the preconditions that help an attack to be successful. The model proposed by Templeton et al. describe an attack being composed of *concepts* or subtasks in an attack scenario. The requirement and the impact of a concept is expressed in terms of capabilities. The correlation is done by matching the capability provided by one concept with the capability required by other.

The NetSTAT framework uses an alternate scheme to utilize the pre-written attack scenarios. It breaks down the attack scenarios into smaller tasks that can be carried out by single sensors and then correlate these tagged alerts [33].

M2D2 is formally defined as a data model for IDS alert correlation. It handles four information types: information about characteristics of the monitored system, information about the known vulnerabilities, information regarding security tools, and information regarding alerts and scans. The information garnered from the security tools, the vulnerability scanners and the IDS, is stored in a relational database. This database also has information on the product obtained from the ICAT vulnerability database [13]. The monitored system is modelled based on perspectives of both network nodes and the product.

The network nodes are modelled based on Vignas TCP/IP networks topology formal model [31]. However, M2D2 makes some extensions to this model, for instance, adding mappings between hostnames and IP addresses. The products are modelled in the form of quadruples using vendor id, product id, version id, and type. Depending upon their usage the products belong to either of the four main product classes namely, OperatingSystem, Server, LocalApp, and Other. The Server class may be further classified into specific server classes like httpServ and ftpServ. Information from the ICAT database [13] is utilized to model known vulnerabilities that affect configurations. Non-CVE/CAN vulnerability names are translated into CVE/CAN vulnerability names [2] using a mapping function. Three access classes - Remote, RemoteUser, and Local - are used to model the prerequisites for an attack. Typically, the RemoteUser class represents remote login with user access privileges. Further, the consequences of the attack are modelled using three classes namely, CodeExec, DoS, and Info. The security tools are modelled either as IDS or the vulnerability scanners. The security tools generate reports which may be mapped to the vulnerabilities. The IDS employ the scheme of misuse detection or anomaly detection. In general, IDS are host or network based security tools while vulnerability scanners affect the hosts only. Moring et al. [1] states that since an alert reflects the state of an IDS, it is modelled as an event. This concept corroborates the earlier definitions of alerts modeled as events in the perspective of other tools. The current models of IDS cover events at various different network layers (IPv4, TCP, UDP, HTTP network events) and web server log events. Moring et al. [1] cover several aggregation techniques in their paper with examples. There is a function that combines all host based and network based alerts that refer to a named target. Additionally, another function identifies all systems that could be vulnerable to an incoming alert. Different approaches are taken depending upon comparison between the number of incoming alerts and the number of false negative alerts. The incoming alerts can be either true or false while the false negative alerts are generated from a non-reactive IDS. Also, an alert similarity function is employed to identify an IDS reactive to the same set of “basic” events [1].

Several attack and vulnerability taxonomies have been proposed to help evaluate if a device is vulnerable to certain attacks or not. Taxonomy of attacks refers to their classification in a way that a user of a device can uniquely identify different attacks. Previous contributions to taxonomy of attacks have concentrated on the techniques employed by the attacker, resources used by the attacker, and the purpose of the attack.

Neumann [20] represents the Program Analysis (PA) Study in the form of nine classes of security *flaws* or vulnerabilities. These nine classes are arranged in the form of four main classes and five subclasses. The four main classes are improper protection, improper validation, improper synchronization, and improper choice of operand or operation. Improper protection comprises of 5 subclasses of vulnerabilities namely - improper choice of initial protection domain, improper isolation of implementation detail, improper change, improper naming, and improper deallocation or deletion. Improper validation involves not having proper checks for critical conditions, allowing type clashes and overflows. Improper synchronization is further classified into 2 subclasses - improper indivisibility, and improper sequencing. Improper choice of operand or operation involves engaging in scheduling algorithms that block certain processes or users from executing, and using wrong arguments or methods. This taxonomy is quite ambiguous in that certain subclasses point to same vulnerabilities and hence the notion of classification for uniqueness is lost.

The Research in Secured Operating Systems security taxonomy [26] identifies seven classes of security vulnerabilities. The classes are - incomplete parameter validation, inconsistent parameter validation, implicit sharing of privileged or confidential data, asynchronous validation or inadequate serialization, inadequate identification or authentication or authorization, violable prohibition or limit, and exploitable logic error. Here, there are are no subclasses as in the PA study but it is similar in that there is still some ambiguity between the classes.

Kommerling et al. [23] classify attacks and vulnerabilities into four major categories namely, software attacks, eavesdropping techniques, fault generation techniques, and micro-probing techniques. The software attacks are meant for taking advantage of security vulnerabilities. The eavesdropping techniques are used to keep track of electromagnetic radiations produced by processors along with the analog characteristics of all the supply and interface connections. The fault generation techniques are employed to spawn malfunctions in the processors to get further access by monitoring abnormal environmental conditions. The micro-probing techniques are used to gain access to the chip surface directly.

Howard [12] propose the "STRIDE" categorization of attacks namely - spoofing identity, tampering with the data, repudiation, information disclosure, and elevation of privilege. Spoofing identity entails making illegal use of someone else's authentication information. Tampering with data involves malicious modification of data. Information disclosure is the exposure of information to unauthorized users. Elevation of privilege refers

to an un-privileged user to gain privileged access.

Xu and Ning [34] proposed a resource based approach to perform alert correlation. In [34], each attack is associated with input resources (prerequisite related resources) and output resources (consequence related resources). [34] focuses on identifying causal relations between individual attacks using input and output resources. The specification of input and output resources is still a manual process, where expert knowledge is essential. We propose a systematic way to specify resources and hence the prerequisites and consequences. We also notice that Chari and Cheng [4] propose BlueBox, which controls the access to system resources for intrusion detection purpose. In this thesis, resources are used to facilitate the intrusion alert correlation.

Chapter 3

Proposed Methodology

3.1 Facilitating Alert Correlation Using Resource Trees

We propose a resource tree based approach to systematically specify the prerequisites and consequences for intrusion alert correlation. To use this approach, we first pre-define a set of resource trees, and then for each type of attack, we examine what resources are related in the prerequisites and consequences, find the suitable resources in resource trees, select appropriate predicates associated with the resources, and finally put these appropriate predicates into the prerequisites and consequences, respectively.

3.1.1 Resources and Resource Trees

In our approach, we identify various *resources*. Generally speaking, *resources* are the objects of interest to the attackers. For example, various network services ("Service" resources) and privileges ("Privilege" resources) on victim hosts. To facilitate the specification of prerequisites and consequences, we are particularly interested in the resources that attackers can exploit or acquire during the course of the intrusion. For example, to launch an *FTP_AIX_Overflow* alert type, the attacker requires *AIX ftpd* service running on the victim machine, and through exploiting the vulnerabilities in that service, the attacker may acquire the root privilege on the victim host. Thus, for *FTP_AIX_Overflow* alert type, the resource related to its prerequisite is *Service* resource, and *Privilege* resource is related

to its consequence.

We observe that some resources can be further classified. For example, *privilege* resource can be divided into *local* and *root* privileges, denoting the normal user privilege and administrative privilege on the system, respectively. This partitioning of resources forms tree structures, which we call *resource trees*. The nodes in the resource tree represent the corresponding resources based on certain classification.

Generally speaking, *resource tree* is a way of representing the hierarchical structure of the resources in a graphical form. *Resource* tree is a tree structure where different nodes represent specific resources and are further labelled with predefined set of predicates. For example, the resource tree for a *privilege* may have two levels, where *privilege* is the root node and the two leaf nodes are *local access* and *root access*. The pre-defined predicates represents the condition the node may possibly satisfy. For example, in the aforementioned *privilege* resource tree, the node *local access* is associated with the predicate *LocalAccess(VictimIP)*, denoting the condition local access to the host *VictimIP*. Here, the variable *VictimIP* represents the alert attributes reported by the intrusion detection system. The different alert attributes reported by the IDS are destination IP address, source IP address, destination port, source port and the time stamp associated with that attack. The variable *VictimIP* is replaced by DestIP or SrcIP after identifying whether source or destination IP is the victim machine, based on the alert dscription. Similarly, we have the variable *VictimPort* that is used to model the port number information of different services. For example, *FTP* service is modelled using the predicate *FTP(VictimIP,VictimPort)* denoting that the FTP service is running on the victim machine at the victim port number. This variable *VictimPort* is replaced by *SrcPort* or *DestPort* depending on the alert description.

For defining a resource tree, we identify a general class of *resource* being exploited by the attacker. This forms the root node. For example, *Operating system*, *Services* and *Privilege* are such resources. After this, we have to classify this general class into sub classes that would denote the specific type of that resource. These sub classes form the child nodes of the resource tree. For example, *Windows* and *Unix* are two sub classes of the Operating System resource. These are specific types of operating system so they form the child nodes of the root Operating System resource. After having classified these resources into specific resources and identifying the child nodes for the different possible resources in the tree, we define predicates for the resource tree. To define the predicates we identify the different ways in which a particular resource can be used by an attacker. For example,

the Operating system resource can be exploited in two ways: the attacker can learn what operating system is running on the machine or the attacker can cause the operating system to crash. These are represented by the predicates *OS(VictimIP)* and *CrashOS(VictimIP)*. The predicates for the child nodes are specified using the general predicates identified for the specific resource tree. For example, the node *Windows* has the two predicates defined based on the general predicates for the Operating resource tree *OSWindows(VictimIP)* and *CrashOSWindows(VictimIP)*. For all the predefined resource trees we identify the general predicates for those resource trees and use them to further attribute the predicates to all the different nodes in that tree. We have predefined seven different resource trees and defined the predicates for different resources in those trees. This gives a better insight in the process of building resource trees and further expanding them.

Note that there are implication relationships between predicates. For example, a *WU-FTPD* service at a victim host can imply there is a *FTP* service at that host. In the previous correlation method such as [21], these implication relationships have to be manually examined. However, through our resource trees, these implication relationships can be naturally derived. In a resource tree, the resource in a child node is a specific type of the resource in the corresponding parent node. Intuitively, the satisfaction of the child node related predicate can imply the satisfaction of the parent node related predicate. For example, from a *service* resource tree, usually we can derive that *WUFTPD(VictimIP, VictimPort)* implies *FTP(VictimIP, VictimPort)*. Through studying the child-parent pairs in a resource tree, the implication relationships can be automatically extracted.

The resource trees are critical to the success of our proposed method. To come up with comprehensive resource trees that can cover different alert types, we have studied 3301 *Snort* alert types as well as several RealSecure alert types, and extracted the various resources related to these types. We report typical resource trees in the next subsection.

3.1.2 Typical Resource Trees

***Operating System* Resource Tree.** In most of the cases the attacker tries to find out what operating system is running on the victim machine. Each operating system has a set of vulnerabilities that may be exploited by attackers. There are certain applications or services that can only run on some specific operating systems. For example, Microsoft Internet Information Services (IIS) runs on Windows platforms. The resource tree related

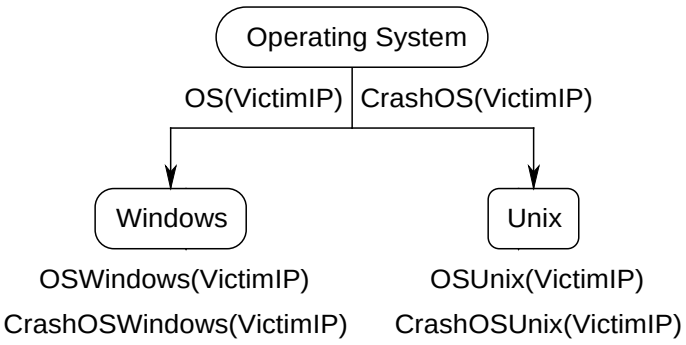


Figure 3.1: Operating System Resource Tree

to operating Systems is shown in Figure 3.1.

In Figure 3.1, the text in each node is the resource name, and the text besides the node is the predicates related to the corresponding node. We follow this convention for all resource trees. *Operating system* resource tree in Figure 3.1 has two levels. The root node *Operating System* is classified into two categories: *Windows* and *Unix* [1]. Operating systems can be exploited or crashed by attackers. For example, *IP_Unaligned_Timestamp* attack can crash some *BSD* operating systems. Another example, in Figure 3.1 node *Windows* has two predicates *OSWindows(VictimIP)* and *CrashOSWindows(VictimIP)*, where the former denotes that the windows operating system is running on the victim machine, and the latter denotes the crashing of the windows operating system. Thus, each node in the tree is associated with two predicates. Note that each leaf node in this tree can be further expanded. For example, Windows can be classified into Windows 2000, Windows XP, and so on. We will show resource tree expansion in a later subsection.

***Service* Resource Tree.** *Service* resources are usually various network services. For example, *ftp* services and *Sadmind* services. Attackers may exploit these services to achieve their goal, for example, gaining privileges on victim hosts. The resource tree for various services is shown in Figure 3.2 [2].

We divide various services into two categories: *fixed port services* and *varying port*

[1] It is easy to observe that Figure 3.1 is not a complete tree for all operating systems because there are other types of operating systems (e.g., Mac OS). We will show how to expand a resource tree in a later subsection.

[2] Note that the variables *IP* and *Port* in the predicates mean the victim host IP and vulnerable service port number, respectively. We use these short variable names only for the sake of brevity in the figure.

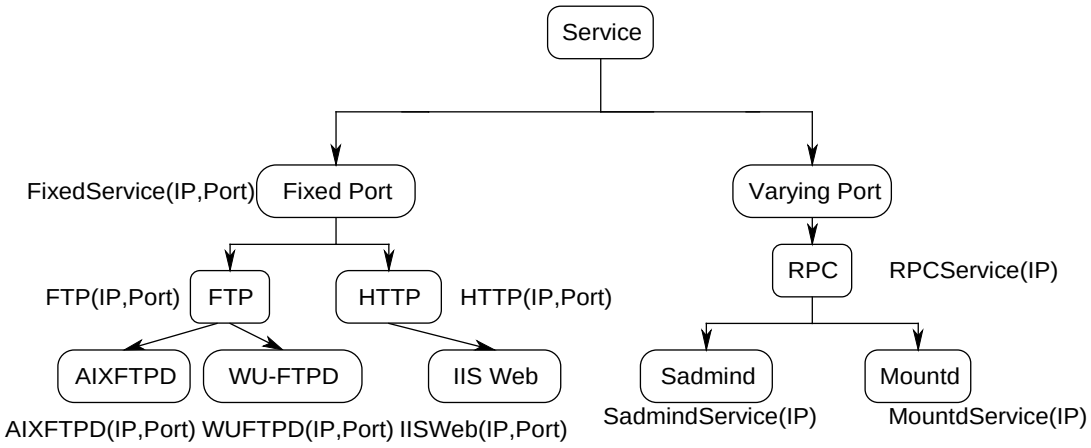


Figure 3.2: Service Resource Tree

services. We define fixed port services as those which use the same port number every time they start executing. For example, *ftp* is a fixed port service (usually with port 21). The varying port services are those which use different port numbers for each instance. For example, the different *RPC* services such as *Sadmind* ,*mountd* are varying port services because they are dynamically allocated port numbers and may use different port number every time they run.

These RPC services are dynamically allocated ports when a request is received to start these services. *Portmapper* is a service that registers the ports of all the RPC services running on a host. When an attacker wants to know the port of a specific RPC service it queries the Portmapper for that. For example, in the attack *RPC portmap sadmind request UDP* attacker queries the Portmapper to learn the port on which *Sadmind* service is running. The destination port reported by the IDS (Snort) is the port number of the Portmapper service i.e. 111 and not of the Sadmind service. Although the attacker learns the port number of the Sadmind service through the response from the Portmapper, it is not reported by the IDS because the attacker is actually contacting port 111 to gain the information. Now, after learning the port number of the Sadmind service the attacker can try and exploit the vulnerabilities in the service. For example, the former attack can be followed by *RPC sadmind query with root credentials attempt UDP* wherein this the attacker tries to exploit the vulnerability in the Sadmind service and gain access to the host running that service. Here, the attacker directly communicates with the Sadmind service

and therefore the destination port reported by the IDS is the one on which Sadmind service is running. For instance, port number 32773, this can be a different port as Sadmind is assigned different port every time it runs. Now, if we tried to model *Sadmind* using the IP address and the port number then the above mentioned two attacks would not be correlated because the port number would not match although they should be correlated. To take care of this kind of situation we model all the varying port services using the service name and the IP address. Thus, in the consequence of the first attack *RPC portmap sadmind request UDP* we would have *SadmindService(DestIP)* as the attacker learns that the victim machine is running Sadmind on a certain port. Also, the prerequisite of the second attack *RPC sadmind query with root credentials attempt UDP* would be *SadmindService(SrcIP)* because the attack requires the Sadmind service running on the victim machine. Thus, using the above mentioned convention we can correlate varying port services and get a better attack scenario.

We notice that various network services may be vulnerable to denial of service (DoS) attacks. Thus, each node in Figure 3.2 may relate to a specific predicate denoting DoS attack to this service is possible (we do not put these DoS related predicates into the figure only because we try to make the figure legible). For example, *FTP* node in the tree can further be related to a predicate *DoSFTP(IP, Port)* representing there may be DoS attack against this service.

***Privilege* Resource Tree.** *Privilege* resources are commonly involved in many attacks. Through certain attacks, the attacker may gain access or escalate access privileges on a victim machine. Some attacks (e.g., installation of a malicious program) may also require some privilege in the victim machine. Figure 3.3 shows a *privilege* resource tree.

We divide privileges into *local access* and *root access*, where *local access* means the attacker has normal user access (limited access) to the victim machine, while *root access* denotes administrative access on the victim machine.

By studying various alert types we learned that there there are alert types that reflect brute force or password guessing on the victim host. To model these attacks we define the predicates that denote the failed login attempts. These attacks indicate that an attacker is trying to gain access to a system, although he is not successful but this can alert the system administrator to take preventive measures to avoid a future compromise of the system. Each node in the Privilege resource tree is attributed with two predicates *Access(VictimIP)* reflecting successful compromise of the system and *AccessFailed(VictimIP)* depicting the

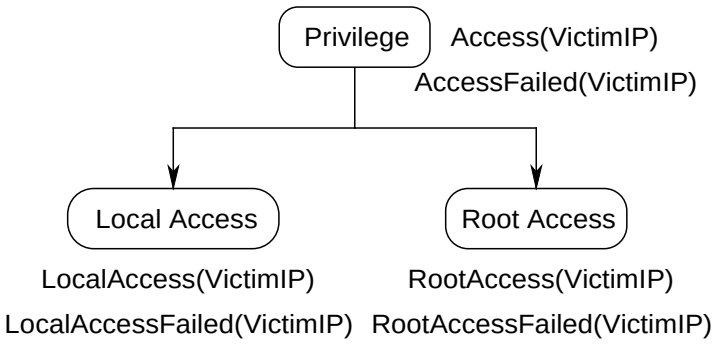


Figure 3.3: Privilege Resource Tree

failed login attempt and possible brute force against the victim machine. For example, the alert type *RSERVICES rlogin login failure* indicates that a remote login attempt has failed, as the description does not mention whether the attempt was for the root access of local user access so we represent the consequence as *AccessFailed(DestIP)*. If this alert appears multiple times that indicates that an attacker is attempting a brute force password guessing attack. An example of the successful login attempt is *RSERVICES rsh froot*, the attacker uses the *RSH* service on the victim machine to gain root access to that host. We represent the consequence for this attack by the *RootAccess(DestIP)* predicate.

File System **Resource Tree.** Sometimes files and directories are critical to certain attacks. For example, *WEB-CGI webplus directory traversal* attack targets at gathering directory information on the victim host, and *RPC mountd TCP export request* attack targets on gathering file system (NFS) information. We define a *file system* resource tree in Figure 3.4, where *file object* can be either directories or files. We also differentiate operations on directories or files (e.g., read, write and delete). The "Name" in the predicate names associated with nodes "Directory" and "File" are directory and file names respectively, which can be obtained from the attack signatures or reported by IDSs.

Malicious Software **Resource Tree.** Attackers might install certain malicious software on a host and then later exploit that malicious code to further launch an attack against the victim host. For example, in *BACKDOOR FsSniffer connection attempt* attack, the attacker uses the backdoor created by *FsSniffer* trojan to gain root privilege on the victim machine. In Figure 3.5, we show *malicious software* resource tree. The tree has

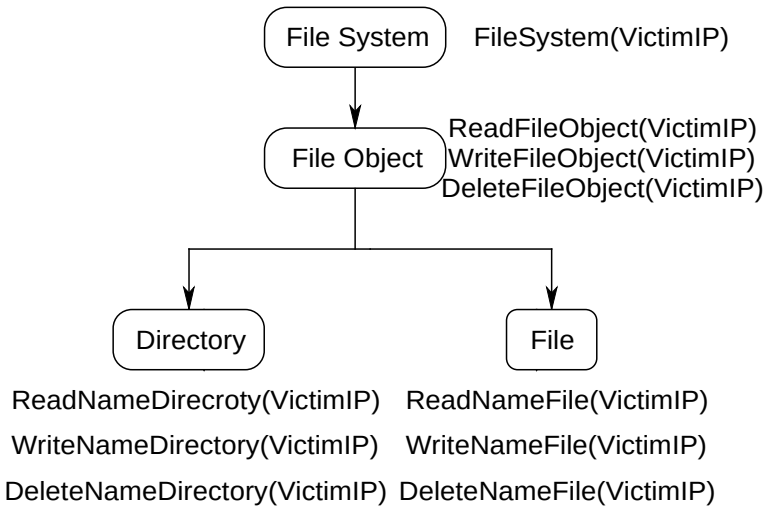


Figure 3.4: File System Resource Tree

zombie and *trojan* nodes, where *zombie* represents a software program that is used to launch DDoS attacks in the network, and *trojan* is a malicious software that runs surreptitiously on the victim machine to create backdoor for the attacker to launch further attacks. The examples of trojans are *Doly* and *Webloin* trojans, and the predicates *DolyTrojan(VictimIP)* and *WebloinTrojan(VictimIP)* denote the presence of these malicious softwares on the victim host. Zombies like *Mstream* zombies use a tired structure of clients and handlers, where handlers control the various distributed clients to attack a victim machine (DDoS attacks). The predicates related to these clients and handlers are represented as *MstreamClient(VictimIP)* and *MstreamHandler(VicitimIP)*, respectively. We represent the presence of distributed clients in the network by *MstreamZombie* predicate. There are no variables associated with this predicate because it is related to a distributed attack and the clients are distributed over various machines where intrusion detection may not be able to detect all of them.

***Application* Resource Tree.** There are several vulnerabilities present in specific applications that allow the attacker to exploit them. For example, **.XM* file handling in *Winamp* application is error prone. In *EXPLOIT winamp XM module name overflow* attack, a user exploits this vulnerability and may gain root privilege on the victim machine. Figure 3.6 shows the *application* resource tree. In this tree, we enumerate the

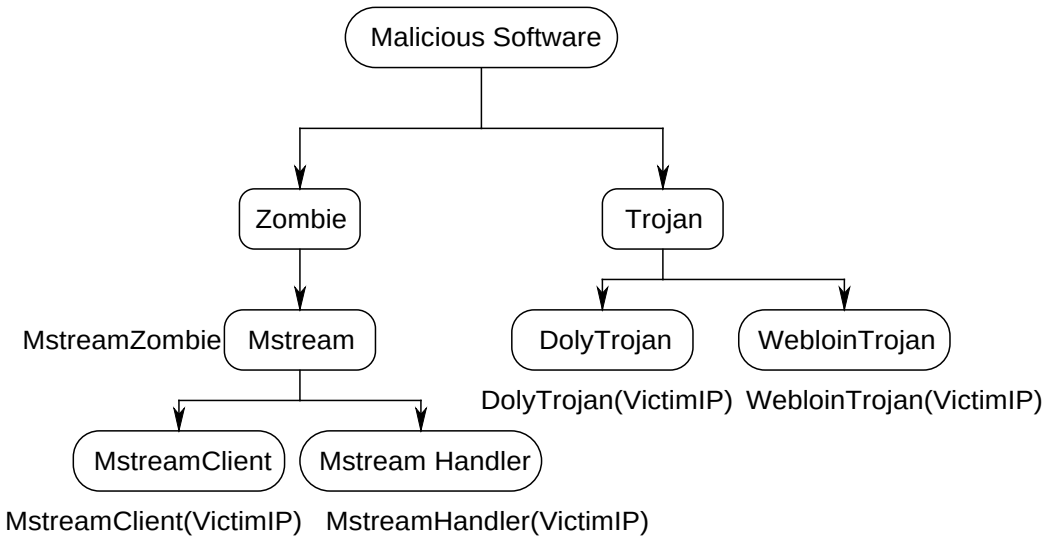


Figure 3.5: Malicious Software Resource Tree

various applications that can be exploited by attackers: *Winamp*, *Winzip* and *Netscape Navigator*. Considering these applications may be crashed by attackers, we also associate the corresponding predicates with them in the resource tree. For example, we have *Crash-NetscapeNavigator(VictimIP)* associated with node *Netscape Navigator*.

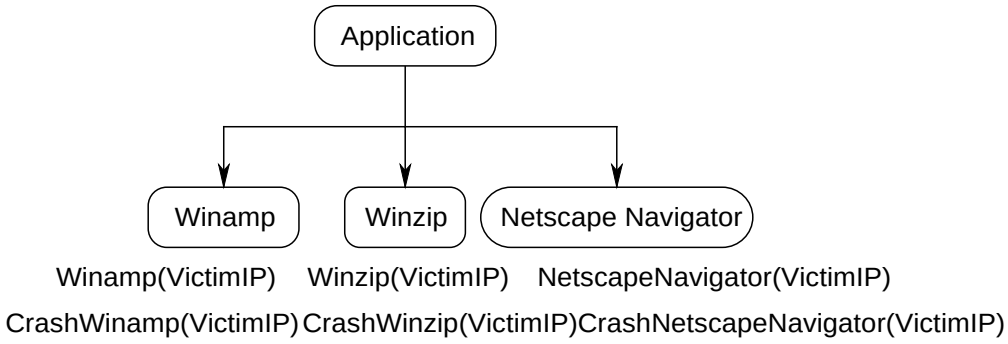


Figure 3.6: Application Resource Tree

***Hardware* Resource Tree.** Hardware resources are another common target of attackers. For example, in the attack *DOS Bay/Nortel Nautica Marlin*, attackers exploit the vulnerability in the Nautica Marlin bridge hardware component to crash the bridge

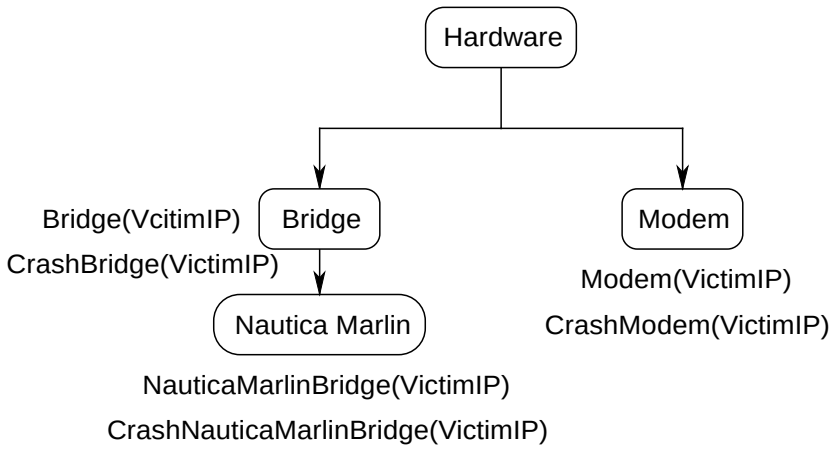


Figure 3.7: Hardware Resource Tree

and interrupt the network traffic. We represent the *hardware* resource tree in Figure 3.7. In Figure 3.7, *Nautica Marlin* represents a specific bridge, and one of its predicates is *NauticaMarlinBridge(VicitmIP)*, denoting the presence of this bridge. Considering that these hardware devices may also be crashed by the attacker (making them non-functional), we also put the crash related predicates with them.

Additional Resource. During our analysis of different alert types, we identified a few alert types that are intended to probe live hosts in the network. For example, *ICMP PING NMAP* attack uses *NMAP* to scan the network for the hosts that are up and running. To use resource trees to cover these probing alert types, we can define a *Live host* resource. The node *Live Host* is associated with the predicate *LiveHost(VictimIP)*, which denotes that the victim host is running in the network. This resource is not further classified and hence does not have a tree structure.

These network probing attacks are usually prelude to further scanning attacks that intend to scan the different ports on the live machine and detect the various services running on those hosts. For example, the *ICMP PING NMAP* attack can be followed by *SNMP request udp* in which the attacker probes the live hosts in the network , revealed by the first attack , for SNMP service and learns the port number of the service. Thus, by having *LiveHost(DestIP)* in the consequence of ICMP PING NMAP and in the prerequisite of the port scanning attack *SNMP request udp*, we can correlate the two attacks.

It is also possible that there may be other resource trees that we did not discover in our investigation of attack signatures. Fortunately, the construction of resource trees is an incremental process. If there are new resources that cannot be fit into any existing resources, we can generate a new resource tree. If a new resource can be put into an existing resource tree, we can expand this tree to accommodate it, which is discussed in the next subsection.

3.1.3 Expanding Resource Trees

Resource trees can be further expanded. When new resources are identified or further classification to the existing resources is required, we can add more nodes to the resource trees.

If the prerequisites and consequences for an alert type cannot be defined using the predefined resource trees, then the existing resource trees can be expanded. After reading the alert description, the resource that is involved in the attack is identified. If the resource is not present in the predefined resource trees then we have to identify the position where we need to insert this newly identified resource. If the new resource can be classified as one of the predefined resources then it should be made the child node of that particular resource. Then predicates to the newly added node will be assigned in similar way as defined for the resource tree to which it is being added. The predicates should describe the same operations as depicted by the predefined predicates for that resource tree. If a resource cannot be classified as one of the predefined resources then a new resource tree for that resource can be built by further classifying that resource and following the same procedure as used for the predefined resource trees.

Here is an example. Given a *operating system* resource tree in Figure 3.1, if we investigate some attacks targeting Mac operating system, then we can add a node *Mac* to Figure 3.1. As another example, if we investigate some attacks specific to *Windows XP* or *Windows 98*, then we can add two nodes *Windows XP* and *Windows 98* as the child nodes of *Windows* in Figure 3.1. The new resource tree after the above expansion is shown in Figure 3.8.

Resource tree expansion helps us easily accommodate the investigation of more alert types, without re-building resource trees from scratch, which is highly desirable.

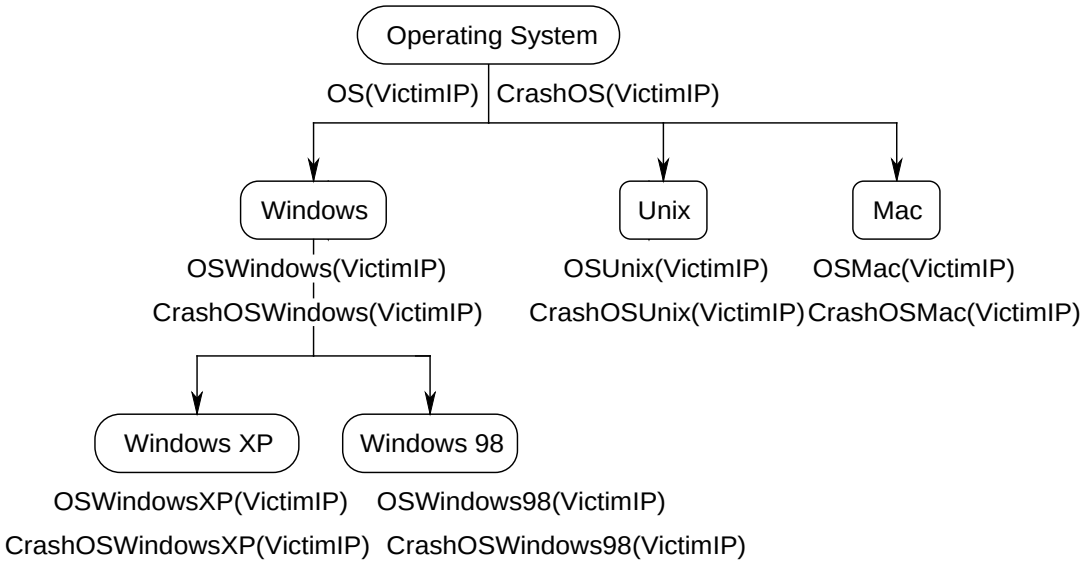


Figure 3.8: Expansion of Operating System Resource Tree

3.1.4 Specifying Prerequisites and Consequences using Resource Trees

To specify the prerequisites and consequences for various attacks, users are required to understand the attacks first. Usually, the detailed information about attacks is provided by alert descriptions (e.g., alert descriptions from *Snort* and *RealSecure*). Users have to go through the corresponding alert description in order to specify the prerequisites and consequences of attacks. Based on the alert description, users first identify the appropriate *resource* trees for both the prerequisite and consequence, then parse down to the desirable nodes in the resource trees and choose the appropriate pre-defined predicates from the corresponding nodes, next replace the variables in the selected predicates with the corresponding alert attributes, and finally put the appropriate predicates into the prerequisite and consequence.

Here we give an example on how to specify the prerequisite and consequence of *FTP CEL overflow attempt*. We first give *Detailed Information* and *Attack Scenarios* provided by the alert description from Snort website [3].

"*Detailed Information: The IBM AIX 4.3.x FTP daemon contains a buffer overflow vulnerability. An attacker can send an overly long string in the CEL command, causing a buffer overflow condition and allowing the attacker to execute arbitrary code.*"

"*Attack Scenarios: An attacker sends a suspiciously large amount of data to the FTP server in the CEL command, causing a buffer overflow condition. The attacker can then execute arbitrary code to obtain root privileges.*"

Based on the above description, it is clear that *FTP CEL overflow attempt* exploits a vulnerability in an *FTP* service to launch buffer overflow attacks, and the attacker may gain root privilege to the victim host if the intrusion succeeds. In order to define the prerequisite and consequence for this alert type, we first identify the related resource trees involved in the attack. From the alert description, it is clear that the attack involves *AIX 4.3.x FTP daemon* and the *root privilege*, where the former is for the prerequisite and the latter is for the consequence. For *AIX 4.3.x FTP daemon*, we choose "Service" resource tree (Figure 3.2), and for *root privilege*, we choose "Privilege" resource tree (Figure 3.3).

Next, we traverse both resource trees to choose desirable nodes. A general guideline to choose appropriate nodes is that we need to choose the most specific node related to the alert type. By most specific node, we means the finer-granularity nodes in the resource classification. This guideline can help us reduce false correlation. Thus, in "Service" resource tree, we choose node "AIX FTPD"[3], and in "Privilege" resource, we choose node "RootAccess".

After node selection, we are required to choose the appropriate predicates associated with the corresponding nodes. For node "AIX FTPD", since *FTP CEL overflow attempt* do not require any DoS attacks as its prerequisite, we can simply choose predicate *AIXFTPD(IP, Port)*. For node "Root Access", predicate *RootAccess(VictimIP)* is chosen.

After appropriate predicates are selected, we need to replace the predicate variables with the alert attributes. Assume for *FTP CEL overflow attempt*, four attributes are reported: {SrcIP, SrcPort, DestIP, DestPort}. Considering that in predicate *AIXFTPD(IP, Port)*, variable *IP* is the victim host IP, which corresponds to *DestIP* reported in the alert, and variable *Port* is the vulnerable service port, which corresponds to *DestPort* reported in the alert, we reformat this predicate as *AIXFTPD(DestIP, DestPort)*, and put it into the prerequisite of the alert type. Similarly, predicate *RootAccess(VictimIP)* is re-formated as *RootAccess(DestIP)*, and put into the consequence of the alert type.

Though the example *FTP CEL overflow attempt* attack is simple, the above proce-

[3]Note that we can also further expand node "AIX FTPD", adding some child nodes with different versions of AIX FTPD daemons. However, to simplify our discussion, here we assume that we do not expand the node, then node "AIX FTPD" should be chosen because it is the most specific node related to the alert type.

dure demonstrates the process of specification of prerequisites and consequences for different attacks. More sophisticated attacks should also follow this procedure.

Chapter 4

Experimental Results

To test the effectiveness of our proposed method, we conducted both usability study and comprehensiveness study. In the following two subsection, we report them respectively.

4.1 Usability Study

To test the ease-of-use of our approach, we conducted a usability experiment with a group of 25 users. These users are graduate students from various fields of study. They were selected from different majors to test how effective is our method for users with or without knowledge of network security. Table 4.1 enumerates the various academic fields of the users who took part in the experiment [1]. Out of the 25 users 12 are doing research in security related fields and 13 students were not familiar with intrusion detection or security terminologies.

We provided each user with the description of our resource tree based method, 20 different alert types (in the form of *Snort* alert types) we randomly selected from some alert data sets, and the alert description of the corresponding alert types. Users are required to first understand our method, and then write the prerequisites and consequences for the

[1] In this table, CS stands for Computer Science, ECE is Electrical and Computer Engineering and IE is Industrial Engineering. The PhD students in computer science department are doing research in fields related to security.

Degree	Major	#Users
PhD: Security	CS	10
PhD	ECE	1
PhD	Textiles	1
Masters	CS	4
Masters	ECE	3
Masters	Architecture	1
Masters	Textiles	1
Masters	Management	1
Masters	Humanities	1
Masters	IE	2

Table 4.1: User Background in Our Experiment

assigned 20 alert types. The description of the methodology had examples of how to use different resource trees to model the prerequisites and consequences.

Users were asked to report the time they took to complete the exercise. To measure the correctness of the user responses we compared it to our predefined set of prerequisites and consequences for those alert types. These detailed information for each user is listed in section 4.1.1.

Based on the experimental result from each user, we performed a set of analysis. In our first analysis, our goal is to study the relationship between the user background and the correctness of the specification. The result of this analysis is shown in Figure 4.1. From Figure 4.1, we observe that there is no significant difference in terms of correctness between users of different academic backgrounds.

We also computed the average correctness for security and non-security users. The correctness for security users is 97.5%, and the correctness for non-security users is 96.9%. These results confirm that our method is applicable for both security experts and non-security users.

In our second analysis, our goal is to study the time distribution to complete the exercises among different users. Figure 4.2 shows the relation between the time and the number of users. From Figure 4.2, we observe that the time taken to complete the exercise do vary. The maximum exercise time is more than the double of the minimum time. In addition, we also computed the average time taken to complete the exercise, which is 50 minutes. This means on an average, a user needs 2 minutes and 30 seconds to complete one

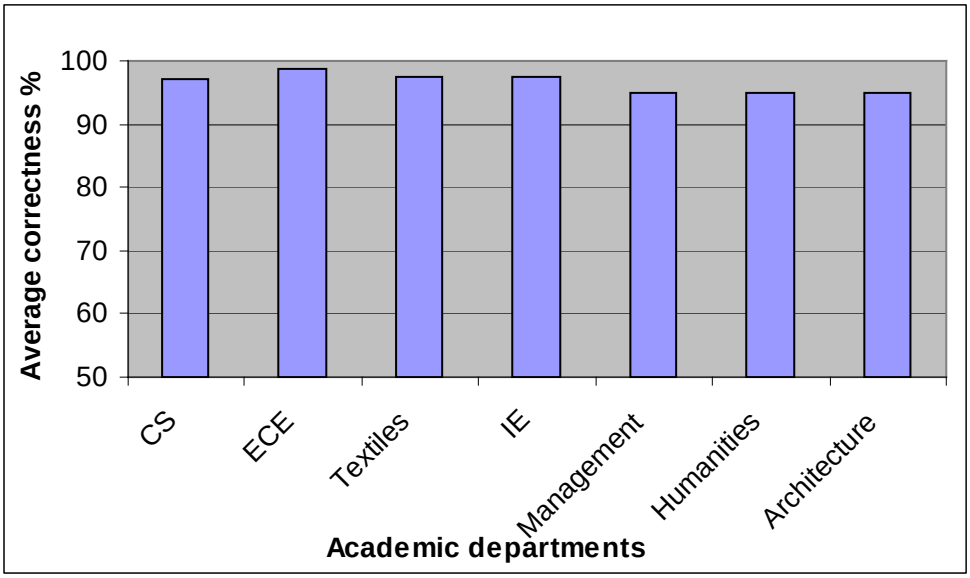


Figure 4.1: Correlation between User Background and Correctness

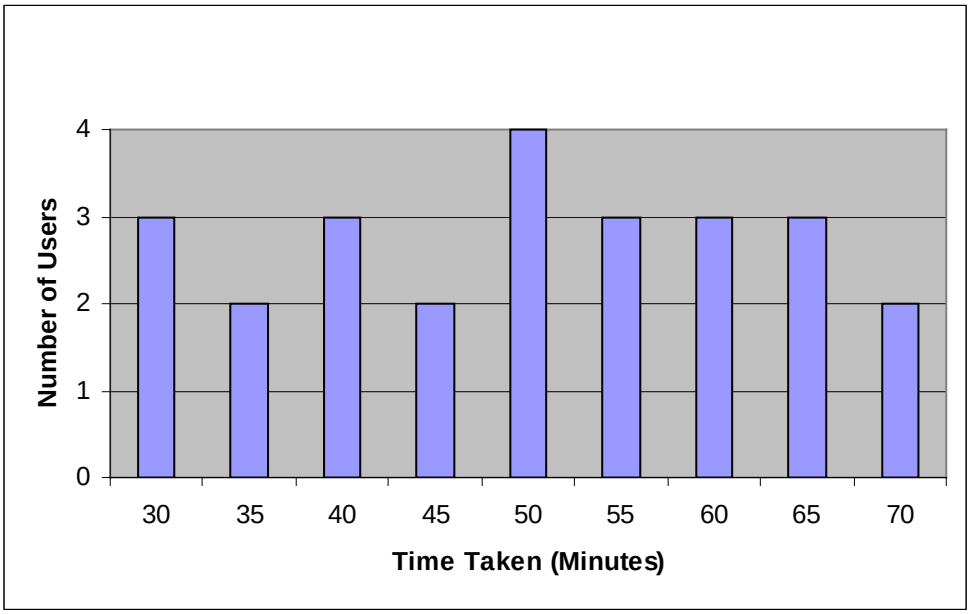


Figure 4.2: Time used by Different Users

alert type, which is quite efficient.

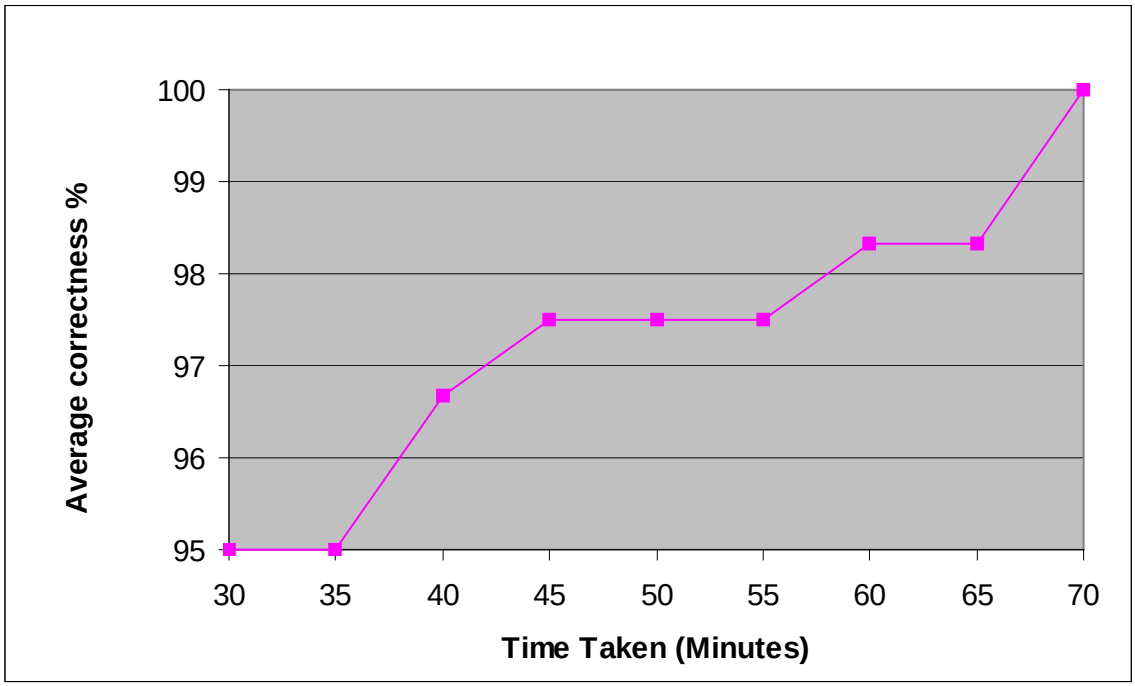


Figure 4.3: Correlation between Time and Correctness

In addition, we also computed the average time taken to complete the exercise for security and non-security users, which is 42.5 and 56.9 minutes for security and non-security users, respectively. We conjecture the reason why the security users took less time is that their background on security can help them understand attack signatures in less time.

In our third analysis, our goal is to study the relationship between the time taken to complete the exercise and the correctness of the specification of prerequisites and consequences. The result is shown in Figure 4.3. In Figure 4.3, we observe that the more time a user spends, the better results (in term of correctness) he/she can achieve. This tells us when specifying prerequisites and consequences, we need to put more time in understanding the attack signature to achieve better correctness.

In our fourth analysis, our goal is to study what mistakes users usually make. We identified three different types of errors: *missing predicate errors*, *extra predicate errors* and *misc errors*. Assume a correct prerequisite or consequence has n predicates. If a user identifies $[1, n-1]$ correct predicates, we call it missing predicate error; if a user identifies n correct

predicates as well as some other incorrect one(s), we call it extra predicate error; otherwise we call it misc error. Among all 25 users, they made 14 mistakes in total where there are 13 missing predicate errors and 1 extra predicate error. The detailed information about the mistakes users made can be found in section 4.1.1. To counter with missing predicate errors, identifying resources correctly both in prerequisites and consequences is crucial. To identify resources correctly, it requires the users spend more time on understanding attack signatures.

4.1.1 Detailed Data in Usability Study

In total there are 25 users involved in our usability study. Here, we list the detailed information about each user in the experiment including the educational background, the time to complete the exercise, the correctness percentage, the number of missing predicate errors, the number of extra predicate errors, and the number of misc errors. From the information shown in table 4.2 it is evident that the majority of the mistakes that are made are missing predicate errors. The reason for this behavior is that the users did not consider multiple resource trees for modelling the alert types. They just selected predicates from a single resource tree instead of considering multiple resource trees where required. For example, for the alert type *NT NULL session* the user just reported one of the predicates *Netbios(DestIPAddress,DestPort)* for the prerequisite part using the *Service* resource and missed the predicate *OSWindows(DestIPAddress)* in the *Operating System* resource tree. For the alert types where multiple predicates are required the user should consider multiple resource trees independently. The user should be able to identify such alert types from the alert description. Only one of the users made an extra predicate error, but this was only one instance and cannot be considered as a general behavior and is attributed to the user's misinterpretation of the alert description.

Table 4.2: Detailed Information about Each User in Our Experiment

Degree	**Major**	**Time**	**Correctness**	**#Missing**	**#Extra**	**#Misc**
PhD: Security	CS	30 min	100 %	0	0	0
PhD: Security	CS	30 min	90 %	2	0	0
PhD: Security	CS	40 min	100 %	0	0	0
PhD: Security	CS	45 min	100 %	0	0	0
PhD: Security	CS	70 min	100 %	0	0	0
PhD: Security	CS	60 min	100 %	0	0	0
PhD: Security	CS	35 min	90 %	2	0	0
PhD: Security	CS	30 min	95 %	1	0	0
PhD: Security	CS	35 min	100 %	0	0	0
PhD: Security	CS	40 min	100 %	0	0	0
Masters	CS	45 min	95 %	1	0	0
Masters	CS	50 min	100 %	0	0	0
Masters	CS	70 min	100 %	0	0	0
Masters	CS	40 min	90 %	1	1	0
PhD	ECE	55 min	100 %	0	0	0
Masters	ECE	65 min	100 %	0	0	0
Masters	ECE	50 min	95 %	1	0	0
Masters	ECE	60 min	100 %	0	0	0
PhD	Textiles	50 min	100 %	0	0	0
Masters	Textiles	60 min	95 %	1	0	0
Masters	Architecture	50 min	95 %	1	0	0
Masters	Management	55 min	95 %	1	0	0
Masters	Humanities	65 min	95 %	1	0	0
Masters	IE	65 min	100 %	0	0	0
Masters	IE	55 min	95 %	1	0	0

4.2 Comprehensiveness Study

In order to see the comprehensiveness of our resource tree based approach, we applied our method to all the different *Snort* alert types and several *RealSecure* alert types. There are currently 3301 alert types defined in the *Snort* signature database, which is categorized into 30 classes. We analyzed each alert type and defined prerequisites and consequences for them using our resource tree based approach.Table 4.3 shows the total number of alert types in each Snort class. Actually, one of the critical issues in our method is the completeness of resource trees. Usually it is not possible to come up with a complete list of all possible resource trees. However, this problem can be mitigated by expanding existing resource trees, or by building new ones if necessary. This study shows that our methodology is comprehensive for all the *Snort* alert types with a few exceptions discussed in the following section.

4.2.1 Description of the Snort alert types analyzed

In the comprehensiveness study we analyzed all the 30 *Snort* class types and specified prerequisites and consequences for all the 3301 currently defined in the *Snort* signature database. This section describes the different alert types present in the various *Snort* classes and how we used our methodology to define prerequisites and consequences for those alert types.

Here we discuss the top ten *Snort* class types that contribute 93.5% of the total alerts. The alerts belonging to the other Snort class types are also modelled in similar manner.

Attempted-admin: This class of *Snort* contains alert types that are aimed at gaining administrative privileges on the victim machine. In this type of attack the attacker tries to exploit the various vulnerabilities present in the different services, operating system or applications running on the victim machine. The alert types in this category are generally defined using the *Service* , *Operating System* and *Application* resource trees for the prerequisite part and *Privilege* resource tree for the consequence part of the alert. An example for this Snort class is *FTP RNTO overflow attempt*. In this attack the attacker tries to gain root access to the server by exploiting a buffer overflow vulnerability in the University of Washington FTP daemon, *WUFTPD*. To define this alert type we need to

Classtype	**# Total attack types**
attempted-admin	346
attempted-user	421
web-application-attack	337
attempted-dos	93
attempted-recon	344
misc-attack	126
rpc-portmap-decode	90
web-application-activity	500
misc-activity	353
protocol-command-decode	477
shellcode-detect	33
successful-admin	9
successful-user	1
trojan-activity	13
unsuccessful-user	11
bad-unknown	63
successful-dos	0
denial-of-service	6
non-standard-protocol	7
successful-recon-limited	2
suspicious-filename-detect	24
successful-recon-largescale	0
unusual-client-port-connection	0
icmp-event	0
suspicious-login	18
system-call-detect	5
network-scan	3
not-suspicious	13
string-detect	2
unknown	2

Table 4.3: Attack Categories for Snort Attack Signatures

use the *Services* resource tree to depict the WUFTPD service running on the victim as the prerequisite and also the *Privilege* resource tree to denote the root access to the server as the consequence. Therefore, this alert type will have *WUFTPD(DestIP,DestPort)* as the prerequisite and *RootAccess(DestIP)* as the consequence. Different alert types in this category are defined in a similar manner.

Attempted-User: The alert types in this Snort class are similar to the ones in the attempted-admin, the difference being that the alert types in attempted-user Snort class are mainly aimed at gaining local user access to the victim host. These alert types also exploit the vulnerabilities in the different services or applications running on the host. An example of an alert type in this category is *RSERVICES rsh echo + +*, where the attacker tries to gain remote access to the victim machine using the *RSH* service running on the UNIX operating system on the victim machine. To model this alert type we again need the *Service* and *Privilege* resource trees along with the *Operating System* resource tree. The prerequisite being *RSH(DestIP,DestPort)* and *OSUNIX(DestIP)* denoting RSH service running on UNIX platform on the destination (victim) machine. The consequence here is *LocalAccess(DestIP)* as the attacker gains normal user access to the system.

Web-application-attack: This is another very common type of attack. This category contains alerts for attacks that are aimed at exploiting the vulnerabilities in the different web applications and the web servers. The consequences of these alert types range from gaining access to the victim machine to causing a denial of service of the vulnerable application or server. For example, in the alert type *WEB-IIS header field buffer overflow attempt* the attacker exploits the buffer overflow vulnerability present in the Internet Information Sserver (IIS web server). The attacker can eventually gain root access or even stop the IIS server from running thus resulting in a denial of service. The prerequisite for this alert type is *IISWeb(DestIP,DestPort)* denoting the IIS web server running on the victim machine and the consequence can be denial of service of the IIS web server or gaining root access , denoted by *DoSIISWeb(DestIP,DestPort)* and *RootAccess(DestIP,DestPort)*. Thus, the Service and the Privilege resource trees are used to model the prerequisites and consequences for this alert type.

Attempted-dos: The alert types in this Snort class are related to attempted denial of service on victim machines. These alert types also denote the presence of various distributed denial of service *zombies* in the network. These zombies usually use a tiered structure of compromised hosts in which there is a *master* (client) and several *slaves*

(handler), where master controls the slaves and causes a DDoS on a victim machine in the network. For example, the alert type *DDOS mstream client to handler* denotes that the compromised source is the *mstream zombie* client and compromised destination is the mstream handler and the client communicates with the handler to launch a DDoS attack. This alert type can be modelled using the *Malicious Software* resource tree and also the *Privilege* resource tree. The prerequisite would be *Access(SrcIP)*, *Access(DestIP)* as the source and destination are compromised and *MstreamClient(SrcIP)* and *MstreamHandler(DestIP)* denoting the presence of the mstream client and mstream handler on the source and destination machines respectively. The consequence of this alert type is that it denotes the presence of the mstream zombie in the network. This is represented by the predicate *MstreamZombie*. There are no variables associated with this predicate because it is related to a distributed attack and the clients are distributed over various machines where intrusion detection may not be able to detect all of them.

Attempted-recon: These alert types denote attempted information leakage from the victim host. In these alert types the attacker tries to gather various information about the victim host. The information ranges from learning what machines are running in the network to probing for specific services running on the host machines. These alert types also involve reading specific files on the victim machine containing sensitive information such as user accounts and passwords etc. For example, in the alert type *FINGER search query* the user exploits the vulnerability in the Finger service daemon running on the victim machine to read the file containing the user accounts. For this alert type, *Finger(DestIP,DestPort)* represents the prerequisite denoting the presence of the finger daemon on the victim destination IP address. Also, the consequence is reading the sensitive file denoted by *ReadFile(DestIP)* using the *FileSystem* resource tree. In this case the specific name of the file is not mentioned in the description so we use the general predicate *ReadFile(VictimIP)* else we could have used the specific file name in the predicate.

Misc-attack: These alert types range from gaining access to the victim to causing denial of service or information leakage. Various services, applications and operating systems are exploited by the attackers to meet their goal. For instance, in the alert type *EXPLOIT Veritas backup overflow attempt* the attacker exploits a buffer overflow vulnerability in the Veritas Back Exec Agent Browser to gain access to the victim machine. This alert type is modelled using the *Application* resource tree and the *Privilege* resource tree. The prerequisite is the presence of the vulnerable application on the victim denoted by

VeritasBackExecAgentBrowser(DestIP) and the consequence is gaining access to the victim denoted by *LocalAccess(DestIP)*.

RPC-portmap-decode: The alert types in this Snort class are mainly aimed at probing the various RPC services on the victim machine. Through these attacks the attacker tries to learn what RPC services are running on which port of the victim machine. These attacks are prelude to further attacks that might exploit the vulnerabilities in the services to compromise the system. For example, in the alert type *RPC portmap rwalld request TCP* the attacker queries the *Portmapper* service running on a *Solaris* host to learn the port number on which *rwalld* service is running. The prerequisite of this alert type are *Portmapper(DestIP,DestPort)* and *OSSolaris(DestIP)* and the consequence is that the attacker learns that the rwalld service is running on the victim machine represented by *rwalldService(DestIP)*.

Web-application-activity: This is a group of alert types that aims at gaining access to potentially vulnerable web applications and through them getting access to the system and launching further attacks. These vulnerable applications can also lead to the compromise of information on the victim machine. For instance, *Hyperseek 2000* is a web application that has a directory traversal vulnerability. An attacker can exploit this to view hidden files and directories on the web server. The alert type *WEB-CGI HyperSeek hsx.cgi access* utilizes this vulnerability by accessing the file *hsx.cgi*. The prerequisite for this alert type is the vulnerable application *HyperSeek2000(DestIP)* and the consequence is the read access to the file *Readhsx.cgiFile(DestIP)*.

Misc-activity: The alert types in this Snort class have a wide range of consequences. These range from system compromise by trojans (backdoors) to buffer overrun in vulnerable applications and services. Some are also probing attacks. These alert types are indicative of malicious activities in the network and not all attacks are successful but still they alert the administrator about the attempts. For instance, the alert type *BACKDOOR HackAttack 1.20 Connect* indicates that an attacker is trying to exploit a machine that is infected with the trojan *HackAttack 1.20* and gain root access to that victim machine. This alert type denotes the trojan activity in the network and can be modelled using the *Malicious Software* and *Privilege* resource trees. The prerequisite would be *HackAttack1.20Trojan(DestIP)* denoting the trojan infected victim machine and the consequence would be gaining root privileges on the victim machine represented by *RootAccess(DestIP)*

Protocol-command-decode: These alert types exploit the vulnerabilities in the

implementation of various protocols. These alert types exploit the services that are using these protocols and aim at gaining access to the machine and in certain cases also involve information theft. For instance, there is a buffer overflow vulnerability in the implementation of *FTP (File Transfer Protocol)* in the *3CDaemon FTP server)*. This allows the attacker to gain unauthorized root access to the server and also cause a denial of service. In the alert type *FTP command overflow attempt* the attacker can gain root access or cause a denial of service on the server by sending a FTP command longer than 400 bytes. This alert type can be modelled by using the prerequisite *3CDaemonFTP(DestIP,Port)* denoting the running of 3CDaemon FTP server on the victim machine and the consequence will be *RootAccess(DestIP)* and *DoS3CDaemonFTP(DestIP,Port)* denoting root access to the server and denial of service of the FTP daemon.

4.2.2 Exceptions in Snort alert types

In our study of the *Snort* alert types we found certain alert types that could not be completely modelled using our methodology. We describe these exceptions in the following discussion.

Alert types with no prerequisites: There are certain alert types whose prerequisite cannot be defined using our methodology. For instance, *ICMP PING* falls into this exception. This alert type is indicative of a scanning attack that reveals that a specific machine is live in the network. Although this alert type has a consequence *LiveHost(DestIP)* which denotes that the specific machine is running in the network but there is not a definitive prerequisite for the alert type. The attack is aimed at scanning the complete network and therefore cannot be denoted by a specific resource tree or even a specific IP address.

An alternative way for this is to have a network resource tree that represents the network and then have a predicate *ScanNetwork* to represent the scanning of the network. But this will be a very specific case. Also, this predicate will not benefit the correlation process as this predicate cannot appear in the consequence part of any attack and will only appear in the prerequisite part. Further, these alert types are very small in number 29 out of a total 3301 alert types. Thus, we avoid defining a new resource tree and unnecessarily increasing the number of resource trees. We leave such alert types with no prerequisites as only their consequence *LiveHost(VictimIP)* is useful for correlation process.

Another type of alerts that do not have prerequisites defined are the ones that are

generated by spurious network traffic. These alerts are generated because of specific bit pattern seen in the traffic or use of unassigned or reserved protocols. These alert types indicate either system compromise or network reconnaissance. For example, *BAD-TRAFFIC ip reserved bit set* is generated when reserved bits of the packets in the network traffic are set. This can indicate a covert channel communication resulting in system compromise attempt and thus have the consequence *Access(DestIP)*. Again, this alert type does not have prerequisites as cannot be described by any of the resource trees and also no specific information about what is required for the intrusion to be successful is described. These alert types can only be prelude to further attacks and as discussed above, only their consequence is required for the correlation process. These alert types are 15 in number. For these 44 alert types we only define the consequences.

Alert types without description: Besides the above discussed alert types, there are 171 alert types that do not have any description. To identify the prerequisites and consequences for any alert type, we first need to study the alert description and in the absence of any documentation for an alert type we cannot define prerequisites and consequences.

Except for these 171 alert types out of the total 3301 we could define prerequisites and consequences for all the remaining 3130 alert types using our resource tree based approach.

4.2.3 Conclusion of the Comprehensiveness study

We described different alert types in the top ten *Snort* class types. We also gave specific examples of how to define the prerequisites and consequences for the different alert types in each Snort class. We have defined prerequisites and consequences for all the 3130 different types of *Snort* alert type, including exceptions, and formed a complete knowledge base [21] for these alerts. As described in section 2.1 an alert type is modelled into a hyper-alert type for the purpose of correlation [21]. The prerequisites and consequences are the main component of these hyper-alert types. A collection of all the hyper-alert types together with all the predicates and the implication relationships is called a knowledge base. This knowledge base acts as a dictionary of attacks for the correlation process [21]. Thus, by defining the complete knowledge base for all the *Snort* alert types, we have made the task of the user of the correlation tool TIAA [22] very simple and efficient. The user just

has to use our predefined knowledge base along with the set of *Snort* alerts to generate correlation graphs using the tool. The users can also define their own knowledge base for any new attacks not covered in our knowledge base using our proposed resource tree based approach and taking the predefined knowledge base as an example. Furthermore, the extensible nature of the resource trees allow the users to model different types of attacks which are not covered by the predefined resource trees.

4.3 Experiment with attack scenario specific datasets

In chapter 3 we proposed a resource tree based approach for defining the prerequisites and consequences for different alert types. We also performed a usability study with real users to analyze the ease of use of our approach in section 4.1. The comprehensiveness study with more than 3000 *Snort* alert types in section 4.2 shows that our method can be effectively used for modelling all of those alert types with few exceptions.

We now want to validate our methodology by using it for alert correlation with actual datasets. We have modelled various individual alerts using our method. By using these predefined prerequisites and consequences with actual attack scenarios and alert data for correlation we want to verify the effectiveness of our methodology in facilitating alert correlation. Here we want to see how efficient is our methodology in modelling prerequisites and consequences for actual alert correlation purposes. For the purpose of alert correlation we modelled all the 3257 *Snort* alert types in the form of *hyper-alert types* [21]. As discussed in section 2.1 an alert type is represented as a hyper-alert type which is a triple (*fact,prerequisite,consequence*). We defined the prerequisites and consequences for all the alert types using our resource tree based methodology. All these hyper-alert types along with the list of predicates used in the prerequisites and consequences, and all the implication relationships between the predicates formed a complete knowledge base [21] for *Snort* alert types. A sample knowledge base used in our experiments is shown in appendix B.1.

We then used two different datasets, discussed in section 4.4, along with TIAA [22], a toolkit for alert correlation, to perform alert correlation and analyze the results.

We also note that similar experiments were carried out by Ning et al. [21] with the DARPA 2000 dataset [17]. The major difference between their experiments and this one is that they had a different knowledge base, the set of prerequisites and consequences. They did not distinguish between different services and also did not have any way of representing different file object operations. Essentially, they did not have a systematic way of defining the prerequisites and consequences, and their method was ad-hoc in nature.

We use the same dataset but with the knowledge base defined using our resource tree based approach. Besides that we also use the treasure hunt dataset, discussed in section 4.4.4, to validate our results. Section 4.4.3 gives the comparison between the results of the previous experiment with our experiments. We use different datasets to show that our defined set of prerequisites and consequences can be effectively used for building attack

scenarios.

To test the effectiveness of the alert correlation results we used the measures of completeness and soundness. Soundness implies if we can derive something then it has to be true. This means that we cannot derive something false. On the other hand completeness implies if something is true then it can be derived. In our experiments for soundness, we see that the alerts that appear in the correlation graph should be true alerts. Any false alerts that are correlated reduce the soundness measure. True alerts can be identified by analyzing the known attack scenario or by analyzing the traffic data. In completeness we look for those true alerts that are missing from the correlation graph. Any missing true alert will reduce the completeness measure. Completeness evaluates how well we can correlate related alerts while soundness evaluates how correctly the alerts are correlated. They are formally defined as follows[21]:

$$\text{Completeness} = \frac{\#correctly\ correlated\ alerts}{\#related\ alerts}$$

$$\text{Soundness} = \frac{\#correctly\ correlated\ alerts}{\#correlated\ alerts}$$

The correctly correlated alerts are the total number of true alerts that are correlated. These are the alerts that appear in the attack scenario or present in the actual network traffic. Related alerts are the number of true alerts that are missing from the correlation graph in addition to the total number of correctly correlated alerts. Correlated alerts in the soundness measure are the total number of alert nodes that appear in the correlation graph.

4.4 Datasets

We experimented with two different datasets. The first dataset was the DARPA 2000 Intrusion detection scenario specific dataset[17]. The second one was the result of a class exercise in a graduate security class taught at UC Santa Barbara [32]. These dataset represent different attack scenarios and using the correlation results for these we show that our methodology can be efficiently used to define prerequisites and consequences for different alert types.

4.4.1 DARPA 2000 dataset

This dataset contained two different attack scenarios. The first scenario (LLDOS1.0) involved different attack phases over multiple networks. The attacker starts by probing the network for hosts running *sadmind* service, then it exploits the Solaris *sadmind* vulnerability on those machines and then further installs DDoS Trojan on these machines and launches the DDoS (distributed denial of service) attack. We analyzed two different datasets for the above scenario involving the inside traffic of the network and the DMZ network traffic. The second scenario(LLDOS2.0.2) was similar to the first one and only differed in the manner the attacker probed the network. The attacker in the second scenario is more stealthy than the first one. For the second scenario also we analyzed both DMZ and inside traffic data.

4.4.2 Experiment Details

To analyze the tcpdump files for the dataset we used Snort [27] as the intrusion detection system. Further, we used MS SQL server as the database backhand to store the alerts generated by Snort. We ran Snort on tcpdump files for each DMZ and inside traffic data for both the scenarios. After the alerts were generated by Snort and stored in the database, we used our predefined knowledge base for *Snort* alert types with *TIAA* [22] to perform alert correlation. The remainder of this section discusses the correlation results along with the correlation graphs.

4.4.3 Comparison with the previous experiment results

This section gives a comparison of the correlation experiments done by Ning et al.[21] and our correlation experiment with the DARPA 2000 dataset [17].

The first difference in both the experiments lies in the knowledge base that was used. In our experiments we used the knowledge base defined using our proposed methodology for all the 3257 *Snort* alert types. While in the previous experiment, Ning et al. used a knowledge base that was only specific to the DARPA dataset. As mentioned earlier their method did not have a systematic approach for defining prerequisites and consequences.

Another important difference was the intrusion detection system that was used to generate alerts from the dataset. In their experiments Ning et al.[21] used *RealSecure*[14]

as the intrusion detection system and we used *Snort*[3] for analyzing the tcpdump files and generating alerts from them.

For the rest of the discussion we would be referring to the experiment in [21] as the *RealSecure* experiment and our experiment as the *Snort* experiment to distinguish between the two experiments.

For the inside traffic in dataset LLDOS1.0 we got three disjoint correlation graphs as shown in the figure 4.4 as compared to a single graph in *RealSecure* experiment, figure 4.5. The reason for the disjoint correlation graphs is that *Snort* IDS fails to report some of the alerts that are part of the scenario. Specifically, Snort does not report the alerts related to communication of the DDoS trojans on the compromised hosts and also the final step of the attack scenario i.e. DDoS attack. But this is not the case with *RealSecure* sensor and these alerts can be seen in the correlation graph in figure 4.5. The alert *Mstrem_Zombie* in figure 4.5 depicts communication between the *DDoS Mstream* trojans on the victim machines followed by *Strem_DoS* that is the final *DDoS* attack. In our experiments we miss the last two steps of the scenario due to the failure of the IDS (*snort*) to report those alerts and therefore, it is the limitation of the IDS and not our knowledge base that makes the attack scenario in the figure 4.4 incomplete.

We are able to correctly correlate the first three steps of the scenario starting from probing machines for *sadmind service* using the *Portmap sadmind request* and *RPC sadmind UDP PING* followed by exploit of the sadmind service using *RPC sadmind UDP NETMGT_PROC_SERVICE CLIENT_DOMAIN overflow attempt* and *RPC sadmind query with root credentials attempt UDP* to gain access to the victim machine. After gaining access the attacker tries to gain remote access on other machines in the internal network which is represented by *TETLNET access* and *RSERVICES rsh root*. These attacks are shown in the RealSecure graphs, figure 4.5 as *Sadmind_Ping*, *Sadmind_Amslverify_Overflow* and *Rsh*.

Another difference that is evident is the number of nodes that are present in the correlation graphs. In the *RealSecure* graph 4.5 there are 44 distinct nodes representing hyper-alerts. On the other hand there are 87 hyper-alerts or nodes present in the *Snort* correlation graphs. The reason is that *Snort* IDS generated a lot of duplicate alerts for the various attacks and they form the distinct nodes in the correlation graph. *RealSecure* also generated duplicate events but the number was less compared to *snort* alerts.

We now compare the completeness and soundness measure for the two experiments. For the *RealSecure* experiment the soundness measure was 93.18% and the completeness

also being 93.18%. The reason for these values is the three false alerts *FTP_Syst* and two instances of *Email_Almail_Overflow* that were correlated. For our experiment using *snort* we got a 100% soundness and completeness measures for the inside traffic of scenario one. There were no false alerts in the correlation graph and also no missing true alert from the graph.

Although the correlation graph 4.4 in our experiment represents a partial attack scenario because of the limitation of the underlying IDS, but the completeness and soundness measures show that our knowledge base is correct and effective for alert correlation.

The comparison of the other correlation graphs for the two scenarios is discussed in appendix A.1

4.4.4 Treasure Hunt Dataset

The second type of dataset that we used was generated at the University of California, Santa Barbara in a Cyber Treasure Hunt competition [32]. It was an exercise carried out in a graduate security class. Two different teams of students (Alpha and Omega) were asked to perform different kinds of attacks, ranging from probing the network for live hosts to scanning machines for running services and compromising the servers. The network consisted of three different types of servers: *MySQL server* (SAA/SOO), *file server* (UAA/UOO) and *transaction server* (TAA/TOO). One of each type was assigned to each team, where "AA" suffix represents the servers for the *Alpha team* and "OO" suffix is for the *Omega team*.

We performed correlation on the alerts generated from the network traffic of both the teams *Alpha* and *Omega*. The correlation graphs and the evaluation results are discussed later in the section.

4.4.5 Correlation Results for Treasure Hunt dataset

Here too we used Snort [27] for generating alerts from the raw tcpdump files and logged them into SQL server. For both the teams Alpha and Omega we analyzed the attacks launched against the file server, MySQl server and transaction server respectively.

In the attack against the file server, the attacker first tries to find out the port on which the *mountd* service is running. After gaining this knowledge the attacker then

Dataset	Completeness	Soundness
Team Alpha	94.23%	94.23%
Team Omega	100%	100%

Table 4.4: Completeness and Soundness for treasure hunt dataset.

lists all the exported file systems on the machine running the mountd service and the users that have permissions to access those file systems. In the subsequent attack against the file server, the attacker mounts the */home* file system and gains access to the file server. This attack scenario is depicted in the graph 4.6 a, for team Alpha and for team Omega in graph 4.7 a. The only difference is that for team Omega Snort does not report the *RPC mountd TCP mount request*. Both the teams follow similar attack pattern.

The attack against MySQL server is depicted in graphs 4.6 c and 4.7 b, for team Alpha and Omega respectively. In this the attacker follows a three step attack to gain access to the server. First the attacker scans the network, *ICMP PING NMAP* to detect the active hosts. This attack is followed by the port probing attack to scan for machines running SNMP service. Once these machines have been identified then the attacker uses a vulnerability in SNMP service to gain access to the MySQL server. We further note that for the traffic for team Alpha graph 4.6 c, there are three false alerts that are correlated *DOS Bay/Nortel Nautica Marlin*. These are correlated because this requires the same port 161 as the SNMP attack *SNMP AgentX/tcp request* and targeted at the same victim IP, this attack can be used to compromise the victim host. Since it is not described in the attack scenario we count it as false alert. These false alerts are not present in the correlation graph 4.7 for the Omega team.

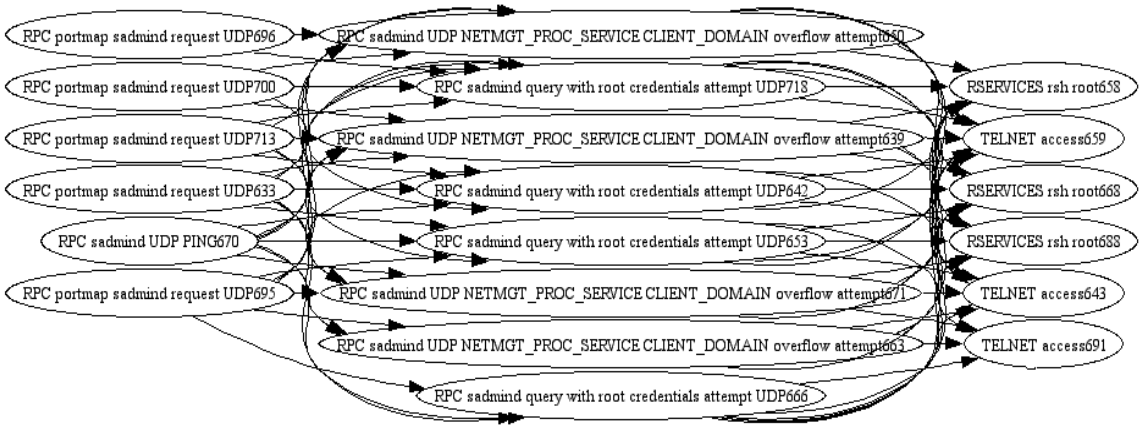
The figures 4.6 b and 4.7 c, represent the correlation graphs for the attack against the transaction server by teams Alpha and Omega respectively. This attack scenario is very similar to the one against MySQL server, in which the attacker first scans the network for live machines and then probes these machines for SNMP service port and exploits the vulnerability in SNMP to gain access to the transaction server.

We now calculate the completeness and soundness measures for this dataset to evaluate the effectiveness of the alert correlation and hence analyze the efficiency of our knowledge base. Table 4.4 shows the result of correlating the treasure hunt dataset and like DARPA 2000 dataset, the correlation process here also misses none of the true alerts and

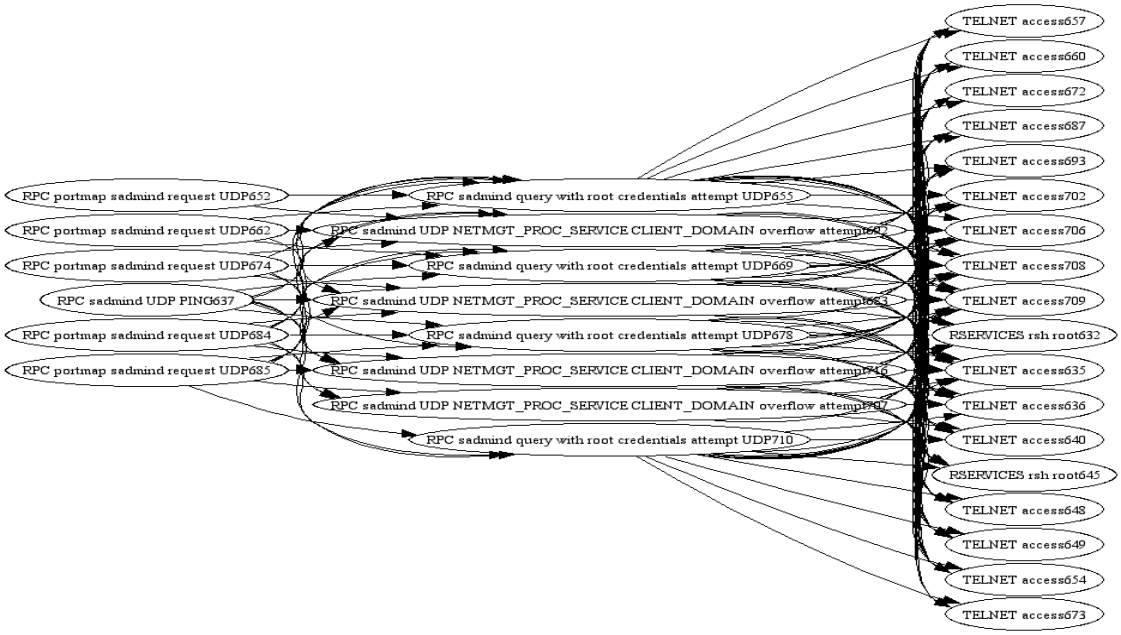
therefore we get same values for completeness and soundness. The values for completeness and soundness for both the scenarios show that we can effectively build attack scenarios using our defined set of prerequisites and consequences.

4.4.6 Conclusion of Correlation results

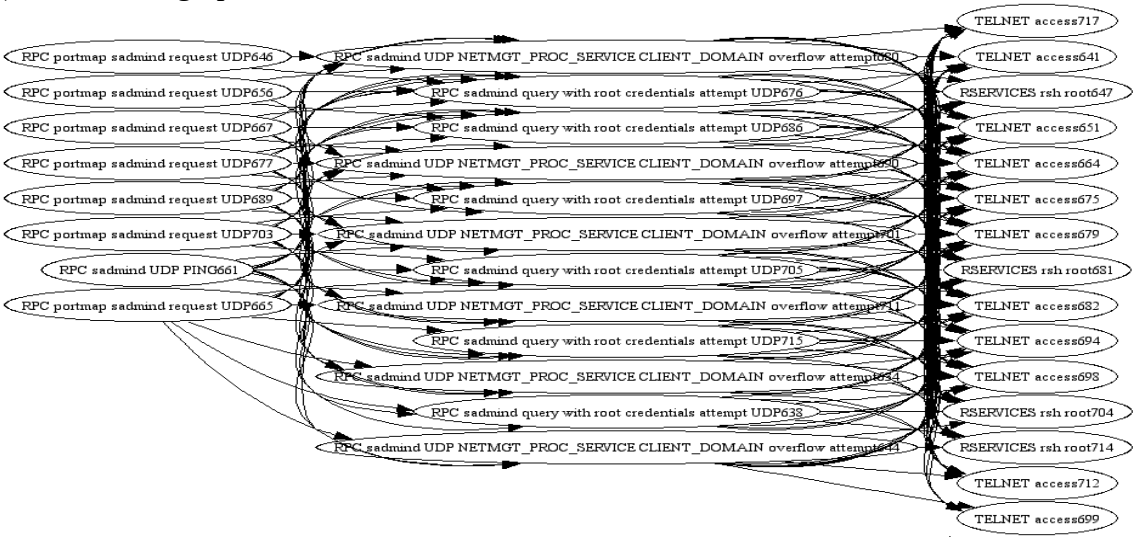
From the correlation results for the two different datasets [17] and [32] it is clear that the attack scenario is generated with quite high values of completeness and soundness. This shows that the prerequisites and consequences defined using our proposed methodology make the alert correlation process efficient. This is due to the fact that knowledgebase [21] (collection of prerequisites and consequences) is the main component of the alert correlation process. It acts as the dictionary for the alerts in the correlation process used in TIAA [22]. The correct formulation of the prerequisites and consequences reduces the false alerts from being correlated with the true alerts. Also, true alerts are not missed in the final correlation graphs. Thus, this leads to the higher values for completeness and soundness.



a) Correlation graph for IP address = 172.016.112.010



b) Correlation graph for IP address = 172.016.112.050



c) Correlation graph for IP address = 172.016.115.020

Figure 4.4: Snort experiment correlation graph for DARPA 2000 inside traffic in LLDOS1.0

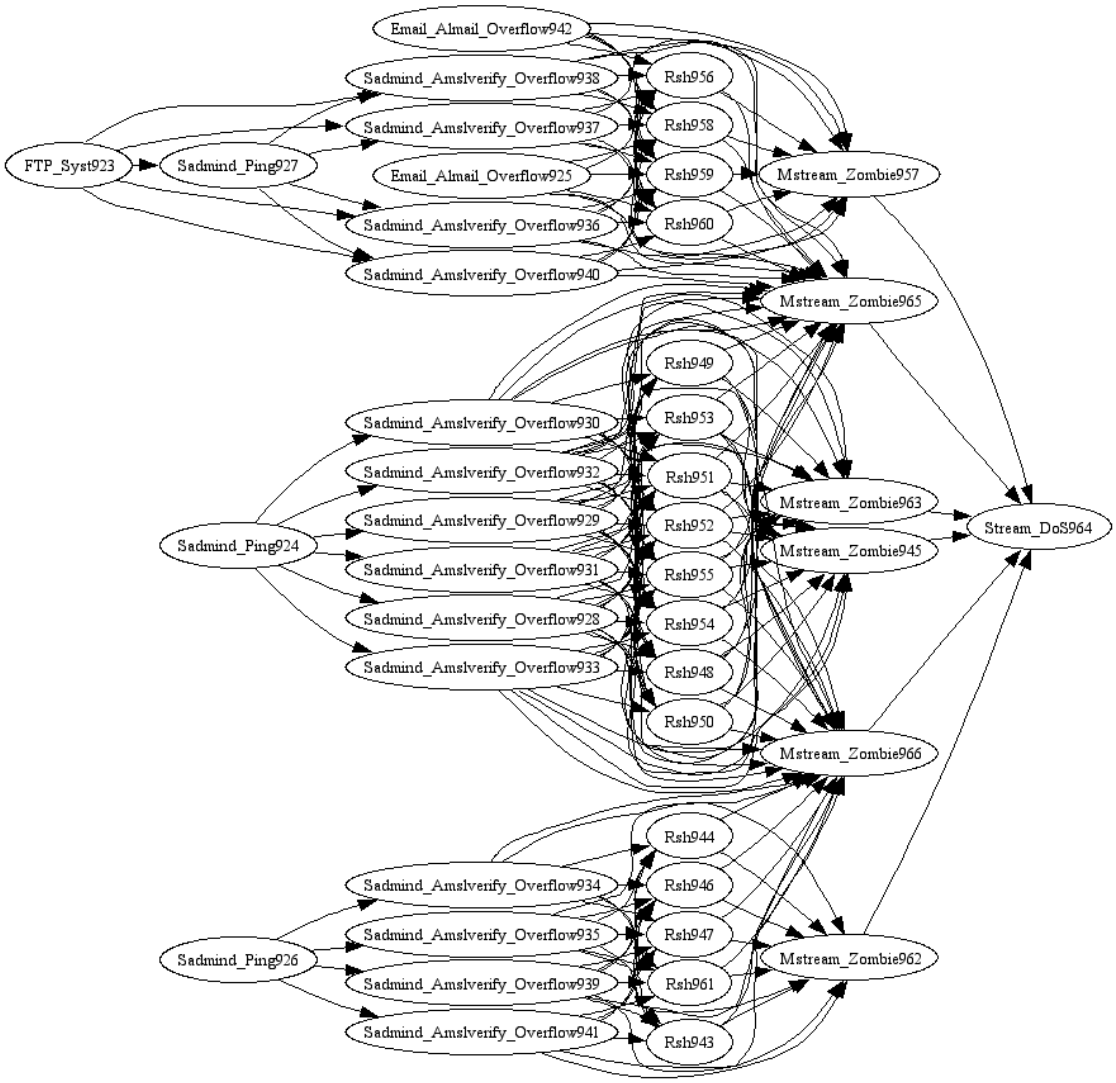
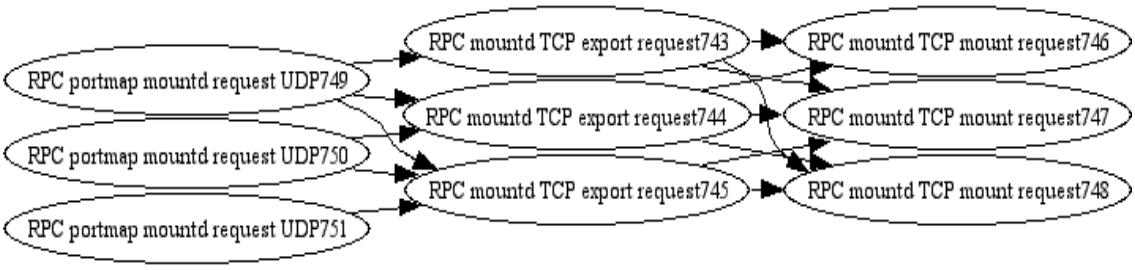
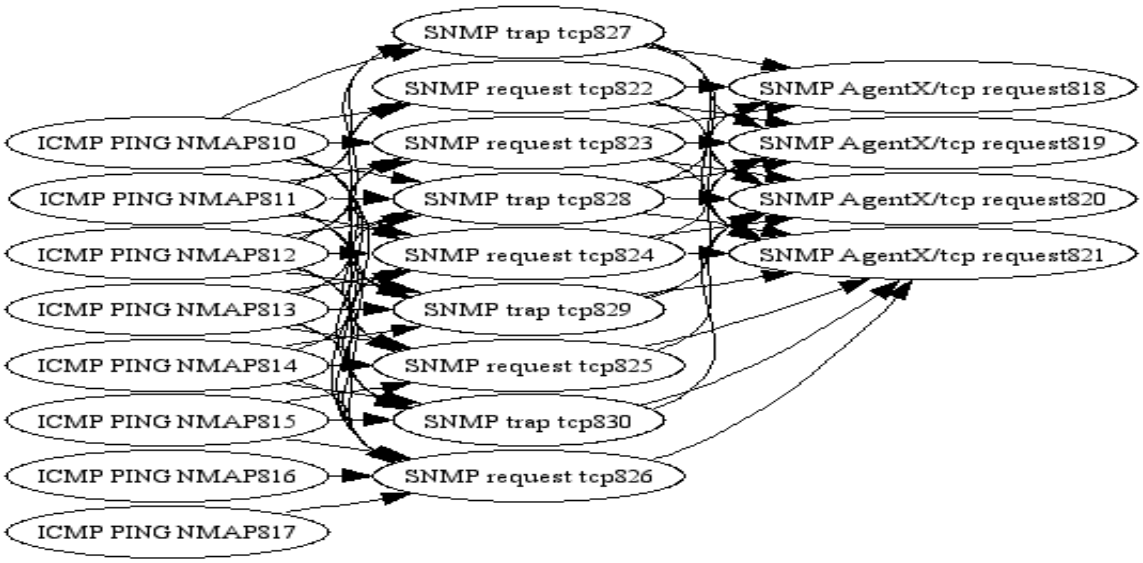


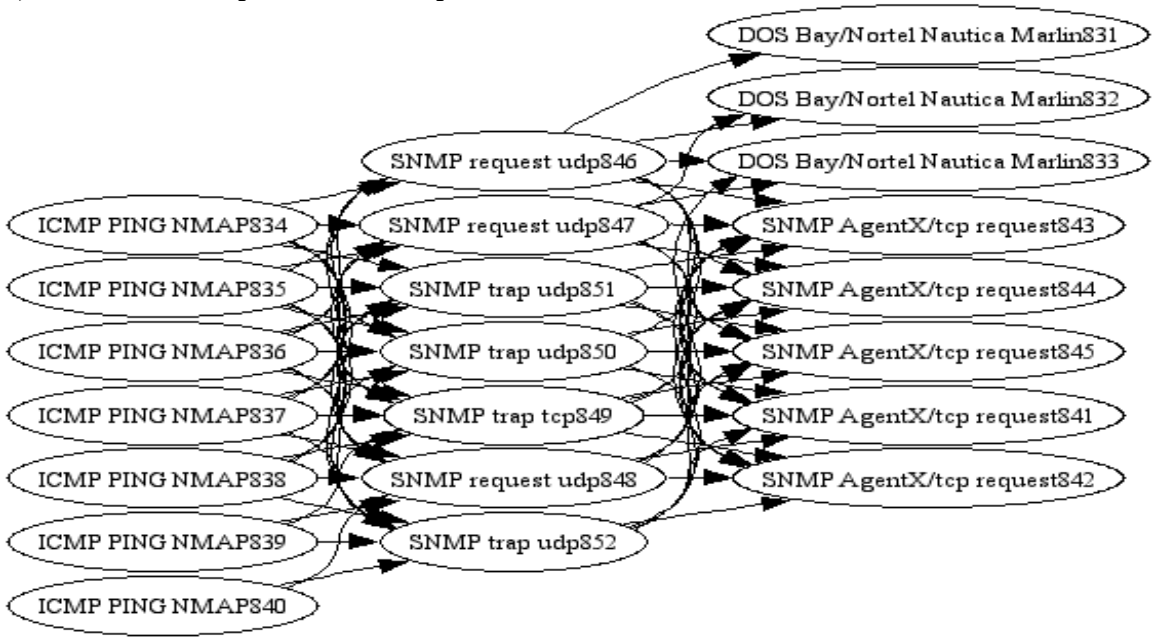
Figure 4.5: RealSecure correlation graph for DARPA 2000 inside traffic in LLDOS1.0



a) Correlation Graph for Team Alpha File Server UAA

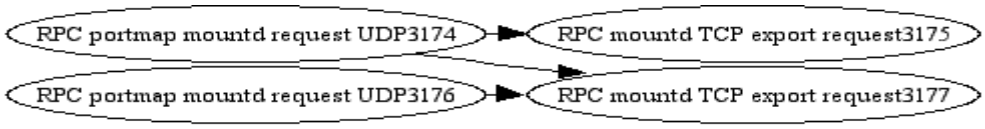


b) Correlation Graph for Team Alpha Transaction Server TAA

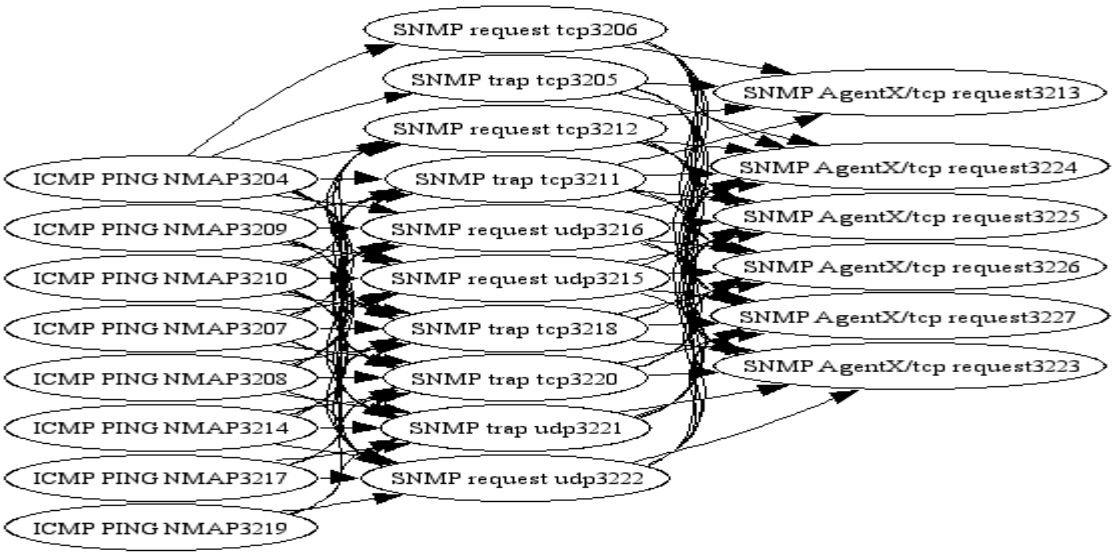


c) Correlation Graph for Team Alpha MySQL Server SAA

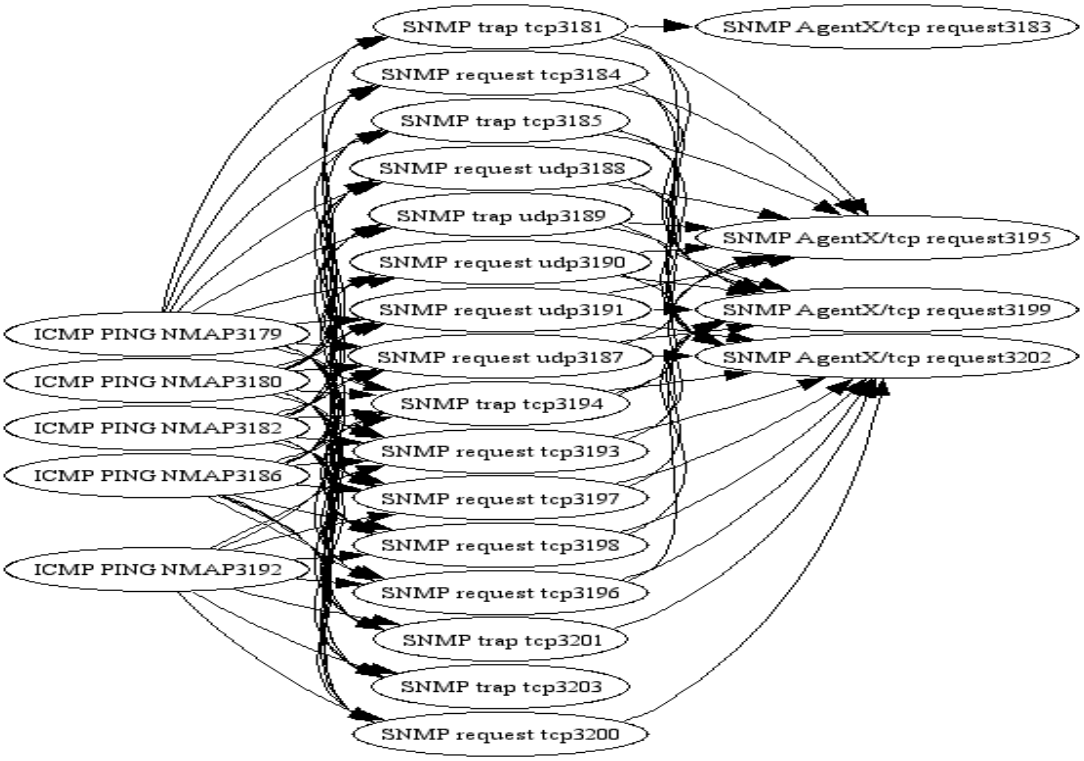
Figure 4.6: Correlation graph for team Alpha traffic in Treasure Hunt



a) Correlation Graph for Team Omega File Server UOO



b) Correlation Graph for Team Omega MySQL Server SOO



c) Correlation Graph for Team Omega Transaction Server TOO

Figure 4.7: Correlation graph for team Omega traffic in Treasure Hunt

Chapter 5

Conclusion and Future work

In this thesis we proposed a resource tree based approach to facilitate the specification of prerequisite and consequences for different alert types. It is observed that attacks can be viewed from the perspective of resources. Example resources can be network services and privileges. We are interested in discovering the necessary resources to launch an attack, and discovering the resources that can be acquired if the intrusion succeeds. We further organize resources into resource trees, where the nodes in the tree may associate with a set of predicates to denote the conditions the corresponding nodes satisfy. To specify the prerequisite and consequence of each alert type, we first look for desirable resource trees related to its prerequisite and consequence, then we parse through the resource trees to select appropriate nodes and predicates, and finally we put the appropriate predicates into the prerequisite and the consequence. This specification process is simple, efficient and systematic. The usability study and the comprehensiveness study demonstrate the effectiveness of our approach. We defined a knowledge base [21] for more than 3000 snort alert types. This acts as a dictionary of attacks for the prerequisite and consequence based technique [21]. The correlation results with different datasets further demonstrate the effectiveness of our approach in facilitating alert correlation.

There are a few open issues to be researched in the future. One critical issue in our approach is the selection of resource trees for prerequisites and consequences. We require users to choose appropriate resource trees, which may bring incorrect specification if the inappropriate resource trees are selected. Thus, effective selection of resource trees

is worth further investigation. Other issues such as appropriate classification of different resources also require further research. In the current system we specify classification of certain resources, but there should be a method to classify different resources that are added to the existing model.

Bibliography

[1] H. Debar B. Moring, L. Me and M. Ducass. M2b2: A formal data model for ids alert correlation. In *Proceedings of the 5th International Symposium, Recent Advances in Intrusion Detection (RAID) 2002*, pages 280–289, Oakland, CA, 2002.

[2] CVE Editorial Board. The common vulnerabilities & exposures. `http://www.cve.mitre.org`.

[3] Brian Caswell and Marty Roesch. Snort: The open source network intrusion detection system. `http://www.snort.org`.

[4] S. Chari and P. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and System Security*, 6(2):173–200, 2003.

[5] Thomas Toth Cristopher Krgel and Clemens Kerer. Decentralized event correlation for intrusion detection. In *4th International Conference on Information Security and Cryptology*, pages 114 – 131. Springer-Verlag, 2001.

[6] F. Cuppens. Managing alerts in a multi-intrusion detection environment. In *Proceedings of the 17th Annual Computer Security Applications Conference*, December 2001.

[7] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[8] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proc. of Recent Advances in Intrusion Detection (RAID 2000)*, pages 197–216, September 2000.

[9] O. Dain and R.K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications*, pages 1–13, November 2001.

[10] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, LNCS 2212, pages 85 – 103, 2001.

[11] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[12] Howard M. Secure systems begin with knowing your threats. `http://archive.devx.com/upload/free/Features/zones/security/articles/2000/10oct00/mh1000[1]-1.asp`, 2000.

[13] ICAT. Icat metabase. `http://icat.nist.gov/icat.cfm`. August, 2003.

[14] Internet Security Systems. RealSecure intrusion detection system. `http://www.iss.net`.

[15] K. Julisch. Mining alarm clusters to improve alarm handling efficiency. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, pages 12–21, December 2001.

[16] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, Nov 2003.

[17] MIT Lincoln Lab. 2000 DARPA intrusion detection scenario specific datasets. `http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html`, 2000.

[18] B. Morin and H. Debar. Correlation of intrusion symptoms: an application of chronicles. In *Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID'03)*, September 2003.

[19] B. Morin, L. Mé, H. Debar, and M. Ducassé. M2D2: A formal data model for IDS alert correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 115–137, 2002.

[20] P. G. Neumann. Computer system security evaluation. In *1978 National Computer Conference Proceedings*, pages 1087 – 1095, June 1978.

[21] P. Ning, Y. Cui, and D. S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 245–254, Washington, D.C., November 2002.

[22] P. Ning, Y. Cui, D. S. Reeves, and D. Xu. Tools and techniques for analyzing intrusion alerts. *ACM Transactions on Information and System Security*, 7(2):273–318, May 2004.

[23] Kommerling O. and Kuhn M.G. Design principles for tamper-resistant smartcard processors. In *In Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, May 1999.

[24] P.A. Porras, M.W. Fong, and A. Valdes. A mission-impact-based approach to INFOSEC alarm correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, pages 95–114, 2002.

[25] X. Qin and W. Lee. Statistical causality analysis of infosec alert data. In *Proceedings of The 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Pittsburgh, PA, September 2003.

[26] J. E. Donnelley W. L. Konigsford S. Tokubo R. P. Abbott, J. S. Chin and D. A.Webb. Security Analysis and Enhancements of Computer Operating Systems. *Institute for Computer Sciences and Technology, National Bureau of Standards*, 1976.

[27] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA conference*, 1999.

[28] S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.

[29] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of New Security Paradigms Workshop*, pages 31 – 38. ACM Press, September 2000.

[30] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 54–68, 2001.

[31] G. Vigna. A topological characterization of tcp/ip security. Technical Report TR-96.156, 1996.

[32] G. Vigna. Teaching hands-on network security: Testbeds and live exercises. *Journal of Information Warfare*, 3(2):8–25, 2003.

[33] Giocanni Vigna and Richard A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7:37–71, 1999.

[34] D. Xu and P. Ning. Alert correlation through triggering events and common resources. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, December 2004.

[35] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, February 2004.

[36] Y. Zhai, P. Ning, P. Iyer, and D.S. Reeves. Reasoning about complementary intrusion evidence. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, December 2004.

Appendix A

Correlation Graphs

A.1 Correlation results for DARPA 2000 dataset

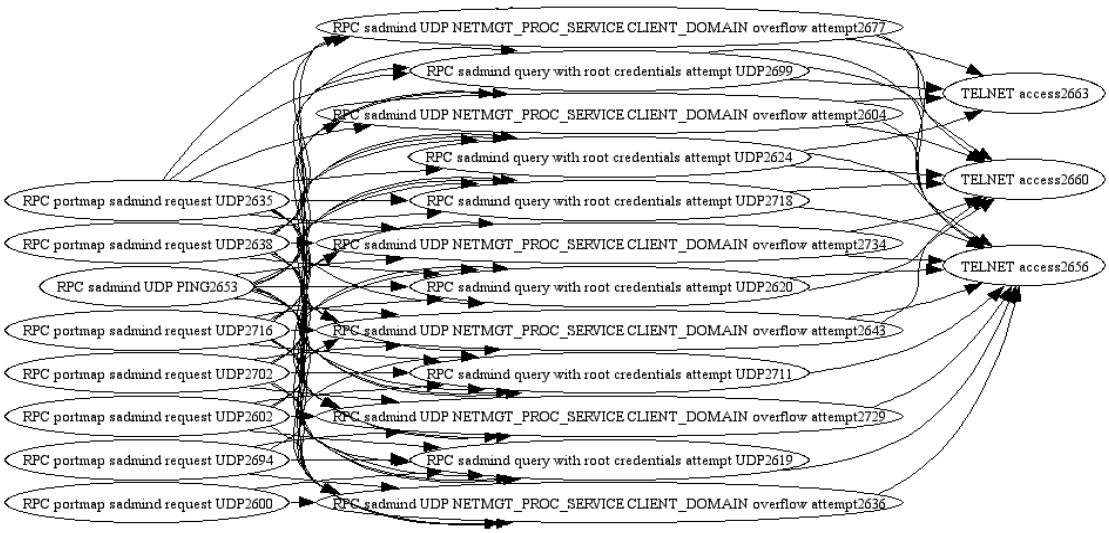
In this section we discuss the different correlation graphs that were generated in the *RealSecure* and *Snort* experiments.

For the DMZ traffic in dataset LLDOS1.0, six disjoint graphs are obtained in both *RealSecure* (figure A.3) and *Snort* (figures A.1 and A.2) experiments. The difference in the two graphs is that in the *Snort* experiment all the graphs represent three phases of the attack staring from the scanning for *sadmind* service, exploiting it and finally gaining remote access on the victim machines in the internal network. Whereas in the figure A.3, only three out of six *sadmind* attacks *Sadmind_Amslverify_Overflow* are successful as shown in figures A.3-a, A.3-b and A.3-f. There are also two false alerts in the figure A.3-a, *FTP_Syst* and *Email_Almail_Overflow* that are correlated. On the other hand, there are no false alerts in the correlation graphs of the *Snort* experiment. That makes the soundness measure for *Snort* experiment 100%. Also, as there are no missed true alerts from the correlation graph, that makes the completeness measure also 100%. In the case of *RealSecure* experiment, due to the presence of two false alerts in the correlation graph, the soundness measure is 94.74%. The completeness measure for *RealSecure* experiment is also 94.74% as no true alerts are missing.

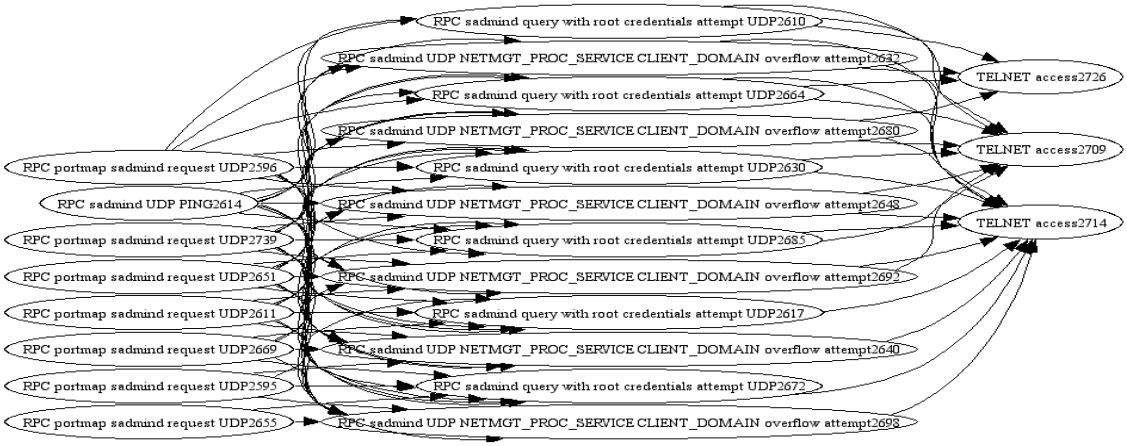
In the second scenario of the inside traffic for LLDOS2.0.2, we get a single correlation graph, shown in figure A.4 for the *Snort* experiment. This graph represents the three phases of the attack. First phase comprises of probing machines for vulnerable *sadmind* service followed by compromise of the machine running the vulnerable service. The second phase involves gaining remote access to other host through *Telnet* because *snort* IDS

does not report the alerts corresponding to the communication between the DDoS trojan. Finally, this is followed by the DDoS attack in the fourth phase of the attack. Therefore, we do not see those attacks in the correlation graph. In the *RealSecure* experiment we get two disjoint graphs, shown in figures A.5-a and A.5-b. Figure A.5-a represents the four phases of the attack starting from exploiting the vulnerable *sadmind* service through *Sadmind_Amslverify_Overflow* in the first phase. This is followed by *FTP_Put* to copy DDoS programs on the victim machine. The third phase involves the communication between the DDoS programs represented by *Mstream_Zombie*. Finally, this is followed by the DDoS attack *Stream_DoS* in the fourth phase of the attack. This graph also contains a false alert *Email_Almail_Overflow* that reduces the soundness measure. Figure A.5-b shows two normal activities *Email_Ehlo* and *Email_Turn* that are correlated. This correlation graph misses certain variations of *Telnet* that are true alerts which reduces the completeness measure to 66.7%. However, the soundness measure is still as high as 92.3% because there is only one false alert. For the *Snort* experiment both the completeness and soundness measures are 100% as there are no missing true alerts and no false alerts present in the correlation graphs.

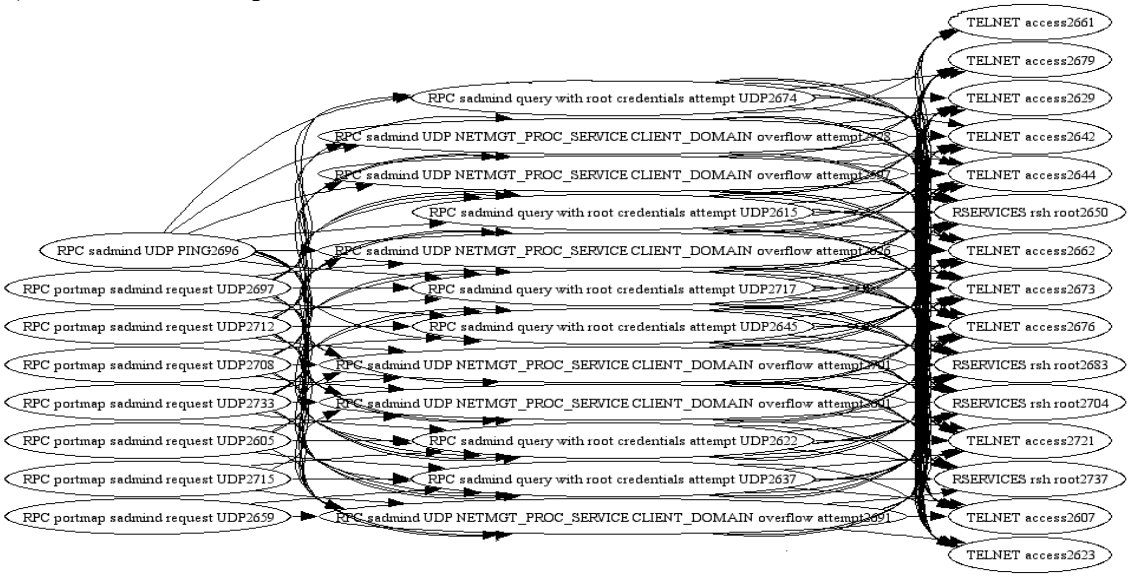
The DMZ traffic in dataset LLDOS2.0.2 resulted in a single correlation graph, shown in figureA.6 for *Snort* experiment. The graph again denotes the three phases that are probing for *sadmind* , exploiting *sadmind* and finally the *Telnet* access. There were no missing true alerts or any false alerts in the correlation graph making the completeness and soundness measures to be 100%. For the *RealSecure* experiment we get two graphs, shown in figure A.7-a and A.7-b. Here, the graph in figure A.7-a represents that the attacker exploits the *sadmind* service on the victim machine and then launches a FTP attack to install some program on the victim, the DDoS program. Here, there are no false alerts in the correlation graph therefore the soundness measure is 100%. However, there are three variations of the *Telnet* that are missing from the correlation graph which reduces the completeness measure to 62.5%.



d) Correlation Graph for IP Address = 172.016.114.020

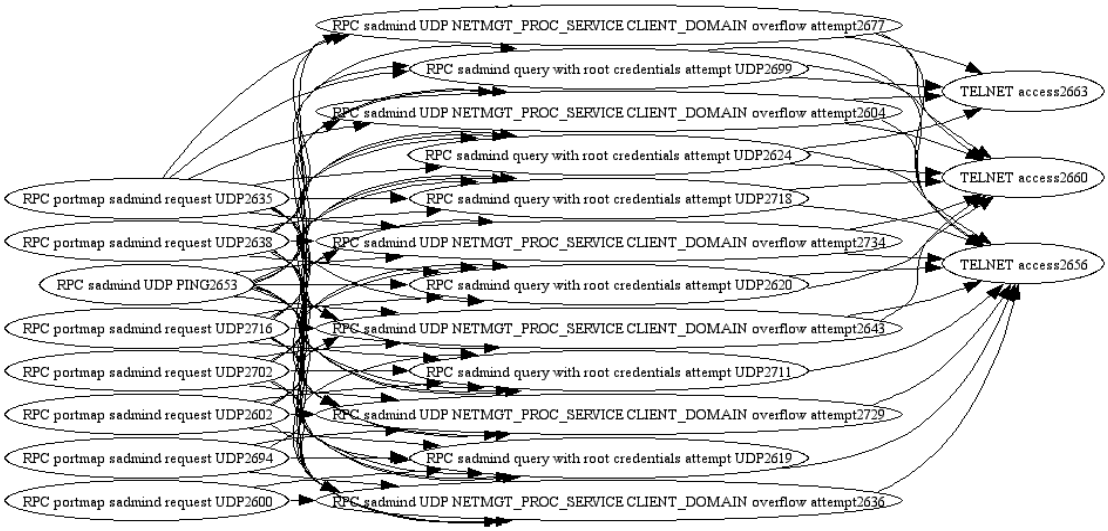


e) Correlation Graph for IP Address = 172.016.114.030

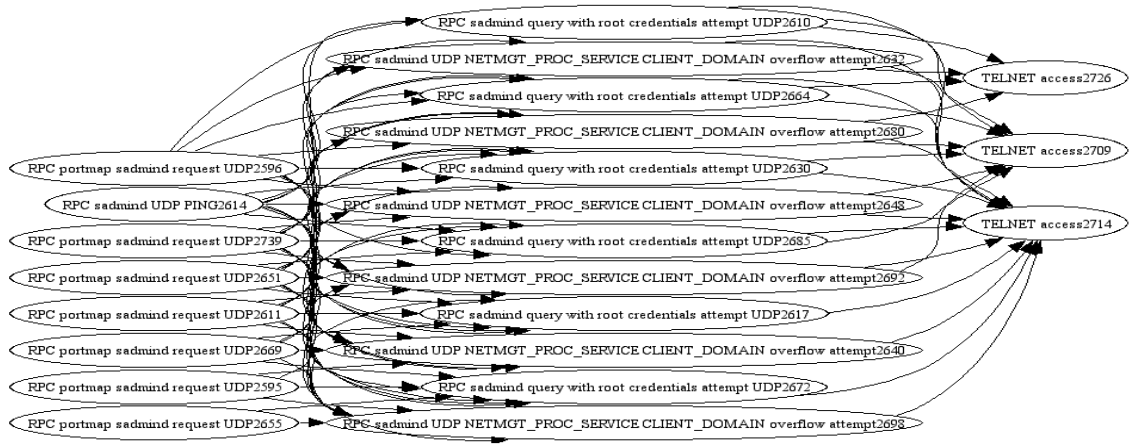


f) Correlation Graph for IP Address = 172.016.115.020

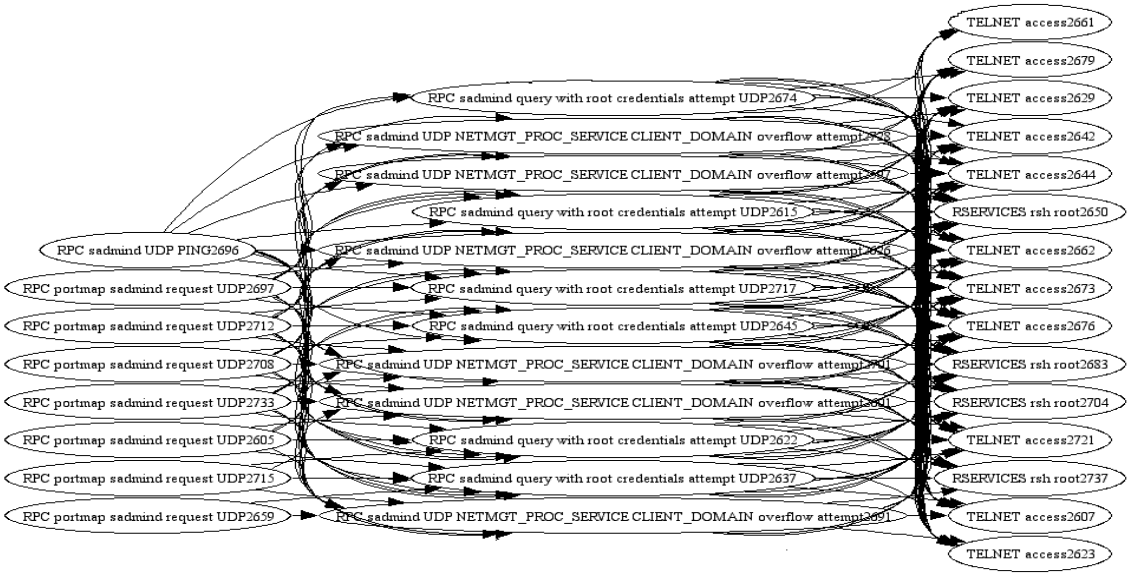
Figure A.1: First three Snort correlation graphs for DARPA 2000 DMZ traffic in LLDOS1.0



d) Correlation Graph for IP Address = 172.016.114.020

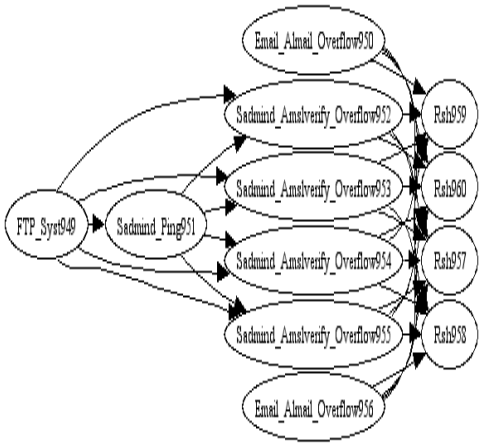


e) Correlation Graph for IP Address = 172.016.114.030

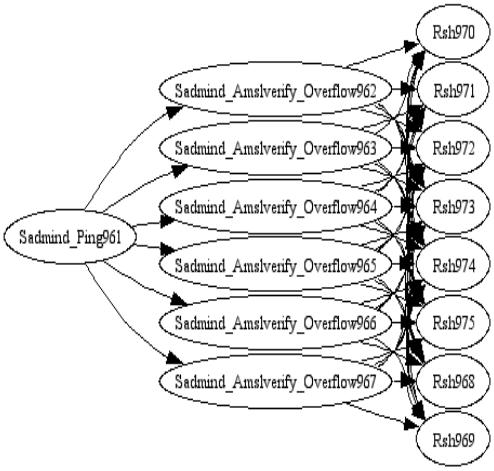


f) Correlation Graph for IP Address = 172.016.115.020

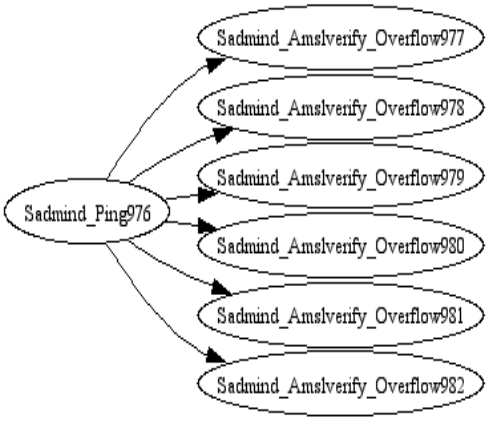
Figure A.2: Remaining three Snort correlation graphs for DARPA 2000 DMZ traffic in LLDOS1.0



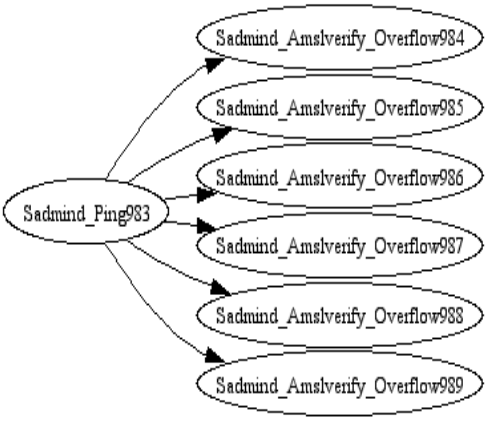
a) Destination IP = 172.016.112.050



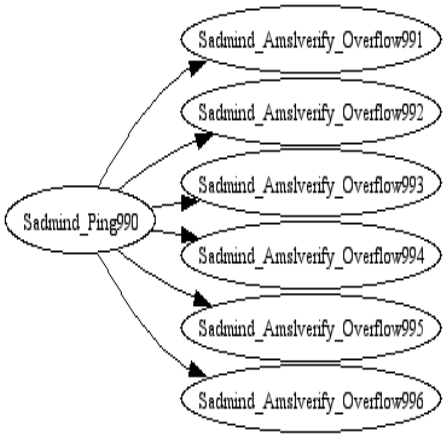
b) Destination IP = 172.016.115.020



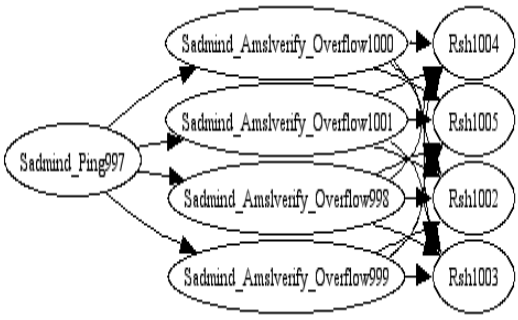
c) Destination IP = 172.016.114.010



d) Destination IP = 172.016.114.020



e) Destination IP = 172.016.114.030



f) Destination IP = 172.016.112.010

Figure A.3: RealSecure correlation graphs for DARPA 2000 DMZ traffic in LLDOS1.0

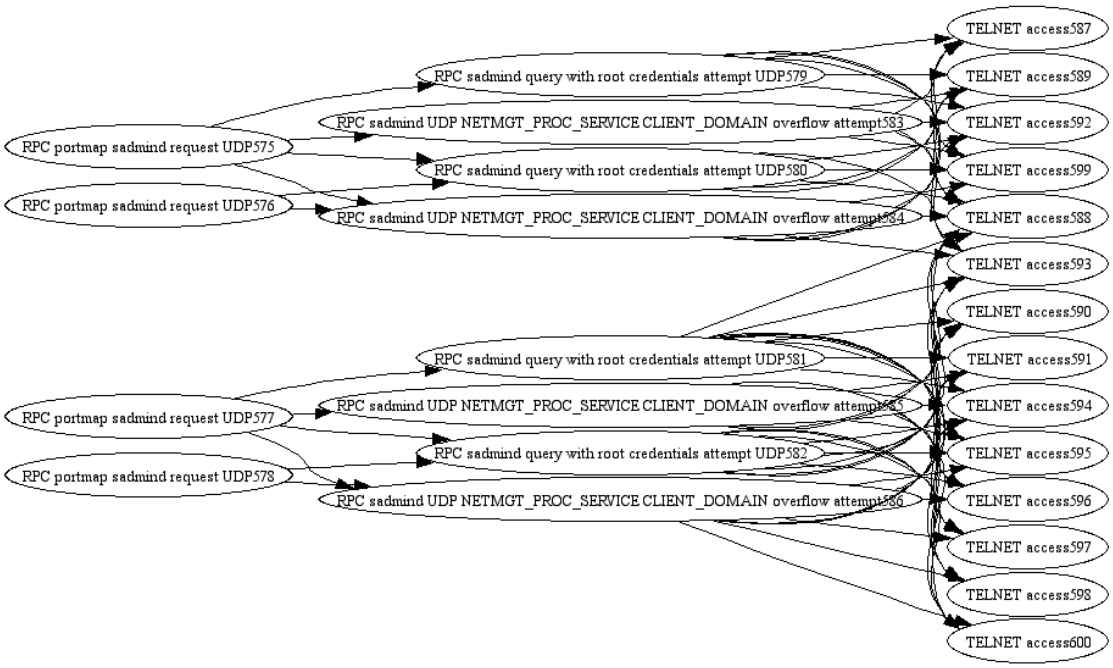
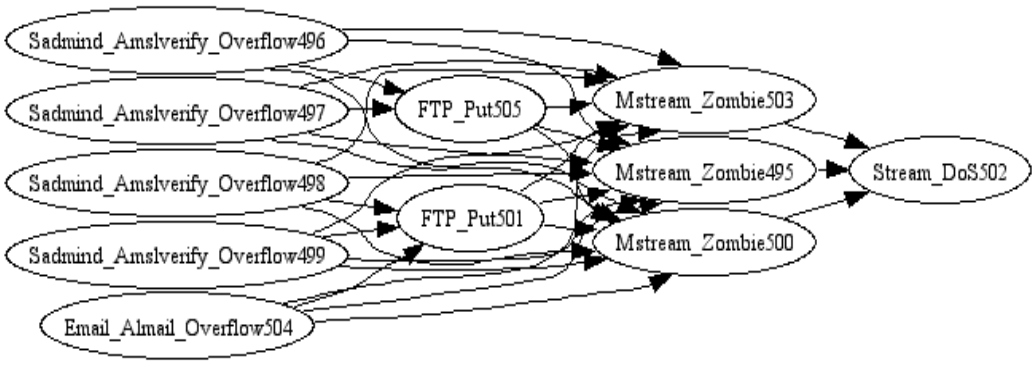
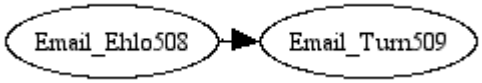


Figure A.4: Snort correlation graphs for DARPA 2000 Inside traffic in LLDOS2.0.2



a) Correlation graph for the attacks



b) Graph for related activities.

Figure A.5: RealSecure correlation graphs for DARPA 2000 Inside traffic in LLDOS2.0.2

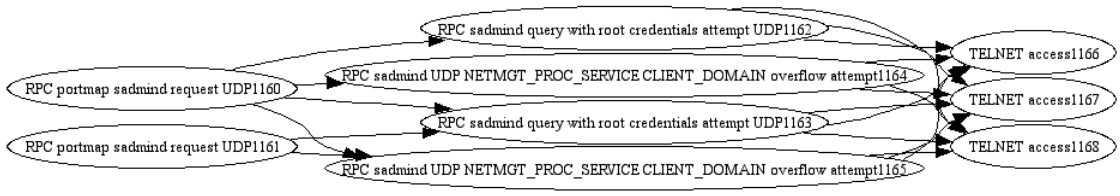
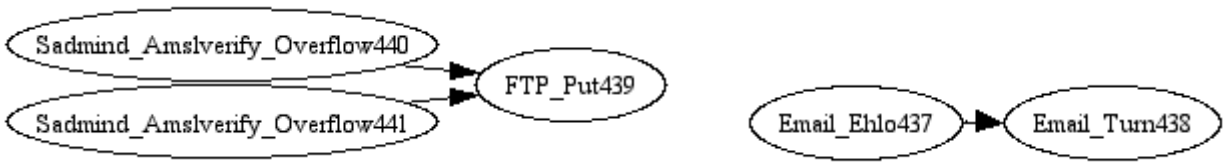


Figure A.6: Snort correlation graphs for DARPA 2000 DMZ traffic in LLDOS2.0.2



a) Correlation graph for the attacks b) Graph for related activities

Figure A.7: RealSecure correlation graphs for DARPA 2000 DMZ traffic in LLDOS2.0.2

Appendix B

Knowledge Base Used in the Experiments

B.1 Knowledge Base Used for Treasure Hunt dataset

```
<?xml version="1.0" encoding="UTF-8"?>
<KnowledgeBase xmlns="ID@NCSU"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="ID@NCSU
   newKnowledgeBase.xsd">


<!-- ================= Predicates ============== -->
<Predicates>
<Predicate Name="Access">
  <Arg id="1" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ReadFileObject">
  <Arg id="2" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ApacheWeb">
  <Arg id="3" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="4" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="ScalperWorm">
  <Arg id="5" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="HTTP">
  <Arg id="7" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="88" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="FixedService">
```

```
  <Arg id="8" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="9" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="LiveHost">
  <Arg id="10" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="MYSQL">
    <Arg id="11" Pos="1" Attr="varchar(15)"></Arg>
    <Arg id="12" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="RootAcess">
  <Arg id="13" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSFreeBSD">
  <Arg id="14" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSWindows">
  <Arg id="15" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="MountdService">
  <Arg id="16" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="PortMapper">
  <Arg id="18" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="19" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="OSUnix">
  <Arg id="20" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="ypservService">
  <Arg id="22" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="LocalAccess">
  <Arg id="23" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSSolaris">
  <Arg id="24" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="SocksProxy">
  <Arg id="25" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="26" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="SNMP">
  <Arg id="27" Pos="1" Attr="varchar(15)"></Arg>
```

```
  <Arg id="28" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="DoSSNMP">
  <Arg id="29" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="30" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="Finger">
  <Arg id="31" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="32" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="ReadFileSystem">
  <Arg id="33" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="NauticaMarlinBridge">
  <Arg id="34" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="OSUNIX">
  <Arg id="35" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="DoSNauticaMarlinBridge">
  <Arg id="36" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="RPCService">
  <Arg id="37" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="Read/etc/passwdFile">
  <Arg id="38" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="Telnet">
  <Arg id="39" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="40" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="nfsdService">
  <Arg id="41" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="FileSystem">
  <Arg id="42" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="MstreamClient">
  <Arg id="43" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="MstreamHandler">
  <Arg id="44" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
```

```
<Predicate Name="MstreamZombie">
</Predicate>
<Predicate Name="MstreamHandler">
  <Arg id="45" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="QVTFTP">
  <Arg id="46" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="47" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="Rsh">
  <Arg id="48" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="49" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="Rexec">
  <Arg id="50" Pos="1" Attr="varchar(15)"></Arg>
  <Arg id="51" Pos="2" Attr="int"></Arg>
</Predicate>
<Predicate Name="CrashOS">
  <Arg id="52" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
<Predicate Name="yppasswdService">
  <Arg id="53" Pos="1" Attr="varchar(15)"></Arg>
</Predicate>
</Predicates>

<!-- ================= Implications ================== -->
<Implications>
<Implication>
  <ImplyingName>RootAccess</ImplyingName>
  <ImpliedName>Access</ImpliedName>
  <ArgMap>
    <ImplyingArg id="13"></ImplyingArg>
    <ImpliedArg id="1"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>ApacheWeb</ImplyingName>
  <ImpliedName>FixedService</ImpliedName>
  <ArgMap>
    <ImplyingArg id="3"></ImplyingArg>
    <ImpliedArg id="8"></ImpliedArg>
    <ImplyingArg id="4"></ImplyingArg>
    <ImpliedArg id="9"></ImpliedArg>
  </ArgMap>
```

```
</Implication>
<Implication>
  <ImplyingName>HTTP</ImplyingName>
  <ImpliedName>FixedService</ImpliedName>
  <ArgMap>
    <ImplyingArg id="7"></ImplyingArg>
    <ImpliedArg id="8"></ImpliedArg>
    <ImplyingArg id="88"></ImplyingArg>
    <ImpliedArg id="9"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>MYSQL</ImplyingName>
  <ImpliedName>FixedService</ImpliedName>
  <ArgMap>
    <ImplyingArg id="11"></ImplyingArg>
    <ImpliedArg id="8"></ImpliedArg>
    <ImplyingArg id="12"></ImplyingArg>
    <ImpliedArg id="9"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>OSWindows</ImplyingName>
  <ImpliedName>OS</ImpliedName>
  <ArgMap>
    <ImplyingArg id="15"></ImplyingArg>
    <ImpliedArg id="41"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>OSSolaris</ImplyingName>
  <ImpliedName>OS</ImpliedName>
  <ArgMap>
    <ImplyingArg id="24"></ImplyingArg>
    <ImpliedArg id="41"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>OSUnix</ImplyingName>
  <ImpliedName>OS</ImpliedName>
  <ArgMap>
    <ImplyingArg id="20"></ImplyingArg>
    <ImpliedArg id="41"></ImpliedArg>
  </ArgMap>
```

```
</Implication>
<Implication>
  <ImplyingName>OSFreeBSD</ImplyingName>
  <ImpliedName>OS</ImpliedName>
  <ArgMap>
    <ImplyingArg id="14"></ImplyingArg>
    <ImpliedArg id="41"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>ReadFileSystem</ImplyingName>
  <ImpliedName>FileSystem</ImpliedName>
  <ArgMap>
    <ImplyingArg id="33"></ImplyingArg>
    <ImpliedArg id="42"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication>
  <ImplyingName>LocalAccess</ImplyingName>
  <ImpliedName>Access</ImpliedName>
  <ArgMap>
    <ImplyingArg id="23"></ImplyingArg>
    <ImpliedArg id="1"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>OS</ImplyingName>
  <ImpliedName>OSSolaris</ImpliedName>
  <ArgMap>
    <ImplyingArg id="41"></ImplyingArg>
    <ImpliedArg id="24"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>OS</ImplyingName>
  <ImpliedName>OSWindows</ImpliedName>
  <ArgMap>
    <ImplyingArg id="41"></ImplyingArg>
    <ImpliedArg id="15"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>OS</ImplyingName>
  <ImpliedName>OSUNIX</ImpliedName>
```

```
  <ArgMap>
    <ImplyingArg id="41"></ImplyingArg>
    <ImpliedArg id="20"></ImpliedArg>
  </ArgMap>
</Implication>
<Implication Phantom="Yes">
  <ImplyingName>OS</ImplyingName>
  <ImpliedName>OSFreeBSD</ImpliedName>
  <ArgMap>
    <ImplyingArg id="14"></ImplyingArg>
    <ImpliedArg id="20"></ImpliedArg>
  </ArgMap>
</Implication>
</Implications>


<!-- ================ Hyper Alert Types ================= -->
<HyperAlertTypes>

<HyperAlertType Name="RPC portmap ypserv request UDP">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="PortMapper">
      <Arg id="18" ArgName="DestIPAddress"></Arg>
      <Arg id="19" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="OSUNIX">
      <Arg id="20" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ypservService">
      <Arg id="22" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

 <HyperAlertType Name="MYSQL show databases attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
```

```
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="MYSQL">
      <Arg id="11" ArgName="DestIPAddress"></Arg>
      <Arg id="12" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadFileObject">
      <Arg id="2" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="ICMP PING NMAP">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Protocol ProtocolName="ICMP"></Protocol>
  <Prerequisite>
  </Prerequisite>
  <Consequence>
  <Predicate Name="LiveHost">
     <Arg id="10" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SCAN nmap XMAS">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
      <Predicate Name="LiveHost">
         <Arg id="10" ArgName="DestIPAddress"></Arg>
        </Predicate>
    </Prerequisite>
  <Consequence>
    <Predicate Name="FixedService">
          <Arg id="8" ArgName="DestIPAddress"></Arg>
          <Arg id="9" ArgName="DestPort"></Arg>
```

```
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SCAN Squid Proxy attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
      <Arg id="10" ArgName="DestIPAddress"></Arg>
    </Predicate>
    </Prerequisite>
  <Consequence>
    <Predicate Name="FixedService">
          <Arg id="8" ArgName="DestIPAddress"></Arg>
          <Arg id="9" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="TELNET access">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="Telnet"></Protocol>
  <Prerequisite>
    <Predicate Name="Telnet">
      <Arg id="39" ArgName="SrcIPAddress"></Arg>
      <Arg id="40" ArgName="SrcPort"></Arg>
    </Predicate>
    <Predicate Name="Access">
      <Arg id="1" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="Access">
      <Arg id="1" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>
```

```
<HyperAlertType Name="RSERVICES rexec password overflow attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="Rsh">
      <Arg id="48" ArgName="DestIPAddress"></Arg>
      <Arg id="49" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="CrashOS">
       <Arg id="52" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RSERVICES rexec username overflow attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="Rexec">
      <Arg id="50" ArgName="DestIPAddress"></Arg>
      <Arg id="51" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="RootAccess">
          <Arg id="13" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC portmap listing TCP 111">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
```

```
  <Protocol ProtocolName="TCP"></Protocol>
  <Prerequisite>
    <Predicate Name="Portmapper">
      <Arg id="18" ArgName="DestIPAddress"></Arg>
      <Arg id="19" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="RPCService">
      <Arg id="37" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC portmap mountd request UDP">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="Portmapper">
      <Arg id="18" ArgName="DestIPAddress"></Arg>
      <Arg id="19" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="MountdService">
     <Arg id="16" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC mountd TCP export request">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="TCP"></Protocol>
  <Prerequisite>
    <Predicate Name="MountdService">
      <Arg id="16" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
```

```
  <Consequence>
    <Predicate Name="ReadFileSystem">
      <Arg id="33" Pos="1" Attr="varchar(15)"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SNMP trap udp">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="SNMP"></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
      <Arg id="10" ArgName="DestIPAddress"></Arg>
        </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SNMP">
          <Arg id="27" ArgName="DestIPAddress"></Arg>
          <Arg id="28" ArgName="DestPort"></Arg>
        </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name=" RPC portmap NFS request UDP">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="PortMapper">
        <Arg id="18" ArgName="DestIPAddress"></Arg>
        <Arg id="19" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="OSUnix">
      <Arg id="20" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="nfsdService">
      <Arg id="41" ArgName="DestIPAddress"></Arg>
```

```
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC mountd TCP mount request">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="TCP"></Protocol>
  <Prerequisite>
    <Predicate Name="FileSystem">
          <Arg id="42" Pos="1" Attr="varchar(15)"></Arg>
    </Predicate>
    <Predicate Name="MountdService">
       <Arg id="16" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadFileSystem">
      <Arg id="33" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SNMP request udp">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="SNMP"></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
          <Arg id="10" ArgName="DestIPAddress"></Arg>
       </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SNMP">
      <Arg id="27" ArgName="DestIPAddress"></Arg>
      <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>
```

```
<HyperAlertType Name="RPC mountd UDP mount request">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
  <Predicate Name="FileSystem">
        <Arg id="42" ArgName="DestIPAddress"></Arg>
    </Predicate>
  <Predicate Name="MountdService">
    <Arg id="16" ArgName="DestIPAddress"></Arg>
  </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadFileSystem">
      <Arg id="33" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC mountd UDP unmount request">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="MountdService">
       <Arg id="20" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SNMP AgentX/tcp request">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="SNMP"></Protocol>
  <Prerequisite>
    <Predicate Name="SNMP">
```

```
      <Arg id="27" ArgName="DestIPAddress"></Arg>
      <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="RootAccess">
      <Arg id="13" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="DoSSNMP">
      <Arg id="29" ArgName="DestIPAddress"></Arg>
      <Arg id="30" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SNMP trap tcp">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="SNMP"></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
      <Arg id="10" ArgName="DestIPAddress"></Arg>
        </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SNMP">
      <Arg id="27" ArgName="DestIPAddress"></Arg>
      <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SNMP request tcp">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="SNMP"></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
          <Arg id="10" ArgName="DestIPAddress"></Arg>
        </Predicate>
```

```
  </Prerequisite>
  <Consequence>
    <Predicate Name="SNMP">
      <Arg id="27" ArgName="DestIPAddress"></Arg>
      <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name=" SCAN Proxy Port 8080 attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="LiveHost">
       <Arg id="10" ArgName="DestIPAddress"></Arg>
        </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="SNMP">
          <Arg id="27" ArgName="DestIPAddress"></Arg>
          <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="SCAN SOCKS Proxy attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="SocksProxy">
      <Arg id="25" ArgName="SrcIPAddress"></Arg>
      <Arg id="26" ArgName="SrcPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="LiveHost">
      <Arg id="10" ArgName="SrcIPAddress"></Arg>
    </Predicate>
```

```
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="ATTACK-RESPONSES 403 Forbidden">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="HTTP"></Protocol>
  <Prerequisite>
    <Predicate Name="HTTP">
      <Arg id="7" ArgName="SrcIPAddress"></Arg>
      <Arg id="88" ArgName="SrcPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="Access">
      <Arg id="1" ArgName="SrcIPAddress"></Arg>
    </Predicate>
    <Predicate Name="ReadFileObject">
      <Arg id="2" ArgName="SrcIPAddress"></Arg>
    </Predicate>
  </Consequence></HyperAlertType>

<HyperAlertType Name="DOS Bay/Nortel Nautica Marlin">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="SNMP">
      <Arg id="27" ArgName="DestIPAddress"></Arg>
      <Arg id="28" ArgName="DestPort"></Arg>
    </Predicate>
    <Predicate Name="NauticaMarlinBridge">
      <Arg id="34" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="DoSNauticaMarlinBridge">
       <Arg id="36" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
```

```
</HyperAlertType>

<HyperAlertType Name="DDOS mstream client to handler">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="TCP"></Protocol>
  <Prerequisite>
    <Predicate Name="Access">
      <Arg id="1" ArgName="DestIPAddress"></Arg>
    </Predicate>
    <Predicate Name="Access">
      <Arg id="1" ArgName="SrcIPAddress"></Arg>
    </Predicate>
    <Predicate Name="MstreamClient">
      <Arg id="43" ArgName="SrcIPAddress"></Arg>
    </Predicate>
    <Predicate Name="MstreamHandler">
      <Arg id="44" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="MstreamZombie">
    </Predicate>
  </Consequence>
</HyperAlertType>


<HyperAlertType Name="FINGER root query">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName=""></Protocol>
  <Prerequisite>
    <Predicate Name="Finger">
      <Arg id="31" ArgName="DestIPAddress"></Arg>
      <Arg id="32" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadFileObject">
      <Arg id="2" ArgName="DestIPAddress"></Arg>
```

```
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name="RPC portmap yppasswd request UDP">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="UDP"></Protocol>
  <Prerequisite>
    <Predicate Name="OSUNIX">
      <Arg id="35" ArgName="DestIPAddress"></Arg>
    </Predicate>
  <Predicate Name="PortMapper">
        <Arg id="18" ArgName="DestIPAddress"></Arg>
        <Arg id="19" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="yppasswdService">
      <Arg id="53" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>

<HyperAlertType Name=" FTP LIST directory traversal attempt">
  <Fact FactName="SrcIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="SrcPort" FactType="int"></Fact>
  <Fact FactName="DestIPAddress" FactType="varchar(15)"></Fact>
  <Fact FactName="DestPort" FactType="int"></Fact>
  <Protocol ProtocolName="FTP"></Protocol>
  <Prerequisite>
    <Predicate Name="QVTFTP">
      <Arg id="46" ArgName="DestIPAddress"></Arg>
      <Arg id="47" ArgName="DestPort"></Arg>
    </Predicate>
  </Prerequisite>
  <Consequence>
    <Predicate Name="ReadFileObject">
      <Arg id="2" ArgName="DestIPAddress"></Arg>
    </Predicate>
  </Consequence>
</HyperAlertType>
```

```
</HyperAlertTypes>

</KnowledgeBase>
```