NetSPA: A Network Security Planning Architecture

by

Michael Lyle Artz

S.B., Computer Science and Engineering
Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002
[June 2002]

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Richard P. Lippmann
Senior Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

NetSPA: A Network Security Planning Architecture

by

Michael Lyle Artz

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Attack scenario graphs provide a concise way of displaying all possible sequences of attacks a malicious user can execute to obtain a desired goal, such as remotely achieving root undetected on a critical host machine. NETSPA, the Network Security Planning Architecture, is a C++ system that quickly generates worst-case attack graphs using a forward-chaining depth-first search of the possible attack space using actions modeled with REM, a simple attack description language. NETSPA accepts network configuration information from a database that includes host and network software types and versions, intrusion detection system placement and types, network connectivity, and firewall rulesets. It is controlled by command line inputs that determine a critical goal state, trust relationships between hosts, and maximum recursive depth. NETSPA was shown to efficiently provide easily understood attack graphs that revealed non-obvious security problems against a realistic sample network of 17 representative hosts using 23 REM defined actions. The largest useful graph was generated within 1.5 minutes of execution. NETSPAexecutes faster and handles larger networks than any existing graph generation system. This allows NETSPA to be practically used in combination with other security components to develop and analyze secure networks.

Thesis Supervisor: Richard P. Lippmann
Title: Senior Staff, MIT Lincoln Laboratory

Acknowledgments

The creation and development of this thesis would not have been possible without the support and motivation of Rich Lippmann, my advisor. It was he who originally presented the idea of an attack graph generation system and he who saw the project right up to the final moments, providing valuable comments and insights along the way. For this and more, I am supremely grateful.

In addition, many people contributed, in one way or another, to this project. I must thank all of my friends for both encouraging me to work and encouraging me to take a break. Both were necessary, and they seemed to know when it was time for each. Jon Salz, especially, provided me with constant knowlegeable advice. Thanks Jon.

Finally, I would like to thank my family, without whose love and support I would not be here today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security is a main concern of most network administrators. The frequency with which new software vulnerabilities are found and subsequently fixed has been increasing steadily over the past several years, with current incident rates as high as almost three per day [4]. This ever increasing number of vulnerabilities and software patches, coupled with widespread access to exploit information and public networks, provides a challenging arena for security administrators. They must combat threats from malevolent attackers, curious crackers, and "script kiddies," novice computer users using prepackaged exploit software. Software vulnerability consequences range from the inadvertent disclosure of relatively unimportant information to fully privileged remote access of critical systems.

A network security administrator must balance possible network compromises against the necessity and convenience afforded by allowing communication between a world-accessible network and a private internal network. For example, to allow users to effectively work on internal projects outside the office, it is necessary to allow them some sort of remote access, most likely in the form of a virtual private network or remote login. This opening to the internal network, however, is one more possible point of entry for an attacker. In addition, this sort of remote authentication also contains the possibility that the attacker will "sniff" the transmission and obtain necessary login information, such as user name and password. While working at home is merely a convenience, many organizations require Internet access to perform necessary, business-related activities. A company that sells products via the World Wide Web must allow outside users access to a database of customer information for authentication and personalization. This access provides additional attack paths for a

skilled intruder.

A plethora of both hardware and software has been developed to combat network attacks at many different levels. Vulnerability scanners attempt to inform an administrator about single-point software vulnerabilities once a system is installed and operational. Intrusion Detection Systems (IDSs) monitor current system and network behavior and raise an alert when some network packet or host behavior violates certain conditions. Firewalls, both in hardware and software, enforce a given security policy by blocking certain types of network access. All of these methods, however, require a physical network to function, and cannot take advantage of a simulated network, nor aid an administrator in planning a secure network.

Attacks sometimes only exploit a single vulnerability. Worms, for example, automated, self-propagating attack software, often search for only a single vulnerability in a host and exploit it if found. Novice computer attackers tend to act in much the same fashion, only exploiting one or few vulnerabilities in order to gain some sort of access into a network. Proficient network attackers, however, often have a goal in mind when attacking a network, and craft their exploits to reach the goal in the easiest way. In addition, expert attackers tend to go through many actions to both limit visibility to intrusion detection systems and quickly exploit targets. Finally, a network attacker often leaves behind malicious software to provide a permanent "back door" into the network. This multi-component attack can be considered a path on a graph of all of the possible attack routes through a network, otherwise known as an attack graph [20, 29].

This thesis describes the Network Security Planning Architecture (NETSPA), a system that provides several necessary augmentations to the existing body of security software. Most importantly, NETSPA generates attack graphs from a network topology, graphs of all of the possible routes an attacker can take to exploit a user-defined network. These graphs and their associated statistics, such as number of hosts compromised and attacker privilege levels, allow a network administrator to determine likely intrusion paths and extrapolate this data to determine the current and future security of the network given past software vulnerability frequencies. As the attack graphs are displayed in near real-time, an administrator can change the network topology slightly, re-compute the graphs for the new topology, and compare the graphs produced from different configurations. This allows an administrator to weigh network security against other factors, such as hardware costs and

ease of maintenance. Finally, NETSPA imports information from several existing security and network planning tools. Existing network configuration information can be obtained through the use of tools such as nmap [11], Nessus [8], and NetViz [22]. Online databases such as ICAT [4] and the Nessus vulnerability plug-ins provide valuable information about attack requirements and effects.

1.1 Outline

The following chapter describes some of the background behind network security issues, including basic types of computer attacks, attack graph information, and existing security software and information resources. Chapter 3 provides an overview of current attack graph generation mechanisms and related attack modeling languages. Chapter 4 discusses the design and Chapter 5 the implementation of NETSPA. Chapter 6 then shows the operation of NETSPA against two different network configurations, one very simple and one much more realistic, with associated security questions and scenarios. Chapter 7 details some additional enhancements that will make NETSPA a more viable security solution, and, finally, Chapter 8 concludes with a few closing remarks on network security and NETSPA's role within that realm.

Chapter 2

Background

This chapter describes the background, terminology, and tools necessary to understand the basic issues involved when combating computer and network intrusions. Section 2.4 explains attack graphs and their importance to network administrators in creating secure networks, and Section 2.5 enumerates important types of existing network security software and information systems, several of which are used to import information into NETSPA.

2.1 Introduction to Vulnerabilities and Exploits

Security administrators are primarily concerned with detecting and fixing software vulnerabilities before they are exploited by outside attackers. Vulnerabilities are weaknesses in software packages and systems that allow the software to operate outside of designed boundaries. Exploits then utilize vulnerabilities to gain otherwise inaccessible information and privilege. Vulnerabilities, and to a lesser extent, exploits, are well documented in online security databases such as the ICAT Meta-base [4] and the Common Vulnerabilities and Exposures [2] list. In addition, these databases have been used to help characterize the frequency at which software vulnerabilities are found and subsequently fixed [16].

Software vulnerabilities generally arise from basic system complexity and bad programming practices, as is the case with most buffer overflows. Other vulnerabilities, however, are inherent in the software design or arise from mis-configuration. For example, FTP transmits login and authentication information "in the clear" via an unprotected channel. ICAT provides several distinctions between vulnerabilities, such as the necessary position of an attacker relative to the target host and the consequences of an exploited vulnerability.

Vulnerabilities can exist in software long before they are found by other users. In [16], the authors concluded, using open information such as that found in ICAT, that the rate of new vulnerabilities found in a high profile software packages, such as a web or DNS server, is between 0 and 7 per year. While this may seem almost inconsequential, when an entire organizational network and all the hundreds of pieces of associated software is taken into account, new vulnerabilities should be expected on a weekly basis.

Exploits take advantage of software vulnerabilities to gain information and privilege. Exploitation of the FTP service could involve listening, or "sniffing," the network to obtain packets destined for another machine. The login information can then be read straight from these plain-text packets, granting an attacker the ability to log into the FTP server and masquerade as a legitimate user. Another exploit of the same FTP server could involve sending specially crafted input that causes the server to execute arbitrary commands at the privilege level of the FTP server.

Software exploits are characterized based on the level of privilege provided and the necessary location of the attacker relative to the target host. Local exploits are those that can only be accomplished when on the same machine as the vulnerable software, whether by being physically at the terminal or logged in remotely. In contrast, remote exploits allow an attack to be executed from a remote location, only necessitating a network connection between the attacker and the vulnerable software's port. Successful exploits generally provide the attacker with privilege equal to that of the running program.

The most common and well studied vulnerabilities and associated exploits arise when a programmer accepts input from a user and copies it into a buffer not big enough for the input. In general, correct practice in such a case is to truncate the input to a size that fits within the buffer. In the past, many programmers simply made the buffer large enough to fit everything that they thought was reasonable and blindly copied the input into it. Clever security analysts, however, found out that it was possible to give the incorrectly written software a piece of data larger than the buffer size, thus overflowing it, and writing to unauthorized places in memory. With a little knowledge of the computing architecture and stack, it is then possible to hijack the program to execute arbitrary code with privileges equal to that of the running program [15]. This is known as a buffer overflow. One reason that buffer overflows are so well known is that they are relatively common and easy to exploit. If the vulnerability is evident in a network program, it often can lead to remote

compromises.

Another instance of incorrect input validation is evident in format string errors. Exploiting one of these vulnerabilities has the same effect as exploiting a buffer overflow, allowing the attacker to run code with the privileges of the program. The format string exploit is enabled, however, due to the incorrect handling of format strings, such as those passed to C functions `printf` and `sprintf`. Specially crafted format strings allow a user to write to arbitrary points in the program's memory. These sorts of vulnerabilities are less common, however they do occur and can be exploited [32].

Logic errors comprise a nefarious class of vulnerabilities that can't quite be pigeonholed. Some are simple mistakes in coding, such as not handling a boundary case or not ending a loop at the right time. Others can be complex, as in not providing a mechanism to handle a special input case. Logic errors tend to be overlooked because the program works correctly under normal conditions and only resorts to incorrect behavior with special case inputs. While buffer overflows and format string exploits generally give the attacker privilege on the machine running the vulnerable software, logic errors behave in numerous ways when exploited. An exploited logic error might release information from the computer, change a local file, or simply crash the software. For example, a naive web server that does not correctly parse the character ';' could allow an attacker to remotely download any file inside of the web server's configuration directory. Another prime example of logic errors is found in race conditions. Multiple asynchronous threads of computation sharing data provide a haven for slight mistakes that might show up if a certain pattern of behavior or certain timing is enacted.

Several other types of computer attacks exist that don't categorize well to specific vulnerabilities. Some types of denial-of-service (DOS) attacks attempt to send requests to a piece of software faster than the system can handle them. This flooding can prevent the attacked software from responding to valid queries coming from other agents, thus rendering it useless. While a DOS attack does not gain an attacker any sort of direct access into the network, it is extremely easy to create, difficult to protect against, and can cause businesses to lose money due to the server down-time.

Possibly the most naive attack available, a brute force attack, attempts to enumerate through all of the possible combinations of input to try and find something that "works." The most common example of such an attack is in password guessing. An attacker can

simply enumerate all of the possible combinations of letters and numbers until the right one is found. While this could take an incredibly long amount of time, brute force attacks can be sped up with any sort of prior knowledge. In the password guessing example, the attacker might first try all of the letters in the person's name, as well as numbers in his birthday and social security number, checking for weak passwords. Similar brute force attacks can be made against cryptographic keys and other authentication information.

Finally, worms comprise another class of computer attacks. Worms are malicious programs that automatically exploit known system vulnerabilities and then use compromised hosts as launching stations for additional automated attacks. Examples of such attacks are the Code Red [5] and Nimbda [35] worms, which propagated by exploiting Microsoft's Internet Information Service (IIS).

2.2 Protecting Against Network Attacks

Numerous ways exist to protect against computer attacks. The most obvious solution is to simply fix the problem in the software and eliminate the vulnerability. While this is also the best solution, several hurdles exist in changing vulnerable software. First of all, attempting to understand and correct all of the hundreds of programs that run on all of the computers in a network is often infeasible. Also, not all software packages are distributed with source code, so it is likely that only the software vendor has the required access to change the behavior of the software. Even if the software vendors fix all of the problems within their respective software, there is still a window of time (sometimes large, depending on the vendor) where the unfixed, vulnerable software is being used and the exploit is known [16]. Once an update is distributed, it is then necessary for the network administrators to find all of the computers that are running the previously vulnerable software and apply the update.

While fixing vulnerabilities in software is the "correct" way to protect against network attacks, the feasibility of using this as the only form of network protection is highly questionable. Other solutions have been developed to protect against software exploits in the face of numerous software vulnerabilities, even those that are still unknown. One trivial and effective mechanism is to simply deny access to the software from unknown sources. This can be accomplished by totally disconnecting the internal network from any outside

networks. As mentioned previously, this solution is inconvenient and often impossible for most organizations to accomplish. Firewalls and their derivatives provide a more common way of denying access to certain software. Packet-based firewalls block network traffic based on criteria about network packets, including origination and destination IP addresses, port numbers, and, in the case of more complicated firewalls, protocol type and packet content. Firewall rules can be set up to block access to software that is running on a certain host-port combination, effectively blocking access to that piece of software across the firewall. While these rules are quite effective, correctly setting up and managing a firewall is a difficult task, especially for large internal networks and complicated security policies.

2.3 Anatomy of a Network Attack

A network attacker executes a series of steps to reach a desired goal. The order and duration of these steps is dependent on several factors including attacker skill level, type of vulnerability to be exploited, amount of prior information, and starting location of the attacker. The attacker's steps range from finding out about the network via port scans, running exploits against the network, cleaning attack evidence, and installing back doors and Trojan software to guarantee easy access to the network at a later date.

There are three basic actions taken by an attacker when executing a computer attack:

Prepare. During this stage, the attacker collects network configuration information using port scanners, banner grabbers, and sniffers. The most important pieces of information are computer IP numbers, operating systems, and open ports with their associated listening software type and version.

Exploit. The attacker has identified a vulnerable piece of software and attempts to exploit it during this stage. This is where the attacker gains privileges and information required. The intruder may execute multiple attacks during this phase.

Leave behind. Once the attacker has obtained the required access level via exploitation, the intruder often installs additional software to allow easy access to the network in the future. Such "leave behinds" might be Trojan software, network sniffers, or additional back-door network services.

Cleanup. At this stage, the attacker attempts to clean up any evidence left by actions

completed in the previous stages. This includes restarting daemons crashed during exploitation, cleaning logs and other information, and installing modified system software designed to hide the presence of other software from normal system commands, such as `ps` and `top`.

The existence and duration of each phase of an attack depends partially on the skill and nature of an attacker. At the bottom of the skill totem are the "script kiddies," novice computer users who download sample exploits from Bugtraq [31] or other vulnerability websites. They then use these exploits and attempt to run them on entire networks, without regard to whether the target system is actually running the vulnerable software. This behavior is usually very "noisy," meaning that it is easily detectable by most intrusion detection systems and appears in most of the log files on the target computer. In addition, if the exploit fails, it often causes the vulnerable software to crash, leaving a trail of failed attacks that is easily traceable back to the attacker. Script kiddies generally spend little time in any stage other than the exploitation stage, as they are more entranced by the "coolness" of "hacking" and have no real intrusion goal other than compromising many systems.

As an attacker increases in knowledge, the amount of time spent in preparation and cleanup phases increases considerably. In addition, the types of attacks executed begin to favor those that do not crash the target software, leave little trace that the machine was ever attacked, and are invisible to intrusion detection systems. Finally, the number of attacks within the exploit phase also increases, as skilled attackers will ofter use a series of seemingly low-level attacks to continually upgrade access levels.

Once an attacker advances beyond the realm of the script kiddie, the intruder often has a specific goal in exploiting the network. Whether this goal is to deface a website, obtain pre-release software, or something more nefarious like stealing credit card numbers, the attacks are more likely to be specifically targeted at a certain objective, rather than simply trying to get into the network. This goal directed behavior, coupled with an attacker's knowledge, leads to multi-component attack paths that traverse several hosts on the path to the goal state.

Other factors, such as initial location of the attacker, prior network information, and type of exploit also determine the amount of time spent in each stage of an attack. For example, an attacker that begins an attack within an organization's internal network has

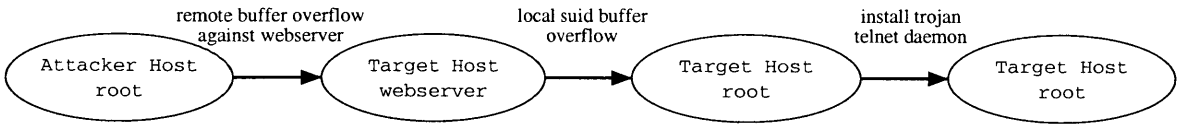

Figure 2-1: An example attack path.

access to a much broader range of network information and connectivity, allowing for quick preparation, whereas an outside attacker requires more time to determine the same amount of network information. Similarly, different exploits require more or less cleanup, depending on how noisy the exploit was, and whether it crashed the vulnerable software.

2.4 Attack Graphs

Attack graphs are a concise way of representing network attacks. Each component attack comprises a transition between two states. A series of attacks then represents a directed path. All the attack paths against a network can be merged to create an attack tree. This attack tree can then again be collapsed into a single graph by combining like states and presenting them to the administrator. Such a graph allows an administrator to determine the most probable attack paths and most security-influential hosts and vulnerabilities by analyzing the "bottlenecks" in the graph.

The nodes of an attack path represent the current state of the system. This includes the current privilege level of the attacker, previous privileges on all machines in the network, and all of the modifications to the network made by the attacker. In the sample attack graph of Figure 2-1, the nodes depict merely the current host of the attacker and the current access on that host, however, the entire state is assumed to exist at each node.

The directed edges of an arc correspond to the actions taken by an attacker including exploits, scans, remote logins, etc. Each action models a state transition. For example, in Figure 2-1, the attacker first executes a remote attack against the web server of the target host and obtains the same privileges as the web server. The intruder then proceeds to exploit a local program that runs with root permissions to obtain administrator access on the target. Finally, the intruder installs a Trojan telnet daemon to ease any later access into the network. The actions performed to transition from one state to another need not simply be exploits, but could be as innocuous as scanning the network or "pinging" a host.

In general, there are many different attack paths for any given network topology. The

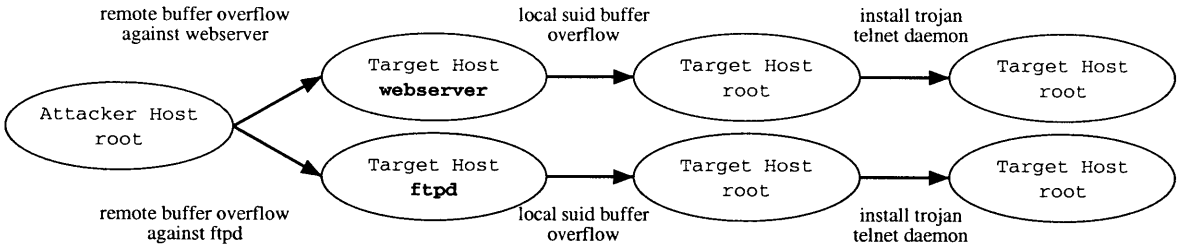

Figure 2-2: An example attack tree.

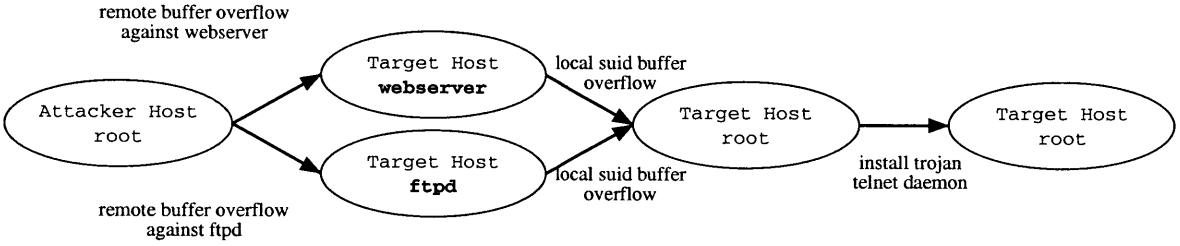

Figure 2-3: An example attack graph.

number of paths grows quickly as the network grows, since more machines translate to more vulnerabilities. All of the possible attack paths can be trivially collapsed into a tree with the same root node, as in Figure 2-2. This attack tree can then be further collapsed into a graph after noting that there are replicated states within the tree. Several combinations of attacks might arrive at the same privilege level and network state. In Figure 2-3, this is shown after the local suid exploit, at which point both of the paths in the graph have root access on the target host. It is important to note that, even though each path in the graph at one point has another type of access (web server, ftpd), once administrator (root) access is achieved on the target host, this is irrelevant. We assume that the exploits did not crash the respective servers, otherwise, this graph would be incorrect, and the resulting state after the local suid exploit would be different, as one path would have a crashed ftpd and the other a crashed web server.

Construction of an attack graph requires several pieces of information about the type of attacker, underlying network topology, and number and type of attacks available to the attacker. Figure 2-4 illustrates these input components. Only three inputs, the attack model, network and host vulnerabilities, and network topology are essential to the creation of a useful attack graph. The attack model defines the state transition relation of an attack by stating its requirements and effects of executing an attack. The network topology limits the physical paths that an attacker can traverse within the network, subject to network

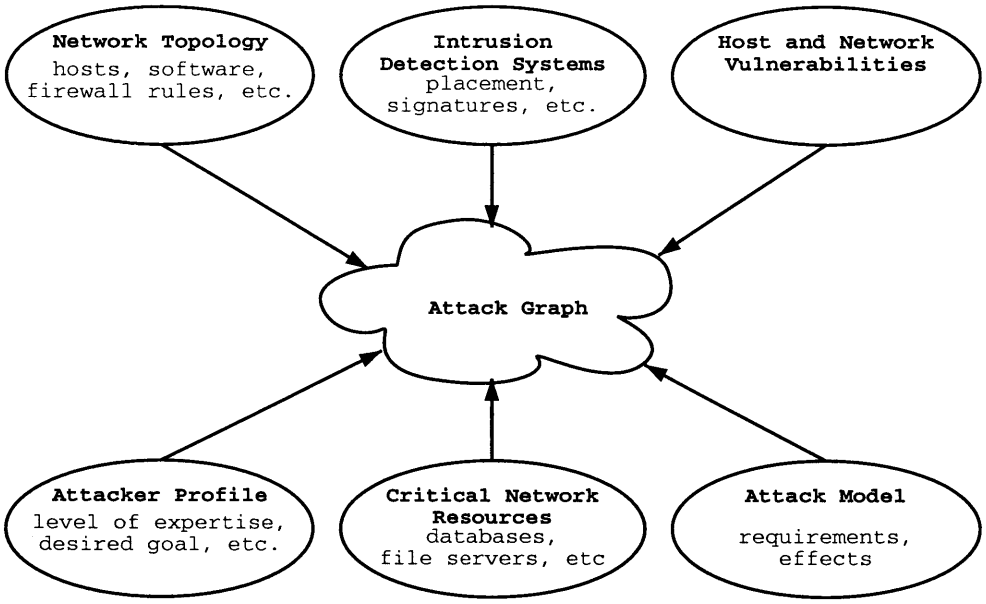

Figure 2-4: Information necessary to create an attack graph.

connectivity and firewall rules. The host and network vulnerabilities and configurations define the possible set of initial actions that the attacker can take against the network using attacks from the attack model.

The other three inputs to the attack graph, attacker profiles, intrusion detection systems, and critical network resources, are not required to generate an attack graph, however they increase the utility of the constructed graph. The attacker profile defines the starting state of the intruder, as well as the methodology that the attacker uses in choosing the next attack to execute. This enables the administrator to optimize a network's security against novice outside attackers, while accepting the possibility of an insider attack. A list of critical network resources also allows the security administrator to prune the complete attack graph to only those states which are judged critical, such as not allowing attacker access to a central billing database. Finally, the placement and type of the intrusion detection systems allow the graph generator to determine which paths are visible.

The resulting attack graph of a network is a valuable security tool [33]. Ideally, the attack graph contains all of the possible paths that an attacker can take through the network, pruned to those of most interest to the security administrator. An administrator can then easily identify critical states that the attacker should be kept out of, and proceed up the graph to find ways to combat the attacker. For example, in Figure 2-3, it might be imperative that the intruder not get root access on the target host. The administrator

can locate the first states in the graph at which the attacker obtains root, and then trace the graph upward from there to find the paths the attacker traversed. In our example, the administrator can tell that both of the paths use the local suid exploit to upgrade privileges from a user to root. Simply by removing or fixing this program, the attacker can be stopped from further exploiting the network. In a real situation, however, remote access of any kind is a much more dangerous threat than a local privilege upgrade, therefore the administrator might opt to fix both the web server and the FTP server. If fixes aren't available, the servers could be turned off or firewalled, again preventing attacker access from the outside. Once the administrator is satisfied with the fixes, the attack graph can be recreated to determine whether the fixes provided any new vulnerable access points.

Attack graphs can also be used to enhance general security. If the attack graphs do not provide any attack paths to critical network resources, several general security metrics can be gleaned from the structure of the graph. For example, an action that is repeated many times within graph can be construed as being an easy, and perhaps necessary, exploit for an attacker. Eliminating this vulnerability can provide an increased level of general security within the network as a whole, in the face of new exploits.

It is important to limit the type of actions allowed within an attack graph to keep it simple enough to be analytically useful. For example, if we allowed the command `ls` (the UNIX "list files") as a valid action, we would create many singleton cycles within the graph, obfuscating the more important information. There is thus a trade-off between the number and type of actions and the usefulness of the resulting attack graph. When too many actions are allowed, the graph becomes fraught with cycles and meaningless paths. Too few, and the graph does not encapsulate enough attack behavior necessary to combat intruders.

While attack graphs are a necessary security tool, creation has often been done by hand. This is tedious, time consuming, and error prone, as graphs for large networks can be quite large and complex. NETSPA automates much of the attack graph generation procedure, while also providing a reusable framework for constructing graphs on a variety of networks.

2.5 Existing Security Software and Resources

NETSPA uses information from a wide variety of existing security sources as input to its graph generation procedure. Each of these sources provides a necessary graph input

component, however alone they do not have the capability to construct a complete and useful graph.

2.5.1 Software Vulnerability Repositories

Several repositories of software vulnerability information exist on the Internet in the form of databases, newsgroups and web-sites. Possibly the most famous and widely used is Bugtraq [31], a mailing list hosted by SecurityFocus [30] frequented by all of the major software vendors and security organizations, as well as many individuals. Here security organizations and individuals post new software vulnerabilities and sample exploits as evidence of an actual problem, sometimes in excess of a hundred messages a day.

CVE, the Common Vulnerability and Exposures list [2], is an attempt to create a "list of standardized names for vulnerabilities and other information security exposures." This standardized naming scheme allows information about various vulnerabilities to be shared, cross-referenced, and distributed. CVE attempts to categorize more than simply software vulnerabilities, but also exposures, those actions taken by an attacker that don't directly compromise the target system but provide additional information to the attacker, such as a network scan or finger probe.

The ICAT Meta-base [4] uses the CVE naming scheme to provide a great deal of meta-information along with the attack names and descriptions. This meta-information includes the locality of the attack, type of vulnerability, exploitation consequence, and several others. In addition, ICAT provides links to possible patches and updates that fix the vulnerability.

Vulnerability information repositories, such as CVE, ICAT, and Bugtraq, generally contain enough information to create the necessary state transition model of an attack, possibly the most important input to an attack graph generator. On their own, however, these repositories do not have the computational ability to turn the natural language descriptions of attacks into a valid computational model of the form required to generate an attack graph. NETSPA thus utilizes the meta-information and descriptions from the information sources to enable a user to create the attack model.

2.5.2 Port Scanners

Port scanners occupy an important place at the low end of the security spectrum. The port scanner is a remote tool that attempts to connect to a computer in a variety of different

ways. If a connection is accepted on a certain port, the port scanner then knows that there is software listening on that port. From this, a user can infer what programs are running on the target system. In addition, full featured port scanners such as nmap [11] can use clever packet creation and response to make a guess about the operating system type and version. By cleverly crafting the probe packets sent to a remote host, a run of nmap can be almost invisible to a probed computer. This sort of technique can force the target software to respond, but not initiate any sort of connection. A good port scanner can compute a fairly accurate view of the network topology, however it has no notion of software vulnerabilities or attacks, and thus cannot be used alone to construct attack graphs.

2.5.3 Vulnerability Scanners

Vulnerability scanners are, in principal, one of the more basic forms of security software. A scanner first collects information about a computer, such as IP address and software versions, by attempting to remotely connect to the host and note what ports and services are available, similar to a port scanner. In addition, several scanners support "banner-grabbing" techniques that actually connect to the target host and note the data (banner) that is returned by the listening application. In the case of a web server, the default banner often tells the type, version, and operating system of the target computer. If the scanner is being run on the same host that is being scanned, a more extensive amount of information can be gleaned and local software is often cataloged. Next, the scanner compares this information against an internal database of known vulnerabilities and the user is flagged when a match, signaling a vulnerability, is found. Nessus [8], a full-fledged security scanner, supports its own scripting language and currently has a database of over 1000 vulnerabilities. SU-Kuang [27], a local scanner, and Net-Kuang [38], a networked version of SU-Kuang, scan for configuration vulnerabilities, those weaknesses created by incorrect configuration of various software components. Their database is not explicit vulnerabilities, such as buffer overflows and denial of service attacks, but invalid configurations. The biggest weakness of security scanners is that they only identify single point system vulnerabilities, and fail to extrapolate this information to find attack scenarios and graphs. In addition, most vulnerability scanners do not have a notion of action effects, only the requirements necessary to execute the action.

2.5.4 Intrusion Detection Systems

IDSs do not prevent attacks from occurring, but instead attempt to detect them as they occur by analyzing behavior. Snort [28], for example, an open source network-based IDS, compares network traffic against an internal database of known attack signatures. When a match is found, Snort generates an alert. Host-based IDSs, such as USTAT [37] and Zone Alarm [39], can detect attacks, including buffer overflows, by determining when software components perform functions that are not part of their normal specifications, such as changing passwords or connecting to the Internet. It is then up to the user to determine whether an actual attack is occurring, or whether the alert is a false alarm. Reducing the incidence of false alarms while keeping the detection rate high is one of the primary research areas of most intrusion detection systems. An IDS is typically a second line of defense, however some security policies require that certain functional "holes" remain in a network, and an IDS provides another level of protection and confidence in the security of a network by detecting when these "holes" are used.

Chapter 3

Related Research

Generating attack graphs is not a novel idea. Several systems have been developed to automatically create attack graphs, and several languages exist to describe attack components. This chapter details these systems, and examines the strengths and weaknesses of each against the methods employed by NETSPA.

3.1 Attack Languages

Quite a bit of effort has gone into creating languages to break down computer attacks into reusable components and create standardized attack descriptions, such as what an attack requires in order to run, and what the effects of the attack are on the network and the state of the attacker.

The STATL language, described in [9], was designed to assist intrusion detection systems in detecting attack scenarios. In addition, a major design goal of STATL was to be able to link together both new and existing attacks into a scenario. To this effect, STATL necessarily abstracts away much of the low-level system details, such as software versions, that are necessary when creating attack graphs using real networks. It has, however, been successfully used as part of the STAT intrusion detection framework [37] to describe many attacks. The Snort signature language [28] has also been used to create attack signatures. In fact, Snort signatures can be converted into the STATL language to populate the NetSTAT IDSs with signatures [10].

LAMBDA [7] is another language designed to work hand-in-hand with intrusion detection systems. LAMBDA allows a scenario to be built incrementally from increasingly

complex modeled actions, using pre- and post-conditions. Each scenario must be pre-defined by hand and LAMBDA does not have any native support for automatically combining attacks to create an attack scenario. Also, like STATL, LAMBDA is intrinsically tied to intrusion detection systems, and has recently been used to correlate IDS alerts as they are received into scenarios [6].

In [24], Ramakrishnan and Sekar propose a new object-oriented modeling language, where each aspect of a system is modeled as a communicating "object." This allows a model of an entire system to be built, at which point the model can reason about possible conditions in which a vulnerability would arise. This type of language is adept at discovering configuration vulnerabilities, as well as those vulnerabilities related to the inherent design of the system. Simple vulnerabilities, however, such as specific buffer overflows, are beyond the scope of the model. Finally, modeling each system in a large network to include file-systems and all running processes would take a tremendous amount of time, and be computationally infeasible to simulate quickly. A recently proposed language called CAML [1] shares this complexity, but evidently it has yet to be applied.

JIGSAW [36] is a requires/provides language designed to join atomic attacks together to create scenarios, with the "provides" section from one action enabling the "requires" section of another attack. The requires/provides ideas behind JIGSAW are very similar to those in the language developed for NETSPA, however JIGSAW fails to describe essential state transitions, such as modifications to the state of the network. In addition, JIGSAW has only been demonstrated on a small number of attacks.

REM, the simple language developed for NETSPA, uses to notion of requires/provides as state transition elements. In addition, REM is very simple to allow for quick and easy modeling of a large number of actions.

3.2 Attack Graph Generation Systems

Because attack graphs are so difficult to create by hand, several graph generation systems have been developed. These systems tend to be very similar to NETSPA in terms of generating attack graphs, but most have limitations that NETSPA attempts to overcome.

In [26], the Ritchey and Ammann create a simplified state model of an attack and use specialized model checkers, such as SMV [18] and SPIN [13] to prove counter-examples to the

assumed security of the modeled network. This has the advantage of using existing model checkers to quickly and efficiently obtain results. The model itself, however, was little more than an extension of a security scanner's database; it modeled only simple configurations of each host and used the current access level as state. While this model works for many attacks, there are whole classes of attacks that this ignores, such as sniffing traffic on a network. Also, an attack graph is not generated, only the single path that contradicts the original assertion. This does not allow the security administrator to reason about the overall security of the network, only the existence of a single path from attacker start state to a goal state. Finally, entering the initial network topology information and host vulnerabilities was all done by hand, coding directly in the model checker's state transition language. This large setup cost is prohibitive for normal security administrators.

In a similar vein, Sheyner et al. also use a modified model checker to create attack graphs [33]. The authors modified the NuSMV [3] system to generate a list of all of the possible counter-examples to their hypothesis, such as "root access on host A is never achieved." This provides a sub-graph of the entire attack graph such that every path ends in a state that violates the original hypothesis. As this model checker works from a goal state backwards, the state space explored by the graph generator has the potential to be smaller than that explored by a similar forward chaining algorithm. This can result in time and space savings, however the additional information found in the complete graph containing all of the possible attack goals requires many passes of the system, one for each goal. Despite the efficiency promised by a model-checking system, the NuSMV system required 2 hours to compute a graph on a network with only 4 hosts and 8 possible attacks. This run might be intractable for larger and more realistic networks.

The work by Sheyner et al. improves considerably over previous model checking graph generators by allowing the network configuration information to be entered using XML structures, as opposed to the native language of the model-checker. The state transition actions, however, must still be programmed directly into NuSMV. The authors acknowledge this issue, and are attempting to create a repository of attacks and actions. NETSPA attempts to overcome this sort of limitation by enabling an extremely simple attack definition language that greatly simplifies the task of creating and distributing new attacks and actions. In NETSPA, the attack information is entered as a series of statements into a simple database instead of adding code using a model-checker's native language.

The software described by Swiler et al. in [34] is the most similar to NETSPA. The system generates attack graphs using a forward chaining algorithm and also takes into account basic network topology. Each action is modeled as a template that can "fire" if certain conditions are met, and each action rewrites the network to change the state. The authors also assign several cost metrics to each action, and attempt to highlight all paths within a given ϵ of the shortest path. This attack model, however, is overly general and has only been shown to work on a very small network of two hosts and five possible actions. Also, the network topology model fails to depict several network fundamentals, such as firewalls and routers. Finally, the authors employ several graph pruning techniques, such as redundant path elimination, to turn the final attack graph back into a tree. This loses important security information in the process such as additional paths to a goal state.

Chapter 4

NetSPA Design

NETSPA was created to fill a void in existing security software. The primary design goal of NETSPA was to create a system that could automatically compute complete attack graphs for real, user-specified networks. This, in turn, lead to three separate sub-goals: allow a user to easily define a network and its resulting configuration, enable quick modeling of realistic actions, and efficiently compute worst-case attack graphs with sufficient meta-information to be easily useful to the user. Secondary to the notion of attack graphs was that of simplicity and information reuse, most notably in the action specifications.

The definition of a network includes many separate components. Each host must be individually defined with running software, ports, and connectivity to a network through interfaces. In addition, hosts can behave as a gateway or firewall and can contain a set of rules to limit the traffic allowed across the gateway host. A host may also behave as a host- or network-based intrusion detection system. Finally, trust relationships between hosts can be established to model normal user traffic.

The worst-case graphs generated by NETSPA illustrate all possible cyber-attack paths. They do not model physical attacks or human engineering attacks. Graph generation does not take into account the skill or predisposition of the attacker. It also assumes that attempts at "security by obscurity," such as passing SMTP traffic through the firewall on a non-standard port, fail. In addition, the model of an IDS is assumed to be "best-case." A host-based IDS always detects an attack launched against it, while a network-based IDS always detects attacks that are visible on the network if it has a signature for the attack.

NETSPA is divided into several modules to achieve its goals, each component and re-

sulting connectivity is shown in Figure 4-1. As seen in the upper left of the figure and illustrated in Section 4.1, the software database is the repository of software information used by NETSPA to make software names consistent. The software database is used by both the action database (Section 4.2) and the input filters to the network model (Sections 4.5 and 4.4, respectively) to create a consistent software naming scheme among network configuration and action definitions. The action database shown in the middle left of Figure 4-1 contains information about every possible action that an attacker can execute against a user-defined network. The creation of this network is aided by the use input filters, shown in the upper right of Figure 4-1, which populate the network model with network configuration information. This network model is then used to create an initial network state, which is provided, along with the database of possible actions and set of existing trust relationships (Section 4.3), to the computation engine (Section 4.6). The computation engine then creates a worst-case attack graph for the specific set of inputs.

4.1 Software Database

The software database contains records of all software run on all hosts within the network. This includes operating systems, network daemons, and applications. The software database exists primarily as way to enforce consistency in software names between the input filters and the action definitions. For example, a version of the popular Windows operating system might be written as "Windows NT 5.0" or "Windows 2000." The software database contains a single record that embodies all of the different "spellings" of a piece of software, allowing the computation engine to function on software indices, as opposed to names. The database can be populated directly from the software list distributed with the ICAT database. As much of the action information is also derived from ICAT, the ICAT software list should be complete against all of the actions within CVE.

4.2 Action Database

The action database is the primary repository of information about all of the types of actions that can be executed, including network scans, remote logins, exploits, Trojan installation, and other behavior common to computer attacks. Every action that can be performed on a network must exist within the action database to allow NETSPA to use the action

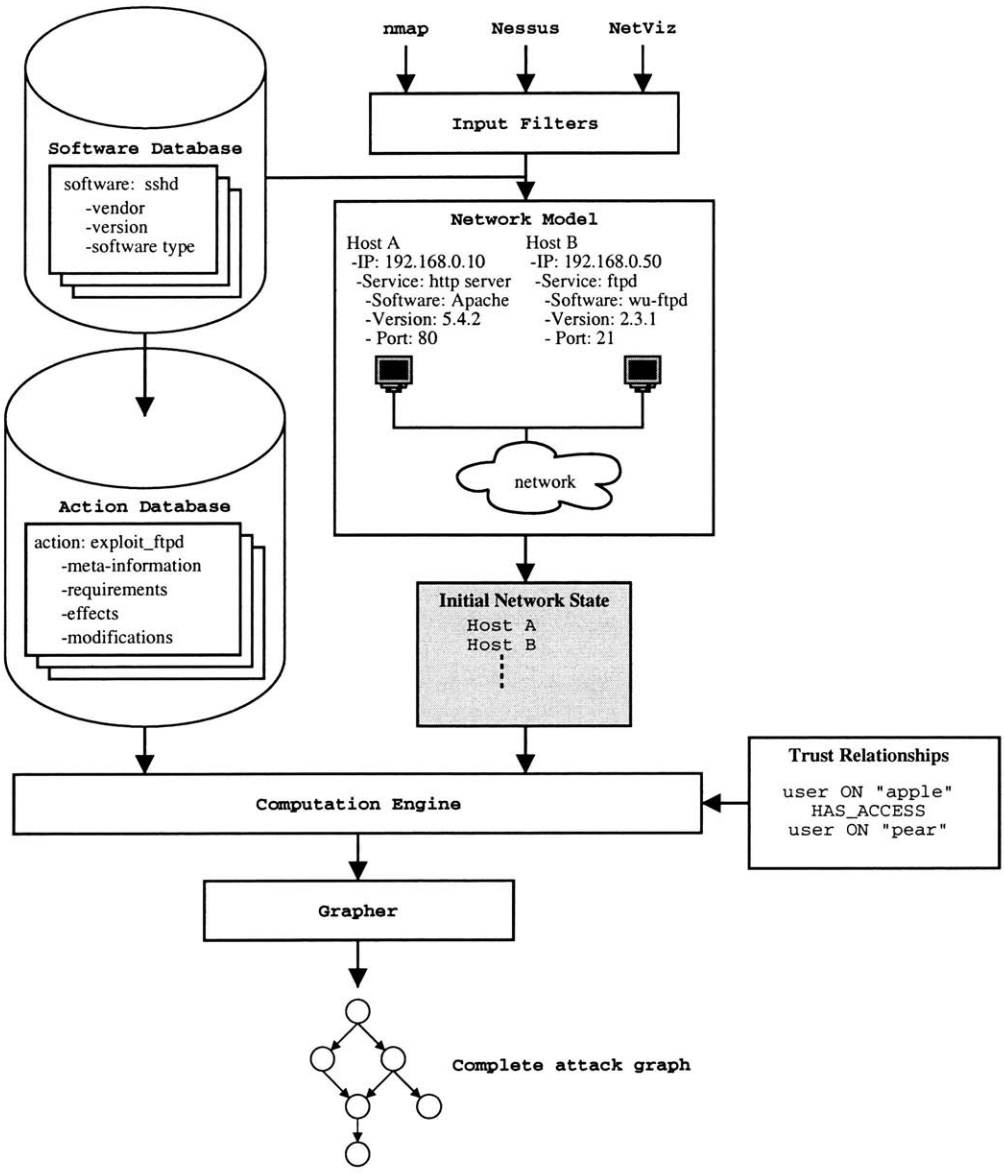

Figure 4-1: NETSPA component diagram.

in constructing an attack graph. Additionally, each action's transition behavior must be modeled for NETSPA to know how to join actions together into a graph. As there is currently no central database of machine parseable action definitions, each action must be entered into the database manually. The simple attack language chosen for NETSPA was designed to facilitate quick attack modeling, allowing a substantial database of actions to be entered without significant effort. Also, an action only needs to be entered once into the database, which can then be shared among many users. The attack language designed for NETSPA was inspired by several existing attack languages, including those of [7, 36]. The following section discusses this language.

4.2.1 Attack Model: Requirements/Effects/Modifies

The model used to define the possible action state transitions is one of the most important pieces of the system, as this defines the expressiveness of each action. Too specific and the model will fail to capture many of the important actions that an attacker can take; too general and the feasibility of implementation drops significantly. For example, a language that can only model direct remote root attacks will fail to comprehend the more subtle attacks, such as modifying a `.rhosts` file to allow remote unauthenticated logins from any host. Alternately, a language that is overly complex for the domain, such as that found in [24], will require the user to create definitions for every piece of the system, including files and permissions of an entire file-system.

The Requirements/Effects/Modifies (REM) keyword language was created to model the requirements of an action, as well as the effects of the action once it is performed. This simple language allows an action to be specified in terms of its state transition behavior. The REM language has three components: requirements for the successful completion of an action, effects on the state of the network and attacker knowledge, and modifies to the current attacker privilege and login state. A more thorough description of the language is given in Appendix A.

Each time an attacker attempts to perform an action, certain conditions must be met for the action to succeed. For example, if the attacker cannot even connect to a machine within the network, possibly because of a firewall, then the attacker cannot execute a remote attack against the protected machine. Similarly, an attacker cannot exploit a local program until logged into the host where the program resides. The conditions necessary for an action

to be successfully performed are the requirements of that action.

Each successfully completed action affects a change on the state of the network. For example, a remote buffer overflow of a network service such as `sshd` will most likely cause the service daemon to crash, while an `.rhosts` exploit might establish a trust relation between two hosts, allowing the a user on one to log into the other host without any password or additional authentication. This network state change is the effects section of an action's REM specification.

Finally, an action can modify the current state of the attacker, including privilege level and current host. If an attacker logs into another machine, the current host and user are modified to that of the new machine and new user. This is considered a modification of the attacker state, and is part of the modifies section of an REM definition.

A complete REM action definition is a set of statements divided into the three previously described sections: requirements, effects, and modifies. The statements within each section are unordered. A sample REM definition for a remote root-level `sshd` overflow is shown in Figure 4-2. The capital words are the keywords of REM that define the information contained within the statement. Words within <> are variables that are bound at the time of action execution. Quotation marks are also required for all string literals within a statement.

In the requirements section, there are two statements which define the necessary state of the network for the action to complete. The first of these statements is a `RUNS` clause, indicating that the `<target_host>`, the host against which the action is being performed, must be running the correct software, as defined by the software signature

```
OpenBSD:OpenSSH Server:2.3:2.3.5 b37
```

The signature of a piece of software is simply a string of the form

```
vendor:product name:common version:exact version
```

Software can also be defined as `TYPE software_type`, where `software_type` defines a valid type of software, such as "remote login server." There can only be a single `RUNS` statement matched within the requirements section of an action. Multiple `RUNS` clauses can exist within the requirements to allow for an attack that works over several versions of the software. The single matched piece of software binds the `<exposed_software>` and associated running `<port>`. This port is set to zero if it is a local application. Any additional software that is necessary for the action to complete can be specified using `ALSO_RUNS` clauses, with the

same syntax as a RUNS statement.

The other statement within the requirements section is the CAN_CONNECT statement, which specifies the necessary network connectivity. In this case, <current_host>, which is bound to the host that the attacker is currently logged into, must be able to connect to <target_host> on <port>, the port that the exposed piece of software is running on. In this case, this is most likely port 22, the standard sshd port, but could be any port on which there is an sshd of the correct signature.

The effects section only contains a single statement. This statement modifies the state of the <exposed_software> (sshd), and forces the software into DOS mode, which is an undetermined state meaning that the software is not accepting new connections, whether due to a software crash or a denial-of-service attack.

Finally, the modifies section changes the <current_host> of the attacker to be the <target_host>, and the <current_user> to be root on the new machine. There are four different access levels within NETSPA: none, nologin, user, and root. Privilege level none provides no access at all on the target host; user has access equivalent to a normal user of the system; root corresponds to having full control over the target host. The nologin user is equivalent to having knowledge about the users of the system but no way to log in, perhaps because of a missing password or a closed account. This user type is often the precursor to a brute-force password breaking attack.

There are many systems with several different access levels that cannot be described by REM. In addition, there are several actions that cannot be exactly defined with REM. The majority of the user groups and actions can be correctly expressed using only the three user groups provided and the simple REM statements. For those few actions that require a more specific language or user definition, it is possible to model the action to be less restrictive, and then allow the user to manually filter this extraneous data. For example, an action that provides 'guest' account access on a Windows 2000 host can be relaxed from the actual action into an action definition that provides 'user' level access. Similarly, an action that allows a user to modify the /etc/passwd file on a UNIX system can be said to provide root access, even though the actual action does not directly give root access. This relaxation of actions results in the creation of additional nodes and paths in the final attack graph that do not necessarily map to real-world situation. As NETSPA strives to provide worst-case attack graphs, this was deemed an acceptable trade-off for the simplicity of the

sshd Remote Buffer Overflow

Requirements	<target_host> RUNS "OpenBSD:OpenSSH Server:2.3:2.3.5 b37" <current_host> CAN_CONNECT <target_host> PORT <port>
Effects	<target_host> RUNS <exposed_software> DOS
Modifies	<current_host> = <target_host> <current_user> = root

Figure 4-2: The REM definition for an sshd remote buffer overflow.

REM language.

Two more keywords exist within the REM language that are not in the previous example: HAS_ACCESS and TRUST_RELATIONSHIPS. When used within the requires section, a HAS_ACCESS statement requires that a user U_i on host H_i must have access U_j on host H_j for the action to complete. A HAS_ACCESS statement within the provides section implies a trust relationship between the two hosts, enabling the attacker to remotely log in to host H_j from H_i without additional authentication. The TRUST_RELATIONSHIPS keyword can be placed in the effects section of an action definition, and provides the attacker with the ability to obtain all clauses in the set of trust relationships, described in the following section, originating from hosts on the current subnetwork. This provides worst case data capture functionality as an attacker will obtain all of the remote login information immediately for the purpose of generating a worst-case graph and finding actual system vulnerabilities, even though, in a realistic situation, the capture of all of the required information might take days or weeks.

4.3 Trust Relationships

The trust relationships, shown in the bottom right of Figure 4-1, are input provided by the user that define the allowable remote authentications between hosts. This is primarily used to model essential user traffic, such as a user obtaining mail via POP or an administrator remotely logging into a DNS server. It is assumed that the authentication mechanisms are point to point and directed, i.e. a specific user can only log in to the mail server from a

specific host, but not the other way around. Trust relationships represent all authenticated user traffic between hosts in the network. The relationships are a set of REM `HAS_ACCESS` statements that enumerate the allowable host-to-host user transitions. For example, if a user, who normally works at host "apple", is allowed to remotely authenticate from "apple" to another host "pear" with user permissions, the following statement would exist in the list of trust relationships:

```
user ON apple HAS_ACCESS user ON pear
```

These trust relationships allow NETSPA to model the the behavior of an attacker attempting to capture user-entered data. It is assumed (as a worst-case) that an attacker with a network data capture device, such as a sniffer, can obtain all user authentication traversing the network. As explained previously in Section 4.2.1, an attacker that installs a sniffer will be able to detect and read all of the network traffic modeled in the trust relationships originating on the same subnetwork as the position of the sniffer. Since the trust relationships are point-to-point, however, the attacker must obtain the necessary privilege level on the "start" host to be able to transition, using the trust relationship, to the provided privilege level on the "end" host. In our previous example, a sniffer installed on the same subnetwork as host "apple" would learn that a user on "apple" could log in as a user to host "pear." The attacker would have to first obtain `user` access on "apple" in order to be able to authenticate to "pear" using this trust relationship.

This model of network data capture assumes that all of the traffic on the network subject to capture by the attacker is sent "in the clear," i.e. unencrypted, and that all of the the remote authentication servers use some sort of access control list (ACL) to limit the allowable user/host login combinations. The ACL behavior is due to the point-to-point nature of the trust relationships. In a realistic situation, a remote login server might not use an ACL, meaning that a valid remote user could log in from any host. This ACL behavior cannot be concisely modeled using the REM syntax. A workaround that enables modeling of a non-ACL server is to create trust relationships from all "local" user machines to the remote login server. For example, a valid set of organizational trust relationships would allow all internal machines on an organization's network to connect to the mail server as `user`. This would be the equivalent of allowing users to check their mail from any machine on the internal network. This is not entirely correct, as non-ACL servers would permit valid logins from an attacker's host. The trust relationships attempt to model actual worst-case

user behavior and establishing such a relationship between unrelated hosts would violate this model.

The trust relationship model can be extended to include SSH, SSL, and other secure protocols by requiring the privilege level at the start of all such connections to be root. This assumes that encrypted network transmissions can only be compromised as `root` on the source machine using a local key-stroke monitor or other such techniques. If the previous trust relationship example modeled a secure login, the statement would become:

```
root ON apple HAS_ACCESS user ON pear
```

This is slightly ambiguous and taken to mean that either an administrator can log in from "apple" to a user on "pear," or that a user on "apple" is transmitting over the network using a method that cannot be directly read, and requires that some form of local data capture be used on "apple" with administrator permissions. A necessary and useful enhancement to NETSPA would be to augment the syntax of the `HAS_ACCESS` REM statement to directly support encrypted transmissions. This would also require additional software types listing those servers that support encrypted transmission, and was deemed unnecessarily complex for the initial tests of NETSPA.

The list of existing trust relationships is created by the user after the network has been created. Currently, the user must input the statements directly into NETSPA, however inputs could be read from a database of configuration file in the future.

4.4 Network Model

All of the network topology information is stored within the network model. Every piece of information about the network must be present within the model for NETSPA to make use of it. This includes hosts, software running on each host, network connections, subnetworks, firewall rules, and intrusion detection system placement and capabilities. This network information can be imported from outside sources using provided filters and the software database, or entered directly by hand. The computation engine later imports all of the information contained in the network model into a collection of objects that compromise the initial state of the network.

4.5 Input Filters

The input filters provide mechanisms to easily populate the network model with information about the network. Filters can take different forms. For example, a Nessus filter that populates the software information for the hosts might be a perl script that matches hosts based on IP address. NETSPA's primary input filter, however, is a NetViz [22] ODBC interface to the network model. NetViz allows a user to create a graphical drawing of a network with all of the necessary configuration information as shown in Figure 4-3, and then export the topology information to the network model, in our case, a database. NetViz was chosen because of the large body of NetViz users who already create models of their network within NetViz. With only minor modifications to existing network diagrams, NetViz can automatically populate the network configuration information.

4.6 Computation Engine

The computation engine is the heart of NETSPA and is responsible for searching through all of the possible attack states to generate the attack graph. This is accomplished by using actions as state transition elements in a rule-based artificial intelligence system. The computation engine takes inputs from the action database and the initial network state and attempts to match the current network configuration, attacker information, attacker privilege state, and the requirements of each of the actions in the database. If a match is found, the computation engine makes sure that the action provides a new, useful state to the attacker, and, if so, calls a grapher (described in the next section) to create a new node using the currently matching action as the joining arc. This process continues recursively until no additional matches can be found in the network state, at which point the computation engine backs up to the previous state and attempts to execute other actions.

A forward chaining rule-based AI system was chosen for NETSPA for several reasons. Most importantly, this implicitly enforces the state sequencing inherent in the REM language. Because an action's requirements must be met before the action can be executed, an action will not be performed until additional actions are executed that provide the necessary state. This intrinsic action behavior lends itself directly to rule-based AI systems. In addition, a snapshot of the entire state of the system must be kept at each node in the graph, which invalidates the use of simple network algorithms, such as shortest-path ap-

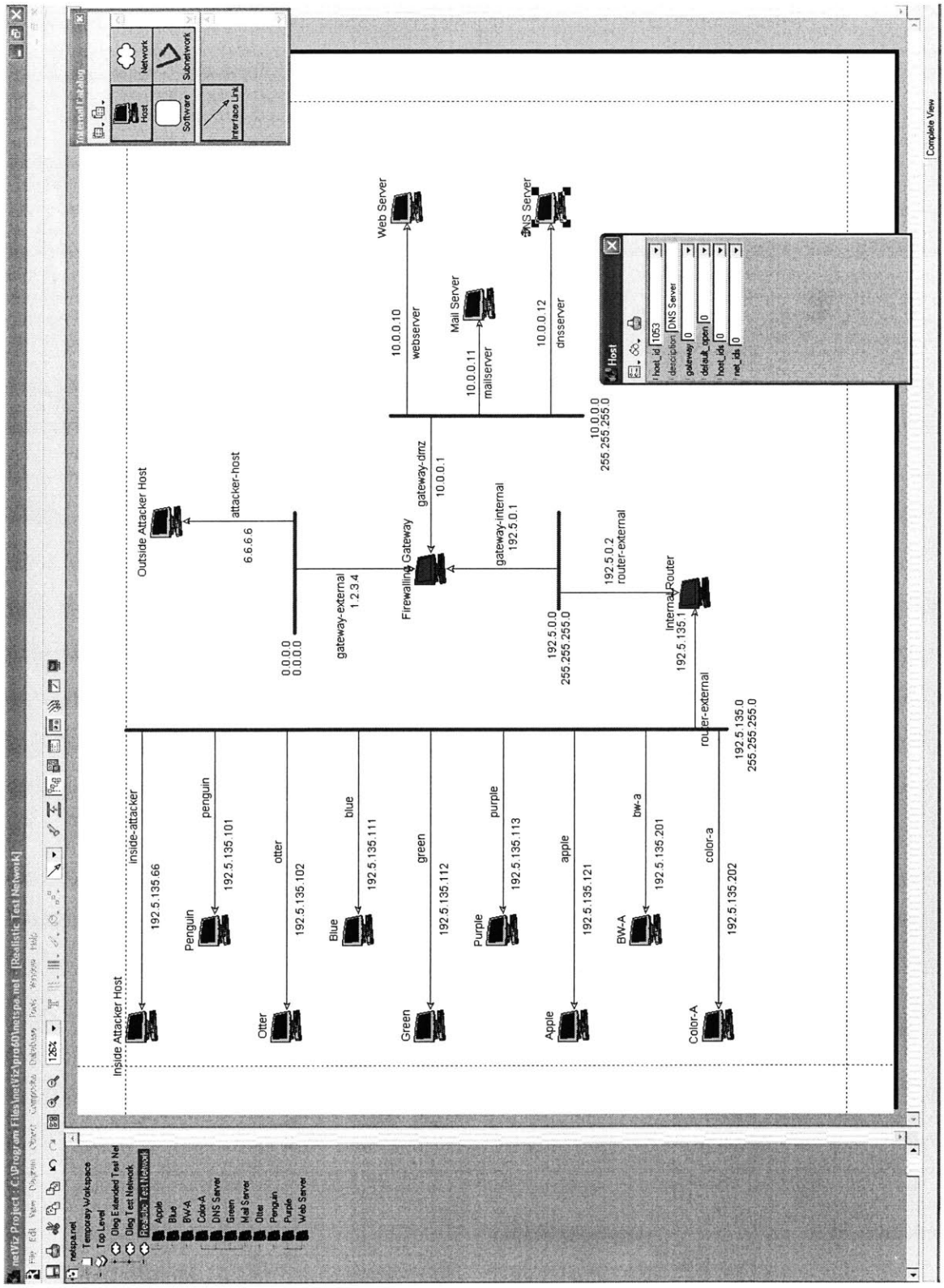

Figure 4-3: Using NetViz to populate the network model.

proaches. It is also simpler than model-checking and more formal methods, as well as being more feasible. In real networks, there will only be a few reachable states and NETSPAwill compute paths in a relatively short amount of time, or the entire graph will not be useful to view and only statistics on paths a few actions long are necessary to demonstrate the insecurity of a network. A limited depth forward search will thus either quickly find the few reachable states or provide information about the explosion of the graph after a limited depth search.

The state explored by the forward chaining algorithm can potentially be much larger than that traversed by an algorithm that works backwards from a goal node. The extra security information that can be gleaned from a graph of all possible paths into a network is significant, and a forward chaining algorithm was selected for implementation. In addition, if the state space is very large and computationally intractable, a forward chaining algorithm can explore the tree, limiting itself to a certain depth, and still provide useful information about the network's security. One goal of this thesis was to determine if simple forward chaining could provide useful and novel information required to design secure networks.

4.7 Grapher

The grapher is simply an interface that receives calls from the computation engine to construct a graph based on the actions executed and the state at each point in the execution. The grapher is also responsible for collapsing the tree provided by the engine into a graph by combining equal states, as described earlier. Finally, the grapher performs any pruning for a specific goal state as required by the user.

Chapter 5

NetSPA Implementation

NETSPA is comprised of several different parts, each of them implemented in a slightly different way. The complete system engine is written in C++ and runs on Red Hat Linux 7.1 [25] using MySQL 4.0.1 [21], unixODBC 2.2.0 [12], and John Maddock's Regex++ regular expression libraries from the C++ `boost` package [17]. To enable the use of NetViz [22] for inputting network configuration data, we used Windows XP Professional Edition [19] and NetViz 6.0 connected to our Linux database server via ODBC.

The rest of this chapter discusses the more interesting implementation details of the various system components, including a description of the different database structures and resulting models, the basic algorithms used in the engine, and the graphing process.

5.1 Databases

The action database, software database, and network model are all implemented using a relational database system. This was done for several reasons, primary among them universal access and ease of modification. Open source RDBMS systems can be accessed in many different ways and the information contained within can easily be easily shared between many different people and applications. In addition, each of the databases has the potential to get very large, and RDBMSs provide a fast and elegant interface into managing large data sets.

The three logical databases together comprise a single physical database, the Network Security Database (NSD). Each NETSPA "database" is represented as a collection of tables within NSD.

5.1.1 Action Database

The action database, as shown in Figure 5-1, is comprised of five tables. Each table contains its own unique integer primary key that can be used to reference the table via other foreign keys. A sample entry into each of the action-related tables is presented in Figure 5-2. The `nsd_actions` table functions as the primary action table that contains all of the basic information about the action. Within `nsd_actions`, the `cve_id` defines the CVE identification of the action, if one exists. The `description` is a large text column that allows a complete natural language description of the attack to be entered. The `locality_information` determines the necessary position of an attacker relative to the target host, "remote" or "local" are the only two current localities. The `icat_severity` is a text description of the severity of the action as imported from the ICAT Meta-base, and `numeric_severity` is the numeric value equivalent. The `publish_date` is the date that the action or vulnerability was made public, and `date_entered` the date that the action was entered into NSD. Finally, `ids_detectable` is a boolean value that determines the action's visibility to a network-based IDS. There is no equivalent for host-based IDSs as it is assumed that all attacks impinging on a host are visible to a host-based IDS. Much of the information in `nsd_actions` can be imported directly from a dump of the ICAT database via provided custom perl scripts.

The `nsd_action_types` and `nsd_action_action_types_map` provide a way to type and categorize actions for further meta-information. An example action type could be "buffer overflow" or "race condition." This information would be entered succinctly in the `type_name` field of `nsd_action_types` and more descriptively in the `type_description` column. The `nsd_action_action_types_map` table simply provides a way to map the type keys to the action keys.

The `nsd_requires`, `nsd_effects`, and `nsd_modifies` tables contain the REM statements necessary for their respective action, mapped via a foreign key column in each table. The requisite REM statement is placed in the `statement` column of the respective table, and the `action_id` field maps to the action key.

The action database can only be partially automatically populated. Most of the meta-information can be obtained through CVE and ICAT. The REM modeling information, however, must be entered by hand for each action. This is where the value of a simple modeling language is shown, because the time to enter in a new action is small, compared

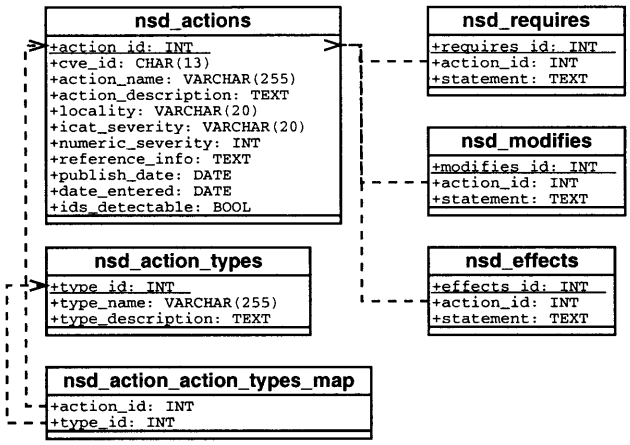

Figure 5-1: Action related tables within the Network Security Database.

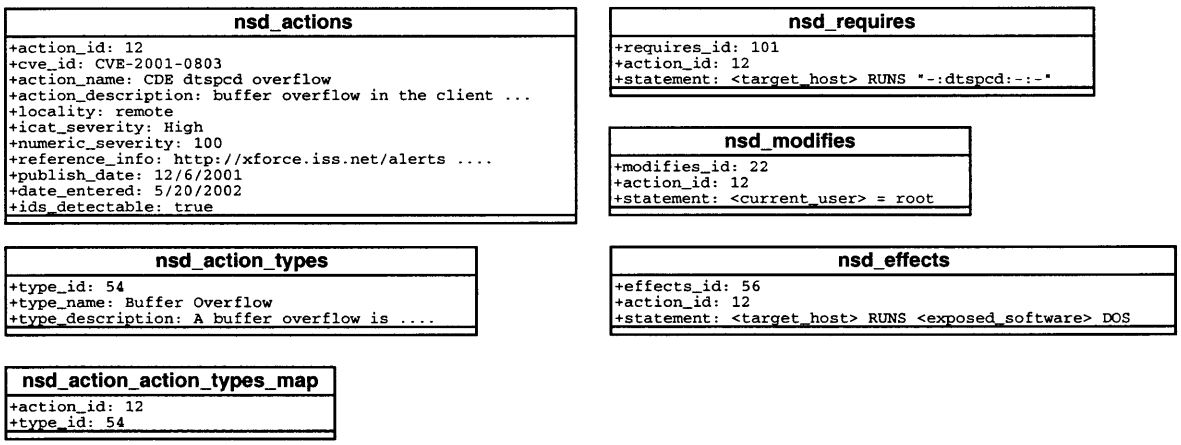

Figure 5-2: A sample entry into the NSD action related tables.

to the other models discussed in section 3.1.

Unlike the network database, the action database stands almost entirely alone and is independent of the network that NETSPA is being run against. This allows all of the NETSPA users to contribute new actions to the database, and allows the database and models to be used for other purposes not directly involving NETSPA. There is, however, a dependence on the constancy of the software database, discussed in the next section.

5.1.2 Software Database

The software database, seen as the tables within the gray box in Figure 5-3, is made up of 3 tables: a table to hold the meta-information such as the vendor and product key, a software types table, and a mapping table to link the two. Like the action database, the software database is separate from the actual execution of NETSPA, allowing the data to be shared among all NETSPA users. A sample entry into the software tables is shown in Figure 5-4.

The `nsd_software` table contains a primary key (`software_id`) to uniquely identify each piece of software, and the `vendor`, `product_name`, `common_version`, and `exact_version` columns identify the software vendor, name, public version numbering, such as the "5.1" in "Microsoft IIS 5.1", and precise versioning, including patch and build level, respectively. The `nsd_software_types` and `nsd_software_software_types_map` define a software type (in fields `type_name` and `type_description`) for possible later reference in an action REM definition. For example, in order to execute a remote login action, a target host must be running a "remote login server," such as rlogin, a telnet server, or an SSH server. Figure 5-4 describes such a mapping between the "CDE" software and "Desktop System" software type.

The records within the software database need to remain static for NETSPA to function correctly. This is due to the fact that statements in the action database refer to a specific signature within the software database that needs to be constant among all runs of NETSPA. To illustrate what can happen when this is not the case, assume that a new action was created and modeled using the software signature

`Open Group:Common Desktop Environment:1.2:1.2.31 p2`

which maps to a valid record within the software database. If the Open Group later changes its names to "Open Software Group" and this update is made in the software database, the action does not map to a valid record within the database. This can result in an action

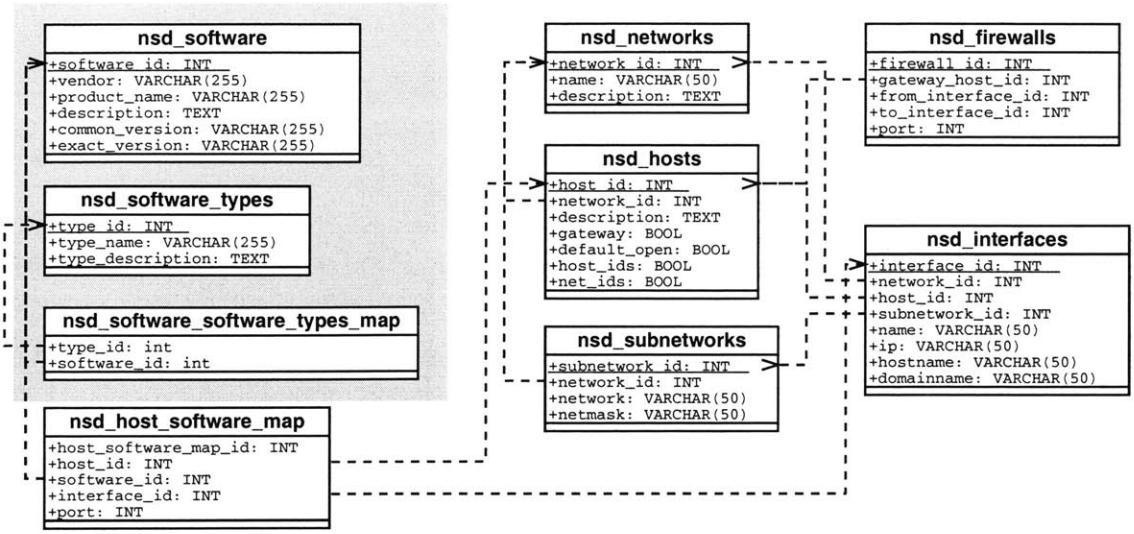

Figure 5-3: Software (gray) and network topology related tables within the network security database.

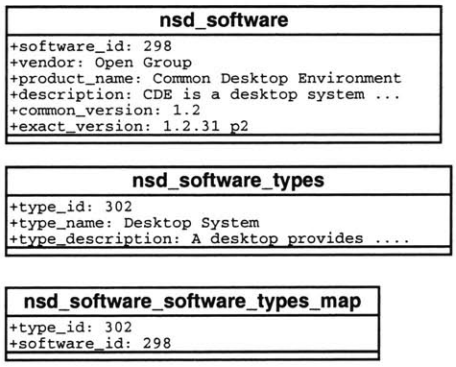

Figure 5-4: A sample entry into the NSD software related tables.

path not being explored because the software signature did not find a match. For these reasons, the index and signature of a piece of software need to remain constant among all runs of NETSPA to allow for the sharing of the action models in the action database. Also, the initial entry of software into the software database can be almost entirely automatically generated from the ICAT software list.

5.1.3 Network Database

The network database is the highly dynamic repository for configuration information of specific networks to run against NETSPA. This information can be entered directly into the database or imported from filters, such as NetViz, into the six tables. The network database is shown as the non-grey parts of Figure 5-3 and a sample entry is given in Figure 5-5

The `nsd_networks` table is the highest level of the network hierarchy, and simply contains a way for many different networks to exist within the database, providing a primary key (`network_id`), name (`name`), and textual description (`description`) of the network.

The `nsd_hosts` table contains a list of all of the hosts, referenced to their particular network using a foreign key mapping in `network_id`. The host also contains boolean columns that describe its state as a gateway (`gateway`), default firewalling rule (`default_open`), and performance as a host- or network-based IDS (`host_ids` and `net_ids` respectively).

The `nsd_firewalls` table defines more rigorous firewall behavior in the form of rules. A rule provides a connection opposite to the default behavior. For example, if the default behavior is closed, a firewall rule would define a valid connection between two host interfaces. Conversely, if the default firewall was open, a rule would specify a denying of the path between two interfaces. A unique table ID (`firewall_id`) is provided, as well as the connection port number (`port`). The other two columns, `from_interface_id` and `to_interface_id`, determine the start and end interfaces, as foreign keys into `nsd_interfaces`, that the rule refers to.

At the same level as a host in the network hierarchy are the subnetworks, defined in `nsd_subnetworks`. The subnetworks are primarily used as a hub for connecting hosts together, and only define their respective networks (`network`) and netmasks (`netmask`).

The `nsd_interfaces` table defines the linkage between the hosts and subnetworks via the `host_id` and `subnetwork_id` fields. Additionally, the interface contains a name in the

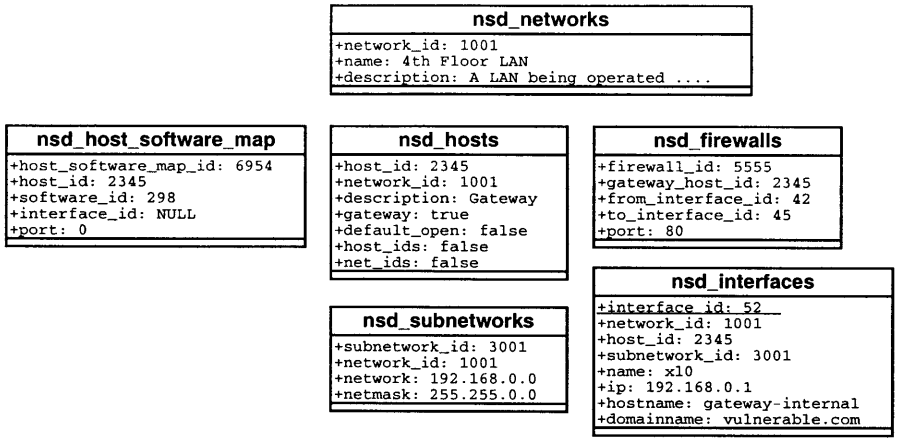

Figure 5-5: A sample entry into the NSD network configuration related tables.

name field such as "xl0," an IP number in ip, a host name in hostname, and a domain name in domainname.

Finally, the nsd_host_software_map designates the software running on a particular host. Since the software could also be running only on a specific interface and port, this information is also included in the map table in the interface_id foreign key reference and the port column. Local software, such as an operating system or local application, would have a NULL value in place of the interface and 0 as the port number.

5.2 Internal Object Models

Before the computation engine begins, all of the information is read from the databases into local C++ objects via a database manager that interfaces to ODBC. The C++ classes closely parallel the database table structure. The following classes are almost identical to their respective database tables, with additional accessor and mutator methods: Network, Host, Interface, Subnetwork and Software. The types tables are enfolded within their respective primary class. Also, the foreign key references within the table define "contains" relationship between the objects. For example, a Network object contains Host and Subnet objects, and a Host object contains Interface objects. The map tables define a pointer linking, such as that between a Subnetwork and an Interface.

All of the connectivity information is contained within the Connectivity data structure, which is is a matrix of boolean values between interfaces and listening interface-port combinations. The information is derived from the foreign key connections contained within

the database. This is accomplished by running a modified depth-first search of the network, once starting at each of the hosts in the network. Each of the firewalling gateways traversed along the way add their rules to the connectivity. Each run of the depth-first search adds a row to the matrix (`Interface` x `Interface:port`), and each rule along the path pertaining to the start host is also added. This technique limits the handling of multi-homed hosts, dynamic network connectivity, and re-routed network connections and would be an important future enhancement for NETSPA.

5.3 Computation Engine

The computation engine implements a rule-based AI scheme to link the possible states in the network and then and searches through them with a recursive, depth first search. The primary search algorithm for NETSPA is illustrated by pseudo-code in Figure 5-6. In short, the computation engine uses an `AttackState` object, described in the next section, to encapsulate state of the attack. The `compute_attack_graph()` method iterates through each of the hosts in the network and each of the actions. Within this double loop, NETSPA determines whether the existing state of the attack supports all of the requirements of the action. In addition, NETSPAdetermines whether the effects and modifies of the algorithm provide new useful state and warrant execution. If all of the conditions are met, a copy of the state is created and the action is performed on the copy. The grapher is then called to add the new state to the graph. The `addNode()` method, described further in section 5.3.1 returns whether the state created by performing the action is a new state (`true`) or whether the same state was already reached by another path (`false`). If the state is new to the graph, then the algorithm recurses, passing on the new state. This results in a depth-first traversal of the possible state space, while maintaining a dynamic state at each point in the process. Once all of the possible host/action combinations for a particular state have been exhausted, the algorithm updates the grapher with a `backup()` call and returns from this level of the recursion. In addition, a limit can be placed on the depth of the recursion to prevent the graph from becoming too large and taking excessively long to generate.

```
void compute_attack_graph(Network net, ActionList actions, AttackState state, Grapher graph)
{
  for(int i = 0; i < net.hosts.size(); i++)
    {
      for(int j = 0; j < actions.size(); j++)
        {
          // make sure that the action's requirements are met and
          // that the action provides novel state
          if((actions[j].requires == state) && (actions[j].effects != state))
            {
              AttackState newState = state.perform(actions[j]);
              // check to see if state has been seen before within the grapher
              if(graph.addNode(newState, actions[j]))
                {
                  // recurse
                  compute_attack_graph(net, actions, newState, graph);
                }
            }
        }
    }
  graph.backup();
}
```

Figure 5-6: The basic search algorithm used by NETSPA.

5.3.1 Attack State

The attack state object (`AttackState`) contains all of the changes to the privilege and network state, as well as the necessary mechanisms for determining whether an action should be executed. This includes all of the updates to the software on all of the hosts, the changes to the network connectivity, the current attacker host and privilege levels on each host, and several other pieces of state information.

The `AttackState` class contains several containers of internal objects that keep track of the state of the attack. The `Connectivity` object specifies new network connections enabled through the use of a `CAN_CONNECT` statement in the effects section of the action definition. `Runs` objects store any changes to software running on the hosts, including starting up or shutting down systems or modifying the state of existing software. Finally, the `HasAccess` objects store any new trust relationships added to the state. Additional state, such as the previous privilege levels on all the hosts is stored in several simple associative map variables within `AttackState`. Each state also has a unique ID number to enable the user to match the graph to a key of information.

The action statements are processed within the `AttackState` using regular expressions to match and return the useful information and attempt to match it to the local state. The primary `RUNS` statement within the requires section of the action defines the `<exposed_software>` and `<port>` and is necessarily matched first. The additional statement variables, `<current_host>`, `<target_host>`, and `<current_user>`, are already set by the computation engine and the existing state.

5.4 Graphing Subsystem

The `Grapher` class provides an interface for the computation engine to invoke a graphing method. `Grapher` contains five methods that all graphing systems must implement in order to comply with the interface. The `Grapher` is constructed with the starting state as the root node. After that, calls to `addNode(AttackState* node, Action* edge)`, insert the state as a node into the graph and join it to the graph with the passed action as an edge. This process connects the "current" node to the new node. The `addNode()` method returns a boolean value as to whether the added node produced a new node within the graph, or just a new linkage between the current node and an existing node of the same state, which would occur if the same state already existed within the graph. The current node is always the node that was last added to the graph, unless the `backup()` method is called, at which point the new current node becomes the latest visited parent of the existing current node. Once the computation engine has exhausted all of the possible paths, the `prune(GoalState g)` method can be called with a particular goal state, at which point the `Grapher` prunes all paths that do not end with the goal state. Also, `getStats()` returns a string representation of the statistics of the graph, such as all the privilege levels obtained at each of the leaf nodes, software modified on the leaf nodes, number of graph nodes and number of graph edges. Finally, once the entire graph has been generated and pruned, a call to `flush(ostream& out)` method writes the graph to the output stream defined by `out`.

Currently, NETSPA only implements a single grapher, the `GraphVizGrapher` that produces a file that can be read and translated by the open source graphing package graphviz [14]. `GraphVizGrapher` operates by keeping a node list and two separate node-node mappings: a child adjacency list and a parent adjacency list. Both of these mappings allow non-unique keys, making a full directed graph possible. Each time `addNode()` is called,

`GraphVizGrapher` first checks to see whether the same state already exists within the graph. If it does, a new parent-child mapping is added to the end of each of the map objects, using the existing current node as the parent, and the existing "same" node within the graph as a child, which is also then set to the current node. `False` is then returned to the computation engine. If the node being added is actually new, it is added to the set of nodes and the parent-child mappings are constructed between the current node and the new node. The current node then becomes the newly added node, and `GraphVizGrapher` returns `true`. Once `backup()` is called, the current node is set to the parent that has been most recently traversed, which is kept track of by a simple "most recent parent" vector.

The `prune()` method takes a `GoalState` as an argument and excises the graph based on this goal. The `GoalState` object contains information about the desired user, host, and visibility. For example, a goal state could be obtaining `root` on Host-A and being invisible to intrusion detection systems. The goal state pruning is done with a depth first traversal (DFS) of the attack graph, comparing the state at each node against the goal state. If a match is found on a leaf node, no further pruning needs to be done for this path, and the DFS continues. If a goal state match is found on a non-leaf node, all of the ancestors of the current matching node are removed from the graph. Finally, if an entire path is enumerated and no goal state is found, all of the single children predecessors along the path from the root node to the current non-matching leaf node are deleted.

The basic `GraphVizGrapher` pruning method is shown in Figure 5-7. A sample DFS traversal (without pruning) of the tree results in the nodes being examined in the order labeled. Node 3 matches the goal state (denoted in grey) as a leaf node, and so the DFS continues. Node 4 matches the goal state and has children, at which point the arcs $\{4, 5\}$ and $\{4, 6\}$ are deleted, as well as node 4. Node 6 contains a parent reachable by another path, and therefore remains for this pruning iteration beginning at node 4. As arc $\{4, 6\}$ was just deleted, the next nodes examined in the DFS are 7, 8, and then 6. Since node 6 is a non-matching leaf node, it is deleted recursively, along with each of its parents that have only a single child. In this case, only node 6 and arc $\{8, 6\}$ are deleted, as node 8 contains another child other than 6. Finally, node 9 is examined, determined to not match the goal state, and nodes 8 and 9, and all necessary arcs are deleted, resulting the the graph shown in (b).

The NETSPA graph generation scheme produces an incremental collapse of the attack

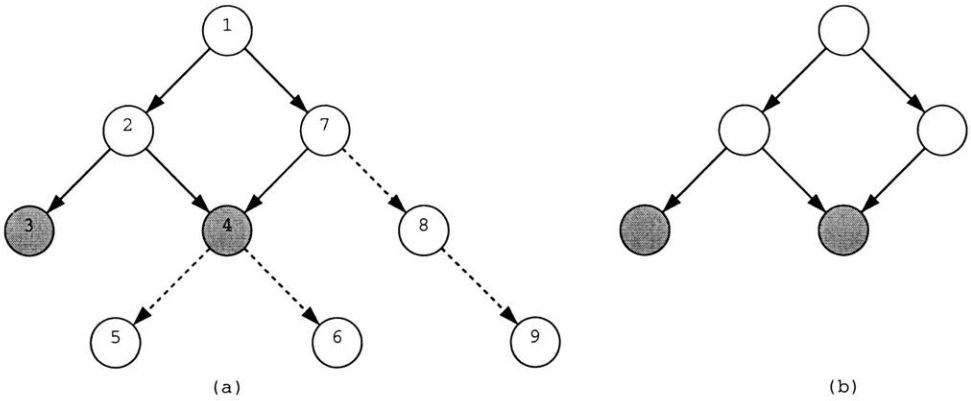

Figure 5-7: Illustration of graph pruning techniques used in NETSPA. The grey nodes denote the first matching goal state. (a) describes the full graph and (b) after the graph is pruned.

tree and allows NETSPA to save on computation time. This is primarily due to the fact that NETSPA does not need to explore the paths that touch existing states within the graph, as they have already been explored previously, due to the depth-first search nature of the computation engine. The pruning mechanism, however, is not incremental, and requires that the entire graph be generated and then pruned back to a goal state.

An alternate graph generation and pruning mechanism would rely on searching the graph in a breadth-first manner. This would enable the pruning to be done incrementally, while the collapsing of states would necessarily need to be accomplished after the entire tree in generated. We believe that entire attack graph provides valuable security information, and have thus optimized NETSPA for this goal.

Chapter 6

Running NetSPA

The following chapter describes the output of NETSPA against two separate networks and action sets. The first section, 6.1, describe the steps necessary to provide NETSPA with the information needed to run, such as the software and network configuration. Section 6.2 shows a sample run of NETSPA against a very small network, taken almost directly from [33]. This was used primarily to provide performance and correctness comparisons for NETSPA. It also makes it possible to demonstrate several features of NETSPA using relatively small, easy to view graphs. Section 6.3 describes runs of NETSPA against a much more realistic and complete small business network, complete with DMZ and realistic firewall rulesets. The design of this network is modified to show the effect of combining DMZ servers into a single host, changing security policies, and installing IDSs. In this network, NETSPA quickly discovered a security weakness in the system that the administrators had not previously considered.

6.1 NetSPA Setup

NETSPA requires several pieces of information before a run of the system. First of all, the action and software databases must be created and populated with all of the necessary actions and software that NETSPAuses to transition between attack states. Several scripts exists within the NETSPA package that provide the SQL (for MySQL) necessary to create the requisite database structures. As previously mentioned, the software information is imported from ICAT, and the action database is currently manually created with the appropriate attack models. To show that each action was correctly modeled, a test net-

work was created where each action correctly matched against only a single host. For later NETSPA users, these models could be distributed as a library of actions and associated software that could be easily imported into a local user's databases.

Once this is complete, the user must specify the network configuration within the network section of the database. NetViz provides a convenient and intuitive way to access this database through a data-driven graphics display. The user simply creates a test network within NetViz and fully specifies the software running on each host. The NetViz interface and sample network creation was previously shown in Figure 4-3.

Three additional pieces of information can be entered by the user to allow NETSPA to generate more specific attack graphs: firewall rules, trust relationships between various users and hosts, and a goal state that defines a critical host/privilege/visibility tuple. In addition, the recursive depth limit can also be set by the user. Currently, the trust relationships, goal state, and depth limit information is entered directly into NETSPA via a command line interface. If the inputs are not provided to NETSPA, the default behavior is dependent on several factors. The behavior of the firewall defaults to that specified in the host configuration table within NSD. If no trust relationships are specified, then the installation of a sniffer does not provide any information to the attacker. If a goal state is not given, NETSPA generates a complete graph of all of the possible actions against the network for a given depth. Finally, in the absence of a depth limit, NETSPA attempts to exhaust the attack tree. The firewall rules are currently entered directly into the network database, however adding this functionality to the NetViz interface in the future would be useful.

6.2 Simple Network

The simple network, shown in Figure 6-1, is useful in showing several basic properties of attack graphs generated by NETSPA as the resulting graphs are relatively small and tractable. The network contains only five separate hosts: an attacker host, a firewalling gateway, Host-A running an FTP server, rlogin service, and a local suid program, Host-B, running an FTP server and an SSH server, and an optional network-based intrusion detection system.

In addition to the network, the action database is populated with five actions, summa-

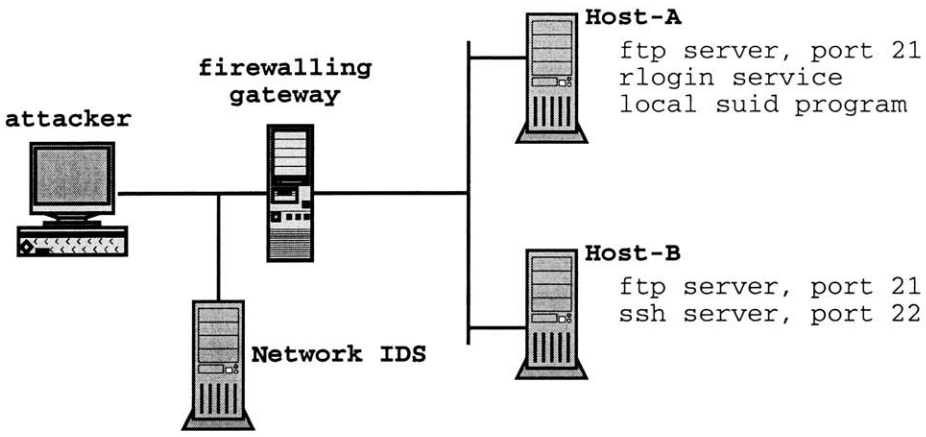

Figure 6-1: A simple network.

rized in Table 6.1. The remote login action is simply that, a login of a user from one host to another. There are two versions of the sshd overflow, one visible and one invisible to any network intrusion detection systems on the path from source to target host, in this case, the IDS host placed before the firewall. The action is a remote buffer overflow in the SSH server process. Since sshd runs with root privileges, this provides the attacker with root access on the target host. The FTP `.rhosts` action stems from a configuration vulnerability. If the FTP server allows public upload access, it is possible to create a `.rhosts` file in the home directory of the FTP user, thus establishing a trust relationship with a remote host. This enables a remote user to log into the target computer through the rsh service without authentication.

The full attack graph produced by a single run of NETSPAis presented in Figure 6-2. Each of the optional inputs, firewall rules, trust relationships, goal state, and depth, was left to their respective default values. Each node is labeled first with a unique ID number. This permits comparison to a description of all the states present in the graph and to later pruned graphs. As NETSPA is entirely deterministic, if the graph is regenerated with the same inputs, all of the state IDs will by identical. Each node also displays the current host (Att = Attacker Host, HA = Host-A, HB = Host-B) and privilege level of the attacker. For example, `S:144, HB root` means that the attacker has obtained `root` privilege levels on Host-B, which corresponds to state 144. The arcs model an action taken by the attacker, and are labeled with the action number and the target host. For example, 4(HA) means executing the FTP `.rhosts` action against Host-A.

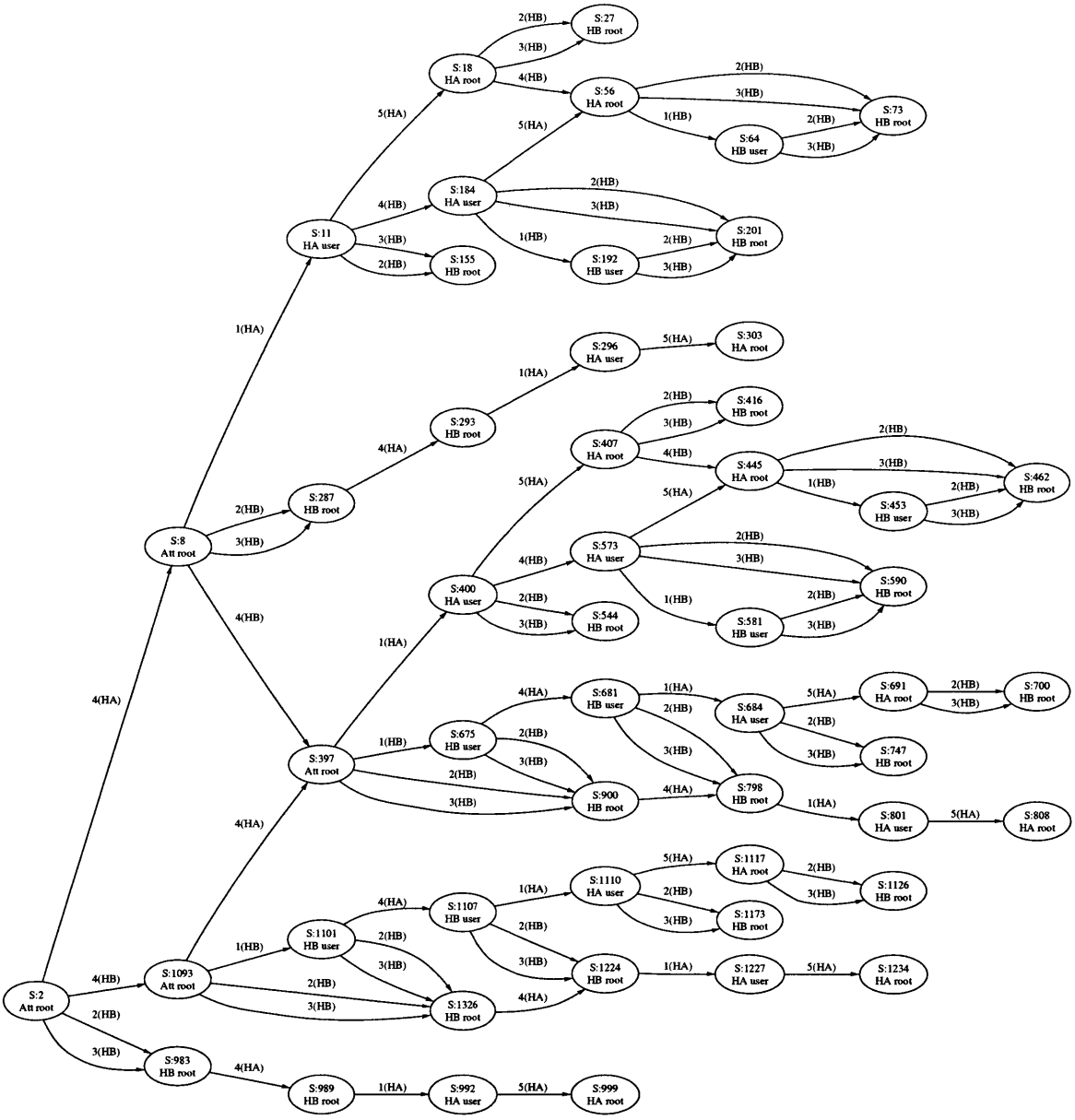

Figure 6-2: The full attack graph for the simple network.

	Action Name	Locality	Visibility	Requirements	Effects
1	remote login	remote	visible	a connection to the remote host and user-level or greater access	moves the attacker to the remote host with given privilege level
2	sshd overflow	remote	visible	a connection to a running SSH server	moves the attacker to the remote host with root privileges
3	sshd overflow	remote	invisible	a connection to a running SSH server	moves the attacker to the remote host with root privileges
4	FTP .rhosts	remote	invisible	a connection to a running FTP server	establishes a trust relationship between the current and target host
5	local overflow	local	visible	host is running vulnerable local suid program	provides root access on host

Table 6.1: List of possible actions against the simple network.

Each node corresponds to a unique state. NETSPA generates a key with the attack graph to enable an administrator to determine the full extent of the attacker state at each node in the graph. Each key entry describes a snapshot of the entire state of the attack up to that point. A sample entry in the key for a single node is shown in Figure 6-3. The key is fairly self explanatory, and simply lists all of the changes to the initial state of the network, including the current host and user level, whether the attack has been seen by an IDS, all of the past hosts and access levels of the attacker, established trust relationships, changes to the running software, changes to the connectivity, and user-defined trust relationships. In the previous example, the attacker is currently on Host-B with root access, and has previously been on Host-A with user access. The attacker has also established trust relationships between the attacker host and Host-A and Host-B. Finally, a previous action (sshd overflow) has caused the SSH server on Host-B to be put into the DOS state.

A specified goal state will prune the full graph to contain only paths leading to that state. The goal state is a definition of a non-allowable state for the attacker; a state that the security administrator is most interested in protecting against. This is illustrated in Figure 6-4, with the goal being undetected root access on Host-A. As there is no IDS, the visibility portion of the goal state does not matter. Each leaf node in the figure matches the goal state and all other extraneous nodes and arcs have been pruned away.

NETSPA can also use action visibility and placement of intrusion detection systems to

```
State ID:544
        Current Host: Host-B
Current User: root
        Already seen by IDS: false
        Primary Attack Host: attacker
        Past Users:
                attacker/root
                Host-B/root
                Host-A/user
        Has Accesses:
                From attacker/user  To Host-A/user
                From attacker/user  To Host-B/user
        Running Software:
                Host:Host-B Software:SSH server Port:22 DOS

        Existing Trust Relationships:
        Connectivity:
```

Figure 6-3: Sample state at a single node as output to the node key.

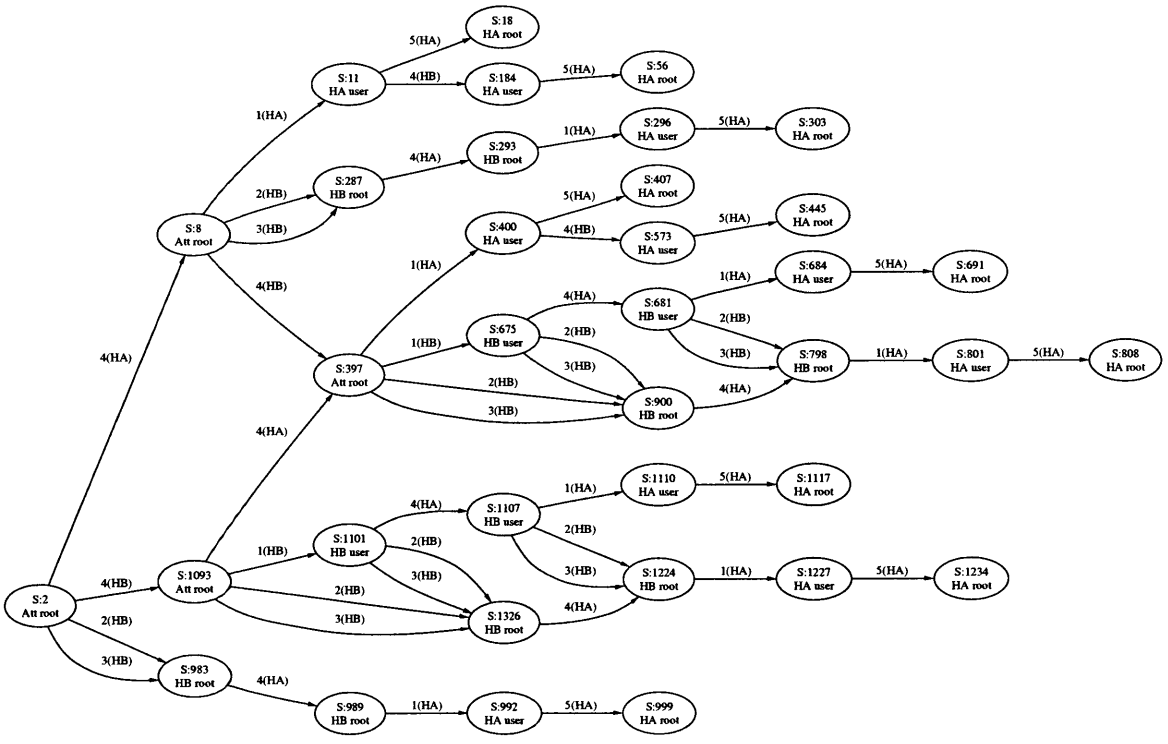

Figure 6-4: An attack graph for the simple network, pruned to the goal of root on Host-A.

alter attack graphs. For example, assume that the primary goal of the administrator is to prevent root access on Host-A, or, if root access is unavoidable, to be able to detect an unauthorized entry with an intrusion detection system (IDS). The administrator then wishes to evaluate the benefits incurred by either installing a network IDS between the firewall and the internal subnet, or placing a host based IDS on either Host-A or Host-B.

For comparison, the entire attack graph was again created with a single network IDS between the firewall and external subnet and is shown in Figure 6-5. The graph looks very similar to the full graph without the IDS, however, the states that have been currently or previously seen by the IDS are shown in grey. The visibility of the attack now modifies the state, resulting in a difference between how the two different `sshd` attacks are drawn.

Figure 6-6 shows the full attack graph from Figure 6-5 pruned to contain only paths that are not visible to the network IDS. This smaller graph enables us to determine several interesting properties about the possible routes to the goal state. For example, action 3, the stealthy sshd overflow, against the secondary host is required on every path from the start state to every leaf, whereas action 4, the FTP .rhosts configuration exploit, against the secondary host only appears as a predecessor of three of the four leaf nodes. This allows us to conclude that removing or upgrading the SSH server could prevent the goal state from being reached. Adding a host IDS to Host-B pruned the graph only slightly from the original, while adding a host IDS to Host-A resulted in no graph at all, meaning that the entire graph was pruned away because there was no way for the attacker to get to the goal state using the given actions, without being detected by the IDS.

The above results were performed on a network that was identical to that used in [33]. Attack graphs were compared to those presented in [33] and found to be fundamentally identical except for small differences in the way the graphs are constructed and displayed. These results confirm the accuracy of the graph generation algorithms, at least for this small network. In addition, the running times on such a small network were comparable, about 5 seconds in each case, showing that NETSPA is at least as fast as the model checker used in [33]. These results also show that a formal model checker is not required to generate attack graphs.

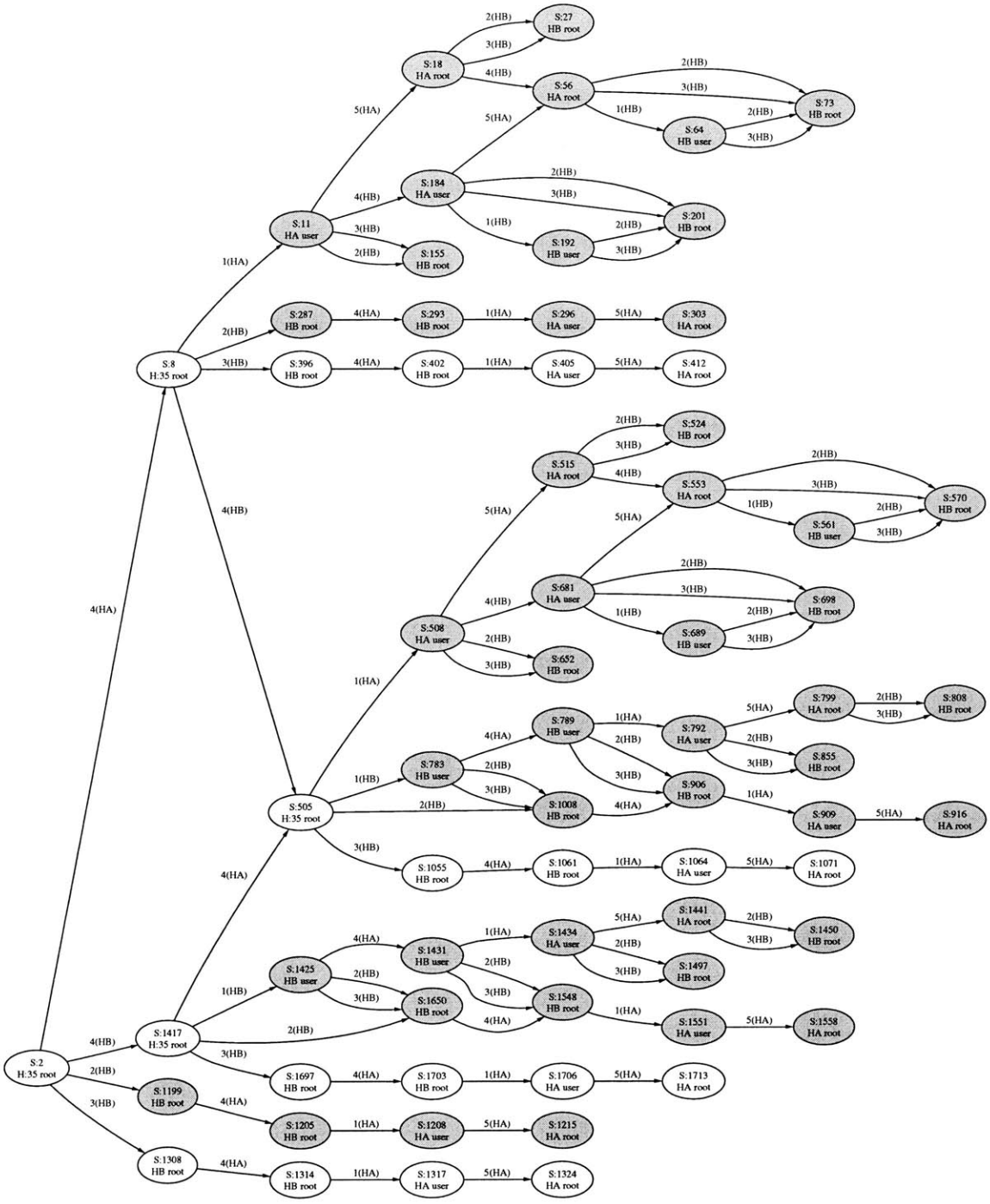

Figure 6-5: The full attack graph for the simple network with states in grey being visible to a network IDS.

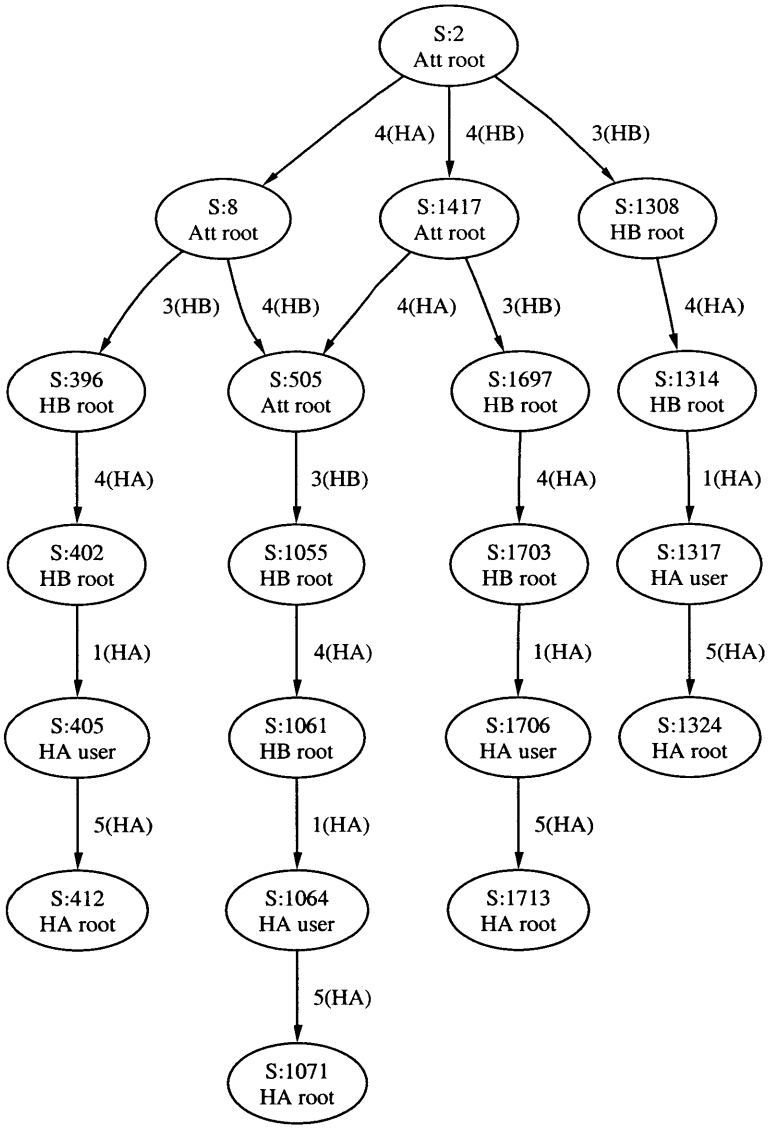

Figure 6-6: An attack graph for the simple network with a network IDS, pruned to the goal of achieving undetected root access on Host-A.

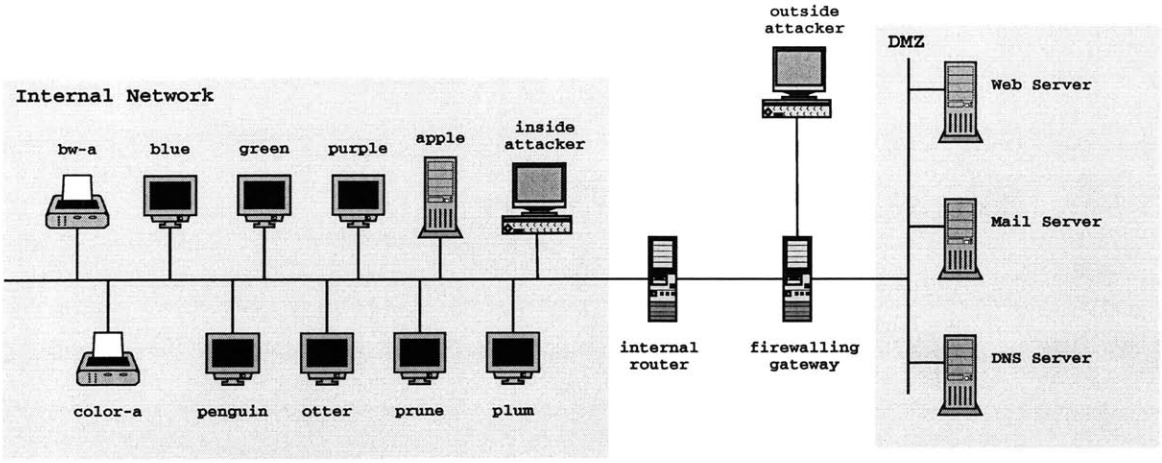

Figure 6-7: A sample realistic small organization network.

6.3 Realistic Network and Sample Execution

The realistic network was modeled directly from an actual small organization network. Nessus was used to scan a network, and a subset of the resulting hosts, host software, and vulnerabilities were used to create the network. Multiple hosts with similar vulnerabilities in the original network are represented as a single host within the network model. A complete list of the vulnerabilities modeled is provided in Appendix B. This realistic network makes it possible to show how NETSPA allows a security administrator to design a secure network in simulation before any hardware is purchased, software upgraded, or security policies changed. Even though the resulting attack graph was often too large to view easily, NETSPA still provided important security information.

The network is illustrated in Figure 6-7. It is comprised of primarily three sections. First, there is the outside network, where there exists only a single host, the outside attacker, akin to a host on the Internet. Next, there is a demilitarized zone (DMZ) where the publicly accessible servers are contained. Finally, there is the internal network where the organization's private users and computers reside. This is a standard method for dividing a network, as it places the high profile and high risk public servers in a separate section of the network. The theory is that the security administrators will focus on the security of the few public computers and not have to concern themselves with the security of every computer in the private network. In most situations, the number of DMZ servers is relatively small compared to the number of internal hosts, making this an attractive security solution.

Each computer's software configuration is outlined in Table 6.2. This shows that, while

the DMZ public servers will often have the latest software and patches installed, the computers within a private network are often much less secure. Also, the DMZ mail and DNS hosts are running a remote login service (`sshd`) to enable a user from within the private network to remotely administer the machines.

Dividing each of the three network sections is a firewalling gateway with several rules to prevent unauthorized access between the different parts of the network. These rules are summarized in Table 6.3. Basically, these rules allow an outside host to only connect to the computers in the DMZ on their respective public server ports (25 for SMTP, 53 for DNS, and 80 for HTTP) to access the organizations web page, send mail to someone within the organization, or lookup a DNS name hosted by the organization. The mail and DNS servers can connect to the internal server named "apple" to forward incoming mail and provide DNS lookup responses. Internal hosts can also connect to the DMZ web server on port 80. Finally, there is a single host name "otter" that can connect to the mail and DNS servers in the DMZ via SSH on port 22, allowing an internal user on "otter" to remotely administer the DMZ servers. It is also assumed that the DMZ web server is administered locally, and has no need of a remote administration service.

There are several questions that a network administrator might have about this network. For example, what can a novice attacker using existing vulnerabilities accomplish if starting from outside the network? What if the novice attacker was inside the network? What if the attacker had the capability to develop new attacks? Would the security of the network be enhanced by combining the three DMZ servers onto one machine? Where is the optimal placement of intrusion detection systems in order to catch both novice and expert attackers? What internal machines are most influential in the overall security of the network? Each of these questions is evaluated in the following sections.

6.3.1 Novice Computer Attacker

The novice computer attacker uses known vulnerabilities found on general security information repositories such as ICAT and Bugtraq. A list of these vulnerabilities and their associated REM definitions is given in Appendix B. The security administrator wishes to evaluate the proposed security of the current network against both insider and external novice attackers.

A run of NETSPA against the network starting at the outside attacker host yields a tree

Host Name	Interface	Software	Port
Outside Attacker	local	none necessary	
	attacker-host(6.6.6.6)		
Firewalling Gateway	local	none (hardware)	
	gateway-external(1.2.3.4)		
	gateway-internal(192.5.0.1)		
	gateway-dmz(10.0.0.1)		
Web Server	local	Microsoft Windows 2000	
	webserver(10.0.0.11)	Microsoft IIS 5.0 HTTP	80
Mail Server	local	RedHat Linux 7.3	
	mailserver(10.0.0.12)	OpenSSH 3.2.2	22
		Sendmail 8.12.3	25
DNS Server	local	FreeBSD 4.6	
	dnsserver(10.0.0.13)	OpenSSH 3.2.2	22
		BIND 9.2.1	53
Internal Router	local	none (hardware)	
	router-external(192.5.0.1)		
	router-internal(192.5.135.1)		
Inside Attacker	local	none necessary	
	inside-attacker(192.5.135.66)		
Penguin	local	RedHat Linux 5.2	
	penguin(192.5.135.101)	Wu-ftpd 2.6.0	21
		generic telnetd	23
		BIND 8.1.2	53
		generic rlogin	514
		generic rpc.nfsd	2049
		MySQL 3.20	3306
Otter	local	RedHat Linux 7.2	
	otter(192.5.135.102)	OpenSSH 2.9	22
		generic rpc.statd	32768
Blue	local	Microsoft Windows 2000 Advance Server	
	blue(192.5.135.111)	generic telnetd	23
		Microsoft ESMTP 5.0	25
		generic smb	139
		generic snmp	161
		generic rpc.nfsd	2049
Green	local	Microsoft Windows ME	
	green(192.5.135.112)	Microsoft IIS 5.0 FTP	21
		Microsoft IIS 5.0 HTTP	80
		generic smb	139
Purple	local	Microsoft Windows NT 4.0	
	purple(192.5.135.113)	generic X11 server	6000
		generic smb	139
Apple	local	Sun Solaris 2.7	
		CDE 2.2	
	apple(192.5.135.121)	Wu-ftpd 2.6.0	
		SSH 1.2.27	22
		generic telnetd	23
		Sendmail 8.10.2	25
		BIND 8.1.2	53
		Apache 1.3.9	80
		PHP 3.0.18	80
		Qpopper 2.53	110
		generic snmp	161
		generic rlogin	514
		MySQL 3.22	3306
		generic dtspcd	6112
Prune	local	Sun Solaris 2.6	
	prune(192.5.135.122)	Sendmail 8.9.3	25
		BIND 8.2.2 p3	53
Plum	local	Sun Solaris 2.6	
		CDE 1.2	
	plum(192.5.135.123)	SunOS FTP Server 5.6	21
		Sendmail 8.8.8	25
		ntpd 4.0.99k	123
		generic rlogin	514
		MySQL 3.22	3306
		generic dtspcd	6113
BW-A	local	HP Laserjet	
	bw-a(192.5.135.201)	HP JetDirect	161
Color-A	local	Tektronix Phaser 840	
	color-a(192.5.135.202)	Micro-HTTP 1.0	80

Table 6.2: The host software configuration for a small, realistic network.

From	To	Port	Rule
all	all	all	deny
outside	internal network	all	deny
outside	DMZ Web Server	80	allow
outside	DMZ Mail Server	25	allow
outside	DMZ DNS Server	53	allow
DMZ Mail Server	apple	25	allow
DMZ DNS Server	apple	53	allow
inside	DMZ Web Server	80	allow
apple	DMZ Mail Server	25	allow
apple	DMZ DNS Server	53	allow
otter	DMZ Mail Server	22	allow
otter	DMZ DNS Server	22	allow

Table 6.3: A set of realistic firewall rules.

with only a single node, meaning that the outside attacker could not progress beyond the start host. This is intuitively obvious, as the only possible points of access to the internal network lie through the DMZ hosts, which are heavily patched and updated so that no known security holes exist.

A novice attacker starting from within the network, however, is entirely different. The internal network is often subject to numerous existing vulnerabilities due to relaxed security rules and trust in the firewall. An internal attacker can often wreak havoc within a network. In this case, the attacker begins from the inside attacker host. Because the theoretical branching factor for internal nodes is so high, the test run is limited to only three levels deep, corresponding to the attacker executing three actions. Even so, the resulting graph is virtually incomprehensible, as is shown in Figure 6-8. With only attack paths of length three, the attacker has obtained root-level access on eight of the ten internal hosts. In addition, the attacker was able to crash or DOS six pieces of software on four internal hosts. Finally, the inside attacker was able to obtain `root` access on both the mail and DNS DMZ servers. To further analyze this DMZ breach, it is possible to prune the graph to only those attacks that reach the mail server. There is a path that, within three actions, achieves root access on the mail server. In addition, if NETSPA is run and pruned to each of the other DMZ servers, the resulting graphs can be manually recombined into Figure 6-9, which details the inside-to-DMZ attack paths. The attacker first breaks into "otter", which is running a vulnerable version of `rpc.statd`. "Otter", as mentioned earlier, is the

machine from which the administrator is allowed to remotely log into both the mail and DNS server via SSH. The attacker then installs and runs a sniffer to obtain remote login information, which includes the fact that an administrator on otter has administrator access on "dnsserver" and "mailserver", servers in the DMZ. The attacker then logs into the remote machines as an administrator using SSH.

This DMZ attack path quickly shows the most vulnerable routes to the DMZ. In this case, there are only 2 paths and they both traverse the same starting nodes. A security administrator can quickly improve the security of the DMZ against internal novice attackers by disallowing the SSH connection between otter and the DMZ servers. This solution, however, might not be agreeable with the organization's requirements to be able to administer the DMZ hosts from within the internal network. A security planner can then look further up the attack graph and find that by improving the security of "otter" by upgrading or turning off the rpc.statd service, the hosts in the DMZ are more protected from internal attacks.

In this example, a "sniffing" action on "otter" represents either capturing network traffic or monitoring key-strokes to determine that the administrator on "otter" can connect as administrator via SSH to the mail and DNS servers in the DMZ. As previously stated, NETSPA attempts to create worst-case attack graphs, which would allow this encrypted information to be sniffed by a user who achieved `root` privileges.

6.3.2 Expert Computer Attacker

Whereas a novice computer user is limited to existing known vulnerabilities, an expert attacker can often craft new exploits. In this case, assume that an advanced user can create a new exploit for any of the hosts in the DMZ. Given past performance, both IIS and BIND tend to be high-risk software packages.

If IIS contains a new remote `root` vulnerability that an expert attacker can exploit from an outside attacker host that is not known to the general security community, NETSPA generates a very small graph with only two actions: first exploit the web server using the new exploit, then install a sniffer on the host. Since the attacker cannot then break into other machines on the DMZ and the web server cannot connect into the private network, the attacker can progress no further.

A new BIND vulnerability, on the other hand, provides more interesting results, as

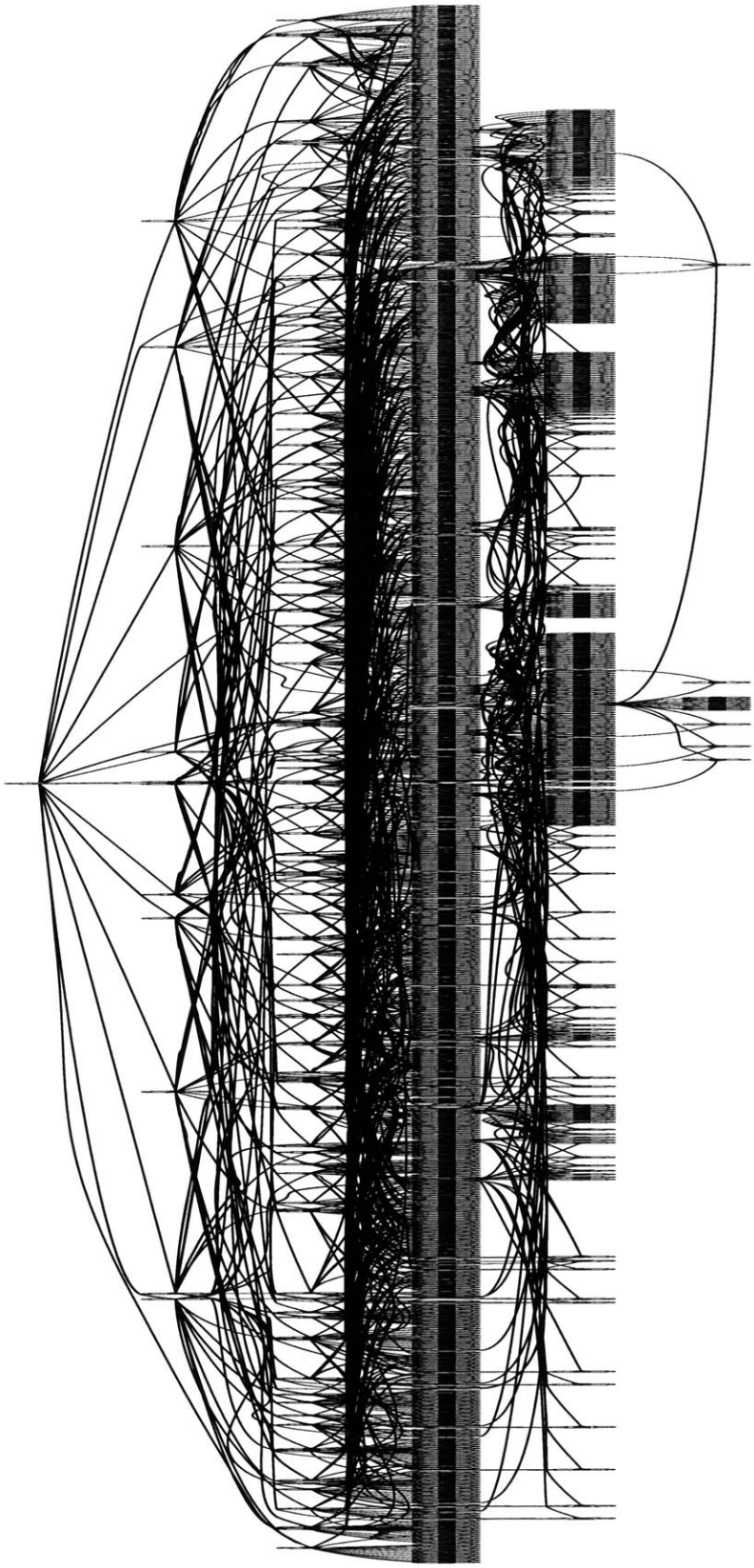

Figure 6-8: Three level attack graph of an insider attack.

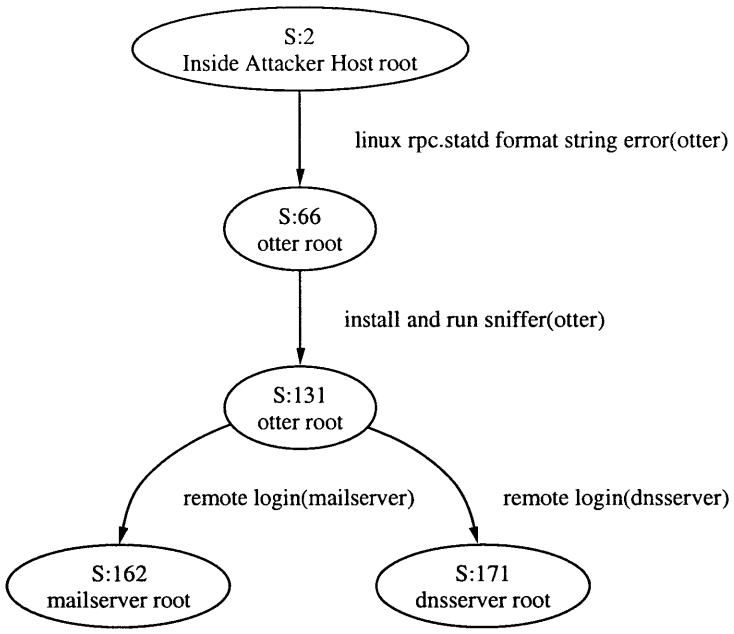

Figure 6-9: Attack graph of the actions that reached the DMZ hosts starting from within the internal network.

shown in Figure 6-10. In this case, the attacker leverages the new BIND exploit against the DNS server. As the DNS server is allowed to connect into the internal network to respond to DNS queries, the attacker is able to use another, existing BIND exploit against host "apple," at which point the attacker is within the internal network and the graph explodes, similar to that explored earlier with the novice internal attacker. This illustrates the danger in relying on firewall rules alone as a security metric; if the essential machines within the internal network are not secure, the entire network can be compromised. System administrators on the real network that was used to create this model were not aware of this problem until it was discovered by NETSPA.

6.3.3 Combining DMZ Server Hosts

A key feature of NETSPA is the ability to design a secure network before the hardware actually exists. In this example, a NETSPA user is attempting to decide between two alternatives: creating a single DMZ server that performs the DNS, SMTP, and HTTP functions, or creating separate servers for each service. This could allow a security administrator to convincingly argue for the funding for more hardware if it can be shown that a multiple source solution is more secure than a single point one.

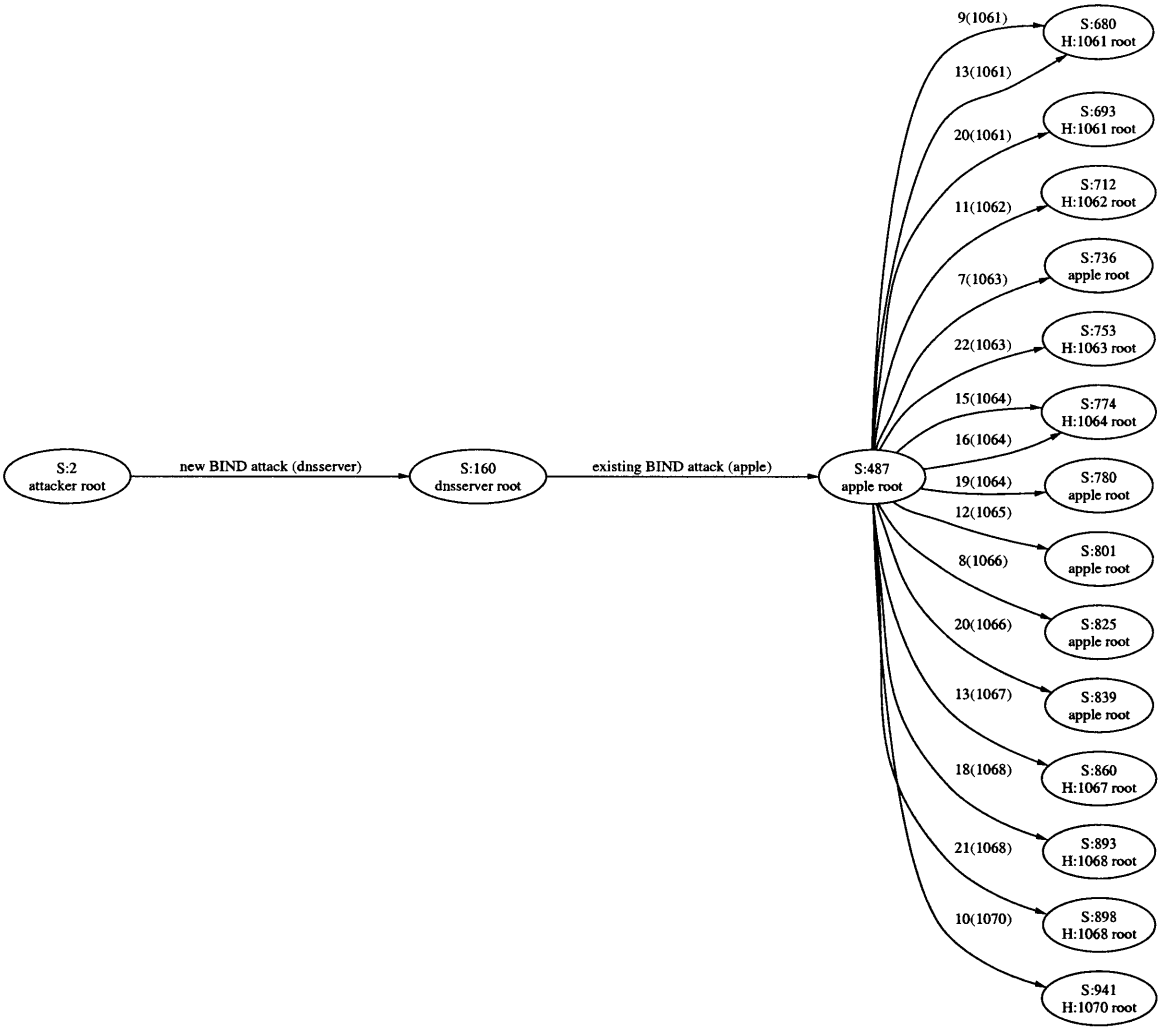

Figure 6-10: Attack graph of an expert outside attacker with a new BIND exploit.

We have already examined the security of the separate hosts, now assume that we combine all of the respective services into a single host.

Another interesting use of attack trees arises from the comparison between the graph derived from original network of Figure 6-7 and that resulting from running NETSPA against a similar network where all of the DMZ servers have been collapsed into a single host. In the case of this "combined-server", a novice attacker can still not attack the DMZ host nor the internal network from an outside starting position because all of the patches are up to date.

A skilled attacker, on the other hand, has several additional means to attack the internal network. Previously, each server resided on a separate, contained machine and we demonstrated that a vulnerability in IIS only compromised the DMZ web server host. If all of the services are running on a single machine, however, a compromise in any of the servers allows the attacker access to the other services. Since the firewall rules must now be set up to allow the single DMZ server to pass SMTP and DNS traffic to the internal network, the attacker now has a path to the internal network after exploiting IIS via ports 53 and 25, which did not exist when the services were hosted on different physical computers. The graphs for a new IIS vulnerability, a new BIND vulnerability, and a new Sendmail vulnerability were combined, as before, into a single graph to illustrate the fact that initial attacker now has three initial points of entry into the internal network via the new IIS, BIND or Sendmail vulnerabilities, where there was previously only one through the vulnerable BIND server.

6.3.4 IDS Placement To Detect External Attacks

An administrator would like to be able to defend the internal network against all attacks from an outside host. Novel vulnerabilities and exploits against the servers in the DMZ, coupled with the existence of valid SMTP and DNS connections between the DMZ and the internal network, however, make this task quite difficult, as demonstrated in the previous sections. Informed placement of IDSs might enable a system administrator to detect new exploits.

In this example, a system administrator wishes to enhance the security of the realistic network by placing either a host-based IDS on "apple" or a network-based IDS on the link between the firewall and internal gateway to detect external attacks. As we have previously shown, a novice external attacker using only existing vulnerabilities cannot even begin to

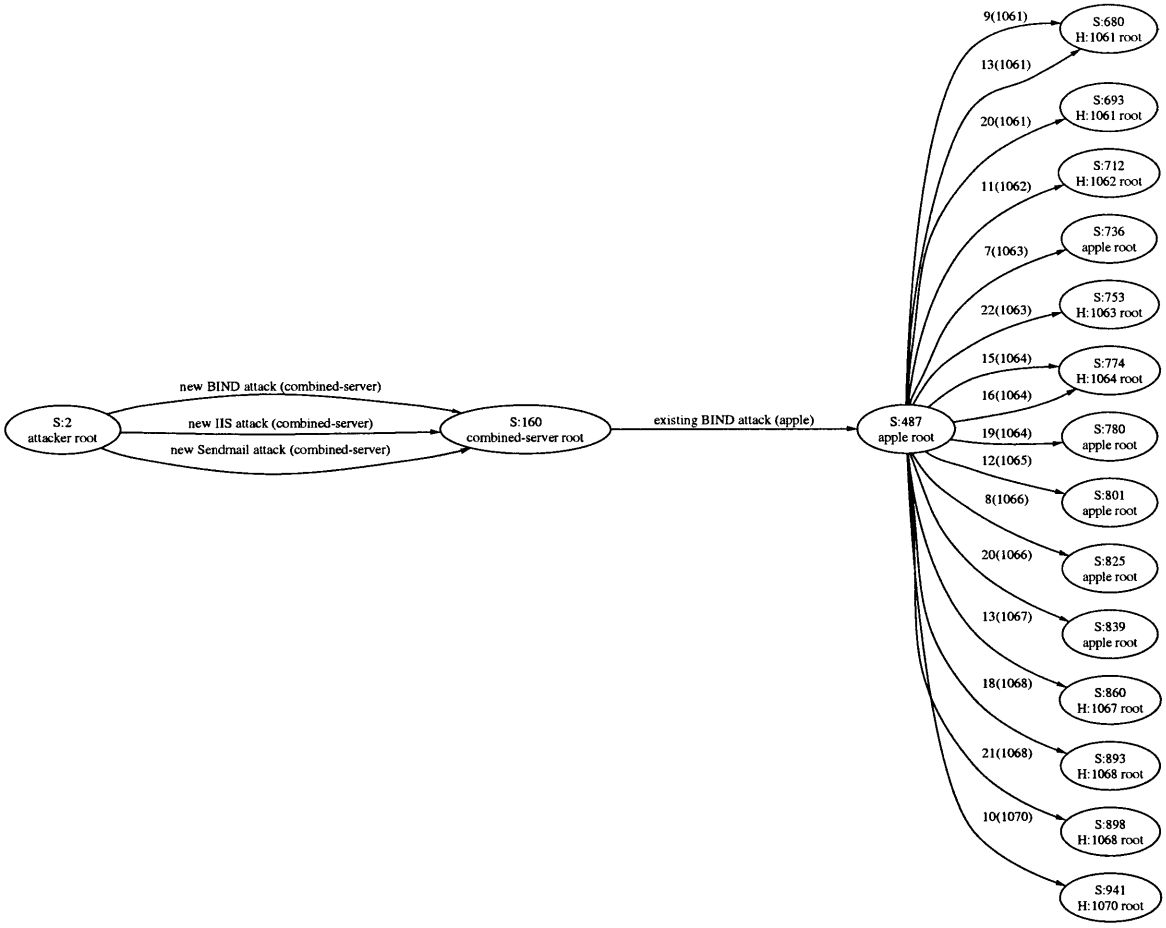

Figure 6-11: Attack graph resulting from a new IIS, BIND and Sendmail vulnerability against a combined server host.

launch an attack against the recent and heavily patched servers in the DMZ. An expert attacker with access to new vulnerabilities is an entirely different story. For this case, we again assume, that the attacker has developed a novel BIND vulnerability. As most network-based IDSs operate on a signature mechanism, this novel attack will be invisible to a network IDS. A host-based system, on the other hand, is specification based and can detect new attacks such as this one, as it causes the server process to operate outside of specified software limits.

The attack graphs for each of the IDS placement possibilities are the same and are shown in Figure 6-12. The attacker first exploits the new BIND vulnerability on the DMZ DNS server and then exploits the internal server, "apple," with a known BIND vulnerability. At this point, both the host- and network-based IDSs detect the second BIND exploit because a signature has already been developed for the network-based IDS.

At first glance, a simple solution to protect the internal network is to upgrade the internal server. The attack graph for an internal server running BIND of the same version as the DMZ DNS server (the latest available, 9.2.1) and the host-based IDS is shown in Figure 6-13. This figure is very similar to Figure 6-12, except that the second action executed by the attacker is the new BIND attack. The host-based IDS can thus detect the new attack being run against the internal server. The network-based IDS alternative, however, provides an interesting result, shown in Figure 6-14. After the upgrade of the DNS server on "apple," the network-based IDS now *fails* to detect the incoming intruder, as the attacker is using a new exploit without an existing IDS signature.

The results from this experiment provide several possible IDS placement and type alternatives. First of all, a host-based IDS on "apple" can detect both existing and novel attacks entering the network, even if the software has been upgraded. The network-based IDS, however, can only detect the attacker entering the network using an existing vulnerability for which a signature exists. The trade-off implicit in running the two different types of IDSs is thus obvious: upgrade the software and run a host-based IDS, or do *not* upgrade the internal DNS server, and run a network-based IDS. This result is non-intuitive, represents an unusual instance of obtaining security by diversity.

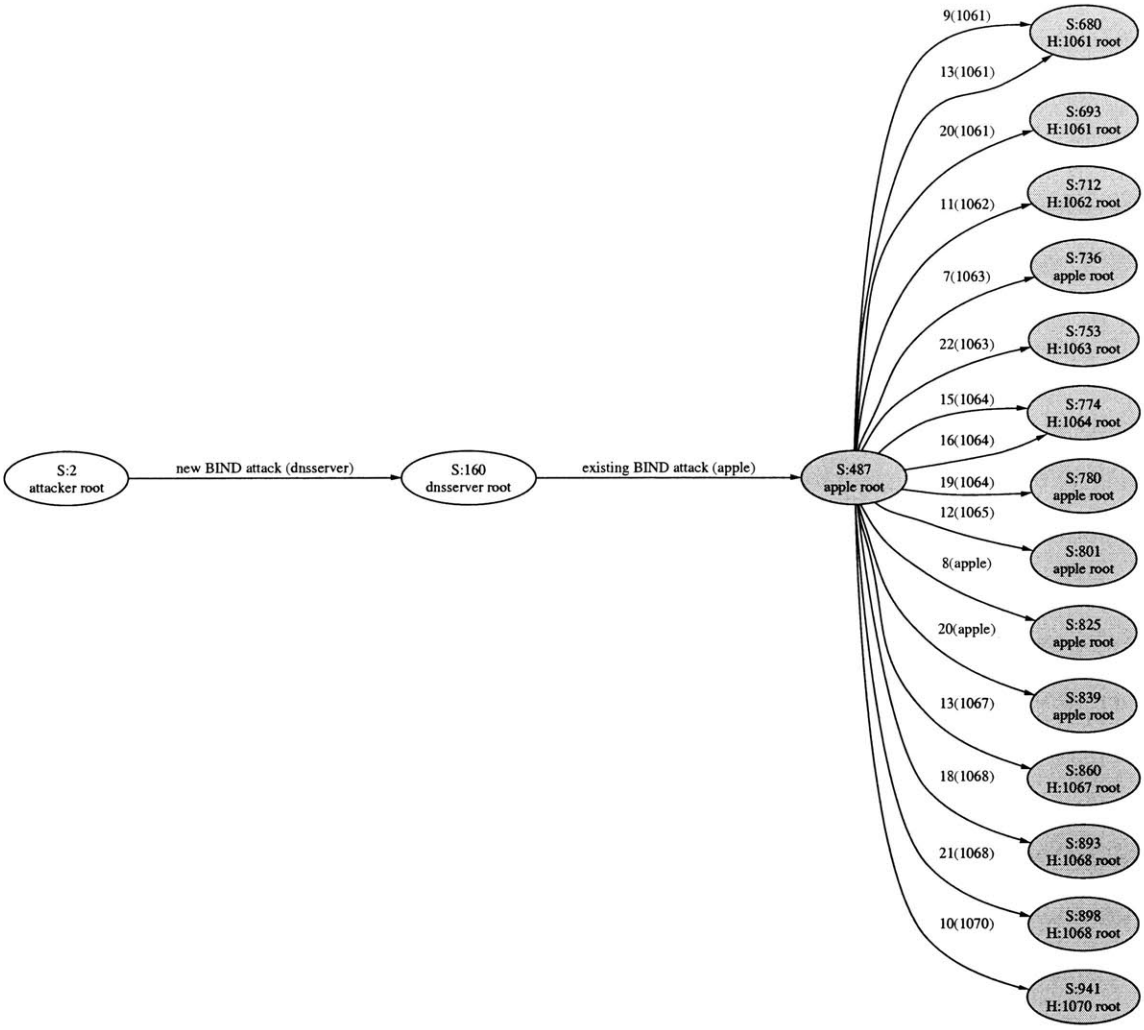

Figure 6-12: The attack graph resulting from an external attacker with a novel BIND vulnerability, an older internal DNS server, and either a host- or network-based IDS.

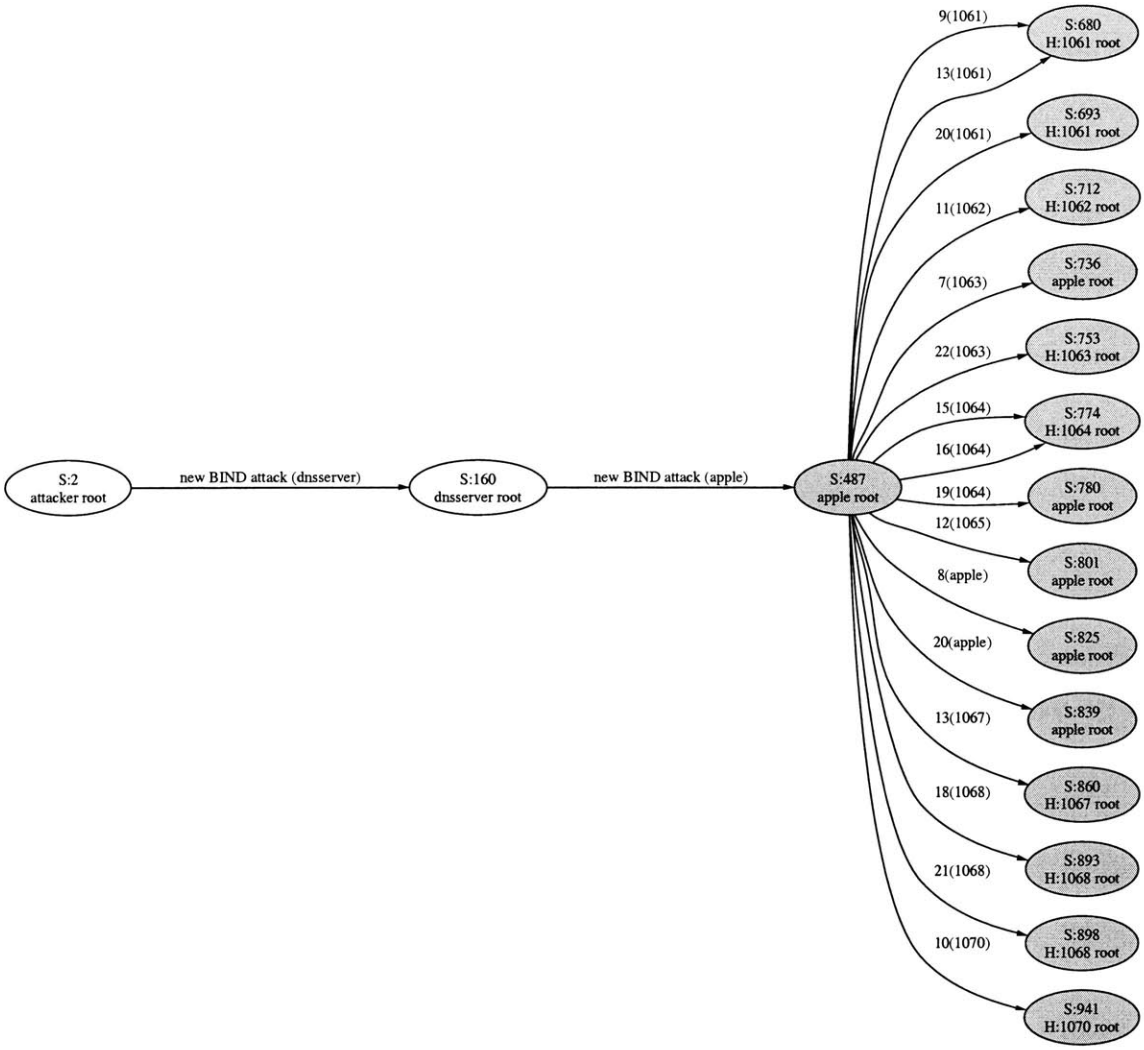

Figure 6-13: The attack graph resulting from an external attacker with a novel BIND vulnerability, an upgraded internal DNS server, and a host-based IDS on "apple."

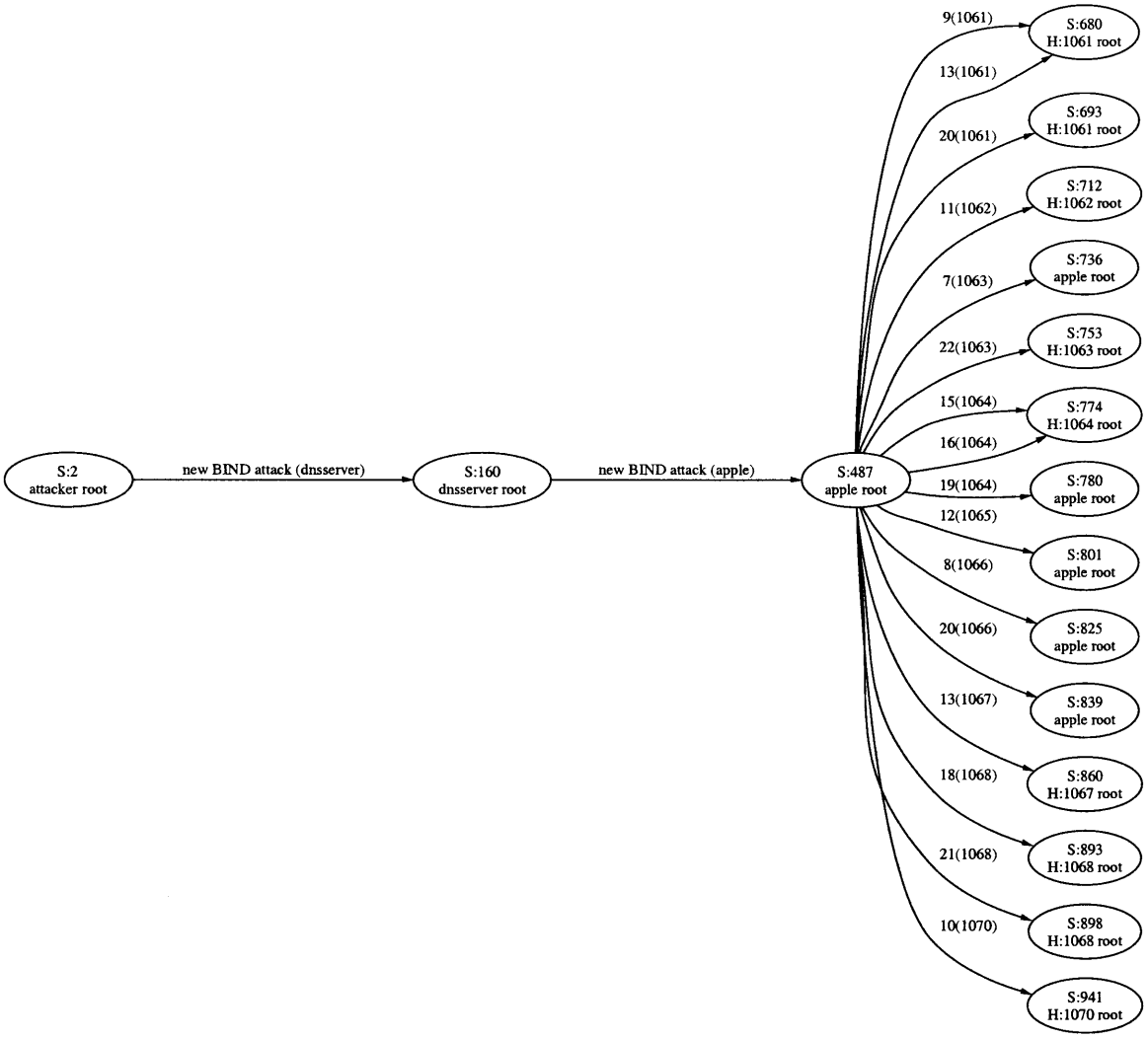

Figure 6-14: The attack graph resulting from an external attacker with a novel BIND vulnerability, an upgraded internal DNS server, and a network-based IDS.

6.3.5 Critical Internal Hosts

It has been estimated that insider attacks account for almost half of all the attacks directed against organizations [23]. Therefore the trust placed in firewalls, IDSs, and other security systems that focus on external hosts is partially misplaced. Determining critical internal hosts is important in analyzing a network's vulnerability to an insider attacker. To do this, we first logically divide the internal hosts into 3 groups: internal servers, administrator machines, and user machines. There is only one internal server, "apple," which handles all of the DNS and mail queries for the internal network. An administrator machine is a host from which an administrator performs standard, root-level system administration tasks, such as adding users or installing and upgrading software. To model this, we create a set of trust relationship rules from the administrator's computer to each of the hosts within an administration category: animal, color, or fruit. The administrator machines are "otter," "blue," and "prune." Thus `root` on "otter" can administer "penguin." In addition, we create another set of trust relationships that enable a global administrator on "otter" who has `root` level access on *all* of the hosts in the network, including the internal and DMZ servers. The allowable login rules are summarized in Table 6.4. In addition to the host grouping, assume that all of the hosts on the internal network are invulnerable to any existing actions. Finally, to simplify the discussion, we are assuming that all of the hosts run some sort of remote login service.

Next, we enable single vulnerabilities against each type of host to determine the results of an insider attack using only a single vulnerability on a single host. Enabling this single vulnerability results in different attack graphs begin generated depending on the type of host compromised. Each host type is discussed in the following sections.

User Host

Enabling a single vulnerability on a user host (i.e. "penguin," "green," "purple," or "plum) results in an attack graph similar to the one shown in Figure 6-15. The attacker can only issue the single attack against the only vulnerable host, in this case a remote `root` overflow versus `rpc.nfsd` against host "penguin." Once on the host, the attacker can install a sniffer. Since the user on this host is not an administrator, no allowable logins exist emanating from this host and no additional vulnerabilities exist in the network, and thus the attacker can

From User	From Host	To User	To Host
root	blue	root	green
root	blue	root	purple
root	prune	root	apple
root	prune	root	plum
root	otter	root	penguin
root	otter	root	blue
root	otter	root	green
root	otter	root	purple
root	otter	root	apple
root	otter	root	prune
root	otter	root	plum
root	otter	root	bw-a
root	otter	root	color-a
root	otter	root	webserver
root	otter	root	mailserver
root	otter	root	dnsserver

Table 6.4: A set of administration login rules.

go no further.

Server Host

An insider that can exploit a vulnerability on a server host results in an attack graph that is identical to that from when the insider exploited a simple user host. The practical difference, however, is that the insider now has access to the services and data provided by the internal server, as well as possible access to the DMZ. Since all vulnerabilities but one were assumed to be fixed for this example, the attacker can not actually reach the DMZ servers, however the ability to pass through the firewall through the SMTP and DNS ports from "apple" to "mailserver" and "dnsserver," respectively could allow additional novel attacks.

Administrator Host

The final case, a new vulnerability on an administrator's machine, is the one that provides the most interesting graph behavior. We have two different classes of administrator: local admin, such as from hosts "blue" and "prune," and global admin, such as from "otter."

Figure 6-16 displays the graph for an insider exploiting a vulnerability in a local admin host, more specifically host "blue." This results in the attacker being able to quickly

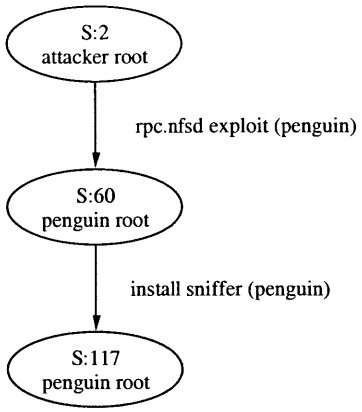

Figure 6-15: Attack graph resulting from a single vulnerability on an internal user host.

achieve `root` level access on all of the hosts within blue's administration group: "purple" and "green." This is accomplished by the attacker first exploiting the "new" SNMP vulnerability, sniffing the network or host to obtain necessary connection and trust relationship information, and then simply logging into the other user hosts with administrator (`root`) privileges.

A new vulnerability in a global administrator host is much more damaging to the entire network. As the global administrator has access to *all* of the hosts on the network, the attacker can exploit the vulnerability, and sniff, at which point the attacker then has `root`-level access on all of the hosts in the network. This is further illustrated by the graph in Figure 6-17. After only 3 actions, the attacker has access to all of the hosts except for the webserver, which is only protected because of the blocking firewall rules.

6.4 Running Evaluation

NETSPA is practical to run in both space and time requirements, however there are some feasibility constraints when fully computing large graphs. A complete graph of the simple network can be generated in roughly 5 seconds on a Pentium III 500 running RedHat Linux 7.1 and using MySQL 4.0 alpha and the `boost` regular expression libraries. The entire graph for the realistic network could not be feasibly computed once an attacker had access to the internal network because it was too large to either compute or view. When within the internal network, the branching factor of the graph was roughly equal to the number of modeled exploit actions, or about 17. Reaching a recursive depth of 3 required only roughly

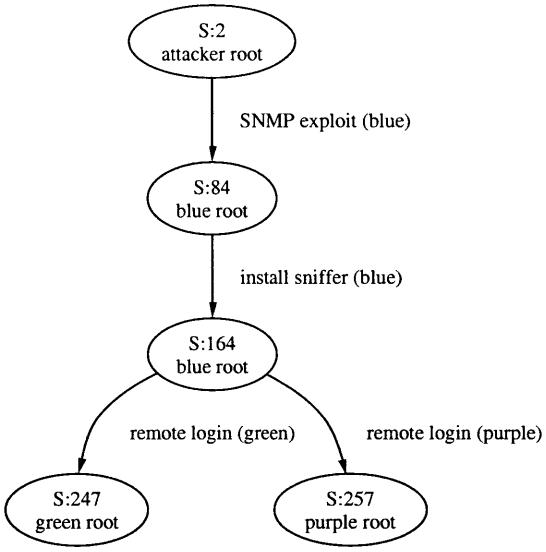

Figure 6-16: Attack graph resulting from a single vulnerability on a host able to administer two other machines in the internal network.

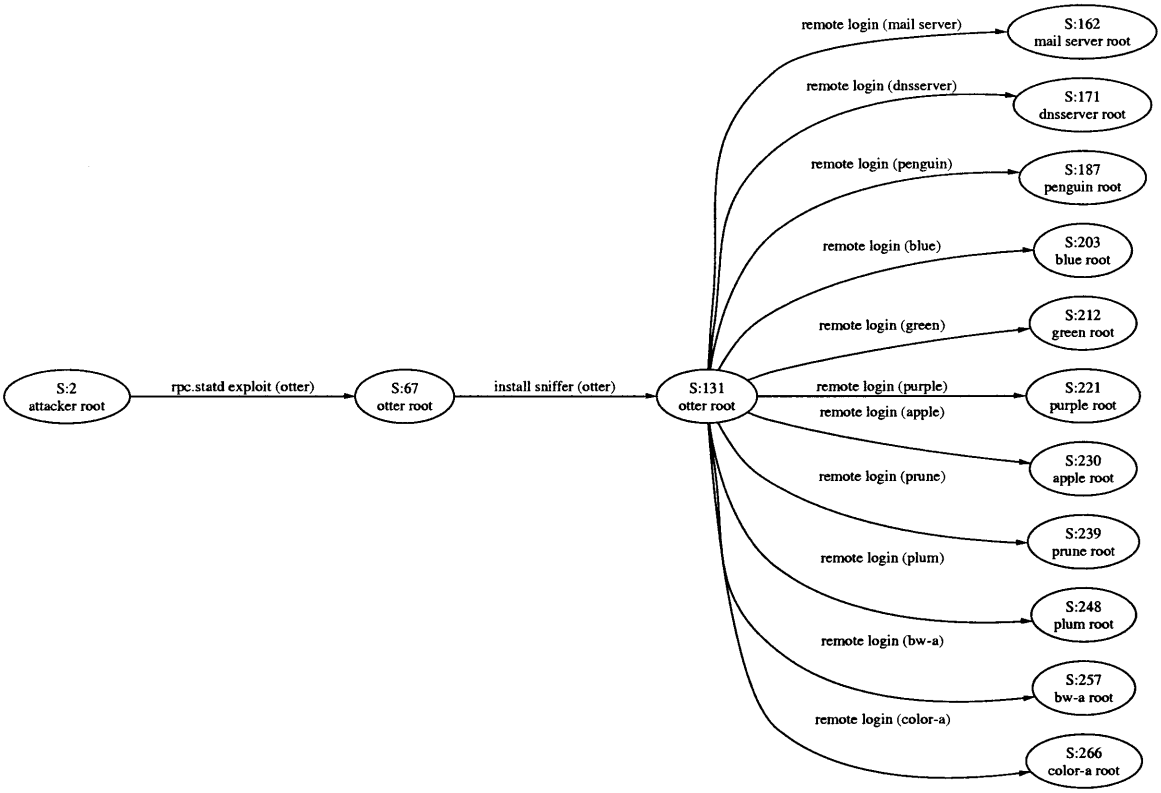

Figure 6-17: Attack graph resulting from a single vulnerability on a host able to administer all of the organization's machines in the network.

1.5 minutes, and the complete generated graph could not be visualized. Statistics generated on the graph, however, demonstrated that every internal host but one was compromised.

NETSPA was profiled using `gprof`, and it was found that the regular expression libraries and associated string operations accounted for more than 70% of the entire running time. This is intuitively obvious, as NETSPA relies heavily on the regular expression libraries and attempts to match at least one REM clause of every action against every host at each level in the recursion. In addition, NETSPAmakes extensive use of the standard C++ container classes to simplify program design and readability, which also appeared high on the list of most-time required functions in the profiler. We are looking into alternative ways to speed up the matching and determine whether the C++ containers are essential to the design of the system.

Chapter 7

Future Work

NETSPA provides a basic architecture upon which many other modules and functions can be built. NETSPA has been shown to work on relatively large, realistic networks with a substantial number of vulnerabilities. NETSPA would derive the most benefit from a larger database of REM defined actions and more test networks to further exercise the system. In addition, a slightly more extensive REM syntax and speed enhancements would provide NETSPA with additional functionality and useability.

The biggest factor contributing to the useability of NETSPA is the action database. Our current database was constructed to illustrate certain concepts on our sample networks, and, as such, only contains 18 specific exploit vulnerabilities and 3 general ones, such as a brute force password crack and sniffer installation. Extending this database would allow for more networks to be modeled and correctly analyzed. The database should be extended by finding vulnerabilities on new modeled networks and adding exploits only for these vulnerabilities. A larger set of out-of-date actions is not necessary.

NETSPA could benefit from additional realistic network testing to exercise the system and discover any additional limitations. One such known limitation is the inability to correctly handle multi-homed hosts, as only a single path is constructed between any two hosts before NETSPA tries to match any actions. Similarly, NETSPA does not currently have the functionality to modify firewall rulesets or routing tables through the course of an attack. Modeling large, realistic networks would discover these limitations and allow for them to be fixed.

The REM language for NETSPA could be enhanced to include slightly more specific

syntax, such as additional privilege levels, to enable an action to be more exactly specified. As it stands, several types of actions need to be relaxed in order for them to be defined in REM, such as the ability to read a file from a remote host or log in as a limited privilege user. An attack graph generated with a richer REM specification will have fewer unnecessary path components.

Graph generation speed also contributes to the useability of NETSPA, however the current implementation is quite efficient. As every action in the database must be attempted against every host in the network at each level in the recursion, limiting the number of actions used by the computation engine would result in a significant speed increase. Several techniques could be implemented within NETSPA to prune the network models and action database.

Hosts within a network running the same basic operating system and software can be realistically combined into a single representative host. This technique limits the number of hosts within a network to the number of distinctly different hosts. At the moment, this must be done manually by the NETSPA user who creates the network, however an automatic procedure would be a useful enhancement. The action database can be pruned before the matching loop to contain only those actions that are possible to execute against the hosts in the specific network. For example, if all of the hosts on a network run a Microsoft Windows operating system, all actions that require other operating systems could be removed from the list, resulting in decreased running times. Once this action pruning is done, we estimate that the number of actions that can possibly be executed against a network is on the order of tens, and not hundreds. This reinforces our position that the "realistic" network accurately models an actual network and that NETSPA provides a scalable graph generation mechanism.

Finally, NETSPA would benefit greatly from additional input and output modules. Currently, only NetViz exists to import information into a network model, and the `GraphVizGrapher` is the only output graphing mechanism. Creating additional ways to lessen time spent by a user modeling a network and increasing the utility of the output graphs would benefit NETSPA and its users. Because of the database nature of the network model, and the standard interface for the `Grapher`, these additional modules should not be difficult to implement.

Chapter 8

Conclusions

Existing security software and security information resources are necessary to analyze the security of a network. They all, however, fail to take into account several networking fundamentals, such as the structure of a network and the existing security policy. Current software is thus limited to single-point vulnerabilities, and overlooks an entire class of system security that involves the interplay between several different single-point vulnerabilities, network topology, intrusion detection systems, and other vulnerability protection mechanisms.

NETSPA augments existing security resources with the ability to generate attack graphs and assist a network administrator in creating secure networks. The resulting attack graphs take into account network and host configurations and individual vulnerabilities, placement of intrusion detection systems, firewalls and associated rulesets, and critical network resources. This allows the security administrator to modify a simulated network and determine the network configurations that minimize attack scenarios and protect critical network elements.

NETSPA is easier to use and faster than other attack graph generation systems. The ease of network configuration entry and the simplicity of the attack language allow a user to quickly and easily define a network and model new exploits and actions. The use of NetViz also provides intuitive user access directly to the network model. Aside from ease of use, NETSPA has proven to be comparable in running times to arguably more efficient model-checking solutions [33], and provides additional information evident in the entire attack graph, as opposed to just a graph where all nodes end in a certain goal state.

Skill	Location	Actions	Internal Hosts Compromised	DMZ Hosts Compromised
Novice	Outside	all known	0/10	0/3
Novice	Inside	all known	8/10	2/3
Novice	Inside	all known	8/10	1/1
Expert	Outside	all known + new IIS exploit	0/10	1/3
Expert	Outside	all known + new BIND exploit	8/10	1/3
Expert	Outside	all known + new Sendmail exploit	0/10	1/3

Table 8.1: A summary of experimental results for a realistic network with multiple DMZ servers.

Skill	Location	Actions	Internal Hosts Compromised	DMZ Hosts Compromised
Expert	Outside	all known + new IIS exploit	8/10	1/1
Expert	Outside	all known + new BIND exploit	8/10	1/1
Expert	Outside	all known + new Sendmail exploit	8/10	1/1

Table 8.2: A summary of experimental results for a realistic network with a single "combined" DMZ server.

A summary of the experiments discussed in Chapter 6 and corresponding results is shown in the tables presented in this chapter. Table 8.1 and Table 8.2 describe the success of novice and expert attackers against the internal and DMZ hosts, both with separate servers and a single combined server. The worst-case scenarios arose when either the attacker was inside the network, or the attacker was able to exploit a new vulnerability against the combined server in the DMZ. Both of these situations allowed the attacker `root` access on 8 out of the 10 inside machines and either 2 out of the 3 DMZ servers, or the single DMZ server. When the DMZ services were split amongst different physical machines, an expert attacker was only able to penetrate the internal network in one of the three cases, exploiting the DNS server, whereas when they were collocated on a single machine, any vulnerability against any of the services resulted in internal network compromises. NETSPA thus clearly demonstrates the relative security of various networks against several different types of attackers, and allows a system administrator to effectively weigh security factors against other considerations, such as useability and cost.

Table 8.3 summarizes the results obtained from running NETSPA with different IDS types and DMZ and internal server versions. An interesting and non-obvious consequence of upgrading internal servers is that a network IDS monitoring traffic flowing into the internal

Actions	IDS Type	Internal Server Version	External Server Version	Visibility
all known + new BIND 9.2.1 exploit	host-based	8.10.2	9.2.1	visible
all known + new BIND 9.2.1 exploit	network-based	8.10.2	9.2.1	visible
all known + new BIND 9.2.1 exploit	host-based	9.2.1	9.2.1	visible
all known + new BIND 9.2.1 exploit	network-based	9.2.1	9.2.1	invisible

Table 8.3: A summary of experimental results with differing IDS types and vulnerable server versions.

Type of Machine With Vulnerability	Internal Hosts Compromised	DMZ Hosts Compromised
user host	1/10	0/3
server host	1/10	0/3
local admin host	3/10	0/3
global admin host	10/10	3/3

Table 8.4: A summary of experimental results enabling one vulnerability at a time.

network will not be able to detect attacks against the internal server as no signatures exist. NETSPA has thus shown that diversity among separate services and components increases a network's security by forcing an attacker to create several new exploits to obtain internal network access.

Finally, Table 8.4 provides a summary of critical internal hosts to be protected in the case of insider attacks. In our cases, a user could exploit a single type of host, be it simple a user's personal host, an internal server machine, the machine of an administrator of a few computers, and the machine of a site-wide administrator. This clearly presents the hosts which, when compromised, impact internal network security more than others. A new vulnerability in any administrator's computer allows instant access to those computers which are under the administrator's jurisdiction. In the case of a site-wide administrator, this effect is instant and devastating, as the attacker has complete administrator access to the entire network, including internal and DMZ servers. These results present a strong case for several separate administrators with non-overlapping administration privileges.

NETSPA has several weaknesses, the largest of which is a limited database of modeled actions, however it has proven its ability to provide useful information in a realistic network setting. Continued use and support of NETSPA will solve most of the current limitations,

and allow NETSPA to become a viable security solution.

Appendix A

REM Description

A complete REM action definition is a set of statements divided into three sections: requirements, effects, and modifies. The capital words are the keywords of REM that define the information contained within the statement. In addition, any words within <> are variables that are bound at the time of action execution and include <current_host>, <current_user>, <target_host>, <target_user>, <exposed_software>, and <port>. Quotation marks are required for all string literals within a statement. The proceeding statement descriptions make use of the following "*variable* ::= literal" bindings:

$host_i$ ::= <current_host>

or <target_host>

$software_i$ ::= "vendor:product name:common version:exact version"

or TYPE ''software type''

$user_i$ ::= none

or nologin

or user

or root

or <current_user>

X ::= integer

A.1 Requirements

host$_i$ `RUNS` *software*$_i$ `[DOS] [TROJAN]`

requires that host$_i$ must run the specified piece of software. The optional `DOS` and `TROJAN` flags are used to determine the required state of the software: `DOS` means that the software is not working properly, whether due to a crash or a denial-of-service attack, and the `TROJAN` flag denotes that the existing software has been replaced by a trojan piece of software. This statement defines the <`exposed_software`> and associated <`port`> variables that can be used in other statements. There can be multiple `RUNS` statement per requires section, however only one will match and define the <`exposed_software`> and corresponding <`port`>.

host$_i$ `ALSO_RUNS` *software*$_i$ `[DOS] [TROJAN]`

defines additional software pieces that must be running in order for action to work. This could include a specific operating system or system tool. The syntax is exactly the same as that of the `RUNS` statement. Multiple `ALSO_RUNS` statements are allowed.

`[`*user*$_i$ `ON` *host*$_i$`]` `HAS_ACCESS` *user*$_j$ `ON` *host*$_j$

requires that *user*$_i$ on *host*$_i$ can login (via authentication or existing trust relationship) to *host*$_j$ as *user*$_j$. If the first user/host pair is left out, <`current_host`> and <`current_user`> are assumed. This statement also sets the <`target_user`> variable, for use in later sections.

host$_i$ `CAN_CONNECT` *host*$_j$ `[PORT` X`]`

requires that *host*$_i$ be able to connect through the network to *host*$_j$ on port X, given all of the firewall rules and possible paths. If the port is omitted, it is assumed to be the port of the exposed software.

A.2 Effects

`[`*user*$_i$ `ON` *host*$_i$`]` `HAS_ACCESS` *user*$_j$ `ON` *host*$_j$

provides a trust relationship between *user*$_i$ on *host*$_i$ and *user*$_j$ on *host*$_j$. If the from user/host pair is not provided, this establishes a blanket trust relation from any user/host combination to the target host.

host$_i$ RUNS {*software*$_i$ or <exposed_software>} [PORT X] [DOS] [TROJAN]

changes the state of the system by starting up the *software*$_i$ on port X. If <exposed_software> is used instead, this changes the state of the exposed software.

TRUST_RELATIONSHIPS

provides the current user with all of the network trust relationships that pertain to any host on the same subnet as <target_host>. This is used to model a sniffer.

A.3 Modifies

<current_host> = <target_host>

transitions the attacker from location <current_host> to <target_host>.

<current_user> = {*user*$_i$ or <target_user>}

transitions the current user from <current_user> to *user*$_i$ or <target_user>.

Appendix B

Realistic Actions

The following list defines the base set of actions used to create attack graphs against the realistic network. Each action and its coresponding description, CVE number (if one exists), locality, visibility to an IDS, and partial REM specification is included.

Remote Login : authorized login from one host to another. Remote. Visible

Requires: `<target_host> RUNS TYPE "Remote Login Server"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`
`HAS_ACCESS <target_user> ON <target_host>`

Modifies: `<current_host> = <target_host>`
`<current_user> = <target_user>`

Brute Force Password Crack : crack a user password after obtaining some prior knowledge of the system. Remote. Invisible.

Requires: `HAS_ACCESS nologin ON <target_host>`

Effects: `HAS_ACCESS user ON <target_host>`

SNMP Information Leak : NETBIOS share information may be published through SNMP registry keys in NT. CAN-1999-0499. Remote. Visible.

Requires: `<target_host> RUNS "-:snmp:-:-"`
`<target_host> ALSO_RUNS "Microsoft:Windows 2000:Advance Server:Advance Server beta 3"`
`<current_host> CAN_CONNECT <target_host>`

Effects: `HAS_ACCESS nologin ON <target_host>`

Install and Run Sniffer : install a sniffer to operate the target machine in promiscuous mode or locally obtain keystrokes. CAN-1999-0530. Local. Visible.

Requires: `HAS_ACCESS root ON <target_host>`

Effects: `<target_host> RUNS "-:sniffer:-:-"`
`TRUST_RELATIONSHIPS`

Linux nfsd Buffer Overflow : buffer overflow in NFS server on Linux allows attackers to execute commands via a long pathname. CAN-1999-0832. Remote. Visible.

Requires: `<target_host> RUNS "-:rpc.nfsd:-:-"`
`<target_host> ALSO_RUNS "RedHat:Linux:5.2:-"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`

Modifies: `<current_host> = <target_host>`
`<current_user> = root`

Tektronix 840 Web Server Vulnerability : web server in Tektronix PhaserLink Printer 840.0 and earlier allows a remote attacker to gain administrator access by directly calling undocumented URLs such as ncl_items.html and ncl_subjects.html. CAN-1999-1508. Remote. Visible.

Requires: `<target_host> RUNS "-:Micro-HTTP:1.0:-"`
`<target_host> ALSO_RUNS "Tektronix:Phaser Network Printer:840:-"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`

Modifies: `<current_host> = <target_host>`
`<current_user> = root`

Linux rpc.statd. Format String Error : rpc.statd in the nfs-utils package in various Linux distributions does not properly cleanse untrusted format strings, which allows remote attackers to gain root privileges. CVE-2000-0666. Remote. Visible.

Requires: <target_host> RUNS "-:rpc.statd:-:-"
<target_host> ALSO_RUNS "RedHat:Linux:7.2:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Modifies: <current_host> = <target_host>
<current_user> = root

List NT Domain Users via Null Session : Windows NT allows remote attackers to list all users in a domain by obtaining the domain SID with the LsaQueryInformationPolicy policy function via a null session and using the SID to list the users. CVE-2000-1200. Remote. Invisible.

Requires: <target_host> RUNS "-:smb:-:-"
<target_host> ALSO_RUNS "Microsoft:Windows NT:4.0:-"
<current_host> CAN_CONNECT <target_host>
Effects: HAS_ACCESS nologin ON <target_host>

BIND 8 Buffer Overflow in TSIG : buffer overflow in transaction signature (TSIG) handling code in BIND 8 allows remote attackers to gain root privileges. CVE-2001-0010. Remote. Visible.

Requires: <target_host> RUNS "ISC:BIND:8.2.2 p3:-"
<target_host> RUNS "ISC:BIND:8.1.2:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Modifies: <current_host> = <target_host>
<current_user> = root

SSH Integer Overflow : CORE SDI SSH1 CRC-32 compensation attack detector allows remote attackers to execute arbitrary commands on an SSH server or client via an integer overflow. CVE-2001-0144. Remote. Visible.

Requires: <target_host> RUNS "SSH Communications Security:SSH:1.2.27:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Modifies: <current_host> = <target_host>
<current_user> = root

IIS ISAPI Print Request Overflow : buffer overflow in Internet Printing ISAPI extension in Windows 2000 allows remote attackers to gain root privileges via a long print request that is passed to the extension through IIS 5.0. CVE-2001-0241. Remote. Visible.

Requires: <target_host> RUNS "Microsoft:IIS HTTP:5:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Modifies: <current_host> = <target_host>
<current_user> = root

IIS ..\ Directory Traversal : directory traversal vulnerability in IIS 5.0 and earlier allows remote attackers to execute arbitrary commands by encoding .. (dot dot) and "characters twice. CAN-2001-0333. Remote. Visible.

Requires: <target_host> RUNS "Microsoft:IIS HTTP:5:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Modifies: <current_host> = <target_host>
<current_user> = root

MySQL .. Directory Traversal : directory traversal vulnerability in MySQL before 3.23.36 allows local users to modify arbitrary files and gain privileges by creating a database whose name starts with .. (dot dot). CVE-2001-0407. Local. Visible.

Requires: <target_host> RUNS "T.C.X DataKonsult:MySQL:3.20:-"
<target_host> RUNS "T.C.X DataKonsult:MySQL:3.22:-"
Modifies: <current_user> = root

NTP Buffer Overflow : buffer overflow in ntpd ntp daemon 4.0.99k and earlier (aka xntpd and xntp3) allows remote attackers to cause a denial of service and possibly execute arbitrary commands via a long readvar argument. CVE-2001-0414. Remote. Visible.

Requires: <target_host> RUNS "Dave Mills:ntpd:4.0.99k:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Effects: <target_host> RUNS <exposed_software> DOS
Modifies: <current_host> = <target_host>
<current_user> = root

IIS DOS via WebDAV Request : Vulnerability in IIS 5.0 allows remote attackers to cause a denial of service (restart) via a long, invalid WebDAV request. CAN-2001-0508. Remote. Visible.

Requires: <target_host> RUNS "Microsoft:IIS HTTP:5:-"
<current_host> CAN_CONNECT <target_host> PORT <port>
Effects: <target_host> RUNS <exposed_software> DOS

Wu-ftpd Glob Handling Error : wu-ftpd 2.6.1 allows remote attackers to execute arbitrary commands via a "~{" argument to commands such as `CWD`, which is not properly handled by the glob function. CAN-2001-0550. Remote. Visible.

Requires: `<target_host> RUNS "University of Washington:Wu-ftpd:2.6.0:-"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`
Effects: `<target_host> RUNS <exposed_software> DOS`
Modifies: `<current_host> = <target_host>`
`<current_user> = root`

CDE `dtspcd` Overflow : buffer overflow in the client connection routine of libDtSvc.so.1 in CDE Subprocess Control Service (dtspcd) allows remote attackers to execute arbitrary commands. CVE-2001-0803. Remote. Visible.

Requires: `<target_host> RUNS "-:dtspcd:-:-"`
`<target_host> ALSO_RUNS "Open Group:Common Desktop Environment:1.2:-"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`
Modifies: `<current_host> = <target_host>`
`<current_user> = root`

Windows SNMP Malformed Management Request : buffer overflow in SNMP agent service in Windows 95/98/98SE, Windows NT 4.0, Windows 2000, and Windows XP allows remote attackers to cause a denial of service or execute arbitrary code via a malformed management request. CAN-2002-0053. Remote. Visible.

Requires: `<target_host> RUNS "-:snmp:-:-"`
`<target_host> ALSO_RUNS "Microsoft:Windows 2000:Advance Server:Advance Server beta 3"`
`<current_host> CAN_CONNECT <target_host>`
Effects: `<target_host> RUNS <exposed_software> DOS`
Modifies: `<current_host> = <target_host>`
`<current_user> = root`

IIS FTP DOS : the FTP service in Internet Information Server (IIS) 4.0, 5.0 and 5.1 allows attackers who have established an FTP session to cause a denial of service via a specially crafted status request. CAN-2002-0073. Remote. Visible.

Requires: `<target_host> RUNS "Microsoft:IIS FTP:5:-"`
`HAS_ACCESS user ON <target_host>`
`<current_host> CAN_CONNECT <target_host> PORT <port>`
Effects: `<target_host> RUNS <exposed_software> DOS`

PHP Upload Buffer Overflow : buffer overflows in (1) `php_mime_split` in PHP 4.1.0, 4.1.1, and 4.0.6 and earlier, and (2) `php3_mime_split` in PHP 3.0.x allows remote attackers to execute arbitrary code via a multipart/form-data HTTP POST request when `file_uploads` is enabled. CAN-2002-0081. Remote. Visible.

Requires: `<target_host> RUNS "PHP:PHP:3.0.18:-"`
`<target_host> ALSO_RUNS "Apache Group:Apache:1.3.9:-"`
`<current_host> CAN_CONNECT <target_host> PORT <port>`
Modifies: `<current_host> = <target_host>`
`<current_user> = user`

OpenSSH Off-By-One Error : off-by-one error in the channel code of OpenSSH 2.0 through 3.0.2 allows local users or remote malicious servers to gain privileges. CAN-2002-0083. Local. Visible.

Requires: `<target_host> RUNS "OpenBSD:OpenSSH:2.9:-"`
Modifies: `<current_user> = root`

Bibliography

[1] Steven Cheung. CAML predicates, December 2001. Personal communications.

[2] Steven M. Christey, David W. Baker, William H. Hill, and David E. Mann. The development of a common vulnerabilities and exposures list. In *Proceedings on the Second International Workshop on Recent Advances in Intrusion Detection (RAID'99)*, Purdue University, West Lafayette, Indiana, September 1999. `http://cve.mitre.org`.

[3] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[4] Computer Security Division at the National Institute of Standards and Technology. ICAT Metabase: A CVE based vulnerability database. `http://icat.nist.gov`.

[5] Cooperative Association for Internet Data Analysis (CAIDA). Code-Red Worms: A Global Threat, November 2001. `http://www.caida.org/analysis/security/code-red/`.

[6] Frederic Cuppens and Alexandre Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[7] Frederic Cuppens and Rodolphe Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proceedings of the Third Workshop on the Recent Advances in Intrusion Detection (RAID 2000)*, Toulouse, France, October 2000.

[8] Renaud Deraison. The Nessus project. `http://www.nessus.org`.

[9] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 2000.

[10] Steve T. Eckmann. Translating snort rules to STATL scenarios. In *Proceedings on the Fourth International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA, October 2001.

[11] Fyodor. nmap: Network security scanner. `http://www.nmap.org`.

[12] Nick Gorham. unixODBC. `http://www.unixodbc.org`.

[13] G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proceedings of the 6th International Conference on Concurrency Theory*, volume 962 of *LNCS*, pages 453–455, Berlin, Ger, August 1995. Springer.

[14] E. Koutsofios and S. North. Drawing graphs with *dot*. Technical report, AT&T Research Labs, Murray Hill, NJ, October 1993. `http://www.research.att.com/sw/tools/graphviz/`.

[15] Ivan Krsul, Eugene Spafford, and Mahesh Tripunitara. Computer vulnerability analysis. Technical Report COAST TR98-07, Purdue University, COAST Laboratory, West Lafayette, IN, May 1998.

[16] Richard Lippmann, Seth Webster, and Douglas Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October 2002.

[17] John Maddock. regex++. `http://www.boost.org/`.

[18] K. I. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[19] Microsoft Corporation. Windows XP. One Microsoft Way, Redmond, WA 98052-6399. `http://www.microsoft.com`.

[20] Andrew P. Moore, Robert J. Ellison, and Richard C. Linger. Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, Carnegie Mellon University, March 2001.

[21] MySQL AB. MySQL. Bangrdsgatan 8, S-753 20 Uppsala, Sweden. `http://www.mysql.org`.

[22] NetViz, Inc. NetViz Professional. 12 South Summit Ave, Suite 300, Gaithersburg, MD 20877. `http://www.netviz.com`.

[23] R. Power. 2001 CSI/FBI Computer Crime and Security Survey. *Computer Security Institute*, Spring 2001. `http://www.gocsi.com/forms/fbi/pdf.html`.

[24] C.R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 2000.

[25] RedHat, Inc. RedHat Linux. 1801 Varsity Drive, Raleigh, NC 27606. `http://www.redhat.com`.

[26] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2000.

[27] B. Robert. Kuang: Rule-based security checking. Technical report, MIT, Lab for Computer Science Programming Systems Research Group, May 1994. Documentation in `ftp://ftp.cert.org/pub/tools/cops/1.04/cops.tar`.

[28] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference (LISA '99)*, Seattle, WA, November 1999. `http://www.snort.org`.

[29] Bruce Schneier. Attack trees. *Dr. Dobbs Journal*, 1999. `http://www.counterpane.com/attacktrees-ddj-ft.html`.

[30] SecurityFocus, Inc. 1660 S. Amphlett Blvd., Suite 128, San Mateo, CA 94402. `http://www.securityfocus.com`.

[31] SecurityFocus, Inc. Bugtraq mailing list. `http://online.securityfocus.com/archive/1`.

[32] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[33] O. Sheyner, J. Haines, S. Jha, R. Lippman, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings on the 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[34] Laura P. Swiler, Cynthia Phillips, David Ellis, and Stefan Chakerian. Computer-attack graph generation tool. In *Proceedings of the 2001 DARPA Information Survivability Conference and Exposition II (DISCEX II)*, Anaheim, CA, June 2001.

[35] System Administration, Networking and Security Institute (SANS). Nimbda worm/virus report, October 2001. `http://www.incidents.org/react/nimda.pdf`.

[36] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms*, Cork, Ireland, September 2000.

[37] G. Vigna, S.T. Eckmann, and R.A. Kemmerer. The STAT tool suite. In *Proceedings of the 2000 DARPA Information Surviveability Conference and Exposition (DISCEX)*, Hilton Head, SC, January 2000.

[38] D. Zerkle and K. Levitt. NetKuang – a multi-host configuration vulnerability checker. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, 1996.

[39] Zone Labs, Inc. ZoneAlarm Pro. 1060 Howard Street, San Francisco, CA 94103. `http://www.zonelabs.com`.