A Taxonomy of Computer Intrusions

by

Daniel James Weber

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

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author .....
Department of Electrical Engineering and Computer Science
May 29, 1998

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

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

A Taxonomy of Computer Intrusions

by

Daniel James Weber

Submitted to the Department of Electrical Engineering and Computer Science
on May 29, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Software and procedures were developed to generate both normal background inter-computer TCP/IP sessions and attacks on a dedicated computer network of UNIX workstations. Network applications such as telnet and sendmail are driven automatically without human intervention to generate artificial traffic. Network monitoring data, including TCP/IP packets, and Sun Basic Security Module (BSM) auditing data, collected while artificial traffic is being generated is being used to evaluate the false alarm and detection rate of new intrusion detection systems. More than twenty different attacks were automated and a small network is currently using the traffic-generating software to emulate traffic between more than 100 hosts and 1000 users. In addition, a taxonomy of UNIX computer intrusions was created and is being used to select a comprehensive set of attacks for evaluating intrusion detection systems.

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

Acknowledgments

I would like to thank all my lab-mates at Lincoln Laboratory, including my thesis advisor, Richard Lippmann, for his tenacity and good humor in getting me through my research and my thesis; Samuel Gorton for his assistance and inspiration in the creation of the taxonomy; and my office-mate Seth Webster for providing IDS models for me to try to break and for putting up with name collisions.

I acknowledge the support of Teresa Lunt at DARPA, the Office of the Secretary of Defense, and Terry Champion, Steve Durst, and Eric Miller at the Air Force Rome Laboratory.

I would like to thank my parents for their support during the last twenty years of my education.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview of Computer Security

Computer security in this thesis is defined as by Garfinkel and Spafford [11]: "A computer is secure if you can depend on it and its software to behave as you expect." That is, private data stays private, un-privileged users remain un-privileged users, and all services of the system continue to function. Many recent news articles illustrate a growing concern with this problem.

In September 1996, subscribers of the Internet Chess Club found themselves unable to access their system for several days due to the actions of an anonymous person on the Internet [17]. In January 1997, a group of German computer users demonstrated an ActiveX control that could transfer money out of a Quicken user's account without the need for a personal identification number [24]. In March 1998, a teenager disabled critical communications at a Massachusetts airport for several hours [18].

More and more computers are being connected to networks, including the Internet, every day. Each system, to varying degrees, is vulnerable to misuse from outsiders. A firm's financial data, a pharmacy's database of its customers' drug needs, or an individual's personal e-mail are possible targets for an attacker. This information might be disclosed beyond its intended audience, subtly altered, or cut-off from legitimate users.

Attacks are not difficult for even the unskilled to launch. Sites on the World

Wide Web freely give out programs that allow novice users to run powerful attacks. The method of attack that was used against the Internet Chess Club was published in semi-underground computer journals, and from there took a minimal amount of knowledge to employ.

Threats are by no means limited to the attacks of random, unskilled people who find attack tools. Insiders in a company, who know what is vulnerable and what is valuable, can cause tremendous damage. More sophisticated computer intruders, skilled hackers who know how to explore a system for its own peculiar vulnerabilities while hiding their actions, are probably rarer but still dangerous.

If intruders can gain enough access to a system, they can use it as a new base to launch further attacks. Breaking into one computer on a network often simplifies breaking into other computers on the network, as those systems may be programmed to automatically trust each other. Even if they don't, critical data is often sent over a network unencrypted, allowing any computer connected to that network to read it. Once a hacker has broken into a computer, it is difficult to restore it to a secure state without wiping it clean and re-installing the original operating system.

1.2 Towards Computer Security

It is very difficult to secure a computer totally, and firewalls cannot keep out all intruders. Intrusion detections systems can help fill the gap.

There is a fundamental trade-off among security, usability, and cost. In general, the more secure a computer is, the more difficult it is to use. One simple method to secure a computer is to unplug it from the network. Such a disconnected computer is not very useful, however. Networks were developed so that computers could exchange data easily.

An alternative is to make each individual machine usable and secure, using two privilege levels. Normal users can have low level privileges while the root user, the master user on a computer, can do just about anything to the system. This is difficult to implement perfectly, however, as commercial operating systems will invariably be

found to contain *security holes* after they have shipped. A common security hole with UNIX systems is that certain programs on the computer, referred to as "setuid root," are specially trusted. When executed by a normal user they have the privileges of the root user, since they need to do some special action that is denied to normal users. An example is checking the free space on a disk. (If normal users could directly access the hard drive, they could read any file they wished.) Writing these programs in a secure fashion, however, is not a simple task, and one that doesn't necessarily have the highest priority for computer vendors. Bugs that allow normal users to run arbitrary code as root are inevitably found in many commercial operating systems, and administrators must go back later to fix or "patch" these holes.

One could also attempt to restrict access to the entire network with a firewall, a computer that sits between the network and its connection to an outside network, such as the Internet. Every piece of data that wants to pass through this boundary must go through the firewall. Firewalls can block certain connections based on simple rules. For example, all login attempts from outside to inside could be automatically rejected. Another rule which a firewall could implement is to drop any network traffic from a site known to harbor problem users, while sending notification of the attempt to the administrator.

Firewalls, however, can have security faults the same way individual computers can, since firewalls are also computers. The security provided by a firewall often degrades over time, as more holes are opened to allow remote users access to more and more services. The rules that the firewall uses to determine which packets are allowed through need to be updated regularly, as out-of-date rules may not provide protection against recent attacks. The firewall could also be bypassed by poor security practices of the internal users. For example, an outsider can mail a user a program which, when run, maps the inside network and mails the findings back out, or an inside user may use a modem to connect to the outside over phone lines.

Use of a firewall and frequent software updates can not prevent attacks. While fixing security holes is a vital part of maintaining a machine, an administrator needs to keep a close eye on the system, including monitoring logs for unusual behavior. A

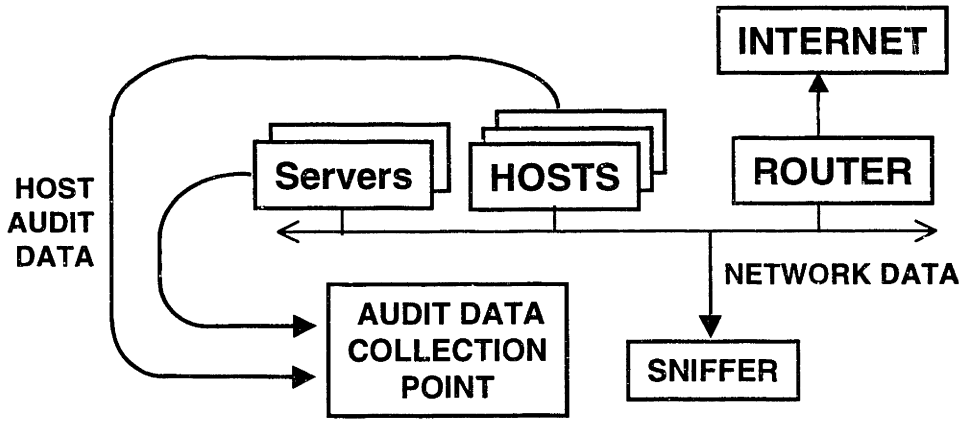

Figure 1-1: Data For an IDS Can Come From a Network Sniffer or Audit Logs

useful tool that can help automate this task is an Intrusion Detection System (IDS).

1.3 Intrusion Detection Systems

An IDS does not necessarily attempt to prevent intrusions to a computer or network of computers; instead, it gathers information from the system, and attempts to detect intruders. Reports of possible intrusions are passed to a human analyst for closer examination.

The data that an IDS gathers come from two sources. One is the computer network: all data passes over the network in packets that are typically visible to any computer that wishes to monitor it. The data-gathering component of the IDS records all of these network packets with a program such as `tcpdump`, a program which captures this network traffic and saves it to a file. So that all data coming in to or going out of the network can be analyzed, this component is usually placed near the network's router.

The second data source an IDS may use is system audit data. Many computer operating systems log important security related events that occur on its system. Logs may be terse, such as only logging failed password attempts, or very verbose, such as the output of Sun's Basic Security Module (BSM) [22], which logs every system call made on the system, including all reading of and writing to the filesystem. After the

data is logged, it must then be collected.

Figure 1-1 has a representation of a typical network, showing hosts and servers along a network, which is connected through a router to the Internet. The sniffer reads network data off of the wire, while some machines are generating their own audit data.

After gathering input data, another component of the IDS processes the results. This may occur in real-time, which allows administrators or the IDS to take immediate defensive action against possible threats, or off-line, in which case the IDS analyzes the data and reports results at a later time to a human analyst, who then must go to the system to inspect the situation more closely.

Signature detection schemes look for a specific sequence of events to detect known attacks. Some signature detection IDSs monitor TCP/IP network telnet connections between computers and watch for certain events that may indicate intruder activity. For example, the command `loadmodule` can be used by normal users to load drivers into the kernel during run-time, though it is seldom used this way in practice. On some UNIX operating systems it also has a serious security hole that allows users to run commands as root. Thus, many signature-based schemes that match on strings trigger on the keyword "loadmodule". However, not all keywords are as discriminating. Strings such as "passwd" and "rhosts" may be useful in identifying hacker activity, but they are common in normal traffic as well, especially by legitimate system administrators. Furthermore, altering the strings to get around the IDS is fairly straightforward. An example of a signature detection system is the Network Security Monitor, described in [12], which uses text strings as the basis for its signatures.

Anomaly detection schemes build up models of the typical behavior of a system and issue warnings when activity deviates from normal. Anomaly detection may monitor many aspects of a system. It can be used to characterize patterns of network traffic, the use of system calls by daemons, or the typing speed of users on the system. Of course, existence of anomalies does not necessarily mean intruder activity, and intruders do not always appear anomalous. Example anomaly detection systems are Haystack [21], for which models of behavior for users and groups must be built by

a human operator, and NIDES [3], which automatically builds short- and long-term profiles for each user.

Bottleneck verification systems watch for processes transitioning from a user-level status to a privileged-level status and verify that the state transition took place via an approved route – the "bottleneck". These systems can be network-based, such as checking that new root prompts were preceded by `su` commands, or host-based, such as watching that the password file is only modified by approved programs, such as `passwd`. A prototype bottleneck verification system is described in [23].

1.4 Structure of This Thesis

The work described in this thesis is part of a larger project to test and evaluate intrusion detection systems. This thesis focuses on ways of creating traffic–both attacks and "normal data"–that can be passed through non-real-time IDSs to see what kind of attacks they recognize and how many false alarms they generate.

Chapter Two gives an overview of testing IDSs. Chapter Three discusses a strategy to simulate many users automatically and repeatedly. Chapter Four attempts to taxonomize computer attacks, particularly in a way that is useful for testing IDSs. Chapter Five presents some aspects of the attacks used and demonstrates the use of the taxonomy. Chapter Six contains conclusions and ideas for future work.

Chapter 2

Testing Intrusion Detection Systems

A major goal of this thesis work is to devise approaches to test and evaluate intrusion detection systems. Test and evaluation approaches are essential to compare the differing techniques in intrusion detection and to point out new directions in research. Initial work examined the performance of one sample IDS, and further work sought to create a framework to test other non-real-time IDSs against the same data.

2.1 Experience With One IDS

Initial work involved a baseline IDS that was similar to the Network Security Monitor, NSM [12]. This IDS is described more thoroughly in [23]. It captures all packets in TCP/IP network traffic and reconstructs the packet contents into TCP sessions. A session is a record of the data sent between one machine and another over the course of a network connection, broken down by the bytes that each computer sent. The IDS then searches each session for a set of pre-programmed keywords that identify a session as "dangerous." These are strings that may indicate either an attempted exploitation of a known security hole or suspicious actions that an intruder night perform after gaining entry.

An entire session is given a warning value on the scale of 0 to 10, based on the

```
}}}#}'}${~ ~!~"|~$zpz#pz'p

UNIX(r) System V Release 4.0 (pascal)


{}~login: joe
Password:
Last login: Fri Jul 18 12:25:46 from plato
Sun Microsystems Inc.   SunOS 5.5       Generic November 1995
joe@pascal: emacs
```

Figure 2-1: A Sample Transcript

number and type of keywords that were found coming from each direction (from the source or from the destination). If this score is above a certain *threshold*, a report is generated for a human analyst. The header of this report contains counts of the keywords that were found in the session, and the body contains all the bytes sent in each direction. Part of a sample transcript is shown in Figure 2-1, listing the bytes that the destination machine sent back. This transcript shows a telnet session for a user named *joe* logging into a host named *pascal* and running the `emacs` editor. The initial line of characters represents environment information passed between computers when a telnet sessions starts.

Ideally, sessions that are actually attacks would receive high scores, while those that are not would receive low scores. In practice, some dangerous sessions get lower scores than some harmless sessions. The user can trade off the detection and false alarm rates by setting a threshold. All sessions that score higher than the threshold are turned into transcripts for a human analyst. The lower the threshold is, the more attacks will be flagged. The higher the threshold is, the fewer false alarms that the humans will need to search to find attacks.

The baseline IDS was testing using two sets of attacks. One set was composed of *old attacks*, attempts to exploit well-known security vulnerabilities, such as trying to log into default accounts. There was also a set of *new attacks*, attacks that had been discovered after development had stopped on the baseline IDS, such as the security hole in the `eject` program that allowed users to become root.

Each set was tested against a large quantity of "normal sessions." These are sessions in which a security breach was not being attempted. To gather this body of data, traffic was collected from fifty sites with similar traffic patterns. Over 3.4 million TCP sessions were captured, and of these 86,015 were telnet sessions.

To test each set of attacks, each attack was inserted ten times into the normal data. There were a total of 30 instances of old attacks and 50 instances of new attacks.

IDSs were compared using a receiver-operating characteristic (ROC) curve. An ROC curve is a visual representation of how well a system can discriminate events (attacks, in this case) from non-events (normal activity). The percentage of false alarms on the X-axis is plotted against the percentage of attacks that were found on the Y-axis, over the range of possible thresholds. The performance of the system is reflected in the behavior of the plotted line, which starts in the lower-left and makes its way towards the upper-right. One general measure of the performance of an IDS is the area under the ROC curve–the higher this area is, the better the system is in differentiating normal sessions from attacks.

If a system were perfect in recognizing attacks and never issued a false alarm, the plot of the ROC curve would be a line that starts straight up the left-hand side of the graph and then continues along the top line. Such a system would have 100% detection with 0% false alarms. If a system were randomly guessing, the plot would be a 45° line.

The first curve in Figure 2-2 shows the ROC curve for the baseline IDS when tested against the old attacks. The IDS achieves a 20% detection rate at a 15% false alarm rate. To achieve a 100% detection rate 45% of normal sessions are incorrectly labeled as attacks. The IDS had a *purity* of about 1:16,000. The purity is the measure of how much background traffic the human analysts would need to look through to find an attack when the IDS was detecting 80% of attacks. For this data, a human analyst using the baseline system would have to examine 16,000 transcripts to find each attack.

The IDS performed much worse against the new attacks. The ROC curve is strictly under the 45° line. In other words, random guessing would yield better results than

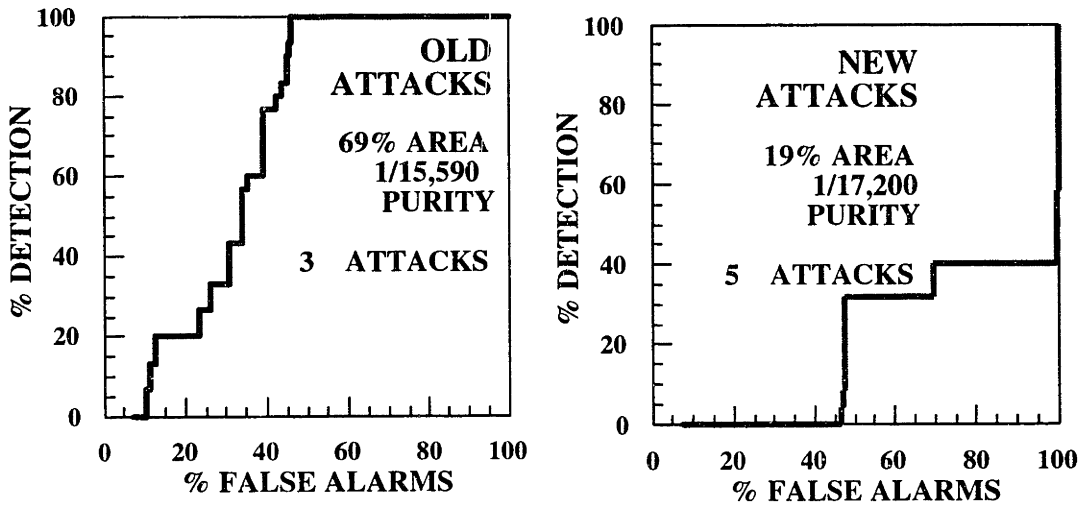

Figure 2-2: ROC Curves

the baseline system.

The false alarms were generated by sessions in which keywords were used in non-nefarious circumstances. For example, a single "login incorrect" in a telnet session pushes its warning value above 7. The string "permission denied", which occurs frequently, also generated a large number of false alarms.

This initial evaluation illustrates the need to use a large amount of normal traffic as well as attacks to evaluate intrusion detection systems. The normal traffic is required to measure false alarm rates while attacks are required to measure detection rates. This evaluation was limited because the normal telnet traffic used was proprietary and private, it had already been pre-selected based on scores of the baseline system, and ground truth concerning the presence or absence of attacks had not been established. In addition, only a small number of attacks were used. A major goal of this thesis was to extend this approach to evaluating IDSs and correct the limitations of this initial evaluation by developing procedures that can be used to generate automatically large amounts of non-proprietary network data and by developing a larger collection of attacks.

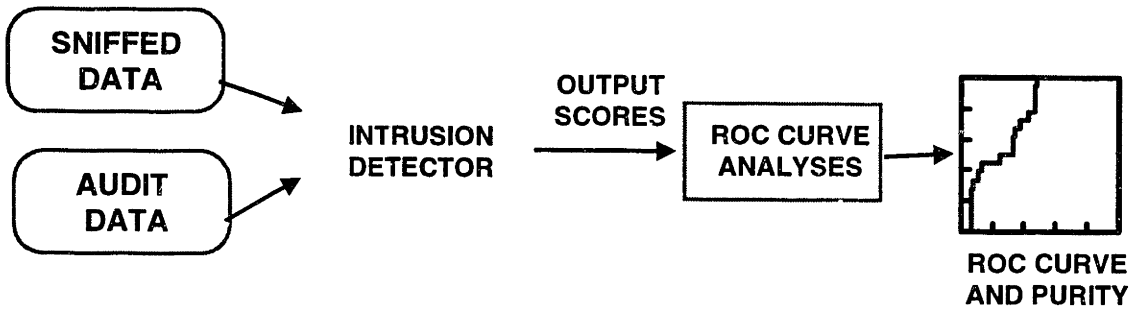

Figure 2-3: Testing Procedure

2.2 Possible Testing Strategies

A methodology for testing intrusion detection systems was proposed in [20]. It called for setting up an isolated network and automating attacks with *expect* [15]. Simple models of normal users, such as secretaries and programmers, were provided.

The methodology presented in [20] serves as a foundation for work performed as part of this thesis. This methodology was extended by simplifying the generation of user actions and attacks using a new file format (the .ex file file format described in section 3.4) to generate action scripts, by creating a much richer set of user actions, by developing many more attacks, and by developing techniques to monitor the progress of a large simulation network.

To test multiple IDSs, a large quantity of data used by IDSs was required (e.g., dumps of network traffic and BSM audit data) that could be distributed to IDS developers. Figure 2-3 shows a model of providing this data to an IDS, which will report its results. The output results will indicate a warning score for each session, and from this an ROC curve can be generated. Each IDS will have an ROC curve associated with it, and these curves can be compared with metrics such as area and purity.

This raises the question of how to generate the data. For accurate IDS evaluations, the ideal way to generate this data would be to capture the data off of an actual network containing real users doing real work. As the network is running, various attacks would be run on the system, breaking in or disabling services.

This is not a realistic strategy, of course. People are using a network in order to

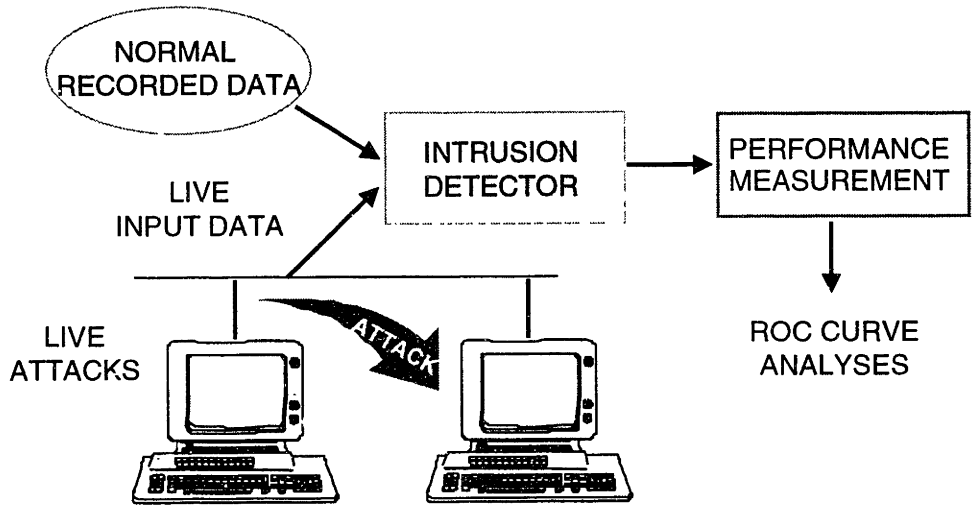

Figure 2-4: Recording Data From A Real Network, Then Adding Attacks

get work done and would not appreciate random interruptions of service or possible exposure of their personal data to other users on the system during security breaches. A second serious problem is the violation of the users' privacy. All the data that is collected will include personal information, such as usernames, passwords, and the contents of electronic mail. These data cannot be distributed to IDS developers.

Another testing strategy that was considered was to acquire data by monitoring real users on an actual network. The identities of the real users could be concealed, by creating new user names and replacing any personal data with statistically-equivalent nonsense text. Attacks would be run on a separate network. The required data would be recorded and then inserted into the data from the actual network. Figure 2-4 depicts attacks being recorded and mixed with the normal data before passed trough an IDS.

Among the reasons why this was not implemented was the difficulty in cleanly inserting attacks into actual data. To insert an attack where the user successfully logs in would require simulating a new process being launched. This will throw off the system's count of process id numbers (PIDs), which are assigned sequentially, unless some existing process in the actual data is removed.

Furthermore, if another user had at that time checked to see who else was on

the system, it would be necessary to modify the output of that user's command in the data. Otherwise, a sufficiently observant IDS would notice a user had logged in and was invisible to everyone else. Other IDSs may notice this inconsistency and determine that this attack was artificially appended into the data. This type of clue is an "artifact" that should be avoided.

It may be possible to work around some of these problems by having dummy users on the actual system while it is being recorded, doing simple token actions that serve as placeholders to be replaced. However, these problems are too complex to tackle at this time, especially when the quality of the output would be uncertain.

An alternative concept is to capture actual user sessions off a real network, and extract the commands that the user had typed and the timing information associated with those commands. Given a series of sessions composed of commands and timing information, it is easy to go back through and add our own attacks, and alter existing sessions to include hacker or anomalous activity. These sessions would then need to be "regenerated," re-run over the network. This time, network traffic and audit data would be consistent and not contain artifacts, because what is seen is the result of running actual commands. Figure 2-5 shows data from a real network being sanitized to replace personal information and analyzed to extract commands. These commands are then regenerated by computers on another network, with attacks added, and new input data for an IDS is recorded. This regeneration scheme is discussed in Chapter Three.

Careful analysis led to the conclusion that purifying the data consistently is too difficult. Sensitive information, including items like the names of machines and directories, is hard to cleanse reliably in a consistent manner, and information concerning the file structure and programs on mounted hosts is difficult or legally impossible to obtain.

A modified approach was developed where types of users are characterized and simulated using "automata" running from *perl/expect* scripts. These users run various random tasks while they are logged in, and their global behavior statistically matches that of actual users from the real network.

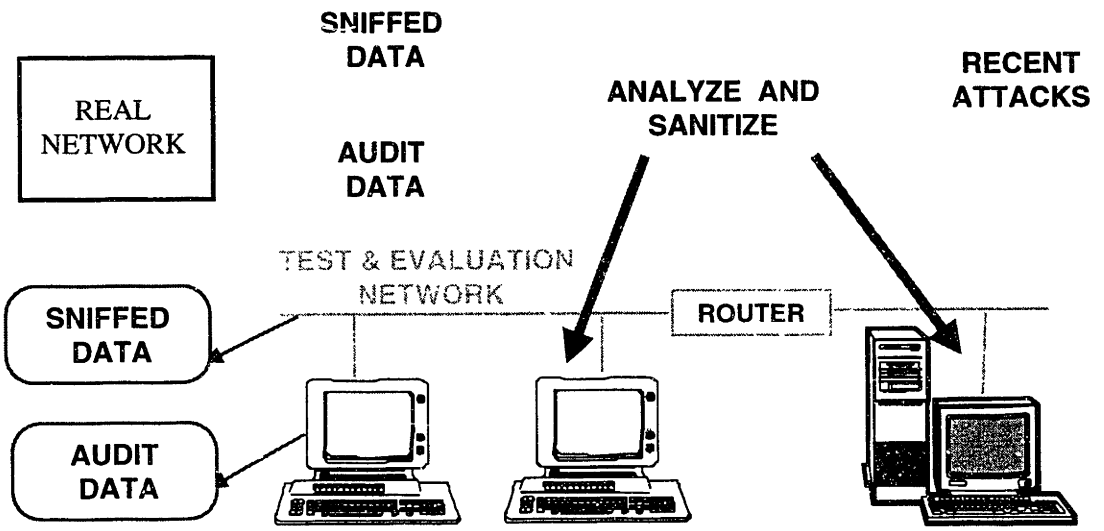

Figure 2-5: A Testing Strategy to Remove Artifacts From Data

2.3 Sessions

A "user session" is a TCP session over a network service, such as telnet, HTTP, or SMTP. Since the IDSs can have very high false alarm rates, and in the real world actual attacks are a very small part of the total data, a large quantity of normal traffic is required.

A sizable quantity of attacks is also needed. In order to see how well the IDS detects an attack, each attack should occur several times in the data, with different levels of "stealthiness." There should be several different kinds of attacks, so that an accurate measurement can be made of the types of attacks that the IDS catches. Some novel attacks, for which "exploit code" is not publicly available, are useful for seeing how IDSs react against attacks they have not been trained to find. The selection of attacks is described more in Chapter Five.

Chapter 3

Traffic Regeneration

In Chapter Two, a methodology was presented for generating data that could be provided to an IDS in a way that minimizes artifacts. Artifacts in the data might be used by an IDS, intentionally or not, to help distinguish between background traffic and actual attacks. To minimize artifacts, an identical approach is used to regenerate both normal data and attacks.

3.1 Regeneration

Regeneration is used to create automatically network TCP/IP sessions containing artificial normal traffic or attacks as if they were being produced by actual computer users.

Goals of regeneration were that traffic generation must be:

- Automatic, requiring no human intervention.
- Reproducible, so that either many sessions or individual sessions could be run again.
- Robust, so that it can be run for a long period of time.
- Recoverable, so that if regeneration fails part of the way through it is possible to continue it later.

- Informative, providing detailed logging information to validate and confirm runs.

3.2 Simulating Multiple Machines

To simulate multiple machines on a network, each UNIX machine on which a regenerator runs has a modified kernel. This modified kernel from Rome Labs [8], a variant of the Linux version of UNIX, allows each program running on the machine to act as if it is using a different IP address.

Briefly, an IP swapping technique works through a device driver called "`ipswap`" in conjunction with the character-special file `/dev/ipst`. When an authorized user wants to launch a network program with a fake IP address, the program's PID and desired fake source address are written to a table by `ipswap`, and the kernel opens a port with that fake IP address.

There are other identifying features of a system that need to be fixed so that it can appear to be multiple machines to the network. Many programs that interact with the network, including `sendmail`, `ftpd`, and `login`, identify the machine they are running on. These programs need to be altered as well.

The regenerator can also function normally without the modified kernel, when multiple machines do not need to be simulated.

3.3 The Framework

As suggested in [20], the *expect* program is used to automate interaction, allowing a shell session to be run as if a user were typing at a keyboard [15]. However, it would be too much work to code up an *expect* script for every session, and it would still require another *expect* program or shell script to call each individual session. Also, such a scheme would make code maintenance difficult.

For this reason, a regenerator was written in *expect* that reads in information about hosts, prompts, commands, and timing from specially formatted ".ex" files

which control separate sessions.

3.4 File Format

There is one .ex file for each session that is to be regenerated. The header of each file contains the following information:

- Time to start
- Estimated duration
- Binary flag indicating whether to telnet to a target port on a remote machine, or start a local shell
- Real IP address of host machine
- Real IP address of target machine
- IP address of virtual host machine
- IP address of virtual target machine
- Target port
- List of prompts

The body of the .ex file also contains the following information for every prompt-response pair which is to be generated:

- Comment field
- Prompt index
- Time to wait
- Time to type
- Command

After scanning the header, the regenerator sequentially issues all the commands included in the prompt-response pairs. For each prompt-response pair, the regenerator first waits for the prompt indicated by the index, waits the indicated amount of time before typing, and then types the command over the time period specified.

This provides a moderate level of control over the timing of a session. If someone steps away from their terminal for an hour and then returns later to type a series of commands, this can be represented by a long "time to wait."

More information could be provided, such as the timing delay between all characters. It would be easy to alter the regenerator program to use this extra data, but it would reduce both the simplicity of the this approach and the readability of the .ex file while not affecting possible artifacts that may be inserted into the data.

The list of prompts are regular expressions. This allows specifying a general description of a prompt even though some details of a user's prompt may change, like the command number or current directory.

Comments can also be inserted into the .ex file. Comments can provide meta-information about the session that can easily be parsed out of the file. For example, comments are inserted to indicate which commands are considered attacks and to indicate which commands generate new TCP/IP sessions.

The commands that the regenerator types in are arbitrary, and they do not need to end in a carriage return. Two regenerators could also be hooked end-to-end to replay both sides of a recorded conversation.

3.5 Regenerating a Single Session

The regenerator program is designed to run one network session. It takes an .ex file as an argument. It opens up this file and reads in the header data. "Time to start" and "duration" are ignored.

When the regenerate program reads the virtual host machine from the .ex file, it uses the IP masquerading kernel to create a socket connected with that IP address. The regenerator then opens a network session to the target computer and port.

The regenerator program enters each command, simulating human typing. If there are more commands left in the list, the regenerator continues the process, pausing between each command. Once all the commands are exhausted, it exits.

The regenerator logs a recording of the session to a file, along with meta-data, such as the actual start time.

Instead of opening a telnet connection to a remote machine on a specific port, the "remote or local" option can be used to start the shell on the regenerator's local host computer. This is useful, for example, in that it allows the use of the computer's local commands for network actions such as mail or ftp, instead of requiring the regenerator or the .ex file to understand the details of each protocol.

3.6 Multiple Sessions

The regeneration scheme was also designed to allow multiple sessions to be run simultaneously, and to provide a graphical user interface to summarize the current status. Another *expect* program, called the master regenerator, can take as an argument a list of filenames. Each of these filenames is a session to be recreated. The master regenerator checks the starting times of each file. When the starting time of a session is reached, a regenerator program is then launched with the correct .ex file as its argument, with the master program monitoring the output of each session. There can be many individual regenerators running at the same time, one per session.

The user interface can report the number of total sessions to be run, how many have begun, and how many have completed. The sessions that completed can be subdivided into those that completed normally, those that reported a serious error, and those that crashed. A session crashing means that the master program lost contact with an individual regenerator. This would indicate that there is a serious bug in the regenerator.

This program worked for testing sample scripts. However, when thousands of scripts were run instances of .ex files that crashed the entire regenerator system were encountered. Because of these problems, the master regenerator was simplified

to create a *perl* script that launches each session at the proper time. Its logging functionality is weaker than that of the more complex program, which maintained three log files: one a summary of the pertinent information for each session, another a log of the errors encountered during regeneration, and the third a verbose record of all state changes that the master regenerator witnesses.

Another issue in scalability involved the exhaustion of pseudo-terminals (pty's). There are a set number of pty's available on a machine, and *expect* uses a pair each time it opens a network connection. "Waiting" for each process completion is essential for efficient operation, but in order to have more than a handful of sessions open simultaneously, it was necessary to add more pseudo-terminals to the system, a typical system administrator activity. The regenerator now checks to make sure a session was able to obtain a pseudo-terminal and launch correctly. If it can't, then the session is skipped, instead of having the entire program crash, which is *expect*'s default behavior.

3.7 Options and Additional Features

To allow for quick testing of a session or a series of sessions, the regenerator or the master regenerator can be run with a speed-up option. This reduces the delays before a session starts up, after seeing the prompt, and between keystrokes by a given factor.

If a particular session requires more complicated interaction than is allowed for by the .ex file format, each command component of a prompt-response pair can specify another *expect* script subroutine. This makes it possible to use some of *expect*'s more useful abilities, such as taking different actions based on what the host machine is sending back in its response. Examples of such scripts are a mail reader, which reacts differently based on how many messages the user has waiting, and an automata to run the UNIX `man` and `more` commands.

Chapter 4

Taxonomy

In addition to traffic generation, work focused on generating computer attacks and developing a taxonomy of attacks. A taxonomy is a method of classifying elements in a hierarchical fashion, such as the taxonomy of life. That taxonomy shows the variety of species that exist and illustrates relationships among similar species.

A taxonomy of computer intrusions classifies attacks. Some previous taxonomies were proposed for the purpose of preventing vulnerabilities. If common faults can be found for each kind of security hole, it may be possible to figure out how to prevent those faults, such as through better checking of code or more careful specification.

The primary purpose for categorizing attacks here is to create good classes from which to draw intrusions for evaluating IDSs. *Equivalence partitioning* categorizes attacks in such a way that an IDS has a roughly equal chance of recognizing each member of a class [20]. This simplifies testing by enabling the IDS to be tested against just a few members of that class. Furthermore, if a new attack is developed and can be placed within an existing equivalence class, it can be determined which IDSs would have found the attack without needing to run that attack against each IDS.

The first section in this chapter discusses the desired features of a taxonomy of computer intrusions or attacks. This is followed by an overview of some previous taxonomies, and a new taxonomy.

In the following discussion, an *intrusion* or *attack* is defined as an instance in which someone gains unauthorized access to a system or disrupts authorized access.

Vulnerabilities or *security faults* are flaws in systems that allow attacks to occur. They can be caused by software errors, by providing features that can be misused, or by poor user validation.

4.1 Goals of a taxonomy

Discussions of desirable characteristics in a taxonomy can be found in several previous works (e.g., [2], [7], and [20]). Features desired in a general taxonomy of computer intrusions include:

- It can be used to guide evaluation of security tools, particularly IDSs.
- It can be used to perform cost-benefit analysis.
- It can help prevent attacks in the future.
- It can be used to explain the cause of an attack.

Features of a taxonomy that are more generic include:

- Each attack should be reliably placed in only one category.
- All possible intrusions should have a place in the taxonomy.
- The taxonomy can be extended in the future.

The relative importance of each category will vary depending upon the intended purpose of the taxonomy. The categories are ordered here in approximate importance for the testing an evaluation of IDSs. Each goal is discussed in turn in the following sections.

4.1.1 It can be used for the evaluation of security tools, particularly IDSs

A useful feature of an attack taxonomy is to allow security tools to be evaluated. If it is possible to say against which kinds of attacks the security tool provides protection,

a site could use this in combination with a cost-benefit analysis to determine the appropriate tools for their situation.

Different partitionings will have different usefulness in testing. If one partitions the attacks by their relative harm, a ranking can be made by how well the IDS finds each attack.

However, among any one class of attacks in that partition, the IDS may find some attacks easily, while missing others entirely. This is the argument behind *equivalence partitioning*, breaking the attacks into classes such that the IDS's chance of finding all intrusions in the class is roughly equivalent. Equivalence classes may be possible to construct for an individual IDS which uses a restricted set of inputs such as BSM audit logs from one host. They may not, however, be possible to construct for multiple IDSs which use different input factors. The goal of creating equivalence classes can only be partially met.

4.1.2 It can be used to perform cost-benefit analysis.

Different groups will face different risks from computer intrusions. An ISP that prides itself on reliability, for example, may find an attack that leaves its customers unable to log in worse than one which allows unauthorized users to have user-level permissions, while a medical database might place the highest priorities on integrity and privacy.

Different groups should be able to use the taxonomy to perform cost-benefit analysis by placing cost values on each subcategory of attack. These costs would estimate the damage that would be done if someone actually performed an attack. Steps could then be taken to isolate the most dangerous attacks for a given site and to select an IDS which is most appropriate and finds the most costly attacks.

4.1.3 It can help prevent attacks in the future.

Many of the previous taxonomies were created in the hope of reducing or eliminating vulnerabilities in the future. By categorizing the different security holes that existed in software, effort could be applied towards fixing the most frequent and dangerous

holes.

4.1.4 It can be used to explain the cause of an attack.

If a service on a machine has suddenly stopped working, a taxonomy of attacks may be of use in determining whether the cause of the denial is due to malicious activity. Even if the exact symptom cannot be found in the taxonomy, one could look nearby for similar intrusions, and inspect one's system for evidence of those attacks.

4.1.5 Each attack should be reliably placed in only one category.

Each intrusion should fit in no more than one category. This has proven to be a difficult requirement. Landwehr [14] pointed out that "often ...such a partitioning is infeasible." Bishop and Bailey's main criticism of other taxonomies [7] was that "they fail to define classification schemes that identify a unique category for each vulnerability."

Also, it should be simple to determine the placement of each element in the taxonomy. Under Aslam's taxonomy [4], for example, even though some faults could correctly be described by more than one location in the chart, a decision tree uniquely determines the classification of each fault.

4.1.6 All possible intrusions should have a place in the taxonomy.

The taxonomy should be complete. It should be possible to categorize all currently known as well as proposed future attacks.

4.1.7 It can be extended in the future.

It is difficult to determine what sort of attacks will be developed in the future. Many new attacks in one category may occur, and the taxonomy would classify them iden-

tically. It should be possible to add new sub-categories or new branches to the taxonomy, without affecting the correctness of previously existing entries.

4.2 Previous work

In 1975, Linde presented generic lists of "functional flaws" (such as authentication, implicit trust, and residue) and operating system attacks (including clandestine code, masquerade, and wire tapping) [16]. Most efforts after that moved towards creating hierarchy in the descriptions of flaws and attacks.

4.2.1 RISOS report, April 1976

The RISOS project's report [1] served as an early overview of issues in computer security. It presented concepts of security flaws and discussed methods for enhancing the security of operating systems.

The report presented a taxonomy of integrity flaws. The primary focus was a taxonomy of security faults in operating systems, although it recognized that "almost all applications software flaws have direct analogies with operating system flaws."

The report briefly mentioned other useful categories for describing an attack. The categories were chosen so that the following sentence would be an accurate representation of the flaw:

> A [class of user] acquires the potential to compromise the integrity of an installation via a [class of integrity flaw] integrity flaw which, when used, will result in unauthorized access to a [class of resource] resource, which the user exploits through the method of [category of method] to [category of exploitation].

- **Class of user:** Users of applications, service users (authorized administrators), or intruders.
- **Class of integrity flaw:** The gap in security may be in physical protection, personnel (sabotage and user error), procedural (including "social engineering"

and Trojan horses), hardware (such as terminal hangups not happening properly), application software, or operating system flaws.

- **Class of resource:** "What does the intruder get?" Possibilities are information, services, and equipment.
- **Category of method:** "What did the intruder do?" Interception, scavenging, preemption, or possession.
- **Category of exploitation:** By his or her actions, the user can deny possession/use, deny exclusive possession/use, or modify the system.

The main part of the taxonomy was seven categories of operating system flaws. These flaws were:

- Incomplete parameter validation. An example is an individual program not checking the lengths of arguments passed to it before storing them in a fixed-size buffer.
- Inconsistent parameter validation. The given example is of two programs that each have valid security mechanisms. One safely writes entries to a permissions file, allowing embedded spaces. The other, which removes entries from the file, does not accept entries with embedded spaces. A user may be unable to revoke privileges after handing them out.
- Implicit sharing of privileged/confidential data. A program could temporarily store sensitive data in a publicly readable location.
- Asynchronous validation/inadequate serialization, such as race conditions.
- Inadequate identification/authorization/authentication. Examples are naming collisions, and programs with "wizard modes" that allow trusted-user access to the system.
- Violable prohibition/limit. These occur when a user can operate outside set bounds, such as a limit on the number of processes or a disk quota.

- Exploitable logic error, which includes bugs in programs. A listed example is of a user who can interrupt a failed password check before the operating system logs it, allowing unlimited attempts to crack passwords. Or, a user error causes a system program to leave the system in an unprotected state.

This taxonomy is not exclusive. For example, on some UNIX systems, when a user accesses a setuid shell script through a symbolic link, there is a window of time when the user can substitute his or her own file, after the system has checked the validity of the original file but before the it has run it, causing the user's file to run with the permissions of the shell script. This would be both "inadequate identification" as well as "asynchronous validation." It might also be classified as an "exploitable logic error."

4.2.2 The PA report, May 1978

The Protection Analysis report from the Information Sciences Institute [5] was an attempt to classify errors in applications and operating systems. It discussed errors from a low-level, theoretical point of view, and sought to create generalized error patterns by which actual software errors could be recognized. One of the purposes of the research was to allow programs to be statically verified so that security flaws could be located and removed.

Ten categories of errors were created, but they "seemed to manifest themselves at differing levels of abstraction." The classification scheme was refined, and a set of four global error categories resulted: domain errors, validation errors, naming errors, and serialization errors.

Domain errors result from incomplete separation of information. Information may be in the wrong domain, or the gateway between domains may be insufficiently checked. Examples include the operating system not clearing a memory block of residual data before assigning it to someone else.

Validation errors include insufficient validation of operands and failure to check boundary conditions.

Naming errors, such as a new object taking the identity–and permissions–of a deleted object

Serialization errors, such as race conditions, result when events incorrectly occur simultaneously, or atomic events are interrupted.

This taxonomy suffers from the same problem as that of the RISOS project, in that one event could be placed in two categories. In fact, [6] shows how Peter Neumann's presentation of the PA project [19] and the RISOS project can each be mapped to the other.

4.2.3 Landwehr, September 1994

Landwehr, Bull, McDermott, and Choi presented taxonomies of flaws in operating systems in order to determine how flaws were being introduced [14]. There were three separate taxonomies with which each flaw would be categorized: by genesis, by time of introduction, and by location.

- The **genesis** taxonomy describes how the flaw was introduced. The top-level categories were maliciously intentional, non-maliciously intentional, and inadvertent. The inadvertent flaws were further categorized in a manner similar to the PA and RISOS taxonomies.

- The **time of introduction** of a flaw could be during development (which includes specification, source code, and object code), during maintenance, or during operation.

- The **location** of the flaw could be in the operating system, in support programs, in individual applications, or in the hardware.

4.2.4 Bishop's Taxonomy, May 1995

Bishop's taxonomy [6] focused on UNIX system and network *vulnerabilities*, as opposed to *attacks*, according to the belief that "a model should highlight the underlying

vulnerability, and not its exploitation." He looks at the previous taxonomies for useful features in his goal of creating a taxonomy that describes vulnerabilities "in a form useful to intrusion detection mechanisms," by which be means both IDSs as well as tools that statically check a system for existing vulnerabilities, such as COPS [9].

There are six axes to Bishop's taxonomy:

- The **nature of the flaw**. Neumann's version of the PA categorization of errors [19] is used:
  - Improper Choice of Initial Protection Domain
  - Improper Isolation of Implementation Detail
  - Improper Change
  - Improper Naming
  - Improper Deallocation or Deletion
  - Improper Validation
  - Improper Indivisibility
  - Improper Sequencing
  - Improper Choice of Operand or Operation
- The **time of introduction**. This was taken to be the same as Landwehr's *time of introduction* taxonomy, modified slightly so that the distinctions between classes was more clear.
- The **exploitation domain**: who can take advantage of the vulnerability, and where the attacker needs to be relative to the system to do so
- The **effect domain**: what can be affected by a use of the vulnerability, such as network sessions or hardware
- The **minimum number of components** needed for the vulnerability to be exploited. This indicates how many programs must be audited by an IDS to detect the exploitation.

- The **source** that caused the vulnerability to become known, such as USENET or academic literature.

4.2.5 Aslam's Taxonomy, August 1995

Aslam [4] presented a taxonomy of security faults in UNIX with the intent to "unambiguously classify security faults into distinct categories." Non-overlapping groupings were required for the construction of a vulnerability database, and he claimed that taxonomies presented by the RISOS report, the PA report, and Landwehr were unsuitable in this regard, as they are "too generic, and do not clearly specify the criterion used for the classification."

Each security fault was categorized into one of three general groups: environment faults, operational faults, and coding faults.

Environment faults include "limitation of the operational environment" and "interaction errors between functionally correct modules."

Operational faults are configuration errors, such as programs installed with improper permissions or in the wrong place.

Coding faults come in two varieties: *synchronization errors* (race conditions and serialization errors) and condition validation errors (failure to properly validate inputs, boundaries, permissions, or handle exceptions)

Aslam also overviewed some software fault detection techniques, and how they could be applied against each of these categories.

A very useful feature that was included was a decision tree for the taxonomy. This aids in the easy cataloging of security faults, and helps provide for a unique classification.

4.2.6 Howard's Taxonomy of Computer and Network Attacks, April 1997

Howard created a taxonomy of computer and network attacks as part of his analysis of security incidents reported to CERT from 1989 to 1995 [13], an *incident* being a series of attacks over time related to the same attacker. He analyzed many categorizations in the literature in his creation of a taxonomy. His taxonomy broke an attack into five fields that had to be "linked" for an intrusion to succeed: attackers, tools, access, results, and objectives.

Attackers Howard presents six types of intruders:

- hackers, who break in to a site for the challenge
- spies, looking for information that will lead to political gain
- terrorists, seeking to cause fear for political gain
- corporate raiders, seeking information from competitors that can lead to financial gain
- professional criminals, seeking personal financial gain
- vandals, who are just out to cause damage

Tools The intruder must use at least one of these in the attack:

- user commands
- a script or program
- autonomous agents, such as a worm or virus
- toolkits, a combination of the previous three tools
- distributed tools, scattered across multiple hosts of the Internet
- a data tap, physically monitoring the electromagnetic emissions of a system

Access The attackers gain access through a *vulnerability* in one of implementation, design, or configuration. They then either make *unauthorized access to* or have *unauthorized use of* system processes, possibly to *files* or *data in transit.*

Results The results of the attack are chosen from

- corruption of information
- disclosure of information
- theft of services
- denial of service

Objectives The objectives of the attack connect back to the attackers in the first field. Possible objectives are

- challenge and status
- political gain
- financial gain
- damage

Any successful attack would need to link at least one path across all these fields, so computer security counter-measures could be applied against any stage.

4.3 A New Taxonomy

A new taxonomy was created for the purpose of testing and evaluating IDSs. One of Bishop and Bailey's criticisms of vulnerability taxonomies was that their point of view was not clearly defined [6]. By taking three different points of view (e.g., from the flawed process that is exploited, from the attacker process, and from the operating system service routines) Bishop and Bailey claimed to get multiple categorizations for a particular attack. Such a criticism is avoided here by taking one clear point of view. It is that of a potential intruder, and his or her level of privilege relative to the target system.

Each attack is categorized as one of the following:

- A user does some action at one level of privilege.

- A user makes an unauthorized transition from a lower privilege level to a higher privilege level.
- A user stays at the same privilege level, but does some action at a higher level of privilege.

This taxonomy requires a way of describing privilege levels, a way of describing transitions, and a way of categorizing actions. These three items are presented in the following sections. A UNIX-like operating system is assumed for the examples, but a similar taxonomy should work with other operating systems as well. Also, although this taxonomy was designed with methods of electronic security in mind, it should be easily adaptable for other potential methods of attack, such as hardware vandalism or social engineering.

4.3.1 Privilege levels

The taxonomy first requires an approach to rank levels of user privileges. A sample ranking of privilege levels is:

O	No access
R	Remote network
L	Local network
M	Modem access
U	User access
S	Root/superuser status

"No access" means the user has no practical access to a system, such as it being in a locked building on a separate network. "Remote network" access refers to having, via other networks, minimal network access to a system. "Local network" means having the ability to read and write to the local network that the target machine uses. "Modem access" refers to the ability to connect directly to a target computer. "User access" refers to having the ability to run normal user commands, and "superuser access" gives the user total software access to the system.

Note that the privilege levels need not be strictly ordered: for example, modem access to a computer is not necessarily more privilege than being able to sniff its network connection.

This list is not as complete as it could be; for example, a process could have the ability to read any file on the system, but not have the ability to write to any file. This state could be listed between *user access* and *root access*. In addition, physical access to a local console keyboard could provide more privilege than network-based access. Many such states could be listed, but the list is kept brief for simplicity and brevity.

It is difficult to attempt to rank all possible privilege levels that an intruder could have for all circumstances. However, an individual site may have a more specific idea of how its systems are configured, so it could create a more detailed and appropriate privilege level structure. For example, having access to an external network could be ranked lower than having access to an internally firewalled network, or having access to machines storing sensitive data could be ranked higher than having access to a static webserver.

Similar structures to this were seen in some other taxonomies, such as the *class of user* in RISOS and the *exploitation domain* in Bishop's taxonomy. This structure is more general and has finer granularity.

4.3.2 Actions

The taxonomy includes the following five actions:

1. Probing: Gathering data about a system. More specific categories of information to be collected are:

   Probe(Users) Users on a machine (and information about these users)

   Probe(Services) Services on a machine

   Probe(Machines) Machines on a network

2. Denial of Service: Hindering legitimate access to the system. Includes degradation of service.

   Deny(Temporary) Temporary denial with automatic recovery

   Deny(Administrative) Denial requiring administrator action to recover

   Deny(Permanent) Permanent denial

3. Interception/Reading of Data:

   Intercept(Files) Files on a system

   Intercept(Network) Network traffic

4. Alteration/Creation of Data

   Alter(Data) Alter stored data

   Alter(Intrusion-Traces) Removing intrusion traces, such as log files

5. Attacker uses system:

   Use(Recreational) The intruder is using the system for enjoyment purposes, such as chatting on IRC to brag.

   Use(Productivity) Using the system for productivity-related purposes, such as using the editor or compiling, excluding intrusion-related uses.

   Use(Intrusion-Related) The intruder is using system resources to help break into other sites, such as using system cycles to crack passwords. Excludes staging of attacks.

   Use(Staging-Attacks) The intruder is using the computer as a stepping-stone to launch attacks on another system or systems.

The RISOS project had the concept of actions, but they were spread over multiple categories (*class of resource*, *category of method*, and *category of exploitation*). Bishop's taxonomy also had the *effect domain*, although this was limited more to what was affected than what the attacker accomplished.

Howard's *attack* category contained descriptions similar to this one. His "corruption of data" category is revised here to include both the creation and alteration of data, and the new category of "probing" is added.

4.3.3 Methods of transition

The user needs to exploit some failure of the security system to perform an attack. Three categories of failure are:

m) Masquerading Misrepresenting oneself. An example could be using a stolen username-password pair, or sending a TCP packet to a machine with the source address forged.

a) Abuse of feature There are legitimate actions that one can perform, and is expected to perform, that when taken to an extreme can lead to failure. Examples are filling up a disk partition with user files or a mail spool with junk mail, or attempting to connect to every port on a machine to determine what vulnerable services it is running.

b) Implementation bug A bug in a trusted program that allows an attack to proceed. Specific examples could be buffer overflows or race conditions.

Many of the previous taxonomies had different implementations of this category. The RISOS project and PA report were primarily concerned with how the security holes were introduced, and Bishop's taxonomy built on those. A parallel could be drawn between Aslam's categories of "coding fault" and "environment fault" and this taxonomy's "implementation bug" and "abuse of feature." The Howard taxonomy has access being provided through a vulnerability either in implementation, design, or configuration.

An individual attack may use more than one of the approaches. For example, any computer can send messages to a machine's `syslogd` daemon, and also forge the packets to come from arbitrary IP addresses. Some versions of `syslogd` will crash if they fail to perform a reverse-DNS-lookup on this address. This exploits both

masquerading and buggy implementations, and it's possible to detect the intrusion via either of these paths.

4.3.4 Representing an Action at a Privilege Level

Every attack category in the taxonomy is represented by a short alphanumeric string. For attacks which simply perform actions at a higher privilege level, this string indicates the user's initial privilege level, followed by the type of security hole exploited (if any), followed by the action performed.

The substrings in this description are the bold strings from the previous three sections. The initial privilege level is indicated by **O**, **R**, **L**, **M**, **U**, or **S** as defined in section 4.3.1, the method of exploitation by **m**, **a**, or **b** as defined in section 4.3.3, and the action is indicated by one of the many strings in 4.3.2.

The following table illustrates strings for three attacks. The SYN flood attack, for example, sends a repeated stream of SYN packets to a port on a target machine. For a short duration after these packets have been sent, no other users can connect to that port on that machine, denying them service.

Examples:

Attack	*String*	*Description*
SYN flood	R-a-Deny(Temporary)	User needs network access to enact a temporary denial of service
sniffing passwords	L-a-Intercept(Network)	User with access to a computer's network reads plaintext passwords
cracking passwords	U-Use(Intrusion)	From a user account, someone uses system cycles to crack passwords

4.3.5 Representing Transitions

To show a transition between two privilege levels, the two privilege levels are written adjacent to each other with the method of transition between them. Two examples of transitions, from user (**U**) to superuser (**S**) and from remote user (**R**) to local user

(U) are shown in the following table.

Examples:

Attack	*String*	*Description*
eject	U-b-S	User exploits bug in eject program to become root
password guessing	R-a-U	User with network access repeatedly guesses passwords

4.3.6 Representing Actions at a Higher Privilege Level

Users may also be able to perform actions with a higher privilege level than they currently possess, but they may not be able to keep those enhanced privileges. These cases are represented as illegal transitions, except the action that is performed is written, following a colon. The following example is an attack where a user creates new files on a UNIX system.

Example:

Attack	*String*	*Description*
ftp-write	U-b-S:Alter(Data)	A bug in some ftp daemons allows a user to create any file on the system that did not exist previously, such as putting a new file in a database. This does not necessarily translate to a root compromise.

4.3.7 Multiple users

As is, the taxonomy treats all user accounts as equivalent. However, this is not always the case. Some users may have less than normal privileges, some users control specific services, and individual users have access to their own files. The following notation allows different types of users to be described.

Users:

Plaintext Usertype	*Formatted Usertype*	*Description*
U' U''	U′ U″	Generic Users
(U-1)	U_{-1}	User "nobody"
(UG)	U_G	Guest
(US)	U_S	System-level Users
(U1615)	U_{1615}	Specific Users

Formatting is suggested for both plaintext and formatted output.

Generic users are used when an attack involves multiple users, such as one user gaining the ability to read another user's files (see example below). User "nobody" refers to the account with minimal privileges on many UNIX systems. "System-level users" generically refers to accounts like "bin" and "daemon," accounts that have special privileges over important parts of the operating system. Specific users are useful for tracing an incident (see the next section).

The following example shows an attacker who wishes to read another user's mail. He creates a program which alters the permissions of the mail directory of any person who executes it.

Examples:

Attack	*String*	*Description*
trojan horse	U'-U":Intercept(Files)	One user asks a second user to run a given program. When the second user does so, her mail directory is made world-readable, allowing the first user to read the files.

4.3.8 Tracing of a session

This taxonomy allows a trace to be made of an incident that is composed of many steps, like the record of a chess match. This is useful, for example, in performing post-mortems of an incident.

Here is an example trace of an attack in which a user breaks into a computer that is running a buggy web-server, and proceeds to acquire more and more privileges until achieving root.

Notation	*English*	*Notes*
R-b-U_{-1}:Intercept(Data)	phf, ypcat passwd	Webserver running as user nobody has a cgi-script allowing the encrypted password file to be read
R-m-U	use cracked password	Off-line, the intruder has figured out a password
U-b-S	use buffer overflow	Intruder goes from user to root via a bug in the `eject` program

4.3.9 Multiple machines

For a given attack or incident, there may be more than one machine from which it is desirable to have a point of view. For example, an intruder may be simultaneously attacking two important systems, or she may be jumping across multiple machines in a network during a trace.

Each machine is represented by the letter "H" followed by a number or letter indicating the machine. Privilege levels are now indicated with respect to a given machine, by writing the machine and the privilege level separated by a dot.

Here is a brief example of IP-spoofing involving three machines. H_A, H_1, and H_2 are on the same network. H_A has been totally compromised by an attacker, and she now wants to attack H_1, which trusts H_2. If a user is logged in to H_2, he can automatically log in to H_1 without needing a password.

The attacker is going to make H_A pretend to be H_2. However, H_1 will try to talk with H_2, which will respond by saying that it never initiated a connection. The attacker needs to stop H_2 from responding. She will accomplish this by sending H_2 a "ping of death" to re-boot it, giving her time to complete her transactions with H_1.

Notation	*English*	*Notes*
H_2.R-b-Deny(Temporary)	ping of death	Attacker prevents H_2 from responding
H_A.S-m-H_1.U	IP spoofing	From the superuser account on machine A, the attacker connects to H_1 as a normal user.

4.4 Evaluation of the taxonomy

4.4.1 Each attack should be reliably placed in only one category.

This taxonomy fails to strictly meet this requirement. Instead, an attack is placed in a category for each distinctive feature it contains. For example, if an attacker breaks in and then both denies users access to a service and launches attacks against other systems, the attack could be described under multiple categories. This is acceptable, since IDSs might detect the attack because of either effect.

4.4.2 All possible intrusions should have a place in the taxonomy.

No potential attacks should be left out. All the attacks in Chapter Five can be described with this scheme. This is a necessary but not sufficient step. It remains to be seen if attacks will occur which cannot be contained in this fashion.

4.4.3 It can be extended in the future.

New sub-categories can be added to the list of possible actions with ease. In the future, for example, if attacks occur which temporarily deny service, some to the file system, some to CPU cycles, a further sub-category could be added to distinguish between the two groups. Similarly, new privilege levels can be added where more detail is needed.

4.4.4 It can be used to perform cost-benefit analysis.

The costs of each attack can be estimated on a per-site basis. To our knowledge, no site has used this taxonomy, or another taxonomy of attacks, for such a purpose.

4.4.5 It can be used to guide evaluation of security tools, particularly IDSs.

Plans to test IDSs are currently underway. It remains to be seen how well this taxonomy will serve in partitioning the attacks into equivalence classes.

4.4.6 It can help prevent attacks in the future.

This taxonomy does not provide any special mechanisms for helping eliminate security holes. However, it can be used with a cost-benefit analysis to suggest easy ways that security may be improved, by seeing if attacks with a high-cost can be changed by administrator actions into similar attacks with a low-cost. For example, a user breaking in and erasing all user files is a permanent denial of service and would likely have a high cost. But enacting a regular backup strategy can reduce the costs of such an attack to the levels of a denial that only requires administrator intervention to fix.

4.4.7 It can be used to explain the cause of an attack.

This taxonomy is useful in diagnosing some attacks. If one notices many temporary denial-of-services attacks one a network, searching through this taxonomy for attacks that meet that requirement can give an idea of the vulnerability being exploited.

4.5 Quick Start

Figure 4-1 gives a quick overview of the approach this taxonomy uses to describe a simple attack.

First, select the privilege level that the user had when the attack occurred, from

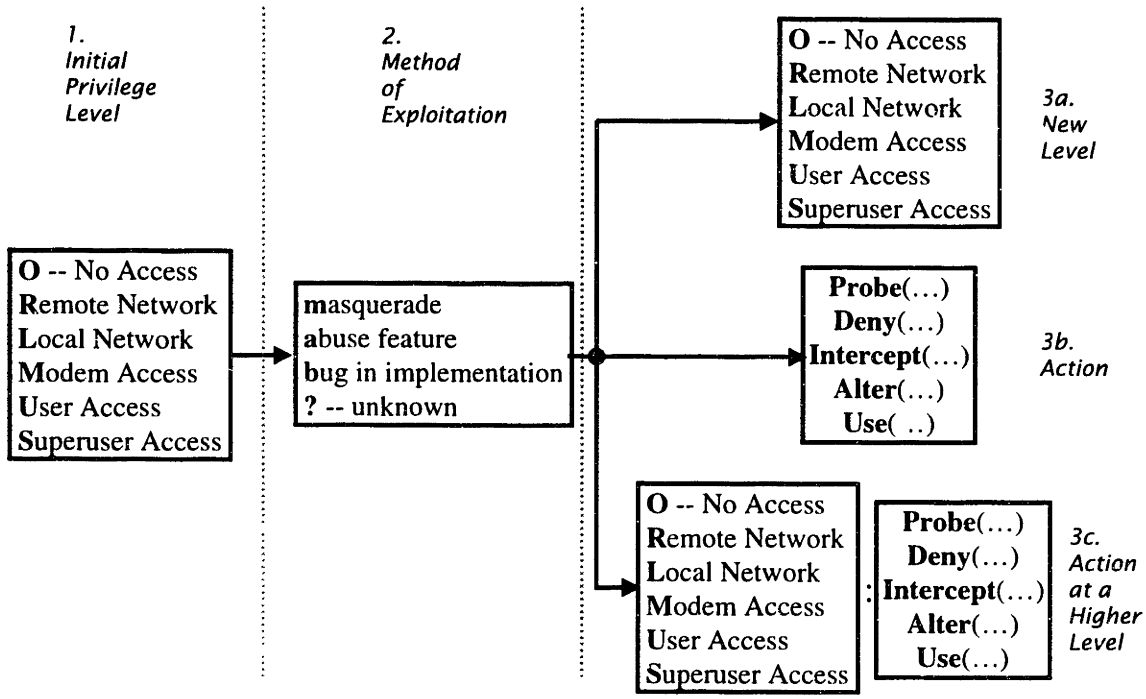

Figure 4-1: A Summary of the Possible Attack Descriptions

"no access" up to "superuser access." If possible, determine the minimum level of privilege that was necessary.

Second, select the method of exploitation, if known. The possibilities are **m***asquerading*, **a***buse of feature*, and *implementation* **b***ug*. Indicate an unknown method with a question mark.

Finally, indicate one of the following three:

- to what new privilege level the user transitioned
- what actions the user performed at the current level of permission, chosen from probing, denial, interception, creation/alteration, or use
- what actions the user performed at a higher permission level, using a concatenation of both a privilege level and an action

4.6 Summary

The following table summarizes descriptions of some common attacks for reference.

Attack	*String*	*Description*
SYN flood	R-a-Deny(Temporary)	User needs network access to enact a temporary denial of service
sniffing passwords	L-a-Intercept(Network)	User with access to a computer's network reads plaintext passwords
cracking passwords	U-Use(Intrusion)	From a user account, someone uses system cycles to crack passwords
eject	U-b-S	User exploits bug in eject program to become root
password guessing	R-a-U	User with network access repeatedly guesses passwords
ftp-write	U-b-S:Alter(Data)	A bug in some ftp daemons allows a user to create any file on the system that did not exist previously, such as putting a new file in a database. This does not necessarily translate to a root compromise.
trojan horse	U'-U"-Intercept(Files)	One user asks a second user to run a given program. When the second user does so, her mail directory is made world-readable, allowing the first user to read the files.

Chapter 5

Attacks

A large representative sample of actual attacks are needed to test an IDS. These attacks should be varied at least with respect to the vulnerability exploited, stealthiness, and types of actions performed after the break-in. This chapter describes the attacks to be used in the training data of an upcoming planned 1998 DARPA off-line intrusion detection evaluation.

5.1 Sources

Most of the attacks developed for the DARPA evaluation are drawn from publicly available sources on the Internet. Bugtraq, a "full-disclosure" mailing list, frequently hosts "exploit code" when a vulnerability is discussed, ostensibly to be used to test one's own system for that weakness. It frequently took some effort to get an attack to work consistently and reliably on our network. Either the attack needed to be tuned to our systems, or our systems needed to have the proper software installed.

Other attacks were based upon the vulnerabilities that are probed by scanning tools, such as SATAN. Some attacks were created from information about potential security holes that was available from computer security groups, such as CERT.

Some new attacks were also created. These attacks are useful for determining how IDSs work against novel attacks, a difficult class for signature-detection IDSs.

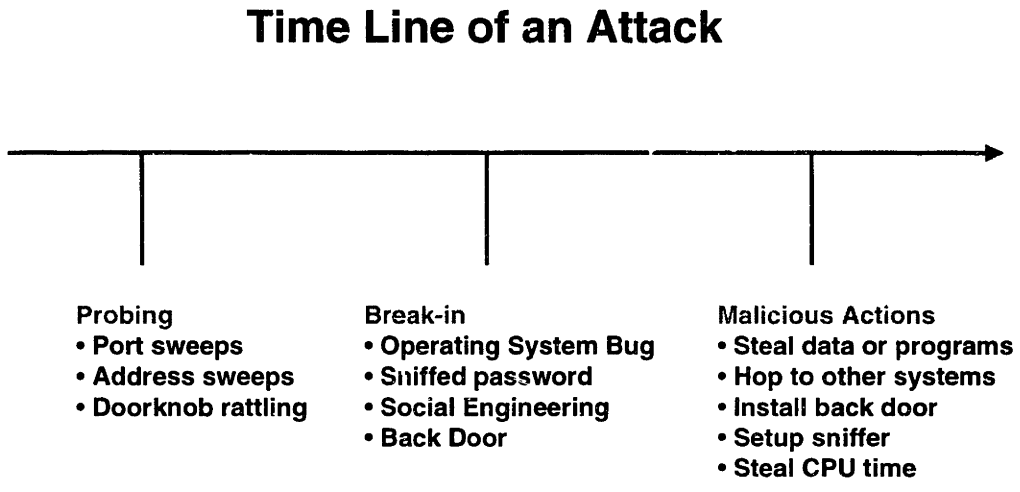

Figure 5-1: Timeline of an Attack

5.2 Timeline of an attack

An attack can be usefully broken up into three stages of time:

1. The time before the attack, where the intruder may be attempting to probe the system for weaknesses.

2. The actual attack.

3. The actions following the attack.

This timeline is useful because different IDSs tend to trigger from events that occur in different parts of the timeline. For example, probing of a network can be unusual, so anomaly-detection systems can trigger on wide-spread probing in stage 1. Since many signature-based IDSs use recorded samples of actual attacks, they may recognize when an attack occurs at stage 2. Other systems may detect when someone is executing (or attempting to execute) dangerous commands or commands normally run by the system administrator. These systems may find stage 3 of the attack, but could also false alarm on many legitimate administrator actions. Of course, systems may be able to find intrusions at more than one stage.

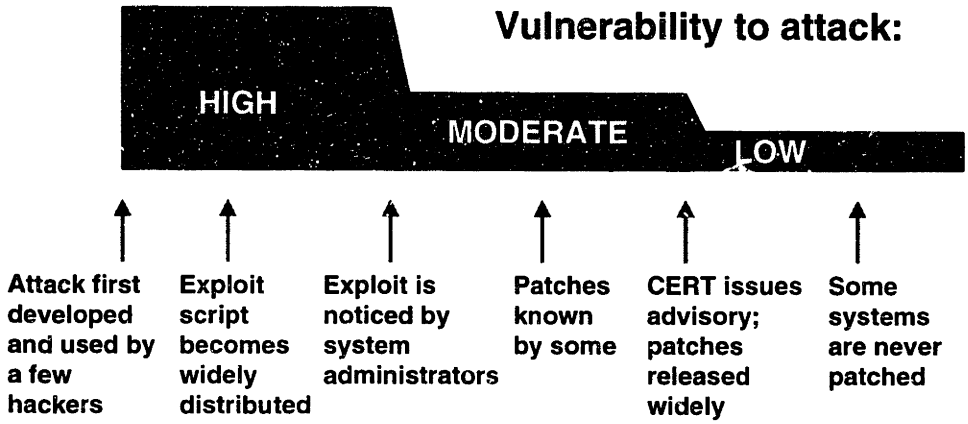

Figure 5-2: An Attack Tends to Become Less Useful Over Time

5.3 Age of an Attack

Each attack has a period of time during which it is most potent. Figure 5-2 shows a simplified example of the use of a supposed attack developed by hackers. As time goes by, more systems become resistant to the intrusion. The length of this time window is quite variable.

Some systems go years without having widely-known security holes fixed. A number of computers in operation today contain these holes, such as guest accounts with default passwords.

Some old attacks, such as trying to overflow the buffer of the `fingerd` daemon, are included in the set of intrusions. It is expected that many IDSs will find these attacks easily.

5.4 Stealthiness of Intruders

Stealthiness is defined as taking steps to evade an intrusion detection system. There are many approaches that an attacker could use to hide from an IDS. For example, if an IDS looks for someone exploiting the loadmodule vulnerability by watching for the string "loadmodule", an intruder could bypass this by setting the csh shell variable

`VAR` to "`module`" and then issuing the command as "`load$VAR`".

Another way of hiding from an IDS involves separating the attacks from the actions that follow it (stages 2 and stage 3 in the timeline). One technique to accomplish this is for the intruder to open two simultaneous connections to the victim machine. After running an exploit to gain enhanced permissions in one session, the hacker actions can be performed in the other session. If the IDS looks at each session independently, it is possible that neither session would look suspicious enough by itself to warrant human inspection. Categories of "SIMT (Single Intruder Multiple Terminal)" and "MIMT (Multiple Intruder Multiple Terminal)" are presented in [20].

5.5 Profiles of intruders

When creating instances of attacks for the simulation, it is useful to have profiles of potential intruders. Howard [13] cited six different types of attackers: hackers, spies, terrorists, corporate raiders, professional criminals, and vandals (see section 4.2.6).

Different intruders into a system will display different kinds of behavior. Some of the simulations of actual attacks should follow such patterns, since the ultimate purpose of an IDS is to detect actual break-ins.

Characteristics that distinguish one attacker from another include the following:

1. Skill at hiding. Some attackers will be unaware of monitoring, or may not understand how to successfully hide. Other attackers may know more about hiding in general, or perhaps the specifics of the particular IDS by which they are being monitored.

2. Source of attacks. Many people attempting to break in to a site have found publicly available scripts on the Internet, and they try them repeatedly until they find a site that is vulnerable. A small fraction of intruders have discovered a vulnerability on their own and are using that knowledge to penetrate systems.

3. Intentions. Some intruders are out to just "collect systems" – to break into as many machines as possible – while some may want to attack a specific ma-

chine. Others could be looking for information that seems to be valuable. For example, they might search for directories that are labelled as "personal" or "confidential."

A multiple-category division of attackers such as Howard's may be useful in creating profiles. However, for simplicity, a two-category approach is used that defines an attacker's *actions* instead of *motivations* as used by Howard. Actions are used as a basis for these categories because they can be easily created in out simulation. The following are two user categories [10]:

Collector Sets out to break-in to as many machines as possible. May install backdoors on a system, and look for other systems to break into. Will not take special actions to hide from an IDS. Working from publicly available scripts. Relatively low-skill.

Spy Looking for important information. Will take steps to minimize the possibility of detection. Knows how IDSs work and will take steps to evade them. Relatively sophisticated.

Twenty Attack Types to be Used in Training
Data For DARPA 1998 Off-line Evaluation

	Solaris Server (audited)	SunOS internal	Linux internal	Cisco Router
Denial of Service	•ping of death •teardrop •neptune •syslogd •back.c •smurf	•ping of death •teardrop •neptune	•ping of death •teardrop •neptune	•land
Remote to User	•dictionary •guest •sniffer •IP spoofing •news •phf	•dictionary •guest •sniffer •IP spoofing	•dictionary •guest •sniffer •IP spoofing •news	
User to Root	•eject •ffbconfig •fdformat	•ps •loadmodule	•perl •ftp-write	
Surveillance/ Probing	•port sweep •ip sweep •iss •finger	•port sweep •ip sweep •iss •finger	•port sweep •ip sweep •iss •finger	•port sweep •ip sweep •iss •finger

Figure 5-3: The Attacks

5.6 The Attacks

Ping of Death **R-b-Deny(Temporary)**

A ping packet of a certain size causes many operating systems to reboot.

CERT Advisory: CA-96.26[1]

Teardrop **R-b-Deny(Temporary)**

A flaw in the IP re-assembly routine on many operating systems can cause the computer to re-boot when it receives certain packets.

CERT Advisory: CA-97.28

Neptune **R-a-Deny(Temporary)**

Floods the target machine with SYN packets. Prevents other users from connecting to the attacked service on the attacked machine.

CERT Advisory: CA-96.21

[1]CERT advisories may be found on the World Wide Web at `http://www.cert.org/`

Syslogd **R-mb-Deny(Administrative)**

A remote user sends a message to the syslogd daemon running on a Solaris computer, with the source address faked. If that new address does not resolve to a DNS name, syslogd crashes.

Back.c **R-b-Deny(Temporary)**

If a request to an Apache web-server contains a large amount n of backslashes, the web-server will take an $O(n^2)$ time to process the request.

Smurf **R-ma-Deny(Temporary)**

By sending broadcast pings to a number of networks, a storm of ping packets many times greater than the amount sent out come back, clogging the network and the computer. By forging the source address, this attack can be directed at a specific target.

CERT Advisory: CA-98.01

Dictionary **R-a-U**

Brute-force trying of many possible passwords against a username

Guest **R-a-U_G**

Trying to log in as the guest user.

Sniffer **L-Intercept(Network)**

User reads plaintext passwords off of an Ethernet.

CERT Advisory: CA-94.01

IP Spoofing **L-m-U**

User pretends to be coming from a machine that the target machine trusts.

CERT Advisory: CA-96.21

News **R-b-U**

Intruder sends a mal-formatted article to the victim machine's news server. A bug in the news server causes the code in the body of the message to be executed.

CERT Advisory: CA-97.08

Phf **R-b-U_{-1}**

A cgi-script contains a bug that lets users who can connect to the webserver run arbitrary code as user "nobody", assuming that's the user running the webserver.

CERT Advisory: CA-96.06

Eject **U-b-S**

A bug in the setuid root program `eject` allows users to run arbitrary code as root.

Ffbconfig **U-b-S**

A bug in the setuid root program `ffbconfig` allows users to run arbitrary code as root.

Fdformat **U-b-S**

A bug in the setuid root program `fdformat` allows users to run arbitrary code as root.

Port Sweep **R-a-Probe(Services)**

Attempt to open network connections to many ports on a computer, to see which services are running.

IP Sweep **R-a-Probe(Machines)**

Attempt connections to a large range of IP addresses, to see which machines respond.

ISS **R-a-Probe(Services)**

Run a commercial scanner that tests machines for common vulnerabilities but does not exploit them.

CERT Advisory: CA-93.14

Finger **R-a-Probe(Users)**

Gather information about which users are on the system and obtain personal information about them.

Land **R-mb-Deny(Administrative)**

If a vulnerable machine receives a SYN packet with the source IP address and port set to be the same as the destination it will lock up.

CERT Advisory: CA-97.28

Ps **U-b-S**

Race condition in `ps` command allows local users to get root access.

CERT Advisory: CA-95.09

Loadmodule **U-b-S**

A bug in the setuid `loadmodule` program allows users to run arbitrary code as root.

CERT Advisory: CA-95.12

Perl **U-b-S**

A flaw in the `perl` program allows users to start shells as root.

CERT Advisory: CA-96.12

Ftp-write **U-b-S:Alter(Data)**

Due to the way that the ftp server dumps core, a user can create any file on the system that did not exist previously.

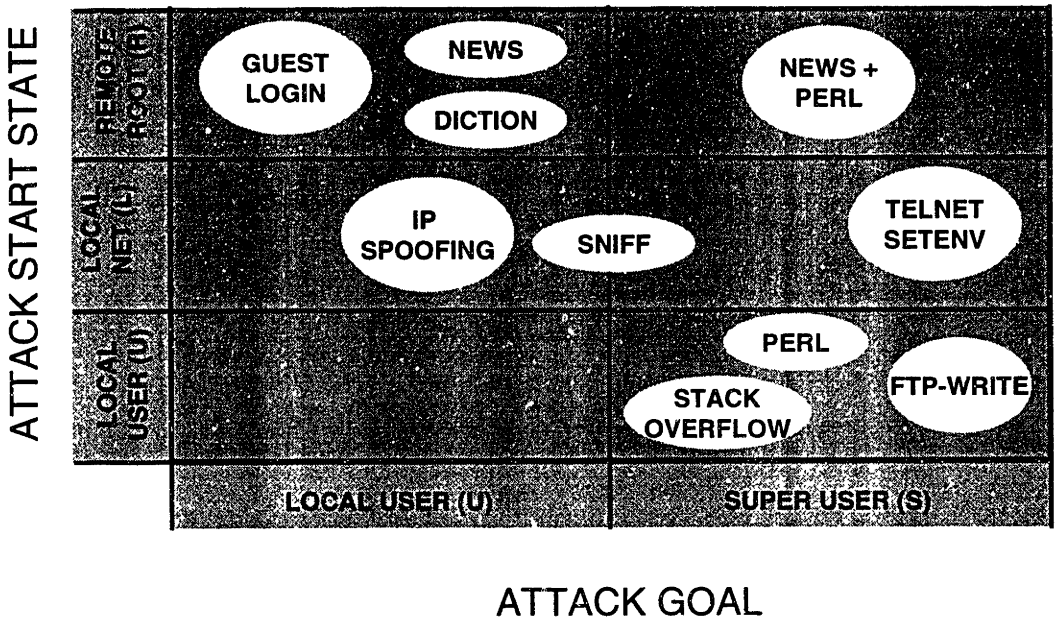

Figure 5-4: Start State vs. End State

5.7 Grouping Attacks Using the Taxonomy

The above attacks can be grouped using the different components of the new taxonomy.

Figure 5-4 groups many of the state-transition attacks by their initial and final privilege levels. More dangerous attacks are those towards the upper right, which need less access and grant more privileges. This figure illustrates how sequences of attacks can be chained. There are a variety of exploits that allow remote users to get the equivalent of local accounts on a machine, and there are many attacks that allow normal users to get root level privileges. The "news + perl" entry points out that simply combining two attacks allows people with nearly no access to a vulnerable system to soon acquire total access.

Action Attacks, Sorted by Method
of Exploitation and State Achieved

	Probe	Deny	Intercept	Alter	Use
Remote Network	•port sweep •IP sweep	•ping of death •teardrop •neptune •syslogd •back.c •smurf •land			
Local Network			•sniffer		
User Access				•ftp-write	

Figure 5-5: Minimum Privilege Levels Needed To Perform Actions

Figure 5-5 shows the initial privilege level and action taken from attacks that undertake actions but do not lead to a change in privilege. This chart shows that hostile actions tend to break down by the access level that is needed to employ them. For example, denial of service attacks can be run with just a minimal amount of access, but seizing data from a system requires closer access, and altering data on a system requires an even higher level of privilege.

	masquerading	abuse of feature	implementation bug
Denial of Service (R-Deny)	•syslogd	•neptune •smurf	•ping of death •teardrop •syslogd •back.c •land
Remote to User (R?U)	•IP spoofing	•dictionary •guest •sniffer	•phf •news
User to Root (U?S)	•stolen password		•eject •ffbconfig •fdformat •ps •loadmodule •perl •ftp-write
Surveillance/ Probing (R-Probe)		•port sweep •ip sweep •iss •finger	

Figure 5-6: Effect vs. Method of Exploitation

Figure 5-6 shows how various privilege transitions and probes are accomplished. For example, for these attacks, all illegal transitions from user to root occur due to implementation bugs or to stolen passwords. This figure also suggests that probing occurs more due to abuse of features than due to improper authentication or bugs.

Chapter 6

Conclusions and Future Work

Chapter One presented the concept of using intrusion detection systems to enhance computer security. Chapter Two discussed methodologies for testing and evaluating IDSs. Chapter Three presented the regeneration scheme that was developed to create large amounts of input data for an IDS. Chapter Four reviewed previous taxonomies and offered a new taxonomy to assist in evaluating IDSs. Chapter Five listed aspects of attacks that should be varied, and showed how the attacks could be categorized by the taxonomy.

6.1 Regenerator

The regenerator is being successfully used and seems to present a good abstraction of interactive network sessions. The software, however, has exhibited problems as it is exposed to more complex situations. Even though it was designed with resistance to total failure as a specific goal, the master regenerator has crashed intermittently, ending all of its child processes and network sessions. The program was simplified and now works well.

Work is continuing on tools written in *perl-expect*, as opposed to the standard *expect* which is built on *TCL*. It is hoped that these tools will prove easier to maintain.

Other avenues of future work include:

- Develop better ways to purify sensitive and personal data out of large sets of

recorded sessions, providing more sets of background data.

- Acquire more samples of actual intruder activity, so that the actions of real hacker activity can be better simulated.

- Create good automatic simulations of many different types of users that can be randomly generated, such that they are not easily distinguishable from actual users.

6.2 Taxonomy

The taxonomy has presented promising ways of organizing attacks. When the evaluation has completed, it will be possible to determine how well it divides attacks into equivalence classes. Future work in this area includes:

- Determine if the methods of exploitation are complete. Would adding an additional category, such as "configuration error," significantly improve the taxonomy?

- Determine if the actions are complete. For example, a way of describing the denied service in addition to the time frame in which it is denied may be desirable.

- Further explore equivalence classes. It may turn out, for instance, that different means of dividing up the attacks are needed for different kinds of IDSs.

Bibliography

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, Lawre· ce Livermore Laboratory, Livermore, CA 94550, April 1976.

[2] Edward G. Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall PTR, Upper Saddle River, NJ, 1994. Referenced in [13].

[3] D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. SAFEGUARD final report: Detecting unusual behavior using the NIDS statistical component. Final technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, December 1995.

[4] Taimur Aslam. A taxonomy of security faults in the UNIX operating system. Master's thesis, Purdue University, West Lafayette, IN 47907, 1995.

[5] Richard Bisbey II and Dennis Hollingworth. Protection analysis: Final report. Technical report, USC/Information Sciences Institute, Marina del Rey, CA 90291, May 1978.

[6] Matt Bishop. A taxonomy of UNIX system and network vulnerabilities. Technical report, University of California at Davis, Department of Computer Science, May 1995.

[7] Matt Bishop and David Bailey. A critical analysis of vulnerability taxonomies. Technical report, University of California at Davis, Department of Computer Science, 1996.

[8] Steve Durst and Terry Champion. The IP-swapping kernel. Unpublished.

[9] D. Farmer and E. H. Spafford. The COPS security checker system. Technical report, Purdue University, West Lafayette, IA 47907, June 1990.

[10] Conversation with Simson Garfinkel, February 1998.

[11] Simson Garfinkel and Eugene Spafford. *Practical UNIX & Internet Security.* O'Reilly & Associates, Inc., Sebastopol, CA 95472, 2nd edition, April 1996.

[12] L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proc. 1990 Symposium of Research In Security and Privacy*, pages 296–304, 1990.

[13] John D. Howard. *An Analysis Of Security Incidents On The Internet, 1989 - 1995.* PhD dissertation, Carnegie Mellon University, 1997.

[14] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[15] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly & Associates, Inc., Sebastopol, CA 95472, 1994.

[16] Richard R. Linde. Operating system penetration. *National Computer Conference*, pages 361–368, 1975.

[17] John Markoff. A new method of internet sabotage is spreading. *New York Times*, September 1996.

[18] Patricia Nealon. Teenager caused dangerous computer breach. *The Boston Globe*, March 1998.

[19] P. G. Neumann. Computer system security evaluation. In *1978 National Computer Conference Proceedings*, pages 1087–1095, June 1978. Referenced in [6].

[20] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. Technical report, University of California, Davis, Department of Computer Science, Davis, CA 95616, September 1995.

[21] S. E. Smaha. Haystack: An intrusion detection system. In *Proc., IEEE Fourth Aerospace Computer Security Applications Conference*, pages 37–44, 1988.

[22] Sun Microsystems, Mountain View, CA 94043. *SunSHIELD Basic Security Module Guide*, 1995. Documentation from the Solaris Operating System.

[23] Seth E. Webster. The development and analysis of intrusion detection algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 02139, 1998.

[24] Nick Wingfield. Activex used as hacking tool. *CNET News.com*, February 1997.

THESIS PROCESSING SLIP

FIXED FIELD: ill. _____ name _____

index _____ biblio _____

▶ COPIES: (Archives) Aero Dewey (Eng) Hum

Lindgren Music Rotch Science

TITLE VARIES: ▶☐ _____

NAME VARIES: ▶☒ Daniel James Weber

IMPRINT: (COPYRIGHT) _____

▶ COLLATION: 68 p.

▶ ADD. DEGREE: B.S. ▶ DEPT.: E.E.

SUPERVISORS: _____

NOTES:

cat'r: date:

page: J49.146

▶ DEPT: E.E.

▶ YEAR: 1998 ▶ DEGREE: M. Eng

▶ NAME: WEBER, Daniel