M.Tech Dissertation

DISTRIBUTED INTRUSION DETECTION

submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

By

Mamata D. Desai
Roll No : 99305903

under the guidance of

Prof. Sridhar Iyer
and
Prof. G. Sivakumar

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai 400076
January 2, 2002

Dissertation Approval Sheet

This is to certify that the dissertation titled, **Distributed Intrusion Detection**, by **Mamata D. Desai** is approved for the award of the degree of **Master of Technology**.

Dr. Sridhar Iyer
(Guide)

Dr. G. Sivakumar
(Co-Guide)

Internal Examiner

External Examiner

Chairman

Date :

Acknowledgments

I would like to thank my guide, **Prof. Sridhar Iyer** and my co-guide, **Prof. G. Sivakumar** for their indispensable ideas and continuous support, encouragement and advice during the completion of this work. I would like to express my gratitude to Prof. Sridhar Iyer for understanding me through my difficult times, and keeping up my enthusiasm, encouraging me, and building up my confidence. A special thank you note goes to my friends and guardian angels, my wingmates on the "third floor" of Hostel 11. These acknowledgments would be incomplete without a mention of my family. Without either of us realizing it, I think my Dad has been one of my greatest role models. My Mommy dearest and her infinite supply of love has kept me going. She has always inspired me to follow my instincts. Thanks to both of them, I've really come a long way.

Mamata Desai
IIT Bombay
December 2001

Abstract

As more and more data goes online, there is a pressing need to secure the dissemination of a large amount of information. Because of the effort required to monitor networks and systems manually, it is not easy to detect attempts at misuse or successful attacks without the help of intelligent Intrusion Detection Systems (IDS).

IDS, much like the security industry, has grown rapidly over the past few years. These tools have become essential security components - as valuable to many organizations as a firewall. However, as in any environment, things change. Networks and crackers are evolving fast, demanding that security tools keep up. Intrusion Detection Systems face several daunting, but exciting challenges in the future and are sure to remain one of our best weapons in the arena of network security.

The modern day Network IDS faces some very challenging problems, like switched environments, increased network traffic, and encryption. Add to that, the performance considerations of an IDS, such as false positives and missed attacks, and the mole hill does become a mountain! The way to go seems to be analysis and data correlation, in which, host IDSs also play an important role. The concept of a management console dedicated to the task of correlating abnormal event notifications, with relevance measures is an emerging one. One can picturize many distributed elements performing specific jobs, each passing the results onto a higher level for correlation and analysis.

In an environment where many machines have similar configurations, a complete portscan on one machine may trigger alarms but slow scans across ports of different machines might go unnoticed and will result in the intruder gaining all the information about the services running on each machine, thus successfully performing a *distributed portscan*.

We focus on detecting a *distributed portscan*, by sniffing packets on the network. Five types of TCP portscans, performed by *nmap* are successfully detected, in scan sweeps of one-to-one, one-to-many, many-to-one and many-to-many hosts. Our approach also manages to detect slow scans which are typically missed by available commercial packages, because of the features that we select to examine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networks have evolved over the years to a point of blissful convenience and flexibility, but within the joyful evolution, blatant breaches of security have surfaced. With the ever-changing world, ever-changing technologies and ever-changing programs coming into existence that make our technical world more exciting and productive, the security world also changes. All networks are vulnerable. New holes, bugs and exploits are found by unscrupulous and unethical individuals to exploit for their own gain. Manufacturers of security products struggle to keep up with fixes, patches, and new releases in an effort to keep up-to-date with the surrounding market.

It takes but one oversight to allow a hacker to invade a company and steal vital competitive information, or even cripple its infrastructure. There is also an asymmetry between the value of the information which may be lost to the hacker - which may be intrinsically small - and the privacy, civil liability, customer liaison and regulatory damage which may be caused to an organization - which may be very great. An organization's greatest weapon against internal and external attacks should be the ability to monitor its networks for unauthorized behavior, which can provide protection, and timely and effective countermeasures in the event of a breach, as well as deterrent against abuse. However, without the proper resources and mandates in place to implement and carry out such a task, the security of an organization's information infrastructure and competitive advantage are at risk.

Everyday, all over the world, computer networks and hosts are being broken into. The level of sophistication of these attacks varies widely; while it is generally believed that most break-ins succeed due to weak passwords, there are still a large number of intrusions that use more advanced techniques to break in. Less is known about the latter types of break-ins, because by their very nature they are much harder to detect. Computer break-ins occur in many ways because systems connected the Internet almost always have certain vulnerabilities. To protect their internal networks, companies install firewalls, powerful defensive software that blocks unauthorized intruders. Nevertheless, determined hackers can usually uncover ways of circumventing a firewall. As attackers become more sophisticated in their initiatives, network managers have realized the significance of employing advanced intrusion detection systems to foil even the stealthiest of attempts effectively.

1.1 Intrusion - An overview

Intrusion may be defined as the potential possibility of a deliberate attempt to

1. access information
2. manipulate information
3. render a system unreliable or unusable

Attacks can be classified on the basis of two criteria, depending on whether or not an attacker is normally authorized to use the computer system, and whether or not a user of the computer system is authorized to use a particular resource in the system. Hence we can broadly divide these into three intrusion classes [Axe00]:

- **External penetrators** - those who are not authorized use of the system
- **Internal penetrators** - those who are authorized use of the system, but are not authorized access to the data, program, or resource accessed
  - *Masqueraders* - who operate under another user's ID and password
  - *Clandestine users* - who evade auditing and access controls
- **Misfeasors** - authorized users of the system and resources accessed, who misuse their privileges

Various attack methods are used by attackers to compromise network security. We can broadly group them into a few categories - (1) Information gathering, (2) Unauthorized access, (3) Disclosure of Information, and (4) Denial of Service. A few example attacks are mentioned below.

- Password Cracking

  If the attacker somehow gains access to the encrypted password file of the users, he can run a password cracker program, which can decrypt passwords. Easily guessable passwords, weak encryption algorithms, export restrictions that prohibit usage of strong cryptography, incorrect usage of strong algorithms, some implementation flaws including backdoors, bugs, etc., make password cracking an easy task. Brute force, dictionary attacks, and rule-based cracking attacks can get past even the most secure encryption algorithms. The attacker thus gains unauthorized access to the system and is masquerading as a valid user.

- Buffer overflow

  On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared `auto` in a routine. Code that does this can cause return from the routine to jump to a random address. Malicious code injected by an attacker could then be executed at this random address. This code is executed under privileges of the owner of the previous process, which, in case of the SMTP daemon, *sendmail*, is the root user.

While the attack requires particularly arcane and detailed knowledge of both assembly language and, in the case of Windows, operating system interface details, once someone has coded an exploit and published it, anyone can use it. The results of these exploits provide interactive command shells on Unix systems and the ability to upload and execute arbitrary programs on Windows systems.

- Network Reconnaissance

  In this case, the attackers use various network discovery tools like *nmap* to generate a complete map of the hosts and services on the target network. Hosts on the network are probed and vital information like open ports, a guess of the operating system running, is collected by the attacker. After knowing the remote operating system and the open ports and services running on the hosts, the attacker can use a variety of attack tools and known vulnerabilities for the system services running on this host.

- Denial of service

  DoS (Denial of Service) attacks cause the loss of access to a resource rather than allow the attacker to gain unauthorized access to the resource. They usually involve overloading a resource such as disk space, network bandwidth, internal tables of memory or input buffers (buffer overflow). The overload causes the host or particular service to become unavailable for legitimate use.

  TCP SYN attack: A sender transmits a flood of phony TCP SYN packets at a rapid rate. The victim destination sends a SYN ACK back to the random source address and adds an entry to the connection queue. Since the SYN ACK is destined for an incorrect or non-existent host, the last part of the "three-way handshake" is never completed and the connection queue fills up rapidly, thus denying essential TCP services (telnet, email, ftp) to legitimate users.

- IP Spoofing

  In this type of attack, the attacker manages to inject custom-made packets into the network, with the source address of a trusted machine. The target machine may release valuable information, which is captured back by the attacker machine. Typically, *rshd* and *rlogind* which use `~/.rhosts` or `/etc/hosts.equiv` files, can lead to a compromised target host by spoofing the IP of the trusted host.

As work environments become more interconnected and exposed, service providers will need increasingly to rely on a wide range of anti-intrusion techniques. Listed here are six approaches.

1. **Prevention** reduces the probability of a successful attack. This includes installing firewalls at your network access periphery, strict forms of authentication, encryption, etc.

2. **Preemption** strikes offensively against likely threat agents prior to an intrusion attempt to lessen the likelihood of a particular intrusion occurring later.

3. **Deterrence** deters the initiation or continuation of an intrusion attempt by increasing the necessary effort for an attack to succeed, increasing the risk associated with the attack, and/or devaluing the perceived gain that would come with success.

4. **Deflection** leads an intruder to believe that he has succeeded in an intrusion attempt, whereas instead he has been attracted or shunted off to where harm is minimized. Setting up "honey pots", decoy systems that appear as vulnerable targets and lure the attacker, is one such attempt.

5. **Countermeasures** actively and autonomously counter an intrusion as it is being attempted. This includes dropping routes which lead to the source machine, launching a counter attack, etc.

6. **Detection** discriminates intrusion attempts and intrusion preparation from normal activity and alerts the authorities. A deflection from normal usage patterns or the occurrence of a known attack signature in audit trails or network data, indicates a positive.

The primary focus of our work is in attempting to detect network reconnaissance attempts, specifically, distributed portscans. When an intruder attempts to break into a system, one of the first things that he does is to check out what all services are running on the machine, and which of these, he can exploit, to gain access. Hence, a scan of the TCP/UDP ports of a system is one of the first things that an attacker performs. Many tools exist, to perform such scans, *nmap* being one of the most popular ones.

Intrusion Detection Systems

Intrusion Detection Systems, or IDSs, have become an important component in the Security officer's toolbox. An IDS installed on a network provides much the same purpose as a burglar alarm system installed in a house. Using various methods, it detects when an intruder/attacker/burglar is present, and subsequently issues some type of warning or alert. IDSs don't fully guarantee security, but when used with a security policy, vulnerability assessments, data encryption, user authentication, access control, and firewalls, they can greatly enhance network safety.

There are many commercial IDS products available in the market. These include: Anzen Flight Jacket (http://www.anzen.com), eNTrax Security Suite (http://www.centraxcorp.com), CyberCop Monitor (http://www.nai.com), NetProwler (http://www.axent.com), Net Ranger (http://http://www.nautidigital.com), RealSecure (http://www.iss.net), etc. Amongst the Open Source products, Snort (http://www.snort.org) takes the cake, followed by Tripwire (http://www.tripwire.com), and the like.

1.2 Our problem

Distributed attacks have recently emerged as one of the most newsworthy, if not the greatest, weaknesses of the Internet. Because of the nature of the attack, it is challenging to trace the attacking machine.

In an academic organization such as ours, many machines have similar configurations. For e.g., all the machines in a lab boot the same operating system, the same kernel, and offer the same network services. A single portscan on a single host generally gets classified as a stray attack, and might not warrant serious action. But stray scans of each machine on this network, will slowly result in the intruder gaining all the information about the services running on each machine, thus successfully performing a *distributed portscan*.

A distributed portscan infact would produce no real alert on a normal network intrusion detection system, and would probably count as a few stray incidents of abnormal packets. The goal at hand was to detect such a distributed portscan. The scan might originate from one machine to many others, or might be performed by many machines, on a single target.

A survey of the currently available IDSs has revealed that there is no freely available IDS yet, that addresses these concerns. Most commercial IDSs are available for the Windows platform, and pack together a lot of other heavyweight "features", which aren't of concern to our setup. The goal was to develop such a system, one that can bridge this 'requirement' gap, developed on Linux, which is the commonly used platform here in IIT Bombay.

1.3 Solution Approach

Our approach was to first study the signatures of five commonly used TCP-based portscans, as sequences of network packets. Depending on the scan type to be detected, certain relevant network packets were analyzed, and the scan signature was identified. These numerous individual scans are then recorded for further analysis, which then identifies the sweeps of the scans as one-to-one, one-to-many, many-to-one and many-to-many hosts. A detailed report is generated as an inference.

1.4 Outline of the report

Chapter 2 briefly describes the types of intrusion detection systems that exist, a few network preliminaries, TCP portscans, and some case studies - tools that we studied. Chapter 3, we describe the problem of distributed portscans, analysis of the scan types, and our approach to solving the problem at a conceptual level. Chapter 4 goes on to describe the implementation details, the data structures and the routines developed. Chapter 5 describes our experimental setup, our data collection and finally the results of our detection. Chapter 6 concludes the report with an analysis of our approach and a look at the future of Intrusion Detection Systems.

Chapter 2

Intrusion Detection Systems

In the present chapter, we examine more closely, the various types of intrusion detection systems that exist, and their distinguishing characteristics. We also attempt to classify them into host-based IDS (HIDS) and network-based IDS (NIDS). A few network preliminaries are then discussed. We then describe the various TCP portscans. This is followed by a brief description of the packages that we studied.

2.1 Anomaly and Misuse detection

Early research uncovered two complementary intrusion detection approaches: *anomaly* detection and *misuse* detection. **Anomaly-based** intrusion detection techniques assume that an intrusion is a deviation from the normal acceptable behavior. The construction of such a detector starts by forming an opinion on what constitutes *normal* for the observed subject (which can be a computer system, a user, network traffic, etc.), and then deciding on what percentage of the activity to flag as *abnormal*, and how to make this particular decision. In other words, anything that deviates a lot from normal, is flagged as intrusive. The drawback of this approach is that the distinction of an intrusion from a legal abnormality depends on the threshold, which needs system-specific tuning. The advantage is that new exploits and unforeseen vulnerabilities can also be detected. Therefore, the intrusion detection system might be complete, i.e., all attacks might be detected, but its accuracy is a difficult issue, as we might get a lot of false alarms.

The other approach, **misuse** detection, also popularly known as **signature-based** detection, decides what an intrusion is, on the basis of knowledge of a model of the intrusive process and what traces it ought to leave in the observed system. Attack signatures and system vulnerabilities are stored in a knowledge base, and any attempt which matches these patterns, is flagged as an intrusion. Thus, any action which is not explicitly recognized as an attack is considered acceptable. However, their completeness depends on the regular update of knowledge about attacks. The maintenance and timely updation of the knowledge-base is time-consuming, although extremely vital here. The detection tool in this case becomes specific to the environment. The advantage of this approach is that the potential for false alarms is very low, and the detailed contextual analysis makes it easy for the security officer to take preventive or corrective action.

The most recent approach seems to be a hybrid of the above two, and are also called **signature inspired**. These form a compound decision in view of a model of both the normal behavior of the system and the intrusive behavior of the intruder. These detectors have - atleast in theory - a much better chance of correctly detecting truly interesting events in the supervised system, since they know both, the patterns of intrusive behavior and can relate them to the normal behavior of the system.

2.2 Host-based and network-based IDS

In the first stage of the project, we had reviewed different types of IDSs, and had come across two important classes of IDSs, host-based IDSs (HIDS) and network-based IDSs (NIDS), depending on the substrate on which they operate.

2.2.1 Host-based Intrusion Detection Systems

Host-based intrusion detection started in the early 1980s before networks were as prevalent, complex and interconnected, as they are today. In this simpler environment, it was a common practice to review audit logs for suspicious activity. Intrusions were sufficiently rare that after-the-fact analysis proved adequate to prevent future attacks.

Today's host-based IDSs remain a powerful tool for understanding previous attacks and determining proper methods to defeat their future application. HIDSs still use audit logs, but they are much more automated, having evolved sophisticated and responsive detection techniques. HIDS typically monitor system, event, and security logs on Windows NT and *syslog* in UNIX environments. When any of these files change, the IDS compares the new log entry with attack signatures to see if there is a match. If so, the system responds with administrator alerts and other calls to action.

HIDSs have grown to include other technologies. One popular method for detecting intrusions checks key system files and executables via checksums at regular intervals for unexpected changes. The timeliness of the response is in direct relation to the the frequency of the polling interval. Finally, some products listen to port activity and alert administrators when specific ports are accessed. This type of detection brings an elementary level of network-based intrusion detection into the host-based environment.

Well-known commercial versions include products from AXENT, Centrax from CyberSafe, RealSecure from ISS and Tripwire. Portsentry is a host-based portscan detector and Hostsentry is a host-based login anomaly detector and response tool, freely available, from Psionic.

Strengths of Host-Based IDSs

1. Verifies success or failure of an attack
2. Monitors specific system activities
3. Detects alerts that network-based systems miss
4. Well-suited for encrypted and switched environments
5. Near-real-time detection and response
6. Requires no additional hardware
7. Lower cost of entry

2.2.2 Network-based Intrusion Detection Systems

A network-based IDS monitors the traffic on its network segment as a data source. This is generally accomplished by placing the network interface card in promiscuous mode to capture all network packets that cross its network segment. These are then picked up and examined by a sensor. Packets are considered to be of interest if they match a signature. Three primary types of signatures are: (1) string signatures, (2) port signatures, and (3) header condition signatures.

String signatures look for a text string that indicates a possible attack. To refine the string signature to reduce the number of false positives, it may be necessary to use a compound string signature. A compound string signature for a common Web server attack might be "cgi-bin" AND "aglimpse" AND "IFS". Port signatures simply watch for connection attempts to well-known, frequently attacked ports. Examples of these ports include telnet (TCP port 23), FTP (TCP port 21/20), SUNRPC (TCP/UDP port 111), and IMAP (TCP port 143). If any of these ports aren't used by the site, then incoming packets to these ports are suspicious. Header signatures watch for dangerous or illogical combinations in packet headers. The most famous example is WinNuke.

Well-known, network-based intrusion detection systems include NetProwler, NetRanger, Shadow, etc. Snort is a very commonly used freely available lightweight NIDS.

Strengths of Network-Based IDSs

1. Lowers cost of ownership
2. Detects attacks that host-based systems miss
3. More difficult for an attacker to "remove" evidence
4. Real-time detection and response
5. Detects unsuccessful attacks and malicious intent
6. Operating system independence

2.2.3 Advantages of monitoring Network Traffic

There exist many different operating system platforms, and hence, host-based systems have only been used on a single operating system at one time. On the other hand, network protocols like TCP/IP, UDP/IP are standard across most major operating system platforms. By using these network standards, the network-based IDS can monitor a heterogeneous set of hosts and operating systems simultaneously.

Second, audit trails are often not available in a timely fashion. Some IDSs are designed to perform their analysis on a separate host, so the audit logs must be transferred from the source host to a different machine for data analysis. Furthermore, the operating system can often delay the writing of audit logs by several minutes. The broadcast nature of a LAN, however, gives the network-based IDS nearly-instant access to all data as soon as this data is transmitted on the network. It is then possible to immediately start the attack detection process.

Third, the audit trails are often vulnerable. In some past incidents, the intruders have turned off audit daemons or modified the audit trail. This action can either prevent the detection of the intrusion, or it can remove the capability to perform accountability and damage control. The network-based IDS, on the other hand, passively listens to the network, and is therefore logically protected from subversion. Since the IDS is invisible to the intruder, it cannot be turned off (assuming it is physically secured), and the data it collects cannot be modified.

Fourth, the collection of audit trails degrades the performance of a machine being monitored. Unless audit trails are being used for accounting purposes, system administrators often turn off auditing. If analysis of these audit logs is also to be performed on the host, added degradation will occur. If the audit logs are transferred across a network or a communication channel to a separate host for analysis, loss of network bandwidth may discourage administrators from using such an IDS. The alternative, namely, a network-based IDS, will not degrade the performance of the hosts being monitored. The monitored hosts are not aware of the IDS, so the effectiveness of the IDS is not dependent on the system administrator's configuration of the monitored hosts.

And, finally, many of the more seriously documented cases of computer intrusions have utilized a network at some point during the intrusion. i.e., the intruder was physically separated from the target. With the continued proliferation of networks and interconnectivity, the use of networks in attacks will only increase. Furthermore, the network itself, being an important component of a computing environment, can be the object of an attack. The IDS can take advantage of the increase of network usage to protect the hosts attached to the networks. It can monitor attacks launched against the network itself, an attack that host-based audit trail analyzers would probably miss.

2.3 Network Preliminaries

Throughout the rest of the report, we assume a TCP/IP internet to work with. In such packet-switched networks, data to be transferred across is divided into small pieces called *packets*, containing a few hundred bytes of data, and identification that enables the network hardware to know how to send it to the specified destination. The software at the receiving computer then reassembles the data and passes it on to the application programs.

Each host on a TCP/IP internet is assigned a unique 32-bit internet address that is used in all communication with that host, called the "IP address". Each address is a pair (*netid, hostid*), where *netid* identifies a network, and *hostid* identifies a host on that network. Routers use the netid portion of an address when deciding where to send a packet. The reader is encouraged to read [Com95] for further details.

2.3.1 An IP datagram

The *Internet Protocol(IP)* is a network layer protocol responsible for best-effort connection less packet delivery across the internet. The service is unreliable because delivery is not guaranteed.

The basic transfer unit of an internet is an *IP datagram*. It is divided into header and data areas. The header contains the source and destination IP addresses and a type field that identifies the content of the datagram. If a datagram is larger in size than the networks transfer unit, it is often broken down into smaller fragments. Flags in the header specify fragmentation details.

Figure 2.1 shows an IP datagram. The contents are:

<table>
<tr><th>0</th><th>4</th><th>8</th><th>16</th><th>19</th><th>24 ... 31</th></tr>
<tr><td>VERS</td><td>HLEN</td><td>SERVICE TYPE</td><td colspan="3">TOTAL LENGTH</td></tr>
<tr><td colspan="3">IDENTIFICATION</td><td>FLAGS</td><td colspan="2">FRAGMENT OFFSET</td></tr>
<tr><td colspan="2">TIME TO LIVE</td><td>PROTOCOL</td><td colspan="3">HEADER CHECKSUM</td></tr>
<tr><td colspan="6">SOURCE IP ADDRESS</td></tr>
<tr><td colspan="6">DESTINATION IP ADDRESS</td></tr>
<tr><td colspan="5">IP OPTIONS (IF ANY)</td><td>PADDING</td></tr>
<tr><td colspan="6">DATA</td></tr>
<tr><td colspan="6">...</td></tr>
</table>

Figure 2.1: The format of an IP datagram.

- *VERS* contains the 4-bit version of the IP protocol.
- *HLEN*, also 4-bits, gives the datagram header length measured in 32-bit words. All fields, except *IP OPTIONS* and *PADDING* have fixed length.
- *TOTAL LENGTH* field gives the 16-bit length of the IP datagram measured in octets, including octets in the header and data.
- *SERVICE TYPE*, also called *Type Of Service (ToS)*, is an 8-bit field that specifies how the datagram should be handled and is broken down into 5 subfields:
  - Three *PRECEDENCE* bits specify datagram precedence, from 0 (normal) through 7 (network control), allowing senders to indicate the importance of each datagram.
  - Bits *D*, *T*, and *R* specify the type of transport the datagram desires. When set, the *D* bit requests low delay, the *T* bit requests high throughput, and the *R* bit requests high reliability.
- *IDENTIFICATION* contains a 16-bit unique integer that identifies the datagram, and is copied into all its fragments.
- *FRAGMENT OFFSET* is a 13-bit field that specifies the offset of the data in the original datagram, measured in units of 8 octets, starting at offset zero.
- *FLAGS* contains 3-bit flags that control fragmentation. The lower-order 2 bits control fragmentation and the first control bit aids in testing whether the datagram may be fragmented.
- *TIME TO LIVE(TTL)* field specifies how long, in seconds, the datagram is allowed to remain in the internet system. A datagram with a zero TTL is discarded by the router and an error message sent back to the source.
- *PROTOCOL* specifies the format of the DATA area - TCP, ICMP, UDP, etc.
- *HEADER CHECKSUM* ensures integrity of header values. It is formed by treating the header as a sequence of 16-bit integers, adding them together using one's compliment

arithmetic, and then taking the one's compliment of the result. It only applies to values in the IP header and not to the data.

- *SOURCE IP ADDRESS and DESTINATION IP ADDRESS* contain the 32-bit IP addresses of the datagram's sender and intended recipient.

- *DATA* shows the beginning of the data area of the datagram.

2.3.2 A TCP segment

The *Transmission Control Protocol(TCP)* is a transport layer protocol that works on top of IP, providing a reliable transport service. The interface between application programs and the TCP/IP reliable delivery service can be characterized by 5 features: (1) Stream orientation, (2) Virtual Circuit Connection, (3) Buffered Transfer, (4) Unstructured Stream, and (5) Full Duplex Connection. TCP uses a technique known as "positive acknowledgement with retransmission" for reliable delivery.

TCP allows multiple application programs on a given machine to communicate concurrently, and it demultiplexes incoming TCP traffic among application programs. Each machine contains a set of abstract destination points called *protocol ports*, identified by a positive integer, and the protocol. TCP uses these protocol ports to identify the ultimate destination of a segment of data, to be passed on to its application program.

TCP actually uses the *connection abstraction*, in which the objects to be identified are virtual circuit connections, not individual protocol ports; connections are identified by a pair of endpoints. An *endpoint* is a pair of integers (*host, port*) where *host* is the IP address for a host and *port* is a TCP port on that host. Thus an example connection might be: (192.168.111.100, 2345) and (144.16.111.14, 25)

The unit of transfer between the TCP software on two machines is called a *segment*. Segments are exchanged to establish connections, to transfer data, to send acknowledgements, to advertise window sizes (for flow control), and to close connections. Because TCP uses piggybacking, an acknowledgement traveling from machine A to machine B may travel in the same segment as data traveling from machine A to machine B, even though the acknowledgement refers to data sent from B to A.

Each TCP segment is divided into two parts, a header followed by data. The header, known as the TCP *header*, carries the expected identification and control information.

Figure 2.2 shows a TCP segment. The contents are:

- *SOURCE PORT and DESTINATION PORT* contain TCP port numbers that identify the application programs at the ends of the connection.

- *SEQUENCE NUMBER* field identifies the position in the sender's byte stream of the data in the segment.

- *ACKNOWLEDGEMENT NUMBER* field identifies the number of the octet that the source expects to receive next.

- *HLEN* field contains an integer that specifies the length of the segment header measured in 32-bit multiples. It is needed because the *OPTIONS* field varies in length, depending on which options have been included.

- *RESERVED* is a 6-bit field marked for future use.

0 4 10 16 24 31

<table>
<tr><td colspan="3">SOURCE PORT</td><td colspan="2">DESTINATION PORT</td></tr>
<tr><td colspan="5">SEQUENCE NUMBER</td></tr>
<tr><td colspan="5">ACKNOWLEDGEMENT NUMBER</td></tr>
<tr><td>HLEN</td><td>RESERVED</td><td>CODE BITS</td><td colspan="2">WINDOW</td></tr>
<tr><td colspan="3">CHECKSUM</td><td colspan="2">URGENT POINTER</td></tr>
<tr><td colspan="4">OPTIONS (IF ANY)</td><td>PADDING</td></tr>
<tr><td colspan="5">DATA</td></tr>
<tr><td colspan="5">...</td></tr>
</table>

Figure 2.2: The format of a TCP segment.

- *CODE BITS* is a 6-bit field to determine the purpose and contents of the segment (establish, close connections, etc.). The bits (from left to right) and their meanings, if set to 1, are:
  - URG: Urgent pointer field is valid
  - ACK: Acknowledgement field is valid
  - PSH: This segment requests a push
  - RST: Reset the connection
  - SYN: Synchronize sequence numbers
  - FIN: Sender has reached end of its byte stream
- *WINDOW* is a 16-bit unsigned integer that advertises how much data the TCP software is willing to accept, every time it sends a segment.
- *CHECKSUM* contains a 16-bit integer checksum used to verify the integrity of the data as well as the TCP header. It prepends a *pseudo header* to the segment and appends the segment with zeros to make the segment size a multiple of 16, and then computes the 16-bit checksum over the entire segment, just like IP header checksum. This allows the receiver to verify that the segment has reached its correct destination, which includes both a host IP address as well as a protocol port number.
- *URGENT POINTER* specifies the position in the TCP segment where the urgent data ends, when the URG code bit is set. TCP accommodates "out of band" signaling by allowing the sender to specify data as *urgent*

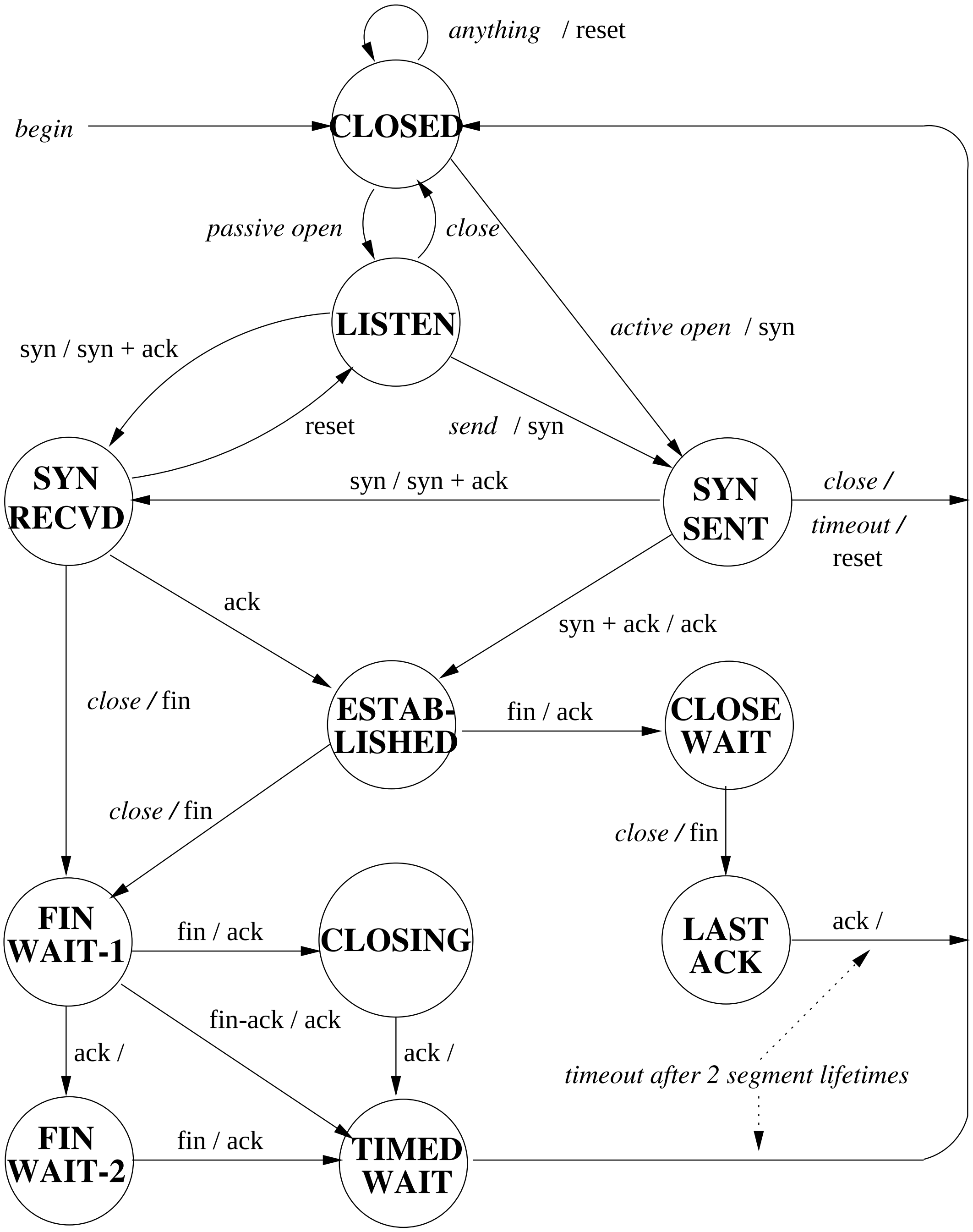

Figure 2.3: The TCP finite state machine.

2.3.3 TCP State Machine

The operation of TCP can be explained with a theoretical model called a *finite state machine*. Figure 2.3 shows the TCP finite state machine, with circles representing the states and arrows representing transitions between them. The label on each transition shows what TCP receives to cause the transition, and what it sends in response.

The TCP software at each endpoint begins in the *CLOSED* state. Application programs must issue either a *passive open* command (to wait for a connection from another machine), or an *active open* command (to initiate a connection). An active open command forces a transition from the *CLOSED* state to the *SYN SENT* state. When TCP follows the transition, it emits a SYN segment. When the other end returns a segment that contains a SYN plus ACK, TCP moves to the *ESTABLISHED* state and begins data transfer.

2.4 Port Scans

To an attacker, a machine is a collection of ports, communication endpoints, which might have processes "bound" to them, and "listening" on them. Many important services run on standardized ports, usually specified in `/etc/services` on UNIX-like systems. To scan the ports of a machine essentially means to discover what ports of a host are listening for connections.

Scan types

A number of techniques have been developed for surveying the protocols and ports on which a target machine is listening. They all offer different benefits and problems. The five most common TCP scans, which also happens to be the scan types that our detector is designed to detect, are described below.

1. **TCP connect() scan**: This is the most basic form of TCP scanning. The connect() system call provided by the operating system is used to open a connection to every interesting port on the machine. If the port is listening, connect() will succeed, otherwise the port isn't reachable. An advantage to this technique is that it doesn't need any special privileges. Any user on most UNIX boxes can use this call. Another advantage is speed. The downside is that this sort of scan is easily detectable and filterable. The target host logs will show connection and error messages for the services which accept the connection and then have it immediately shutdown.

2. **Stealth SYN scan**: This technique is often referred to as "half-open" scanning, because we don't open a full TCP connection. A SYN packet is sent, as if we are going to open a real connection, and wait for a response. A SYN/ACK indicates the port is listening. An RST is indicative of a non-listener. If a SYN/ACK is received, we immediately send an RST to tear down the connection (the kernel does this). The primary advantage of this scanning technique is that fewer sites will log it. Unfortunately, we need root privileges to build these custom SYN packets.

3. **Stealth FIN scan**: There are times when even SYN scanning isn't clandestine enough. Some firewalls, packet filters and programs watch for SYNs to restricted ports to detect these scans. FIN packets, on the other hand, may be able to pass through undetected.

The idea is that closed ports tend to reply to a FIN packet with the proper RST. Open ports, on the other hand, tend to ignore the packet in question. This is required TCP behavior.

4. **Xmas scan**: This is very similar to the FIN scan mentioned above. The FIN scan uses a bare FIN packet as the probe, while the Xmas tree scan turns on the FIN, URG, and PSH flags.

5. **Null scan**: This is also similar to the FIN scan, except that the Null scan turns off all flags.

Attackers typically go beyond this initial step, and determine what services run on the open ports, and typically use well-known vulnerabilities of these services, to gain illegal access to the system. Thus, a portscan is an indication that a potential intruder is trying to gain more information about your system, and possibly planning an attack. It is often a pre-cursor to an attack and a critical phase in itself, and its detection at this stage itself can reduce the damage by a multitude.

2.5 Sniffit

Sniffit [Br98] is a network sniffer for TCP/UDP/ICMP packets. **sniffit** produces very detailed technical information about the packets flowing through your network (SEQ, ACK, TTL, Window, ...) and packet contents in different formats (hex or plain text, ...) **sniffit** can, by default handle ethernet and PPP devices, but can easily be forced into using other devices. The original distribution can be downloaded from `http://sniffit.rug.ac.be/sniffit/sniffit.html`

The sniffer can easily be configured in order to 'filter' the incoming packets (to make the sniffing results easier to study). The config file allows you to be very specific on the packets to be processed. It uses the libpcap library developed at Berkeley Laboratory.

sniffit also has an interactive mode for active monitoring, and can also be used for continuous monitoring on different levels.

2.6 Tcpdump

Tcpdump [Ste01] is a tool for network monitoring and data acquisition. This software was originally developed by the Network Research Group at the Lawrence Berkeley National Laboratory. The original distribution is available via anonymous ftp to ftp.ee.lbl.gov, in tcpdump.tar.Z. More recent development is performed at `http://www.tcpdump.org/` Tcpdump uses libpcap, a system-independent interface for user-level packet capture.

The traffic of interest can be specified by command line arguments when tcpdump is run and tcpdump will only dump the required traffic.Tcpdump supports a variety of output options. By default it outputs packet headers in a user readable format to the standard output. This can be altered by specifying the option `-w <file>`, which tells tcpdump not to parse and print the packets, but to dump the raw packets to a file.The bytes of data from a packet to be dumped is by default 68 and can be altered by specifying the option `-s snaplen`, where `snaplen` is the length of a packet to be dumped.

It also has options to convert IP addresses to DNS names, or to keep them as they are. It can listen on a specified interface, or it can even take the input "dump" from a file, which has been previously saved using the `-w` option.

An **expression** selects which packets will be dumped. If no expression is given, all packets on the net will be dumped. Otherwise, only packets for which expression is 'true' will be dumped. This is the same **bpf filter** that **ngrep** also uses.

2.7 Nmap

Nmap [Fyo00] ("Network Mapper") is an open source utility for network exploration or security auditing. It was designed to scan large networks, as well as single hosts. Nmap uses raw IP packets to determine what hosts are available on the network, what services (ports) they are offering, what operating system (and OS version) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics. Nmap is free software, available with full source code under the terms of the GNU GPL. It can be downloaded from `http://www.insecure.org/nmap/`

nmap supports a large number of scanning techniques such as: UDP, TCP connect(), TCP SYN (half open), ftp proxy (bounce attack), Reverse-ident, ICMP (ping sweep), FIN, ACK sweep, Xmas Tree, SYN sweep, and Null scan. **nmap** also offers a number of advanced features such as remote OS detection via TCP/IP fingerprinting, stealth scanning, dynamic delay and retransmission calculations, parallel scanning, detection of down hosts via parallel pings, decoy scanning, port filtering detection, direct (non-portmapper) RPC scanning, fragmentation scanning, and flexible target and port specification.

2.8 PortSentry

PortSentry [Cra99] is part of the Abacus Project suite of security tools. It is a program designed to detect and respond to port scans against a target host in real-time. It can be downloaded from `http://www.psionic.com/abacus/portsentry`. It is a portscan detection and active defense system. PortSentry can detect the strobe-style scans (full connect() scans), SYN/half open scans, FIN scans, Null scans, Xmas tree scans, UDP scans (not really stealth scans per se), and any odd packet with flags not matching the above.

Features

Some of the features of PortSentry are listed below:

- Runs on TCP and UDP sockets to detect port scans against your system. PortSentry is configurable to run on multiple sockets at the same time so only one copy needs to be started to cover dozens of tripwired services.

- Stealth scan detection (Linux only right now). PortSentry will detect SYN/half-open, FIN, NULL, X-MAS and any odd-ball packet stealth scans. Four stealth scan operation modes are available.

- PortSentry will react to a port scan attempt by blocking the host in real-time. This is done through configured options of either dropping the local route back to the attacker, using the Linux **ipfwadm/ipchains** command, BSD **ipfw** command, and/or dropping the attacker host IP into a TCP Wrappers `hosts.deny` file automatically.

- PortSentry has an internal state engine to remember hosts that connected previously. This allows the setting of a trigger value to prevent false alarms and detect "random" port probing.
- PortSentry will report all violations to the local or remote syslog daemons indicating the system name, time of attack, attacking host IP and the TCP or UDP port a connection attempt was made to.
- Once a scan is detected, the host system will turn into a blackhole and disappear from the attacker. This feature stops most attacks.

PortSentry will bind to pre-defined TCP and UDP ports to wait for a connection, it will then react to block the host. This is how version 0.50 and below worked. This is compatible with most UNIX systems. Two new modes of operation in **portsentry** eliminate the need to bind to all ports and check, but can listen on a raw socket and analyze connections.

2.9 Snort

Snort [Mar01] is an open source network intrusion detection system, capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis and content searching/matching in order to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, and much more. Snort uses a flexible rules language to describe traffic that it should collect or pass, as well as a detection engine that utilizes a modular plug-in architecture. Snort has a real-time alerting capability as well, incorporating alerting mechanisms for syslog, user specified files, a UNIX socket, or WinPopup messages to Windows clients using Samba's smbclient.

Snort is free software, available with full source code under the terms of the GNU GPL. It can be downloaded from `http://www.snort.org/`

Snort has three primary functional modes. It can be used as a straight packet sniffer like **tcpdump** (section 2.6), a packet logger (useful for network traffic debugging, etc), or as a full blown network intrusion detection system.

Snort logs packets in either **tcpdump** binary format or in Snort's decoded ASCII format to a hierarchical set of directories that are named based on the IP address of the remote host.

Plug-ins allow the detection and reporting subsystems to be extended. Available plug-ins include database or XML logging, small fragment detection, portscan detection, and HTTP URI normalization, IP de-fragmentation, TCP stream reassembly and statistical anomaly detection. Snort has three primary run-time modes: sniffer, packet logger, and network intrusion detection.

- **Sniffer Mode:** When in this mode, Snort reads and decodes all packets from the network and dumps them to the stdout. To put Snort into straight sniffing mode, use the "-v" verbose switch. Various option switches format the packets differently. The traffic that shows up in this mode can be filtered by using BPF filters.

- **Packet Logger Mode:** This mode logs the packets to the disk in their decoded ASCII format. This mode is activated merely by specifying a directory to log packets to with the "-l" switch. Packets can be logged into specified logging directory in a hierarchy of directories based on the IP addresses of the packets on the wire. They can also be logged in terms of the network being monitored. They can also be logged in their raw binary format, to the disk. This allows them to be run through other tools like Ethereal, tcpdump, etc. Packet logger mode can be mixed with sniffer mode switches with no ill effects, however logging performance may be impacted by the slowness of the terminal.

- **Intrusion Detection Mode:** Snort enters IDS mode when a configuration file is specified with the "-c" switch. Output formats, rules, preprocessor configuration, etc are all specified in the configuration file. Logger mode is essentially disabled when in IDS mode, but that's because you specify which packets you want to log when in IDS mode. When an alert rule goes off the alert data is logged to the alerting mechanism (be default a file called "alert" in the logging directory) in addition to being logged to the logging mechanism. The default logging directory is `/var/log/snort`, which can be changed.

  You can use something like "rt" or just "tail -f" it to give a running display of system alerts. Alerts can also be sent to syslog (and monitored with something like swatch), or they can be sent out as WinPopup messages with smbclient. There are a variety of other alerting and logging mechanisms available.

2.9.1 Snort subsystems

Snort's architecture is focused on performance, simplicity, and flexibility. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem. These subsystems ride on top of the libpcap promiscuous packet sniffing library, which provides a portable packet sniffing and filtering capability. Program configuration, rules parsing, and data structure generation takes place before the sniffer section is initialized, keeping the amount of per packet processing to the minimum required to achieve the base program functionality.

The packet decoder

The decode engine is organized around the layers of the protocol stack present in the supported data-link and TCP/IP protocol definitions. Each subroutine in the decoder imposes order on the packet data by overlaying data structures on the raw network traffic. These decoding routines are called in order through the protocol stack, from the data link layer up through the transport layer, finally ending at the application layer. Speed is emphasized in this section, and the majority of the functionality of the decoder consists of setting pointers into the packet data for later analysis by the detection engine. Snort provides decoding capabilities for Ethernet, SLIP, and raw (PPP) data-link protocols. ATM support is under development.

The detection engine

Snort maintains its detection rules in a two dimensional linked list of what are termed Chain Headers and Chain Options. These are lists of rules that have been condensed down to a list of common attributes in the Chain Headers, with the detection modifier options contained in the Chain Options. For example, if forty five CGI-BIN probe detection rules are specified in a given Snort detection library file, they generally all share common source and destination IP addresses and ports. To speed the detection processing, these commonalities are condensed into a single Chain Header and then individual detection signatures are kept in Chain Option structures.

These rule chains are searched recursively for each packet in both directions. The detection engine checks only those chain options which have been set by the rules parser at run-time. The first rule that matches a decoded packet in the detection engine triggers the action specified in the rule definition and returns.

The logging/alerting subsystem

The alerting and logging subsystem is selected at run-time with command line switches. There are currently three logging and five alerting options. The logging options can be set to log packets in their decoded, human readable format to an IP-based directory structure, or in tcpdump binary format to a single log file. The decoded format logging allows fast analysis of data collected by the system. The tcpdump format is much faster to record to the disk and should be used in instances where high performance is required. Logging can also be turned off completely, leaving alerts enabled for even greater performance improvements.

Alerts may either be sent to syslog, logged to an alert text file in two different formats, or sent as WinPopup messages using the Samba smbclient program. The syslog alerts are sent as security/authorization messages that are easily monitored with tools such as swatch. WinPopup alerts allow event notifications to be sent to a user-specified list of Microsoft Windows consoles running the WinPopup software. There are two options for sending the alerts to a plain text file; full and fast alerting. Full alerting writes the alert message and the packet header information through the transport layer protocol. The fast alert option writes a condensed subset of the header information to the alert file, allowing greater performance under load than full mode. There is a fifth option to completely disable alerting, which is useful when alerting is unnecessary or inappropriate, such as when network penetrations tests are being performed.

2.9.2 Snort plug-ins

Snort version 1.5 introduces a major new concept, plug-ins. There are two types of plug-in currently available in Snort: detection plug-ins and preprocessors. Detection plug-ins check a single aspect of a packet for a value defined within a rule and determine if the packet data meets their acceptance criteria. For example, the TCP flags detection plug-in checks the flags section of TCP packets for matches with flag combinations defined in a particular rule. Detection plug-ins may be called multiple times per packet with different arguments.

Preprocessors are only called a single time per packet and may perform highly complex functions like TCP stream reassembly, IP de-fragmentation, or HTTP request normalization. They can directly manipulate packet data and even call the detection engine directly with their modified data. They can perform less complex tasks like statistics gathering or threshold monitoring as well.

SPADE and SPICE

SPADE stands for Statistical Packet Anomaly Detection Engine and is produced by Silicon Defense (`http://www.silicondefense.com/`). It is a Snort plug-in to report unusual, possibly suspicious, packets. SPICE is the Stealthy Probing and Intrusion Correlation Engine. It is part of Silicon Defense's work. It will eventually consist of two parts, an anomaly sensor (SPADE) and a portscan correlator. The basic operation of this will be that Spade will monitor the network and report anomalous events to the correlator. The correlator will then group these events together and send out reports of portscans, even those that have been crafted to be difficult to detect (e.g., they probe slowly, from different sources, or they randomize the scan). This distribution is the sensor component of Spice. The correlator is under active development.

What Snort does

SPADE will review the packets received by Snort, find those of interest (TCP SYNs into your homenets, if any), and report those packets that it believes are anomalous along with an anomaly score.

The anomaly score that is assigned is based on the observed history of the network. The fewer times that a particular kind of packet has occurred in the past, the higher its anomaly score will be. Packets are classified by their joint occurrence of packet field values. For example, packets with destination IP of 10.10.10.10 and destination port of 8080 might be one kind of packet.

To do this, a probability table is maintained that reflects the occurrences of different kinds of packets in history, with higher weight on more recent events. We would know, for example, that P(dip=10.10.10.10, dport=8080) is 10% but that P(dip=10.10.10.10, dport=8079) is 0.1%. The anomaly score is calculated directly from the probability. For a packet X, $A(X) = -\log_2 P(X)$. So the anomaly score for a 10.10.10.10, 8080 packet is 3.32 (not very anomalous) and the score for a 10.10.10.10, 8079 packet is 9.97 (fairly anomalous).

At any given time, a reporting threshold is defined for the sensor. For each event that exceeds this threshold, an alert is sent. It is sent to the same place(s) that a rule-based alert would be sent to (e.g., Snort alert file, syslog, etc.).

In addition to reporting anomalous events, SPADE can also be configured to generate reports about the network on which it is run. For example, it can tell you the amount of entropy in your destination ports and in your source ports given your destination ports or produce periodic reports of the number of packets seen and order statistics such as median of the anomaly scores produced.

What Snort doesn't do

SPADE cannot tell you if a particular reported packet is bad or hostile. It merely knows that certain packets are relatively unusual and has an idea how unusual. You should expect to see alerts about benign activity.

It also cannot report things like attempts to exploit CGI vulnerabilities on a popular web server. This would depend on looking at the packet contents and SPADE just looks at certain parts of the header.

SPADE will not group related anomalous events together. That will be the job of the correlator. We can use SnortSnarf (`http://www.silicondefense.com/snortsnarf/`) to help with this task; version 090700.1 generates a special section to browse anomaly reports. Snort-Snarf is a Perl program to take files of alerts from Snort and produce HTML output intended for diagnostic inspection and tracking down problems. The model is that one is using a cron job or something similar to produce daily/hourly file of snort alerts. This script can be run on each such file to produce a convenient HTML breakout of all the alerts.

SPADE Output

SPADE produces two types of messages, which are sent to wherever Snort usually sends alerts (e.g., alert file, syslog, etc.).

The more common one has the message "spp_anomsensor: Anomaly threshold exceeded: A", where A is a number. This indicates that the packet mentioned was assessed as anomalous and the anomaly score was A.

SPADE may also periodically produce messages of the form: "spp_anomsensor: Threshold adjusted to T after X alerts (of N)". This indicates that the alerting threshold was changed to T. This happens when one of the threshold adapting mechanisms is used. The message also gives information about the number of alerts (X) sent since the last time the threshold was adjusted and the total number of packets (N) accepted by SPADE during that time.

2.9.3 Conclusions about Snort

Snort is a flexible tool with a wide variety of uses. It is intended to be used in the most classic sense of a network intrusion detection system. It examines network traffic against a set of rules, and alerts administrators to suspicious network activity so that they may react appropriately. There are many other areas where Snort can be useful as well.

Snort was designed to fulfill the requirements of a prototypical lightweight network intrusion detection system. It has become a small, flexible, and highly capable system that is in use around the world on both large and small networks. It has attained its initial design goals and is a fully capable alternative to commercial intrusion detection systems in places where it is cost inefficient to install full featured commercial systems.

2.10 Scope of our work

In an academic environment like IIT Bombay, we have many machines with identical configurations. An attacker trying to learn information about a single machine can do so by scanning a few ports on multiple machines. Such a scan cannot be easily detected by any of the existing freely available Intrusion Detection Systems. We describe the details of developing such a system from chapter 3 onwards.

Chapter 3

Distributed Intrusion Detection

3.1 Motivation for distributed IDS

When we talk about Distributed Intrusion Detection, we refer to the "detection" of distributed intrusion attempts, rather than the notion of a "distributed" detection of intrusion, which consists of a framework of numerous data collection agents and a "distributed" analysis of the same.

3.1.1 Our setup

In the Software Lab of Computer Science and Engineering Department, IIT Bombay, we have a cluster of 26 machines with IP addresses ranging from 192.168.211.221 (`pro-01.cse.iitb.ac.in`) to 192.168.211.246 (`pro-26.cse.iitb.ac.in`) with the same configuration - all boot the same kernel - Linux 2.2.12-20, mount the same disk partitions - `/home1`, `/home2`, `/home3`, `/home4`, from the NFS and NIS server 192.168.211.250 (`athira.cse.iitb.ac.in`), and have the same services running on each of them, shown below in table 3.1. Some other stray ports like 6000+ may also be listening for connections to the X server, if any.

Port	State	Service
21/tcp	open	ftp
22/tcp	open	ssh
23/tcp	open	telnet
79/tcp	open	finger
111/tcp	open	sunrpc
513/tcp	open	login
514/tcp	open	shell
738/tcp	open	unknown

Table 3.1: Open ports on 192.168.211.241

Suppose an intruder wants to exploit the password cracking attack. First he needs to check all the machines to see which one of them he can use, to gain access. The telnet or ssh services might have very well been running on obscure ports, and so, the first thing that he needs to check is the array of open ports on these machines.

3.1.2 Portsentry and Snort

We looked at two IDSs that can detect portscan attempts, an HIDS **portsentry** in section 2.8 and an NIDS **snort** in section 2.9.

Portsentry binds to all the ports to be monitored. Hence, all these ports show up as "open" to the intruder. The intruder might get more curious and return, and possibly find a real problem with our system, which portsentry cannot defend.

Portsentry has this concept of listening to a "list" of ports. It will monitor only those ports which are specified in a configuration file. Many a times, after the first installations, system administrators rarely bother to update their configurations, though changes to the system keep happening. This is typically true when multiple administrators handle the machines. Hence, an important service on an obscure port might not be "watched" by portsentry.

Portsentry has a state engine that remembers hosts that connected to it. Hence, using multiple source IP addresses to scan the machine will not be detected if the scan threshold is not exceeded.

Snort currently has a preprocessor for detecting portscans. Here, a portscan is defined as TCP connection attempts to more than P ports in T seconds. Ports can be spread across any number of destination IP addresses, and may all be the same port if spread across multiple IPs. Version 1.8 could only do one-to-one and one-to-many portscan detection.

3.1.3 The problem

If the attacker decides to scan one single machine, a host-based software like portsentry (section 2.8) can detect this scan as a huge number of connections would alarm the system administrator. If the scan is executed over a short period of time, a network-based software like snort (section 2.9) would also be able to detect it.

The problem arises in the following situation: Suppose that knowledge about the identical configuration of our machines is common and the attacker decides to scan a few ports on each target machine, maybe using different source IP addresses (hosts). Also, in addition to the "distributed" nature of the probe, he executes the scans slowly, at a rate lesser than snort's detection threshold.

Such an attack will probably go unnoticed or raise a few alarms in portsentry logs. But, we must note that portsentry will have to be running on all the 26 machines. If this was done too, there is no way of assembling all the 26 log files and making an intelligent conclusion about the few "source" machines and the "target" machines.

Hence, the situation to be dealt with is a possibly slow, random, distributed, scan of any of the target systems. We classify these scan sweeps into four categories. In all these types, the number of ports scanned on each target machine is insignificant.

- **One-to-one** scan: This is a portscan performed by one single source on one single target machine.

- **One-to-many** scan: This is a scan performed by one source machine on more than one target machines.

- **Many-to-one** scan: More than one source machine scans a single target machine. The state engine of systems like portsentry is misled here.

- **Many-to-many** scan: A group of source machines (more than one) performs scans on a group of target machines (more than one)

Our aim is to detect all these scan sweeps, across the five types of TCP scans described in section 2.4.

3.2 Analysis of the scan types

Each of the five scan types described in section 2.4 have typical signatures. We performed all these scans using nmap, and collected the resulting dumps of traffic, and came up with the following analysis for each of these scans.

3.2.1 TCP connect() scan

The source machine in this case executes the *connect()* system call and uses *shutdown()* to tear down the connection. Here, the typical sequence of packets for an open port on the target machine is:

Seq. No.	Direction	TCP Flags set
1	source to target	SYN
2	target to source	SYN, ACK
3	source to target	ACK
4	source to target	ACK, FIN, RST

Table 3.2: Sequence of packets for a TCP connect() scan for an open port

For a closed port on the target machine, the sequence of packets is:

Seq. No.	Direction	TCP Flags set
1	source to target	SYN
2	target to source	RST, ACK

Table 3.3: Sequence of packets for a TCP connect() scan for a closed port

This is the only scan which can be performed by a normal user too. All other attacks require building custom packets to send on the network, and that requires superuser privileges.

All closed ports react to SYN packets as shown in table 3.3 and hence this is considered a part of all the scan signatures.

3.2.2 Stealth SYN scan

The stealth SYN scan does not complete the 3-way TCP connection establishment handshake, and tears down the connection with an RST after the target host responds with a SYN/ACK, indicating an open port.

Seq. No.	Direction	TCP Flags set
1	source to target	SYN
2	target to source	SYN, ACK
3	source to target	RST

Table 3.4: Sequence of packets for a stealth SYN scan for an open port

3.2.3 Stealth FIN scan

The stealth FIN scan sends a bare FIN packet to the target host. Open ports ignore this packet and closed ports react with an RST.

Seq. No.	Direction	TCP Flags set
1	source to target	FIN

Table 3.5: Sequence of packets for a stealth FIN scan for an open port

3.2.4 Xmas scan

The Xmas scan is similar to the stealth FIN scan. It also turns on the FIN, PSH and URG flags. Open ports ignore this packet and closed ports react with an RST.

Seq. No.	Direction	TCP Flags set
1	source to target	FIN, PSH, URG

Table 3.6: Sequence of packets for an Xmas scan for an open port

3.2.5 Null scan

The Null scan is also like the stealth FIN scan. It sends a packet to the target, with all flags turned *off*. Open ports ignore this packet and closed ports react with an RST.

Seq. No.	Direction	TCP Flags set
1	source to target	*none*

Table 3.7: Sequence of packets for a Null scan for an open port

3.3 Our approach

We describe the design of a network-based portscan detector, which would pick up packets from the underlying broadcast LAN and deliver it to our detector program. Typically, this would result in huge amounts of network data to be analyzed. We needed to distinguish the scan signatures from normal TCP traffic.

The algorithm for the detector runs like this:

1. Capture each packet on the network and examine it further.

2. Check if the packet is of type TCP.

3. Extract the source and destination IP addresses and ports from the packet.

4. For each scan type (TCP connect(), stealth SYN, stealth FIN, Xmas, Null) to be detected, maintain a "connection list" of valid TCP connections to watch for.

5. Whenever a scan signature is detected, we store this connection record, i.e. source and target IP addresses and ports, into two correlation lists, *srcIP* and *tarIP*. At the same time, this connection record is deleted from the connection list.

6. In one pass through the correlation lists, we examine the number of targets that each source has scanned, and the number of sources that each target got scanned by.

7. We then list out the one-to-one, one-to-many, many-to-one, and many-to-many scan sweeps found in the correlation lists.

Firstly, only TCP packets are processed further. The concept of a "TCP connection" is central to the detection algorithm. A list of "live" connections between two TCP endpoints is maintained. This is done by observing the SYN, ACK, FIN and RST packets. Whenever a SYN packet from host a, port b to host c, port d is observed, a connection record is made for this connection. All further packets from (a, b) to (c, d) are regarded as part of this connection. This record is purged from the connection list only when a FIN or an RST from (a, b) to (c, d) or (c, d) to (a, b) is observed.

When a new packet is received, a search is made through the connection list, to find out if this packet belongs to a live connection. If it does, the packet number and flags are examined further for scan signatures. If the packet doesn't belong to any connection, it might be a SYN packet, in which case, a new entry is made to the connection list. It might be a retransmitted packet of an earlier connection too. It is ignored in this case.

If the packet doesn't belong to any connection, it might also be indicative of a FIN, Xmass or Null tree scan. the TCP flags for this packet to are checked see if FIN, URG, PSH, are turned on, or if all the flags are turned off. If any of these are positives, the connection attempt is a scan. For detecting TCP connect() type scans, the connection record is updated as and when corresponding packets are seen. When the fourth packet is spotted as an RST from the source to the target, it is indicative of a TCP connect() scan.

When we conclude that there was an attempt to portscan some machine *target*, by some machine *source*, we remove the entry from the connection list and add the details of this positive scan into a two correlation lists. One list maintains details of source hosts - each node represents one source host and maintains information on the number of targets scanned, counts on how many times each target was scanned, and a pointer to each such target into the target list. The other list is identically structured - each node represents one target host and maintains information on the number of sources that portscanned it, counts on how many times each source did so, and a pointer to each such source into the source list.

In order to present a correlated conclusion, we make passes through the source and destination IP lists, and list the scan sweep types as one-to-one, one-to-many, many-to-one or many-to-many depending on the number of targets scanned by a source and the number of sources scanning a target.

In chapter 4, we will further examine these data structures and algorithms in detail.

Chapter 4

Our implementation

This chapter describes the implementation of the portscan detector, along with a TCP connect() scanner routine. The data structures and algorithms of the main routines are described.

4.1 TCP connect() scanner

A TCP connect() scanner which performs the TCP connect() scan (section 3.2.1) was built. It connects to a specified machine on a specified TCP port, using *connect()* system call. The next call is a *shutdown()* call which closes the connection.

4.2 Data structures

Each node in the connection list is a `struct session_info` shown in figure 4.1. Each node stores connection information like source and target IP and port numbers, timestamp of the last packet seen on this connection, and status of SYNs and ACKs received. It also has a pointer to the next node in the connection list. Each node is representative of one live connection in the network.

Struct session_info

conn: ip1:port1-ip2:port2
xconn: ip2:port2-ip1:port1
proto: tcp/udp
tslp: timestamp of the last packet seen
SASrecd: flag 0/1... SYN/ACK-SYN recd
ACKrecd: flag 0/1... ACK recd
session_info *next: Next session node in linked list

Figure 4.1: A node in the connection list

When a TCP SYN segment is received, existing connections in the list are searched through. If a record for this source and target IP address and ports exists, it is a retransmitted packet, and the entry in the connection list is updated - the timestamp is changed. If this entry doesn't exist, a new node is created and added to the end of the connection list. Function `add_list` "adds" this packet to the list.

When a FIN or RST is seen, it signifies the end of the connection. This connection record is then searched in the connections list, using the function `search_list`, which returns a pointer to this connection record. Function `delete_list_ptr` then deletes this node from the connection list.

When a scan is detected, the corresponding connection record is purged from the connection list, and the **correlation lists** - source list pointed to by *srcIP* and target list pointed to by *tarIP* - are updated with this record. The **source list** is a list of nodes, each node corresponding to a source machine which performs portscans. Similarly, the **target list** is a list of nodes, each node corresponding to a target machine, which is portscanned by some machine in the source list.

Struct mac

IP: IP address of the source/target machine
lenmacptr: length of the list of pointers to the other list
macptr ptr_struct: has pointer to targets for a source
mac *next: Next machine node in correlation list

Figure 4.2: A node in the source or target correlation lists

Each node in these lists is a `struct mac` shown in figure 4.2. For the source list, each node contains the IP address, number of target hosts scanned, a pointer to the next node in the list, and ptr_struct, a `struct macptr`, shown in figure 4.3. This contains a pointer to the node in the target list, that this source has portscanned, a count of the number of times this source host has scanned that target host, and a pointer to the next target scanned by this host.

Struct macptr

mac *macptr: Indicates machine in the other list
count: A count of scans between these 2 machines
macptr *next: Next node in the list

Figure 4.3: A node that points to the target scanned (when in a source mac) or the scanner source (when in a target mac)

The correlation lists are therefore interconnected, with some amount of similarity. For the test case one-to-many scan described in table 5.2, a snapshot of the correlation lists looks as shown in figure 4.4.

4.3 Routines

The central flow of control of the detector is:

1. Open the network device to read from (live/offline)

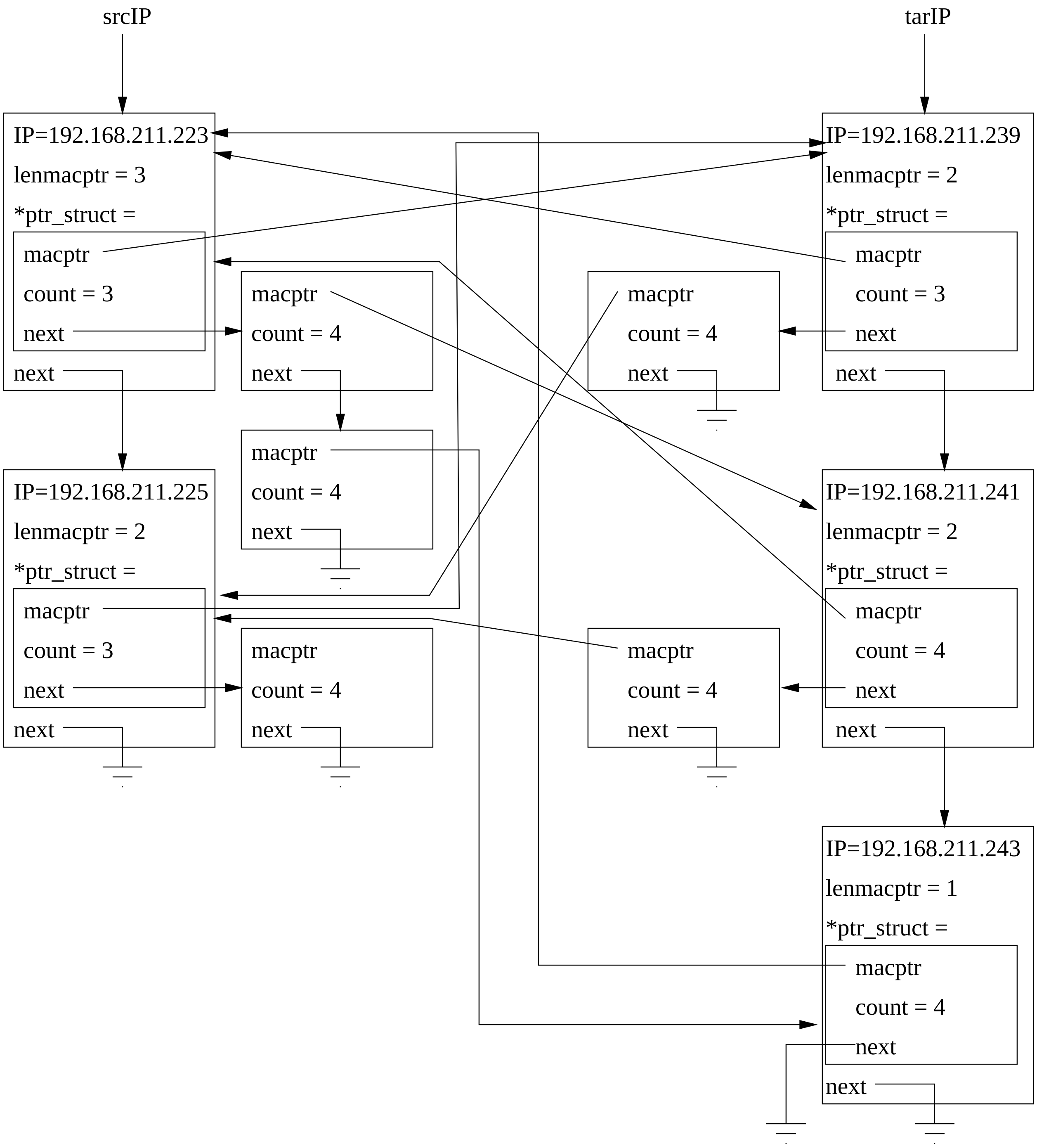

Figure 4.4: Correlation lists for the one-to-many scan in table 5.2

2. Depending on the option specified on the commandline, for the scan to be detected, call a routine in a `pcap_loop()` call, to handle every packet, till EOF. The individual routines are described in section 4.4

Option	TCP Scan type	Routine called
`-sT`	connect()	`packethandler`
`-sS`	stealth SYN	`packethandler_S`
`-sF`	stealth FIN	`packethandler_F`
`-sX`	Xmas	`packethandler_X`
`-sN`	Null	`packethandler_N`

Table 4.1: Options passed to *detect* and routine called

3. Call routines: `onetoone()`, `onetomany()`, `manytoone()` and `manytomany()` to go through `srcIP` and `tarIP` lists and print scan sweeps discovered. These are described in section 4.5.

4. Call `deletenode(srcIP)` and `deletenode(tarIP)` to free the `srcIP` and `tarIP` lists.

4.4 Detection

In this section, algorithms for detecting each type of TCP portscan are described.

4.4.1 TCP connect() scan

This is the scan described in section 3.2.1. For each packet seen by the network interface, the following is done:

1. If the packet is a SYN and not-ACK, it is the first packet of a connection. Search for this packet in the connection list.
   `struct session_info *search_pointer = search_list (conn, xconn);`
   - If found, reset all fields of `*search_pointer`.
   - If not found,
     `add_list(conn, xconn, packet_header->timestamp);`
2. Any other packet, search if the record exists in the connection list.
   `struct session_info *search_pointer = search_list (conn, xconn);`
   - If found in the list,
     - If the packet is a FIN or RST, and not-SYN, and ACK, and the ACK/SYN has been received, and the third ACK has also been received, this is the fourth packet of the connection - the signature of this scan. Add the source and the target IP to the correlation lists. Remove this entry from the connection list.
     - If the packet is an RST, and an ACK and not-FIN and not-SYN, and the SYN/ACK has not been received, this is the second packet of the connection, and its an RST, indicative of a closed port. This is also an indication of a

scan. Add the source and target IP to the correlation lists. Remove the entry from the connection list.

  - If the packet is the second packet in a TCP connection, update the entry in the connection list with the new timestamp and set the SYN/ACK received flag.
  - If the packet is the third packet in a TCP connection, update the entry in the connection list with the new timestamp and set the ACK received flag.
  - If the packet is the fourth packet in a TCP connection, this is a normal TCP connection and this entry can be safely deleted from the connection list.
  - If the packet is a FIN or and RST, this entry should be deleted from the connection list.

- If not found, if the packet is a SYN and ACK, its the second packet of a connection, add it to the list.
  `add_list(conn, xconn, packet_header->timestamp);`

4.4.2 Stealth SYN scan

This is the scan described in section 3.2.2. For each packet seen by the network interface, the following is done:

1. If the packet is a SYN and not-ACK, it is the first packet of a connection. Search for this packet in the connection list.
   `struct session_info *search_pointer = search_list (conn, xconn);`
   - If found, reset all fields of `*search_pointer`.
   - If not found,
     `add_list(conn, xconn, packet_header->timestamp);`
2. For any other packet, search if the record exists in the connection list.
   `struct session_info *search_pointer = search_list (conn, xconn);`
   - If not found in the list, and if it is a SYN and an ACK, add it to the connection list. Discard everything else and return.
   - If the packet is found in the connection list,
     - If it is the second packet in the connection and and RST and ACK and not FIN, it is an indication of a closed port being scanned. Add this connection to the correlation lists and delete it from the connection list.
     - If it is the second packet in the connection, update the record in the connection list.
     - If it is the third packet in the connection and an RST, it is indicative of a stealth SYN scan - the signature of this scan. Add this connection to the correlation lists and delete it from the connection list.
     - If it is the third packet in the connection and not an RST, it is a normal TCP connection and hence delete this entry from the connection list.

4.4.3 Stealth FIN scan

This is the scan described in section 3.2.3. For each packet seen by the network interface, the following is done:

1. If the packet is a SYN, search the connection list for this record. If one is found, update the entry with the new timestamp. If not, add this entry to the connection list.
2. For any other packet, search if the record exists in the connection list.
   - If not found in the connection list, and if it is the second second packet in the connection, with a SYN and an ACK, add this record to the connection list.
   - If not found in the connection list, and if it is a FIN and not-ACK, it is an indication of a FIN scan - add this record to the correlation lists and remove the entry from connection list.
   - If found in the list, and it is a FIN and ACK, its the last packet of the connection, remove the entry from the connection list.
   - If found in the list, and this is the second packet of the connection, and an RST and ACK, indicative of a closed port scan, add this record to the correlation lists, and remove the entry from connection list.

4.4.4 Xmas tree scan

This is the scan described in section 3.2.4. For each packet seen by the network interface, the following is done:

1. If the packet is a SYN, search the connection list for this record. If one is found, update the entry with the new timestamp. If not, add this entry to the connection list.
2. For any other packet, search if the record exists in the connection list.
   - If not found in the connection list, and if it is the second second packet in the connection, with a SYN and an ACK, add this record to the connection list.
   - If not found in the connection list, and if it is a FIN, not-ACK, PSH and URG, it is an indication of an Xmas scan - add this record to the correlation lists and remove the entry from connection list.
   - If found in the list, and it is a FIN and ACK, its the last packet of the connection, remove the entry from the connection list.
   - If found in the list, and this is the second packet of the connection, and an RST and ACK, indicative of a closed port scan, add this record to the correlation lists, and remove the entry from connection list.

4.4.5 Null scan

This is the scan described in section 3.2.5. For each packet seen by the network interface, the following is done:

1. If the packet is a SYN, search the connection list for this record. If one is found, update the entry with the new timestamp. If not, add this entry to the connection list.

2. For any other packet, search if the record exists in the connection list.
   - If not found in the connection list, and if it is the second second packet in the connection, with a SYN and an ACK, add this record to the connection list.
   - If not found in the connection list, and if it is a packet with *no* flags set, it is an indication of an Xmas scan - add this record to the correlation lists and remove the entry from connection list.
   - If found in the list, and it is a FIN and ACK, its the last packet of the connection, remove the entry from the connection list.
   - If found in the list, and this is the second packet of the connection, and an RST and ACK, indicative of a closed port scan, add this record to the correlation lists, and remove the entry from connection list.

4.5 Correlation

In this section, the algorithm for inferring the sweep types from the correlation lists, is described.

- **One-to-one scan:** Starting from the first node pointed to by *srcIP*, look for all nodes with `lenmacptr` equal to one. Print this source IP and follow the `ptr_struct.macptr` to get the target IP.

- **One-to-many scan:** Starting from the first node pointed to by *srcIP*, look for all nodes with `lenmacptr` greater than one. For all such nodes: Traverse through their `ptr_struct` lists and list out the target IP addresses for this source IP address.

- **Many-to-one scan:** Starting from the first node pointed to by *tarIP*, look for all nodes with `lenmacptr` greater than one. For all such nodes: Traverse through their `ptr_struct` lists and list out the source IP addresses for this target IP address.

- **Many-to-many scan:** Starting from the first node pointed to by *srcIP*, look for all nodes with `lenmacptr` greater than one. For all such nodes: Traverse through their `ptr_struct` lists and list out those source IP addresses which have `lenmacptr` greater than one.

Chapter 5

Experiments and Results

5.1 The experimental setup

Further to the description in section 3.1.1, we selected a LAN segment on the hub and the exact setup is shown in figure 5.1. Three machines are used as source machines and three machines are target machines. One machine is in promiscuous mode and captures all the network data, which is finally given to our detector program.

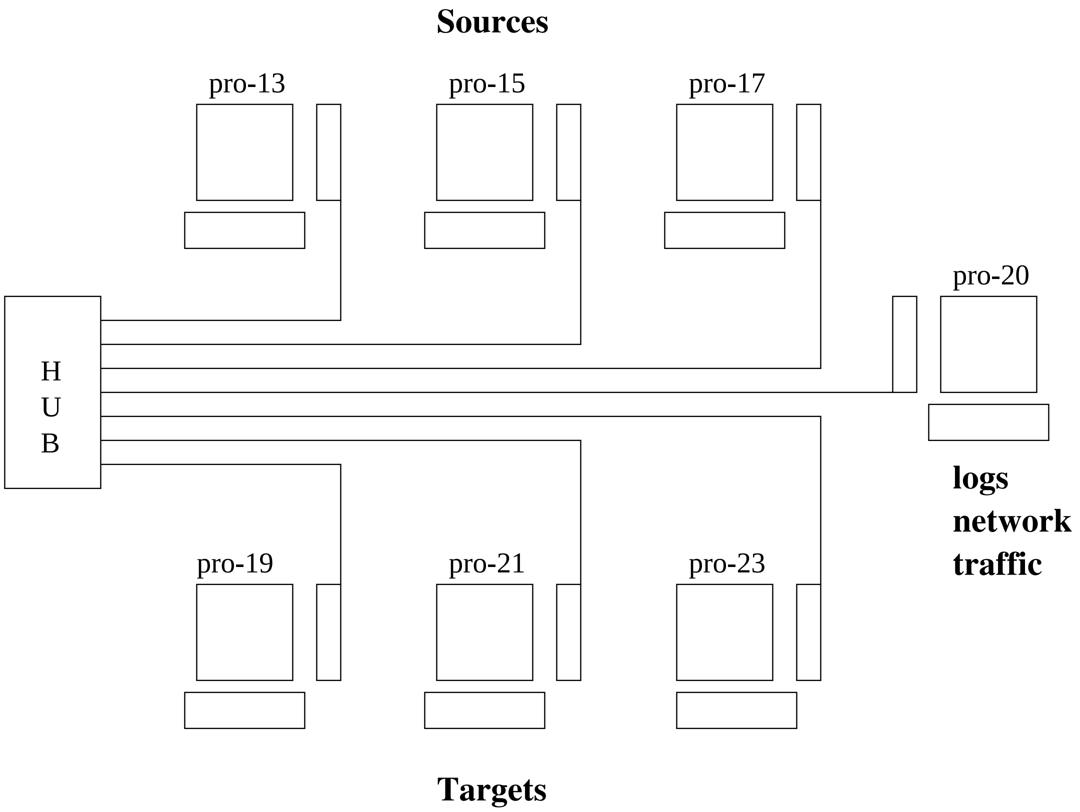

Figure 5.1: The experimental setup.

5.2 Scan sweeps

The following scan sweeps are performed:

The machine pro-13 connects to machine pro-19 on ports 25 and 119 and immediately sends an RST to tear down the connection, thus executing a one-host-to-one-host TCP connect() scan. Similarly, machine pro-15 scans ports 21, 23, and 80 of pro-21 and pro-17 scans pro-23 on ports 22 and 79. (Table 5.1).

Source	Target	TCP Ports
pro-13	pro-19	25, 119
pro-15	pro-21	21, 23, 80
pro-17	pro-23	22, 79

Table 5.1: One-to-one scan

The machine pro-13 connects to machine pro-19 on ports 7, 20, 21, to pro-21 on ports 22, 23, 25, 53, and to pro-23 on ports 69, 79, 80, 88 and immediately sends an RST to tear down the connection, thus executing a one-host-to-many-host TCP connect() scan. Similarly, machine pro-15 scans ports 110, 111, 119 of pro-19 and ports 139, 143, 194, 220 of pro-21. (Table 5.2).

Source	Target	TCP Ports
pro-13	pro-19	7, 20, 21
	pro-21	22, 23, 25, 53
	pro-23	69, 79, 80, 88
pro-15	pro-19	110, 111, 119
	pro-21	139, 143, 194, 220

Table 5.2: One-to-many scan

The machine pro-13 connects to machine pro-21 on ports 443, 513, 518. pro-15 connects to pro-21 on ports 873, 3130, 6667, and pro-17 connects to pro-21 on ports 107, 20, 21, 23. Source machines then immediately send an RST to tear down the connection, thus executing a many-hosts-to-one-host TCP connect() scan. (Table 5.3).

Source	Target	TCP Ports
pro-13	pro-21	443, 513, 518
pro-15	pro-21	873, 3130, 6667
pro-17	pro-21	107, 20, 21, 23

Table 5.3: Many-to-one scan

The machines pro-13, pro-15 and pro-17 all connect to machine pro-19 on ports 7, 20, 21, 79, to pro-21 on ports 80, 113, 119, 139, to pro-23 on ports 143, 194, 667. Source machines then immediately send an RST to tear down the connection, thus executing a many-hosts-to-many-hosts TCP connect() scan. (Table 5.4).

Source	Target	TCP Ports
pro-13	pro-19	7, 20, 21, 79
	pro-21	80, 113, 119, 139
	pro-23	143, 194, 667
pro-15	pro-19	7, 20, 21, 79
	pro-21	80, 113, 119, 139
	pro-23	143, 194, 667
pro-17	pro-19	7, 20, 21, 79
	pro-21	80, 113, 119, 139
	pro-23	143, 194, 667

Table 5.4: Many-to-many scan

5.3 Results

The machine `pro-20` was running *tcpdump* when the scans were performed. The results from the detect program for the scans are shown below:

The dump file corresponding to the one-to-one scan is passed to the detector and the observed output is recorded. The program successfully detects three one-to-one scans, from pro-13 (192.168.211.233) to pro-19 (192.168.211.239), from pro-15 (192.168.211.235) to pro-21 (192.168.211.241), and from pro-17 (192.168.211.237) to pro-23 (192.168.211.243) (scan of table 5.1).

```
[mamata@sapphire - ~/prog/many1] ./detect expt/one-to-one
Device opened offline
No. of packets processed     : 252
No. of packets not examined  : 158
No. of analyzed connections  : 12
No. of incomplete connections: 1

-----------------One to One--------------------
Source: 192.168.211.233        Target: 192.168.211.239
Source: 192.168.211.235        Target: 192.168.211.241
Source: 192.168.211.237        Target: 192.168.211.243

-----------------One to Many--------------------

-----------------Many to One--------------------

-----------------Many to Many--------------------
```

The dump file corresponding to the one-to-many scan is passed to the detector and the observed output is recorded. The program successfully detects two one-to-many scans, from pro-13 (192.168.211.233) to pro-19 (192.168.211.239), pro-21 (192.168.211.241) and pro-23 (192.168.211.243) and from pro-15 (192.168.211.235) to pro-19 (192.168.211.239) and pro-21 (192.168.211.241) (scan of table 5.2).

Some other stray scans are also picked up in addition to the ones we performed. Some of these are due to reverse *ident* lookups.

```
[mamata@sapphire - ~/prog/many1] ./detect expt/one-to-many
Device opened offline
No. of packets processed     : 266
No. of packets not examined  : 182
No. of analyzed connections  : 23
No. of incomplete connections: 1

-----------------One to One--------------------

-----------------One to Many--------------------
Source: 192.168.211.233
 Targets:
        192.168.211.239        192.168.211.241        192.168.211.243
Source: 192.168.211.235
 Targets:
        192.168.211.239        192.168.211.241

-----------------Many to One--------------------
Target: 192.168.211.239
 Sources:
        192.168.211.233        192.168.211.235
Target: 192.168.211.241
 Sources:
        192.168.211.233        192.168.211.235

-----------------Many to Many--------------------
Source: 192.168.211.233
 Targets:
        192.168.211.239        192.168.211.241
Source: 192.168.211.235
 Targets:
        192.168.211.239        192.168.211.241
```

The dump file corresponding to the many-to-one scan is passed to the detector and the observed output is recorded. The program successfully detects one many-to-one scans, from pro-13 (192.168.211.233), pro-15 (192.168.211.235), and pro-17 (192.168.211.237) to target pro-21 (192.168.211.241) (scan of table 5.3).

```
[mamata@sapphire - ~/prog/many1] ./detect expt/many-to-one
Device opened offline
No. of packets processed     : 357
No. of packets not examined  : 188
No. of analyzed connections  : 15
No. of incomplete connections: 0
```

```
-----------------One to One--------------------

-----------------One to Many--------------------

-----------------Many to One--------------------
Target: 192.168.211.241
 Sources:
        192.168.211.233         192.168.211.235         192.168.211.237

-----------------Many to Many--------------------
```

The dump file corresponding to the many-to-many scan is passed to the detector and the observed output is recorded. The program successfully detects one many-to-many scan, from pro-13 (192.168.211.233), pro-15 (192.168.211.235), and pro-17 (192.168.211.237) to targets pro-19 (192.168.211.239), pro-21 (192.168.211.241), and pro-23 (192.168.211.243) (scan of table 5.4).

These scans also show up in the one-to-many and the many-to-one listings, as is expected.

```
[mamata@sapphire - ~/prog/many1] ./detect expt/many-to-many
Device opened offline
No. of packets processed     : 985
No. of packets not examined  : 524
No. of analyzed connections  : 52
No. of incomplete connections: 2

-----------------One to One--------------------

-----------------One to Many--------------------
Source: 192.168.211.233
 Targets:
        192.168.211.239         192.168.211.241         192.168.211.243
Source: 192.168.211.235
 Targets:
        192.168.211.239         192.168.211.241         192.168.211.243
Source: 192.168.211.237
 Targets:
        192.168.211.239         192.168.211.241         192.168.211.243

-----------------Many to One--------------------
Target: 192.168.211.239
 Sources:
        192.168.211.233         192.168.211.235         192.168.211.237
Target: 192.168.211.241
 Sources:
        192.168.211.233         192.168.211.235         192.168.211.237
Target: 192.168.211.243
```

```
 Sources:
        192.168.211.233        192.168.211.235        192.168.211.237

-----------------Many to Many--------------------
Source: 192.168.211.233
 Targets:
        192.168.211.239        192.168.211.241        192.168.211.243
Source: 192.168.211.235
 Targets:
        192.168.211.239        192.168.211.241        192.168.211.243
Source: 192.168.211.237
 Targets:
        192.168.211.239        192.168.211.241        192.168.211.243
```

5.4 Other scans

In a similar way, dumps of network traffic were collected for the other four TCP scan types, stealth SYN, stealth FIN, Xmas and Null scan. They were passed through the detector program, indicating the type of scan to be detected. The result was that all the scans performed by us were detected. The results are very similar in to the ones in section 5.3 and are avoided here in the interest of brevity.

5.5 Conclusion

Five TCP portscans were performed by three source machines, on three target machines, with one machine passively collecting all the network traffic. These dumps were then passed to the detector program. All the scans performed and captured in the network data were detected. The scan sweeps were also sorted out into one-to-one, one-to-many, many-to-one, and many-to-many host scans. Similar results were observed when the input to the detector was live network traffic.

Chapter 6

Conclusions and Future Work

An intruder planning an attack on a system typically scans all ports of the system to check open ports, and services running on them. He may then proceed to use known vulnerabilities of the services to gain unauthorized access to the system. Hence, a portscan is a pre-cursor to an attack on the system. A portscan detector is therefore a primary anti-intrusion tool. In our academic setup, many machines have identical configurations, and hence scanning multiple ports across different machines will essentially provide the intruder with the same *complete* data as would scanning a single host. These type of *distributed portscans* are difficult to detect using the currently available IDS tools.

Hence we see that it is very important to detect distributed portscans, by one source host on a multitude of target hosts, and also from a number of source hosts to a number of target hosts. Five types of TCP scans are considered: TCP connect() scan, stealth SYN scan, stealth FIN scan, Xmas scan, and Null scan.

To solve this problem, we analyze network data, instead of collecting data from host logs. The five types of TCP scans to be detected were analyzed and their signatures were studied. A TCP connect() scanner was also developed. Three machines on the LAN were then designated hosts for performing the scans, also called **source machines**. Three machines, **target machines**, were the machines being scanned. One machine was passively sniffing all the data on this broadcast LAN segment.

Packets were captured and grouped into connections. For each of the five scan types being detected, the sequence of packets in a connection was analyzed, and the scan attempts were identified. These were then separated out and entered into the *srcIP* and *tarIP* correlation lists, which were then used to detect the scan sweep type, one of: one-to-one, one-to-many, many-to-one, and many-to-many.

Slow scans and distributed many-to-one and many-to-many hosts scans which go undetected by Snort's portscan preprocessor and portsentry, can be detected by our portscan detector.

The prototype detector currently detects only five types of TCP scans. UDP scans are not considered. Also ICMP probes are ignored. The concept of capturing all the network data works fairly well with broadcast environments created by the use of hubs. Switched environments have only one way of listening to all the ports of a machine - configuring one control port on the switch to mirror the traffic on other ports - which would result in a lot of increased traffic.

The future of Intrusion Detection Systems lies in data correlation. New IDSs will produce results by examining input from several different sources. The way to solve this challenge lies in data mining being performed on different data sets. The concept of a management console dedicated to the task of correlating abnormal event notifications, with relevance measures is an emerging one. One can picturize many distributed elements performing specific jobs, each passing the results onto a higher level for correlation and analysis.

It is expected that the future IDSs will merge all of the independent network components and tools which exist today, into a complete and cooperative system, dedicated to keeping networks stable. There will be many distributed elements performing specific jobs, each passing the results onto a higher level for correlation and analysis. As always, the ultimate authority will be our own judgment.

Bibliography

[Br98] Brecht Claerhout (coder@reptile.rug.ac.be) . Sniffit v0.3.7.beta Packet sniffer and monitoring tool. July 1998. *http://sniffit.rug.ac.be/sniffit/sniffit.html.*

[Axe00] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy, 2000.

[Com95] Douglas E. Comer. *Internetworking with TCP/IP, Vol I: Principles, Protocols, and Architecture.* Prentice-Hall Inc., 1995. ISBN 81-203-1053-5.

[Cra99] Craig H. Rowland (crowland@psionic.com) . Psionic Portsentry v1.0, Port scan detection and active defense system. November 1999. *http://www.psionic.com/abacus/portsentry/.*

[Fyo00] Fyodor (fyodor@insecure.org). Nmap v2.53 stealth port scanner. May 2000. *http://www.insecure.org/nmap/.*

[Mar01] Martin Roesch (roesch@clark.net). Snort v1.7, open source network intrusion detection system. January 2001. *http://www.snort.org/.*

[Ste01] Steve McCanne, Craig Leres and Van Jacobson. Tcpdump v3.6.2, LBNL's protocol packet capture and dumper program. February 2001. *http://www.tcpdump.org/.*