Contents

List of Figures

Abstract

Intrusion detection systems are a mechanism by which attacks against Internet hosts can be detected, and in some cases, thwarted. In this work, we create a taxonomy of intrusion detection systems into three major classes, and describe the capabilities and limitations of each class.

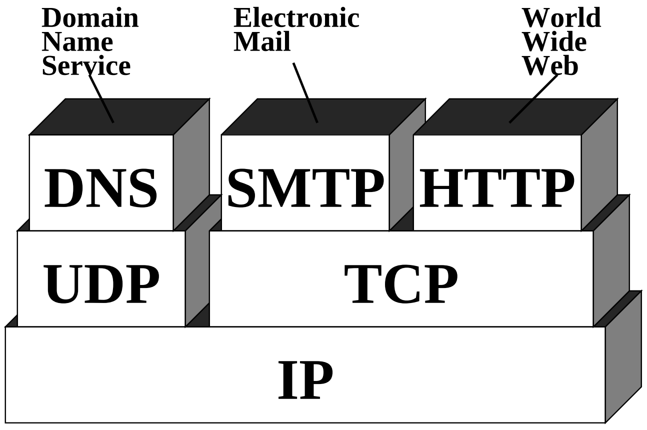
We also describe `ipse`, a general tool for monitoring networks, and show how to utilize it to build intrusion detection systems. Finally, we describe some of the other functionalities a tool such as `ipse` can provide.

Chapter 1

Introduction and Motivation

The Internet is a public packet-switched network. Machines connected to the Internet are called *hosts*, and for the most part, any host can send chunks of data, called *packets*, onto the network, destined for any other host. This is the method by which hosts on the Internet communicate. The Internet Protocol, or IP [7], is the standard language for the packets that flow across the Internet, and this standard is what enables computers all over the world to effectively communicate with each other. All Internet services, including the World Wide Web, electronic mail, domain name service, etc., utilize protocols that are built on top of the standard Internet Protocol.

Figure 1: Internet services are all built on top of the standard Internet Protocol (IP)



An important property of IP packets is that they contain no strong authentication informa-

corollary of this property is that hosts usually cannot distinguish between packets sent by an authorized user, and packets sent by an unauthorized intruder: a *cracker*.[1]

Hosts on the Internet are subject to a variety of attacks from crackers whose goals range from simple vandalism of web pages, to denial of service attacks that prevent machines from communicating via the Internet, to infiltration of machines in order to steal sensitive information [29]. These attacks generally arise due to problems with the design [12, 33], use [14, 40], or implementation [24, 25, 26, 19, 23] of the network services provided or used by these hosts.

In order to protect ourselves from the crackers, we monitor our networks for indications of a cracker *intrusion*. It may seem that this approach is too "reactive", and that fixing the problems that the crackers are exploiting would be a better way to go. If we *could* have a perfectly secure system while still being connected to the Internet, that would be ideal; unfortunately, new problems are discovered all the time, so fixing them all is a very large perpetual task. In addition, if your system comes from a commercial vendor, it is likely that you cannot fix it *at all* without the vendor's help. If the vendor is slow or uncooperative, you may be out of luck. In any case, monitoring adds an extra layer of protection to any existing network, and, as we will see below, can be realized without any changes to the existing hosts.

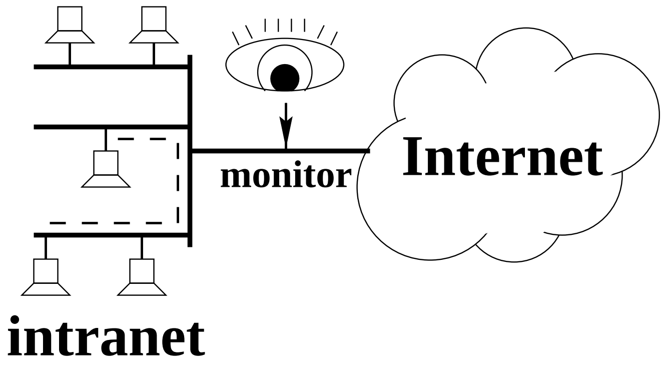
We apply the principle of *orthogonal security*: distinguish those portions of your system that you wish to keep secure from those portions that are untrusted, and carefully monitor the connections between them. In this case, we are going to monitor the portion of the network that connects the machines that we wish to keep secure, from all others. The former set may include just a single important machine (a Kerberos server [41], for example), in which case we can simply monitor the network to which it is directly attached, or it may be as large as an entire institution's intranet, in which case we monitor the connections between the intranet and the rest of the world.

We call this set of hosts the *trusted network*. Note that there is a distinction between wanting to keep a certain host secure on the one hand, and trusting it on the other. Unfortunately, with some network topologies, such as the one in Figure 2, monitoring the link between a corporate intranet and the rest of the world will miss attacks performed *by* one of the hosts on the intranet against another. Note that in the past, technologies such as Javascript and Java have been used

[1]Please refrain from using the term "hacker" to mean "someone who breaks into computer systems". "Cracker" is the appropriate term.

to bypass intrusion detection in just this way, and to cause one machine on the “inside” to attack another. If you do not actually trust the machines you are protecting, you should monitor the connections among them, as well as the connections between them and the rest of the world. Following the principle of orthogonal security, monitoring becomes easier if the trusted network is physically separate from the untrusted network, with only a few (possibly one) links between them; these links are the ones to monitor.

Figure 2: Monitoring the connection between a corporate intranet and the rest of the world



t of hosts that are not trusted form the *untrusted network.* This set will, in general, of the Internet outside of your administrative control, and, as mentioned above, : your control. If in doubt, it is of course safer to put hosts in the untrusted network d one.

em which attempts to accomplish this monitoring goal is commonly called an *intru-stem*, or IDS. There are a number of intrusion detection systems currently available, rch world and commercially. The most recent example of a research system is Vern 3]. The major contribution of Bro is its scripting language used to specify the be-onitor: this adds simplicity to the potentially complex job of monitor configuration, e of flexibility and possible performance.

ncreteness, we first describe our own tool, `ipse`, which can easily be used to .

Chapter 2

The Design of `ipse`

The Internet Protocol Scanning Engine, or `ipse`, is a general tool than can be used to build a number of different kinds of useful tools for examining IP traffic. In this chapter we discuss the design of `ipse`, and describe how to use it to construct an intrusion detection system. We also describe some other uses for this tool.

`ipse` is composed of two main pieces:

- loadable **modules**, which contain code to handle “interesting” packets
- a simple **framework**, which demultiplexes network packets to the various modules.

2.1 Modules

The first, and most important, part of `ipse` is its ability to use and stack *loadable modules*, of which there are two main types: *filter* modules and *policy* modules.

Filter modules organize incoming packets into logical groups, and perform other actions to organize the flood of incoming information. For example, the TCP filter module accepts individual TCP/IP packets and outputs a series of separate *streams*, each of which corresponds to a single TCP/IP connection. Similarly, the NFS filter module accepts UDP packets to or from port 2049, and outputs separate streams corresponding to transactions with different NFS servers.

Filter modules can be stacked, as diagrammed in Figure 3: the telnet filter module, for example, takes the output of the above TCP filter module, and removes telnet IAC control sequences from TCP streams corresponding to telnet sessions.

A more interesting example: a filter module that performs NFS spoofing detection takes the streams that were output by the above NFS filter module, and removes NFS requests and replies from it, unless some request had two *different* replies (the reasons behind this will be discussed in Chapter 3). The output of the stacked filter module is a set of streams containing only “suspicious” replies and the requests that initiated them.

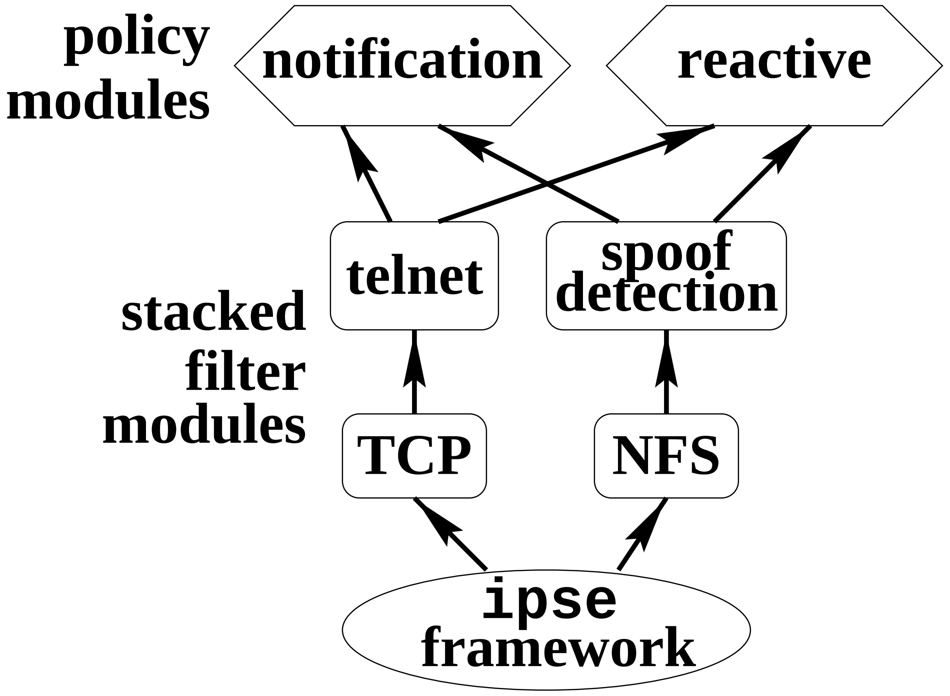
As can be seen in the above examples, filter modules not only organize incoming information, but also remove information they deem to be “uninteresting”. In the examples above, the TCP filter would discard packet retransmits (though a different module specifically watching for TCP spoofing would certainly not!), the telnet filter discards IAC commands, and the NFS spoofing filter discards “normal” NFS requests and replies.

Policy modules collect the output of the chains of filter modules, and interact with the environment, based on their contents.

The simplest kind of policy module is a *notification module.* Notification modules simply make the environment aware that some anomalous event has occurred, but do not make any attempt to stop it. Actions that could be taken by notification policy modules include simply logging the output of the NFS spoofing filter, or sending an email page to the sysadmin if they detect an attack in progress.

A more complex kind of policy module is the *reactive module.* Reactive modules take some action to deal with events they notice; this can be thought of as an “in-band” response, as opposed to the “out-of-band” response provided by notification modules. An example of a reactive module is one that actively tries to foil a SYN flood in progress (SYN floods and methods to foil them will be described in Chapter 3).

Figure 3: Stacked modules in `ipse`



2.2 Framework

The second part of `ipse` is a simple framework that organizes the modules, and arranges for network packets to be read, filtered, and demultiplexed to the various modules.

Modules receive their packets by supplying a packet filter and a callback routine to the framework. The framework watches packets on a network interface, and ignores the ones that satisfy none of the packet filters supplied by the user-specified modules. As well, the user can optionally give an additional packet filter directly to the framework if he is only interested in a subclass of the packets that a module would otherwise handle.

For example, the WWW log-gathering module specifies a packet filter for packets to or from TCP port 80. Running just this module would cause the framework to handle every such packet that goes by its network. However, if you were only interested in, for example, outgoing requests from your network, and not incoming WWW requests to your server, you could specify an additional packet filter that would remove packets to or from your server's port 80.

When the framework receives a filtered packet, it is passed to each of the modules whose packet filter accepts it.

2.3 Building an intrusion detection system with `ipse`

Given a tool like `ipse`, building an intrusion detection system on top of it is quite straightforward. The hard part, of course, is deciding what kinds of intrusions to detect. As a general security policy, you should choose a *restrictive* model that knows about certain "good" behaviours, and flags anything else as suspicious. This is in contrast to a *permissive* model, which only flags certain predefined "bad" behaviours. The main benefit of a restrictive model is that it is more likely to detect new, unanticipated forms of intrusion than is a permissive model. The downside of the restrictive model is that there is possibly a high administrative overhead involved in coming to a consensus of what you and your users think is "good" (especially for a large user base).

Exactly what constitutes "good" behaviour is naturally dependent on your situation. As a start, figure out what services you intend your trusted network to be offering to the world, and what Internet services the users of the trusted network intend to use. For instance, you may decide to allow your internal DNS server to make DNS queries of the outside world, incoming HTTP connections to your web server, SMTP connections to and from your mail server, outgoing HTTP connections to anywhere, and incoming ssh (secure shell) connections from your employees' homes.

Once you know that information, construct an `ipse` module with a packet filter that is the negation of the acceptable traffic. The policy as to what action to perform when bad packets are observed is left up to you; as will be seen below, a wide variety of responses are possible, ranging from simple logging to all-out counterattack.

Once you detect all unexpected packets, you may then wish to impose some additional checks on the "good" traffic. For example, you may wish to add a module to watch the DNS traffic for mismatched responses, to detect SYN flooding attacks on your web server, or to watch the incoming HTTP connections for CGI exploit attempts. These attacks, and others, are described in more detail in the following chapter.

2.4 Other uses of `ipse`

A general network monitoring tool such as `ipse` can be used for more than creating intrusion detection systems. In this chapter, we list some other examples.

2.4.1 Phantom loghost

In addition to creating intrusion detection systems, another security-related use of `ipse` is to create a *phantom loghost.* When crackers infiltrate a network, one thing they tend to do is to try to destroy or alter any system logs they can find.

A common defense against this is to have a separate *loghost.* All hosts send their logs over the network to the loghost, which records them. The idea is that the loghost should be kept especially secure, but naturally, the loghost presents a tempting target for a cracker.

With `ipse`, you can set up a loghost on a machine with no IP address. This phantom loghost can simply sniff the log messages off the network, and record them, but a cracker has no way to send packets to the machine in an attempt to destroy the logs.

2.4.2 Network traces

In [31], the author reports using `ipse` to gather *network traces* of HTTP traffic at UC Berkeley: the author uses a module that reconstructs the contents of HTTP requests and responses from the packets coming in and going out over UC Berkeley's dial-in modem lines in order to determine characteristics of World Wide Web usage. Unlike other attempts to gather similar traces, no web clients, proxies, or servers needed to be instrumented to obtain the data; the information was simply read off the network wire as it went by.

2.4.3 Improving TCP behaviour

Because TCP interprets packet loss as congestion, TCP connections over lossy links (including wireless links) tend to have poor performance. In this situation, `ipse` can be used in a manner similar to [5]; an `ipse` module can watch for TCP packets that are not acknowledged, and, since it knows about the lossy link, can retransmit the packets much more effectively than can the originating host (which thinks it is transmitting over an extremely congested link).

Chapter 3

The Power of Intrusion Detection Systems

The goal of this chapter is to analyse the classes of attacks and possible defenses that an intrusion detection system can offer. We stratify the general class of intrusion detection systems by the capabilities they possess:

Class I: a system that can merely read packets from a network

Class II: a system that can also inject (write new packets) into a network

Class III: a system that can not only read and inject packets, but can also modify or delete existing packets

We discuss below the capabilities and limitations of these three classes of IDS. In our case, `ipse` is usually in Class II: it can write, as well as read packets. However, as discussed below, `ipse` can be configured to run *on* a router or gateway in order to provide a system that is in Class III. We call this configuration “router `ipse`”.

3.1 Class I

We will first examine the various types of intrusion attempts that can be detected by an IDS that only has the ability to *read* packets crossing the boundary between the trusted and untrusted networks. As we will see, even these simple systems (generally called “packet sniffers”) can detect a large class of potential intrusions.

We divide packet sniffers into three classes, according to the amount of *state* they use: none, a small amount, or a large amount. The amount of state used by a packet sniffer affects how scalable the system is; if no state is needed, simply running a number of independent sniffers, each watching packets with certain (fixed) addresses may be sufficient. If state is kept, it is important to partition the work to the various sniffers so that all packets that may be relevant to some piece of state are gathered by the same sniffer.

3.1.1 No state

We first examine the kinds of intrusions that can be detected by a very simple packet sniffer with no state.

Unauthorized network connections: The simplest job for a packet sniffer is to spot packets from or destined to network locations or services that are not authorized under, for example, a corporate security or computer usage policy. Examples of unauthorized connections that could be detected include Internet Relay Chat services [37], network games, remote logins from or to unauthorized sites, remote printing, or access to certain kinds of web sites.

Note that connectionless services (those that use UDP or multicast, for example) may be harder to block, as the source address of a UDP packet can trivially be forged (this is not to say that the source address of TCP packets can not be forged, however).

IP header games: A large class of recent network attacks [22, 28, 29] exploited bugs in various operating systems’ IP fragment reassembly code, or other code used to handle IP headers that appear infrequently, or appear with illegal values.

A packet sniffer can detect when unusual or extraordinary IP headers appear on packets. These can include suspicious or illegal fragmentation (extremely short fragments, or fragments exceeding the claimed length of the packet), IP source routing, large ping packets, and so on.

It should be noted that some classes of attacks against intrusion detection systems given in [39] can themselves be detected simply by flagging unusual packet fragmentation.

3.1.2 Small state

In order to detect the above kinds of problems, the packet sniffer involved need not keep state; it could examine each arriving packet separately and make independent determinations of whether the packet was “interesting”. In order to detect the following kinds of attacks, the packet sniffer needs to keep some state in order to correlate packets that arrive at different times. The amount of state needed in these cases is small; one kilobyte of state per host on the trusted network is usually more than enough.

SYN flooding: In a SYN flooding attack, the attacker sends a TCP SYN packet to the target host (which responds with a SYN/ACK packet), but never sends the ACK packet that would complete the TCP three-way handshake [6, 19]. While the target host is waiting for the response, some of its resources (typically entries in a small table of outstanding handshakes) are consumed. The attacker proceeds by sending a very large number of these “naked SYNs” and causes the target host to be unable to accept any further network connections.

A SYN flood can be detected by a packet sniffer that keeps limited state: the number of outstanding SYNs (possibly a moving average) to each host on the trusted network. If this count gets too high, an admin can be quickly notified. This is important, because generally SYN floods can only be traced to their attacker *while they are in progress*.

False server: This kind of attack occurs when an attacker sees a request for some kind of RPC or remote database query over a network, and returns an incorrect result more quickly than the real server returns the correct result. This attack is most commonly used against NFS [40], where the attacker watches for queries for the permission bits on executable files, and maliciously returns the result “this program is setuid root” before the real server has a chance to respond. This attack often allows a non-root user on the target machine to elevate himself to root privileges. It is also used against the DNS distributed database [35], where the attacker causes the target to be misinformed about the mapping of IP addresses to domains, thus defeating some domain-based access controls.

These attacks can be detected simply by watching for two *different* responses to the same query. The state that needs to be kept by the packet sniffer is that of what queries have been answered, a timestamp, and a secure hash of the reply.

In order for such an attack to succeed, the attacker must respond *before* the real server does, so the second reply (that of the real server) should appear no longer after the first reply than the server takes to respond to a query. Therefore, if it is known how long it takes for the real server to respond to a query, state entries can be discarded after this amount of time has elapsed.

A *secure* hash is used, rather than a simple checksum, so that the attacker, knowing the hash algorithm, cannot construct a false reply tailored so as to have the same checksum as the real reply.

3.1.3 Large state

If the packet sniffer has the ability to store quite a bit more state (on the order of the number of active TCP connections), there are more attacks it can detect. Now we will assume that the sniffer has the ability to reconstruct the TCP stream contained in the data flow. See [39] for caveats to this, but see below for a possible fix. Once the sniffer has the ability to read the data in TCP connections, it can watch for events at the *application layer*. Some common examples are the following:

Suspicious logins: Why is user `fred` logging in to different machines on the protected network, *from* different machines around the Internet? Why is `root` logging in from a machine in the Netherlands? Perhaps their passwords are circulating among the cracker circles.

Weak passwords: By watching the login session, the sniffer may be able to determine users' passwords.[1] If the local security policy is to allow only "strong" passwords, or better, not allow unencrypted login sessions at all, the sniffer can detect a violation.

Mail fraud: A sniffer can detect intrusion attempts from subscribers to the `sendmail` bug-of-the-month club [1, 2, 3, 4, 8, 9, 10, 11, 13, 15, 16, 17, 18, 20, 21, 24]. Older (and even newer, unfortunately) versions of mail daemons, and `sendmail` in particular, are infamous for having bugs exploitable by sending malconstructed email messages to vulnerable hosts.

CGI exploits: Common Gateway Interface [36] programs run on a web server in order to execute complicated tasks in response to a query. Unfortunately, unless written very carefully, these programs can allow remote users full access to the web server machine. The first well-publicized exploit of this was the "phf" vulnerability [27], which occurred because a CGI program called "phf", which *shipped with* the most popular web server software at the time, contained such a vulnerability, and, once known in the cracker community, made breaking into web server hosts very easy. A sniffer that can watch traffic to your web server can detect this and other related intrusion attempts.

3.2 Class II

If an IDS can, in addition to simply *watching* the packets flow between the trusted and untrusted networks, also *inject* new packets of its own construction, a number of new options are open to it. In addition to simply *detecting* intrusion attempts, and sounding alarms, it may attempt to take more proactive steps.

Below we describe some of the actions that can be taken by such an IDS, called an "active sniffer".

Terminate unauthorized network connections: If an unauthorized network connection is detected, an active sniffer has the option of terminating it entirely, by sending a RST (reset) packet to

[1] Some would say this itself is a good reason to run the sniffer, but they're probably wearing the wrong colour hat.

one or both endpoints.

One recommended way to set up such a system (with a small user base) would be to initially reset every connection that is not "obviously good" (connections to port 80 of the web server, or port 25 of the mail server, for example), and then to add connections that should be allowed as users complain that they can't do something in particular. "Restrictive" systems such as these tend to protect much better against unanticipated attacks than do "permissive" systems that allow everything except that which is "known bad".

Foil some network attacks: If an active sniffer notices a SYN flood in progress, not only can it notify an administrator, so that a trace can be undertaken, but it can attempt to *foil* the attack entirely. It does this by sending RST packets to the target of the SYN flood, one for each naked SYN that was received. This forcefully resets each half-open connection, and frees the target's resources, allowing it to accept connections from clients that complete the three-way handshake.

Minimize or correct damage: Some attacks cannot be foiled by resetting the intruding connection, often because they are not connection-based in the first place. Intrusions based on UDP or multicast are examples, as are some of the IP header games.

Instead of foiling the attack, an active sniffer can try to *limit* the damage done, or even correct it entirely. An example would be where the active sniffer detects a *routing attack*, in which the attacker disrupts the information stored in routers in order to either cause a denial of service, or to cause more packets to be routed to himself, so that he can read all the traffic, or perform other classes of attacks.

In this case, the active sniffer, noticing bizarre routing information arriving, would not have the opportunity to prevent that information from affecting the router's state. However, it may be able to send the correct information (say, a copy of the last "good" routing information it saw), in order to correct the state of the router and prevent the denial of service of the attack.

Gather intelligence or counterattack: This is the most interesting class of actions that an active sniffer can perform. Here, when the active sniffer detects an intrusion in progress, it can try to gather information on the attacker, by using the `finger`, `rwho`, or `whois` protocols, among others. Examples of this kind of intelligence gathering are given in [30] and [42].

A more aggressive IDS may even decide to *counterattack*: perform active action against the attacker's machine, in order to stop the attack.[2] An example of this (though with a fairly tame counterattack) is Fred Cohen's `all.net` site, which (until recently), upon receiving an incoming `telnet` connection, would send nasty email to the administrator of the originating machine, threatening all manner of legal action.

It is important to beware of "fingerstorm"-like effects, in which each side's intelligence gathering or counterattack measures cause the other's to react. The effect is so named because the most obvious occurrence is to use the `finger` protocol to identify a remote intruder who is `finger`ing one of your machines [30]. If the intruder is running a similar system, badness will happen unless the potential fingerstorm is detected.

3.3 Class III

If your intrusion detection system is running *on* a router or a gateway, and if it can examine, and potentially modify or delete, incoming packets before forwarding them, as well as injecting packets it constructs itself, you can achieve even stronger results. Unfortunately, in order to accomplish this, actual changes to the network infrastructure may need to be done. In contrast, sniffers and active sniffers merely need to be attached as a node to an existing network in order to protect it.

Below we describe some of the advantages of a Class III IDS.

Prevention of some attacks: Just as a Class II IDS can foil some kinds of attacks by resetting the intruding connection, a Class III IDS can prevent the attack simply by deleting the packets. This prevention is also effective in the cases where a Class II IDS is only able to minimize damage after the fact (see above).

A Class III IDS running on a gateway machine effectively combines the roles of an intrusion detection system and a firewall.

[2]Be careful! This could be a very sticky legal situation.

Packet assembly and reordering: The authors of [39] point out that many intrusion detection systems have a common flaw if they need to understand application layer information over a TCP connection: the IDS has no way of knowing how the various hosts it is protecting will perform operations such packet reassembly, packet reordering, resolution of overlapping TCP segments, packet timeouts, etc. Because different operating systems display different behaviours for some of these operations, the authors claim that the IDS watching the network can not accurately know the state or contents of the connection, as each protected host sees it.

A Class III IDS has a way around this problem: if the IDS itself reassembles and reorders packets arriving from the untrusted network, it can present to the hosts on the trusted network a very simple view of the TCP stream, that all hosts are likely to understand in the same way. In addition, the IDS will see the same data as the hosts see, so that it can effectively spot and squish attempted intrusions.

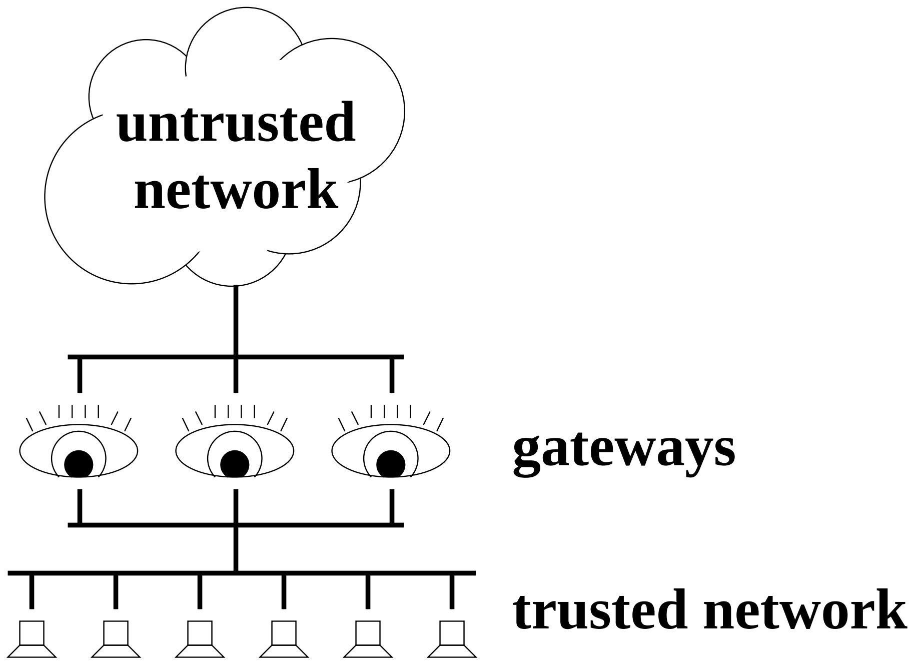
If the overhead associated with reordering packets leads to a bottleneck at the IDS, the system can be easily scaled by having multiple gateways, each running a Class III IDS, arranged in parallel, as in Figure 4. Each IDS is configured to assemble, reorder, and pass on packets to a subset of the destination hosts and ports, and to discard packets not to a member of its subset. Of course, every destination host and port should be handled by exactly one IDS. More discussion of scalability will be presented in the next chapter.

3.4 Caveats

Close attention must be placed to the failure modes of an intrusion detection system. If the IDS believes it has detected an intrusion, when none has actually occurred (a *false positive*), the results will vary greatly, depending on the action the IDS takes. If the IDS simply logs an unnecessary entry or causes an unnecessary notification, the only ill effect would be wasted administrative time. On the other hand, if the IDS decides to reset an active (innocuous) connection, or to counterattack a friendly site, the effects could obviously be much worse.

It is also important to remember what classes of behaviour an IDS *cannot* detect. As men-

Figure 4: Scaling Class III intrusion detection systems



oned earlier, attacks occurring entirely *within* the trusted network (or entirely outside, of course) ould not be detected by a system monitoring the network bridging the trusted and untrusted reions. As well, an IDS cannot reliably detect *tunneling*, wherein information is leaked from a host ıside of an innocuous protocol (often HTTP). These are not failures of the system, however, beause they require that a trusted host already be compromised, or be colluding with the outside. ' your only defense is an intrusion detection system, then a trusted host compromised once could ause a large cascade of problems.

By using the principle of orthogonal security, explained above, it is easy to provide deense in depth: multiple, orthogonal, layers of security that can back each other up in the event of failure. In particular, an intrusion detection system is no substitute for solid host security, and ossibly a separate firewall as well. A robust security system such as this can mitigate the effects of ne piece not noticing an attack in progress (a *false negative*).

Chapter 4

`ipse` Performance

4.1 Implementation

The framework for `ipse` is built on top of `libpcap`, a portable library for user-level packet capture from Lawrence Berkeley National Laboratory. Using `libpcap` allows for two major benefits:

Portability: the methods used to promiscuously access raw packets on a network interface vary widely from operating system to operating system. The `libpcap` library allows uniform access across platforms to this functionality, providing much greater ease of portability.

BPF support: the BSD Packet Filter [34] is a standard mechanism implemented in a number of operating systems that allows an application to instruct the kernel to pass only a particular subset of the incoming network packets across the kernel-user boundary (thus saving unnecessary context switches). The `libpcap` library will use the BPF on those operating systems in which it is available, and emulate it in the user level (so as to maintain portability) on those in which it is not.

Other common network applications, such as tcpdump [32], use `libpcap` to achieve the same goals.

The standard `libpcap` library is useful to create a Class I intrusion detection system. In order that `ipse` may be used as a Class II or a Class III IDS, we additionally made some modifications to `libpcap` that add the capability to inject arbitrary packets onto the network, in addition to promiscuously reading the packets that arrive.

The current version of `ipse` runs on the Linux and Solaris operating systems. This allows cheap, commodity PC's to easily be adapted into intrusion detection systems, and placed in strategic locations on a network.

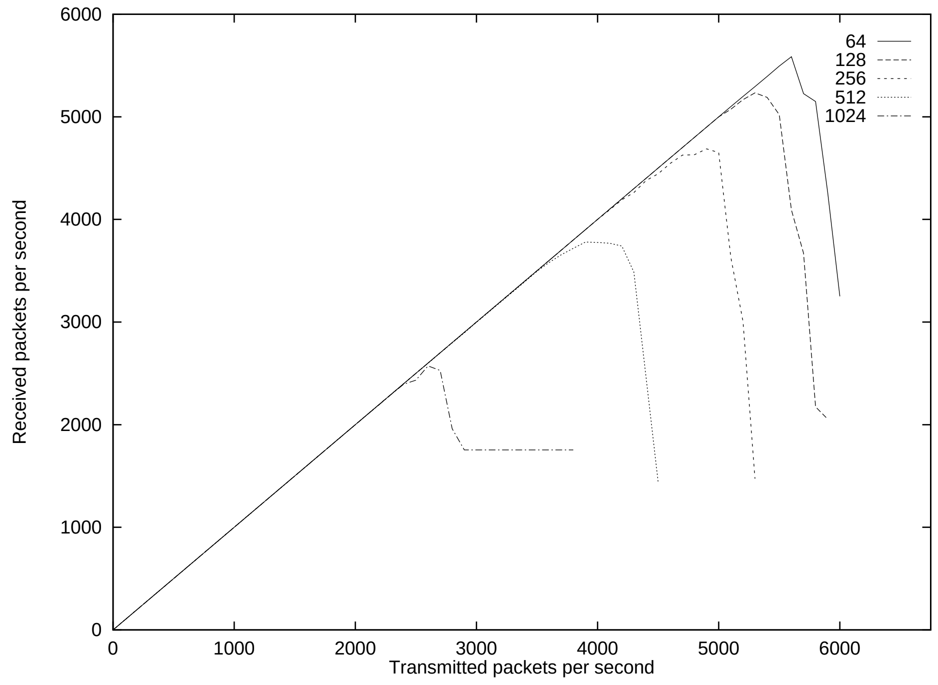
4.2 Measurement

The goal of any intrusion detection system is, of course, to detect intrusions. It is therefore important that the system not get overloaded and end up missing packets (an attacker may *intentionally* overload an IDS in an attempt to slip an attack through, for example). Because of this, we must know what kind of performance we can expect.

For our experiments, we used a normal PC (Pentium Pro 200; nothing special by today's standards) running `ipse` and the Linux operating system. We sent UDP packets of various sizes and at various rates over the network and had an `ipse` module perform the task of counting them. (The packet generator was running on an identically-configured machine, connected by a 100 Mbps ethernet.) In this way, we graph the average rate of reception of the packets against the transmission rate, for each size of packet from 64 bytes to 1024 bytes. The result is shown in Figure 5.

We examine the graph to find the maximum throughput (packet rate times packet size) for which this configuration of `ipse` receives every transmitted packet. We find that the worst case occurs when small packets are being sent very quickly, thus causing the operating system to perform many context switches (note that we asked `ipse` to ask the BPF for *every* packet: a somewhat pessimal situation). In this case, we can read 64-byte packets at a rate of 5000 packets per second without dropping any, for a throughput of 2.5 Mbps. This is certainly fast enough to watch a modem bank or a T1 connection, but not enough for an ethernet (though note that with a mix of large and small packets, we *can* handle over 10 Mbps without packet drops).

Figure 5: Packet reception rates for varying sizes of packets (in bytes) on a commodity Linux PC



4.3 Scalability

How, then, do we handle connections with higher throughput? If we are talking about a single, high-throughput connection (such as a T3), we can simply arrange independent intrusion detection systems in parallel, as in Figure 4. In this case, the parallelism is simple, because the different intrusion detection systems never need to send state to each other.

A more complex situation occurs when the connection between the trusted network and the untrusted network consists of *more than one* separate link, and especially when they are "inverse multiplexed"; that is, packets between two given hosts need not all traverse the same one of these links.

In this case, there are a number of options. The first option is to allow for the exchange of *inter-IDS state* between the intrusion detection systems. Normally this would consist of an extra network containing only the intrusion detection systems, on which they pass around data that will allow all packets that are part of a given network connection to be handled on one node. This is obviously a very complicated situation.

The better option is to try to find another “cut” in the network topology that separates the trusted network from the untrusted; you could move the intrusion detection systems closer to the hosts on the trusted net, for example. By doing this, there are more places in the network to watch, but no place needs to share state with any other place. This may require more intrusion detection systems (though some of them may be able to be placed on more than one network), but they are greatly simplified, since the inter-IDS state is eliminated. Remember also that they can be cheap PC’s running free software.

In these ways, you should be able to arrange one or more PC’s running `ipse` into effective intrusion detection systems to monitor higher-bandwidth connections to larger trusted networks.

Chapter 5

Conclusion

Intrusion detection systems can be divided into three major classes:

Class I: systems that can merely read packets from a network

Class II: systems that can also inject packets into a network

Class III: systems that can not only read and inject packets, but can also modify or delete existing packets

In this work, we have described the capabilities and limitations of each class of IDS, and shown how different classes of IDS can be used to detect, ameliorate, or prevent certain kinds of network intrusion by crackers.

We have also described `ipse`, a general tool for monitoring networks, and shown how any class of IDS can be built with it. In addition, we have demonstrated some other useful applications of this tool.

No security system should rely on only one kind of protection. The principle of *orthogonal security* can be used to provide defense in depth: what one layer misses, another may catch. An effective combination of intrusion detection, host security, and well-stated policies is your best hope for a secure network.

Bibliography

[1] [8lgm]-Advisory-16.UNIX.sendmail-6-Dec-1994, December 1994.

[2] [8lgm]-Advisory-17.UNIX.sendmailV5-2-May-1995, May 1995.

[3] [8lgm]-Advisory-17.UNIX.sendmailV5.22-Aug-1995, August 1995.

[4] [8lgm]-Advisory-20.UNIX.sendmailV5.1-Aug-1995, August 1995.

[5] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proc. of the 15th International Conference on Distributed Computing Systems*, May 1995.

[6] V. G. Cerf, Y. K. Dalal, and C. A. Sunshine. Specification of internet transmission control program. RFC 675.

[7] Vint Cerf and Bob Kahn. A protocol for packet network intercommunication. *IEEE Trans. Comm.*, 22(5), May 1974.

[8] CERT advisory CA-88.01. `http://www.cert.org/advisories/CA-88.01.ftpd.hole.html`.

[9] CERT advisory CA-90.01.

[10] CERT advisory CA-93.15. `http://www.cert.org/advisories/CA-93.15.SunOS.and.Solaris.vulnerabilities.html`.

[11] CERT advisory CA-93.16.

[12] CERT advisory CA-94.01. http://www.cert.org/advisories/CA-94.01.ongoing.network.monitoring.attacks.html.

[13] CERT advisory CA-94.12.

[14] CERT advisory CA-94.15. http://www.cert.org/advisories/CA-94.15.NFS.Vulnerabilities.html.

[15] CERT advisory CA-95.05.

[16] CERT advisory CA-95.08. http://www.cert.org/advisories/CA-95.08.sendmail.v.5.vulnerability.html.

[17] CERT advisory CA-95.11.

[18] CERT advisory CA-96.20. http://www.cert.org/advisories/CA-96.20.sendmail_vul.html.

[19] CERT advisory CA-96.21. http://www.cert.org/advisories/CA-96.21.tcp_syn_flooding.html.

[20] CERT advisory CA-96.24. http://www.cert.org/advisories/CA-96.24.sendmail.daemon.mode.html.

[21] CERT advisory CA-96.25. http://www.cert.org/advisories/CA-96.25.sendmail_groups.html.

[22] CERT advisory CA-96.26. http://www.cert.org/advisories/CA-96.26.ping.html.

[23] CERT advisory CA-97.04. http://www.cert.org/advisories/CA-97.04.talkd.html.

[24] CERT advisory CA-97.05. http://www.cert.org/advisories/CA-97.05.sendmail.html.

[25] CERT advisory CA-97.23. `http://www.cert.org/advisories/CA-97.23.rdist.html`.

[26] CERT advisory CA-97.24. `http://www.cert.org/advisories/CA-97.24.Count_cgi.html`.

[27] CERT advisory CA-97.25. `http://www.cert.org/advisories/CA-97.25.CGI_metachar.html`.

[28] CERT advisory CA-97.28. `http://www.cert.org/advisories/CA-97.28.Teardrop_Land.html`.

[29] CERT summaries. `ftp://ftp.cert.org/pub/cert_summaries/`.

[30] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[31] Steven D. Gribble and Eric A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.

[32] V. Jacobson, C. Leres, and S. McCanne. The tcpdump manual page. Lawrence Berkeley Laboritory, June 1989.

[33] Brian Kantor. BSD rlogin. RFC 1258.

[34] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter 1993 USENIX Conference*, January 1993.

[35] P. Mockapetris and K. Dunlap. Development of the domain name system. In *Proc. ACM SIGCOMM '88 Symposium.* ACM Computer Communications Review, August 1988.

[36] NCSA. The common gateway interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`.

[37] Jarkko Oikarinen and Darren Reed. Internet relay chat protocol. RFC 1459.

[38] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. USENIX Security Conference*, January 1998.

[39] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. `http://www.secnet.com/papers/ids-html/`, January 1998.

[40] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings*, Berkeley, CA, Summer 1985. Usenix Association.

[41] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proc. USENIX Winter Conference*, February 1988.

[42] Wietse Venema. TCP wrapper: Network monitoring, access control, and booby traps. In *Proc. USENIX UNIX Security Symposium*, September 1992.