Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection

Thomas H. Ptacek
tqbf@securenetworks.com

Timothy N. Newsham
newsham@securenetworks.com

Secure Networks, Inc.

January, 1998

"Not everything that is counted counts, and not everything that counts can be counted."
Albert Einstein

"...yes, a game where people throw ducks at balloons, and nothing is what it seems..."
Homer J. Simpson

Abstract

All currently available network intrusion detection (ID) systems rely upon a mechanism of data collection—passive protocol analysis—which is fundamentally flawed. In passive protocol analysis, the intrusion detection system (IDS) unobtrusively watches all traffic on the network, and scrutinizes it for patterns of suspicious activity. We outline in this paper two basic problems with the reliability of passive protocol analysis: (1) there isn't enough information on the wire on which to base conclusions about what is actually happening on networked machines, and (2) the fact that the system is passive makes it inherently "fail-open," meaning that a compromise in the availability of the IDS doesn't compromise the availability of the network. We define three classes of attacks which exploit these fundamental problems—insertion, evasion, and denial of service attacks—and describe how to apply these three types of attacks to IP and TCP protocol analysis. We present the results of tests of the efficacy of our attacks against four of the most popular network intrusion detection systems on the market. All of the ID systems tested were found to be vulnerable to each of our attacks. This indicates that network ID systems cannot be fully trusted until they are fundamentally redesigned.

1 Introduction

Intrusion detection is a security technology that attempts to identify and isolate "intrusions" against computer systems. Different ID systems have differing classifications of "intrusion"; a system attempting to detect attacks against web servers might consider only malicious HTTP requests, while a system intended to monitor dynamic routing protocols might only consider RIP spoofing. Regardless, all ID systems share a general definition of "intrusion" as an unauthorized usage of or misuse of a computer system.

Intrusion detection is an important component of a security system, and it complements other security technologies. By providing information to site administration, ID allows not only for the detection of attacks explicitly addressed by other security components (such as firewalls and service wrappers), but also attempts to provide notification of new attacks unforeseen by other components.

Intrusion detection systems also provide forensic information that potentially allow organizations to discover the origins of an attack. In this manner, ID systems attempt to make attackers more accountable for their actions, and, to some extent, act as a deterrent to future attacks.

1.1 The CIDF Model of Intrusion Detection Systems

There are many different ID systems deployed world-wide, and almost as many different designs for them. Because there are so many different ID systems, it helps to have a model within which to consider all of them. The Common Intrusion Detection Framework (CIDF)[1] defines a set of components that together define an intrusion detection system. These components include event generators ("E-boxes"), analysis engines ("A-boxes"), storage mechanisms ("D-boxes"), and even countermeasures ("C-boxes"). A CIDF component can be a software package in and of itself, or part of a larger system. Figure 1 shows the manner in which each of these components relate.

The purpose of an E-box is to provide information about events to the rest of the system. An "event" can be complex, or it can be a low-level network protocol occurrence. It need not be evidence of an intrusion in and of itself. E-boxes are the sensory organs of a complete IDS— without E-box inputs, an intrusion detection system has no information from which to make conclusions about security events.

A-boxes analyze input from event generators. A large portion of intrusion detection research goes into creating new ways to analyze event streams to extract relevant information, and a number of different approaches have been studied. Event analysis techniques based on statistical anomaly detection[2], graph analysis[3], and even biological immune system models[4] have been proposed.

E-boxes and A-boxes can produce large quantities of data. This information must be made available to the system's operators if it is to be of any use. The D-box component of an IDS defines the means used to store security information and make it available at a later time.

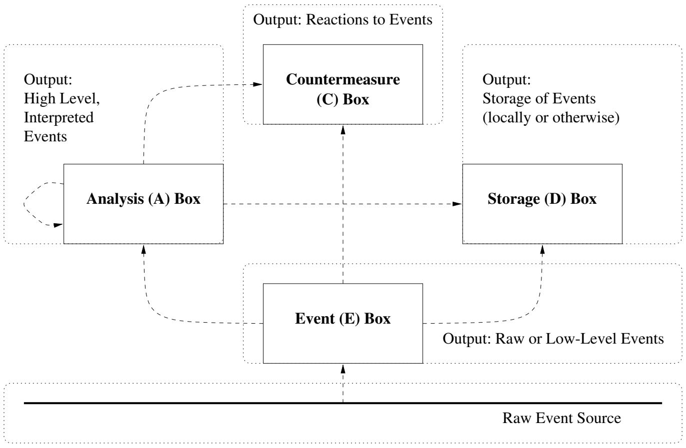

Figure 1: CIDF component relationships

Many ID systems are designed only as alarms. However, most commercially available ID systems are equipped with some form of countermeasure (C-box) capability, ranging from shutting down TCP connections to modifying router filter lists. This allows an IDS to try to prevent further attacks from occurring after initial attacks are detected. Even systems that don't provide C-box capabilities can be hooked into home-brewed response programs to achieve a similar effect.

1.2 Network Intrusion Detection and Passive Analysis

Many ID systems are driven off of audit logs provided by the operating system, detecting attacks by watching for suspicious patterns of activity on a single computer system. This type of IDS is good at discerning attacks that are initiated by local users, and which involve misuse of the capabilities of one system. However, these "host based" (and multi-host) intrusion detection systems have a major shortcoming: they are insulated from network events that occur on a low level (because they only interpret high-level logging information).

Network intrusion detection systems are driven off of interpretation of raw network traffic. They attempt to detect attacks by watching for patterns of suspicious activity in this traffic. Network ID systems are good at discerning attacks that involve low-level manipulation of the network, and can easily correlate attacks against multiple machines on a network.

It's important to understand that while network ID has advantages over host-based ID, it also has some distinct disadvantages. Network ID systems are bad at determining exactly what's occurring on a computer system; host-based ID systems are kept informed by the operating system as to exactly what's happening. It is probably impossible to accurately reconstruct what is happening on a system by watching "shell", "login", and "telnet" sessions.

Network ID systems work by examining the contents of actual packets trans-

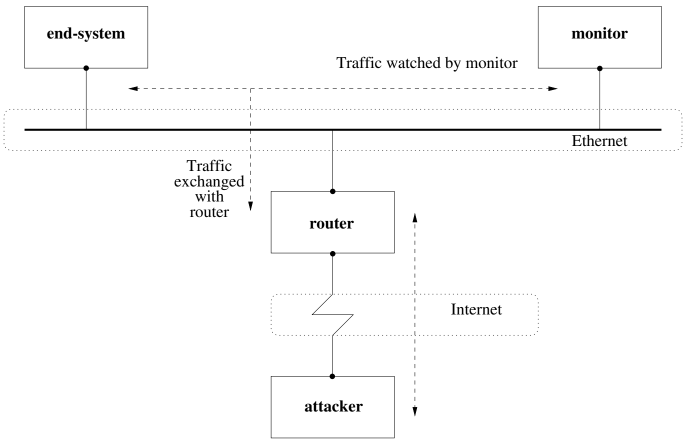

Figure 2: An example network topology using a passive monitor

mitted on the network. These systems parse packets, analyzing the protocols used on the network, and extract relevant information from them. This is typically accomplished by watching the network passively and capturing copies of packets that are transmitted by other machines.

Passive network monitors take advantage of “promiscuous mode” access. A promiscuous network device, or “sniffer”, obtains copies of packets directly from the network media, regardless of their destination (normal devices only read packets addressed to them). Figure 2 shows a simplified network topology in which a passive network monitor has been deployed.

Passive protocol analysis is useful because it is unobtrusive and, at the lowest levels of network operation, extremely difficult to evade. The installation of a sniffer does not cause any disruption to the network or degradation to network performance. Individual machines on the network can be (and usually are) ignorant to the presence of sniffer. Because the network media provides a reliable way for a sniffer to obtain copies of raw network traffic, there's no obvious way to transmit a packet on a monitored network without it being seen.

1.3 Signature Analysis

The question of what information is relevant to an IDS depends upon what it is trying to detect. For a system that is monitoring DNS traffic, the names of the hosts being queried for (and the responses to these queries) might be relevant. For a system attempting to detect attacks against FTP servers, the contents of all TCP connections to the FTP port would be interesting.

Some attacks can be discerned simply by parsing IP packets; an attempt to circumvent a packet filter using IP fragments is clearly observable simply by examining the fragment offset fields of individual IP fragments. Other attacks occur over multiple packets, or must be interpreted outside the context of the actual protocol (for instance, a DNS query might only be relevant if it involves

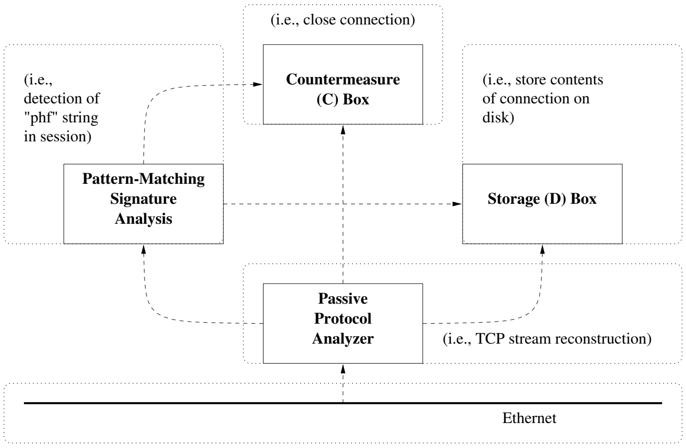

Figure 3: CIDF model of a network IDS

a certain host).

Most ID systems identify such attacks using a technique called "signature analysis" (also called "misuse detection"). Signature analysis simply refers to the fact that the ID system is programmed to interpret a certain series of packets, or a certain piece of data contained in those packets, as an attack. For example, an IDS that watches web servers might be programmed to look for the string "phf" as an indicator of a CGI program attack.

Most signature analysis systems are based off of simple pattern matching algorithms. In most cases, the IDS simply looks for a substring within a stream of data carried by network packets. When it finds this substring (for example, the "phf" in "`GET /cgi-bin/phf?`"), it identifies those network packets as vehicles of an attack.

Signature analysis and passive protocol analysis together define the event generation and analysis techniques used by the majority of commercially available ID systems. Figure 3 shows how these components fit into the CIDF model. For simplicity's sake, the remainder of this paper refers to systems that work like this as "network ID systems."

1.4 The Need for Reliable Intrusion Detection

Because of its importance within a security system, it is critical that intrusion detection systems function as expected by the organizations deploying them. In order to be useful, site administration needs to be able to rely on the information provided by the system; flawed systems not only provide less information, but also a dangerously false sense of security. Moreover, the forensic value of information from faulty systems is not only negated, but potentially misleading.

Given the implications of the failure of an ID component, it is reasonable to assume that ID systems are themselves logical targets for attack. A smart intruder who realizes that an IDS has been deployed on a network she is attacking will likely attack the IDS first, disabling it or forcing it to provide false information (distracting security personnel from the actual attack in progress, or framing someone else for the attack).

In order for a software component to resist attack, it must be designed and implemented with an understanding of the specific means by which it can be attacked. Unfortunately, very little information is publicly available to IDS designers to document the traps and pitfalls of implementing such a system. Furthermore, the majority of commercially available ID systems have proprietary, secret designs, and are not available with source code. This makes independent third-party analysis of such software for security problems difficult.

The most obvious aspect of an IDS to attack is its "accuracy". The "accuracy" of an IDS is compromised when something occurs that causes the system to incorrectly identify an intrusion when none has occurred (a "false positive" output), or when something occurs that causes the IDS to incorrectly fail to identify an intrusion when one has in fact occurred (a "false negative"). Some researchers[5] discuss IDS failures in terms of deficiencies in "accuracy" and "completeness", where "accuracy" reflects the number of false positives and "completeness" reflects the number of false negatives.

Other attacks might seek to disable the entire system, preventing it from functioning effectively at all. We say that these attacks attempt to compromise the "availability" of the system.

1.5 Points of Vulnerability in ID Systems

Each component identified by the CIDF model has unique security implications, and can be attacked for different reasons.

As the only inputs of raw data into the system, E-boxes act as the eyes and ears of an IDS. An attack against the event generation capabilities of an IDS blinds it to what's actually happening in the system it's monitoring. For example, an attack against the E-box of a network IDS could prevent it from obtaining packets off the network, or from appropriately decoding these packets.

Some intrusion detection systems rely on sophisticated analyses to provide security information. In such systems, the reliability of the A-box components used is important because an attacker that knows how to fool them can evade detection — and complicated analytical techniques may provide many avenues of attack. On the other hand, overly simplistic systems may fail to detect attackers that intentionally mask their attacks with complex, coordinated system interactions from multiple hosts[6].

The need for reliable data storage is obvious. An attacker that can subvert the D-box components of an IDS can prevent it from recording the details of her attack; poorly implemented data storage techniques can even allow sophisticated attackers to alter recorded information after an attack has been detected, eliminating its forensic value.

The C-box capability can also be attacked. If a network relies on these countermeasures for protection, an attacker who knows how to thwart the C-box can continue attacking the network, immune to the safety measures employed by the system. More importantly, countermeasure capabilities can be fooled into reacting against legitimate usage of the network — in this case, the IDS can actually be turned against the network using it (often un-detectably).

It is apparent that there are many different points at which an intrusion detection system can be attacked. A comprehensive treatment of all potential vulnerabilities is far outside the scope of this paper. Rather than attempting to document general problems common to all ID systems, we focus on a specific class of attacks against certain types of intrusion detection systems.

There exist several serious problems with the use of passive protocol analysis as an event-generation source for signature-analysis intrusion detection systems. This paper documents these problems, presents several attacks that exploit them to allow an attacker to evade detection by ID systems, and verifies their applicability to the most popular commercial ID systems on the market.

2 Problems with Network ID Systems

Our work defines two general problems with network intrusion detection: first, that there is insufficient information available in packets read off the wire to correctly reconstruct what is occurring inside complex protocol transactions, and next, that ID systems are inherently vulnerable to denial of service attacks. The first of these problems reduces the accuracy of the system, and the second jeopardizes its availability.

2.1 Insufficiency of Information on the Wire

A network IDS captures packets off the wire in order to determine what is happening on the machines it's watching. A packet, by itself, is not as significant to the system as the manner in which the machine receiving that packet behaves after processing it. Network ID systems work by predicting the behavior of networked machines based on the packets they exchange.

The problem with this technique is that a passive network monitor cannot accurately predict whether a given machine on the network is even going to see a packet, let alone process it in the expected manner. A number of issues exist which make the actual meaning of a packet captured by an IDS ambiguous.

A network IDS is typically on an entirely different machine from the systems it's watching. Often, the IDS is at a completely different point on the network. The basic problem facing a network IDS is that these differences cause inconsistencies between the ID system and the machines it watches. Some of these discrepancies are the results of basic physical differences, others stem from different network driver implementations.

For example, consider an IDS and an end-system located at different places on a network. The two systems will receive any given packet at different points in time. This difference in time is important; during the lag, something can happen on the end-system that might prevent it from accepting the packet. The IDS, however, has already processed the packet—thinking that it will be dealt with normally at the end-system.

Consider an IP packet with a bad UDP checksum. Most operating systems will not accept such a packet. Some older systems might. The IDS needs to know whether every system it watches will accept such a packet, or it can end up with an inaccurate reconstruction of what happened on those machines.

Some operating systems might accept a packet that is obviously bad. A poor implementation might, for example, allow an IP packet to have an incorrect checksum. If the IDS doesn't know this, it will discard packets that the end-system accepts, again reducing the accuracy of the system.

Even if the IDS knows what operating system every machine on the network runs, it still might not be able to tell just by looking at a packet whether a given machine will accept it. A machine that runs out of memory will discard incoming packets. The IDS has no easy way to determine whether this is the case on the end-system, and thus will assume that the end-system has accepted the packet. CPU exhaustion and network saturation at the end-system can cause the same problem.

Together, all these problems result in a situation where the IDS often simply can't determine the implications of a packet merely by examining it; it needs to know a great deal about the networking behavior of the end-systems that it's watching, as well as the traffic conditions of their network segments. Unfortunately, a network IDS doesn't have any simple way of informing itself about this; it obtains all its information from packet capture.

2.2 Vulnerability to Denial of Service

A "denial of service" (DOS) attack is one that is intended to compromise the availability of a computing resource. Common DOS attacks include ping floods and mail bombs — both intended to consume disproportionate amounts of resources, starving legitimate processes. Other attacks are targeted at bugs in software, and are intended to crash the system. The infamous "ping of death" and "teardrop" attacks are examples of these.

Denial of service attacks can be leveraged to subvert systems (thus compromising more than availability) as well as to disable them. When discussing the relevance of DOS attacks to a security system, the question of whether the system is "fail-open" arises. A "fail-open" system ceases to provide protection when it is disabled by a DOS attack. A "fail-closed" system, on the other hand, leaves the network protected when it is forcibly disabled.

The terms "fail-open" and "fail-closed" are most often heard within the context of firewalls, which are access-control devices for networks. A fail-open firewall stops controlling access to the network when it crashes, but leaves the network available. An attacker that can crash a fail-open firewall can bypass it entirely. Good firewalls are designed to "fail-closed", leaving the network completely inaccessible (and thus protected) if they crash.

Network ID systems are passive. They do not control the network or maintain its connectivity in any way. As such, a network IDS is inherently fail-open. If an attacker can crash the IDS or starve it of resources, she can attack the rest of the network as if the IDS wasn't even there. Because of the obvious susceptibility to DOS attacks that network ID systems have, it's important that they be fortified against them.

Unfortunately, denial of service attacks are extremely difficult to defend against. The resource starvation problem is not easily solvable, and there are many different points at which the resources of an IDS can be consumed. Attacks that crash the IDS itself are easily fixed, but finding all such vulnerabilities is not easily done.

3 Attacks

We discuss in this paper three different types of attacks against sniffer-based network ID systems. Two of them attempt to subtly thwart protocol analysis, preventing the signature-recognition system from obtaining adequate information from which to draw conclusions. The third leverages simple resource-starvation attacks to disrupt or disable the entire system.

All of our attacks involve an attacker that is specifically manipulating her network usage to create abnormal, or even pathological, streams of traffic. In most cases, they require low-level packet forgery. However, unlike normal "spoofing" attacks, these techniques are simplified by the fact that the attacker is manipulating her own sessions, not attempting to disrupt those of other users. Two of our attacks are new [1] , and specific to traffic analysis systems (though not necessarily to intrusion detection). Both are mechanisms by which an attacker can fool a protocol analyzer into thinking that something is (or is not) happening on the network. The first of these, which we call "insertion", involves an attacker stuffing the system with subtly invalid packets; the second, "evasion", involves exploiting inconsistencies between the analyzer and an end system in order to slip packets past the analyzer.

3.1 Insertion

An IDS can accept a packet that an end-system rejects. An IDS that does this makes the mistake of believing that the end-system has accepted and processed the packet when it actually hasn't. An attacker can exploit this condition by sending packets to an end-system that it will reject, but that the IDS will think are valid. In doing this, the attacker is "inserting" data into the IDS — no other system on the network cares about the bad packets.

We call this an "insertion" attack, and conditions that lend themselves to insertion attacks are the most prevalent vulnerabilities in the intrusion detection systems we tested. An attacker can use insertion attacks to defeat signature analysis, allowing her to slip attacks past an IDS.

To understand why insertion attacks foil signature analysis, it's important to understand how the technique is employed in real ID systems. For the most part, "signature analysis" uses pattern-matching algorithms to detect a certain string within a stream of data. For instance, an IDS that tries to detect a PHF attack will look for the string "phf" within an HTTP "GET" request, which is itself a longer string that might look something like "`GET /cgi-bin/phf?`".

The IDS can easily detect the string "phf" in that HTTP request using a simple substring search. However, the problem becomes much more difficult to solve when the attacker can send the same request to a webserver, but force the IDS to see a different string, such as "`GET /cgi-bin/pleasedontdetectthisforme?`". The attacker has used an insertion attack to add "leasedontdetectt", "is", and "orme" to the original stream. The IDS can no longer pick out the string "phf" from the stream of data it observes.

Figure 4 gives a simple example of the same attack. An attacker confronts the IDS with a stream of 1-character packets (the attacker-originated data stream),

[1] Vern Paxson of LBNL presented a paper describing several of the same attacks as we do at roughly the same time.[17]

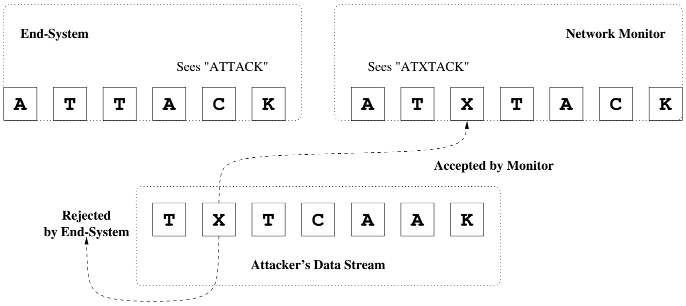

Figure 4: Insertion of the letter 'X'

in which one of the characters (the letter 'X') will be accepted only by the IDS. As a result, the IDS and the end system reconstruct two different strings.

In general, insertion attacks occur whenever an IDS is less strict in processing a packet than an end-system. An obvious reaction to this problem might be to make the IDS as strict as possible in processing packets read off the wire; this would minimize insertion attacks. However, another severe problem ("evasion" attacks) occurs when this design approach is taken.

3.2 Evasion

An end-system can accept a packet that an IDS rejects. An IDS that mistakenly rejects such a packet misses its contents entirely. This condition can also be exploited, this time by slipping crucial information past the IDS in packets that the IDS is too strict about processing. These packets are "evading" the scrutiny of the IDS.

We call these "evasion" attacks, and they are the easiest to exploit and most devastating to the accuracy of an IDS. Entire sessions can be carried forth in packets that evade an IDS, and blatantly obvious attacks couched in such sessions will happen right under the nose of even the most sophisticated analysis engine.

Evasion attacks foil pattern matching in a manner quite similar to insertion attacks. Again, the attacker causes the IDS to see a different stream of data than the end-system — this time, however, the end-system sees more than the IDS, and the information that the IDS misses is critical to the detection of an attack.

In the insertion attack we mentioned above, the attacker sends an HTTP request, but muddies its contents on the IDS with additional data that make the request seem innocuous. In an evasion attack, the attacker sends portions of the same request in packets that the IDS mistakenly rejects, allowing her to

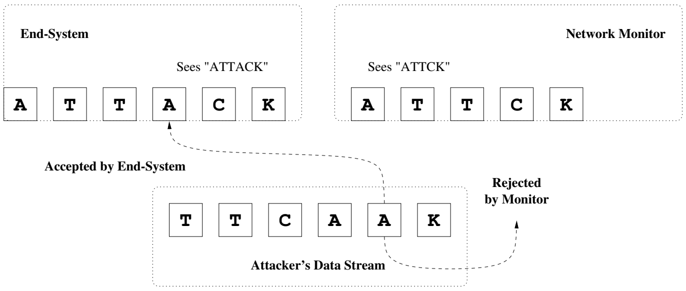

Figure 5: Evasion of the letter 'A'

remove parts of the stream from the ID system's view. For example, the original request could become "`GET /gin/f`", which would have no meaning to most ID systems. Figure 5 shows the same type of attack.

3.3 Real World Insertion and Evasion

In reality, insertion and evasion attacks are not this easy to exploit. An attacker usually does not have the luxury of injecting arbitrary characters into a stream. However, these attacks can come into play well before pattern matching becomes a consideration. One example of a place in which insertion attacks can be leveraged at a very low level is stream reassembly. To understand how insertion and evasion play into reassembly, we'll first explain what we mean by the term.

Many network protocols are simple and easy to analyze. They involve one system sending a single request to another, and waiting for that system to respond. For example, a network monitor can easily determine the purpose of a single UDP DNS query by looking at one packet.

Other protocols are more complex, and require consideration of many individual packets before a determination can be made about the actual transaction they represent. In order for a network monitor to analyze them, it must statefully monitor an entire stream of packets, tracking information inside each of them. For example, in order to discover what is happening inside of a TCP connection, the monitor must attempt to reconstruct the streams of data being exchanged over the connection.

Protocols like TCP allow any amount of data (within the limits of the IP protocol's maximum packet size) to be contained in each discrete packet. A collection of data can be transmitted in one packet, or in a group of them. Because they can arrive at their destination out of order, even when transmitted in order, each packet is given a number that indicates its place within the intended order of the stream. This is commonly referred to as a "sequence

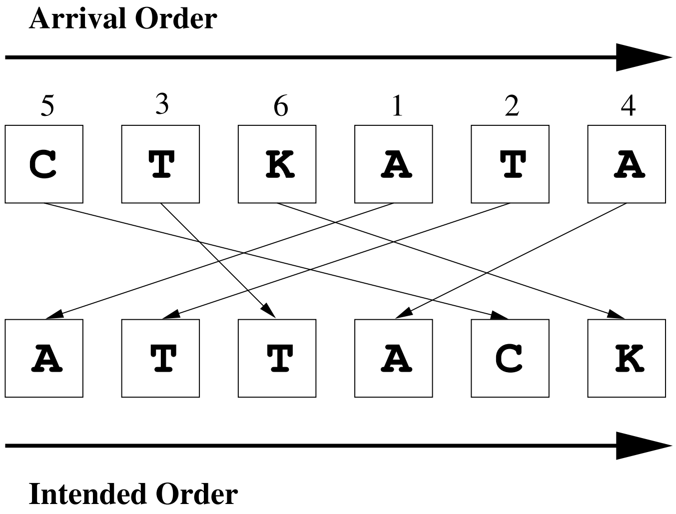

Figure 6: Sequenced reassembly

number", and we call collections of packets marked with sequence numbers "sequenced".

The recipient of a stream of TCP packets has the responsibility of re-ordering and extracting the information contained in each of them, reconstructing the original collection of data that the sender transmitted. The process of taking a collection of unordered, sequenced packets and reconstructing the stream of data they contain is termed "reassembly". Figure 6 shows an example of how a stream of data tagged with sequence numbers might be reassembled.

Reassembly issues manifest themselves at the IP layer, as well; IP defines a mechanism, called "fragmentation", that allows machines to break individual packets into smaller ones. Each individual fragment bears a marker that denotes where it belongs in the context of the original packet; this field is called the "offset". IP implementations must be able to accept a stream of packet fragments and, using their offsets, reassemble them into the original packet.

Insertion attacks disrupt stream reassembly by adding packets to the stream that would cause it to be reassembled differently on the end-system—if the end system accepted the disruptive packets. The inserted packets could change the sequencing of the stream (consuming hundreds of sequence numbers), preventing the IDS from dealing properly with the valid packets that follow it. Packets can be inserted that overlap old data, rewriting the stream on the IDS. And, in some situations, packets can be inserted that simply add content to the stream which changes its meaning.

Evasion attacks disrupt stream reassembly by causing the IDS to miss parts of it. The packets lost by the IDS might be vital for the sequencing of the stream; the IDS might not know what to do with the packets it sees after the evasion attacks. In many situations, it's fairly simple for the attacker to create an entire stream that eludes the IDS.

3.4 Ambiguities

In many cases, defending against insertion and evasion attacks is easy. The behavior that an attacker is exploiting to insert packets into the IDS is, in these cases, simply wrong. The IDS might not be verifying a checksum or examining a header field correctly; fixing the problem merely involves modifying the IDS

Section	Info Needed	Ambiguity
Section 4.1.1	Network Topology	IP TTL field may not be large enough for the number of hops to the destination
Section 4.1.1	Network Topology	Packet may be too large for a downstream link to handle without fragmentation
Section 4.1.2	Destination Configuration	Destination may be configured to drop source-routed packets
Section 4.3.1	Destionation OS	Destination may time partially received fragments out differently depending on its OS
Section 4.3.3	Destination OS	Destination may reassemble overlapping fragments differently depending on its OS
Section 5.2.2	Destination OS	Destination host may not accept TCP packets bearing certain options
Section 5.2.2	Destination OS	Destination may implement PAWS and silently drop packets with old timestamps
Section 5.4.3	Destination OS	Destination may resolve conflicting TCP segments differently depending on its OS
Section 5.5.1	Destination OS	Destination may not check sequence numbers on RST messages

Figure 7: Ambiguities identified in this paper

to check these things.

In some cases, however, fixing the problem is not easy. There are situations in which a network monitor cannot determine by looking at a packet whether it will be accepted. This can be due to varying end-system behavior (one operating system might process a packet differently from another). Basic network ambiguities can also cause problems. In some cases, unless the IDS knows exactly what path the packet is going to take to get to its destination, it won't know whether it will actually arrive there.

Attacks that exploit these kinds of problems cannot easily be defended against unless the IDS has a source of information that resolves the ambiguity. If the IDS knows what operating system is running on the destination system, it may be able to discern whether a packet is acceptable to that system. If the IDS can reliably track the topology of the network, it may be be able to determine whether or not a packet will ever be received by an end-system. In general, we say a traffic analysis problem is "ambiguous" if an important conclusion about a packet cannot be made without a secondary source of information.

Figure 7 shows the ambiguities this paper identifies. Each ambiguity can potentially be resolved if the IDS has certain information (either a reliable view of the topology of the network, the configuration of the end-systems it's watching, or the OS and version of those systems). This is, of course, not an exhaustive list.

The next two sections of this paper provide examples of how insertion and evasion attacks affect protocol analysis at the network (IP) and transport (TCP) layers. These sections provide real-world examples of attacks on IP network ID systems in great detail, working from the basic attacks we've defined here.

Line	Description
229	No IP addresses set yet
232	Received packet is too short to be an IP datagram.
240	Received packet is too short to be an IP datagram.
247	IP version isn't '4'
253	IP "header length" field too small
257	IP "header length" is set larger than the entire packet
269	Bad header checksum
278	IP "total length" field is shorter than "header length"
348	Packet has IP options and ip_dooptions() returns an error
437	Not addressed to this host
450	Too small to be a fragment

Figure 8: FreeBSD 2.2 ip_input() packet discard points (*netinet/ip_input.c*)

4 Network-Layer Problems

We begin our discussion of specific, observable problems in network intrusion detection systems at the IP layer. An insertion or evasion problem occurring within the IP processing of an IDS affects all higher levels of processing as well; a problem that allows an attacker to insert an arbitrary IP packet allows that attacker, by extension, to insert an arbitrary (well-formed) UDP or ICMP packet. It is thus extremely important that an ID system be immune to insertion or evasion attacks on this level.

4.1 Simple Insertion Attacks

There are many ways that an attacker can send an IP packet that only an IDS will accept. We collected candidate methods by examining the IP driver source code of the 4.4BSD operating system. Any condition that causes 4.4BSD to drop a received packet must be accounted for in an intrusion detection system. An inconsistency between 4.4BSD and an IDS represents a potential insertion or evasion attack against that IDS. Figure 8 lists all the points in FreeBSD 2.2's "ip_input" routine that discard received datagrams.

4.1.1 Bad Header Fields

The easiest way for an IP datagram to be discarded by an endpoint is for it to have an invalid header field. The header fields of an IP packet are described in RFC731[7].

One problem with attempting to use packets with bad header fields for insertion attacks is that doing so often prevents the packet from being forwarded by Internet routers. This makes it difficult to use such packets for an attack, unless the IDS is situated on the same LAN as the attacker (in which case the attacker can already manipulate the IDS via packet forgery). A good example is the "version" field; assigning a value other than 4 to this field will prevent the packet from being routed.

Another problem with using bad header fields is the fact that some of them need to be correct for the packet to be parsed correctly ("correctly" here meaning "in the manner intended by the attacker"). For instance, incorrectly specifying the size of the IP packet itself, or the size of its header, may prevent the IDS from locating the transport layer of the packet.

One IP header field that is easy to neglect is the checksum. It may seem unnecessary for an IDS to verify the accuracy of the checksum on each captured IP packet; however, a datagram with a bad checksum will not be processed by most IP implementations. An IDS that does not reject packets with bad checksums is thus vulnerable to a very simple insertion attack.

A harder problem to solve is the TTL field. The TTL (time to live) field of an IP packet dictates how many "hops" a packet can traverse on its way to its destination. Every time a router forwards a packet, it decrements the TTL. When the TTL runs out, the packet is dropped. If the IDS is not on the same network segment as the systems it watches, it is possible to send packets that only the IDS will see by setting the TTL just long enough for the packet to reach the IDS, but too short for the packet to actually arrive at its destination.[17]

A similar problem occurs in relation to the "Don't Fragment" (DF) flag in the IP header. The DF flag tells forwarding devices not to split a packet up into fragments when the packet is too large to be forwarded, but instead to simply drop the packet. If the maximum packet size of the network the IDS is on is larger than that of the systems it watches, an attacker can insert packets by making them too large for the destination network and setting the DF bit.[17]

Both of these problems can lead to ambiguities that are only solveable if the IDS has an intimate knowledge of the topology of the network it is monitoring.

4.1.2 IP Options

The IP checksum problem is fairly simple to solve; an IDS can reasonably assume that if the checksum is wrong, the datagram will not be accepted by the end-system it's addressed to. A trickier problem is that of parsing IP options. This is more likely to vary between hosts, and the interpretation of options requires specialized processing.

For example, most end-systems will drop a packet that is "strict source routed"[9] when the host's own address is not in the specified source route. It is reasonable for an IDS to drop such packets, avoiding an insertion attack. However, many operating systems can be configured to automatically reject source routed packets. Unless the IDS knows whether a source-routed packet's destination rejects such packets, the correct action to take is ambiguous.

Examination of source route options on IP packets may seem like an obvious requirement for a security program. However, there are other options that must be accounted for that are less obviously relevant. For instance, the "timestamp" option requests that certain recipients of the datagram place a timestamp within

Line	**Option**	**Description**
837	Any	Bad option length
858	Source Route	Option offset is less than '4'
866	Strict Source Route	This host is not one of the listed hops
886	Source Route	This host is configured to drop source routed packets
911	Source Route	No route to next hop in route
927	Record Route	Option offset is less than '4'
943	Record Route	No route to next hop
957	Timestamp	Option length is too short
960	Timestamp	Timestamp recording space is full and the overflow counter has wrapped back to zero
971	Timestamp	Not enough record space to hold timestamp and IP address
985	Timestamp	Not enough record space to hold timestamp and IP address
995	Timestamp	Bad timestamp type given

Figure 9: FreeBSD 2.2 ip_dooptions() packet discard points

the packet. The code that processes the timestamp option can be forced to discard the packet (if the option is malformed). If the sniffer does not validate the timestamp option in the same manner as the end systems it watches, the inconsistency can be exploited. Figure 9 lists all the places in which FreeBSD 2.2's option processing code discards incoming datagrams.

Most IP option processing problems in the 4.4BSD option processing code results in the transmission of an ICMP error message, notifying the sender of the errant datagram of the problem. An IDS could potentially listen for such messages to determine whether an oddly-specified option is correct. This is not always reliable; some operating systems (Sun Solaris, for instance) will rate-limit ICMP, suppressing the error messages. Furthermore, tracking ICMP responses to datagrams bearing options requires the IDS to keep state for each IP packet; this will consume resources and potentially allow an attacker an avenue for a denial of service attack.

4.2 MAC Addresses

Although obviously not an IP problem per se, the same implications for insertion attacks exist due to link-layer addressing. An attacker on the same LAN as a network monitor can direct link-layer frames to the IDS, without ever allowing the host specified as the IP destination to see the packet.

If the attacker knows the link-layer address of the IDS, she can simply address her fake packet to the IDS. No other system on the LAN will process the packet,

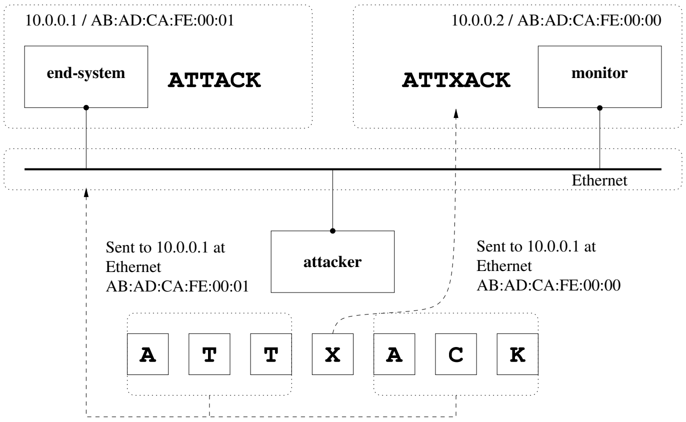

Figure 10: Insertion Attacks at the Link Layer

but, if the IDS doesn't check the MAC address on the received packet, it won't know this. Figure 10 shows an example of an attacker that inserts a character in the IDS by directing a packet to the IDS via the Ethernet link-layer.

Even if the attacker doesn't know the link-layer address of the network monitor, she can exploit the fact that the network monitor is operating in promiscuous mode by addressing the frame to a fake address. Again, unless the IDS verifies the destination address in the IP header against the correct link-layer address (and can do so reliably), it can be fooled by falsely-addressed link-layer frames.

4.3 IP Fragmentation

IP packets can be broken into smaller packets, and reassembled at the destination. This is termed "fragmentation", and is an integral part of the IP protocol. IP fragmentation allows the same information to travel over different types of network media (which may have different packet size limits) without limiting the entire protocol to an arbitrary small maximum packet size. A detailed explanation of IP fragmentation can be found in Stevens[8], or in RFC791[9].

Because end-systems will reassemble a stream of IP fragments, it is important that a network monitor correctly reassemble fragments as well. An IDS that does not correctly reassemble fragments can be attacked simply by ensuring that all data is exchanged between machines using artificially fragmented packets.

4.3.1 Basic Reassembly Problems

Streams of IP fragments usually arrive in order. The last fragment in a stream is clearly marked (the IP header contains a flag that specifies whether more fragments follow a given packet). However, even though it rarely happens, the protocol allows fragments to arrive in any arbitrary order. An end system must be able to reassemble a datagram from fragments that arrive out of order.

Because fragments usually arrive in order, it's easy to make the mistake of assuming that they always will. An IDS that does not properly handle out-of-order fragments is vulnerable; an attacker can intentionally scramble her fragment streams to elude the IDS. It's also important that the IDS not attempt to reconstruct packets until all fragments have been seen. Another easily made mistake is to attempt to reassemble as soon as the marked final fragment arrives.

Another significant problem is the fact that received fragments must be stored until the stream of fragments can be reassembled into an entire IP datagram. An IDS can be attacked by flooding the network with partial, fragmented datagrams, which will never be completed. A naive IDS will run out of memory as it attempts to cache each fragment, since the fragmented packets are never completed.

End-systems must deal with this problem as well. Many systems drop fragments based on their TTL, to avoid running out of memory due to over-full fragment queues. An IDS that eventually drops old, incomplete fragment streams must do so in a manner consistent with the machines it's watching, or it will be vulnerable to insertion (due to accepting fragment streams that end-systems have dropped already) or evasion (due to dropping fragments that end-systems have not yet discarded) attacks.

4.3.2 Overlapping Fragments

It has long been known that there are serious security implications arising from interactions between fragmentation and network access control devices (like packet filters). Two well-known attacks involving fragmentation allow attackers to potentially evade packet filters by employing pathological fragment streams. The first of these attacks involves simply sending data using the smallest fragments possible; the individual fragments will not contain enough data to filter on.

The second problem is far more relevant to ID systems. It involves fragmentation overlap, which occurs when fragments of differing sizes arrive out of order and in overlapping positions. If a fragment arriving at an end-station contains data that has already arrived in a different fragment, it is possible that the newly arrived data may overwrite some of the old data.

This presents problems for an IDS. If the IDS does not handle overlapping fragments in a manner consistant with the systems it watches, it may, given a stream of fragments, reassemble a completely different packet than an end-system in receipt of the same fragments. An attacker that understands the specific inconsistency between an end-system and an IDS can obscure her attack by couching data inside of overlapping fragment streams that will be reassembled differently on the two systems.

Overlap resolution is further complicated by the fact that data from conflicting fragments is used differently depending on their positions. In some situations, conflicts are resolved in favor of the new data. In others, the old

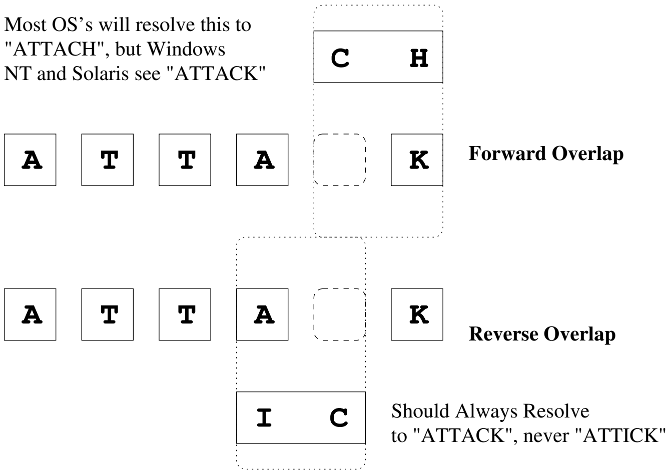

Figure 11: Forward and Reverse Overlap

Operating System	Overlap Behavior
Windows NT 4.0	Always Favors Old Data
4.4BSD	Favors New Data for Forward Overlap
Linux	Favors New Data for Forward Overlap
Solaris 2.6	Always Favors Old Data
HP-UX 9.01	Favors New Data for Forward Overlap
Irix 5.3	Favors New Data for Forward Overlap

Figure 12: IP fragment overlap behavior for various OS's

data is preferred and the new data is discarded. An IDS that does this incorrectly is vulnerable to evasion attacks. Figure 11 shows the different scenarios involved in fragmentation overlap.

4.3.3 Effects of End-System Fragmentation Bugs

ID systems aren't the only IP implementations that can incorrectly handle overlapping fragments. The IP drivers in end-systems can have bugs as well. The complexity of IP fragment reassembly makes the existence of incorrect implementations quite likely. Unless the IDS knows exactly which systems have non-standard drivers, it is incapable of accurately reconstructing what's happening on them.

For example, Windows NT resolves overlapping fragments consistently in favor of the old data (we were unable to create a fragment stream that forced Window NT to rewrite a previously received fragment). This differs from 4.4BSD, which resolves conflicts as suggested by the standard (in favor of the new data in cases of forward overlap)[10]. Figure 12 gives examples of how several popular operating systems resolve overlap.

The end result is that fragmentation reassembly is different on the end-system depending on the operating system. Unless the IDS knows which OS the system is running, it will have absolutely no way of knowing what form of conflict resolution was performed, and thus no conclusive evidence of what was actually reassembled.

4.3.4 IP Options in Fragment Streams

IP packets can bear options. When an IP packet is fragmented, the question arises as to whether the options from the original packet should be carried on all the fragments. RFC791[9] dictates that certain IP options are to be present in every fragment of a datagram (for example, the "security" option), and others must appear only in the first fragment.

A strict implementation of IP could discard fragments that incorrectly present options. Many implementations do not. If the IDS doesn't behave exactly like the machines it's watching in this respect, it will be vulnerable to insertion and evasion attacks.

4.4 Forensic Information from IP Packets

It is an unfortunate fact that the IP version 4 protocol is in no way authenticated. This poses some problems to ID systems attempting to collect evidence based on information seen in IP headers; anyone can forge an IP packet appearing to come from some arbitrary host.

This problem is particularly severe with connectionless protocols. In connection-oriented protocols, a weak conclusion can be drawn as to the origin of a session based on whether a valid connection is created; the sequence numbers employed by protocols like TCP provide at least cursory assurance that the attack is originating at the address it appears to come from. An IDS can observe that a connection uses consistantly correct sequence numbers and have a reasonable assurance that it's not being blindly spoofed.

Unfortunately, no such assurance exists with connectionless protocols; an attack against the DNS, for instance, could be sourced from any address on the net. It is important that operators of ID systems be aware of the questionable validity of the addressing information they're given by their system.

5 TCP Transport-Layer Problems

A large portion of the attacks detected by ID systems occur over TCP connections. This imposes the requirement that an IDS be able to reconstruct the flow of data passing through a stream of TCP packets. If the IDS can't do this in a manner consistent with end systems it's watching, it is vulnerable to attack.

For normal TCP connections, initiated by innocuous network applications like "telnet", this is not difficult. Against an attacker, who is stretching the TCP protocol to its limits (and, in exploiting OS bugs, beyond those limits) to avoid detection, the problem is far more difficult.

There are many different ways to implement a TCP connection monitor. Each has its advantages, and each has serious flaws. The lack of a canonical "Right Way" to process a captured stream of TCP packets is a major problem with network ID systems.

5.1 Definition of Terms

TCP connection monitoring is a complicated subject. In order to simplify our discussion, we define several terms describing information used by the monitor to track and record information flowing through a TCP session. For the most part, these terms are synonymous with those used by the BSD TCP implementation.

Every TCP connection has four identifiers (two for the client, two for the server) which distinguish it from any other connection on the network. These are the client (or source) and server (or destination) IP addresses, and the client and server TCP port numbers. Two connections cannot exist on the network that share these identifiers. We'll refer to this information as the "connection parameters".

The TCP protocol specification (RFC793[12]) defines several "states" that any given connection can be in. In this paper, we refer only to states observable by an IDS (those involving the actual exchange of data between two hosts). The vast majority of all possible connections exist in the "CLOSED" state, meaning that no connection currently exists using those parameters. An active, established connection is said to be in "ESTABLISHED" state. We'll introduce other states when they become relevant to our discussion.

TCP implements a reliable, sequenced stream protocol. By "reliable", we mean that each end of a connection can determine whether data it has sent was successfully received, and can do something to remedy the situation when it isn't. TCP is "sequenced" because it employs "sequence numbers" to determine where any piece of data represented in a packet belongs within a stream.

In order for an IDS to reconstruct the information flowing through a TCP connection, it must figure out what sequence numbers are being used. We call the process that an IDS goes through to determine the current valid sequence numbers for a connection "synchronization". A scenario in which the IDS becomes confused about the current sequence numbers is termed "desynchronization".

When an IDS is desynchronized from a connection, it cannot accurately reconstruct the data being passed through the connection. In many cases, ID systems become completely blinded (not reconstructing *any* data from the connection) when this occurs. Thus, a major goal of an attacker is to desynchronize the IDS from her connections.

Along with sequence numbers, TCP tracks several other pieces of information about a connection. TCP defines a flow-control mechanism that prevents one side of a connection from sending too much data for the other side to process; this is tracked through each side's "window". TCP also allows for out-of-band data to be sent in a stream, using the "urgent pointer".

This collection of state information can be represented internally on an end-system in any manner. We refer to the abstract concept of the block of information that an implementation must manage to follow a single connection as a "TCP control block", or "TCB". A network IDS must maintain a TCB for every connection that it watches.

5.1.1 IDS State Transition

TCBs are only useful for connections that are not (in fact) in CLOSED state. Because it would be infeasible for an IDS to maintain a TCB for every possible connection, any network IDS defines a mechanism by which TCBs can be created for newly detected connections, and destroyed for connections that are no longer relevant.

In our discussion of IDS TCP problems, we isolate three different points at which the processing of a connection by an IDS can be subverted. These are TCB creation (the point at which an IDS decides to instantiate a new TCB for a detected connection), stream reassembly (the process an IDS uses to reconstruct a stream associated with an open TCB), and TCB teardown (the point at which the IDS decides to retire a TCB).

Contributing to attacks against each of these three points are data insertion attacks, which can allow an attacker to confuse the IDS as to what data is actually arriving at the end-system. In some cases, such as within the context of stream reassembly, data insertion attacks make the reliable monitoring of a TCP session practically impossible; it is thus important the the IDS not be vulnerable to insertion attacks. This is not an easy goal to achieve.

5.2 Simple Insertion Attacks

As with the IP protocol, there are several different ways in which a single packet can be inserted into an IDS. TCP input processing is complex, and there are many different cases that can cause a received packet to be dropped. As always, if an IDS doesn't process TCP packets in the same manner as the end-systems it's monitoring, it is potentially vulnerable to insertion attacks.

As with our analysis of IP monitoring, we used the source code to the 4.4BSD kernel to obtain candidate cases for potential insertion attacks. Again, any point in 4.4BSD's tcp_input() function that causes a received packet to be dropped without complete processing was identified as a possible problem. Figure 13 lists points in FreeBSD 2.2's tcp_input() code where incoming segments are dropped.

A TCP segment is acknowledged if the receiving system generates a message in response to the segment; when this occurs, we indicate whether this is via an RST or ACK message. The transmission of a message in response to a bad segment is significant because an IDS could potentially detect invalid segments by examining the manner in which they are acknowledged, though this is complicated both by resource and efficiency issues, as well as the potential for inconsistant behavior across different operating systems.

5.2.1 Malformed Header Fields

Data from a TCP packet can be extracted and used in reassembly without looking at many of the header fields. This makes it dangerously easy to design a TCP session monitor that is vulnerable to packet insertion; it is important to validate the header fields of a TCP packet before considering its data.

One very easily overlooked field is the "CODE", which determines the type of message being sent in a given TCP segment. The TCP code is specified as a series of binary flags. Certain combinations of these flags are invalid, and should result in a discarded packet. Additionally, many TCP implementations will not accept data in a packet that does not have the "acknowledge" ("ACK") flag set.

According to the TCP specification, TCP implementations are required to accept data contained in a SYN packet. Because this is a subtle and obscure point, some implementations may not handle this correctly. If an IDS doesn't consider data in a SYN packet, it is vulnerable to a trivial evasion attack; if it does, it may be vulnerable to insertion attacks involving incorrect end-system implementations.

Another often overlooked TCP input processing issue is checksum computation. All TCP implementations are required to validate incoming packets with the Internet checksum. Many ID systems fail to perform this check; packets can be inserted into these systems simply by sending TCP segments with intentionally corrupt checksums.

5.2.2 TCP Options

As in IP, it is important that the IDS process TCP options correctly. Unfortunately, processing of TCP options is significantly trickier than processing IP options. One reason for this is the fact that several TCP options have only recently been created (timestamp and window scale, for instance). Another is the fact that TCP specifies rules for when a TCP option can appear within the context of a connection. Certain options can be invalid in certain connection states.

RFC1323[13] introduces two new TCP options designed to increase the reliability and performance of TCP in high-speed environments. With these new

Line	**Acknowledged?**	**Condition**
295	No	Actual received packet too short
312	No	Bad checksum
323	No	Offset too Short (into TCP header) or too long
331	No	Actual received packet too short
369	RST	No listening process
382	RST	No listening process
384	No	Connection is in CLOSED state
404	No	Packet other than SYN received in LISTEN state
409	RST	ACK packet received in LISTEN state
423	No	Can't track new connections
628	No	Received RST packet in LISTEN state
630	RST	ACK packet received in LISTEN state
632	No	Any packet without SYN received in LISTEN state
639	No	Broadcast or Multicast SYN received
643	No	Out of resources
655	No	in_pcbconnect() failure
662	No	Out of resources
773	No	ACK packet, bad sequence numbers
789	No	In SYN_SENT state, received packet other than SYN
796	No	In SYN_SENT state, received packet has bad CC.ECHO
936	No	In TIME_WAIT state, packet has bad CC option
945	No	Any other packet received in TIME_WAIT state
979	ACK	Bad timestamp (too old)
993	No	In T/TCP, no CC or bad CC on non-RST packet
1048	RST	Listening user process has terminated
1087	ACK	Packet is out of receive window
1156	No	ACK bit not set on non-SYN data packet
1175	RST	ACK packet, bad sequence numbers
1234	No	Duplicate ACK
1300	ACK	ACK packet sent out of window
1443	ACK	In TIME_WAIT state, received ACK

Figure 13: FreeBSD 2.2 tcp_input() packet drop points (*netinet/tcp_input.c*)

options came the possibility that TCP options could appear on packets that were not SYN segments, a departure from the previous convention. RFC1323 dictates that options can only appear in non-SYN segments if the option has been specified and accepted previously in that connection.

Because certain TCP implementations may reject non-SYN segments containing options not previously seen, it's important that the IDS not blindly accept such a packet. On the other hand, some end-systems may simply ignore the bad options, but continue to process the packet; if the IDS doesn't correctly determine what the end-system has done, it will either be vulnerable to an insertion attack or another trivial packet evasion attack.

Another concept defined by RFC1323 is PAWS, or "protection against wrapped sequence numbers". Systems implementing PAWS track timestamps on segments; if a segment is received that contains a timestamp echo that is older than some threshold time, it is dropped. An attacker can trivially create a TCP segment with an artificially low timestamp, which will cause PAWS-compliant TCP stacks to drop the packet without further processing.

Not only does the IDS need to know whether the end-system supports PAWS, but it also needs to know what the end-system's threshold value for timestamps is. Without this information, an IDS may erroneously process invalid TCP segments, or, even worse, make an incorrect guess as to the validity of a segment and enable evasion attacks.

5.3 TCB Creation

The first point at which TCP session monitoring can be subverted is in TCB creation. The TCB creation policies of an IDS determine the point at which it begins recording data for a given connection, as well as the initial state (sequence numbers, etc) used to synchronize the monitoring with the actual session.

TCB creation is a troublesome issue. There are many different methods that can be employed to determine when to open a TCB, and none of the straightforward methods is without problems. Some techniques are obviously inferior to others, however, and it's important to indicate which these are. TCB creation establishes the initial state of a connection, including its sequence numbers; the ability to forge fake TCBs on the IDS can allow an attacker to desynchronize future connections that use the same parameters as the forged connection.

TCB creation as a concept revolves around the TCP three-way handshake (or "3WH"), which is an exchange of TCP packets between a client (the "active opener" of a connection) and server (the "passive opener"). The 3WH establishes the initial sequence numbers used for that connection, along with any other parameters (the use of running timestamps, for instance) that may be important.

There are very few options available to an end-system in implementing TCB creation; a TCB cannot be completely opened until a three-way handshake is completed successfully. Without the 3WH, the two ends of a connection have no agreed-upon sequence numbers to use, and will be unable to exchange data.

An IDS, on the other hand, has many options. ID systems can attempt to determine the sequence numbers being used simply by looking at the sequence numbers appearing in TCP data packets (we refer to this as "synching on data"), or it can rely entirely on the 3WH. Compromises can be made to either approach; information from a 3WH can be used, but not relied upon, by the IDS, and the IDS does not necessarily need to wait for an entire 3WH before opening a TCB.

We attempt to outline all the straightforward mechanisms for establishing TCBs on an IDS here. This is by no means a complete list of all the ways this task can be accomplished, but these are the techniques that we expect to see utilized in typical ID systems.

5.3.1 Requiring Three-Way Handshake

The first decision for IDS designers to make is whether or not to rely completely on the three-way handshake for TCB initiation. An IDS that relies on the 3WH will not record data in a connection for which it did not observe a handshake.

This has a few distinct disadvantages. The first and most obvious is the fact that the IDS will miss entirely any TCP connection for which it does not see the 3WH. This obviously presents problems at program initialization time (the IDS will only be able to see connections that start after it does), but also presents a serious opportunity for connection evasion by an attacker who can prevent the IDS from seeing the 3WH.

Another problem occurs in combination with TCP reassembly. If an IDS uses the 3WH to determine the initial sequence numbers of a connection, and then validates data against those sequence numbers, it can potentially be tricked into desynchronization by an attacker who forges a realistic-looking (but fake) handshake. If the IDS records the sequence numbers from the handshake, a real connection, using different sequence numbers but the same parameters, will be undetectable as long as the attacker-created TCB is open.

TCP options compound this problem. In order to correctly deal with TCP extensions such as PAWS, the IDS *must* see the three-way handshake (the handshake determines whether the use of certain options is legitimate with the connection). If the IDS fails to detect this, it will be vulnerable to insertion attacks against some operating systems (notably 4.4BSD).

The Effects of Filtering on Handshake Detection Many security-conscious networks have network filtering in place that makes it difficult for a remote attacker to send packets to the network that have source addresses of machines behind the filter. This technique, which is referred to as "inside-outside" filtering or "spoof-protection", makes some attacks against TCB creation harder; the attacker, trying to trick the IDS into opening or desynchronizing a TCB, cannot easily forge server response packets.

An IDS can take advantage of this by trusting packets that appear to originate from machines behind such filters (the IDS assumes that the presence of these filters makes forging such packets impossible). Trusted packets can be used as a reliable indicator of connection state.

It's important to base the decision on whether to "trust" a packet off the source address on the packet, and not on the type of TCP message it contains. An IDS that "trusts" SYN+ACK packets, assuming that they are server response messages and thus protected by packet filters, cannot accurately detect attacks against network clients (in which the filtered addresses are the clients, not the servers).

Of course, the IDS must be configured to know which addresses are trustworthy and which aren't. An IDS which blindly relies on the fact that addresses on its own LAN are spoof-protected will be completely vulnerable if no actual spoof protection exists. The configuration of the IDS must be consistent with that of the actual packet filters.

Requiring Full Handshake An IDS that requires a full 3WH will not record data for a connection until it sees and accepts all 3 packets in the three-way handshake. Two of these packets are sent by the client (and thus, for server attacks, can be considered under the complete control of an attacker), and 1 of them is sent by the server. In TCP terminology, this means that the IDS doesn't start recording until the connection enters ESTABLISHED state.

As mentioned previously, requiring a complete handshake makes it dangerously easy to miss connections (due to packet evasion techniques, simple performance problems on the TCP monitor that cause it to miss packets, or even attacker-induced performance problems).

Allowing Partial Handshake An IDS that requires at least a partial 3WH will not record data for a connection until it sees some portion of the handshake occur. Evidence of a three-way handshake validates TCB initiation (we'll see that there are problems with blindly creating TCBs to synch up to data streams), and potentially reduces the ability of an attacker to trick the system into creating false TCBs. Requiring only partial handshakes also decreases the probability that a connection will be missed due to packet drops under load.

The question that then arises is "what portion of the three-way handshake needs to be seen by the IDS before a TCB is created?". An IDS can create a TCB when it sees the initial connection solicitation (the client SYN), or when it sees the server return a positive response (the server SYN+ACK). In the presence of inside-outside filtering, it can be difficult for an attacker to spoof the server response; server SYN+ACK responses are thus a more reliable indication that a connection is occurring. If an attacker cannot spoof the server response, the SYN+ACK also contains the valid sequence numbers for the connection, allowing the IDS to more accurately initialize the TCB.

In either case, it's important to note that until the handshake is completed, a connection doesn't actually exist. The only indication an IDS has that a connection isn't being spoofed is when then the client responds to the server SYN+ACK with an ACK confirming the server's initial sequence number. If an IDS uses partial handshakes to open TCBs, it can be tricked into opening TCBs for nonexistent connections.

5.3.2 Data Synchronization

The alternative to requiring a three-way handshake to open a TCB is to deduce the initial state of a connection by looking at data packets, presumably after a connection has been opened. Since the IDS is not an active participant in the connection, it doesn't necessarily even have to consider 3WH packets; it is entirely feasible to track normal connections simply by looking at ACK packets (packets containing data).

The primary advantage of this technique, which we refer to as "synching on data", is that the sniffer picks up more data than systems that require handshakes. The system can recover from the loss of an important 3WH packet, and can detect connection that began before the program was started. Unfortunately, synching on data creates the possibility that the sniffer will accept data that doesn't correspond to any open connection.

Worse still, ID systems that synch on data and are strict about sequence number checking can be desynchronized by an attacker who pollutes the observable connection state with forged data before initiating her attack.

Using SYN Packets A potential antidote to this problem is to allow the IDS to synch on data, but have it pay attention to 3WH packets that occur sometime after it starts recording data. These systems will initialize connection state from the first observed data packets, but will re-initialize themselves if they see evidence that a real 3WH is being performed (the 3WH is then presumed to set the real state, and previous state and data recorded should be regarded as intentionally faked).

It is important that this technique be implemented reliably. Because the process of combining data synchronization with handshake synchronization necessarily allows the monitor to resynchronize the connection based on some packet input, poor implementations can result in TCP session monitors that can be desynchronized (due to falsely injected 3WH packets) at will by an attacker.

One poor implementation strategy relies solely on client SYN packets to resynchronize the connection. If a SYN packet is received sometime after the TCB is opened, the IDS resets the appropriate sequence number to match that of the newly received SYN packet. An attacker can inject fake SYN packets at will; all she needs to do is send a SYN packet with a completely invalid sequence number, and the IDS will be desynchronized. Legitimate data being exchanged on the connection will no longer (as far as the IDS is concerned) have valid sequence numbers, and the IDS, discarding the valid data, will be blinded.

One simple way to address this problem is to only accept the first SYN packet seen on a connection. Presumably, this will be the legitimate three-way handshake packet, and not a forged desynch attempt.

This does not work. There are three major problems with this approach: the IDS remains vulnerable to desynch attacks on connections that start before the program does (it never examines the original 3WH, so no legitimate SYN will ever appear on the connection), the IDS has no reliable way to determine whether any given SYN is in fact the first SYN to appear on the connection (packet drops complicate this), and, most importantly, an attacker can permanently desynchronize the connection by inserting an invalid SYN packet before the legitimate connection starts.

A better approach is to rely on SYN+ACK packets to resynchronize. As long as the attacker can't forge a valid looking SYN+ACK packet from the server, the IDS can make the assumption that SYN+ACKs from the server are legitimate and represent real connection handshakes.

There are problems associated with this too. If the IDS is observing a stream of data, for which it has not yet detected a three-way handshake, it does not necessarily know which host is the client and which is the server. The observation of a 3WH determines which end is the client and which is the server. An attacker can forge a SYN+ACK packet that makes it appear like her end of the connection is the server; if the IDS cannot determine correctly whether that is the case, it will be desynchronized.

Ignoring SYN Packets A TCP monitor need not resynchronize on 3WH packets; SYN packets can be ignored entirely, and data be used as the basis for sequence number initialization. If this is implemented in a naive fashion, any forged data packet can potentially desynchronize the connection. A smarter implementation might only consider (for synchronization purposes) data packets that originate from local hosts, assuming that the attacker cannot forge packets appearing to come from these hosts.

5.4 TCP Stream Reassembly

The most difficult task for a network intrusion detection system to accomplish is the accurate reconstruction of the actual data being exchanged over a TCP connection. TCP provides enough information for an end-system to determine whether any piece of data is valid, and where that data belongs in the context of the connection. Even so, the 4.4BSD code to manage this process is over 2000 lines long, and is some of the most involved in the entire TCP/IP protocol implementation.

The end-points of a connection have a distinct advantage over an observing monitor — if they miss data, the other side of the connection will automatically retransmit it after some period of time. Both participants of the connection can actively manipulate the other, to ensure that their data is exchanged correctly.

The TCP session monitor does not have this luxury. If it misses a packet, it cannot (practically) request retransmission — moreover, it cannot easily detect whether a missing piece of data is due to out-of-order packet arrival or a dropped packet. Because the IDS is strictly a passive participant in the connection, it is quite easy for it to miss data.

This problem is made even more acute by the fact that proper reassembly of a stream of TCP packets requires accurate sequence number tracking. If an IDS misses enough packets, it can potentially lose track of the sequence numbers. Without some recovery mechanism, this can permanently desynchronize the connection. The techniques used by an IDS to recover from packet loss (and resynchronize with the connection) can also be attacked.

5.4.1 Basic Reassembly Problems

Some ID systems do not use sequence numbers at all. Instead, they insert data into the "reassembled" stream in the order it is received. These systems do not work. An attacker can blind such a system simply by accompanying her connection with a constant stream of garbage data; the output of the monitor's TCP driver will be meaningless.

These systems do not work even on normal TCP streams. The arrival of TCP segments out of order is a normal occurrence (happening whenever the route between TCP endpoints changes and reduces the latency of the path between them)[18]. Unfortunately, when this happens, the ID system does not correctly re-order the packets. The output of the system is again inaccurate. Of course, an attacker could also send her stream of data out of order; the end-system will correctly reassemble, and the effectively crippled IDS will see meaningless data.

5.4.2 Challenges To Reassembly

Even if the system does check sequence numbers, there is no assurance that a given segment (even with correct sequence numbers) will be accepted by the end-system to which it is addressed. Several issues can cause a TCP implementation to drop properly sequenced data. The simplest of these are the IP and TCP insertion problems, but other, higher-level issues present problems as well.

One major problem the IDS must cope with is each end-system's advertised window. The "window" of a connection represents the number of bytes of data it will accept, preventing the other end of the connection from sending too much data for it to buffer. Data sent past the window is discarded. In addition, the time at which the IDS detects the change in the window is different from the time at which the end-system detects the change and reacts to it. Packets that arrive within the period of time that the IDS and the end-system are inconsistent can cause problems. An IDS that does not account for this in some manner is potentially vulnerable to an insertion attack.

The information available to the IDS from captured packets provides one useful indication of end-system state — the acknowledgment sequence number. The acknowledgment number represents the next sequence number an end-system expects to see. Presumably (end-system TCP bugs can break this assumption), any valid piece of data will eventually be acknowledged by an ACK message.

It may be apparent at this point that an IDS can reliably monitor a stream simply by waiting for acknowledgment before acting on a piece of data. This is not as easy at it may seem. The acknowledgment number is cumulative; it represents the next expected piece of data within the context of the entire connection. Every segment sent is not necessarily directly acknowledged — even though an acknowledgment is generated in response to it. Several segments

Operating System	TCP Overlap Behavior
Irix 5.3	Favors New Data for Forward Overlap
HP-UX 9.01	Favors New Data for Forward Overlap
Linux	Favors New Data for Forward Overlap
AIX 3.25	Favors New Data for Forward Overlap
Solaris 2.6	Favors New Data for Forward Overlap
FreeBSD 2.2	Favors New Data for Forward Overlap
Windows NT 4.0	Always Favors Old Data

Figure 14: TCP Overlap Behavior in Various Operating Systems

worth of data can be acknowledged by one ACK; an IDS cannot simply wait for an acknowledgement to each individual packet it sees.

Another great problem in IDS stream reassembly is the fact that an attacker can send several identically sequenced packets with varying data. The header information will not change from packet-to-packet (except the checksum), and each packet will alter end-system state in exactly the same manner, but only one of the packets will actually be processed by the destination host. Unfortunately, only the end-system knows which one was actually processed. There is not enough information exchanged on the wire for a IDS to determine which packet was valid.

Worse still, an insertion attack against an IDS coupled with this ambiguity can allow an attacker to determine which packets will be accepted by the IDS, by sending segments that the end-system will reject without acknowledging, and then sending valid packets after some brief delay. The IDS will most likely accept the bad data and move the sequence space forward, causing it to ignore the valid data and potentially desynchronizing the IDS from the actual connection. This is very similar to the TCP hijacking attack described by Laurent Joncheray[14].

5.4.3 Overlap

Like IP fragments, TCP segments can arrive out of order and in varying sizes. As in IP fragmentation, this can cause new data to overlap old data. As always, if the IDS does not resolve this problem in a manner consistent with the hosts it's watching, it will not accurately reassemble the stream of data.

The rules for handling TCP segment overlap are quite similar to those of reassembling fragmented IP datagrams. In some cases, end-systems will resolve the conflict in favor of the old data; in others, the conflict is resolved in favor of the new data. There is, again, a great potential for bugs here, and, as in IP reassembly, a bug on either the end-system or the IDS is exploitable by the attacker. Figure 14 details the overlap resolution behavior of various operating systems.

Using overlapping TCP segments, it is possible for an attacker to create a stream of packets that will assemble to a completely innocuous string if sent alone, or to an attack signature if it's accompanied by a single overlapping segment. Playing with segment overlap allows the attacker to literally rewrite the packet stream on the destination host, and, unless the IDS resolves overlap in exactly the same manner as the end-system, it will not see the attack.

5.4.4 Endpoint TCP Overlap Bugs

As in IP fragmentation overlap resolution, there is a large potential for inconsistency of implementation between vendors in TCP reassembly code. As an example, Windows NT resolves conflicts in out-of-order TCP segments consistently in favor of the old data, and 4.4BSD resolves conflicts as indicated in the RFC, occasionally in favor of the new data. As with fragmentation reassembly, unless the IDS knows how each system on the network reassembles streams containing conflicting segments, it will be unable to accurately monitor certain types of end-systems.

5.4.5 Summary of Reassembly Issues

These issues do not present a great problem for most connections; most of the TCP segments in a normal connection arrive in-order, and there aren't any fake TCP segments injected into the stream specifically to confuse the IDS. However, in the real world, an attacker trying to evade an IDS will attempt to make the TCP stream as hard to monitor as possible, and will stretch the limits of the protocol to do this.

Vulnerabilities in IDS TCP reassembly code are insidious because they are not immediately obvious; a specific problem may manifest itself only when the IDS is given some pathological sequence of inputs. The majority of the time, the IDS may appear to be reassembling TCP streams perfectly. Testing IDS TCP implementations for problems is time consuming and expensive; it's easy for a vendor to skip this testing almost entirely.

5.5 TCB Teardown

The TCB teardown policies of an IDS determine the point at which the system ceases recording data from a connection. TCB teardown is necessary because the state information required to track a connection consumes resources; when a connection ceases to exist, it no longer makes sense to dedicate resources to tracking it. A system that did not destroy old TCBs at some point would be trivially defeatable, simply by flooding it with meaningless connections until it ran out of resources to track future connections.

In TCP, connections close after they're explicitly requested to do so. Two TCP messages (RST and FIN) exist specifically to terminate a connection. Barring sudden crashes on both endpoints, TCP connections are only terminated by the exchange of these messages. Because TCP explicitly provides notification of terminated connections, it may be logical to design an IDS that uses these messages to decide when to close a connection TCB.

This is not enough to adequately manage the per-connection resource problem. TCP connections do not implicitly "time out". A connection can be alive without the exchange of any data indefinitely. TCP provides a mechanism to ensure that both hosts are alive, by periodically exchanging messages, but this mechanism is not commonly used and takes far too long to recognize dormant connections to be of practical use. Without some method to time out arbitrary dormant connections, the IDS remains attackable simply by flooding it with connections that do not explicitly terminate.

The problem with TCB teardown is that an IDS can be tricked into tearing down a connection that is still active, and thereby force the system to lose state. Within the context of a pattern matching engine, this means that the stream of input abruptly terminates. An attacker that can induce the incorrect termination of the TCB tracking her can prevent pattern matching from working by abruptly halting pattern matching before the complete attack signature passes across the network.

On the other hand, an IDS that fails to tear down a TCB for a connection that really has closed is also vulnerable; as soon as the connection is legitimately closed, its parameters can be re-used for a new connection with completely different sequence numbers (technically, the systems must wait for a period of time before reusing connection parameters [12] — not all operating systems enforce this). In the absence of synchronization recovery techniques, this can completely blind the IDS to entire connections.

Because an ID system's TCB teardown policies can be attacked, their design is relevant to our discussion. We've identified a few options that can contribute to how an IDS ceases to track connections, and will discuss their ramifications here. This is by no means an exhaustive summary of all the possible options.

5.5.1 Using TCP Connection Teardown Messages

One possible way for an IDS to determine when to stop tracking a connection is to listen for TCP control messages that indicate the connection is being shut down. Doing so allows an IDS to quickly recover resources for connections that have actually terminated, and also prevents desynchronization for new connections using the same parameters. Unfortunately, because some connection termination request messages may be under the control of an attacker, there is significant risk involved in trusting these messages.

TCP provides two connection teardown messages. The first message allows for "orderly" connection teardown, where both sides of the connection acknowledge the end of the connection and ensure that their data is completely sent before the connection closes. The second message abruptly terminates a connection due to error.

FIN Processing TCP provides orderly teardown via the FIN message. A system sending a FIN message is indicating that it has finished sending data, and is ready to close the connection. FIN messages are acknowledged, and each side of the connection sends a message to shut it down.

In the presence of inside-outside filtering, FIN messages are reliable indicators of terminated connections. A connection is not completely terminated until both sides send a FIN message, and acknowledge the other side's message. An attacker cannot fake the FIN shutdown of a connection without forging packets that appear to come from the server.

RST Processing It's not enough for an IDS to rely on FIN messages to terminate connection TCBs. TCP provides a method to abruptly notify the other end of a connection that the connection has been closed, using the Reset (RST) message. RST segments are not acknowledged; the only way to know if an RST message has been accepted by an end-system is to see if it continues sending data on the connection. The only way to do this practically within an IDS is to time the connection out after seeing an RST; however, this means that an IDS can potentially mistakenly shut down a connection that is alive but dormant.

The RST problem is more severe due to end-system TCP bugs. Technically, an RST message is only valid if it is correctly sequenced — RST messages with spurious sequence numbers (which can be created by an attacker in an effort to illicitly tear down connections) should be ignored. Not all operating systems check the sequence number on RST messages.

5.5.2 Relying on Timeouts for TCB Teardown

An alternative to using TCP connection teardown messages is to simply time connections out when they become dormant for some threshold time period. This prevents the IDS from being fooled by false TCP teardown messages, and potentially simplifies the IDS TCP code.

There is a cost to this simplicity — systems that rely on timeouts for TCB teardown can easily be circumvented. In what has been termed the "Sneakers" attack (after the famous suspense movie, where Robert Redford evades a sophisticated alarm system by employing a similar technique), the attacker renders the sum of her movements undetectable to the IDS by waiting for the IDS to time out between packets.

The Sneakers attack is particularly troublesome because, as we noted previously, the IDS *must* employ some form of connection timeout TCB teardown, as dormant TCP connections can remain established for far longer than the IDS can devote resources to track them. If an attacker can induce this timeout, either by waiting long enough or by filling the IDS with enough interesting (but meaningless) connections that it is forced to garbage-collect older connections, she can potentially evade the IDS by causing it to lose state.

Additionally, systems which completely ignore TCP teardown messages can be desynchronized when the connection is legitimately closed. Even though the connection has ceased to exist, the IDS maintains a TCB for it until it times out. If a new connection occurs using the same parameters before the connection times out on the IDS, the system will be desynchronized, due to the use of different sequence numbers on the new connection.

This attack can be carried out without any specialized code; an attacker simply uses "telnet" to create a connection, closes the connection, and re-opens it. If the sequence numbers on her machine change enough between the two connections, a vulnerable IDS will not be able to track the second connection.

6 Denial of Service Attacks

Denial of service attacks against ID systems are severe because, by their very nature, passive ID systems "fail open" — unlike a good firewall, access to the network isn't cut when a monitor system becomes unresponsive. A basic goal, then, for an attacker is to cause the IDS to fail without losing access to the machines being attacked.

Some denial of service attacks exist due to buggy software. An IDS that crashes when it receives a certain bad packet, or a series of bad control messages, or anything else that can be cued by a remote attacker, can be defeated instantly. Fortunately, these kinds of bugs are quickly and easily fixed by vendors. Unfortunately, finding all such bugs requires painstaking software audits.

It is also interesting that some ID systems can themselves be used to launch denial of service attacks on other systems. An ID system that includes a countermeasure capability, such as the ability to set packet filters in reaction to an attack, can be fooled via false positives (due to forged attacks) to react to attacks that haven't actually occurred.

6.1 Resource Exhaustion

There are many different types of denial of service attacks that are valid against ID systems. The attacks we'll discuss here all involve resource exhaustion — the attacker identifies some point of network processing that requires the allocation of some sort of resource, and causes a condition to occur that consumes all of that resource. Resources that can be exhausted by an attacker include CPU cycles, memory, disk space, and network bandwidth.

The CPU processing capabilities of an IDS can be exhausted because the IDS spends CPU cycles reading packets, determining what they are, and matching them to some location in saved network state (for example, an IP fragment needs to be matched to the other fragments of the datagram it represents). An attacker can determine what the most computationally expensive network processing operations are, and force the IDS to spend all its time doing useless work.

ID systems require memory for a variety of things. TCP connection state needs to be saved, reassembly queues need to be maintained, and buffers of data need to be created for pattern matching. The system requires memory simply to read packets in the first place. As the system runs, it allocates memory as needed to perform network processing operations (for example, the receipt of an IP fragment means that the ID system will need to obtain memory to create and maintain an IP fragment queue for that packet). An attacker can determine which processing operations require the ID system to allocate memory, and force the IDS to allocate all its memory for meaningless information.

At some point, most ID systems will need to store logs of activity on disk. Each event stored consumes some amount of disk space, and all computers have a finite amount of disk space available. An attacker can create a stream of meaningless events and, by having them continually stored, eventually exhaust all disk space on the IDS, which will then be unable to store real events.

Finally, network ID systems track activity on the networks they monitor. For the most part, they are capable of doing this only because networks are very rarely used to their full capacity; few monitor systems can keep up with an extremely busy network. The ID system, unlike the end-systems, must read everyone's packets, not just those sent specifically to it. An attacker can overload the network with meaningless information and prevent the ID system from keeping up with what's actually happening on the network.

Other resources exist as well, depending on the design of the system. For instance, in systems that set router filters in response to attacks, we must consider the fact that the router has a limited capacity for storing filter entries; at some point, the router's filter storage will be completely consumed, and the system will be unable to add new entries. An ID system that doesn't take this into account can be defeated by forcing it to spend the router's filter storage on reactions to fake attacks.

The basic problem with resource consumption on an IDS is that the system must simulate the operation of all the machines it's watching, in order to track what's actually occurring on them. The end-systems themselves only need to concern themselves with network traffic that directly involves them. The IDS, which is spending more resources coping with the network than any other system on the network, is thus inherently more prone to resource starvation attacks than the end-systems.

This problem is exacerbated by the fact that most network ID systems operate in "promiscuous" mode, reading all traffic off the wire, regardless of its destination. Resources can be consumed on the IDS by the processing of traffic that isn't even destined for a real machine; apart from the network bandwidth consumed by this traffic, no other system on the network will be affected by this. Again, performance on the IDS is degraded to an greater extent than on the end-systems it's trying to track, making it more difficult for the IDS to keep up and giving the attacker an edge.

6.1.1 Exhausting CPU Resources

An attacker's goal in exhausting an ID system's computational capability is to prevent it from keeping up the network. A CPU-starved IDS will not process captured packets quickly enough and, as these packets fill the buffering capacity of the operating system, captured data starts being dropped.

An example of why this occurs is useful. On 4.4BSD Unix, packet capture is accomplished through the "Berkeley Packet Filter" (BPF) device. BPF interacts directly with low level network drivers (such as the Ethernet interface driver), taking snapshots of packets before they're handed up to the IP layer for processing. As packets are captured by BPF, they are stored in a kernel buffer, where they stay until an application reads them out.

If an application doesn't read data out of the buffer faster than the buffer is filled up by newly captured packets, space for queuing up captured packets runs out. When this happens, captured packets are necessarily dropped before the application ever has a chance to examine them.

An attacker can prevent an ID system from keeping up with packet capture by forcing it to spend too much time doing useless work. In order to do this, the attacker must identify operations that she can force the IDS to perform that consume large amounts of processing time.

In many ID systems, this is easy; inefficient algorithms are used to process, save, and look up state about network traffic. The attacker can cause the system to process information that forces these algorithms to work in their worst-case conditions.

A concrete example of this is IP fragmentation. As IP fragments arrive, they must be stored, until all the related fragments arrive. To facilitate reassembly, most systems store fragments in the order that their data will appear in the final packet. This means that, as each fragment arrives, the system needs to locate the correct fragment storage area, and then find the right place in that area to store that specific fragment.

Many systems use a simple ordered list to store incoming fragments. As new fragments arrive, the system must locate the correct list for that packet, and then do a full linear lookup to determine whether the new fragment was already received and, if not, where in the list the fragment should go. As new fragments arrive, this list gets longer, and the time required to look up fragments in the list increases. An attacker can force this process to operate in its worst case by sending large amounts of traffic using the smallest possible fragments — large amounts of CPU cycles will be consumed tracking tiny IP fragments.

Some protocol parsing can be expensive by itself. An IDS that needs to somehow analyze encrypted traffic may spend a large amount of time simply decrypting packets (encryption and decryption can be extremely expensive operations). While the demand for this kind of processing is not now very great, it will increase as technologies such as IP-sec[11] are deployed.

6.1.2 Exhausting Memory

ID systems require memory to operate. Different types of protocol processing have differing memory requirements. An attacker that can force an IDS to consume all available memory resources can render the system nonfunctional; the system may simply quit abruptly when it runs out of memory, or it may thrash trying to squeeze more space out of slow virtual memory systems, causing the same effects as CPU exhaustion.

An attacker trying to exhaust memory on an IDS examines the system, trying to determine the points at which the system allocates memory. The attacker attempts to isolate network processing events that cause the system to allocate memory for a long duration of time; the attacker then induces this processing by sending packets that the IDS will be forced to process in that manner. After being flooded with such packets for some time, the IDS will run out of memory to process the incoming packets.

Some ID systems employ "garbage collection" to automatically reclaim memory that is not being actively used. Unfortunately, used incorrectly, garbage collection can present its own problems. A garbage collection system that isn't aggressive enough in reclaiming memory will not be able to keep up with demand, and will only slow down memory exhaustion attacks. A garbage collection system that is too aggressive will consume memory that is needed for real processing, causing the system to incorrectly process network traffic.

Examples of attackable memory allocations include TCP TCB creation (the attacker creates a flurry of connections to various hosts on the ID system's network, or, using packet forgery, creates a flood of entirely fake connection) and TCP reassembly (the attacker sends large amounts of traffic in streams of out-of-order data that will need to be reassembled, forcing the system to consume memory not only for the data but also for reassembly queues).

6.1.3 Exhausting Network Bandwidth

Perhaps the simplest way to starve an IDS of resources is simply to create too much raw network traffic for the system's low-level network interface to keep up with. As each packet arrives, the interface must copy the packet off the wire and into a buffer, interrupt the system, and cause the system to copy the packet into the kernel. The interface is capable of handling only a limited amount of traffic before it is overwhelmed by the load and starts dropping incoming packets.

Although modern network interfaces operate efficiently enough to keep up with drastically high network loads, older hardware cannot do so. The point at which old ISA-bus based network interfaces become saturated is drastically lower than the point at which the network media itself becomes saturated. If an attacker creates enough traffic, she can prevent such interfaces from keeping up without saturating the network itself.

Targeted packet floods can also work in some circumstances. On switched networks, it's possible to create large amounts of traffic that will only be seen by certain systems. If an attacker can create a flood of packets that will only be switched to the IDS, she can flood the IDS while maintaining the ability to communicate with the machines she's attacking.

This type of attack is closely related to CPU exhaustion, and, indeed, many times the system will run out of CPU cycles long before the network interface is saturated. Regardless of which component of the system fails first, the effect is the same for the attacker; the IDS cannot keep up with the network, and misses significant packets.

6.2 Abusing Reactive ID Systems

In some circumstances, the IDS itself can become an instrument of denial of service attacks. If the IDS has a "reactive" countermeasure capability, and is vulnerable to attacks that create false positives, it can be forced to react to attacks that don't actually exist. The countermeasures employed can be subverted to completely block access for legitimate traffic, or to shut down valid connections. In these cases, the reactive capabilities of network ID systems are actually doing more harm than good.

The most basic problem with reacting to attacks discovered by monitoring IP traffic is that the IP addresses are not always trustworthy. An attacker can forge traffic appearing to come from almost any IP address, and, if this traffic appears to contain an attack, the ID system may react to it. In some circumstances, this is very easy to do.

For example, many attacks occur over "connectionless" protocols, for which the attacker doesn't need to see the responses to her packets. Instead, she simply creates and blindly sends forged packets, and the IDS is fooled into believing that the attack is coming from somewhere that it isn't. Good examples of this include ICMP ping floods, SYN floods, "death" packets (such as the ping-of-death attack involving large ICMP echo requests), and UDP packet storms.

Even attacks that involve TCP connections can be faked if the IDS doesn't correctly identify the three-way handshake. If the IDS doesn't require a handshake at all before recording data, TCP attacks can be faked as easily as ping floods; even if it does, the specific manner in which it tracks handshakes can be attacked for the same effect.

The essential issue here is that the attacker can trigger alarms about events occurring from fake addresses. The IDS, which has no idea what the "real" source of the attack was, reacts falsely to the forged events by restricting connectivity to the faked addresses. The addresses used by the attacker can be specifically chosen to maximally affect overall connectivity (for example, the attacker can cut off access to all the network's DNS servers).

The amount of damage that can be caused by such attacks depends on the manner in which the IDS reacts to attacks in general. Some ID systems limit themselves to shutting down TCP connections that appear to be vehicles of attack; these systems can be abused to shut down legitimate connections (by forging traffic that makes it appear that an attack is being performed using those connections), but cannot easily be abused to impact overall connectivity, unless specific TCP connections are vital for the network's connectivity (for instance, BGP4 routing).

Other systems have more effective ways to react to attacks; they modify router filters on the fly to cut all traffic from sites that appear to be originating attacks. These systems pay for that extra power by being vulnerable to more damaging denial-of-service subversions; an attacker that can cause the IDS to recognize false attacks can cut all access of to critical network resources by strategically forging addresses.

Regardless of what countermeasures are actually employed, it is important to realize that such facilities are dangerous as long as an attacker can forge attacks. Some types of attacks may never be a legitimate basis for deployment of countermeasures, simply due to the fact that they can be performed blindly using forged addresses. Other attacks can only be safely reacted to if the IDS has a rock-solid network processing implementation.

7 Methodology

We support our assertions regarding vulnerabilities in ID systems with the results of extensive tests against actual, commercially available intrusion detection systems. The purposes of these tests were to ascertain characteristics of each subject' s TCP/IP implementation, and to provide concrete examples of actual attacks that could be performed against them. Our tests were designed to be easily repeatable, and to illustrate in the most obvious possible manner the deficiencies of each tested system.

7.1 Overview

Each of our tests involve injecting packets onto a test network, on which the subject ID system was running. By tracking the subject's administrative console output, we were able to observe many characteristics of the system's underlying TCP/IP implementation. To this extent, all of our tests involved consideration of the subject as a "black box". All our tests involved the TCP protocol.

In most cases, the tests involved interactions between our injected packets and a third host, representing a hypothetical "target" of attack. In each test, this target host was the explicit addressee of all of our packets. The presence of the target host allowed us to easily create "real" TCP connections for the subject IDS to monitor.

In addition, the target host also acted as a "control" for our experiments. The target's reactions to our injected packets allowed us to observe empirically the behavior of a "real" TCP/IP implementation, and contrast that behavior to the deduced behavior of the subject IDS.

All of our tests involved mimicking a "PHF" webserver attack. The PHF attack exploits a specific Unix CGI script ("phf") to attempt to gain access to a webserver. We used PHF because the attack is detected by all our subject ID systems, and because the attack is easily reproduced using standard TCP network tools (like "telnet"). In order to reproduce a PHF attack, we sent the string "`GET /cgi-bin/phf?`" to the target host.

In each test, we created network conditions that could make it appear as if a PHF attack was being attempted. In each test, the specific packets injected into the network differed subtly. The subject ID system reacted to each test by either reporting or not reporting a PHF attack. By considering the ID system's output and the specific types of packets used for the test, we were able to deduce significant characteristics of the subject IDS.

Before conducting complicated or subtle tests against the subject, we conducted a series of "baseline" tests. The purpose of these tests was to ensure that the subject IDS was configured properly and was functioning at the time our tests were conducted, and that the IDS did in fact detect a PHF attack based on our PHF reproduction string.

In almost all test cases, a process on the target host ran which accepted incoming TCP connections on the HTTP port and printed any input obtained from the machine's TCP stack. By examining the output of this process, we were able to deduce whether the subject IDS should have detected the attack based on the network conditions we created.

7.2 Tools Used

The primary tool we employed in our tests was CASL, a specialized scripting language developed at Secure Networks, Inc. that allows for programmable generation and capture of raw packets. Each of our tests used a CASL script to inject packets onto the network, and, in most cases, read and parse the responses. A more detailed overview of CASL is provided in [15].

Our target host ran FreeBSD 2.2, an implementation of 4.4BSD. The 4.4BSD TCP/IP stack is one of the best documented and most easily obtainable IP implementations available, and FreeBSD is by far the most popular BSD implementation. FreeBSD 2.2 was, at the time of our testing, the most recent "stable" release of the operating system.

For each test, we used Hobbit's "netcat" tool[16] to listen on TCP port 80 and print the input from the target host's TCP stack. Hobbit's tool is an all-purpose, bare-bones diagnostic program that is widely available, popular, and documented; in its "listening" mode, the tool simply accepts an incoming connection, and prints each character of data the TCP driver presents to it.

As we ran each test, we observed the specific packets being transmitted on the network using LBL "tcpdump"[19]. Tcpdump is a low-level network diagnostic tool that passively monitors networks in promiscuous mode, and prints summaries of each captured packet. We ran the "tcpdump" tool from the test platform on the first execution of each specific test script. Tcpdump provided us with IP-level packet traces to accompany our test results, which made it easier to discern exactly what was happening on the network during each of our tests.

Our test network was non-switched 10BaseT Ethernet. The hosts on the network included the IDS, the target host, and the test platform. The network was dormant at the time we conducted our tests.

7.3 Test Execution

Each of our tests involved a CASL script, run from an interpreter on the test platform, which generated and injected packets addressed to the target host. We define each of these tests in terms of the script's name, its specific network interactions, the IDS characteristic it attempts to ascertain, and its validity to the 4.4BSD TCP/IP driver (that is, whether our target host completely and accurately reconstructed the PHF string our test attempted to send).

A test that was not "valid" to 4.4BSD should not have resulted in the detection of a PHF attack by the subject IDS. We suggest that the subject IDS should not detect attacks in "invalid" tests, and should reliably detect attacks within the valid ones.

In cases where the IDS failed to detect an attack in either type of test, we re-initialized the IDS and re-ran the test multiple times. Before concluding that a subject IDS was not detecting our attack signatures, we re-ran the baseline test to confirm its operational integrity, and immediately ran the considered test.

7.4 Test Definitions

Name	**baseline-1**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment.
Behavior Tested	Is the IDS configured properly, and does our test string adequately reproduce a PHF attack to the subject?
Target Validity	Valid
Name	**baseline-2**
Operation	Complete a TCP handshake, send the test string in a series of ordered, 1-character TCP data segments.
Behavior Tested	Is the IDS configured properly, and does our test string adequately reproduce a PHF attack to the subject?
Target Validity	Valid
Name	**frag-1**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment which is broken into 8-byte IP fragments and sent in order.
Behavior Tested	Does the subject IDS perform IP fragment reassembly at all?
Target Validity	Valid
Name	**frag-2**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment which is broken into 24-byte IP fragments and sent in order.
Behavior Tested	Does the subject IDS perform IP fragment reassembly at all?
Target Validity	Valid
Name	**frag-3**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment which is broken into 8-byte fragments, with one of those fragments sent out of order.
Behavior Tested	Can the subject IDS handle basic out-of-order IP fragmentation reassembly?
Target Validity	Valid

Name	**frag-4**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment which is broken into 8-byte fragments, with one of those fragments sent twice.
Behavior Tested	Can the subject IDS handle reassembly when fragments are completely duplicated?
Target Validity	Valid

Name	**frag-5**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment broken into 8-byte fragments, sent completely out of order and with an arbitrary duplicated fragment.
Behavior Tested	Can the subject IDS handle reassembly in pathological (but correct) cases?
Target Validity	Valid

Name	**frag-6**
Operation	Complete a TCP handshake, send the test string in a single TCP data segment which is broken into 8-byte fragments, sending the marked last fragment before any of the others.
Behavior Tested	Does the subject IDS correctly wait for all fragments to arrive before attempting reassembly?
Target Validity	Valid

Name	**frag-7**
Operation	Complete a TCP handshake, send a stream of fragments containing the signature string with the word "GET" replaced with the string "SNI". Send a forward-overlapping fragment rewriting the "SNI" back to "GET" on the target host.
Behavior Tested	Does the subject IDS correctly handle forward overlap in IP fragments?
Target Validity	Valid

Name	**tcp-1**
Operation	Complete a TCP handshake, simulate the disconnection of the target host from the network, and send the target string in a series of 1-byte TCP data segments.
Behavior Tested	Does the subject IDS wait for TCP acknowledgment from the target before acting on data from captured packets?
Target Validity	Inapplicable

Name	**tcp-2**
Operation	Complete a TCP handshake, send the test string in a stream of 1-byte TCP data segments where the sequence number wraps back to zero.
Behavior Tested	Does the IDS correctly deal with wrapped sequence numbers?
Target Validity	Valid

Name	**tcp-3**
Operation	Complete a TCP handshake, send the test string in a stream of 1-byte TCP data segments, duplicating entirely one of those segments.
Behavior Tested	Does the IDS correctly handle completely duplicate TCP segments?
Target Validity	Valid

Name	**tcp-4**
Operation	Complete a TCP handshake, send the test string in a stream of 1-byte TCP data segments, sending an additional 1-byte TCP segment which overlaps a previous segment completely but contains a different character.
Behavior Tested	Does the subject IDS correctly handle duplicate TCP segments?
Target Validity	Valid

Name	**tcp-5**
Operation	Complete a TCP handshake, send the test string, with the letter 'c' replaced with the letter 'X', in a series of 1-byte TCP data segments. Immediately send a 2-byte TCP data segment that overlaps (forward) the modified letter, rewriting it back to 'c' on the target host.
Behavior Tested	Can the subject IDS handle overlap in out-of-order TCP streams?
Target Validity	Valid

Name	**tcp-6**
Operation	Complete a TCP handshake, send the test string in a series of 1-byte TCP data segments, and increase the sequence number by 1000 midway through the stream.
Behavior Tested	Does the IDS "recover" from sudden changes in the sequence number?
Target Validity	Invalid

Name	**tcp-7**
Operation	Complete a TCP handshake, send the test string in a series of 1-byte TCP data segments, interleaved with a stream of 1-byte data segments for the same connection but with drastically different sequence numbers.
Behavior Tested	Does the subject IDS check sequence numbers during reassembly?
Target Validity	Valid

Name	**tcp-8**
Operation	Complete a TCP handshake, send the test string in a series of 1-byte TCP data segments, with one of those segments sent out of order.
Behavior Tested	Can the subject IDS handle basic out-of-order TCP reassembly?
Target Validity	Valid

Name	**tcp-9**
Operation	Complete a TCP handshake, send the test string in a series of 1-byte TCP data segments, sent in random order.
Behavior Tested	Can the IDS handle pathological out-of-order TCP reassembly?
Target Validity	Valid

Name	**tcbc-1**
Operation	Do not complete a TCP handshake, but send the test string in a series of 1-byte TCP data segments as if a handshake had occurred for some arbitrary sequence number.
Behavior Tested	Does the IDS require a handshake before it will start recording data from a connection?
Target Validity	Invalid

Name	**tcbc-2**
Operation	Complete a TCP handshake, send the test string in a series of 1-byte TCP segments, interleaved with SYN packets for the same connection parameters.
Behavior Tested	Does the IDS resynchronize on a SYN packet received after a complete TCP handshake?
Target Validity	Valid

Name	**tcbc-3**
Operation	Do not complete a TCP handshake, but send a stream of arbitrary data at a random sequence number as if one had occurred. Use the same connection parameters to connect with "netcat" and type the test string in manually.
Behavior Tested	Can the IDS be desynchronized due to badly sequenced fake data prior to a real connection initiation?
Target Validity	Valid

Name	**tcbt-1**
Operation	Complete a TCP handshake and immediately shut the connection down with an RST. Re-connect over the same parameters, with drastically different sequence numbers, and send the test string in a series of 1-byte TCP data segments.
Behavior Tested	Does the IDS correctly resynchronize after a connection is legitimately torn down with an RST?
Target Validity	Valid

Name	**tcbt-2**
Operation	Complete a TCP handshake and send the test string in a series of 1-byte TCP data segments. Midway through the stream, tear the connection down with an RST (but continue to send the rest of the data segments).
Behavior Tested	Does the IDS stop recording data when it sees an RST?
Target Validity	Invalid

Name	**insert-1**
Operation	Complete a TCP handshake and send the test string in a series of 1-byte TCP data segments, each with a bad IP checksum.
Behavior Tested	Does the IDS verify the IP checksum on received packets?
Target Validity	Invalid

Name	**insert-2**
Operation	Complete a TCP handshake and send the test string in a series of 1-byte TCP data segments, each with a bad TCP checksum.
Behavior Tested	Does the IDS verify the TCP checksum on received segments?
Target Validity	Invalid

Name	**insert-3**
Operation	Complete a TCP handshake and send the test string in a series of 1-byte TCP data segments, none of which have the ACK bit set.
Behavior Tested	Does the IDS accept TCP data in segments without the ACK bit?
Target Validity	Invalid

Name	**evade-1**
Operation	Complete a TCP handshake, include the test string in the initial SYN packet.
Behavior Tested	Does the IDS accept data in a SYN packet?
Target Validity	Valid

7.5 Summary

Because our tests are scripted, they are well-defined, easily repeated, and fast. After defining and perfecting the test suite, we were able to completely test new ID systems in a matter of minutes. The majority of our testing time was spent defining new tests. Running the individual tests against ID systems took negligible time.

We are in the process of releasing the scripting tool that we used for the tests to the public. When this process has completed, we intend to make the suite of IDS test scripts we've developed available to the public as well. It is our hope that our work can define a framework within which arbitrary network ID systems can quickly be evaluated.

Our test suite is by no means complete; we provide these test results to support the points in our paper, not to define a complete evaluation process for ID systems. With the tools to conduct these tests in the hands of the community, we hope that our tests can be extended to define a more complete test suite.

8 Results

We applied our tests to four of the most popular network intrusion detection systems on the market. In each case, our tests identified serious, exploitable problems in the manner that the IDS reconstructed data transmitted on the network. The results of our tests are not surprising, and we believe that they support the basic points we make in this paper.

In many cases, the ID systems we tested had general problems that precluded them from passing entire collections of specific tests. For example, none of the systems we tested correctly handled IP fragmentation; thus, the systems incorrectly handled all the specific fragmentation tests. We ran every test we could against each ID system.

One of the systems we tested, WheelGroup's NetRanger system, is available only with its associated hardware. We were unable to test this system on our own network, but rather had to obtain the cooperation of an organization already using the product. This prevented us from running many of our tests against this product; NetRanger was the second system we tested, and we added many tests after our first (and only) exposure to the system. One of our tests ("tcp-1") also required us to have access to the local network the test machine was on — we did not have this access for NetRanger.

Another system we test, Network Flight Recorder's NFR system, is not an intrusion detection system per se, but rather a network monitoring engine that can be used to build an intrusion detection system (among many other things). Our results are significant to the usage of NFR as an automated network IDS, but not necessarily to the product as a whole.

It's important to note that the number of "failed" tests each product has is not necessarily an indication of the relative quality of the product. The number of tests each IDS passes is biased heavily based on the presence of specific bugs. Our test suite was not designed to provide a "score" for each product, but rather to highlight specific characteristics about them.

8.1 Specific Results

The systems we tested were Internet Security Systems' "RealSecure" (version 1.0.97.224 for Windows NT), WheelGroup Corporation's "NetRanger" (version 1.2.2), AbirNet's "SessionWall-3" (version 1, release 2, build v1.2.0.26 for Windows NT), and Network Flight Recorder's "NFR" (version beta-2).

We present the overall results from our tests for every IDS in Figure 15. Each individual IDS is described after the table, along with an interpretation of the results.

For each test, a plus sign ('+') indicates that the IDS saw a PHF attack as a result of the packets our test injected. A minus sign ('-') indicates that the IDS reported no attack after we ran our test. A question-mark ('?') indicates that we were unable to perform the test on that product.

8.2 Overviews of Specific ID Systems

8.2.1 ISS RealSecure

ISS RealSecure is an automated network intrusion detection system. We performed our tests on the Windows NT version of the product, although it is available for Unix platforms as well.

RealSecure appears to have two independent network monitor components. The first of these handles signature recognition within captured packets; the second provides a "realtime playback" capability that allows administrators to watch the information being exchanged in a TCP connection in real-time.

We found significant differences between the playback facility and the signature recognition facility. Unlike RealSecure's signature recognition engine, the

Test Name	*Expected Result*	RealSecure	NetRanger	SessionWall	NFR
baseline-1	+	+	+	+	+
baseline-2	+	+	+	+	+
frag-1	+	-	-	-	-
frag-2	+	-	-	-	-
frag-3	+	-	-	-	-
frag-4	+	-	-	-	-
frag-5	+	-	-	-	-
frag-6	+	-	-	-	-
frag-7	+	-	?	-	-
tcp-1	-	+	?	+	?
tcp-2	+	+	+	-	-
tcp-3	+	+	+	+	+
tcp-4	+	+	+	+	+
tcp-5	+	+	+	+	+
tcp-6	-	-	+	+	+
tcp-7	+	-	+	+	+
tcp-8	+	-	-	-	+
tcp-9	+	-	?	-	-
tcbc-1	-	+	-	-	+
tcbc-2	+	+	?	-	-
tcbc-3	+	-	-	+	+
tcbt-1	+	-	?	+	+
tcbt-2	-	+	?	-	+
insert-1	-	+	-	-	+
insert-2	-	+	+	-	+
insert-3	-	+	?	-	+
evade-1	+	+	-	-	+

Figure 15: IDS Test Suite Results

playback system does not appear to sanity check TCP packets before presenting their contents to the user. No sequence number checking was performed in session playback, and out-of-order packets were displayed out of order. An attacker can trivially obscure her actions in RealSecure session playback simply by accompanying her connection with a stream of meaningless, unsequenced TCP packets for the connection; she can also confuse administrators by sending all her packets out of order.

The most significant problem with RealSecure, as with all the other systems we tested, was that it did not handle IP fragmentation reassembly at all. An attacker can completely evade RealSecure by fragmenting every packet she sends across the network.

RealSecure also appeared to have serious problems with TCP reassembly when duplicate segments appeared on the network. RealSecure never detected an attack in any of the tests we ran that involved sending multiple TCP segments with the same sequence number, even though those tests resulted in valid reassembly of the test string on the target host.

RealSecure does not appear to pay attention to TCP RST messages. We were able to desynchronize RealSecure by closing a connection with a client RST message, and then immediately reconnecting using the same parameters. RealSecure recognized attacks in streams even after their connection was reset. RealSecure also does not appear to pay attention to TCP SYN messages; we were able to desynchronize RealSecure from our connections by preceding them with arbitrary data segments with random sequence numbers.

Finally, RealSecure was vulnerable to all of our insertion attacks. It did not appear to check IP or TCP checksums, nor did it verify that the ACK bit was set on TCP data segments.

8.2.2 WheelGroup NetRanger

NetRanger is an automated network intrusion detection system by WheelGroup Corporation. NetRanger interfaces a passive network monitor with a packet filtering router, creating a "reactive" IDS; the ability to respond in realtime to attacks by "shunning" addresses (filtering them at the router) is a major feature of the system.

We had very limited access to the NetRanger system. The hardware requirement (and price) of this system made it impractical for us to obtain our own copy for testing; rather, we relied on the cooperation of an organization already using the product. Because of this, our tests were performed over the global Internet, and we were only able to perform certain tests (due to timing issues). Our test results for NetRanger still showed major problems.

Like all the systems we reviewed, NetRanger (in the version we tested) is completely unable to handle fragmented IP packets. An attacker can evade NetRanger completely by fragmenting all her packets.

We were able to evade NetRanger by injecting duplicate sequenced segments with different data into our connection stream (the "tcp-8" test). NetRanger did not detect data in a SYN packet, so an attacker can evade many of NetRanger's checks by putting crucial data in her initial SYN packet.

We were able to desynchronize NetRanger from our connections by preceding the connection with fake, randomly sequenced data. NetRanger failed to detect attacks in a connection, using the same parameters, that followed this.

Finally, NetRanger was vulnerable to one of our insertion attacks (it doesn't appear to validate TCP checksums). NetRanger did appear to reliably verify IP checksums.

Many of our tests were not performed against NetRanger. We can't conjecture as to whether NetRanger is vulnerable to the attacks those tests measure. Hopefully, these tests can be run against NetRanger in the future.

8.2.3 AbirNet SessionWall-3

SessionWall is an automated network intrusion detection system by AbirNet. We tested the Windows NT version of AbirNet SessionWall-3. Although AbirNet appears to have realtime connection playback capabilities, we were unable to get it working in the evaluation copy we used for our tests.

Of all the ID systems we tested, AbirNet appeared have the most strict network monitoring system. SessionWall-3 did not record data for connections that weren't marked by a three-way handshake. It stopped recording when a connection was torn down with an RST packet. This prevented our TCB desynchronization tests from disrupting the system; however, the strictness of SessionWall's implementation may be attackable as well (insertion of RST packets, for instance, could be used for evasion attacks).

SessionWall validated IP and TCP checksums, and did not accept data without the ACK bit set. It did not appear to wait for acknowledgment before accepting data, however.

We were able to desynchronize SessionWall-3 from our connections by injecting fake SYN packets into our stream; the SYNs were ignored by the endpoint, but SessionWall apparently resynchronized to them and lost pattern matching state. Like NetRanger, SessionWall-3 also failed to detect data in SYN packets. SessionWall did not reassemble overlapping TCP segments in a manner consistant with 4.4BSD, and is thus vulnerable to an evasion attack.

Like all the systems we reviewed, SessionWall-3 is completely unable to handle fragmented IP packets. An attacker can evade SessionWall-3 by fragmenting all her packets.

8.2.4 Network Flight Recorder

NFR is a network monitoring engine by Network Flight Recorder. Unlike the other systems we tested, NFR is not an automated network intrusion detection system; rather, NFR provides a network monitoring component that can be used in a variety of applications. NFR is user-programmable and extensible, and available in source code.

We reviewed NFR because its engine could be used as an automated network intrusion detection system. This is not the intent of the product, and our results do not have significant bearing on NFR's non-security uses. Additionally, because NFR is completely programmable (the product is really an interpreter for a network programming language), it is possible for users of the product to address many of the concerns we bring up in our paper without modifying the underlying engine.

NFR was able to handle IP fragmentation; we verified this in an independent testing process that confirmed NFR's ability to reassemble a fragmented UDP packet (we were able to perform this test because of NFR's available source code). Unfortunately, NFR was unable to handle pattern matching in a TCP stream that was sent in fragmented IP packets; it therefore "failed" all of our fragmentation tests.

NFR, in version beta-2, was vulnerable to all our insertion attack tests. It did not verify IP or TCP checksums, and accepted data without the ACK bit set. NFR detects data in SYN packets.

NFR does not immediately tear down a connection TCB when an RST is seen. We were able to desynchronize NFR by sending spurious SYN packets in our connections, but were unable to successfully desynchronize it with any of our other tests. NFR did not reassemble overlapping TCP segments consistantly with 4.4BSD, and is thus vulnerable to an evasion attack.

9 Discussion

Our tests revealed serious flaws in each system we examined. Every IDS we examined could be completely eluded by a savvy attacker. We have no reason to believe that skilled attackers on the Internet don't already know and aren't already exploiting this fact. Many of the problems we tested for were minor, and easily fixed. The very presence of such vulnerabilities leads us to believe that ID systems have not adequately been tested.

The ability to forge packets, and the ability to "insert" packets into ID systems, makes it fairly trivial for an attacker to forge "attacks" from arbitrary addresses. The ability to react to attacks by reconfiguring packet filters was a major advertised feature of many of the systems we tested. Our work shows that this capability can be leveraged against the system operators by an attacker; these facilities may do more harm than good.

Several of the problems we outline in this paper have no obvious solution. Without adding a secondary source of information for the IDS, allowing it to conclusively identify which packets have been accepted by an end-system, there appear to be ways to create connections that cannot be tracked by passive ID systems. Since the network conditions an attacker needs to induce to elude an IDS are abnormal, an IDS may be able to detect that an attack is occurring; unfortunately, this will be all that an IDS will be able to say.

Regardless of whether a problem is obviously solvable or not, its presence is significant to both IDS users and designers. Users need to understand that the manner they configure the IDS (and their network) has a very real impact on the security of the system, and on the availability of their network. Designers need to understand the basic problems we identify with packet capture, and must begin testing their systems more rigorously.

Finally, the security community (buyers of network ID systems, designers of such systems, as well as interested third parties like us) as a whole can do much to enhance the reliability and security of intrusion detection systems. Additional, independent third-party analysis and testing of ID systems will, to a large extent, define how secure these systems will be in the future.

9.1 Implications to Operators

There are several things that can be done by IDS operators to enhance the overall security of the system as a whole. Additionally, IDS operators need to understand that the outputs of their systems must be read critically; "session playback" data may not represent what's actually occurring in a session, and the source addresses of attacks may not be valid at all.

One critically important step that must be taken before an IDS can be effectively used is to set up "spoof protection" filters, which prevent attackers on the Internet from injecting packets with addresses forged to look like "internal" systems into the network. Bidirectional packet forgery can completely confuse network intrusion detection systems.

It's important to understand that an attacker that successfully breaks into an IDS-protected network probably controls the IDS. An attacker with direct access to the network she's attacking can forge valid-looking responses from systems she's attacking. These forged packets can prevent the IDS from obtaining any coherent picture of what's happening on the network. As soon as an IDS records a "successful" attack on the network, administrators should assume that all bets are off, and further attacks are occurring without the knowledge of the IDS.

An attacker can fool "session playback" facilities into playing arbitrary data back to the operators. Session playback may not accurately represent what's happening inside of a connection. Real-time connection monitoring (when based on an ID system's reconstruction of what's happening in a TCP stream, rather than on printing and recording every packet on the wire) should not be trusted.

Finally, it's of critical importance that ID system operators do not configure their system to "react" to arbitrary attacks. An attacker can easily deny service to the entire network by triggering these reactions with faked packets; ID systems that reconfigure router filters are particularly vulnerable to this, as an attacker can forge attacks that appear to come from important sites (like DNS servers), and cause the IDS to cut off connectivity to these sites.

One possible step that can be taken to mitigate the risk of countermeasure subversion is to allow the system to be configured never to react to certain hosts. None of the systems we tested appeared to allow this type of configuration. Of course, if an attacker can spoof connections from the "untouchable" hosts, she can exploit this to evade countermeasures entirely.

Attacks that can be trivially forged (ping floods, UDP-based attacks, etc.) should not be reacted to; an attacker can, simply by forging packets, cause countermeasures to be deployed that might disrupt the network. Systems that aren't strict about reconstructing TCP sessions (ie, that don't wait for three-way handshakes before recording data) present the same vulnerability for TCP connections as well.

9.2 Implications to Designers

This paper has particularly great relevance to designers of intrusion detection systems, as it outlines in detail many attacks that such systems need to be resistant to. In that sense, this entire paper presents conclusions relevant to IDS designers. However, there are some overall issues that need to be addressed by IDS vendors.

Most of the problems we outline in this paper occur only when very abnormal series of packets are injected onto the network. Overlapping IP fragments or TCP segments are not common; connections consisting entirely of overlapping segments are almost certainly attacks. Even if it's not possible to reliably reconstruct information contained in such streams, it is possible to alert administrators to the presence of the abnormal packets.

Of course, this doesn't work as a design strategy; the value of an IDS is drastically reduced when all it can tell an administrator is "I've detected an attack against this host, but can't tell you specifically what it is." Nevertheless, some information is better than the complete lack of information available now.

The most important issue that vendors need to address is testing. Some of the problems we discovered were so basic (the conditions leading to these problems occur frequently even in normal traffic) that it appeared as if no in-depth testing had been performed at all. We found severe flaws in systems that attempted to address problems — for instance, a program that reassembled fragments, but could not perform signature recognition in packets that had been fragmented.

Testing network intrusion detection systems is not simple. In order to test a network IDS, carefully coordinated streams of forged packets need to be injected onto a network; tools that are capable of doing this in a manner flexible enough to test ID systems are products in and of themselves. Our work defines the beginning of a framework within which ID systems can be tested, and, hopefully, the availability of our tools will increase the ability of vendors to test their systems.

9.3 Implications to the Community

The number of attacks against network ID systems, and the relative simplicity of the problems that were actually demonstrated to be exploitable on the commercial systems we tested, indicates to us that network intrusion detection is not a mature technology. More research and testing needs to occur before network intrusion detection can be looked to as a reliable component in a security system.

Much of this research must be done independently of the vendors. No credible public evaluations of network intrusion detection systems currently exist. The trade press evaluates security products by their features and ease of use, not by their security. Because network intrusion detection is so fragile, it's important that they receive more scrutiny from the community.

Our paper defines methods by which network intrusion detection systems can be tested. It is obvious that our tests can be extended, and that our methodology can be improved. Everyone stands to benefit from such work, and it is hoped that our work can serve as a catalyst for it.

One issue that drastically impacted our ability to test ID systems was the availability of source code. Only one product we reviewed made source code available. Because intrusion detection is so susceptible to attack, we think it's wise to demand source code from all vendors. Products with freely available source code will obtain more peer review than products with secret source code. If our work makes anything clear, it's that marketing claims cannot be a trusted source of information about ID systems.

References

[1] S. Staniford-Chen, "Common Intrusion Detection Framework," `http://seclab.cs.ucdavis.edu/cidf/`

[2] H. S. Javits and A. Valdes "The SRI Statistical Anomaly Detector," In *Proceedings of the 14th National Computer Security Conference*, October 1991.

[3] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip and D. Zerkle, "GrIDS – A Graph-Based Intrusion Detection System for Large Networks," In *The 19th National Information Systems Security Conference*, 1996.

[4] K. L. Fox, R. R. Henning, J. H. Reed and R. P. Simonian, "A Neural Network Approach towards Intrusion Detection," In *Proceedings of the 13th National Computer Security Conference*, October 1990.

[5] P. A. Porras and A. Valdes, "Live Traffic Analysis of TCP/IP Gateways," To appear in Internet Society's *Networks and Distributed Systems Security Symposium*, March 1998.

[6] N. F. Puketza, K. Zhang, M. Chung, B. Mukherjee and R. A. Olsson , "A Methodology for Testing Intrusion Detection Systems," *IEEE Transactions on Software Engineering*, vol. 22, pp. 719-729, October 1996.

[7] M. StJohns, "Authentication Server," RFC 931, TPSC, January 1985.

[8] W. R. Stevens, *TCP/IP Illustrated, Vol 1.* Addison-Wesley, Reading, MA, 1994.

[9] J. Postel, "Internet Protocol - DARPA Internet Program Protocol Specification," RFC 791, USC/Information Sciences Institute, September 1981.

[10] J. Postel, "Internet Protocol - DARPA Internet Program Protocol Specification." RFC 791, USC/Information Sciences Institute, Section 3.2, line 1099, September 1981.

[11] R. Atkinson, "Security Architecture for the Internet Protocol." RFC 1825, Naval Research Laboratory, August 1995.

[12] J. Postel, "Transmission Control Protocol - DARPA Internet Program Protocol Specification," RFC 793, USC/Information Sciences Institute, September 1981.

[13] V. Jacobson, R. Braden and D. Borman, "TCP Extensions for High Performance," RFC 1323, LBL, ISI, Cray Research, May 1992.

[14] L. Joncheray, "A Simple Attack Against TCP," In *5th USENIX UNIX Security Symposium*, June 1995.

[15] Secure Networks, Inc., *Custom Attack Simulation Language (CASL)*, User manual, 1998.

[16] Avian Research, netcat, Available for download at `ftp://avian.org/src/hacks/nc110.tgz`

[17] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," In *7th Annual USENIX Security Symposium*, January 1998.

[18] V. Paxson, "End-to-End Internet Packet Dynamics," In *ACM SIGCOMM '97*, September 1997, Cannes, France.

[19] Lawrence Berkeley National Laboratory, tcpdump, Available for download at `ftp://ftp.ee.lbl.gov/tcpdump.tar.Z`

Thanks

This work would not have been possible without the assistance of many people. Several people gave us valuable input and criticism, and some of our tests would not have been possible without the cooperation of companies running ID systems. We'd like to express our sincere appreciation for this help.

This work was made possible by Secure Networks, Inc. We'd like to thank Alfred Huger, Oliver Friedrichs, and Jon Wilkins for their assistance with this project.

We obtained valuable comments from several of the vendors we reviewed. We'd specifically like to thank Marcus Ranum of Network Flight Recorder, Mike Neumann of EnGarde, and Elliot Turner of MimeStar for their comments and critiques of our technical work.

Vern Paxson of LBL published, as this document was being finished, a paper regarding the design of his real-time network intrusion detection system, "Bro"[17]. His paper details several attacks against network ID systems (many of which we did not catch ourselves). We'd like to thank Mr. Paxson for his extremely valuable input on our own work.

Of course, we appreciate greatly the fact that Network Flight Recorder made their source code available to the public for review. This was a courageous and honorable thing to do (especially in a market as competitive as this), and NFR's approach to source code release is a model that should be followed by other vendors.

Finally, this paper would not have been possible without the assistance of Jennifer Myers at EnterAct, L.L.C., who effectively rewrote our technical results into a coherant document.

About CASL

Our tests were made possible by the development of a security tool called CASL. CASL is a network protocol exploration tool designed to allow security auditors to quickly and easily simulate network events at a very low level. With a minimal amount of effort, CASL can effectively be used to forge any kind of IP packet. With slight programming ability, CASL can be used to perform complex protocol interactions with other networked hosts.

CASL was inspired by tools like Darren Reed's well-known "ipsend" utility, which allowed experimenters to forge a large number of IP packets. However, CASL expands significantly on these types of tools. Some of the benefits of CASL over its predecessors include:

- A complete programming language, with most typical high-level language control constructs (e.g., "if", "while", and "for" statements), and designed to be as easy to learn and use as shell-script languages, but with network programming functionality rivaling that of "C" code.

- The ability to create arbitrary packets — not just the ones we thought up as we designed the program! Unlike some tools, which allow users to to create arbitrary packets by including “raw” data (presumably built with some other tool), CASL allows users to lay out the format of new packet types with an expressive and simple “record” syntax, allowing protocol header fields to be laid out bit-by-bit and byte-by-byte.
- The ability to input packets, reading promiscuously off the wire, and quickly extract information from them. Network reads use familiar “tcp-dump” expressions to select packets, and any number of packets can be read in and examined simultaneously.

CASL is a self-contained, free-standing program that doesn't depend on other network or programming tools to operate. It can be installed quickly, and a CASL script will work on any supported platform. The tool is small, and consumes a fairly low amount of resources; CASL programs can easily share a system with other large applications, and don't consume the large amounts of memory and CPU that general-purpose languages (like Perl and Tcl) tend to.

We designed this tool to meet the needs of two very different audiences: on one hand, CASL is expressive and powerful enough to be a useful tool for experienced, fluent “C” programmers; on the other, it's simple enough to be picked up by a nonprogrammer as quickly as Bourne shell scripting. A CASL script can be little more than a 5 line packet template for users who simply want to forge packets, or it can be tens or hundreds of lines of functional code, with loops, variables, conditionals, subroutines, and other high-level-language capabilities.

We are making CASL available for free for noncommercial use, in the hopes that it can be used to further the state of the art in security research. For more information about CASL, contact Secure Networks Inc.

About Secure Networks, Inc.

Secure Networks, Inc. is a security research and development company located in Calgary, Alberta, Canada. In addition to extensive publically available security research results, Secure Networks also sells security assessment tools. You can find out more about our work at http://www.secnet.com. Secure Networks is reachable via email at “info@secnet.com”, and via telephone at 403-262-9211.