15

Intrusion Detection

> Behold, the fool saith, "Put not all thine eggs in the one basket"—which is but a manner of saying, "Scatter your money and your attention"; but the wise man saith, "Put all your eggs in the one basket and—*watch that basket!*"
>
> —PUDDIN'HEAD WILSON'S CALENDAR

It is important to post sentries near things you wish to protect, and an *intrusion detection system (IDS)* helps perform this function. As commercial products, these security tools have been promoted as the ultimate solution to network intrusions, and many IT managers have proclaimed that their network was secure because they had installed the latest firewall and IDS. These can help, but they're far from a panacea.

There are several types of intrusion detection systems. *Network IDSs (NIDSs)* eavesdrop on network traffic, looking for an indication of an intrusion. Various host-based systems scan files or traffic for incoming viruses; some analyze system call patterns or scan for changed files.

IDSs are plagued by several inherent limitations. False positives (false alarms) occur when an IDS incorrectly concludes that an intrusion occurred. False negatives are actual intrusions that are missed by the IDS. For most intrusion detection systems, both of these are unavoidable and occur with such frequency as to greatly limit their value. It usually requires human intervention to determine evilness or the lack thereof, and some of the sources of weird packets may be too difficult to fix.

Finally, network IDS systems usually work by sniffing the network traffic and gluing the packets together into streams of data. It is easy to do a fair job of this—it seems almost trivial. Most sniffers do just this, but a number of papers, such as [Ptacek and Newsham, 1998], [Paxson, 1998], and especially [Handley *et al.*, 2001], point out that this job is nearly impossible to get exactly right. The problem is that a sniffing program needs to know the states of the TCP/IP stacks at both ends of the communication, plus the idiosyncrasies of their implementation details. For example, suppose that two packets arrive containing overlapping data. It is TCP's job to reassemble the stream of data, and now it has two versions for the overlapping region. Should

Licensed under a Creative Commons Attribution-Non-Commericial-
NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license
https://creativecommons.org/licenses/by-nc-nd/4.0/

it use the first copy or the last? The RFCs are silent on this, and implementations vary. If the overlapping data in the two packets doesn't match, which version should the NIDS assume was delivered?

The overlapping data problem may seem to be contrived, and it is rare in the wild, but programs such as *fragrouter* [Song *et al.*, 1999] intentionally modify TCP/IP streams to confuse eavesdroppers. *Fragrouter* takes scripts written in a little language that define the kinds of pathologies desired on the packet stream. Outgoing packet streams can be distorted so badly that the monitoring host may be incapable of decoding the data stream.

Four places need to process pathological TCP/IP packet streams correctly: clients, servers, firewalls, and NIDSs. [Handley *et al.*, 2001] propose an intervening device to normalize the packet stream. One can imagine adding such functionality to a firewall, making it behave more like a circuit-level gateway. Some firewalls already do some of this; they reassemble fragmented packets to protect against short fragment attacks. True circuit-level gateways (see Section 9.3) cleanse IP streams as well.

This sort of issue was the basis of our recommendation in the first edition that corporations avoid direct IP connectivity between their corporate networks and the Internet, and use application- or circuit-level firewalls instead.

These odd, contrived packets are rare in normal Internet traffic. IDS software should notice when an unusual number of packets are fragmented, contain small TTL values, or have other unusual pathologies. Unfortunately, these do occur in legitimate traffic, and can't be used as the sole indicia of malicious activity. To give just one example, *traceroute*—a very normal network diagnostic program—is a leading cause of small TTLs.

False negatives are an obvious problem: An attack occurred and we missed it. False positives are a particular problem, because they are very hard to avoid without disabling the desired features of the IDS. People are expensive. People competent enough to do a good job of monitoring these alarms quickly tire of them, and step quickly through the reports or ignore them entirely. False alarms can also come from configuration errors.

When evaluating an IDS system, always check out the false-positive rates.

15.1 Where to Monitor

It is important to understand the limitations of IDSs before you consider installing one. The most important question to ask is "What is the purpose of the IDS?" One legitimate reason to install an IDS outside of your firewall is to justify funding requests to your boss (this is a threat model in which management is the enemy). There is no point in monitoring the outside of your network to see if you are under attack—you are. That's not to say that you should ignore the outside, but it is probably more valuable to record and store the outside traffic for later examination than to attempt real-time intrusion detection. If you are a researcher trying to learn about new attacks, such information is invaluable. However, there is too much traffic going by, and an IDS is too weak a tool to do real-time analysis. It is a fine place to train people who are learning about networks and IDS devices.

IDS devices become more useful when deployed near important assets, inside the various

security layers. They are like a video camera installed in a bank vault, a final layer of assurance that all is well. The more restricted the normal access to a network or a machine is, the more sensitive the rules should be for the detectors. People probably shouldn't be issuing Web queries from the payroll computer.

15.2 Types of IDSs

Different kinds of IDSs have different strengths and weaknesses. Signature-based IDSs have a database of known attacks; anything matching a database entry is flagged. You don't get many false positives if the system is properly tuned, but you are likely to experience false negatives because they only recognize what is in the database. Unfortunately, the sweet spot between overly broad signatures (which match normal traffic) and overly narrow signatures (which are easy to code around) is hard to find. At the very least, signature-based systems should incorporate context, and not just rely on string matches.

Anomaly-based IDSs, which look for *unusual* behavior, are likely to get false positives and false negatives. They work best in an environment with a narrowly defined version of *normal*, where it is easy to determine when something is not supposed to occur. The more special purpose a machine is, the more constrained normal behavior is, and the less prone it is to false positives.

Anomaly detection is an interesting area of research, but so far has yielded little in the way of practical tools. [Forrest *et al.*, 1996] and [Ko *et al.*, 2000] have produced some interesting results. Forrest developed a tool that monitors processes running on a computer and examines the system calls. The tool has a notion of what a normal pattern of calls is, and recognizes when something happens that is not supposed to. The tool uses n-grams of system calls and slides a window across the sequence of calls that a process has executed. If the behavior of a process varies beyond a certain threshold from known trace behavior, an anomaly is signaled. The key feature that the tool looks for is the order of the calls in a sequence. Certain calls are preceded by others, and if enough calls are preceded by the wrong calls, it assumes that there's trouble. [Somayaji and Forrest, 2000] describe how to slow down or abort processes that behave too badly.

As noted, IDSs can be host-based or network-based. The two are complementary, not mutually exclusive; each has its strengths and weaknesses. Host-based systems tend to know the state of their own machine, which simplifies the processing of the data flows, but the software can be subverted if the host is compromised. Network-based devices are stand-alone units and are presumably more resistant to attack or even detection. On several occasions we have advised IDS designers to cut the transmit lead on their Ethernet cables, or at least suppress the emission of packets in the software. That's hard to do with today's twisted-pair Ethernets on some platforms; however, there are dedicated hardware devices designed to tap networks without any possibility of transmitting onto them.

For some environments, such as DMZs, our favorite kind of IDS is a *honeypot*—a machine that nobody is supposed to touch. Any source of traffic to that machine is at the very least misbehaving, and more likely evil. A honeypot might not work in an open corporate environment, but is well suited to a dedicated network, which should not have anything except dedicated machines.

A honeypot on the public Internet can be useful for studying hacker behavior, though some

hackers have learned to avoid them. One of the prettiest examples is Niels Provos' *honeyd* [Spitzner, 2002, Chapter 8]. It mimics an entire network, populated by many different sorts of machines. However, you can't rely on this for determining if someone has penetrated a single machine; at most, it can detect scans. To cope with the false positives and false negatives, some people use multiple IDSs whose outputs are correlated. Time correlation can be used to detect "low and slow" attacks.

15.3 Administering an IDS

An intrusion detection system requires a significant amount of resources. IDSs have to be installed in strategic locations, configured properly, and monitored. In most environments, they will have to deal with an amazing amount of broken network traffic. For example, an HP printer driver we used tried to find everything on the subnet without knowing about masks, so it scanned an entire /16 network looking for an HP printer. Network management software sometimes does the same. Someone running an IDS has to be able to deal with this kind of traffic and must be tolerant of a lot of noise [Bellovin, 1993]. They also have to make sure they do not become too complacent because IDSs tend to cry wolf.

15.4 IDS Tools

Many IDS tools are available, both free and commercial. Sniffers, such as *snort* (see the following section), *ethereal*, and *bro* [Paxson, 1998], are very useful. *Ethereal* provides a nice GUI that enables you to reproduce TCP streams so that you can view application-level data. It can also dump network traffic to a file for later investigation.

Commercial products range from pure snake oil to fairly useful tools. Some products try to apply AI techniques to the problem. Others collect distributed information and try to assemble an overall view of an attack.

15.4.1 Snort

Perhaps the most popular free intrusion detection program is *snort*, developed by Martin Roesch. *Snort* is open source, and there is an active community of users and contributors; see `http://www.snort.org/`. The program is available on a wide variety of platforms—it works anywhere that *libpcap* runs.

Snort can be used in several ways. It can sniff a network and produce *tcpdump*-formatted output. It can also be used to log packets so that data mining tools and third-party programs can do after-the-fact analysis on network traffic. The most interesting feature of *snort* is its ability to design a ruleset that recognizes certain traffic patterns. Many rules are available for *snort*, and they are often shared among users and posted on the Internet. *Snort* can be configured to recognize *nmap* probes, known buffer overflow attacks, known CGI exploits, reconnaissance traffic, such as attempts to fingerprint the operating system based on characteristics of the network stack, and many other kinds of attack for which an administrator wants to configure a rule.

Here is a sample rule taken from [Roesch, 1999]:

```
activate tcp !$HOME_NET any -> $HOME_NET 143 (flags: PA;
        content: "|E8C0FFFFFF|bin|; activates: 1;
        msg: "( buffer overflow!";)
   dynamic tcp !$HOME_NET any -> $HOME_NET 143
        (activated_by: 1; count: 50;)
```

The preceding rule specifies that an alert should be sent when an IMAP buffer overflow is detected. At that point, the next 50 incoming packets headed for port 143 should be logged. Some of these packets probably contain revealing information about the attack that might be interesting to a network analyst or administrator.

Note, though, that there's a flaw here: The "PA" flag specification means that both the PUSH and ACK bits must be set on the packet for it to be matched by this rule. It's pretty trivial for an attacker to evade it by ensuring that PUSH isn't set.

As you would expect from all useful intrusion detection tools, *snort* provides flexible alerting mechanisms, ranging from a pop-up window on the screen to e-mail and pager notifications. There are *snort* user groups that get together and compare data dumps, share rulesets, ponder false positives, and discuss possible enhancements to the program. There is also on online forum with plenty of useful information at `http://snort.rapidnet.com/`.

And yes, there is the usual arms race between attackers and the *snort* script writers.