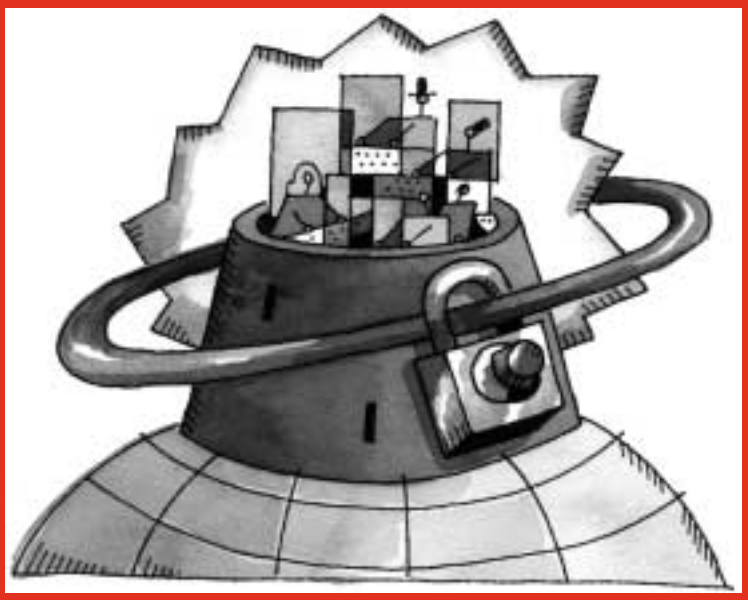
;login:
THE MAGAZINE OF USENIX & SAGE
December 2002 • volume 27 • number 6

# Focus Issue: Security

Guest Editor: Rik Farrow

## inside:





USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

# automated log processing

**by Jared Allison**

Jared Allison is an Internet security engineer at UUNet. He has spent much of his time working on projects that automate traffic analysis, intrusion detection, and router and system security configuration checking in large-scale networks. Most of his work is done with C, Perl, and PHP.

*jallison@UU.NET*



## Hindsight Is 20/20

It happens to everyone – one of your systems is compromised and you go back through the logs to see what happened. How did the intruder break in? Then you see it:

```
Jun 14 13:08:58 computer.company.com sshd[9779]: log: Connection from
452.312.34.58 port 3552
Jun 14 13:09:01 computer.company.com sshd[9779]: fatal: Local: crc32
compensation attack: network attack detected
```

(Names and numbers have been changed to protect the innocent.)

As you trace it through the logs, you can see the whole sequence of events. But why didn't you notice this *before* it happened? Or at least quickly afterward? When going through log files to determine what happened and what needs to be fixed, it is often easy to see entries that indicated an attempted intrusion. If only you'd seen those logs beforehand! Of course, nobody has the time to constantly watch their logs for scans, dictionary attacks, misconfigured equipment, and so on. Even given the time, it would be very difficult to pick these events out of the noise. To have a chance at keeping up, monitoring tasks must be automated with scripts that can filter out the noise and report only information that is relevant and useful. Though there is some time delay inherent in log processing, logs can still be useful in proactive security. Let's look at some straightforward examples of how to use logs to help prevent intrusions and increase the visibility of events on your network.

## Log Files and Low-Hanging Fruit

Simple port scans are one of the most common and easy-to-detect indicators of wrongdoing. This is especially true of worms and IRC "botnets" that don't attempt to be stealthy about their scanning. Using an intrusion detection system to watch for attempted break-ins can be very effective and has been treated in many books. When an IDS isn't an option, however, you can still detect a lot of the simple intrusion attempts through log analysis. Firewall logs from access lists, for example, can provide enough information to detect port scans (protocol, source and destination IPs and ports, and times). The question is, what to do with that information? How do you pick out the scans from the noise? Once you pick out the scans, then what do you do?

Before you can take any action, you have to find the scan activity. The first step toward doing that is to parse your logs into a usable format. Perl is great for this. You are going to need to be able to search through and group the data a number of different ways. Perl alone will work for this as well, but an easier and more flexible solution is to use a database. Writing your Perl parsing script to dump the interesting part of the logs in, say, CSV format and loading the whole file into the database at once is much faster than interfacing directly to the database and loading one row at a time. These scripts are very simple and generally look something like this:

```
while( defined($line=<LOGFILE>) ) {
    if( $line =~    /(\d+)-(\d+)\s+                  # date
                     (\d+:\d+:\d+)\s+:               # time
                     (\S+)\s+                        # protocol
                     (\d+\.\d+\.\d+\.\d+):(\d+)\s+    # source IP and port
                     ->\s+
                     (\d+\.\d+\.\d+\.\d+):(\d+)      # destination IP and port
                    /x ) {
```

```
        print TEMPFILE ( "$1-$2 $3", (getprotobyname($4))[2], ",",
                unpack( "N", inet_aton($5) ), ",$6,",
                unpack( "N", inet_aton($7) ) ",$8, "\n" );
        }
}
```

The particular regular expression will vary depending on the format of the log files. Finally, you should define a generalized table layout so that you can import logs from many different types of devices (e.g., routers, firewalls, or even tcpdump) into the same database table. At a minimum, your database table should have the following fields:

| | |
|---|---|
| Data Source | (This field could be text – perhaps an enumerated type.) |
| Date & Time | (Store these in the same field if possible.) |
| Protocol | (Use /etc/protocols to convert this to an 8-bit unsigned integer.) |
| Source IP Address | (Store as a 32-bit unsigned integer.) |
| Source Port | (Store as a 16-bit unsigned integer.) |
| Destination IP Address | (Store as a 32-bit unsigned integer.) |
| Destination Port | (Store as a 16-bit unsigned integer.) |
| Action | (Was this packet permitted or denied?) |

A quick note on ICMP – it is useful to store the ICMP type and code information. An easy way to do this without wasting space is to store them in the source port and destination port fields, respectively. Just remember to check the protocol in your scripts, or you might spend a lot of time trying to track down why anyone would use those weird low ports.

Once your data is loaded into a database, a fairly easy way to look for obvious scans is to group the data as follows. Grouping by the source and destination IP address and then looking for large numbers of different destination ports will reveal potential host scans, since a conversation between two hosts is usually confined to, at worst, a small number of ports. Likewise, grouping by the source IP and destination port and then looking for large numbers of different destination addresses reveals network scans. Expressed in SQL language, a search for network scans might look like:

```
SELECT source_ip_address, destination_port
COUNT(DISTINCT(destination_ip_address))
AS DST_IP
FROM my_log_table
GROUP BY source_ip_address, destination_port
HAVING DST_IP > 10
```

Examine the output with some Perl and you've got a list of potential scans. You may want to add in some code to adjust your sensitivity to different ports and address ranges. For example, some UDP/53 packets bouncing off of a firewall is probably not as important as TCP/22 packets, so you might want to require 50 of those packets before you get upset instead of 10. Then again, maybe it is more important; you should adjust the sensitivity in a manner appropriate to your situation.

There are several options for what to do with the list of IP addresses that appear to have scanned your equipment. One fairly extreme option would be to block all packets from that IP address to any of your systems. This would help keep intruders out and could be automated, but that could lead to a self-inflicted denial of service and so is

not a good idea. In deciding what to do, it is important to bear in mind that usually the IP address that the scan came from is not actually the IP address of the computer the scanner was sitting at. It is often another machine that has been compromised. In light of that, a good solution is to send a polite but firm email to the system administrator detailing what happened and to include some logs. This will usually solve the problem and can be safely scripted using a Perl WHOIS module to find administrator contacts for IP addresses (*whois.abuse.net* works well if you have a domain name). It is wise to design the script so that it errs on the side of caution when deciding whether to send mail. Crying wolf will not help your cause. If the recipient of the mail fixes the system, it will cut off one potential avenue of attack and have the added benefit of raising the bar for security in general. Even if nothing comes of the email, by tracking scan activity you will gain insight into what services may be exploitable.

The same general process can be applied to a host's authentication logs. Making effective use of the logs is simply a matter of defining a pattern of activity that appears unusual. For example, the use of `su` to elevate privileges may not be uncommon, but what about more than one failed attempt to become root without success? A few attempts followed closely by success might indicate a forgotten or mistyped password. However, without a successful attempt closely following, it starts to sound more suspicious. It is straightforward to write a Perl script that parses syslogs and checks for successful `su` attempts within, say, 15 seconds after an unsuccessful attempt. The script could then mail the administrator a list of suspicious usernames and times of attempted accesses.

## Catching Misconfigurations

A major threat to the security of any system is misconfiguration – whether accidental or intentional. An administrator may configure a system in a way that puts it in an insecure state. Good logging can help detect this condition, hopefully before it becomes a problem. It can also help educate users and administrators by pointing out which commands can create security holes. The following example outlines how to detect someone misconfiguring a router.

Many routers support command-level accounting via the TACACS+ protocol. If command accounting is enabled, the TACACS+ accounting server will receive logs of every command entered (for a certain privilege level), what time it was entered, and which user entered it. These command logs can be parsed by a Perl script and, optionally, loaded into a database. Whether from the database or as it goes through the logs, the Perl script can then watch for commands such as `no access-list` or `no service password-encryption` that could create security problems. An alert could be paged or emailed to an administrator should such a command be entered. In a network with a small number of administrators, you could even make the script mail directly to the person who made the change (since you aren't all sharing the same passwords, right?).

## Future Work

Simple scripts and databases are great for catching the easy stuff, but what about more stealthy attacks? Expanding the time span of your search will help catch some things, but this requires more and more processing power and storage space. One potential solution is to combine data from several different sources into one system. With data from application syslogs, firewall logs, and command accounting logs, for example, you could assign probabilities or levels of alert to different events from each. This would give a script a common ground for comparing events from different sources of

data. If three events, each with a low probability of occurrence, happen on the same system, there is a greater cause for alarm than if any occur individually. In this way, several events that by themselves would not be significant enough to cause alarm can be put together into a more complete picture. For example, a few TCP/22 packets denied to a few hosts on a subnet is perhaps nothing to be concerned about, but if you also notice a few unusual syslog messages from sshd on a machine on that subnet, and a few files that root owns get changed, then it would be a good idea to look at that machine. The more disparate sources of data are included, the more difficult it will be for attackers to slip through unnoticed.