USENIX Association

Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

Log Monitors in BSD UNIX

Brett Glass, *Glassware*
P.O. Box 1693
Laramie, WY 82073-1693
http://www.brettglass.com/mailbrett.html
BSDCon 2002 slides at http://www.brettglass.com/logmonitors/

Abstract

A log monitor is a process, or daemon, which monitors log messages produced by a computer system and the programs which run on it. A properly designed log monitor can recognize unusual activity (or inactivity), alert an administrator to problems, gather statistics about system activity, and/or take automatic action to contain a threat. It can even "learn," over time, what is normal and identify message traffic that may betray an abnormal situation. Languages which facilitate string processing and pattern matching, such as Perl and SNOBOL4, are good choices for the implementation of log monitors for this reason. Included in this paper is source code for a log monitor that identifies and blocks attacks from Code Red, sadmind/IIS, Nimda, and similar worms. Policy issues -- including the usefulness of "amnesty" to prevent inadvertent blocking of innocent third parties -- are discussed. The work described in this paper --which is still progressing at this writing -- will hopefully culminate in the release of a general purpose log monitoring facility.

[Note: Because the camera-ready copy of this paper was submitted two months prior to the BSDCon 2002 conference to allow time for printing, and was required to conform to strict size limits, the version which appears online may contain more detail as well as new information gleaned from ongoing work. For the latest version, and/or to follow up the many references via HTML links, access the master copy at http://www.brettglass.com/logmonitors/paper.html on the World Wide Web.]

1 Introduction

System administrators' time is valuable. Few, if any, can afford to spend many hours each day poring over the voluminous log files generated by systems and network applications. Yet, if an administrator fails to recognize quickly and respond to anomalous events chronicled in log messages, systems or entire networks can be abused, "hijacked" for use in illicit activities, and/or removed from service by malicious parties. Log monitors aid the administrator by automatically filtering and digesting the flood of log information -- and, in some cases, responding on his or her behalf.

2 What is a Log Monitor?

A *log monitor* is an agent which responds automatically to conditions revealed by one or more system log messages. The response may consist of autonomous action to handle a situation and/or the notification of a human administrator.

A *stateful log monitor* is a log monitor that infers the presence of a condition requiring attention by compiling data from more than one log message. It may simply note the number and/or frequency of log messages related to a particular type of activity or may generate more sophisticated cumulative statistics from those messages.

2.1 Capabilities of Log Monitors

What can a log monitor do? Among other things, it can:

- Detect abnormal usage patterns
- Recognize system or network abuse (e.g. spamming and mail bombing)

- Catch worms and other malware
- Detect vulnerability scans (e.g. port scans)
- Detect intruders (or attempted intrusions)
- Detect resource shortages (e.g. slow response times, out-of-memory conditions, out-of-disk conditions, inadequate swap space)
- Detect imminent or actual system failures
- Compile statistics in real time (including running averages, etc.)
- React to conditions by notifying an administrator and/or taking immediate action

While it is useful for a log monitor to be able to recognize failures or threats for which it has been given a human-crafted "signature," it can also incorporate more subtle heuristics. By accumulating statistics about what constitutes "normal" activity, log monitors may be able to recognize anomalous behaviors that a human system administrator might at first overlook, such as the cessation of events which normally happen with a given frequency.

2.2 Examples of Existing Log Monitors

`swatch` ("Simple Watchdog") [1] , developed by Stephen Hansen and Todd Atkins of Stanford University, and `2swatch` [2], an enhanced version of `swatch` developed by the Pacific Institute for Computer Security (PICS) research group at the San Diego Supercomputer Center, are two very simple general-purpose log monitors. `swatch` accepts a configuration file whose entries consist of patterns and lists of actions. (A sample pattern/action block is shown in Listing 1.)

```
watchfor /ANONYMOUS FTP LOGIN REFUSED FROM (\S*)/
 mail addresses=admin,subject=Attempted anonymous FTP from $1
 exec blackhole $1
```

Listing 1: Sample pattern/action block for swatch

It then "tails" a system log file looking for the regular expression specified in the "watchfor" line of the block. If the expression is matched, swatch takes the series of actions that follow. Possible actions include producing a beep, writing a message in a specified color to the console, sending an e-mail message including the log entry that was matched, or executing an arbitrary command. (In the example above, the $1 causes the text matched by the subexpression within the first set of parentheses to be interpolated.) `swatch` incorporates features to allow throttling and to recognize patterns only during certain hours of the day and/or days of the week.

`2swatch` extends `swatch` by offering the possibility of deferred as well as immediate action. `2swatch` can accumulate a series of messages matching a pattern into a report that is e-mailed as a single message. This prevents mail systems from being flooded with e-mail messages that each contain a single log entry. Unfortunately, neither `swatch` nor `2swatch` can perform stateful monitoring beyond their respective throttling and accumulation functions, though it is of course possible to implement statefulness via cleverly designed external programs or scripts. Nor can these two programs, by themselves, correlate entries from more than one log. A final drawback of these two programs is that they have been released under licensing terms (the GPL and a unique "no commercial use" license, respectively) which hamper or prohibit their reuse in commercial products.

The primary objective of the work described in this paper is to develop a generalized and portable framework which eases the creation of customized log monitors and overcomes the limitations described above. It is intended that the results of this work, which is initially being performed on FreeBSD, be released under "truly free" (i.e. MIT- or BSD-style) licensing terms so as to permit adaptation to other operating systems and commercial as well as non-commercial reuse.

2.3 Log Monitors vs. Log Analyzers

A *log analyzer* differs from a log monitor in that it does not operate in real time (or nearly real time) but is run against system logs after the fact. Many such utilities exist, especially for Web and mail servers. At this writing, log analyzers are more common than log monitors. Most implementations of BSD UNIX come with primitive but helpful log analysis scripts that report significant log events to an administrator via e-mail. These scripts are often found at `/etc/daily`, `/etc/weekly`, and `/etc/monthly` and are usually run at appropriate intervals by the `cron(8)` [3] daemon. (In recent versions of FreeBSD, the default `/etc/crontab` file instead activates Trainer and Somers' `periodic(8)` [4] utility, which in turn runs these scripts from the `/etc/periodic` directory.) Apache's `logresolve` program performs a simple analysis of Apache log files that consists primarily of efficient reverse domain name resolution. Tom Boutell's `Wusage` [5] web log analysis utility, released as Shareware, is a much more sophisticated log analyzer for Apache; it can generate daily, weekly, monthly, and annual reports.

Some log analyzers are designed to "wake up" periodically and scan the latest entries in a system log, acting, if appropriate, upon what they "see." (Kai's Spamshield [6] , which detects incoming spam and blocks the sender via a "blackhole" route, is an example.) If the interval between scans is sufficiently short, such log analyzers can perform some of the functions of a log monitor, detecting and handling conditions which require timely but not instantaneous attention.

3 Log Monitors and BSD syslogd

In BSD, the kernel and most daemons traditionally log their activities via `syslogd`, the system logging daemon, originally written by Eric Allman. Messages from the kernel are received via the pseudo-device `/dev/klog`, while messages from applications are received via the UNIX domain socket `/var/run/log` or via UDP socket 514. In the early, more trusting days of the Internet, `syslogd` listened on this socket and by default accepted any message that came in. But in most modern UNIX implementations it does not do this, because -- as stated in many versions of the `syslogd` manual pages -- it was "equivalent to an unauthenticated remote disk-filling service." [7]

3.1 syslogd Facilities, Priorities, and Tags

The standard Berkeley `syslogd(8)` [8] is configured via the file `/etc/syslog.conf`, which specifies the disposition of log messages. In traditional Berkeley UNIX, the messages are sorted according to a *facility* code and a *priority* or *severity level*. These parameters are used to route each log message to one or more destinations, including the system console, the terminals of specific users, other machines, files, and/or programs. Modern UNIX and UNIX-like systems have many more facilities than were contemplated in the original UNIX logging scheme. Nonetheless, most implementations have hewn to the traditional list of facilities defined in 4.4BSD for the sake of compatibility and tradition. A few (such as FreeBSD) have added additional facility codes, including **`console`**, **`ntp`**, and **`security`**, and DEC ULTRIX uses facility number 10 (**`authpriv`** in many versions of BSD) to log events for its AdvFS filesystem. [9] Unfortunately, these additions and conflicts may hinder software portability. What's more, the facilities which were added frequently do not reflect a consistent design philosophy. It is unclear, for example, why the authors of FreeBSD's `syslogd` implementation felt that NTP was deserving of its own facility code while DNS, DHCP, PPP, and HTTP (or, for that matter, any of the many other protocols listed in `/etc/services`) were not.

Further confusing the issue of how to classify log messages is the fact that many recent versions of `syslogd` have added the ability to sort messages via strings called "tags," which are transmitted to the logging daemon along with each message. Tags usually (but not always) contain the name of the program that generated the message. (At this writing, the ability to sort messages by tag is available in OpenBSD and FreeBSD but not in NetBSD.) To facilitate logging across networks, some implementations of `syslogd` add yet

another sorting criterion: they allow messages to be dispatched according to the name of the host from which they originated.

As if all of this weren't enough, there's yet another fly in the ointment. While Berkeley `syslogd` itself is able to sort messages by facility, severity, tag, and originating host, it does not record the facility and severity level in each log message, and in most implementations there is no way to make it do so. This makes it difficult for a log monitor (and, in some cases, for humans) to use these same criteria to sort messages that syslogd has aggregated into a single log file. The version of `syslogd` found in recent versions of FreeBSD is one of the few that provides a solution to this problem. If the daemon is started with the -v ("verbose") command line option, the facility and priority level are logged as numbers. Specifying -v twice causes them to be shown by name between angle brackets (e.g. `<auth.notice>`). The latter choice, while it consumes a bit more disk space, makes the logs much easier both to read and to monitor. It would be a relatively simple matter to add similar capabilities to other `syslogd` implementations.

3.2 Monitoring Techniques for Use With syslogd

A log monitor can monitor the output of `syslogd` in one of three ways. The most common technique is to "tail" the log, either by accepting the output of the "tail -f" as input or by incorporating a module that provides equivalent functionality (e.g. Matija Grabnar's File::Tail [10] for Perl). The downsides of this technique are twofold. First, the application (or the `tail(1)` utility) must "poll" the file at regular intervals to see if it has grown. This consumes resources even if there is nothing to be done. Secondly, if the log file is "rotated" (most often done by renaming it and creating a new file for future input), the application may be left "watching" the old file (which is no longer growing) and miss all subsequent messages. (This is a problem of some, but not all, "tail" implementations.)

Another technique is to open the log file every so often, collect the last *n* lines, and then close the file. This technique avoids problems when logs are rotated, but may cause the log monitor to miss important messages if the log has grown more quickly than expected (as can happen during an unusual situation). Repeated opening and closing of the file also creates substantial overhead.

The best technique, when it is possible to use it, is to instruct `syslogd` to pipe messages directly to a log monitor process. This option is not available in all implementations of `syslogd`; however, it is present in FreeBSD's syslogd and BalaBit's `syslog-ng` [11] and can be easily added to other logging daemons. Some versions of syslogd, including the Berkeley-derived Linux `syslogd` and Core-SDI's modular syslog (`msyslog`) [12], cannot pipe directly to an arbitrary program but can send output to a named pipe where an application is listening.

FreeBSD's `syslogd` makes piping to applications especially easy by handling nearly all of the logistics for the programmer. Because `syslogd` handles the logistics of distributing each log message to the required destinations (including the log monitor), the log monitor process is unaffected by log file rotation. It merely has to be prepared to save its state and exit if `syslogd` restarts (see below).

One feature of `syslogd` which may actually defeat the purpose of a log monitor is automatic output compression. When `syslogd` sees two or more identical messages bound for the same destination, it outputs the first and then counts (but does not output) the duplicates. After a predetermined delay, it outputs a message of the form "Last message repeated *n* times" indicating the number of copies received during the delay period. If still more copies of the same message arrive, the process is repeated with a longer delay between reports. Ironically, this feature -- which cannot be turned off in any version of `syslogd` known to the author -- is just the opposite of what is needed for effective log monitoring. (It is precisely when an unexpected flood of repeated messages arrives that it is most useful for a log monitor to take prompt, autonomous action. But if notice of those messages are delayed, it cannot do so.) The author has submitted a patch to the

FreeBSD Project which disables repeat counting on messages piped to a program. (With the patch in place, if the same messages also go to a file or terminal, compression will still occur on those outputs.) It may also be desirable to disable compression when logging to a remote host, since -- while this would cause an increase in network traffic -- it would facilitate the implementation of remote log monitors. Similar modifications should be made to other logging daemons that direct output either to programs or to named pipes, to facilitate the use of log monitors.

3.3 Processing Piped Output from FreeBSD's syslogd: Caveats and Tricks

While implementing his first group of trial log monitors under FreeBSD, the author learned by experience how to deal with the quirks of this particular logging daemon and operating environment. FreeBSD's `syslogd` does not start a program to which messages are "piped" until it has output for it. When it does start such a program, it executes it via `sh(1)` so that command line processing may performed. To avoid the overhead of a vestigial shell process, it is best to use the "exec" command (as shown above) to launch the log monitor script or program. The log monitor should expect to receive messages via standard input; its standard output and standard error file handles will be redirected to `/dev/null`.

Secure programming practices are of the utmost importance when one is creating log monitors. Piped applications started by FreeBSD's `syslogd` run with the same uid as `syslogd` itself -- normally root. Because a key function of log monitors is to take adminstrative action as a result of what they observe in a stream of log messages, they often must run as the superuser and may not be able to accomplish their intended functions if they "drop" privileges for safety. (In a capabilities-based system, it may be possible for a log monitor to drop capabilities that the author knows it will never use.) It is therefore especially important to avoid potential buffer overflows, format string vulnerabilities, and security holes that might arise when unfiltered input is passed directly to system APIs or programs. "Tainting" and/or extremely careful validation of input is strongly recommended.

Should an application which receives piped output from `syslogd` terminate of its own accord, it will be restarted when there is more input for it. However, syslogd may itself need to request that a log monitor terminate. (The most common situation in which this will occur is when `syslogd` receives a "hangup" signal -- SIGHUP -- indicating that it must restart and reread its configuration file.) `syslogd` indicates its desire to shut down the log monitor by closing the pipe it has created to the log monitor's standard input. The log monitor then has a predetermined amount of time -- 60 seconds in most implementations -- to save any state it wishes to preserve and terminate. If it takes longer, syslogd will attempt to kill it by sending it a "terminate" signal -- SIGTERM. To ensure that `syslogd` is able to kill a log monitor that is frozen, it is advisable for a log monitor *not* to catch SIGTERM unless it may need a very long time to save its state.

4 Log Monitors and Apache

The Apache Web server [13] is perhaps most widely deployed application for UNIX-like operating systems which does not normally log via `syslogd`. (Apache can be easily configured to log errors via `syslogd`, but there is no built-in option that allows the logging of all traffic this way.) Fortunately, Apache's own logging facilities are more powerful and flexible than those of `syslogd` itself, so logging output via `syslogd` is rarely necessary.

Apache can pipe log messages to external programs on all UNIX and UNIX-like platforms, making the implementation of log monitors, log rotators, and other such utilities very straightforward even on systems whose `syslogd` implementation does not support piped commands.

4.1 Creating Primitive Log Monitors Within Apache

Apache's logging modules also provide conditional output, pattern matching, and custom formatting. These features not only facilitate the use of

external log monitors but in some cases allow primitive log monitors to be implemented directly within Apache itself. The fragment in Listing 2, when added to the `httpd.conf` server configuration file (either in the main body or within the <VirtualHost> directives), will automatically block traffic from worms such as Nimda [14] , Code Red [15] , and sadmind/IIS [16] .

```
# Flag requests for URIs containing common strings from Nimda-like worms
# (including Code Red, sadmind/IIS). Note that the patterns below are regexes;
# remember to escape dots and other characters with special significance!
SetEnvIf Request_URI "/winnt/system32/cmd\.exe" worm
SetEnvIf Request_URI "/scripts/root\.exe" worm
SetEnvIf Request_URI "/MSADC/root\.exe" worm
# Don't use the following patterns if you use "upreferences" in URIs
SetEnvIf Request_URI "/\.\." worm
SetEnvIf Request_URI "\.\./" worm

# Block attackers who send the patterns above within URIs. The command below
# uses a blackhole route. It's more efficient to firewall (the command
# will vary depending upon the firewall in use) or to use SSH to add rules to
# an upstream firewall to block the attacker, but this method has the
# advantage that it is relatively independent of configuration. If several
# commands must be executed, or if postprocessing of output is desired, it
# is best to invoke a script or compiled program rather than doing all the
# work from within httpd.conf.
 CustomLog "|exec sh" "route -nq add -host %a 127.0.0.1 -blackhole" env=worm

# Note that no input from the client is used in the shell command, so this
# set of directives is not subject to exploits via crafted strings. If strings
# from the client were used, stronger input validation would be in order.
```

Listing 2: Worm blocker implemented entirely within Apache's httpd.conf

While BSD and Apache cannot be "infected" by the worms targeted by the directives in Listing 2, a worm can nonetheless tax a Web server by consuming processes in the Apache process pool, glutting system logs with error messages, and infecting susceptible machines elsewhere on the network.

The configuration shown in Listing 2 uses Apache's SetEnvIf [17] directive (implemented by the module mod_setenvif) to perform regular expression matching on incoming URIs. It then uses the CustomLog [18] directive (implemented in mod_log_config) to do conditional logging based on the results of the match. When a worm is detected, Apache pipes a specially formatted "log message" -- actually a command -- to a shell for execution. (The "exec sh" may at first glance seem redundant. However, it is necessary to restart the shell -- which is initially invoked so as to accept a command as an argument -- so that it will accept commands via standard input instead.) The command creates a "blackhole" route on the host machine and locks out the attacker. If the Web server does double duty as the gateway between the Internet and an office LAN (as is often the case in small office/home office networks), blackholing the attacker will also protect the machines that sit "behind" the server. If there is a firewall upstream of the Web server, it may be desirable to replace the command that creates a blackhole route with one that causes the firewall to block the attacker.

While the author had great success with this simple log monitor (which he crafted during the early morning hours of 18 September, 2001 when Nimda began to spread), it clearly has many deficiencies. For example, it does not check to see whether an attack is coming from the Web server's own address (which can easily happen if the machine is doing double duty as a NAT router or dial-up server.) to prevent it from blackholing itself! Nonetheless, this example is a valuable proof of concept. It demonstrates that the ability to apply a general pattern matching facility to log messages, and then execute commands based on those messages, are sufficient to allow the creation of a useful, if not perfect, log monitor. More sophisticated tools -- such as languages with built-in pattern matching -- make it even easier to

write quite sophisticated agents.

4.2 Monitoring Techniques for Use With Apache

To construct effective log monitors for use with Apache, one must understand the conventions it uses when piping log output to applications. Like `syslogd`, Apache expects piped applications to be trustworthy. It runs them with the permissions accorded to Apache's master process, which usually runs as the superuser. (Apache, in its recommended configuration, maintains a single privileged "master" process which forks a pool of unprivileged processes to handle incoming requests.) As mentioned earlier, a log monitor often requires this high privilege level if it is to take autonomous administrative action. Thus, good programming practices are of paramount importance.

Unlike `syslogd`, Apache starts piped applications as soon as it has finished reading its configuration file. This gives them time to start up before receiving the first message, improving response time at the expense of overhead. If a piped application terminates, Apache restarts it the next time a message is to be delivered to it. Like `syslogd`, Apache normally uses `sh(1)` to parse command lines and set up file redirection for piped commands.

It is important to remember, when writing log monitors for Apache, that users' log formats may vary. It is therefore best to use a custom log format that the log monitor expects -- or, alternatively, to monitor the error log, whose format cannot be customized and is therefore *almost* fixed. (One aspect of the error log format can be changed via configuration: the way hosts are identified. If HostNameLookups [19] is on, domain names are output instead of numerical IP addresses.) Note that Apache allows error log messages to be sent only go to one destination. (If there is more than one ErrorLog directive, each overrides the previous ones rather than supplying an additional destination.) Fortunately, Apache also records errors in the access logs, so using ErrorLog to feed messages to a log monitor still allows a human to review messages denoting errors.

4.3 An Extensible Worm Blocker/IDS for Apache

After implementing the "quick and dirty" Apache worm blocker described above, the author decided to create a more powerful, extensible, and maintainable one. SNOBOL4 [20] [21] was chosen because of this language's prodigious pattern matching, text handling, and parsing abilities.

The 28 executable lines of SNOBOL4 in Listing 3 detect infection attempts from worms such as Code Red, Nimda , and sadmind/IIS and "blackhole" the attacking machine. Unlike the earlier example, however, this SNOBOL program completely parses, and validates the fields of, each Apache ErrorLog message before taking action. This eliminates any chance of a "false positive," which might occur if a regular expression in the earlier example happens to match part of a legitimate request.

```
* An Extensible worm blocker/IDS for Apache in SNOBOL4
* Copyright (c) 2001 by Brett Glass
* Released under the "MIT" license; see http://www.opensource.org/licenses/mit-license.html
*
* This program accepts the piped error output from the
* Apache Web server and spots lines indicating an attack
* from Nimda.A or similar worms, including Code Red,
* sadmind/IIS, and Nimda.E. It can then firewall or
* blackhole the attacking host. Add it to your Apache
* configuration by inserting a line such as
*
* ErrorLog "|exec snobol4 -b /usr/local/bin/wormblock.sno"
*
* Also, make sure that HostNameLookups is off so that the
* log messages contain numeric IP addresses.
*
* This program is designed to be easily extensible to catch
* a wide variety of potential exploits.
*
```

```
* An Apache error log message generated by the Nimda worm
* might look like this (wrapped for readability):
*
* [Thu Nov  1 12:46:07 2001] [error] [client 12.98.224.154]
*  File does not exist: /usr/local/www/data/textorics/scripts/
* ..%5c../winnt/system32/cmd.exe
*
* Build up SNOBOL patterns for Apache ErrorLog messages.
* Use the pattern "WS" for whitespace (tabs or blanks)
        WS = SPAN(' ' CHAR(9))
        DIGITS = '0123456789'
        WEEKDAY = 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun'
        MONTH = 'Jan' | 'Feb' | 'Mar' | 'Apr' | 'May' | 'Jun' |
+        'Jul' | 'Aug' | 'Sep' | 'Oct' | 'Nov' | 'Dec'
        DAYOFMONTH = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,31) *GE(NUMBER,1)
        HOUR = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,23)
        MINUTE = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,59)
        SECOND = MINUTE
        DAYTIME = HOUR ':' MINUTE ':' SECOND
        YEAR = SPAN(DIGITS)
        DATEANDTIME = '[' WEEKDAY WS MONTH WS DAYOFMONTH WS DAYTIME WS YEAR ']'
        OCTET = (SPAN(DIGITS) $ NUMBER) *LE(NUMBER,255)
        IPADDRESS = OCTET '.' OCTET '.' OCTET '.' OCTET
        ERRSTR = '[error]'
        CLIENTINFO = '[client' WS (IPADDRESS . CLIENTIP) ']'
        FILEERR = 'File does not exist:'
        FILENOTFOUNDERROR = DATEANDTIME WS ERRSTR WS CLIENTINFO WS
+        FILEERR WS REM . PATH
        DANGEROUSPATH = '/winnt/system32/cmd.exe' | '/scripts/root.exe' |
+        '/MSADC/root.exe' | "/.." | "../"
LOOP    LOGLINE = INPUT* Anchor the matching of the error message for efficiency
        &ANCHOR = 1
* We're using unevaluated expressions ("thunks") and so must do full scans
        &FULLSCAN = 1
LOGLINE FILENOTFOUNDERROR :F(LOOP)
* Scan the path, using an unanchored match, for strings betraying a worm
        &ANCHOR = 0
        &FULLSCAN = 0
        PATH DANGEROUSPATH :F(LOOP)
        HOST(1,'logger -t wormblock -pauth.notice Nimda or similar attack detected!'
+        'Blocking IP address ' CLIENTIP)
        HOST(1,'route -nq add -host ' CLIENTIP ' 127.0.0.1 -blackhole')
        :(LOOP)

* Note that a blackhole route is a brute force blocking method. It
* allows the first SYN to arrive but blocks the outgoing SYN-ACK,
* causing the TCP three-way handshake to fail. Its advantage is
* that it works on nearly any system regardless of configuration.

* If you're running a firewall, you can replace the route command
* above with ones that add firewall rules. Here are samples for
* FreeBSD's ipfw:
*       HOST(1,'/sbin/ipfw -q add deny all from any to ' CLIENTIP)
*       HOST(1,'/sbin/ipfw -q add deny all from ' CLIENTIP ' to any')
* The commands for ipf and pf are similar.
* If you want to block attacks at a different machine (say, the
* firewall that guards your entire network), you can use SSH to send
* the firewall similar commands. The exact commands required will
* depend upon your network and firewall configurations.

END
```

Listing 3: Extensible worm blocker/IDS for Apache in 28 lines of SNOBOL4

The example in Listing 3 was written for Philip Budne's free Macro SNOBOL4 for UNIX [22] , which compiles on most BSD UNIX implementations and is present in the NetBSD and FreeBSD ports collections. It is readily portable to other implementations of the language including Catspaw SPITBOL [23] . The author has found SNOBOL4 to be extremely useful for log monitoring -- even more so than Perl -- because its extremely powerful recursive pattern matching allows it to completely parse a log message by executing a single line of code. SNOBOL4 also allows more extensive input validation to be done within a pattern than can easily be done within a

Perl regular expression. For example, in the patterns DAYOFMONTH, HOURS, MINUTES, and SECONDS patterns, the GE() and LE() predicates are applied to strings of digits during pattern matching to ensure that the numbers they represent are within allowable limits. SNOBOL's pattern matching engine can backtrack (or indicate failure) if these conditions are not met.

4.4 Refinements to the Initial Design

Note that the code in Listing 3, like that in Listing 2, immediately and unconditionally blocks any host which attacks the server on which it is running. As noted earlier, this could be an unfortunate administrative decision under certain circumstances. For example, if an infected dial-up user is blocked, subsequent users of that dial-up line will not be able to reach the site. If an attacking machine is behind a NAT router or a proxy, every other user arriving from the same site may be blocked. (This is a particular concern in the case of AOL, which passes all HTTP requests through caching proxies to conserve bandwidth and anonymize users.) Also, a malicious third party could post (or e-mail to users) links which, when followed, caused a block to be put in place.

Fortunately, it is relatively simple to make refinements to the log monitor shown above to handle these problems. A "do not block" list can be added to ensure that the machine does not block itself. By requiring two or more hits from an IP address before blocking it, the monitor can reduce the chances of an accidental block or of a block caused by a maliciously distributed link. Use of the MAPS Dial-up List (DUL) [24] to recognize dial-ups, plus an "amnesty" policy, can protect against long term blocking of dial-ups, though at the expense of a few more hits from worms.

Other refinements suggested during previous presentations of this work include:

- The ability to notify an administrator of the current block list (and/or "repeat offenders") so that s/he can notify administrators by phone or e-mail;
- The ability to mail or page an administrator when an attack is detected from a host on the "do not block" list (or under other conditions);
- The ability to gather statistics (such as the number of hits received from a particular Web address or subnet per hour) and automatically notice anomalies;
- The ability to place the log monitor on a separate machine, so as to preserve both copies of logs and information about attacks or malfunctions in the event of catastrophic system failure or tampering; and
- The ability to view a display and/or reports detailing the log monitor's actions.

All of these refinements are easy to add due to the flexibility of the SNOBOL4 language, which provides associative arrays, record types, indirect references, and other features typically found in high level interpreted languages.

5 Creating a Generalized Logging and Log Monitoring Facility

The goal of the author's ongoing research, however, is not to perfect any one special-purpose log monitor. It is, rather, to learn, via the creation of a diverse collection of log monitors, what features are desirable in a generalized logging and log monitoring facility. The long term goal is to create a single facility -- much more powerful than the `swatch` and `2swatch` scripts mentioned earlier -- which can subsume the functions of simple log monitors and facilitate the generation of more sophisticated ones. Such a facility should be usable across a wide range of operating system platforms, and should allow the creation of monitors via a process which consists as much as possible of configuration rather than programming. Features of this facility are expected to include:

- Compatibility with the "legacy" logging facilities and facility/severity codes of current UNIX implementations;
- The ability to apply pre-written message parsing templates to messages (akin to the "distillation" process used by Lire [25]

but performed in real time) so that rules can refer to message field by name;

- The ability to identify and report messages which were not parsed (possibly indicating an obsolete template and/or a software problem);
- The ability to access all information associated with a log message and the process that generated it -- including the identity of the program, effective user and group ids, facility and severity codes, point of origin (if not on the local system), etc.;
- Accumulation of statistics (e.g. number of e-mail messages received from a specific user or IP address) for use in rules;
- The ability to correlate log messages and statistics produced by different applications, e.g. a POP server and an SMTP server;
- The ability to generate one or more periodically refreshed displays (e.g. bar graphs) based on log statistics;
- The ability to query external databases such as DNS blacklists;
- The ability to maintain, save, and restore internal databases (e.g. of blocked hosts and times at which they were blocked) and report their contents at runtime;
- The ability to "fire" rules at specific times or intervals as well as in response to messages;
- The ability to send log messages to, and accept them on or from, arbitrary UDP or TCP ports;
- The ability to log to another machine via an encrypted connection (e.g. through SSH or SSL);
- Stronger authentication than that implemented in current versions of `syslogd` (most of which use source IP address and port number);
- Flexible notification facilities, including the ability to send notices via e-mail, pager, IRC, and instant messaging systems;
- The ability to issue commands to firewalls, routers, bridges, managed hubs, and remote power controllers; and
- The ability to allow or deny users access to facilities (e.g. by changing group memberships, changing a user's login shell to `/etc/nologin`, or removing and restoring passwords).

Input from system architects and administrators regarding suggested features and functionality is welcome.

6 Conclusions

Any computer system which is connected to the Internet, and/or subject to misuse by its users, requires constant vigilance to fend off attacks and thwart abuse. Because system administrators cannot be expected to monitor system activity 24x7, intelligent agents -- or log monitors -- can alert them to situations that require attention and "hold the fort" until help arrives. Log monitors can also perform highly routine chores -- such as blocking worms and spammers – automatically.

While a handful of "plug and play" log monitors now exist, none contain the features necessary to allow them to perform sophisticated stateful monitoring. The author's goal is to fill this gap by implementing a collection of complex log monitors and then creating a generalized facility which can subsume all of their functions.

To ease the implementation of log monitors, the logging facilities in different UNIX implementations -- which have diverged in subtle ways and often hide useful information from administrators and intelligent agents alike -- should be updated or replaced with a more modern scheme that is backward-compatible with what exists today. It will then be much easier to implement a generalized log monitoring facility that runs on a wide variety of platforms.

Of course, no log monitoring system can completely replace the insight or talents of a

human administrator. As Bruce Schneier, founder of Counterpane Network Security, writes in a white paper posted at http://www.counterpane.com/msm.html :

> *Network attacks can be much more subtle than a broken window. Much depends on context. Software can filter the tens of megabytes of audit information a medium-sized network can generate in a day, but software is too easy for an attacker to fool. Intelligent alert requires people. People to analyze what the software finds suspicious. People to delve deeper into suspicious events, determining what is really going on. People to separate false alarms from real attacks. People who understand context.*[26]

The correct approach, therefore, is not one that eliminates people but one that uses intelligent agents -- log monitors -- as a first line of defense. This frees skilled administrators from the tedium of reviewing logs, so that they may focus on the bona fide anomalies detected by log monitors and on other problems more worthy of their talents.

7 Acknowledgments

Thanks to the many attendees of BSDCon Europe 2001 who provided many useful suggestions regarding this ongoing work, and to "shepherd" Gregory Neil Shapiro who guided the preparation of this paper. Thanks also to the authors and maintainers of the BSDs and of the other utilities mentioned in this document for their contributions to the state of the art. Trademarks mentioned in this document are the property of their respective owners.

References

[1] Stephen E. Hansen and E. Todd Atkins. Centralized System Monitoring With Swatch. In *Proceedings of the Seventh Systems Administration Conference (LISA)*, Monterey, CA, Nov. 1993. Paper URL: http://www.stanford.edu/~atkins/swatch/lisa93.html . Software at URL: ftp://ftp.stanford.edu/general/security-tools/swatch.

[2] Pacific Institute for Computer Security (PICS) research group, San Diego Supercomputer Center (SDSC). 2swatch. Software at URL: ftp://ftp.sdsc.edu/pub/sdsc/security/PICS/2swatch/.

[3] Paul Vixie and contributors. cron. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=cron&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[4] Paul Traina and Brian Somers. periodic. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=periodic&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[5] Tom Boutell. Wusage. Software and documentation at URL: http://www.boutell.com/wusage/.

[6] Kai Schlichting. Kai's Spamshield. Software and documentation at URL: http://spamshield.conti.nu/.

[7] Eric Allman, The University of Califonia at Berkeley, The FreeBSD Project and contributors. syslogd. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslogd&apropos=0&sektion=8&manpath=FreeBSD+5.0-current&format=html.

[8] Eric Allman, University of California at Berkeley and contributors. syslogd. 4.4BSD Lite2 version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslogd&apropos=0&sektion=8&manpath=4.4BSD+Lite2&format=html.

[9] Eric Allman, The University of Califonia at Berkeley, The FreeBSD Project and contributors. syslogd.conf. FreeBSD 5.0-current version documented at URL: http://www.FreeBSD.org/cgi/man.cgi?query=syslog.conf&apropos=0&sektion=5&manpath=FreeBSD+5.0-current&format=html.

[10] Matija Grabnar. File::Tail. Software at URL: http://www.cpan.org/modules/by-module/File/File-Tail-0.98.tar.gz.

[11] BalaBit IT Ltd. syslog-ng. Software and documentation at URL: http://www.balabit.hu/en/downloads/syslog-ng/.

[12] Core-SDI. msyslog. Software and documentation at URL: http://community.corest.com/pub/msyslog/.

[13] The Apache Software Foundation. Apache HTTPD Server Project. Software and documentation at URL: http://www.apache.org/.

[14] Computer Emergency Response Team (CERT). Advisory CA-2001-26: Nimda Worm. At URL: http://www.cert.org/advisories/CA-2001-26.html.

[15] Computer Emergency Response Team (CERT). Advisory CA-2001-19: "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. At URL: http://www.cert.org/advisories/CA-2001-19.html.

[16] Computer Emergency Response Team (CERT). Advisory CA-2001-11: sadmind/IIS Worm. At URL: http://www.cert.org/advisories/CA-2001-11.html.

[17] The Apache Software Foundation. mod_setenvif. Documentation at URL: http://httpd.apache.org/docs/mod/mod_setenvif.html#setenvif.

[18] The Apache Software Foundation. mod_log_config. Documentation at URL: http://httpd.apache.org/docs/mod/mod_log_config.html#customlog.

[19] The Apache Software Foundation. Apache Core Features.

HostNameLookups directive. At URL: http://httpd.apache.org/docs/mod/core.html#hostnamelookups.

[20] R. E. Griswold, J. F. Poage, I. P. Polonsky. The SNOBOL4 Programming Language, 2nd Edition. Bell Telephone Laboratories/Prentice-Hall, 1971.

[21] Phil Budne. Phil's SNOBOL Resources Page. At URL: http://people.ne.mediaone.net/philbudne/snobol.html.

[22] Phil Budne. Macro Implementation of SNOBOL4 in C (C-MAINBOL). Software and documentation at URL: http://people.ne.mediaone.net/philbudne/src.html#snobol.

[23] Mark Emmer. Catspaw SPITBOL. Information at URL: ftp://ftp.snobol4.com/specshet.pdf

[24] Mail Abuse Prevention System (mail-abuse.org). MAPS DUL Introduction. At URL: http://www.mail-abuse.org/dul/intro.htm.

[25] LogReport Foundation. Report Production Line. At URL: http://www.logreport.org/documentation/about=architecture.

[26] Bruce Schneier. Managed Security Monitoring: Network Security for the 21st Century . At URL: http://www.counterpane.com/msm.html.