5

INVESTIGATING EVIDENCE FROM LINUX LOGS

The computer term *log* originates from an ancient sailor's technique for measuring the speed of a moving ship. A wooden log attached to a long rope was thrown overboard behind the ship. The rope had regularly spaced knots that sailors would count as the moving ship distanced itself from the floating log. They could calculate the speed of the ship from the number of knots counted over a period of time. Regular measurements of the ship's speed were recorded in the ship's "log book" or log.

Over time, the word *log* came to represent a variety of recorded periodic measurements or events. Log books are still used by organizations to document visitors entering buildings, the delivery of goods, and other activities that need a written historical record. The concept of a computer login and logout was created to control and record user activity. Early time-sharing computer systems were expensive and needed to keep track of computing resources consumed by different users. As the cost of storage capacity and processing power dropped, the use of logging expanded to nearly all parts of a modern computer system. This wealth of logged activity is a valuable source of digital evidence and helps forensic investigators reconstruct past events and activity.

Traditional Syslog

The traditional logging system on Unix and Unix-like operating systems such as Linux is *syslog*. Syslog was originally written for the sendmail software package in the early 1980s and has since become the de facto logging standard for IT infrastructure.

Syslog is typically implemented as a daemon (also known as a collector) that listens for log messages from multiple sources, such as packets arriving over network sockets (UDP port 514), local named pipes, or syslog library calls (see Figure 5-1).

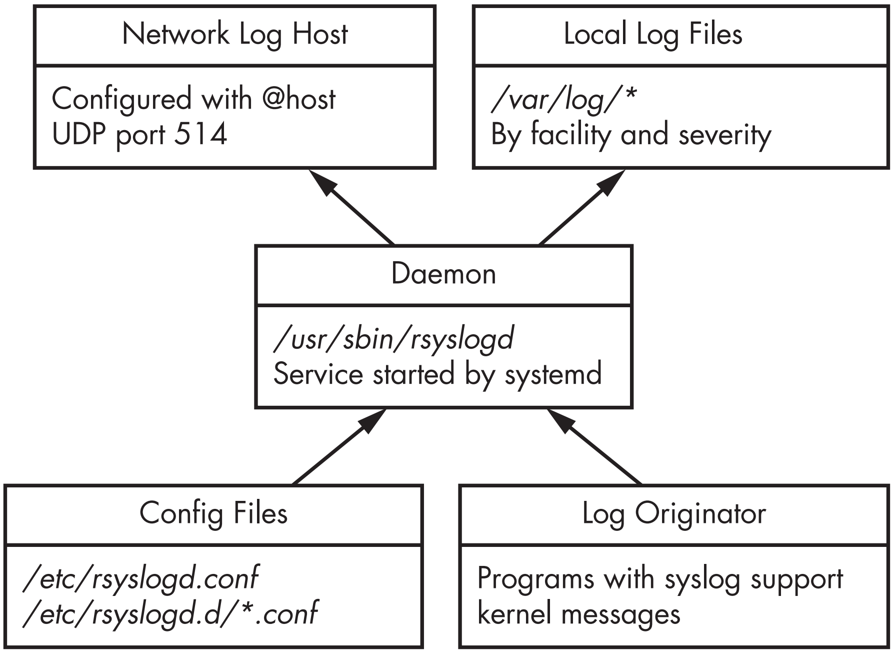

Figure 5-1: Traditional syslog architecture (rsyslog)

The syslog architecture and network protocol is defined in RFC 5424. Linux distributions have historically included one of several implementations of syslog for local system logging, the most common being *rsyslog*.

Syslog Facility, Severity, and Priority

The syslog standard defines the format of messages and several characteristics of log entries. These characteristics are *facility*, *severity*, and *priority*.

The message facility allows the categorization of logs depending on a subsystem. RFC 5424 documents 24 syslog message facilities. The rsyslog .conf(5) man page and the Linux *syslog.h* header file define the facilities as follows:

```
0   kern: kernel messages
1   user: random user-level messages
2   mail: mail system
3   daemon: system daemons
4   auth: security/authorization messages
5   syslog: messages generated internally by syslogd
6   lpr: line printer subsystem
7   news: network news subsystem (obsolete)
8   uucp: UUCP subsystem (obsolete)
9   cron: clock daemon
10  authpriv (auth-priv): security/authorization messages
11  ftp: FTP daemon
12  reserved
13  reserved
14  reserved
15  reserved
16  local0: reserved for local use
17  local1: reserved for local use
18  local2: reserved for local use
19  local3: reserved for local use
20  local4: reserved for local use
21  local5: reserved for local use
22  local6: reserved for local use
23  local7: reserved for local use
```

Some of these facility codes, like `news` (Usenet) or `uucp` (Unix-to-Unix copy) are obsolete and might be explicitly redefined by a system administrator at a local site. The last eight "local" facilities are reserved specifically for local sites to use as needed.

One internal facility called `mark` is often implemented separately from the syslog standard. If used, the syslog daemon generates `mark` log entries, together with a timestamp, at regularly defined intervals. These markers indicate that the logging subsystem was still functional during periods of time when no logs were received. In a forensic examination, the marks are interesting as potential indicators of the absence of certain activity, which can be useful information in an investigation.

There are eight severity levels, with zero being the most severe. The highest numbers generate the most volume of information and are often enabled on demand for troubleshooting or debugging. The severity level can be represented as either a numeric value or a text label. The levels are listed here together with the short or alternate names and description:

```
0  emergency (emerg or panic): system is unusable
```

```
1  alert (alert): action must be taken immediately
2  critical (crit): critical conditions
3  error (err): error conditions
4  warning (warn): warning conditions
5  notice (notice): normal but significant condition
6  informational (info): informational messages
7  debug (debug): debug-level messages
```

These severity levels are interesting from a forensic readiness perspective. If a particular syslog-generating component is at heightened risk or suspicion, or if there is an ongoing incident, the logging severity can be changed temporarily to increase the verbosity of the logs. Some tools and documentation may use the word priority when referring to severity.

The priority, or *PRI* value, of a syslog message is calculated from the facility and severity (by multiplying the facility by eight and then adding the severity). The syslog daemon can use the priority number to decide how to handle the message. These decisions include the location and file to save, filtering, which host(s) to forward messages to, and so on.

Syslog Configuration

The configuration of the local syslog daemon is important to know in a forensic investigation. The configuration file entries (both defaults and administrator customization) direct the investigator to where logs are located, which severity levels have been logged, and what other logging hosts are involved. Common syslog daemon configuration file locations are:

- */etc/syslog.conf*
- */etc/rsyslog.conf*
- */etc/rsyslog.d/*.conf*
- */etc/syslog-ng.conf*
- */etc/syslog-ng/**

These are plaintext files that any text editor can read. The examples here include BSD syslog, rsyslog, and syslog-ng implementations.

The configuration files define the location and contents of the logs managed by the daemon. A typical syslog configuration line has two fields: the selector and the action. The *selector* field is composed of the facility and severity (separated by a dot). The *action* field defines the destination or other action taken when logs match the selector. The following is an example rsyslog configuration file:

```
#*.debug        /var/log/debug
kern.*          /var/log/kern.log
mail.err        /var/log/mail.err
*.info          @loghost
```

The first line is commented out and intended for debugging when needed. The second line sends all kernel logs to */var/log/kern.log*, regardless of severity. In the third line, mail logs with a severity of *error* or more are sent to the */var/log/mail.err* logfile. These files are stored locally and can be easily located and examined. The last line sends all log messages (any facility) with a severity of *info* or more to another host on the network. The `@` indicates a network destination and `loghost` is a central logging infrastructure.

The network destinations are especially interesting for an investigation because they indicate a separate non-local source of log data that can be collected and examined. If identical logs are stored both locally and on a remote log host, the correlation can be interesting if the data doesn't match. A mismatch may indicate malicious modification of one of the logs.

On Linux systems, the */var/log/* directory is the most common place to save logs. However, these flat text files have scalability, performance, and reliability challenges when high volumes of log data are ingested. Enterprise IT environments still use the syslog protocol over the network, but messages are often saved to high-performance databases or systems designed specifically for managing logs (Splunk is a popular example). These databases can be a valuable source of information for investigators and enable a quick iterative investigative process. Very large text-based logfiles can take a long time to query (`grep`) for keywords compared to database log systems.

Analyzing Syslog Messages

A syslog message transmitted across a network is not necessarily identical to the corresponding message that is saved to a file. For example, some fields may not be saved (depending on the syslog configuration).

A program with built-in syslog support, also known as an *originator*, uses programming libraries or external programs to generate syslog messages on a local system. Programs implementing syslog are free to choose any facility and severity they wish for each message.[1]

To illustrate, let's take a look at the `logger`[2] tool for generating syslog messages:

```
$ logger -p auth.emerg "OMG we've been hacked!"
```

The syslog message from this example can be observed traversing a network. When captured and decoded by `tcpdump`, it looks like this:

```
21:56:32.635903 IP (tos 0x0, ttl 64, id 12483, offset 0, flags [DF],
proto UDP (17), length 80)
    pc1.42661 > loghost.syslog: SYSLOG, length: 52
        Facility auth (4), Severity emergency (0)
        Msg: Nov  2 21:56:32 pc1 sam: OMG we've been hacked!
```

1. The syslog daemon or program used may have some restrictions. For example, the `logger` program may prevent users from specifying the kernel facility.
2. See the logger(1) man page for more information.

Some information (like severity or facility) in the original syslog messages might not be stored in the destination logfiles depending on how the syslog daemon is configured. For example, a typical rsyslog configuration will log the syslog message from the preceding example as follows:

```
Nov  2 21:56:32 pc1 sam: OMG we've been hacked!
```

Here, the severity and facility are not logged locally; however, the syslog daemon is aware of them when the message arrives and may use this information to choose the log destination. On the `loghost`, the UDP port numbers (the source port in particular) are also not logged unless the site is logging firewall traffic or using netflow logging.

Most syslog systems log a few standard items by default. Here is an example of a typical log entry generated by rsyslog:

```
Nov  2 10:19:11 pc1 dhclient[18842]: DHCPACK of 10.0.11.227 from 10.0.11.1
```

This log line contains a timestamp, the local hostname, the program that generated the message together with its process ID (in square brackets), followed by the message produced by the program. In this example, the `dhclient` program (PID 18842) is logging a DHCP acknowledgement containing the machine's local IP address (10.0.11.227) and the IP address of the DHCP server (10.0.11.1).

Most Linux systems use log rotation to manage retention as logs grow over time. Older logs might be renamed, compressed, or even deleted. A common software package for this is logrotate, which manages log retention and rotation based on a set of configuration files. The default configuration file is */etc/logrotate.conf*, but packages may supply their own logrotate configuration and save it in */etc/logrotate.d/** during package installation. During a forensic examination, it is useful to check whether and how logfiles are rotated and retained over time. The logrotate package can manage any logfile, not only those generated by syslog.

Forensic examiners should be aware that syslog messages have some security issues that may affect the evidential value of the resulting logs. Thus, all logs should be analyzed with some degree of caution:

- Programs can generate messages with any facility and severity they want.
- Syslog messages sent over a network are stateless, unencrypted, and based on UDP, which means they can be spoofed or modified in transit.
- Syslog does not detect or manage dropped packets. If too many messages are sent or the network is unstable, some messages may go missing, and logs can be incomplete.
- Text-based logfiles can be maliciously manipulated or deleted.

In the end, trusting logs and syslog messages involves assessing and accepting the risks of integrity and completeness.

Some Linux distributions are starting to switch over to the systemd journal for logging and aren't installing a syslog daemon. It is likely that locally installed syslog daemons on desktop Linux systems will decline in popularity, but in server environments, syslog will remain a de facto standard for network-based logging.

Systemd Journal

The shortcomings of the aging syslog system have resulted in a number of security and availability enhancements. Many of these enhancements have been added to existing syslog daemons as non-standard features and never gained widespread use among Linux distributions. The systemd journal was developed from scratch as an alternative logging system with additional features missing from syslog.

Systemd Journal Features and Components

The design goals and decisions of the systemd journal were to add new features to those already found in traditional logging systems and integrate various components that had previously functioned as separate daemons or programs. Systemd journal features include:

- Tight integration with systemd
- `stderr` and `stdout` from daemons is captured and logged
- Log entries are compressed and stored in a database format
- Built-in integrity using forward secure sealing (FSS)
- Additional trusted metadata fields for each entry
- Logfile compression and rotation
- Log message rate limiting

With the introduction of FSS and trusted fields, the developers created a greater focus on log integrity and trustworthiness. From a digital forensics perspective, this is interesting and useful because it strengthens the reliability of the evidence.

The journal offers network transfer of messages to another log host (central logging infrastructure) in a similar way to traditional logging, but with a few enhancements:

- TCP-based for stateful established sessions (solves dropped packet issue with UDP)
- Encrypted transmission (HTTPS) for confidentiality and privacy
- Authenticated connections to prevent spoofing and unauthorized messages
- Message queuing when `loghost` is unavailable (no lost messages)
- Signed data with FSS for message integrity

- Active or passive message delivery modes

These networking features allow a more secure logging infrastructure to be built, with a focus on integrity and completeness. A significant problem with syslog was the UDP-based stateless packet transmission. With the systemd journal, reliability and completeness of log transmission is addressed.

If the journal networking features are used, check the */etc/systemd/journal-upload.conf* file for the `"URL="` parameter containing the hostname of a central log host. This is a forensic artifact that may point to the existence of logs in a different location and may be important on systems for which logging is not persistent.

Figure 5-2 shows the architectural component diagram of systemd journal networking.

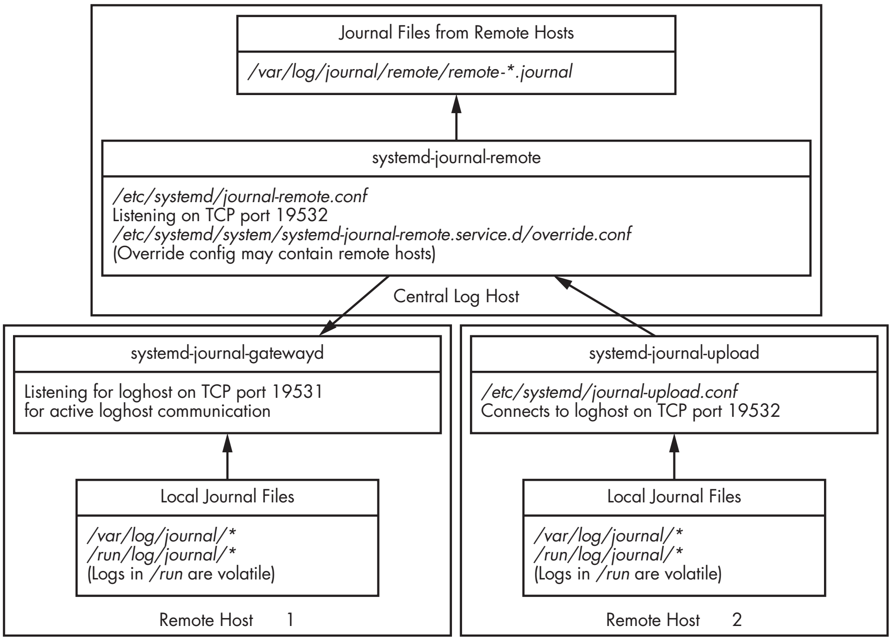
Journal Files from Remote Hosts
/var/log/journal/remote/remote-*.journal
systemd-journal-remote
/etc/systemd/journal-remote.conf
Listening on TCP port 19532
/etc/systemd/system/systemd-journal-remote.service.d/override.conf
(Override config may contain remote hosts)
Central Log Host
systemd-journal-gatewayd
Listening for loghost on TCP port 19531
for active loghost communication
Local Journal Files
/var/log/journal/*
/run/log/journal/*
(Logs in /run are volatile)
Remote Host 1
systemd-journal-upload
/etc/systemd/journal-upload.conf
Connects to loghost on TCP port 19532
Local Journal Files
/var/log/journal/*
/run/log/journal/*
(Logs in /run are volatile)
Remote Host 2

Figure 5-2: Systemd journal networking

See the systemd-journal-remote(8), systemd-journal-gatewayd(8), and systemd-journal-upload(8) man pages for more information about the journal networking features. Although those features are innovative and greatly improve traditional logging, they are systemd specific and not compatible or well known outside the Linux community.

Systemd Journal Configuration

Understanding the configuration of the systemd journal helps us assess the potential for finding forensic evidence on a system. The journal functions as a normal Linux daemon (see Figure 5-3) called systemd-journald and is well documented in the systemd-journald(8) man page. You can find the *enable* status of the journal daemon at boot time by examining the systemd unit files (*systemd-journald.service*).

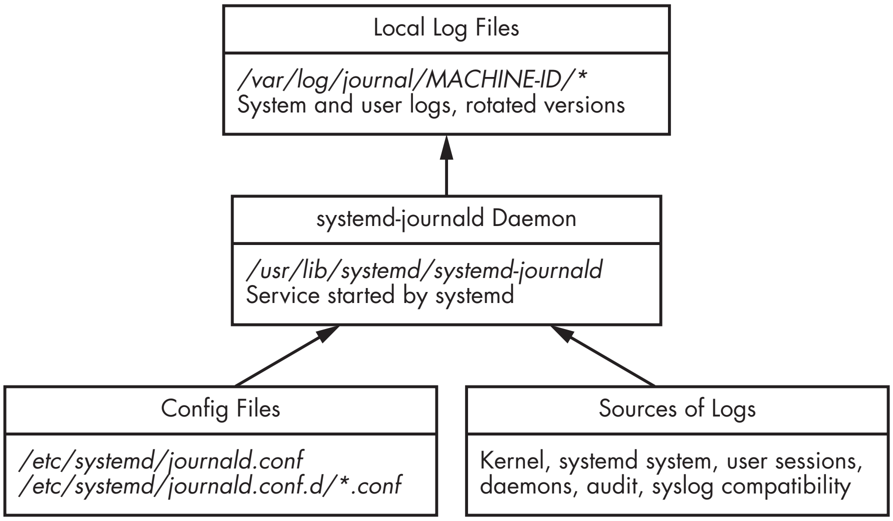
Local Log Files
/var/log/journal/MACHINE-ID/*
System and user logs, rotated versions
systemd-journald Daemon
/usr/lib/systemd/systemd-journald
Service started by systemd
Config Files
/etc/systemd/journald.conf
/etc/systemd/journald.conf.d/*.conf
Sources of Logs
Kernel, systemd system, user sessions, daemons, audit, syslog compatibility

Figure 5-3: Systemd journal daemon

The systemd journal has several configuration parameters that define aspects of its operation (described in the journald.conf(5) man page). Common configuration file locations for the journal are as follows:

- */etc/systemd/journald.conf*
- */etc/systemd/journald.conf.d/*.conf*
- */usr/lib/systemd/journald.conf.d/*.conf*

The configuration file specifies whether logs are volatile or persistent with the `"Storage="` parameter. Persistent logs, if configured, are stored in a binary format in */var/log/journal/*. If logs are configured to be volatile, they will be stored in */run/log/journal/* and exist only when the system is running; they are not available for postmortem forensic analysis. If `"ForwardToSyslog=yes"` is set, journal logs are sent to the traditional syslog system on the local machine and stored in local logfiles (*/var/log/*) or possibly forwarded to a central log host.

On systems with a persistent journal, the */var/log/journal/* directory contains a subdirectory named after the machine-id (as found in */etc/machine-id*) that contains the local journal logfiles. The magic number identifying a journal file is the initial byte sequence 0x4C504B5348485248 or LPKSHHRH.

The journal files contain both system and user logs. System logs are generated by system services and the kernel. User logs are generated by user login sessions (shell or desktop) and various programs that a user executes. Users may read their own logs, but they are not permitted to modify or write to them directly.

Here is an example of a system with a machine-id of 506578466b474f6e88ec fbd783475780 and the corresponding directory with journal logfiles:

```
$ ls /var/log/journal/506578466b474f6e88ecfbd783475780
user-1001@0005aa24f4aa649b-46435710c1877997.journal~
user-1001@dd54beccfb52461d894b914a4114a8f2-00000000000006a8-0005a1d176b61cce.journal
system@e29c14a0a5fc46929ec601deeabd2204-0000000000000001-00059e3713757a5a.journal
user-1001@dd54beccfb52461d894b914a4114a8f2-0000000000000966-0005a1d17821abe4.journal
system@e29c14a0a5fc46929ec601deeabd2204-000000000000189c-00059e37774baedd.journal
user-1001.journal
system.journal
```

Normal journal logs have a file extension of **.journal*. If the system crashed or had an unclean shutdown, or if the logs were corrupted, the filename will end in a tilde (**.journal~*). Filenames of logs that are in current use, or "online," are *system.journal* and *user-UID.journal* (where UID is the numeric ID of a user). Logs that have been rotated to an "offline" or "archived" state have the original filename followed by @ and a unique string. The unique string between the @ and *.journal* is broken into three parts that describe the content of the logfile.

Let's analyze the composition of a long journal filename, as shown in this example:

```
/var/log/journal/506578466b474f6e88ecfbd783475780/system@e29c14a0a
5fc46929ec601deeabd2204-000000000000189c-00059e37774baedd.journal
```

The deconstructed parts are as follows:

- `/var/log/journal/` The location (path) of persistent journal files
- `506578466b474f6e88ecfbd783475780/` The machine-id directory
- `system@` Indicates a system logfile that has been archived
- `e29c14a0a5fc46929ec601deeabd2204` A sequence ID
- `-000000000000189c` The first sequence number in the file
- `-00059e37774baedd` Hexadecimal timestamp of the first log entry
- `.journal` Indicates a systemd journal logfile

The hexadecimal timestamp refers to when the first entry was added to the journal. For the familiar epoch in seconds, convert this timestamp to decimal and then strip off the last six digits.

If the system is receiving journal logs over the network from other hosts (by `systemd-journal-upload` or `systemd-journal-gatewayd`), a *remote/* directory may exist that contains logs for each remote host. These logs will have filenames like *remote-HOSTNAME.journal*.

The journal logs the systemd boot process and follows the starting and stopping of unit files until the system is shut down. Linux systems maintain a unique 128-bit boot-id that can be found (on a running system) in */proc/sys/kernel/random/boot_id*. The boot-id is randomly generated by the kernel at every boot, and it acts as a unique identifier for a particular duration of uptime (from boot to shutdown/reboot). The boot-id is recorded in the journal logs and used to distinguish time periods between boots (for example, `journalctl --list-boots`) and to show logs since the last boot (for example, `journalctl -b`). These journalctl options can also be applied to a file or directory for offline analysis. The boot-id may be of interest during a forensic examination if any malicious activity was known to have occurred during a specific boot period.

Analysis of Journal File Contents

If commercial forensic tool support for journal files is unavailable, you can copy and analyze the journal files on a separate Linux analysis machine using the `journalctl` command. This command allows you to list the journal contents, search the journal, list individual boot periods, view additional log metadata (journald specific), view `stderr` and `stdout` from programs, export to other formats, and more.

After copying the desired journal files or the entire journal directory to your analysis machine, you can use `journalctl` file and directory flags to specify the location of the journal files to be analyzed:

```
$ journalctl --file <filename>
$ journalctl --directory <directory>
```

Specifying a file will operate only on that single file. Specifying a directory will operate on all the valid journal files in that directory.

Each journal file contains a header with metadata about itself, which you can view by using the `--header` flag of `journalctl`; for example:

```
$ journalctl --file system.journal --header
File path: system.journal
File ID: f2c1cd76540c42c09ef789278dfe28a8
Machine ID: 974c6ed5a3364c2ab862300387aa3402
Boot ID: e08a206411044788aff51a5c6a631c8f
Sequential number ID: f2c1cd76540c42c09ef789278dfe28a8
State: ONLINE
Compatible flags:
Incompatible flags: COMPRESSED-ZSTD KEYED-HASH
Header size: 256
Arena size: 8388352
Data hash table size: 233016
Field hash table size: 333
Rotate suggested: no
Head sequential number: 1 (1)
Tail sequential number: 1716 (6b4)
Head realtime timestamp: Thu 2020-11-05 08:42:14 CET (5b3573c04ac60)
Tail realtime timestamp: Thu 2020-11-05 10:12:05 CET (5b3587d636f56)
Tail monotonic timestamp: 1h 29min 53.805s (1417ef08e)
```

```
Objects: 6631
Entry objects: 1501
Data objects: 3786
Data hash table fill: 1.6%
Field objects: 85
Field hash table fill: 25.5%
Tag objects: 0
Entry array objects: 1257
Deepest field hash chain: 2
Deepest data hash chain: 1
Disk usage: 8.0M
```

The output provides a technical description of the journal file, the timestamps of the period covered (head and tail), the number of logs (`Entry objects`), and other statistics. You can find more information about the journal file format here:[3] *https://systemd.io/JOURNAL_FILE_FORMAT/*.

The following example is a basic listing of a specific journal file's contents using the `journalctl` command:

```
$ journalctl --file system.journal
-- Logs begin at Thu 2020-11-05 08:42:14 CET, end at Thu 2020-11-05 10:12:05 CET. --
Nov 05 08:42:14 pc1 kernel: microcode: microcode updated early to revision 0xd6,
date = 2020-04-27
Nov 05 08:42:14 pc1 kernel: Linux version 5.9.3-arch1-1 (linux@archlinux) (gcc (GCC)
10.2.0, GNU ld (GNU Binutils) 2.35.1) #1 SMP PREEMPT Sun, 01 Nov 2020 12:58:59 +0000
Nov 05 08:42:14 pc1 kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-linux root=
UID=efbfc8dd-8107-4833-9b95-5b11a1b96875 rw loglevel=3 quiet pcie_aspm=off
i915.enable_dpcd_backlight=1
...
Nov 05 10:11:53 pc1 kernel: usb 2-1: Product: USB Flash Drive
Nov 05 10:11:53 pc1 kernel: usb 2-1: Manufacturer: Philips
Nov 05 10:11:53 pc1 kernel: usb 2-1: SerialNumber: 070852A521943F19
Nov 05 10:11:53 pc1 kernel: usb-storage 2-1:1.0: USB Mass Storage device detected
...
Nov 05 10:12:05 pc1 sudo[10400]:      sam : TTY=pts/5 ; PWD=/home/sam/test ; USER=root ;
COMMAND=/usr/bin/cp /etc/shadow .
Nov 05 10:12:05 pc1 sudo[10400]: pam_unix(sudo:session): session opened for user
root(uid=0) by (uid=0)
...
```

In this example, `system.journal` is the name of the file being analyzed. The first line is informational, indicating the time period contained in the output. Some of the output is from the kernel, similar to the output from the `dmesg` command. Other lines are similar to syslog, starting with a timestamp, hostname, daemon name, and the process ID in square brackets, and

3. The best resource for understanding the journal is the systemd source code: *https://github.com/systemd/systemd/tree/master/src/journal*.

ending with the log message. The `journalctl` command may also add other informational lines like `-- Reboot --` to indicate the end of a boot period (and the start of a new boot-id).

Each log entry has journal-specific metadata stored together with the log message. A full extraction of a journal entry can be done with a verbose output (`-o verbose`) parameter. The following is a verbose journal entry from the OpenSSH daemon:

```
$ journalctl --file system.journal -o verbose
...
Thu 2020-11-05 08:42:16.224466 CET [s=f2c1cd76540c42c09ef789278dfe28a8;i=4a9;
b=e08a206411044788aff51a5c6a631c8f;m=41d525;t=5b3573c2653ed;x=a1434bf47ce8597d]
    PRIORITY=6
    _BOOT_ID=e08a206411044788aff51a5c6a631c8f
    _MACHINE_ID=974c6ed5a3364c2ab862300387aa3402
    _HOSTNAME=pc1
    _UID=0
    _GID=0
    _SYSTEMD_SLICE=system.slice
    SYSLOG_FACILITY=4
    _CAP_EFFECTIVE=1ffffffffff
    _TRANSPORT=syslog
    SYSLOG_TIMESTAMP=Nov  5 08:42:16
    SYSLOG_IDENTIFIER=sshd
    SYSLOG_PID=397
    _PID=397
    _COMM=sshd
    _EXE=/usr/bin/sshd
    _CMDLINE=sshd: /usr/bin/sshd -D [listener] 0 of 10-100 startups
    _SYSTEMD_CGROUP=/system.slice/sshd.service
    _SYSTEMD_UNIT=sshd.service
    _SYSTEMD_INVOCATION_ID=7a91ff16d2af40298a9573ca544eb594
    MESSAGE=Server listening on :: port 22.
    _SOURCE_REALTIME_TIMESTAMP=1604562136224466
...
```

This output provides structured information with unique identifiers, systemd information, syslog `FACILITY` and `PRIORITY` (severity), the process that produced the log message, and more. The systemd.journal-fields(7) man page describes the fields of a journal log entry.

Journal files are saved in a binary format that's open and documented. The `journalctl` tool can be used to perform various examination tasks on journal files, but some forensic investigators may prefer to export the journal contents into another format for analysis. Two useful output formats are *export* and *json*. The export format is similar to the verbose format, with each entry separated by a blank line (this is technically a binary format, but it contains mostly readable text). The *json* output generates the journal entries in JSON for easy scripting or ingesting into other analysis tools. Here are two command line examples of creating *.json* and *.export* files with the full contents of a journal file:

```
$ journalctl --file system.journal -o json > system.journal.json
$ journalctl --file system.journal -o export > system.journal.export
```

The new files created are *system.journal.json* and *system.journal.export*, which other (non-Linux) tools can easily read. Another output format is *.json-pretty*, which produces JSON in a more human-readable format.

Searching journal files is done by including match arguments in the form `FIELD=VALUE`, but the exact value you're searching for needs to be specified. This type of search can be useful for extracting logs from a particular service. For example, to extract all logs from the `sshd.service` unit:

```
$ journalctl --file system.journal _SYSTEMD_UNIT=sshd.service
-- Logs begin at Thu 2020-11-05 08:42:14 CET, end at Thu 2020-11-05 10:12:05 CET. --
Nov 05 08:42:16 pc1 sshd[397]: Server listening on 0.0.0.0 port 22.
Nov 05 08:42:16 pc1 sshd[397]: Server listening on :: port 22.
...
```

Regular expressions (regex) can be used with the `--grep=` parameter, but they can search only the message fields, not the journal metadata. The search syntax is not very flexible for forensic investigators, and it may be easier to export the journal to another format and use familiar tools like `grep` or other text search tools.

It is worth mentioning that the systemd journal can log `stdout` and `sdterr` of daemons and other unit files. With traditional syslog, that information was typically lost because the daemon would detach from the controlling terminal when it started. Systemd preserves this output and saves it to the journal. You can list this output by specifying the `stdout` transport:

```
$ journalctl --file user-1000.journal _TRANSPORT=stdout
```

Transports specify how the journal received the log entry. There are other transports like syslog, kernel, audit, and so on. These transports are documented in the systemd.journal-fields(7) man page.

If a journal file contains FSS information, the integrity can be checked using the `--verify` flag. In the following example, a journal file is checked, and `PASS` indicates that the file integrity is verified:

```
$ journalctl --file system.journal --verify
PASS: system.journal
```

If a journal file has been tampered with, it will fail the verification:

```
$ journalctl --file user-1002.journal --verify
38fcc0: Invalid hash (afd71703ce7ebaf8 vs.49235fef33e0854e
38fcc0: Invalid object contents: Bad message
File corruption detected at user-1002.journal:38fcc0 (of 8388608 bytes, 44%).
FAIL: user-1002.journal (Bad message)
```

In this example, the FSS integrity failed at byte offset 0x38fcc0 of the journal file, with a log entry that was maliciously modified. If a logfile has been tampered with in multiple places, the verification will fail at the first instance of tampering.

When investigating incidents that happened during a known window of time, extracting logs from an explicit time frame is useful. The `journalctl` command can extract logs with a specified time range using two flags: `-S` (since) and `-U` (until). Any logs existing since the time of `-S` until (but not including) the time of `-U` are extracted.

The following two examples are from a Linux forensic analysis machine where journal files have been copied to an evidence directory for examination using the `journalctl` command:

```
$ journalctl --directory ./evidence -S 2020-11-01 -U 2020-11-03
$ journalctl --file ./evidence/system.journal -S "2020-11-05 08:00:00"
-U "2020-11-05 09:00:00"
```

In the first example, the directory containing the journal files is specified and logs from November 1 and November 2 are extracted. The second example specifies a more exact time range and extracts logs from 8 AM to 9 AM on November 5. See the journalctl(1) man page for other variations of the time and date string.

The new features of systemd's journal mechanism are very much aligned with forensic-readiness expectations. The systemd journal offers log completeness and integrity, which are fundamental concepts in digital forensics.

Other Application and Daemon Logs

Programs are not required to use syslog or the systemd journal. A daemon or application may have a separate logging mechanism that completely ignores system-provided logging. Daemons or applications may also use syslog or the journal, but with non-standard facilities or severities and their own message formats.

Custom Logging to Syslog or Systemd Journal

Syslog provides a C library function for programs to generate syslog messages. Systemd provides an API for programs to submit log entries to the journal. Developers are free to use those instead of developing their own logging subsystems. However, the facilities, severities, and format of the message, are all decided by the developer. This freedom can lead to a variety of logging configurations among programs.

In the following examples, each program uses a different syslog facility and severity for logging similar actions:

```
mail.warning: postfix/smtps/smtpd[14605]: ❶ warning: unknown[10.0.6.4]: SASL LOGIN
 authentication failed: UGFzc3dvcmQ6
...
```

```
auth.info sshd[16323]: ❷ Failed password for invalid user fred from 10.0.2.5 port 48932 ssh2
...
authpriv.notice: auth:  pam_unix(dovecot:auth): ❸ authentication failure; logname= uid=0
 euid=0 tty=dovecot ruser=sam rhost=10.0.3.8
...
daemon.info: danted[30614]: ❹ info: block(1): tcp/accept ]: 10.0.2.5.56130 10.0.2.6.1080:
 error after reading 3 bytes in 0 seconds: client offered no acceptable authentication method
```

These logs describe failed logins from a mail server (`postfix`) ❶, secure shell (`sshd`) ❷, an imap server (`dovecot` using `pam`) ❸, and a SOCKS proxy (`dante`) ❹. They all use different facilities (`mail`, `auth`, `authpriv`, `daemon`), and they all use different severities (`warning`, `info`, `notice`). In some cases, additional logs may contain more information about the same event at different facilities or severities. Forensic examiners should not assume all similar log events will use the same facility or severity, but rather should expect some variation.

Daemons may choose to log to a custom or user-defined facility. This is usually defined in the daemon's configuration or from compiled-in defaults. For example:

```
local2.notice: pppd[645]:  CHAP authentication succeeded
local5.info: TCSD[1848]:  TrouSerS trousers 0.3.13: TCSD up and running.
local7.info: apache2[16455]:  ssl: 'AH01991: SSL input filter read failed.'
```

Here a `pppd` daemon is using `local2` as the facility, the `tcsd` daemon that manages the TPM uses `local5`, and an Apache web server (`apache2`) is configured to use `local7`. Daemons can log to whatever facility they want, and system administrators may choose to configure logging to a desired facility.

When an investigation is ongoing or an attack is underway, additional logging may be needed (possibly only temporarily). If there are heightened risks involving potential suspects or victims, logging can be selectively increased to support the collection of digital forensic evidence. For example, consider these potential entities for which selective increased logging could be used:

- A particular user or group
- A geographical region or specific location
- A particular server or group of servers
- An IP address or range of IPs
- Specific software components running on a system (daemons)

Most daemons provide configuration options to increase the verbosity of logging. Some daemons offer very granular possibilities of selective logging. For example, Postfix configuration directives allow increased logging for a specific list of IP addresses or domain names:

```
debug_peer_level = 3
debug_peer_list = 10.0.1.99
```

In this example, a single IP address is selected for increased logging, using Postfix's internal debug levels (3 instead of the default 2). The configuration documentation for each daemon will describe possibilities for verbose, debug, or other selective logging adjustments.

As described in the previous section, the `stdout` and `stderr` of a daemon started with systemd will be captured and logged to the journal, which is also useful from a forensic readiness perspective. If a daemon allows for verbose or debugging output to the console, it can be temporarily enabled for the duration of an incident or investigation.

Independent Server Application Logs

Often applications will manage their own logfiles without the use of local logging systems like syslog or the systemd journal. In those situations, logs are typically stored in a separate logfile or log directory, usually in the */var/log/* directory.

Larger applications may be complex enough to warrant multiple separate logfiles for different subsystems and components. This may include separate logfiles for the following:

- Application technical errors
- User authentication (logins, logouts, and so on)
- Application user transactions (web access, sessions, purchases, and so on)
- Security violations and alerts
- Rotated or archived logs

The Apache web server is a good example. It typically has a separate directory like */var/log/apache2/* or */var/log/httpd/*. The contents of the directory may include logs for the following:

- General web access (*access.log*)
- Web access for individual virtual hosts
- Web access for individual web applications
- Daemon errors (*error.log*)
- SSL error logging

Applications will typically specify the log location, content, and verbosity in their configuration files. A forensic examiner should check for those log locations if it is not otherwise obvious.

Some application installations may be fully contained in a specific directory on the filesystem, and the application may use this directory to store logs together with other application files. This setup is typical of web applications that may be self-contained within a directory. For example, the Nextcloud hosting platform and Roundcube webmail application have such application logs:

- *nextcloud/data/nextcloud.log*
- *nextcloud/data/updater.log*
- *nextcloud/data/audit.log*
- *roundcube/logs/sendmail.log*
- *roundcube/logs/errors.log*

Keep in mind that these logs are generated in addition to the web server access and error logs (apache, nginx, and so on). With web applications, a forensic examiner may find logs in multiple places related to a particular application, event, or incident.

Some applications may store logs in databases instead of text files. These are either full database services like MySQL or Postgres, or local database files like SQLite.

Another interesting log related to programs installed on a system is the *alternatives* log. The alternatives system was originally developed for Debian to allow installation of several concurrent versions of similar programs. Multiple distributions have adopted the alternatives mechanism. The `update-alternatives` script manages the symbolic links to generic or alternative application names located in the */etc/alternatives/* directory. For example, several symlinks are created to provide a `vi` program alternative:

```
$ ls -gfo /usr/bin/vi /etc/alternatives/vi /usr/bin/vim.basic
lrwxrwxrwx 1      20 Aug  3 14:27 /usr/bin/vi -> /etc/alternatives/vi
lrwxrwxrwx 1      18 Nov  8 11:19 /etc/alternatives/vi -> /usr/bin/vim.basic
-rwxr-xr-x 1 2675336 Oct 13 17:49 /usr/bin/vim.basic
```

The timestamp of the */etc/alternatives/* symlink indicates when the last change was made. This information is also recorded in the *alternatives.log* file:

```
$ cat /var/log/alternatives.log
...
update-alternatives 2020-11-08 11:19:06: link group vi updated to point to /usr/bin/vim.basic
...
```

This is a system-wide method of assigning default applications (analogous to XDG defaults for desktop users) and helps build a picture of which programs were used on a system. See the update-alternatives(1) man page[4] for more information.

During a forensic examination, pay close attention to error logs. Error messages reveal unusual and suspicious activity, and help to reconstruct past events. When investigating intrusions, error messages appearing before an incident can indicate pre-attack reconnaissance or prior failed attempts.

4. This might be update-alternatives(8) on some distributions.

Independent User Application Logs

When a user logs in to a Linux system, standard logs are created by the various components of the system (login, pam, display manager, and so on). After a user has logged in to their desktop or shell, further logging may also be saved in locations specific to that user.

The systemd journal saves persistent logs specific to a user's login session in */var/log/journal/*MACHINE-ID/user-UID*.journal*, where *UID* is a user's numeric ID. This log (and the rotated instances) contains traces of a person's login session activity, which may include information like the following:

- Systemd targets reached and user services started
- Dbus-daemon activated services and other activity
- Agents like gnupg, polkit, and so on
- Messages from subsystems like pulseaudio and Bluetooth
- Logs from desktop environments like GNOME
- Privilege escalation like `sudo` or `pkexec`

The format of user journal files is the same as system journal files, and you can use the `journalctl` tool to analyze them (described earlier in the chapter).

Other logs may be saved by programs as they are run by a user. The location of such program logs must be in a directory writable by the user, which generally means they are somewhere in the user's home directory. The most common places for persistent logs are the XDG base directory standards such as *~/.local/share/*APP/* or *~/.config/*APP/*, (where *APP* is the application generating user logs).

The following example shows a Jitsi video chat application log stored in *~/.config/*, which contains error messages:

```
$ cat ~/.config/Jitsi\ Meet/logs/main.log
[2020-10-17 15:20:16.679] [warn] APPIMAGE env is not defined, current
 application is not an AppImage
...
[2020-10-17 16:03:19.045] [warn] APPIMAGE env is not defined, current
 application is not an AppImage
...
[2020-10-21 20:52:19.348] [warn] APPIMAGE env is not defined, current
 application is not an AppImage
```

The benign warning messages shown here were generated whenever the Jitsi application started. For a forensic investigator, the content of these messages may not be interesting, but the timestamps indicate every time the video chat program was started. Trivial errors like this are potentially interesting for reconstructing past events.

Some programs ignore the XDG standard and create hidden files and directories at the root of the user's home directory. For example, the Zoom video chat application creates a *~/.zoom/log/* directory with a logfile:

```
$ cat ~/.zoom/logs/zoom_stdout_stderr.log
ZoomLauncher started.
cmd line: zoommtg://zoom.us/join?action=join&confno=...
...
```

This Zoom log contains a wealth of information, including traces of past conference IDs that were used.

Temporary or non-persistent logs may also be found in *~/.local/cache/* APP/*, as this cache directory is intended for data that can be deleted; In this example, the `libvirt` system for managing the user's KVM/QEMU virtual machines has a log directory with a file for each machine:

```
$ cat ~/.cache/libvirt/qemu/log/pc1.log
2020-09-24 06:57:35.099+0000: starting up libvirt version: 6.5.0, qemu version: 5.1.0,
kernel: 5.8.10-arch1-1, hostname: pc1.localdomain
LC_ALL=C \
PATH=:/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/home/sam/script \
HOME=/home/sam \
USER=sam \
LOGNAME=sam \
XDG_CACHE_HOME=/home/sam/.config/libvirt/qemu/lib/domain-1-linux/.cache \
QEMU_AUDIO_DRV=spice \
/bin/qemu-system-x86_64 \
...
```

Performing a search for **.log* files or directories called "log" across a user's home directory will produce an initial list of files to analyze. Linux applications can produce a significant amount of logs and persistent data that's saved whenever the user runs various programs.

The analysis of individual application logs is outside the scope of this book, but it is worth mentioning that many popular apps store significant amounts of information about past use in a user's home directory. This information often contains a history of files opened, remote host connections, communication with other people, timestamps of usage, devices accessed, and more.

Plymouth Splash Startup Logs

During startup, most desktop distros use the Plymouth system to produce a graphical splash screen while the system is booting. The ESC key can be pressed while waiting to switch to console output. Non-graphical servers can also use Plymouth to provide visible output while a system is booting. The output provides color status indicators with green `[  OK  ]` or red `[FAILED]` messages for each component.

This Plymouth console output is typically saved to the */var/log/boot.log* file; for example:

```
$ cat /var/log/boot.log
...
[  OK  ] Started Update UTMP about System Boot/Shutdown.
[  OK  ] Started Raise network interfaces.
[  OK  ] Started Network Time Synchronization.
[  OK  ] Reached target System Time Synchronized.
[  OK  ] Reached target System Initialization.
[  OK  ] Started Daily Cleanup of Temporary Directories.
[  OK  ] Listening on D-Bus System Message Bus Socket.
[  OK  ] Listening on Avahi mDNS/DNS-SD Stack Activation Socket.
[  OK  ] Started Daily apt download activities.
[  OK  ] Started Daily rotation of log files.
[  OK  ] Started Daily apt upgrade and clean activities.
[  OK  ] Started Daily man-db regeneration.
[  OK  ] Reached target Timers.
[  OK  ] Listening on triggerhappy.socket.
[  OK  ] Reached target Sockets.
[  OK  ] Reached target Basic System.
...
```

This file contains escape codes needed to produce the color indicators. It is safe to view, even if your analysis tool warns that it is a binary file.

Failed components during boot will also appear in the boot log:

```
$ cat /var/log/boot.log
...
[FAILED] Failed to start dnss daemon.
See 'systemctl status dnss.service' for details.
[  OK  ] Started Simple Network Management Protocol (SNMP) Daemon..
[FAILED] Failed to start nftables.
See 'systemctl status nftables.service' for details.
...
```

Rotated versions of the boot log may also exist in the */var/log/* directory.

This boot log can be interesting to analyze in a forensic investigation. It shows the sequence of events during previous boots and may provide useful error messages. For example, the preceding error message indicates that the Linux firewall rules (`nftables`) failed to start. If this were an investigation of a system intrusion, that could be a critical piece of information.

Kernel and Audit Logs

The logging described so far has been generated by userspace programs, daemons, and applications. The Linux kernel also generates log information from kernel space, which can be useful in a forensic investigation. This section explains the purpose of kernel-generated messages, where they are located, and how to analyze them.

The Linux audit system is composed of many userspace tools and daemons to configure auditing, but the auditing and logging activity is performed from within the running kernel. This is the reason for including it here together with the kernel logging mechanism. Firewall logs are also produced by the kernel and would fit nicely in this section, but that topic is covered in Chapter 8 on the forensic analysis of Linux networking.

The Kernel Ring Buffer

The Linux kernel has a cyclic buffer that contains messages generated by the kernel and kernel modules. This buffer is a fixed size, and once it's full, it stays full and starts overwriting the oldest entries with any new entries, which means kernel logs are continuously lost as new messages are written. Userspace daemons are needed to capture and process events as they are produced. The kernel provides */dev/kmsg* and */proc/kmsg* for daemons like systemd-journald and rsyslogd to read new kernel messages as they are generated. These messages are then saved or forwarded depending on the log daemon's configuration.

The `dmesg` command is used on a running system to display the current contents of the ring buffer, but that isn't useful in a postmortem forensic examination. The ring buffer exists only in memory, but we can find traces of it in the logs written to the filesystem. During boot, the kernel begins saving messages to the ring buffer before any logging daemons are started. Once these daemons (systemd-journald, rsyslogd, and so on) start, they can read all the current kernel logs and begin to monitor for new ones.

It is common for syslog daemons to log kernel events to the */var/log/kern.log* file. Rotated versions of this log may include *kern.log.1*, *kern.log.2.gz*, and so on. The format is similar to other syslog files. For example, the saved kernel logs from a compressed rotated log from rsyslogd on a Raspberry Pi look like this:

```
$ zless /var/log/kern.log.2.gz
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] Booting Linux on physical CPU 0x0
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] Linux version 4.19.97-v7l+ (dom@buildbot) ...
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] CPU: ARMv7 Processor [410fd083] revision 3
(ARMv7), cr=30c5383d
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] CPU: div instructions available: patching
division code
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] CPU: PIPT / VIPT nonaliasing data cache,
PIPT instruction cache
Aug 12 06:17:04 raspberrypi kernel: [    0.000000] OF: fdt: Machine model: Raspberry Pi 4
Model B Rev 1.1
...
```

The rsyslogd daemon has a module called `imklog` that manages the logging of kernel events and is typically configured in the */etc/rsyslog.conf* file.

Systemd stores kernel logs in the journal with everything else. To view the kernel logs from a journal file, add the `-k` flag, as follows:

```
$ journalctl --file system.journal -k
-- Logs begin at Thu 2020-11-05 08:42:14 CET, end at Thu 2020-11-05 10:12:05 CET. --
Nov 05 08:42:14 pc1 kernel: microcode: microcode updated early to revision 0xd6, date =
 2020-04-27
Nov 05 08:42:14 pc1 kernel: Linux version 5.9.3-arch1-1 (linux@archlinux) (gcc (GCC)
 10.2.0, GNU ld (GNU Binutils) 2.35.1) #1 SMP PREEMPT Sun, 01 Nov 2020 12:58:59 +0000
Nov 05 08:42:14 pc1 kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-linux root=UUID=efbfc8dd
-8107-4833-9b95-5b11a1b96875 rw loglevel=3 quiet pcie_aspm=off i915.enable_dpcd_backlight=1
...
```

The */etc/systemd/journald.conf* has a parameter (`ReadKMsg=`) that enables processing of kernel messages from */dev/kmsg* (which is the default).

For a forensic examiner, kernel messages are important to help reconstruct the hardware components of a system at boot time and during system operation (until shutdown). During this period (identified by the boot-id), a record of attached, detached, and modified hardware devices (including manufacturer details) can be seen. In addition, information about various kernel subsystems such as networking, filesystems, virtual devices, and more can be found. Some examples of information that you can find in the kernel logs include:

- CPU features and microcode
- Kernel version and kernel command line
- Physical RAM and memory maps
- BIOS and mainboard details
- ACPI information
- Secure boot and TPM
- PCI bus and devices
- USB hubs and devices
- Ethernet interfaces and network protocols
- Storage devices (SATA, NVMe, and so on)
- Firewall logging (blocked or accepted packets)
- Audit logs
- Errors and security alerts

Let's look at some examples of kernel messages that are interesting in a forensic investigation or that may raise questions regarding the existence of the message.

In this example, information about a particular mainboard is provided:

```
Aug 16 12:19:20 localhost kernel: DMI: System manufacturer System Product
 Name/RAMPAGE IV BLACK EDITION, BIOS 0602 02/26/2014
```

Here, we can determine the mainboard is an ASUS Republic of Gamers model, and the current firmware (BIOS) version is shown. The mainboard model may provide some indication of system use (gamer rig, server, office PC, and so on). The firmware version may be of interest when examining security relevant vulnerabilities.

Newly attached hardware will generate kernel logs like the following:

```
Nov 08 15:16:07 pc1 kernel: usb 1-1: new full-speed USB device number 19 using xhci_hcd
Nov 08 15:16:08 pc1 kernel: usb 1-1: New USB device found, idVendor=1f6f, idProduct=0023,
 bcdDevice=67.59
Nov 08 15:16:08 pc1 kernel: usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
Nov 08 15:16:08 pc1 kernel: usb 1-1: Product: Jawbone
Nov 08 15:16:08 pc1 kernel: usb 1-1: Manufacturer: Aliph
Nov 08 15:16:08 pc1 kernel: usb 1-1: SerialNumber: Jawbone_00213C67C898
```

Here, an external speaker was plugged in to the system. This log information associates a specific piece of hardware with a machine at a specific point in time, and indicates that a person was in physical proximity to plug in the USB cable.

The following is an example kernel message about a network interface's mode:

```
Nov  2 22:29:57 pc1 kernel: [431744.148772] device enp8s0 entered promiscuous mode
Nov  2 22:33:27 pc1 kernel: [431953.449321] device enp8s0 left promiscuous mode
```

A network interface in *promiscuous mode* indicates that a packet sniffer is being used to capture traffic on a network subnet. An interface may enter promiscuous mode when a network administrator is troubleshooting problems or if a machine has been compromised and is sniffing for passwords or other information.

A kernel message about a network interface's online/offline status may look like this:

```
Jul 28 12:32:42 pc1 kernel: e1000e: enp0s31f6 NIC Link is Up 1000 Mbps Full Duplex,
 Flow Control: Rx/TX
Jul 28 13:12:01 pc1 kernel: e1000e: enp0s31f6 NIC Link is Down
```

Here, the kernel logs indicate that a network interface came online for nearly 50 minutes before going offline. If this were an intrusion or data theft investigation, observing an interface suddenly appearing could indicate an unused network port was involved. And if an unused physical Ethernet port was involved, it could mean that there was physical access to the server (which then means that you should check CCTV footage or server room access logs).

When analyzing the kernel logs, try to separate the boot logs from the operational logs. During boot, there will be hundreds of log lines in a short period that are all associated with the boot process. The kernel logs generated after booting is finished will indicate changes during the operational state of the machine until shutdown.

You can temporarily increase the verbosity of kernel logs during an ongoing investigation or attack to generate additional information. The kernel accepts parameters to specify increased (or reduced) logging in several areas. See *https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/kernel-parameters.txt* for more information about the kernel parameters (search for "log"). These parameters can be added to GRUB during system startup (see Chapter 6 for more information).

Individual kernel modules may also have verbose flags to increase logging. Use **modinfo** with the kernel module name to find possible debug options. Here is an example:

```
$ modinfo e1000e
filename:       /lib/modules/5.9.3-arch1-1/kernel/drivers/net/ethernet/intel/e1000e/e1000e.ko.xz
license:        GPL v2
description:    Intel(R) PRO/1000 Network Driver
...
parm:           debug:Debug level (0=none,...,16=all) (int)
...
```

In this example, Ethernet module e1000e has a `debug` option that can be set. The options for individual modules can be specified by placing a **.conf* file in the */etc/modprobe.d/* directory. See the modprobe.d(5) man page for more information.

The Linux Auditing System

The Linux Auditing System is described in the README file of the source code: "The Linux Audit subsystem provides a secure logging framework that is used to capture and record security relevant events." Linux auditing is a kernel feature that generates an audit trail based on a set of rules. It has similarities to other logging mechanisms, but it is more flexible, granular, and able to log file access and system calls. The `auditctl` program loads rules into the kernel, and the `auditd` daemon writes the audit records to disk. See the auditctl(8) and auditd(8) man pages for more information. Figure 5-4 shows the interaction between the various components.

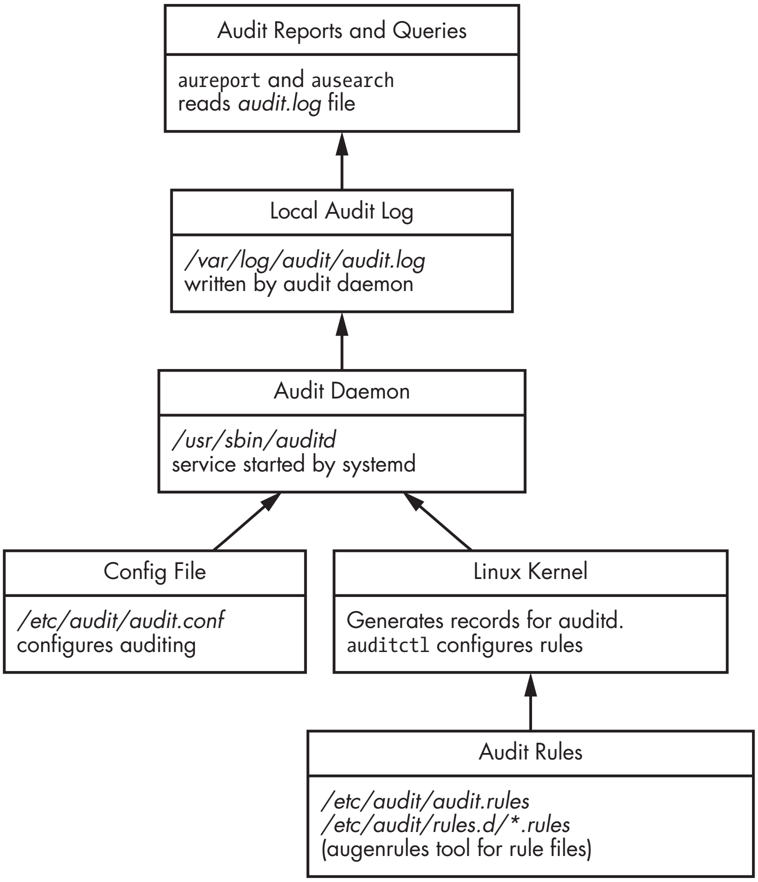
Audit Reports and Queries
aureport and ausearch reads audit.log file
Local Audit Log
/var/log/audit/audit.log written by audit daemon
Audit Daemon
/usr/sbin/auditd service started by systemd
Config File
/etc/audit/audit.conf configures auditing
Linux Kernel
Generates records for auditd. auditctl configures rules
Audit Rules
/etc/audit/audit.rules
/etc/audit/rules.d/*.rules
(augenrules tool for rule files)

Figure 5-4: Linux Auditing System

There are three kinds of audit rules:

- Control rules: overall control of the audit system
- File or "watch" rules: audit access to files and directories
- Syscall: audit system calls

Audit rules are loaded into the kernel at boot time or by a system administrator using the `auditctl` tool on a running system.[5] The audit rules are located in the */etc/audit/audit.rules* file. See the audit.rules(7) man page for more information about audit rules.

A collection of separate rule files located in */etc/audit/rules.d/*.rules* can be merged with the */etc/audit/audit.rules* file using the *augenrules* file. The audit rules file is simply a list of arguments that would be provided to `auditctl` commands.

5. This is similar to firewall rules that are loaded into the kernel with a userspace tool (`nft`).

Here are several examples of audit rule lines as seen in a rule file:

```
-D
-w /etc/ssl/private -p rwa
-a always,exit -S openat -F auid=1001
```

The first rule deletes all current rules, effectively creating a new rule set. The second rule watches all the files in the */etc/ssl/private/* directory (recursively). If any user or process reads, writes, or changes the attributes on any files (like SSL private keys), an audit record will be generated. The third rule monitors a specific user (UID 1001 specified with `auid=`) for all files opened. Presumably this user is at heightened risk of attack or under suspicion.

The default location of the audit log is */var/log/audit/audit.log* where `auditd` writes new audit records. This is a plaintext file with *FIELD* = *VALUE* pairs separated by spaces. The current list of field names is available at *https://github.com/linux-audit/audit-documentation/blob/master/specs/fields/field-dictionary.csv*. This file can be examined in its raw format, but the `ausearch` and `aureport` tools provide normalization, post-processing, and more readable output.

The *audit.log* file can be copied to a Linux analysis machine where `ausearch` and `aureport` can be used with the `--input` flag to specify the file.

An audit record format can be raw or enriched. Enriched records additionally resolve numbers to names and append them to the log line. An example enriched audit record from a */var/log/audit/audit.log* file looks like this:

```
type=USER_CMD msg=audit(1596484721.023:459): pid=12518 uid=1000 auid=1000 ses=3
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 msg='cwd="/home/sam"
cmd=73797374656D63746C20656E61626C652073736864 exe="/usr/bin/sudo" terminal=pts/0
res=success'^]UID="sam" AUID="sam"
```

The same audit record produced with the `ausearch` tool looks like:

```
$ ausearch --input audit.log
...
time->Mon Aug  3 21:58:41 2020
type=USER_CMD msg=audit(1596484721.023:459): pid=12518 uid=1000 auid=1000 ses=3
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 msg='cwd="/home/sam"
cmd=73797374656D63746C20656E61626C652073736864 exe="/usr/bin/sudo" terminal=pts/0
res=success'
...
```

This command produces a formatted output of the entire *audit.log* file. Here the date is converted from epoch format, and some control character formatting corrections are made.

You can specify `csv` or `text` for the output format. The `csv` format is useful for importing into other tools. The `text` format produces a single readable line for each audit record:

```
$ ausearch --input audit.log --format text
...
```

```
At 20:05:53 2020-11-08 system, acting as root, successfully started-service
man-db-cache-update using /usr/lib/systemd/systemd
At 20:05:53 2020-11-08 system, acting as root, successfully stopped-service
man-db-cache-update using /usr/lib/systemd/systemd
At 20:05:53 2020-11-08 system, acting as root, successfully stopped-service
run-r629edb1aa999451f942cef564a82319b using /usr/lib/systemd/systemd
At 20:07:02 2020-11-08 sam successfully was-authorized sam using /usr/bin/sudo
At 20:07:02 2020-11-08 sam successfully ran-command nmap 10.0.0.1 using /usr/bin/sudo
At 20:07:02 2020-11-08 sam, acting as root, successfully refreshed-credentials root
using /usr/bin/sudo
At 20:07:02 2020-11-08 sam, acting as root, successfully started-session /dev/pts/1
using /usr/bin/sudo
At 20:07:06 2020-11-08 sam, acting as root, successfully ended-session /dev/pts/1
```

See the ausearch(8) man page for other specific queries of the audit log.

To generate a report of statistics from an audit logfile, the `aureport` command can be used:

```
$ aureport --input audit.log

Summary Report
======================
Range of time in logs: 2020-08-03 13:08:48.433 - 2020-11-08 20:07:09.973
Selected time for report: 2020-08-03 13:08:48 - 2020-11-08 20:07:09.973
Number of changes in configuration: 306
Number of changes to accounts, groups, or roles: 4
Number of logins: 25
Number of failed logins: 2
Number of authentications: 48
Number of failed authentications: 52
Number of users: 5
Number of terminals: 11
Number of host names: 5
Number of executables: 11
Number of commands: 5
Number of files: 0
Number of AVC's: 0
Number of MAC events: 32
Number of failed syscalls: 0
Number of anomaly events: 5
Number of responses to anomaly events: 0
Number of crypto events: 211
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 136
Number of events: 22056
```

This summary may be useful for inclusion in a forensic report or to help guide where to look next in a forensic examination.

You can generate individual reports for each of these statistics. For example, the following generates a report on logins:

```
$ aureport --input audit.log --login

Login Report
============================================
# date time auid host term exe success event
============================================
1. 2020-08-03 14:08:59 1000 ? ? /usr/libexec/gdm-session-worker yes 294
2. 2020-08-03 21:55:21 1000 ? ? /usr/libexec/gdm-session-worker no 444
3. 2020-08-03 21:58:52 1000 10.0.11.1 /dev/pts/1 /usr/sbin/sshd yes 529
4. 2020-08-05 07:11:42 1000 10.0.11.1 /dev/pts/1 /usr/sbin/sshd yes 919
5. 2020-08-05 07:12:38 1000 10.0.11.1 /dev/pts/1 /usr/sbin/sshd yes 950
```

See the aureport(9) man page for the flags needed to generate other detailed reports about the other statistics.

The `aureport` and `ausearch` commands can also specify a time period. For example, this report is generated for the time period between 9 AM and 10 AM (but not including 10 AM) on November 8:

```
$ aureport --input audit.log --start 2020-11-08 09:00:00 --end 2020-11-08 09:59:59
```

Both `aureport` and `ausearch` use the same flags for the time range.

The `aureport` and `ausearch` commands have flags to interpret numeric entities and convert them to names. Do not do this. It will replace the numeric user IDs and group IDs with the matching names found on your own analysis machine, not from the suspect disk under analysis. The `ausearch` command also has a flag to resolve hostnames, which is not recommended when performing a forensic examination. This will potentially trigger a DNS network request, which could produce inaccurate results or otherwise compromise an investigation.

Summary

In this chapter, we have identified the locations of typical logs found on a Linux system. You have learned how to view these logs and the information they may contain. You have also seen examples of tools used to analyze logs in a forensic context. This chapter has provided the background on Linux logs that are referenced throughout the rest of the book.