logd (Android L)

Android L defines a new, much needed logging mechanism with its `logd` daemon. This daemon serves as a centralized user-mode logger, as opposed to the traditional Android's /dev/log/ files, implemented in kernel ring buffers. This not only addresses the main shortcomings of the ring buffers - their small size and resident memory requirements, but also allows `logd` to integrate with SELinux auditing, by registering itself as the `auditd`, which receives the SELinux messages from the kernel (via netlink), and records them in the system log.

Another important new feature provided by `logd` is **log pruning**, which allows the automatic clearing or retaining of log records from specific UID. This aims to solve the problem of logs being flooded with messages from overly-verbose processes, which make it harder to separate the wheat from the chaff. `logd` allows for white lists (UIDs or PIDs whose messages will be retained for longer) and ~blacklists (UIDs or PIDs whose messages will be quickly pruned), using the new `-P` switch of `logcat`.

The `logd` service is defined in /init.rc as follows:

Listing 4-7: The `logd` definition in /init.rc

```
service logd /system/bin/logd
    class core
    socket logd stream 0666 logd logd      # Used by CommandListener thread
    socket logdr seqpacket 0666 logd logd  # Used by LogReader thread
    socket logdw dgram 0222 logd logd      # Used by LogListener thread
    seclabel u:r:logd:s0
```

Note this service is designed with not one, but four sockets:

- **/dev/socket/logd:** The control interface socket.
- **/dev/socket/logdw:** A write-only socket (permissions 022 = `-w--w--w-`).
- **/dev/socket/logdr:** A read-write socket, designed for reading. Unlike the `logd` UN*X domain socket, this is a seqpacket (sequential packet) socket.
- **An unnamed NetLink socket:** Used when `logd` also provides `auditd` functionality for SELinux messages

The `logd` spawns listener threads over its sockets, as well as threads for clients (spawned on demand). The threads are individually named (using `prctl(2)`) so you can see them for yourself in `logd`'s /proc/$pid/task/ when `logd` is running.

As with the traditional logs, `logd` recognizes the log buffers of main, radio, events, and system, along with a new log - crash - added in L. These logs are identified by their "log ids" (lids), numbered 0 through 5, respectively.

System properties used by `logd`

The `logd` recognizes several system properties, all in the `logd` namespace, which toggle its behavior. Those are well documented in the README.property file in `logd`'s directory, shown here for convenience:

Listing 4-logdprops: Properties used by `logd`

```
name                        type default  description
logd.auditd                  bool  true   Enable selinux audit daemon
logd.auditd.dmesg            bool  true   selinux audit messages duplicated and
                                          sent on to dmesg log
logd.statistics.dgram_qlen  bool  false  Record dgram_qlen statistics. This
                                          represents a performance impact and
                                          is used to determine the platform's
                                          minimum domain socket network FIFO
                                          size (see source for details) based
                                          on typical load (logcat -S to view)
persist.logd.size           number 256K   default size of the buffer for all
                                          log ids at initial startup, at runtime
                                          use: logcat -b all -G

# persist.logd.size.logname can be used to set buffer sizes for individual logs
```

Controlling `logd`

Clients can connect to `/dev/socket/logd` to control logd with an array of protocol commands. Commonly, the client doing so is the `logcat` command, which has been modified to use the socket, rather than the legacy `ioctl(2)` codes over `/dev/log`. The commands are shown in Table 4-9:

Table 4-9: `logd` protocol commands

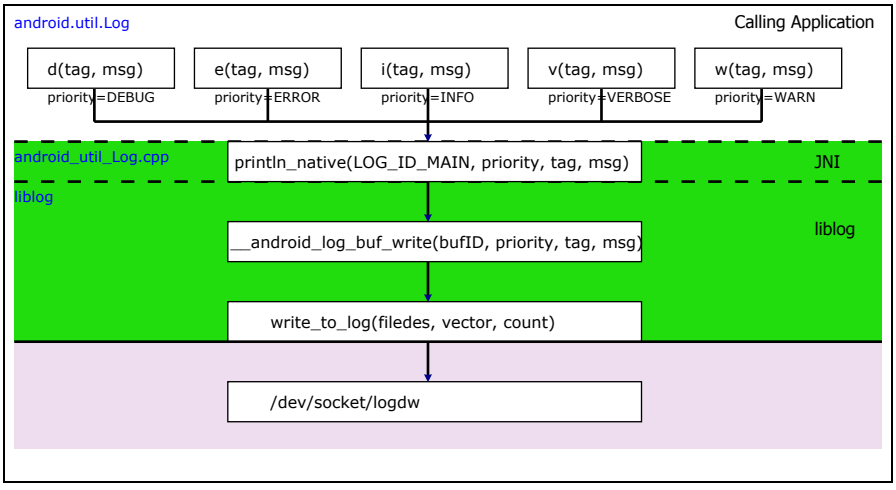
Command	logcat switch	Purpose
clear *lid*	-c	For callers with log credentials, this clears the specified log's buffers
getLogSize *lid*	-g	Get maximum size of log specified by *lid*
getLogSizeUsed *lid*		Get actual size of log specified by *lid*
setLogSize *lid*	-G	Set Maximum size of log specified by *lid*
getStatistics *lid*	-S	For callers with log credentials, this retrieves statistics - # of log messages by PID, etc.
getPruneList	-p	Get prune list (all logs)
setPruneList	-P	Set prune list (all logs)
shutdown		Force daemon exit. Surprisingly, this doesn't require any credentials.

The commands in gray require the caller to possess log credentials - be root, possess a primary GID of root, system, or log, or a secondary GID of log. To verify the last case the code of `logd` uses a crude method, of parsing the caller's `/proc/`*pid*`/status` and sifting through its "Groups:" line.

Writing to logd (logging)

Android's logging mechanism is supplied by `liblog`, and therefore applications remain oblivious to the underlying implementation of logging. As of L, both Bionic and `liblog` can be compiled to use `logd` (by `#define`ing `TARGET_USES_LOGD`), which then directs all the logging APIs to use `logd` rather than the traditional `/dev/log` files, which have, in effect, become legacy. Effectuating the change is a simple matter, since all system logging APIs eventually funnel to `liblog`'s `__android_log_buf_write` (or Bionic's `__libc_write_log`), which then open the `logdw` socket (instead of `/dev/log`), and write the log message to it. Figure 4-logag shows the flow of log messages from the application all the way to `logd`. A similar flow occurs for event log (android.util.EventLog) messages.

Figure 4-logag: The Android logger architecture



Reading from logd (logcat)

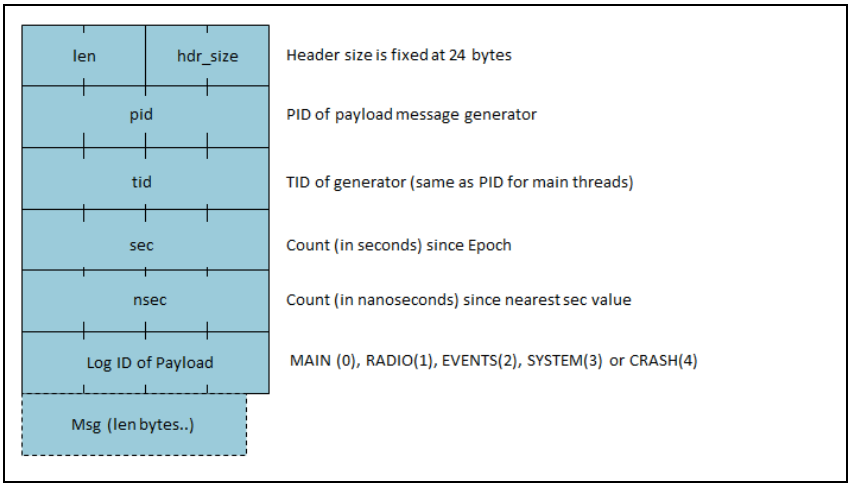
The familiar `logcat` command in L still sports the same command-line arguments it has in the past. Its underlying implementation, however, has rewritten to use `logd` through an updated `liblog` API. Clients such as logcat can connect to the logd reader socket (`/dev/socket/logdr`), and instruct the `LogReader` instance of `logd` to provide the log by writing parameters to it, as shown in the following table:

Table 4-logdr: Parameters recognized by `logd` over the reader socket

Parameter	Provides
lids=*value*	Log IDs
start=*value*	Start time from log to dump (default is EPOCH, start of log)
tail=*value*	Number of lines from log to dump (as per `tail(1)` command)
pid=*value*	Filter by PID originator of log messages
dumpAndClose	Tells reader thread to exit when log dumping is done

Log records are serialized into a `logger_entry_v3` structures before being passed to the reader over the socket. The structure format is shown in the following figure:

Figure 4-logdmsg: The format of a `logd` message



Putting all the above together, we can now observe `logd` in action, through the `logcat` command, as shown in the following experiment:

Experiment: Observing `logcat`

Using `strace` will allow you a behind-the-scenes look at the workings of `logcat` - including its connection to `logd`, the command it sends to dump the log, and the serialization of log messages:

Output 4-logcat: `logcat` under `strace`, annotated

```
# Tracing logcat during an adb logcat operation shows messages are received
# from file descriptor 3, and sent to file descriptor 1 (stdout)

root@generic:/# strace logcat
...
connect(3, {sa_family=AF_LOCAL, sun_path="/dev/socket/logdr"}, 20) = 0
write(3, "stream lids=0,3,4", 17)       = 17  # Dump main, system, crash
...
# \16 = 14 bytes (payload). \30 = 24 bytes (header). T\1 = 340 (PID) ... \3\0\0\0 = System
recvfrom(3, "<\16\30\0T\1\0\0m\1\0\0\275bbT$+\2374\3\0\0\0\6Act".., 5120, 0, NULL, 0) = 3668
write(1, "E/ActivityManager(  340): ANR in"..., 5472) = 5472
# In case you missed the connect(2) call above (e.g. if attaching to logcat), you can still
# look through its /proc/..fd entry, to see file descriptor 3 is a socket - which you can
# also deduce from the use of recvfrom(2):
root@generic:/# cd /proc/$LOGCAT_PID/fd
root@generic:/proc/337/fd # ls -l | grep "3 "
lrwx------ root     root              2014-11-11 14:24 3 -> socket:[2442]
# Looking through /proc/net/unix, which shows domain sockets, we can find the socket
and its remote endpoint (next inode number) - which happens to be logdr
root@generic:/proc/337/fd # grep 2442 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03  2442
root@generic:/proc/337/fd # grep 2443 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03  2443 /dev/socket/logdr
```

Sifting through `logd`'s thread to find and trace the `logd.reader.per` thread instance will show you the logging from the perspective of `logd`, and is left as an exercise for the reader.