546

AT891-0208 RMC
N. A. Camillone, D. H. Steves and K. C. Witte

IBM Technical Disclosure Bulletin Vol. 34 No. 7B December 1991

RELIABLE, EXTENSIBLE AUDIT TRAIL STORAGE IN AIX

Disclosed is a mechanism for providing audit trail storage. This mechanism provides for reliable and recoverable storage of audit records, and the mechanism itself is extensible to support different modes of auditing.

Audit subsystems in operating systems must provide some means of storing audit records for later analysis. The simplest solution is to simply direct output into a named file, as the accounting subsystem does with accounting records. This approach implies a certain amount of inefficiency, since audit trails tend to get quite large and so later records will be stored in the triple-indirect blocks of the file. In addition, it is difficult to recover from storage or file system failures, since no alternate storage media is normally specified. This approach also does not allow for easy post-processing of the audit records. As a result of these shortcomings, audit subsystems normally write records in fixed or limited size bins.

The disclosed mechanism provides bin-based audit trail storage in the following manner. At system initialization time, the auditbin daemon is started. The first step performed by this daemon is to start audit bin logging in the kernel. It does this by invoking the auditbin() system call. This call allows its invoker to define two audit bins, the current bin and the next bin. What is unique here is that these are defined by descriptor, not by name. This allows for different forms of storage options besides filesystem-based objects like files and named pipes. Also worth noting is that only two bins are in use at any one time. This allows for simpler recovery processing. As part of the auditbin() call, the auditbin can specify a maximum bin size. A key part of the bin auditing design is that the daemon will usually wait at this point instead of returning from the call.

After bin auditing is enabled, system auditing is turned on and the system will begin writing records into the current bin. When the bin reaches the maximum size specified by the auditbin daemon, the system will switch to the next bin and the auditbin daemon is awoken and will return from the call. At this point, the auditbin daemon will invoke post-processing filters on the full bin. These filters can be used to compress the bins or do post-selection of specific events or other tasks. The filters that are used for post-processing are fully configurable by the system administrator.

After the filters complete their task (which usually involves appending the bin to an audit trail file), the auditbin daemon invokes the auditbin system call once again, but this time only supplies a next bin, since the system already has a current bin. If the current bin is full by the time the auditbin daemon 'returns', the system will switch bins at that point. Otherwise, the auditbin daemon will wait again for a full bin. This mode of processing will continue until either the auditing system is shut down or there is an error.

A key feature of this mode of bin processing is that crash recovery is considerably simplified. There are only two bins in existence at any time. One of these will be the current bin and the other bin is either the next bin (and thus empty) or is being processed by the post-processing filters. All bins contain time-stamped header and trailer records. When the system is initialized the auditbin daemon runs recovery if it finds that either bin is partial or full. Because of the way in which bins are filled, the only possible states are:

- 0 partial, 0 full
  the system terminated normally and no recovery is necessary.
- 1 partial, 0 full
  the system crashed before the current bin was full and after the last bin was completely processed.
- 1 partial, 1 full
  the system crashed before the last bin was completely processed.
- 0 partial, 1 full
  the system crashed after the bins were switched but before any records were written into the current bin and before the last bin was completely processed.

If there is a full bin, it will be recovered first since it must be older than any partial bin. Recovery consists of rerunning the post-processing filters which are designed to be idempotent. Then if there are any partial bins, these are recovered by appending a bin trailer, which indicates that the bin was terminated with an error. This flag is used by the filters to signal the possible presence of incomplete records. The partial bin is then 'fed' to the post-processing filters as well. This completes the recovery.

Equally important is the reliability of this system. Because the auditbin daemon is so closely associated with the kernel audit logger, any errors will be reported almost immediately to the daemon. In addition, the alternating bin strategy means that the audit logger always has an extra descriptor to which to append records in the case that a write to the descriptor for the current bin fails.

The two descriptor mechanism also provides for great flexibility in configuring the audit trail. The scheme described above represents just one option and is a function of the auditbin daemon rather than the underlying audit subsystem. It is, in fact, possible to replace the whole auditbin daemon with other daemons which provide entirely different audit trail options. As one example, consider a network audit subsystem where the audit trail is collected and stored on one system (the audit server) in the network and the other systems just send audit records to that system. This could be done by having the auditbin daemon set up a connection to the audit server and then use the socket descriptor as the current descriptor for the auditbin() call. The size of the audit bins should be specified as 0 (implying infinite). The system audit logger will then append records to the socket until an error is returned or until the auditing subsystem is shutdown. As an option to aid fault tolerance, the second file descriptor can be used to specify a local file, so that if an error occurs, audit record will be written into the local file while the auditbin daemon tries to recover from the fault.

As another example, consider a system where only a simple audit trail file is required, like the accounting file. The auditbin daemon would open this file and pass its file descriptor as the current descriptor. The bin size would be set to 0, as in the above example. In this case, the auditbin daemon need not even wait, since as in the accounting subsystem, no recovery or fault-tolerance capabilities are necessary.

The disclosed mechanism provides a recoverable, reliable and extensible bin auditing capability, as described above, in an extremely efficient fashion. To implement this mechanism required only one system call, one stanza in a configuration file and one command.