542

AT891-0206 RMC
D. H. Steves and K. C. Witte

IBM® Technical Disclosure Bulletin Vol. 34 No. 8 January 1992

92A060059

AUDITING OBJECTS BY NAME IN AIX

Disclosed is a mechanism for auditing operating system objects by name in a UNIX*-based operating system, such as AIX**. This mechanism provides either greater specificity or more efficiency than other existing mechanisms, and also provides the non-circumventability property of a reference monitor.

Auditing is one of the two basic mechanisms used to implement security in operating systems. Auditing provides detective security based on user accountability. All auditing systems define the event types which may be recorded in the audit trail on a per-user basis. The granularity with which event types are defined is crucial to the effectiveness of an auditing system, since this definition determines for which actions a user may be held accountable.

The following types of actions are generally considered to be auditable:

- use of the system authentication mechanism
- access to objects
- administrative actions
- production of printed material
- security-relevant events, such as failed log-ins and object accesses

Of these, the second is the most common type of event and the most important, since it is directly related to the principal function of most computer systems - that is, to store and transform information. But auditing access to objects is also the most difficult in a UNIX-based system, due to two factors. First, file accesses (files are the predominant type of object in a UNIX system) - normally read and write - are made via file descriptors and not by name. Processes can perform several operations on file descriptors, including bequeathing them to a child process or sending them to an unrelated process, and these operations can be used to mask the real 'perpetrator' of an access. Second, auditing characteristics are not file system information and so could not be stored in the file itself. Instead, it would be considered as meta-information and this information is stored in inodes separate from the actual file, and these inodes exist only when the file itself exists. If the file is transitory or if the file is updated by replacement and not in place, then the inode used for a file will constantly change and the auditing characteristics would be lost.

There are several methods that have been used in the past to audit objects. The most common method is to require that all file-system events be recorded in the audit trail, and this information is then pieced together afterwards to determine object accesses. This method will work, but it is extremely inefficient, both in writing the audit trail and in processing the audit trail. This method requires that every file creation, file open, file read, file write, file descriptor control and file close operation be audited. In addition, each process creation must be audited. Moreover, in a system which supports sockets for inter-process communication, it is possible to transfer descriptors directly from one process to another, and so all socket creation and descriptor transfer operations must also be audited. Since most files on a system are not interesting, this is extremely inefficient to the point of being ineffective. Further, the costs of processing this information to perform the actual auditing are also quite high, since it is possible that each file would have to be tracked across several processes. Lastly, note that this is not extensible to objects which are not in the filesystem name space.

A second method is to put the auditing characteristics for a file in the inode for that file, but, as was noted above, this information can be lost if an application updates the file by replacement, which is quite common in UNIX applications, and cannot be used for nonpersistent objects. This method also is not extensible to objects that are not in the filesystem name space.

A third method is to store the auditing characteristics in the directory entry for the file. This method works for both non-persistent files and for files which are updated by replacement, but it requires substantial modifications to directories and the system utilities which access them, since directory entries can exist for nonexistent files. Another shortcoming is that the parent object must exist for the file to be audited. Also, this method is not extensible to non-filesystem objects.

The last method for auditing objects by name is to do the auditing on a per-directory basis. Each access to a file in an audited directory will generate an audit event. This method has the same advantages and disadvantages of the third method, but, in addition, suffers from a lack of granularity. Directories are not generally considered units of auditing, and so files in a directory will not, in general, be in the same auditing classes. For instance, the /etc directory contains the passwd file, which is used to store security information and it contains the filesystems file, which is used to store information about mounted filesystems.

The disclosed mechanism has been designed to address the above issues, and works as follows. The system auditor defines all objects in the system which are to be audited by name, giving the object name and type and the events which are to be generated for each type of access. During system initialization, this information is used by the system audit event logger in the kernel to build hash tables for each directory which contains objects to be audited. These hash tables define name to event mappings. During pathname lookup for each directory component, these hash lists are used to determine if the object is to be audited. If so, a structure in the vnode created for that object is set to the events for that name. When the object is accessed and there is an event defined for that mode of access, the audit logger is notified and an audit record is generated for that event if it is in one of the event classes to be audited for the current process. The logging of events is noted in the vnode record, so that duplicate accesses are not logged.

This mechanism addresses each of the problems for the other methods described above. The mechanism is highly specific, since it not only allows auditing of objects by name, but even allows distinguishing between different access modes. And since the auditing characteristics are not stored in filesystem structures, this mechanism will correctly audit accesses to non-persistent files and files which are updated with replacement. Last, the mechanism is easily extensible to different types of objects because of the method of storing the auditing characteristics and because each object to be audited is typed and this typing can be used to define new name spaces.

More importantly, this mechanism can be implemented efficiently as well. For directories with no audited objects, there is no overhead in the lookup of the pathname component. For directories with audited objects, the overhead will be proportional to the number of audited objects and, since the names of audited objects are maintained in a hashed data structure, this is likely to be far less than the overhead of the component lookup, which is done with sequential search.

* Trademark of UNIX System Laboratories, Inc.
** Trademark of IBM Corp.