The FreeBSD Audit System

Robert N. M. Watson
University of Cambridge
TrustedBSD Project
rwatson@FreeBSD.org

Wayne Salamon
TrustedBSD Project
wsalamon@FreeBSD.org

Abstract

This paper describes the Common Criteria security event auditing implementation added to the FreeBSD operating system by the TrustedBSD Project. Audit is a critical element in operating system security evaluation and operation, but both the standards-based and operational requirements are complex. This paper describes the requirements, FreeBSD kernel implementation, extensible file format adopted from OpenSolaris BSM, mechanisms used for processing and maintaining the audit trail, and the OpenBSM audit library and tool set. Of importance is not just the content of audit records, but also the reliability guarantees associated with the queuing and delivery mechanisms.

1 Introduction

This paper describes the security audit system developed by the TrustedBSD Project for the FreeBSD operating system [12]. Security event auditing refers to the capability to perform detailed and reliable logging of security events, and is a necessary feature for the deployment of systems in many security-sensitive environments.

We describe the operational and standards requirements for auditing, the design and implementation of data gathering and record management, and an audit record format. The audit implementation involves modifications to both the kernel and user space code in FreeBSD, and involve collecting and managing kernel and application audit data. Audit is present in many commercial UNIX operating systems as it is required for Common Criteria evaluation. However, there is a significant gap relating to the design choices systems literature, which we hope to help fill via a worked example.

Portions of the design and implementation of audit on the FreeBSD operating system platform are derived from the audit implementation developed by the authors for inclusion in Apple Computer's Mac OS X system [6]. The Darwin implementation released by Apple under the BSD license has become the foundation for the FreeBSD implementation [1]. The design of both the FreeBSD and Darwin audit implementations has been strongly influenced by the published Solaris BSM audit API and audit trail format, de facto industry standards that are used by a large number of existing applications.

2 Requirements for Event Auditing

A modern, trusted operating system contains many components that are used to establish the basis of trust. One of these components is audit, a facility that allows the administrator to trace a broad range of security-relevant events to the authenticated user that initiated that event. In the operating system domain, the user is represented by a process executing some task on behalf of the user. The requirement that the event be traced to an authenticated user is important, as it means that normal UNIX process credential uids are not sufficient: they may change as part of setuid applications that continue to act on behalf of the original user.

An important design choice for an audit subsystem is the selection of events that must be audited. The CAPP protection profile requires that the following classes of events be supported [7]:

Controlled Operations Access to controlled files, network services, and other objects. In practice, any access that involves an access control decision, such as a privilege check, permission check, or ownership check.

Authentication Authentication success and failure, use of security services, etc. These are typically user space constructs in UNIX.

Security Management Change of user credentials, audit controls, etc, which consist largely of auditing changes to configuration files in `/etc` in UNIX.

Common applications of security audit include post-mortem analysis, intrusion detection systems, and general system operation monitoring. A key requirement, both from standards and a practical perspective, is that while the audit system may be able to capture very detailed information, that it can be configured not to do so. Typical dimensions of interest are the identity of the authenticated user, object properties such as owner, type, and method, and the nature and results of any access control decisions. By varying the granularity of the audited events, the administrator can control the amount of information that is contained in the audit logs, as well as monitor the system for specific types of events, such as attempted administrative access. Pre-selection occurs when a decision is made about whether to log events

while the event is occurring; post-selection occurs during review or reduction of existing audit trails.

The audit system records information about accesses to controlled objects by process subjects. Kernel events are systems calls and mostly correspond to object accesses that are mediated by discretionary access controls. However, many other types of accesses are audited, such as use of root privilege, system shutdown, and changes to the audit system configuration itself.

The kernel "owns" the audit log, and is the only writer, providing for a non-bypassable audit trail. However, many security-relevant events are implemented in applications. In contrast to the existing syslog facility [5], untrusted processes are not permitted to write directly to the audit trail. These programs must therefore submit audit records to the kernel for inclusion in the audit log. UNIX discretionary access controls are typically used to protect both the audit trail and configuration.

A key reliability concept is the correspondence between auditable events and generation of audit records. Standards require a strong mapping: if an event is auditable, and the event is to be allowed to occur, it must be audited. This presents a significant reliability challenge—in the event of an exhaustion of space, should the event be permitted to occur? Two undesirable choices are available: the system as a whole may halt due to store exhaustion, or that events may occur without proper auditing. Standards dictate that the systems must be able to fail stop in the presence of an audit failure, but that this may be configurable behavior. When an external failure occurs, such as power loss, the audit system must also be able to place a defined upper bound on record loss, typically implemented via a fixed bound on the number of records queued for asynchronous storage.

2.1 Common Criteria and Controlled Access Protection Profile

The Controlled Access Protection Profile (CAPP) [7] is part of the Common Criteria (CC) [2] suite of information assurance documents. The goal of CC is to establish a guide for developing and evaluating an information technology product in regards to its security features. Common Criteria deals with the implementation of the product and not the administration. The CC addresses the confidentiality, integrity, and availability of information maintained within the system, and concentrates primarily on threads from human activities. Auditing addresses critical traceability requirements in systems evaluated under the CC.

A protection profile is used to lay out a set of security requirements that a system should fulfill. There are many protection profiles defined under the CC framework; CAPP deals specifically with the discretionary access controls that are used to mediate access to data by users, roughly corresponding with the C2 class of the Trusted Computer System Evaluation Criteria (TCSEC). CAPP requires an audit capability with certain features, such as auditing of subject and object interactions, login/logout events, and administrative control of auditing, and many more. The implementation of auditing on FreeBSD meets all of these requirements, although no formal evaluation has taken place at the time of writing.

3 Structure of an Audit Implementation

The FreeBSD audit implementation consists principally of the following components:

sys/security/audit Reliable kernel audit record queue, system call auditing.

contrib/openbsm BSM API library, documentation, and audit-related utilities.

etc/security Configuration files.

usr.sbin/auditd Audit management daemon.

As audit support must be non-bypassable and reliable, audit record queuing and audit event capture for the majority of auditable events occurs in the kernel. In addition to audit records generated in the kernel, trusted user applications may also submit records to the kernel using the `audit()` system call. Review and management of the audit trail is performed using user space applications, subject to the permissions associated with the audit trail and configuration files.

3.1 The BSM Audit Trail Format

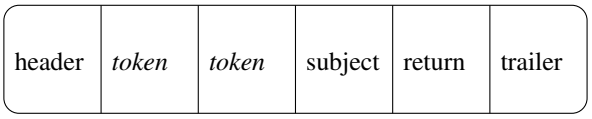

Figure 1: Audit Record Format

The format chosen for the audit trail is defined by Sun Microsystems as part of the Basic Security Module (BSM) [11]. The rationale for choosing this format is that it is a de facto audit trail standard, permitting the use of existing processing and management tools. Furthermore, the BSM record format defined for this project is designed to be independent of processor architecture, and may be easily extended to support features present in FreeBSD that are not present in Solaris.

A BSM audit log consists of one or more audit records. Each record is composed of a series of tokens representing data elements, with every record containing header, subject, return, and trailer tokens. As type and length information is included with each record, parsers are able to skip records they do not recognize, resulting in limited forward compatibility as new record and to-

ken types are added. The audit record format is shown in Figure 1.

The header token contains general information about the audit event, including the total record length, event type, and a timestamp. The subject token contains user IDs, PID, and other information about the process. The return token contains the system call return value as well as the success/failure indication. The trailer token closes out the audit record and stores the entire length of the audit record.

There are many other token types used in an audit record, and each type is based on the nature of the object represented. For example, a file object is represented by two token types: the `path` token, and the `attribute` token, containing ownership and other information about the file object. Figure 2 shows an example of a complete audit record.

3.2 Audit Events and Classes

In the FreeBSD kernel, audit events are associated with system calls. The selection of which system calls are audited is based on several factors. If the call is used to access a protected object, super-user privilege is required, or the audit configuration is changed, then an audit record may be created. In practice, the majority of system calls fall into the above categories, by virtue of traversing a file system path (requiring access control checks), or interacting with other processes.

Event classes permit the configuration of audit preselection based on broad categories of related audit events. Audit events are assigned to zero or more classes; example classes include *process*, *network*, as well as several classes associated with file access. The kernel maintains an internal table that maps events to classes, and this table can be changed via the `auditon()` system call. The initial event to class mapping is stored in an audit configuration file that is loaded into the kernel when auditing is started.

Each user name can have an associated set of event classes to audit. There are two audit class masks associated with the user's processes: a mask selecting successful events, and a mask selecting unsuccessful events. Using the audit event masks, the administrator can finely control the auditing of events for each user. These masks are stored as part of the process control block and are set when the user logs in.

4 FreeBSD Changes

McAfee Research, under contract to Apple, Inc., added audit support to the Darwin 7.x kernel, which is the foundation for the Mac OS X 10.3 release. All of the source code is released under a BSD open source license, including the kernel event auditing, user space programs, and modifications to existing programs such as `login`. The Darwin source code forms the basis for FreeBSD audit, although many changes have been made after the initial merge. Some of the changes have been submitted back to the Darwin maintainers.

Two parallel projects were created under the TrustedBSD project to manage the audit source code for FreeBSD. One project manages the core audit functionality in the kernel and user space programs. The other project, OpenBSM [8], manages the common audit definitions, and libraries used to read and write audit records.

This section describes the source code used within the kernel. The tools and applications used for auditing are described in Section 5. The OpenBSM project and related code is discussed in Section 6.

4.1 Kernel Audit Processing

The changes to the FreeBSD kernel can be grouped into three areas: auditing of system calls; managing audit records internally; and the commitment of audit records to the file system along with user space notifications.

System calls form the central point where events in the kernel that require auditing take place. The reason is that all controlled objects are accessed via system calls, with the appropriate access controls invoked. The audit record that is created for these events will include information about the process (subject) credentials and the object traits (ownership, etc.). Audit event identifiers are assigned to system calls in system call tables; several system calls may share the same event identifier if the underlying service is the same. For native system calls, the assignment occurs in `syscalls.master`; for ABI emulation, the assignment occurs in the system call table for the ABI implementation.

Figure 3 shows the generation of the audit record within the kernel. On system call entry (`audit_syscall_enter()`), the process's audit masks, or the default audit mask, are used to decide whether an audit record is to be created. This is an opportunity for the audit implementation to suspend execution of the thread before an auditable action can be performed if the queue depth has reached its limit, or if insufficient resources are available to successfully commit the audit record to disk. If selected, then the kernel form of the audit record is added to the thread, and subject information is captured.

Once the system call is entered, the parameters and object information are captured and stored in the kernel audit record. In most cases, the object information is captured near the beginning of the system call. For path name lookups, however, the object information (the vnode) is captured in the centralized lookup code (`namei()`, `lookup()`), where paths are copied into the kernel.

```
header,188,1,open(2) - read,write,creat,0,Wed Oct 19 19:50:51 2005, + 290 msec
argument,3,0x180,mode
argument,2,0xa02,flags
path,/usr/home/wsalamon/audit3/tools/regression/audit/test/file/temp2.duFV
subject,666,root,wheel,root,wheel,500,777,99,0.0.0.66
return,success,23
trailer,188
```

Figure 2: Example Audit Record

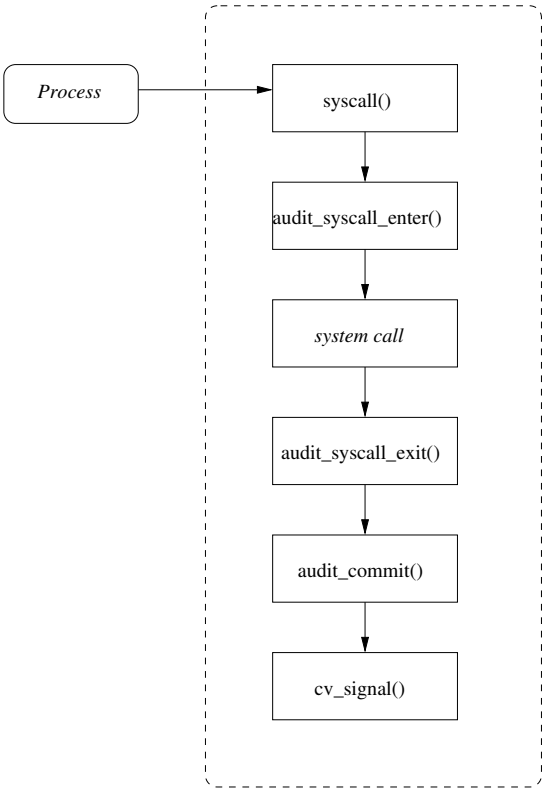

Figure 3: Audit Record Creation Kernel Flow

At system call exit (`audit_syscall_exit()`), further pre-selection is applied on the return value, and the record is either committed to the record queue or abandoned. The `audit_worker` kernel thread then is signaled to begin asynchronously processing the record. For the `audit()` system call, two audit records are committed: an audit record describing the system call itself, and the user-provided audit record.

The internal queue of audit records is processed by a kernel thread `audit_worker`, which is represented in Figure 4.

Records are removed from the queue by `audit_record_write()`, which checks the file system for sufficient free space, and for the need for log rotation. If any limit is reached, a trigger is sent to

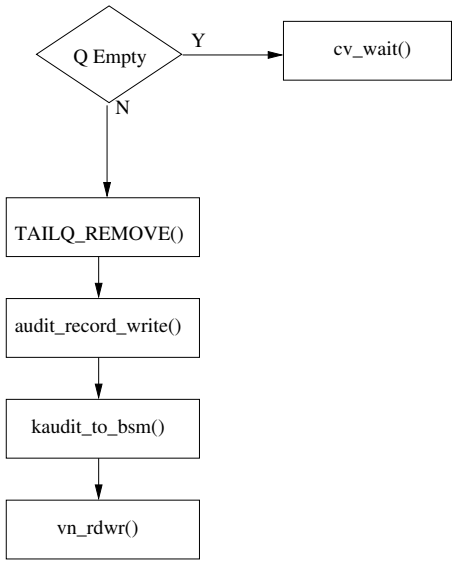

Figure 4: Audit Worker Kernel Task

`auditd` so the audit trail can be managed. If necessary, audit will be suspended, and if the record cannot be written and the kernel is so configured, the kernel will panic. Prior to panicking the system, all pending audit records in the event queue will be flushed to disk, allowing the system to fail stop when no further auditing is possible. While more sophisticated recovery behavior is possible, most possible recovery activities employ privileged operations that cannot be permitted to occur without auditing; unless specifically configured, the default is to drop audit records when space is exhausted.

If the file checks succeed, `kaudit_to_bsm()` converts the audit data from the internal kernel data structure to a BSM audit record that can be written to the audit trail. During the conversion process, BSM tokens are created describing the event, including identifying the event type and a set of event-related tokens describing any objects and additional arguments. Finally, a subject token describing the process, return code token, and trailer token are generated. The example record in Figure 2 shows two argument tokens and a path token representing the file object, along with the header, subject,

return, and trailer tokens that are added to every record. After token generation, the complete record is written to the file system with `vn_rdwr()`.

4.2 Audit Related System Calls

The `auditon()` system call controls the kernel audit configuration, such as enabling/disabling, setting of global and process audit masks, creating audit event to class mappings, and setting the maximum audit log file size. This call is also used to set the policy for what action to be taken when audit fails: drop the record and continue, or panic.

Several other system calls were added to get and set audit masks and other information for the current process. The `auditctl()` system call is used to set the audit log file name. Finally, the `audit()` system call is used to submit audit records for inclusion in the log.

4.3 Kernel to User Space Communication

Communication from the kernel to the audit daemon is done via the `/dev/audit` special file, while communication from the daemon to kernel is done via system calls. The special file is not writable, and is only used to send triggers from the kernel to the daemon, such as the log rotation trigger. When an application wants to send a trigger to the daemon (audit shutdown, for example), the trigger is routed through the kernel: the application uses the `auditon()` system call, and the kernel sends the trigger.

As the kernel is the only writer to the log file, it is responsible for monitoring the state of the file and informing the audit daemon when action must be taken. Several triggers are defined to send a status to the audit daemon:

OPEN NEW The current audit log is full. The maximum size of the audit log is a configurable parameter. This limit is *soft* and auditing continues.

LOW SPACE Free space on the log's file system is low. The amount of minimum free space is configurable. This limit is *soft* and auditing continues.

NO SPACE Free space on the log's file system is exhausted. This limit is *hard*: either a panic occurs, or audit is suspended.

5 User Space Components

In order to fully support the audit requirements, several application programs are part of the audit system in FreeBSD. The core application is the audit daemon, responsible for audit configuration management and audit trail files. Also included are audit reduction tools, and modifications to a number of management tools and security components to submit necessary audit records.

5.1 The Audit Daemon

The audit daemon, `auditd`, runs early in the boot process, prior to any user logins. The primary duty of the audit daemon is to manage the audit log files, but is also responsible for several other aspects of the audit system, including startup and shutdown. Audit records are submitted by the daemon for each of the startup, shutdown, and rotation events, as mandated by CAPP. In addition to the triggers mentioned in Section 4.3, other triggers accepted by the daemon are:

READ FILE The audit daemon will read the configuration files and apply any changes to the audit configuration.

CLOSE AND DIE Close the audit log file, disable auditing, and exit.

5.1.1 Audit Startup and Shutdown

The audit daemon is responsible for audit startup and shutdown. During startup, the daemon loads the event to class mapping into the kernel. Next, the daemon sends the name of the audit log file to the kernel using the `auditctl()` system call. This call will also cause auditing to be started.

During audit shutdown, the daemon informs the kernel that auditing is to be disabled, and renames the current log file to contain the end timestamp in the name.

5.1.2 Audit Log Management

When the audit daemon rotates the log file, the new file name is passed to the kernel, and the previous file is renamed. The naming convention uses start and end timestamps in order to easily recognize the period of auditing that each file captures.

The audit daemon is responsible for managing the on-disk log files, but does not monitor the status of the current log file, which is the responsibility of the kernel, as discussed in Section 4.3.

5.2 Audit Control Files

Several text files are used to configure the audit system, and are stored in the `/etc/security` directory. These files are consumed by the audit applications and BSM libraries, but not directly by the kernel: the audit daemon will push configuration information into the kernel using the `auditon()` system call. The configuration files are:

audit_class Classes of audit events.

audit_control Settings for the audit log directories, default audit masks, and minimum free space for the file system that holds the logs.

audit_event Audit events and class assignments.

audit_user Audit masks for specific users.

audit_warn Script that is invoked at a user exit from the audit daemon. Typically, this script logs the warning, but could take other actions such as send an email to the administrator.

The audit configuration files are sensitive, and can be modified only by the administrator. The `audit_user` and `audit_control` files are not be readable by normal users in order to meet the CAPP requirement of preventing users from knowing which events are audited. Audit configuration information may only be queried from the kernel by privileged processes.

5.3 Audit Trail Tools

The audit logs can be examined by using the tools that were ported from the Darwin source code:

auditreduce Selects records from the audit log based on user ID, date, event, or other criteria.

praudit Presents audit records in human-readable form.

Using these tools, administrators may perform post-selection to identify audit records of interest, and convert them to text; this implementation does not yet support the OpenSolaris XML output format.

6 The OpenBSM Project

OpenBSM is a bundling of the user space components of the TrustedBSD audit implementation, including:

- bsm kernel and user space header files
- libbsm, an implementation of the BSM API with some extensions
- documentation of the BSM file format and API
- sample /etc/security configuration files
- praudit and auditreduce command line tools
- testing components

Available under a BSD license, OpenBSM is intended to provide a portable foundation for audit implementations across multiple platforms, including FreeBSD and Darwin. Primary changes from the Apple BSM implementation are significant cleanups and enhancements, including transition to an endian-independent file format, extensive documentation, and support for 64-bit BSM records present in more recent Solaris releases.

7 Future Directions

There are several areas within the audit system and externally that will be enhanced in future releases. One area is better integration of the Mandatory Access Control system with auditing. Also, a review of the audit coverage for all system calls is necessary to ensure completeness.

Testing and performance analysis is also required; a suite of test programs was ported from the Darwin code base for use with FreeBSD, and that suite continues to grow.

Several applications still require audit support, such as the `ssh` daemon and others that generate security-relevant events. The audit daemon itself needs to be tested for fault tolerance and the proper handling of audit log rotation in extreme conditions.

8 Conclusion

This paper has described an audit system for the FreeBSD operating system, characterized by adherence to standards, including CAPP, and implementation of portable APIs and record formats. The implementation is available under a BSD license as part of the TrustedBSD Project.

Acknowledgment

The authors would like to thank Kevin Van Vechten and Ron Dumont of Apple Computer, Inc. for support of the OpenBSM project, and gratefully acknowledge the contribution of the Apple BSM implementation under a BSD license. We also thank Tom Rhodes of the TrustedBSD project for contributing documentation, and other members of the TrustedBSD development community for the submission of bug reports.

References

[1] *Apple Developer Connection: Darwin*, `http://developer.apple.com/darwin/`.

[2] *The Common Criteria Evaluation and Validation Scheme*, `http://niap.nist.gov/cc-scheme/pp`.

[3] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley, 2005.

[4] *FreeBSD*, `http://www.freebsd.org`.

[5] *FreeBSD System Log API*, `http://www.freebsd.org/cgi/man.cgi?query=syslog&format=html`.

[6] *Apple Mac OS X* `http://www.apple.com/macosx/`.

[7] National Security Agency/Information Systems Security Organization, *Controlled Access Protection Profile*, available at `http://niap.nist.gov/cc-scheme/pp/pp/PP_CAPP_V1.d.pdf`.

[8] *OpenBSM*, `http://www.openbsm.org`.

[9] *OpenDarwin*, `http://www.opendarwin.org`.

[10] *OpenSolaris*, `http://www.opensolaris.org/`.

[11] Sun Microsystems, Inc. *System Administration Guide: Security Services* Part No: 816-4557, Sun Microsystems, 2005. Available at `http://docs.sun.com/app/docs/prod/solaris.10`

[12] *TrustedBSD*, `http://www.trustedbsd.org`.