A Model of Security Monitoring

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

Abstract

We present a formal model of security monitoring that distinguishes two different methods of recording information (logging) and two different methods of analyzing information (auditing). From this model we draw implications for the design and use of security monitoring mechanisms. We then apply the model to security mechanisms for statistical databases, monitoring mechanisms for computer systems, and backups, to demonstrate the model's usefulness.

Introduction

Although often used interchangeably, *auditing* and *logging* describe two distinct actions; logging is simply making a record, and auditing is analyzing that record [8]. Computer systems use logging to provide information used to restore file systems and databases to consistent states after crashes [9,10,14,16]; they also require logging for security purposes, for example in computer systems ranging from those meeting the Department of Defense trusted computer guidelines for class C2 or higher [17], in electronic fund transfer systems [12], and other types of data processing [2]. The computers log information relevant to the security of the system; in most cases, they audit this log (often called an *audit trail* or *activity log*) and take action consistent with the state of the system as recorded in the log. This audit has three steps. First, information in the log is *reduced* to eliminate data. The remaining information is *analyzed* either to format the data in the log or to determine whether or not a compromise has occurred or would occur if a specific action were taken; and finally, the audit mechanism *notifies* a program or an auditor of the results.

The entry and exit of people from a secured building provides an analogy. At the door is a security guard who signs people in and out. This record is a log, and the guard is performing the logging function. At the end of the day, the main office obtains the logs. To determine if there is a potential security problem, a clerk first eliminates all names showing both entry and exit; this is the reduction step. The clerk then determines if the people still in the building are authorized to be there at night; this is the analysis step. If someone who should not be there is still in the building, the clerk calls the security office and directs them to locate and to escort the person out of the building; this is the notification step.

To show that the terms "logging" and "auditing" apply to diverse situations, consider the same building with a new rule: employees must escort visitors within the building. When a visitor arrives, the guard records personal information and who the visitor is to see (the logging step.) The guard then determines how to contact that employee, which may entail calls to numerous people (the reduction step). When the guard reaches that person, the guard asks whether the employee will escort the visitor (the analysis step). Finally, the guard informs the visitor of the result (the notification step.) While these are not the conventional uses of the words "logging" and "auditing," they certainly fall under the purview of the definitions above.

We should at this point distinguish our notion of "logging" and "auditing" from the orthogonal concepts of "passive auditing" and "active auditing" as defined in [1]. "Passive auditing" is essentially logging with the expectation that the log will be available for analysis; whether or not the log is analyzed is irrelevant to our notion of "logging," since we are separating logging from the auditing process entirely. "Active auditing" is the complement of "passive auditing," and determines if the information in the log constitutes one (or more) of a set of exceptional conditions and if so, takes action. Our use of "auditing" simply refers to the analysis and taking of action. Note that action may be taken even if no exceptional condition has occurred; this may be done to reassure systems administrators that the system is still functioning.

This paper provides a formal model of logging and auditing based on the effects of the implementation of each. The third section discusses some practical implications of this model, and the fourth applies this model to varying situations and analyzes properties to demonstrate the model's robustness.

The Model

Logging and auditing involve recording and analyzing the state of a system. Following [4], we assume that the set of entities E and a set of well-formed commands C can characterize the computer system completely. Intuitively, E is what the system is composed of and C is the set of events that can cause it to change. For all $e \in E$, $val(e)$ is the set

The support of grant NAG 2-480 from the National Aeronautics and Space Administration to Dartmouth College is gratefully acknowledged.

of values associated with the entity, $val(E)$ is the set of all values of all $e \in E$, and $VAL(E)$ is the set of values that all entities in E may assume. The set N_E of strings names the entities in E, and the function $h_E: E \rightarrow N_E$ maps each entity to a name. The set N_V of strings names the possible values of the entities in E, and the function $h_V: E \rightarrow VAL(E)$ maps each value into a string. Similarly, the set N_C of strings names the commands in C, and the function $h_C: C \rightarrow N_C$ maps each command to a name. To simplify some definitions, we require that the null command be in C.

Definition. A *system state* s is a 1-tuple (E). The collection S of all possible states is the *state space*. The *relevant part of the system state* $\sigma \subseteq s$ is the subset of (E) under consideration. The collection Σ of the relevant parts of all possible system states is the *relevant state space*.

As an example, consider the protection state of a system described by the triple (S,O,A), where S is the set of subjects, O the set of objects, and A the matrix of rights, within the system [11]. Here the entities E corresponds to the pairs of subjects and objects, and the values $val(E)$ of each entity is the corresponding access right in the access matrix. In this case, the relevant part of the system is the entire protection state, so $\sigma=(E)$. Note that this ignores entities not relevant to the protection state, so it does not describe the state of the system.

Definition. A *system* is a 4-tuple (C,S,s_0,T) where s_0 the initial state, and $\mathrm{T}: C \times S \rightarrow S$ is the *system transform*.

Informally, T the set of mappings that reflect the change of state. During the life of the system, a series of these functions will execute; the next definition captures this notion.

Definition. Let N be the set of nonnegative integers. A *system history* is a function $\Pi: \mathbf{N} \rightarrow C \times S$ such that the second element of $\Pi(0)$ is s_0, and

$$\forall n \in \mathbf{N} \; [\; [\; \Pi(n)=(c,s) \text{ and } \Pi(n+1)=(c^*,s^*) \;] \rightarrow s^* \in \mathrm{T}(c,s) \;]$$

A *relevant state history* is a function $\pi: \mathbf{N} \rightarrow C_\Sigma \times \Sigma$ such that

$$\forall n \in \mathbf{N} \; [\; \Pi(n)=(c,s) \rightarrow \pi(n)=(c,\sigma) \;]$$

If $\Pi(i)=(c,s)$ and $\Pi(i+1)=(c^*,s^*)$, we write T_i for that member of T corresponding to c^* and mapping the system state from s to s^* (that is, $\mathrm{T}_i: S \rightarrow S$ is the same as $\mathrm{T}: c \times S \rightarrow S$ where c is the first element of $\Pi(i+1)$). Also, we shall abbreviate $\Pi(i)=(c,s)$ by writing c as c_i and s as s_i. Similarly, σ_i corresponds to the relevant parts of s_i.

Consider now the effect of a projection $\tau: C_\Sigma \times \Sigma \rightarrow \Sigma$ of T on σ_{i-1}. Even though $\mathrm{T}_i(s_{i-1})=s_i$, it need not be true that $\tau_i(\sigma_{i-1})=\sigma_i$ because σ_{i-1}, and hence τ_i, may lack information necessary to produce σ_i. Intuitively, consider the relevant part of the state of a system to be the protection state of a file f. Here,

$$\sigma = \{ \; f, \text{ all subjects } \}$$

and

$$val((s,f)) = \{ \; s\text{'s set of rights over } f \; \}.$$

The system transform function τ must capture all changes to σ. Note that the state includes information about only one passive object, f, but does not include such rights as the ability of a subject to write directly to the disk. By writing directly to the disk directory, which contains the controlling representation of permissions for all files on the disk, a subject can change its access rights to the file, and hence σ; but since σ does not include access rights to the disk, there is no τ for which $\tau_i(\sigma_{i-1})=\sigma_i$. But since s_{i-1} does include those permissions, given s_{i-1}, T_i will produce the next state s_i.

We define the term *inclusive* to capture this notion.

Definition. The relevant state space Σ is *inclusive* if

$$\forall i \; [\; \mathrm{T}_i(S_{i-1})=S_i \rightarrow \exists \tau_i \; [\; \tau_i(\sigma_{i-1})=\sigma_i \;] \;]$$

In English, this says that if each element of σ captures enough information about the state of the system so that some projection of T_i can map σ_{i-1} into σ_i, then at any time i, the (relevant parts of the) next state can be determined simply by looking at σ_i. Thus only those parts of $\sigma_i \subseteq S_i$ are relevant. Of course, τ_i is just the projection of T_i into the space Σ. If $S_{i-1} \neq S_i$ and $\sigma_{i-1}=\sigma_i$, then τ_i is the identity function even though T_i is not. Note that T_i will never be the identity function, because in that case no change of state has occurred.

From here on, we shall simply deal with relevant parts of the state and assume that they are inclusive.

A logging function abstracts relevant parts of the state and turns them into output.

Definition. Let $\lambda_{state}: \Sigma \rightarrow N_E \times N_V$; then λ_{state} is a *state logging function*. Let $\lambda_{change}: C_\Sigma \times \Sigma \rightarrow N_C \times N_E \times N_V$; then λ_{change} is a *state logging function*. Collectively, call $\lambda=\lambda_{change} \cup \lambda_{state}$ where $\lambda: C \times \Sigma \rightarrow O$ and $O = N_E \times N_V \cup N_C \times N_V \times N_E$ is the output of the logging function.

Intuitively, the state logging function records the relevant components of the state of the system, and the change logging function records the specific event or action that causes the system to alter relevant components of the state as well as the new values of those components. The output of the logging function is some data recording the state or transition. For example, return to the instrumented kernel, except this time assume all system calls print a log message whenever they change rights a user has over a file, or write to either the disk directory or in-core copies of that directory; in the latter two cases they record the altered directory entry. Using the system call method, N_C are the names of the system calls, N_E the names of the files and users, and N_V the new settings of the protection modes; as each output log message prints the system call name, the name of the entity altered and the entity altering it, and the new protection mode, $O = N_C \times N_E \times N_V$, so this is change logging. If the protection modes of all files were recorded periodically, N_E would be the names of the files and users, and N_V the settings of the protection modes; as each output log message contains the name of the entity scanned and the associated protection mode, $O = N_E \times N_V$, so this is state logging. Note

that both types of logging functions may make entries in a log.

Definition. An output $o=(n_c,n_e,n_v)$ (or (n_e,n_v)) is *invertible* if there is a unique command c, entity e, and value $val(e)$ for which $h_C(c)=n_c$, $h_E(e)=n_e$, and $h_V(val(e))=n_v$ (or if there is a unique entity e, and value $val(e)$ for which $h_E(e)=n_e$ and $h_V(val(e))=n_v$).

Intuitively, an output being invertible means that the value of the entity before the logging can be determined from the log.

Definition. A *log* L of a system is a sequence of outputs $o_0,o_1,\cdots$ such that $\forall i \exists j\ [\ \lambda(\pi(j))=o_i\ ]$ The log is *unique* if each o_i is invertible. The log is *complete* if it is unique and the sequence of relevant parts of the state generating the output o_i, $i \geq 0$, is the relevant state history of the system.

Uniqueness means simply that there is only one sequence of states that could have produced the particular log. Note that this does not mean there could only be one state history, because the log may consist of outputs of states scattered throughout the state history; to obtain a unique state history, the log must be complete as well.

Proposition. Let $\sigma_0,\sigma_1,\cdots$ be the relevant state history of a system. A log $L=\{\ o_0,o_1,\ldots,o_m\ \}$ is complete if and only if the following conditions hold:

(1) each o_i is invertible;

(2) $o_0=\lambda_{state}(\sigma_0)$, and

(3) for all $i \geq 1$, $o_i=\lambda(\pi(i))$.

Proof. (*only if*) Assume conditions (1)-(3) hold. Then by (1), the log is unique; by (2), inverting the first element of the log gives the first element in the state history; and by (3), the sequence of relevant parts of the states o_i is generated by a logging function being applied to the ith element in the relevant state history.

(*if*) Assume the log is complete. Then by definition (1) holds. As there is no transformation τ_0, there can be no corresponding command, so $o_0=\lambda_{state}(\sigma_0)$, and (2) holds. Finally, by assumption relevant parts of the states in the relevant state history are inclusive, so for each pair of such states (σ_{i-1}, σ_i) there is a function $\tau_i(\sigma_{i-1})=\sigma_i$ and therefore a corresponding command $c_i \in C_\Sigma$ for which $\pi(i)=(c_i,\sigma_{i+1})$; hence $\lambda(\pi(i))=o_i$. This proves (3), and hence the proposition.

This means that a change log is *not* sufficient to generate a complete log; a state logging function must also generate output corresponding to the initial state of the system.

This proposition says that to track a system accurately, the part of the state being logged must be inclusive, enough information must be logged to reconstruct the state, some initial state must be logged, and then after *every* transition either the new state or the action causing the transition must be recorded.

Auditing consists of the *reduction* of the log, the *analysis* of the result, and the *notification* of a user or program. The function $r: O \rightarrow O$ reduces the messages from λ irrelevant outputs from the log; the function $a: O \rightarrow \Omega$ analyzes the reduced output of the log into a sequence of audit messages $\omega_i \in \Omega$; and finally the function $n: \Omega \rightarrow \Sigma \times \Omega$ notifies the appropriate people of the results, and possibly causes the system to alter the relevant parts of the state. For convenience, we collapse these into a single function α: $O \rightarrow (\Sigma,\Omega)$. Let $O_i=\{\ o_k \mid k \leq i\ \}$. If $\alpha(O_i)=(\sigma_i,\omega_i)$, then the auditing function does not alter the state of the system and is said to be *informative*. If $\alpha(O_i)=(\sigma_j,\omega_i)$ where $j \neq i$, then the auditing function feeds a response back into the system, altering its state; it is said to be *responsive*.

This captures the notion of an auditing mechanism being able to alter the state of the system in response to a problem. As an example, consider a backoff scheme for login attempts with delay x seconds. This mechanism allows the user to log in over a telephone line. After the nth failure, the system waits x^n seconds before allowing the next attempt. In this case, the auditing function would take the log output and on failure modify the state to wait for x^n seconds before allowing another attempt. This is an example of responsive auditing. If, however, the auditing mechanism simply kept statistics and did not modify the system state (for example, no backoff was done), it would be informative auditing.

The Practise

This model suggests many practical considerations, the most basic lying in the concept of "relevant state." The state to be logged must be inclusive, for if not, some information affecting the state of the system may not alter that state immediately; the resulting unexplainable change would diminish the value of the log. State logging mechanisms must record *all* parts of the relevant state, and change logging mechanisms must record *all* actions that affect the relevant state, or else it will not be possible to derive any state accurately from the log, and the log may not reflect even changes indicating an attack on the system. This implies that logging mechanisms should always be designed in synchrony with the computer system so they are an integral part of both the structure and the components of the system, as [1] pointed out.

The need to record transitions or states accurately raises the question of cost. Logging a state σ_{i-1} may require scanning a substantial portion of the system; if so, as computer systems usually change state very rapidly, recording σ_{i-1} every millisecond would make the system unusable. So, state logging mechanisms make entries periodically, resulting in an incomplete state log because the mechanism does not record a complete state history. Suppose at time i the system is in state σ_i, and the state log has outputs corresponding to states $\sigma_{t_0},\sigma_{t_1},\cdots$. If $t_i \neq i$, some states will have no corresponding output in the log, and if $t_i \gg i$, the logging mechanism will not record many states, leaving a very large window of vulnerability when an attacker can make changes to σ_i, obtain whatever is desired, and then restore the original σ_i. On the other hand, obtaining the relevant information about the transition τ_i requires instrumenting only the system

call or the external event handler causing the change, which impacts the users much less because the transitions τ_i from σ_{i-1} to σ_i are recorded as they occur. If all events (including external exceptions) are instrumented and logged, and the initial state is known, then by the proposition, the state log derived from the change log would be complete. But most implementations of a change logging mechanism focus on tracking events that indicate an attack, and for that reason their implementation either makes no record of the initial state σ_0 or assumes σ_0 is secure. Since this assumption means the log is not complete, the system may initially be in a nonsecure state, and an attacker could gain control of the computer. While the steps the attacker takes would show in the log, unless additional precautions are taken, the attacker could simply erase the appropriate entries in the log.

The composition of the log entries o_i affects the robustness of the log. By the proposition, a complete log records a state history. In a change log, each state σ_k in that history depends on the initial output and all previous outputs $o_i = \tau_i$, $i \leq k$; whereas for a state log, each state σ_k depends only on $o_k = \sigma_k$. An attacker therefore need alter only one output in a change log to conceal some change to the state of the system, but in a state log must alter each output subsequent to the change which he is concealing. So the attacker must modify more messages to conceal changes of state with state logging than with change logging.

Some monitoring systems attempt to abstract intent from a sequence of actions or changes of state. Since state logging does not indicate *how* a change of state occurred, in practise the state logging mechanism does not indicate why the state changed, but merely the new state σ_k of the system. A change log, on the other hand, indicates the action τ_k causing the system state to change and preserves enough information to determine the new state. For example, a computer system logged changes to the protection modes of a file. If a user's rights over a file were altered, a change logging mechanism recording system calls would indicate if the cause was a system call to change that user's rights, or a direct write to the protection information in the disk directory. A state logging mechanism would simply indicate that the rights had changed without indicating how. The change log therefore provides information that a system security officer can use to determine if the sequence of events was an attempted attack, an error, or indicates that a user bears further monitoring. So in addition to monitoring the state, change logging may also be used to monitor users' actions which, in turn, may be used to detect attempts to thwart security [7,13].

The central theme of the auditing portion of the model is that the auditing function takes output from the logging function and translates it into two components: a state (which may be new, involving a transition) and an output. Three components then are relevant: first, getting the output from the logging mechanism to the auditing mechanism uncorrupted; second, getting the auditing output from the auditing mechanism to its destination uncorrupted; and third, preventing interference with the transition from the old to the new state when the auditing mechanism so requires.

In the simplest types of computer systems, both informative and responsive auditing mechanisms lie on the host being audited. Unless this host has a trusted computing base, there is little if any guarantee that a determined attacker cannot interfere with the logging orauditing mechanisms. The quick response is to move the audit mechanism to another machine, which introduces a new angle of attack, namely via the transmission software and hardware; and here the difference between informative and responsive auditing becomes quite important.

Assume the auditing mechanisms lie on a remote, physically secure computer called the *audit machine* (which may be a personal computer or a workstation.) The logs are also maintained on the audit machine, and the logging mechanism on the other machines write to the audit machine over a secure communication channel. The auditing mechanism does all reductions and analyses on the audit machine.

Consider first the security of the transmissions from the main computer to the audit machine. As the audit machine is physically secure, the attacker (presumably) cannot penetrate the facility and erase or alter the log. Since the communications channel into the audit machine does not allow previously sent messages to be erased, the attacker cannot erase the log. If a trusted authentication mechanism ensures that messages sent to the audit machine are genuine log messages, the attacker cannot even forge log entries or other messages. As the auditing software is not resident on the main computer, the attacker cannot tamper with it. This leaves two vulnerable areas: the logging software (which is resident on the main machine) and the notification mechanism.

The logging software may be attacked in one of several ways, the result being that logging is disabled (which the audit machine can detect easily), many genuine but spurious messages are produced (and these messages will be eliminated during the reduction phase) or the messages produced will be incorrect and misleading. To prevent this requires protecting the logging software.

The importance in the distinction between responsive and informative auditing lies in the interaction of the computing system with the auditing subsystem. There are two aspects to this. First, the computer system must send information to the auditing subsystem when an informative auditing mechanism is in place; but with a responsive auditing mechanism, the auditing subsystem must send information back to the (relevant component of the) computer system to enable the transition to the new state. Thus, the notification phase of an informative auditing mechanism can proceed through the audit machine and not involve the monitored host at all. Then the attacker cannot alter the results of the audit by tampering with a message from the auditing system to the auditor except by physically intercepting it because the message is never on the audited computer; it is composed and printed on the audit machine. If the responsible person has a computer available, the auditing software can write the information, encrypted using a public-key cryptosystem, onto a floppy disk or to tape. The auditor obtains the medium, loads the data onto his computer, decrypts the message, and

acts accordingly. Since public key cryptosystems can be used to ensure both privacy and authenticity, the auditor would have a firm basis for accepting the results of the audit. The attacker could not tamper with the results without the auditor learning about it.

However, responsive auditing requires the results of the audit to be transmitted back to the audited computer so that it may act upon the result. This means that the communications channel between the audited computer and the audit machine is two-way, and an attacker may attack two pieces of software: the logging routine (as noted) and the routines that act upon the results of the audit. So, responsive auditing schemes have a greater window of vulnerability than informative auditing schemes.

The second aspect of the interaction of the computing system with the auditing subsystem that affects security belongs to the realm of human factors. If the auditing process is informative, a human must sift through the results to determine what is and is not significant. Experience has shown that if the ratio of what is significant to what is not significant is low, humans may very well miss important results. Further, if the output is not clear, succinct, and easy to understand, the administrators may overlook something. This often leads to the inclusion of mechanisms which suppress irrelevant information, since human beings will tend to miss important information present among a mass of irrelevant information. Yet such mechanisms usually suppress important information unintentionally, and so present a danger that the mechanism's design must deal with. However, with responsive auditing an automatic mechanism does the winnowing, so no ignore mechanism is necessary, and the program or subsystem that receives the output of the audit prescribes the format.

Performance considerations touch on both these aspects. Since informative auditing involves no change of state, the mechanism can run when the system is lightly loaded or not available to regular users, so its impact on the system from the users' perspective is minimal. Responsive auditing, however, controls whether the system changes from one state to another and so must occur after a query or command makes an entry (or set of entries) in a log but before the system can respond. The auditing mechanism must be able to run at any time, even when users are on the system, and will therefore impact the performance much more than an informative auditing system will. Worse, since the audit takes place whenever the command or query is issued, the impact may be very consistent rather than infrequent. Preserving the audit trail in reduced form may ameliorate this impact by allowing the auditing mechanism to reduce only the entries made since the last audit, and to combine the result with previously reduced data.

A major problem of both types of auditing systems is to preplan precisely what characteristics are to be audited. As observed in [1], people designing audit systems "...tend, from time to time, to create their own special purpose [auditing systems] designed only to satisfy their own initial requirements." An auditing package may satisfy all needs for a time, but when applied to a new situation, fail miserably. As an example, consider a responsive audit mechanism for a small statistical database that works by creating a matrix for queries, and applying linear analysis to the matrix to determine if answering a query will allow the questioner to deduce an individual record [3]. Such an audit tool can determine if the database will be compromised in time $O(n^2)$, which for a small n is acceptable. But as the number of entries grows, the time needed for the audit mechanism to analyze the rows of the matrix for linear independence becomes unacceptably high. Notice that this problem is less serious with informative audit mechanisms, because they do not take action to block commands or queries; the only people impacted are the recipients of the audit results.

Finally, adding a security monitoring system as an afterthought frequently produces serious problem. Such systems can in general be evaded far more easily than can security monitoring mechanisms designed into the system. As an example, consider a file monitoring program which logs changes to files on the system. If the program is not built into the kernel, then it must use a special library to make entries in the log, and a clever attacker can avoid linking that library (by creating one of his or her own, which issues the appropriate supervisor call without making an entry in the log, for example.) If the program is built into the kernel of the system, though, it cannot be (easily) subverted, because an attacker must replace the kernel with one that does not monitor – a decidedly nontrivial task!

Examples and Discussion of the Model

Some examples will make the ideas in the model more concrete. So, in this section we shall consider some logging and auditing schemes, place them within the above model, and discuss some security problems with each.

Statistical Database Control: Random-Sample-Queries

This method, introduced in [6], takes a query q concerning some class C of records in the database and applies to each record $r \in C$ a selection function $f(C,r)$ to determine whether or not r is to be used to compute the response to q. The selection function may either choose records randomly, in which case the same query may produce answers based on two different sample sets, or consistently, in which case the same query would produce the same answer, computed over the same sample set each time it is asked.

Here the relevant part of the state is

$$\sigma_i = \{\ r \mid r \in C\ \}$$

the logging function is

$$\lambda(c,\sigma_i) = R(\sigma_i)$$

where c is the query and $R(\sigma_i) = \{\ (n_e,n_v)\ \}$ a set of pairs of names and values of the records to be used. So the log is simply the names and values of records to be used. The input to the audit function is $O_i = \{\ R(\sigma_i)\ \}$, and the audit function is

$$\alpha(O_i) = (\sigma_i - \{\ r \mid f(r,C) \neq 1\ \}, \omega_i)$$

(where ω_i is any written record made). Note that the audit function alters the state to conceal those records not selected

for the sampling, so the auditing is responsive; since the logging function operates on the state of the system, it is state logging.

Statistical Database Control: Query-Set-Overlap

This control records all sets D_i, $i=1,\ldots,n$ about which queries have been answered, and answers a new query about a set C if and only if the number of records in $C \cap D_i$ is less than some parameter (for all $i=1,\ldots,n$).

The relevant part of the state is the queries answered so far, so

$$\sigma_i = \{\ D_k \mid 1 \le k \le i\ \}$$

Notice the querying of C changes the state, so the transition function τ_i is simply C. Hence

$$\lambda(c \times \sigma_i) = (C, \nu, \nu)$$

where ν is the empty entity and its associated value. If the query C can be answered, it must be added to the set, so the input to the audit function is $O_i = \sigma_i \cup \{\ C\ \}$, and the audit function is

$$\alpha(O_i) = \begin{cases} (\sigma_{i+1}, \Omega) & \text{if } C \text{ can be answered} \\ (\sigma_i, \Omega) & \text{if not} \end{cases}$$

As the auditing mechanism may change the state of the system, the auditing is responsive, and as the transition functions (queries) are logged, the logging is change logging.

Computer System Monitoring: File System Scanner

A set of programs (for example see [18]) scans file systems every night, recording users' rights over files and transmitting the list to another computer, where they are compared to a master list. If there is a discrepancy, the audit system notifies administrators of any problems via electronic mail on the machine on which the audit takes place.

Here, the state

$\sigma_i = \{$ users' rights over the files named in the master list $\}$

The logging function is

$$\lambda(c, \sigma_i) = R(\sigma_i)$$

where c is the null command and $R(\sigma_i) = \{\ (n_e, n_v)\ \}$ is a set of pairs of subject and object names and the set of rights the subject has over the object. The logging function just outputs a representation of the relevant set of rights to the audit machine. The input to the audit function is $O_i = \{\ R(\sigma_i)\ \}$, and the audit function is

$$\alpha(O_i) = (\sigma_i, \omega_i)$$

The logging done here is state logging because it captures parts of the state of the relevant files, and the auditing is informative because the state of the system is not altered. Here, ω_i is the letter mailed to the system administrators.

Computer System Monitoring: Auditing Subsystem

An auditing subsystem [15] instruments the kernel of a workstation to record specific system calls, and from that log produce an audit trail to enable reconstruction of events leading to a breach of security.

Here the state of the system

$\sigma_i = \{$ the values of monitored characteristics $\}$

(see [15] for a description). The logging function

$$\lambda(c_i, \sigma_i) = (n_{ci}, \nu, \nu)$$

is the function which maps the instrumented events into output (and ν is the empty entity and its value). The input to the audit function is $O_i = \{\ (n_{ck}) \mid 1 \le k \le i\ \}$, and the audit function is

$$\alpha(O_i) = (\sigma_i, \omega_i)$$

where ω_i is the output that another program can prettyprint. This clearly is change logging, and since its primary purpose is to allow reconstruction of events culminating in a breach of system security, the auditing is informative only.

Backing Up Computer Systems

The data on computer systems is often backed up by copying the data from the computer system to some *backup medium* such as tapes. Assume the entire file system is dumped (an "epoch dump"). Here the state of the system is

$\sigma_i = \{$ the contents of all files on the system $\}$

and the logging function

$\lambda(c, \sigma_i) = \{$ those contents dumped in a usable format $\}$

Since the logging function is recording the system state, it is a state logging function.

Discussion of the Examples

Both query-set-overlap controls and the auditing subsystem assume that the change log is accurate; if an attacker is able to subvert either system's log, reconstructing a successful attack on either system might be impossible. For this reason, the logging mechanism must be an integral part of the system. The auditing subsystem in fact recognizes this and requires that only authorized users be able to access the log if it is stored locally; since the subsystem is implemented on a workstation with enhanced security features [5], the designers believe the underlying computing base provides sufficient security. Similarly, if query-set-overlap is used*, the log must be kept in a protected area (either locally or remotely.) This would require some trusted communication path or trusted computing base. On the other hand, to defeat random sample query controls and the file system scanner, an attacker would have to tamper with every invocation of the state function f for a particular record r, or with every message involving a set of files, to prevent a change from being entered into the log; this is certainly possible, but can probably be more easily detected than a change to just one log message.

* Since the auditing system for a query-set-overlap control would have to compare the current query with every past query, it should be noted that this technique is infeasible under most practical conditions.

Both statistical database controls require that the auditing mechanism respond promptly to entries in the log, because the requested statistic cannot be released (or denied) until the auditing mechanism answers. If the system has been successfully penetrated, the attacker can alter this response to whatever is desired; this would allow him to obtain records which should be concealed. (The records might be on a remote host, and so the attacker may not be able to get access to them directly even if the machines on which the auditing mechanism and the statistical database manager reside is penetrated.) The file system auditing mechanisms do not suffer from this vulnerability if the audit is performed on a machine other than the one being audited and the results are transmitted to the relevant people using a physically secure printer that cannot be tampered with. In this sense, informative audits are less susceptible to compromise because the window in which an attacker can alter logs or results is smaller.

Conclusion

The model of logging and auditing that we have described is comprehensive enough to encompass very different schemes used in a variety of contexts; for example, statistical database query control and file access monitoring systems do not seem to be related and yet they create closely related security problems, and the mechanisms designed to improve the security of one will also improve the security of another. It also identifies many practical problems in security monitoring. By using this model to classify different auditing schemes, their usefulness for a given situation may be more readily apparent since much of the analysis stems from the particular classification. This will assist designers and system managers in their analysis of security monitoring products and schemes.

References

1. Bonyun, D., "The Role of a Well Defined Auditing Process in the Enforcement of Privacy Policy and Data Security", *Proceedings of the 1981 Symposium on Security and Privacy*, 19-25 (April 1981).
2. Bowman, R. and Kendall, P., "Security and Auditability – Mutually Compatible Objectives in the EDP Environment", *Security Audit and Control Review*, vol. 1, no. 3, 35-47, Summer 1982.
3. Chin, F. and Ozsoyoglu, G., *Auditing and Inference Control in Statistical Databases*, University of California, San Diego, CA, December 1980.
4. Cornwell, M. R., "A Software Engineering Approach to Designing Trustworthy Software", *Proceedings of the 1989 Symposium on Security and Privacy*, 148-156 (May 1989).
5. Cummings, P. T., Fullam, D. A., Goldstein, M. J., Gosselin, M. J., Picciotto, J., Woodward, J. P. L. and Wynn, J., "Compartmented Mode Workstation: Results Through Prototyping", *Proceedings of the 1987 Symposium on Security and Privacy*, 2-12 (April 1987).
6. Denning, D., "Secure Statistical Databases Under Random Sample Queries", *ACM Transactions on Database Systems*, vol. 5, no. 3, 291-315, September 1980.
7. Denning, D., "An Intrusion-Detection Model", *Proceedings of the 1986 IEEE Symposium on Privacy and Security*, 118-131 (April 1985).
8. *Webster's Third New International Dictionary of the English Language*, G. & C. Merriam Company, Springfield, MA, 1981.
9. Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. and Traiger, I., "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*, vol. 13, no. 2, 223-242, June 1981.
10. Haerder, T. and Reuter, A., "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys*, vol. 15, no. 4, 287-318, December 1983.
11. Harrison, M. A., Ruzzo, W. L. and Ullman, J. D., "Protection in Operating Systems", *Communications of the ACM*, vol. 19, no. 8, 461-471, August 1976.
12. Kinnon, A. and Davis, R. H., "Audit and Security Implications of Electronic Fund Transfer", *Computers and Security*, vol. 5, no. 1, 17-23, March 1986.
13. Lunt, T. F. and Jagannathan, R., "A Prototype Real-Time Intrusion-Detection Expert System", *Proceedings of the 1988 IEEE Symposium on Privacy and Security*, 59-66 (April 1988).
14. Mitchell, J. G. and Dion, J., "A Comparison of Two Network-Based File Servers", *Communications of the ACM*, vol. 25, no. 4, 233-245, April 1982.
15. Picciotto, J., "The Design of an Effective Auditing Subsystem", *Proceedings of the 1987 Symposium on Security and Privacy*, 13-22 (April 1987).
16. Sturgis, H., Mitchell, J. and Israel, J., "Issues in the Design and Use of a Distributed File System", *Operating Systems Review*, vol. 14, no. 3, 55-69, July 1980.
17. U.S. Department of Defense, "Trusted Computer System Evaluation Criteria", DOD 5200.28-STD, National Computer Security Center, Fort Meade, MD, December 1985.
18. Wood, P. and Kochan, S., *UNIX System Security*, Hayden Books, Indianapolis, IN, 1985.