Goal-Oriented Auditing and Logging

Matt Bishop
Christopher Wee
Jeremy Frank

Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

This paper presents a technique for deriving audit requirements from security policy, with examples for informal specifications. Augmenting these requirements with a system model allows an analyst to determine specific functions within the system that must be audited. We demonstrate the effectiveness of this technique by deriving audit criteria for the Network File System, and show that the results would detect numerous well-known attacks upon implementations of that protocol.

1. Introduction

The development of techniques to audit computer systems sprang from the need to trace access to sensitive or important information stored on computer systems, as well as access to the computer systems themselves. *Logging* records events or statistics to provide information about system use and performance; *auditing* analyzes these records to present information about the system in a clear and understandable manner.

With respect to computer security, logs provide a mechanism for analyzing the system security state, either to determine if a requested action will put the system in a non-secure state, or to determine the sequence of events leading to the system being in a non-secure (compromised) state. If the log records all events that cause state transitions, as well as the previous and new values of the objects changed, one can reconstruct the system state at any time. Even if a only subset of this information is recorded, one might be able to eliminate some possible causes of a security problem; what remains provides a valuable starting point for further analysis.

Two distinct but related problems arise: what information to log, and what information to audit. The events and actions to audit require a knowledge of the security policy of the system, what attempts to violate that policy involve, and how such attempts could be detected. From this last part comes what to log: what commands must a user use to (attempt to) violate the security policy, what system calls must be made, who must issue the commands or system calls and in what order, what objects must be altered, and so forth. Logging all events implicitly provides all

this information; the problem comes in discerning which parts of the information are relevant, which is the problem of determining what to audit.

Prior work has focussed on either the mechanics of logging or upon the reduction of specific characteristics (information) to obtain a better understanding of specific changes to system state. In this paper, we focus on the process of selecting what characteristics and events should be logged to provide a framework for appropriate auditing. We treat logging and auditing initially as an integral part of the specification of a system to enforce a security policy, and from that extend our work to include systems in which logging mechanisms were added without any overall design or goal. We consider a model that allows the modeler to derive characteristics or events that should be logged. Throughout what follows, we assume there is a security policy, and the logs are to record events (or characteristics) that indicate attempts to breach the policy. Because no computer security mechanisms are perfect, this also means that the logs will reflect any successful breaches (unless the logs themselves are tampered with; for our purposes, assume the logs are stored on write-once media).

Our basic approach is to view a security policy as a statement of constraints upon states and commands. The converse of these constraints provide a description of violations of system security; from these one can derive criteria to audit for. Combining these with the system model gives the low-level commands and data that must be logged to enable effective auditing.

The next section summarizes prior work in logging and auditing. Section 3 presents our goal-oriented auditing and logging methodology using both formal and informal security policies. In section 4, several well-known formal models, and a less formal but very widely used model, are examined and auditing criteria derived. To show the effectiveness of our methodology, in section 5 we present the NFS protocol and derive auditing criteria, and from that logging requirements. The next section shows that these criteria and requirements allow one to detect several widely-known attacks on NFS. We conclude by presenting guidelines necessary for effective auditing and logging, and analyze some common errors in logging subsystems.

2. Prior Work

Anderson [1] first proposed using audit trails to monitor threats. The use of existing audit records suggested the development of simple tools to check for unauthorized access to systems and files. The premise, that the logging mechanism was in place and active, required augmenting

the logs with additional information, but Anderson did not propose modifying the basic structure of the system's logging design, the implication being that the redesign of the security monitoring mechanism was beyond the scope of the study.

Bonyun [6] argued that a single, well-unified logging process was an essential component of computer security mechanisms. His work discussed the implications of such a process, how the users of the output (auditors and security officers) should dictate what the logs contained, issues about the extraction of logging information from the system, and a simple taxonomy of the analyses of the logs. The paper touches upon the derivation of the nature of the information that the log is to record, stating that the goals of the logging process drive which characteristics and events should be logged, but on this topic nothing more is said.

Banning *et al*. [2] also hint at the statement of the relationship of goals to the type of information logged by saying that the goals of the auditing process determine what information is logged. Again, the way in which the goals drive the determination of those types is not specified.

In many other papers, the authors discuss the way in which they use data presented in logs, or the classification of such information as useful or useless towards their end. For example, Picciotto [14] presents a sophisticated audit capability for a Compartmented Mode Workstation [9]. However, the specific use of the logs is under the control of system security officers, who determine what is to be analyzed. Sibert [19] gives a cursory rationale for the events audited in SunOS MLS. A presentation of the VAX VMM auditing facility [18] describes another auditing system without discussing how the events being monitored were chosen.

Intrusion detection mechanisms that focus on anomalous behavior have also driven research in auditing and logging. In some systems, generic logs were analyzed to determine measures that could be used to rate user sessions as anomalous. In other systems, the developers of the statistical measures have analyzed audit trails resulting from known attacks to derive the characteristics useful to log. A third approach was simply to use characteristics or events that seemed relevant.

3. Analysis of the Problem

A *security policy* is a description of a partition of the states of a computer system into secure (called *good*) and non-secure (called *bad*) states. The goal of auditing is to determine when a system transitions from a good state to a bad one, or to determine which transitions occurred from a bad to a bad state (call these *illegal state transitions*). For the moment, this analysis focuses on

only the first type of transitions.

Consider a Turing machine $\mathtt{T}$ over a finite alphabet $A = \{ a_0, ..., a_n \}$; let a_0 be the blank. The set of states S has a distinguished member s_0 representing the initial state. A state transition function σ: $S \times A \rightarrow S$ describes state transitions, and the function μ: $S \times A \rightarrow A \cup \{ r, l \}$ describes the motion of the head and the symbol it prints on the tape.

Partition the set S into subsets G and B, where the elements of G are the good states and the elements of B are the bad states. We note that the trivial cases are uninteresting, for if $B = \varnothing$ then the system can never enter a bad state, and if $G = \varnothing$ by definition the system can never enter a good state. The objective of the auditing is to establish if the current state is a member of B, and if so what state and symbol caused $\mathtt{T}$ to enter that state.

Assume $s_0 \in G$. Let $s \in S$ and $a_s \in A$. When $s \in G$ and $\sigma(s, a_s) \in B$, a security violation occurs. Hence, a simple way to determine what causes illegal state transitions is to record each s and a_s for which $s \in G$ and $\sigma(s, a_s) \in B$. A similar observation applies for illegal state transitions from one state $s \in B$ to another state $s' \in B$.

This suggests a very simple methodology for determining what information to log:

1. Determine all $g \in G$, $a' \in A' \subseteq A$ such that $\sigma(s, a') \in G$.
2. Log any $g \in G$, $a'' \in A'' \subseteq A$ such that $\sigma(s, a'') \in B$.

Both steps are well-defined as σ is a function and A' and A'' are partitions of A. Defining this methodology on a Turing machine shows that the methodology applies to any computer. The proof is immediate from Turing's thesis.

Let $s_0, ..., s_i$ be a sequence of states such that $s_{j+1} = \sigma(s_j, a_x)$ where $a_x \in A$. Given a statement of the security policy, one analyzes it to determine which states are secure ("good") and which are not secure ("bad"). Let a security policy P represent the secure operation of a computer system. The policy consists of rules $\{ p_1, ..., p_n \}$ to which secure states conform. If for any state s_i of the system, the state does not satisfy the jth policy rule (written $\neg p_j(s_i)$), then state s_i is not secure because the policy has been violated.

The system state changes as transitions occur. Further, as transitions occur due to process actions and not spontaneously, assuming the system starts in a secure state s_0, we can show:

Claim. For each nonsecure state s_i, there is at least one $j \leq n$ and one $k \leq i$ such that $s_{k+1} = \sigma(s_k, a_x)$, $p_j(s_k)$, and $\neg p_j(s_{k+1})$.

The issue is to determine the set of nonsecure transitions; that is, we need to find the set of pairs (a_x, s_k) for which s_k is secure and for which $\sigma(s_k, a_x)$ produces a nonsecure state. For a formal Turing machine, one need only list all elements of G and all elements of A, and execute each transition to see if the result is in G or A:

Claim. The complexity of finding all nonsecure transitions in `T` is $O(|G|\,|A|)$.

If the Turing machine corresponding to the system being analyzed has so many states that the elements of G cannot be described efficiently, this number may be so large as to make the problem intractable. For our discussion, we implicitly assume that the set G is described efficiently enough to keep the complexity tractable. This assumption is very practical, as the description of G can be obtained from the system specifications.

Taking secure states to be those which conform to the constraints making up the security policy, we may rewrite the above methodology as:

Determine which elements of the domain of the state transition function will produce a state which violates a constraint. Log all such elements.

As an example, suppose the states with the head over the symbol a_{23} are non-secure. While we could deal with σ, μ has as its range the Cartesian product of symbol and direction, so it is more convenient to work with given the characterization of non-secure states. For each product with a_{23} as an element of a pair in its range, determine the pairs in the corresponding domain. Then log each occurrence of the pair; that is, make an entry in a log when the Turing machine enters a state s with the head positioned over symbol a, and (s, a) is in the above list of pairs.

We proceed from Turing machines to more complex security models and policies.

4. Examples

Wirth's seminal paper [22] discusses the refinement of a programming problem to a working program. The refinement of a security policy to audit conditions is similar: one begins with the broadest expression of system design, and derives constraints at each refinement of the design by comparing actions (which cause state transitions) to the security policy, and restricting those in conflict with that policy. Note that each refinement brings the design to a new (usually more concrete) level of abstraction, and it is there we begin.

Let A_i be the set of actions possible on the system at a given level of abstraction m_i. The security policy provides a set of constraints p_i which the design must meet in order for the system to be

secure; this implies that the functions which could cause those predicates to fail must be audited. As an example, consider the design hierarchy suggested by PSOS [11]. At each level, the design for m_i is verified not to violate the predicates p_i. The predicates control specific actions at that level; thus, by auditing those actions, one can determine if an attempt to breach security has occurred. Thus, enough information must be logged to allow auditing at these points.

In what follows, we record constraints as "*action* $\Rightarrow$ *condition*.". Implication requires that the *action* be true (which means that the action occurred, in this context) before any valid conclusion about the *condition*. can be deduced. While this notation is unusual, it allows us to simply list constraints against which records can be audited; if the record's recorded action is a "read", for example, and the constraint's *action* is a "write," then the constraint is vacuously true. Further, the goal of the auditing is to determine if the policy was violated (causing a breach of security), so the result (success or failure) of the operation should match the satisfaction of the constraint. That is, if the constraint is true, the result is irrelevant,[1] but if the constraint is false and the operation is success, a security violation has occurred.

This section applies the methodology to three archetypal examples of security policies. The Bell-LaPadula model is the oldest mathematical model of a governmental security policy and is a standard policy in the military world. The Clark-Wilson model was the first articulation of an integrity policy that was not based on a multi-level security model (such as Bell-LaPadula). The Chinese Wall model is a realistic model of how some financial firms must guard their data and their actions.

4.1. Bell-LaPadula Model

The Bell-LaPadula model with tranquillity [3] is the classic security policy model. It is a multilevel model, disallowing subjects from reading entities at a higher level, or subjects from writing to entities at a lower level. (The absence of tranquillity modifies the second requirement somewhat). The simplest form of this policy linearly orders the levels L_i (for this section, $i > j$ means that $L_i > L_j$). A subject S has the level $L(S)$, and the object O the level $L(O)$. Under this policy, a system state is illegal if S reads O when $L(S) \leq L(O)$ or if S writes to O is $L(S) \geq L(O)$. Corresponding constraint are:

1. If the policy includes availability, this statement must be modified to reflect denial of service attacks. None of the models in this section deal with denial of service, but the conclusion revisits this possibility.

$$S \text{ reads } O \Rightarrow L(S) \geq L(O) \tag{1}$$

$$S \text{ writes } O \Rightarrow L(S) \leq L(O) \tag{2}$$

Auditing for security violations merely requires auditing for writes from a subject to a lower-level object, or reads from a higher-level object, and checking for violations of constraints (1) and (2). Thus, logs must contain security levels of the subjects and objects involved, the action (to determine which constraint applies), and the result (success or failure).[2] From those logs, testing for the violation of constraints (1) and (2) is trivial.

We now extend the analysis to include compartments composed of a security level and a set of one or more categories. Associated with each entity is a set of category labels chosen from the set of all categories C. The subject S is in categories $C(S)$, and the object O categories $C(O)$. Under the Bell-LaPadula model, S can write to O if, and only if $L(S) \leq L(O) \wedge C(S) \subseteq C(O)$ and S can read from O if, and only if, $L(S) \geq L(O) \wedge C(S) \supseteq C(O)$. This gives as constraints:

$$S \text{ reads } O \Rightarrow L(S) \geq L(O) \wedge C(S) \supseteq C(O) \tag{3}$$

$$S \text{ writes } O \Rightarrow L(S) \leq L(O) \wedge C(S) \subseteq C(O) \tag{4}$$

So, to audit a system implementing the Bell-LaPadula model, the logs must record the action (read or write), the levels and the compartments of the subject and object involved, and the result of the reclassification. Then the auditing simply checks each transaction to see if constraints (3) and (4) hold.

As a quick note, removing the assumption of tranquillity (the requirement that entities have fixed compartments) adds very little complication from an auditing perspective. Without tranquillity, a subject can change the security level or categories of any subject or object it controls to a level no greater than its own (this allows declassification). The command to do this and the old and new security levels and categories must be recorded.

To summarize, in the most general case, auditing Bell-LaPadula systems requires logging:

- for reads and writes, the subject's security level and compartments; and the object, its security level and compartments; and the result of the action;
- for variable security levels, the subject or object, its old and new security level and compartments, the security level and compartments of the subject changing the object, and the result.

2. Strictly speaking, the names of the subject and object may be omitted. In practise, they would assist the analyst in determining the nature and cause of the violation, and so should be included.

4.2. Clark-Wilson Model

The Clark-Wilson model [8] provides the commercial analogue to the governmental policies represented by Bell-LaPadula. The model was the first to provide a basis for assessing integrity in a realistic commercial environment, the first not to be based on the access control matrix model (and not to use a lattice representation of the computer system; an earlier integrity model [13] focussed on integrity in a commercial environment but used the lattice model to extend the work of Biba and Bell-LaPadula). Thus, the Clark-Wilson model is worth examining from an auditing point of view.

The Clark-Wilson model views integrity assurance as requiring a certification process and an enforcement process. Certification is the process whereby a security officer or administrator verifies that the initial state of the system is secure (called "valid"), that programs (called "transformation procedures" or "TPs") move the state of the system from a valid state to a valid state, that TPs write enough information to an append-only log to reconstruct the operation, and that TPs operate only on objects (called "constrained data items" or "CDIs") on which they have been certified. For our purposes, we ignore these, as the techniques for ensuring compliance are not discussed within the framework of the model. (In a sense, the certification requirements are the assumptions of the model, and so are presumed to hold.)

The enforcement rules are relevant to auditing; if they break down, integrity will be compromised. The logging mechanism therefore must log all actions which violate the enforcement rules. We consider them separately; the rules are paraphrased from [8]. In order to enforce them, the system must log four sets of information and all changes to those sets. Enforcement rule (E1) requires that the system maintain for each TP_i a set of CDIs upon which it has been certified; call this set $CDI(TP_i)$. Enforcement rule (E2) requires that the system maintain for each TP_i and each user u_j a list of CDIs that TP_i may manipulate on u_j's behalf; call this set $AUTH(TP_i, u_j)$. Enforcement rule (E3) requires that a user be authenticated; let the set $ISAUTH$ contain the set of authenticated users, so $u_j \in ISAUTH$ after u_j has been authenticated. Additionally, enforcement rule (E4) requires that the system maintain a list of the sets of users (agents) that can certify entities; let the set $CERT(u_j, TP_i)$ contain the set of entities that u_j can certify for TP_i.

E1. The system must ensure that the only manipulation of a CDI is by a TP certified on that CDI.
The corresponding constraint is:

TP_j manipulates $CDI_i \Rightarrow CDI_i \in CDI(TP_j)$ (5)

Thus, at each manipulation, the log must record the TP manipulating the CDI, the CDI being manipulated, the set of CDIs upon which that TP is certified, and the result. Then auditing simply tests each entry against constraint (5) and reports violations of that constraint. The mechanism for recording $CDI(TP_j)$ may either place a list of all members in the log entry, or may record an initial value and all subsequent changes to that set; but the mechanism must provide enough information to enable the auditor to deduce or obtain a list of the CDIs in that set at the time of the transaction.

E2. The system must ensure that TPs acting on behalf of users manipulate only those objects that the TP is authorized to manipulate on that user's behalf.

This gives the constraint:

u_i executes TP_j to manipulate $CDI_k \Rightarrow CDI_k \in AUTH(TP_j, u_i)$ (6)

Hence on each transaction, the logging mechanism must record TP_j, CDI_k, and u_i, and the audit mechanism must be able to determine the membership of the set $AUTH(TP_j, u_i)$, implying either set membership must be recorded at each transaction or an initial membership and changes must be recorded. Then the auditor simply checks that the constraint (6) holds when the transaction was executed.

E3. The system must authenticate the identity of each user attempting to execute a TP.

Enforcement rule E3 requires either that the user attempting to execute the TP has been authenticated in the past, or is authenticated before the TP executes. The constraint is:

u_i executes $TP_j \Rightarrow u_j \in ISAUTH$ (7)

Thus, the log must record the success or failure of any authentications, as well as TP_j and u_i, for each attempted transaction. Auditing consists of checking that constraint (7) is met for each attempted transaction.

E4. Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, those associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

The first part of this enforcement rule requires that only members of a set of distinguished users, called agents, be allowed to change the list of entities associate with a TP and with a particular user invoking the TP. If one considers the lists themselves as CDIs and the changing of the list to be a TP, the first part of this enforcement rule follows from (E2), and the constraint:

u_i certifies TP_j to manipulate $CDI_k \Rightarrow CDI_k \in CERT(u_i, TP_j)$ (8)

and logging requirements follow immediately. The second part of (E4) is ambiguous as stated. If its interpretation is that any agent that can certify CDI_k for TP_j cannot execute TP_j on CDI_k, the constraint is:

u_i executes TP_j to manipulate $CDI_k \Rightarrow CDI_k \notin CERT(u_i, TP_j)$ (9a)

but if the interpretation is that any agent that can certify CDI_k for TP_j cannot execute *any* transaction procedure on CDI_k, then the constraint is

u_i executes TP_j to manipulate $CDI_k \Rightarrow \forall x\ [\ CDI_k \notin CERT(u_i, TP_x)\ ]$ (9b)

where x ranges over the set of TPs. In either case, the set of agents that certify, and the set of entities that they certify, must be recorded, as must the user, transformation procedure, constrained data items for each transaction, and sufficient information to derive the membership of the set of entities that can certify TP_x to operate on u_i. The audit would check that both constraints ((8) and (9a) or (9b)) hold.

To sum up, the system must record the user, the TP, the CDI(s), and the result for each transaction. Further, the system must also record in some form the set of CDIs upon which a TP is certified, the set of CDIs that a TP may manipulate on behalf of a user, and the set of authenticated users; this can either be recorded at each transaction, or the sets recorded initially and all changes also logged.

4.3. Chinese Wall Model

The Chinese Wall model [7] presents a more complex commercial policy in which both integrity and confidentiality are essential; it mimics the rules that stock brokerage houses use and is, in fact, patterned on the requirements of English law. The model partitions the set of all subjects and objects into "conflict of interest" classes (called COIs), and partitions each COI into "company datasets" (called CDs). Let the COI of subject S be $COI(S)$ and the company dataset of subject S be $CD(S)$; for object O, define $COI(O)$ and CD(O) similarly. O may contain sanitized information, which can be read by anyone, or unsanitized information, which can only be read by a subject in another COI class, or by one in the same COI class as the object and which has already accessed another object in $CD(O)$. The predicate $san(O)$ is true if O contains only sanitized information, and false if it contains unsanitized information.

Let $CD_H(S)$ be the set of all CDs accessed by the subject S so far. Then S can read O if and

only if $COI(O) \neq COI(S)$ or $\exists O'(CD(O') \in CD_H(S))$ (intuitively, if S has read some object in a dataset it can read other objects in the same dataset; but it cannot read objects in other datasets in the same conflict of interest class as the original object), and S can write to O if and only if S can read O and $\neg\exists O'(COI(O) = COI(O') \wedge S$ can read $O' \wedge \neg san(O'))$ (intuitively, if S can read O, then S can write to O unless S can read an object in a different CD and that object contains unsanitized information). The constraints follow immediately:

$$S \text{ reads } O \Rightarrow COI(O) \neq COI(S) \vee \exists O'(CD(O') \in CD_H(S)) \tag{10}$$

$$S \text{ writes } O \Rightarrow (S \textit{ canread } O) \wedge \neg\exists O'(COI(O) = COI(O') \wedge S \textit{ canread } O' \wedge \neg san(O')) \tag{11}$$

where S *canread* O is true if the consequent of (10) holds.

To validate that these constraints hold for each transaction, the auditor must be able to determine for each transaction the elements of each COI and CD as well as the set of CDs that a particular subject has accessed. Any sanitization is also relevant. Then for each transaction, logging the subject and object identifiers, the action, and the result (success or failure) will enable the auditor to determine whether constraints (10) and (11) are satisfied.

To summarize, in the most general case, auditing Chinese Wall systems requires logging the subject and object names. The system must also log when an object is sanitized, and provide sufficient information to enable an auditor to reconstruct the membership of relevant COIs and CDs.

4.4. General Comments

The three models above show that analyzing the specific rules and axioms of the model gives specific requirements for logging enough information to detect security violations. Interestingly enough, one need not assume the system begins in a secure (or valid) state, because all the models assert that the above rules are necessary for secure operation, but not sufficient, and auditing tests necessity. That is, if the auditing of the above logs shows a security violation, the system is not secure; but if they show no violation, the system may still not be secure, because if the initial state of the system is non-secure, the results will (most likely) be a non-secure state. Hence, if one desires to use auditing to detect that the system is not secure rather than detect actions that violate security, one needs also to capture the initial state of the system. In all cases, this means recording the information that would be logged on changes to the state, at start time.

The level of abstraction of the analyses of the models is above that of a Turing machine. This eliminates the problem of enumerating the states; instead, system states are classified using sets

with membership determined by one or more constraints. This is an example of a specification that controls system behavior and allows us to model the system, implicitly, as a Turing machine. However, in the abstraction, precision can be lost; for example, the actions constituting a "write" on a Turing machine are clear. The actions constituting a "write" within the scope of the policy may not be. But they must be modelled.

This section discussed logging requirements quite generically; for example, the discussion of the Bell-LaPadula model asserted specific types of data to be recorded during a "write." In an implementation, instantiating "write" may embody other system-specific operations ("append," "create directory," *etc*.). Further, the notion of a "write" may be quite subtle, for example including alteration of protection modes, setting the system clock, and so forth. How this affects other entities is less clear, but typically involves using covert channels to write (send) information. These also must be modelled.

Naming also affects the implementation of logging criteria. Typically, objects have multiple names by which they can be accessed; however, if the criteria involve the entity, the system must log all constrained actions with that entity regardless of the name used. For example, each UNIX file has at least two representations: first, the usual one (accessed through the file system), and second, the low-level one (composed of disk blocks and an inode, and accessed through the raw disk device). Logging all accesses to a particular file requires that the system log accesses through both representations. Systems generally do not provide logging and auditing at the disk block level (due to performance); however, this means that UNIX systems generally cannot log all accesses to a given file.

The next section demonstrates how these considerations affect the derivation and implementation of logging and auditing criteria by examining the Network File System protocol.

5. Application to a Computer System

Many sites allow computers and users to share file systems, so that one computer (called a *client host*) requests access to another's (a *server host*'s) file system. The server host responds by *exporting* a directory of the server host's file system; the client host *imports* this information and arranges its own file system so the imported directory (called the *server host' s mount point*) appears as a directory in the client host's file system (this directory is called the *client host' s mount point*).

Such a situation has many potential security problems, and we present the derivation of logging criteria and audit criteria for detecting several security problems. We then validate our analysis by showing that use of these criteria allows the auditor to detect several well-known exploitations of these problems. This section discusses criteria for the servers only; the derivation for clients is similar.

The site involved in the analysis is connected to the Internet. It runs a local-area network with several UNIX [16] systems and shares file systems using the Network File System [20]. We next present a quick overview of this protocol. Then comes a discussion of policy, from which the logging and auditing criteria are derived. This section concludes by looking at several attacks from a widely-distributed NFS security analysis tool.

5.1. The NFS Protocol

When a client host wishes to mount a server's file system, its kernel contacts the server host's MOUNT server with the request. The MOUNT server first checks that the client is authorized to mount the requested file system, and how. If the client is authorized to mount the file system, the MOUNT server returns a *file handle* naming the mount point of the server's file system. The client kernel then creates an entry in its file system corresponding to the server's mount point. In addition, either the client host or the server host may restrict the type of accesses to the networked file system. If the server host sets the restrictions, the programs on the server host which implement NFS will enforce the restrictions. If the client host sets the restrictions, the client kernel will enforce the restrictions, and the server programs will be unaware that any restrictions are set.

The UNIX system represents process identity by a *user identification number* (UID) and a *group identification number* (GID). For this example, we assume the protocol underlying NFS transports this information to the NFS server program on the server host.[3]

When a client process wishes to access a file, it attempts to open the file as though the file were on a local file system. When the client kernel reaches the client host's mount point in the path, the client kernel sends the file handle of the server host's mount point (which it obtained during the mount) to resolve the next component (name) of the path to the server host's NFS server using a LOOKUP request. If the resolution succeeds, this server returns the requested file handle. The client kernel then requests attributes of the component (a GETATTR request); the NFS server

3. This is the AUTH_SYS authentication mechanism of the RPC protocol [21].

supplies them. If the file is a directory, the client kernel iterates (passing the directory's file handle and the next component of the path in a LOOKUP request, and using the obtained file handle to get the attributes in a GETATTR request) until it obtains a file handle corresponding to the desired file object. The kernel returns control to the calling process, which can then manipulate the file by name or descriptor; the kernel translates these manipulations into NFS requests that are sent to the server host's NFS server.

As NFS is a stateless protocol, the NFS servers do not keep track of which files are in use. The file handle is a capability; possession of that handle allows the possessor to manipulate the corresponding file. Further, many versions of NFS require the kernel to present the requests[4], while some accept requests from any user. In all cases, the server programs can identify the user making the request by examining the contents of the underlying RPC messages.

Strictly speaking, the MOUNT protocol is not part of the NFS protocol, but as it is central to the NFS implementation's functioning, it must be included in the analysis. This raises a homogeneity issue: MOUNT is part of the audit criteria for the NFS protocol, of which it is not a part. The ability to include different systems (MOUNT and NFS) and have the requirements for logging and auditing specify the interface is one of the strengths of this goal-oriented approach.

5.2. The Site NFS Policy

The goal of the example site policy is to regulate sharing of file systems among all systems on its local area network (with individual restrictions enforced through the NFS mechanism). All imported file systems are supposed to be as secure as the local file systems. Thus, we have the following security policy for the servers:

(P1) NFS servers will respond only to authorized clients.

The site authorizes only local hosts to act as clients. Under this policy, the site administrators could allow hosts not on the LAN to become clients, and so the policy could be less restrictive than the above statement suggests.

(P2) The UNIX access controls regulate access to the server's exported file system.

Once a client has imported a server host's file system, the client host's processes may access that file system as if it were local. In particular, accessing a file requires search per-

4. Validation is from the originating port number; the NFS implementations assume that only the superuser (operator) can send requests from ports numbered under 1024.

mission on all the ancestor directories (both local and imported).

An important ramification is the effect of the UNIX policy on file type. Only the local superuser can create device (block and character special) files locally, so users should not be able to create device files on any imported file system (or change an existing file's attributes to make it a device file). However, this policy does not restrict a client host from importing a file system that has device files.

(P3) No client host can access a non-exported file system.

This means that exporting a file system allows clients access files at or below the server host's mount point. Exporting a file system does not mean a client host can access any file on the server host; the client can access only exported files.

These policies produce several constraints:

(C1) file access granted ⇒ client is authorized to import file system, user can search all parent directories and can access file as requested, and file is descendant of server host's file system mount point.

(C2) device file created or file type changed to device ⇒ user is superuser (UID of 0)

These follow immediately from (P1), (P2), and (P3).

(C3) possession of a file handle ⇒ file handle issued to that user

As the MOUNT and NFS server processes issue file handles when a user successfully accesses a file, possession of a file handle implies that user could access the file. If another user acquires the file handle without accessing either server, that user might access files without authorization.

(C4) operation succeeds ⇒ a similar operation local to the client would also succeed

This follows from (P2). For example, as an ordinary user cannot mount a file system locally, the MOUNT operation should fail if the requesting user is not a superuser.

From the claim in section 3, a transition from a secure to a non-secure state can occur only when an NFS-related command is issued. Table 1 lists the NFS commands that a client may issue. One set takes no arguments and performs no actions; these do not affect the security state of the system. A second set takes file handles as arguments (as well as other arguments), and returns data (including status information). The third set also takes file handles as arguments, and returns file handles as results.

Those operations which take file handles as arguments require that the auditor validate con-

request	*arguments*	*action*
No arguments		
NULL	none	no action
WRITECACHE	none	*unused*
Returns non-file handle		
GETATTR	fh	get attributes of the file
SETATTR	fh, attrib	set attributes of the file
READ	fh, off, ct	get ct bytes at position off from file
WRITE	fh, off, ct, data	write ct bytes of data at position off to file
REMOVE	dh, fn	delete named file in directory
RENAME	dh1, dh2, fn1, fn2	rename file
LINK	fh, dh, fn	create link named fn for file in directory
SYMLINK	dh, fn1, fn2, attrib	create slink named fn1 for fn2 in directory
READLINK	fh	get file name that symbolic link refers to
RMDIR	dh, fn	delete named directory in directory
READDIR	dh, off, ct	read ct bytes at position off from directory
STATFS	dh	get file system information
Returns file handle		
ROOT	none	get root file handle (*obsolete*)
CREATE	dh, fn, attrib	create file fn in directory with attributes
MKDIR	dh, fn, attrib	create directory fn in directory with attributes
LOOKUP	dh, fn	get file handle of named file in directory

Table 1. NFS operations. In the above, fh is “file handle”, “fn” is “file name”, dh is “directory handle” (effectively, a file handle), “attrib” are file attributes, off is “offset” (which need not be a byte count; it is positioning information), ct is “count”, “link” is “direct alias”, and “slink” is “indirect alias”.

straint (C3); hence, when a server issues a file handle, the file handle, the user to whom it is issued, and the client to which it is sent must be recorded:

(L1) When a file handle is issued, the server must record the file handle, the user (UID and GID) to whom it is issued, and the client host making the request.

The semantics of the UNIX file system say that access using a path name requires that the user be able to search each directory. However, once a file is opened, access to the file requires the file descriptor, and is not affected by the search permissions of parent directories. From the operation arguments, file handles seem to refer to open objects; for example, SYMLINK creates a symbolic link, which is effectively a write to a directory object; the argument to SYMLINK is the directory’s handle. Hence, file handles resemble descriptors more than path names, so the auditor need not verify access permission whenever a user supplies a file handle[5]. The only issue is whether the

5. The next subsection discusses the alternate approach.

server issued the file handle to the user performing the operation:

(L2) When a file handle is supplied as an argument, the server must record the file handle and the user (UID and GID).

A file handle allows its possessor to access the file to which the handle refers. Any operation which generates a file handle must record the user and relevant permissions for the object in question. For example, on a LOOKUP, recording the search permissions of the containing directory enable the auditor to determine if the user should have had access to the named file. On a CREATE, recording the write permissions of the containing directory indicate whether the use could legitimately write to the containing directory.

(L3) When a file handle is issued, the server must record the relevant attributes of any containing object.

Finally, whether the operation succeeds or fails, the system must record the operation's status so the auditor can verify the result:

(L4) Record the results of each operation.

As each operation performs a different function, we derive the audit criteria of each operation separately.

MOUNT. On a MOUNT operation, constraints (C1) and (C4) requires the audit criteria:

(A1) Check that the MOUNT server denies all requests by unauthorized client hosts or users to import a file system that the server host exports.

("Unauthorized users" refers specifically to those users who could not perform the operation locally.) This means the MOUNT server must record (L1) and (L4)

LOOKUP. On a LOOKUP operation, constraints (C1) and (C3) give the audit criteria:

(A2) Check that the file handle comes from a client host and a user to which it was issued.

(A3) Check that the directory has the file system mount point as an ancestor and that the user has search permission on the directory.

Note that the check for the client being authorized to import the file system (in (C1)) is implicit in (A2), as if the client host is not authorized to import the file system, the client host will not obtain the file handle for the server host's mount point. Performing this audit requires logging (L2), (L3) (the relevant attributes being owner, group, type, and permission), and (L4). Audit criterion (A3) requires recording the name of the file being looked up; from this and the file handle, the auditor

can reconstruct the ancestors of the file.

(L5) Record the name of the file argument in the LOOKUP operation.

GETATTR. Constraint (C3) gives the audit requirement that the file handle is indeed legitimate, *i.e.* (A2), and whether the operation succeeds (L4). The logging requirement is the data for this check, *i.e.*, (L2).

SETATTR. Constraint (C3) gives the audit requirement that the file handle is indeed legitimate, *i.e.* (A2). Constraint (C2) requires the (new) type of the file not be "device":

(A4) Check that the new type of the file is not "device," or that the user is the superuser.

Constraint (C4) requires the requester be either the owner or the superuser:

(A5) Check that the operation fails if the user is neither the superuser nor the owner of the file.

The logging requirement is the data for (A2), namely (L2); whether it succeeded (L4); and

(L6) Record the type and owner of the file both before and after the command is executed.

READ. Constraint (C3) requires checking that the file handle is legitimate (A2). Hence the system must log the information in (L2), and whether the operation succeeds (L4).

READLINK. As this is effectively a READ operation, the requirements are the same as for READ.

WRITE. This is analogous to the READ request.

CREATE. The requestor must have write access on the containing directory; the NFS server assumes the requester has that access. As the create operation is like a write to a directory, the logging and auditing requirements are simple modifications of those of the WRITE request. Thus from constraint (C3), logging requirement (L2) and audit requirement (A2) apply; from constraint (C4), logging requirement (L3) also applies; the result of the operation (L4) indicates whether the constraints were obeyed. Validating the constraints hold also requires:

(A6) Check that the UID and GID of the client process and the owner, group, type, and permission attributes of the directory give the client permission to create the file, and that the result conforms to this determination.

REMOVE. The requestor must have write access on the containing directory and may have to meet other constraints[6] (which this analysis ignores). As the remove operation is like a write to a

6. On many versions of the UNIX operating system, the requestor must own the file being deleted.

directory, the logging and auditing requirements are analogous to the CREATE request, except that the auditing checks the remove right.

RENAME. As the rename operation is like a remove and a create operation, the logging and auditing requirements are those of the REMOVE request for the directory containing the file, and those of the CREATE request for the directory which will create the file. This observation does not imply any ordering of the CREATE and REMOVE operations.

LINK. As the link operation is like creating a file in a directory, the logging and auditing requirements are those of the CREATE request for the directory which will create the file.

SYMLINK. The requestor must have write access on the containing directory; the NFS server assumes the requester has that access. As the SYMLINK request is like a CREATE, the logging and auditing requirements are those of the CREATE request.

MKDIR. This is analogous to the CREATE request, except it creates a directory.

RMDIR. This is analogous to the RMDIR request, except it deletes a directory.

READDIR. This is analogous to the READ request, except it reads from a directory file.

STATFS. This is analogous to the GETATTR request, except it returns file system attributes.

Table 1 summarizes the above requirements for logging and auditing.

5.3. Discussion

If we took the alternate approach (requiring access checking when a file handle is presented), we would augment the audit and logging criteria to check that the owner, group, and protection modes of the objects involved allowed the operation. For example, the READ request would require the following additional audit criterion to ensure compliance with (C4):

(A7) Check whether the UID and GID of the client process and the logged attributes of the file give the client permission to read the file, and that the result conforms to this.

This requires logging the information in (L3) as well. Such an approach would treat file handles as file names, and attempt to enforce the Principle of Complete Mediation (stating that the system must validate every access to every object) [17] more precisely than the above model.

We now present several well-known attacks on NFS and show that meeting the above requirements allows an auditor to detect those attacks.

request	*logging*	*auditing*
MOUNT	(L1), (L4)	(A1)
GETATTR	(L2), (L4)	(A2)
SETATTR	(L2), (L4), (L6)	(A2), (A4), (A5)
READ	(L2), (L4)	(A2)
WRITE	(L2), (L4)	(A2)
REMOVE	(L2), (L3), (L4)	(A2), (A6) [“delete permission”]
RENAME	(L2), (L3), (L4)	(A2), (A6) [“create permission” on destination directory, “delete” on source directory]
LINK	(L2), (L3), (L4)	(A2), (A6) [“create permission”]
SYMLINK	(L2), (L3), (L4)	(A2), (A6) [“create permission”]
LOOKUP	(L2), (L3), (L4), (L6)	(A2), (A3)
READLINK	(L2), (L4)	(A2)
RMDIR	(L2), (L3), (L4)	(A2), (A6) [“delete permission”]
READDIR	(L2), (L3), (L4)	(A2), (A6) [“delete permission”]
STATFS	(L2), (L4)	(A2)
CREATE	(L2), (L3), (L4)	(A2), (A6) [“create permission”]
MKDIR	(L2), (L3), (L4)	(A2), (A6) [“create permission”]

Table 2. Summary of logging and auditing requirements for NFS. The rights in brackets are the rights that the auditor checks when validating auditing requirements.

5.4. Attacking NFS

This section discusses several attacks exploiting security flaws in some implementations of NFS [10]. Assuming the system implements logging and auditing as outlined above, an auditor can detect all of these attacks. As above, this analysis assumes that the underlying network provides authenticity, confidentiality, and availability sufficient to allow logging and auditing mechanisms to meet those requirements.

Unauthorized host mounts file system. Although NFS implementations provide an access control mechanism, use of this mechanism is optional, and many implementations have bugs.[7] When the MOUNT server receives a MOUNT request, it logs the client host (see (L1)). Then if the MOUNT request succeeded, (A1) instructs the auditor to check that the client host was authorized to mount the file system, and if the MOUNT request failed, (A1) instructs the auditor to check that the client host was not authorized to mount the file system. In either case, the auditor will detect

7. For example, one vendor supplied a version that required system administrators to list authorized hosts by name. If the list was more than 256 characters long, the implementation simply turned off access control and allowed any host to mount the file system.

attempts by an unauthorized host to mount the file system.

Mounting through the portmapper. The second attack relies on an error in the configuration of the access control list. If the access control list includes the server host, a client host can request that the *portmapper* service on the server host act as a proxy to the server host's MOUNT server. The client host then relays mount requests to the mount server through the portmapper, which will then forward all requests as though they came from the server host to the mount server. This gives the client host the same mount rights that the server has over that file system,[8] and as the NFS server is stateless, it cannot invalidate the use of handles obtained in this way. Again, (L1) states that the MOUNT logs will show the server host requesting the mounting of the server's exported file system, and (A1) instructs the auditor to check that the requesting host (server host) is authorized to mount the file system. As the server host should never be so authorized, this detects the second attack.

Guessing file handles. The third attack is the simplest in concept but the most complex in execution. As file handles are capabilities, if an attacker can guess (or copy) a file handle, that attacker can access files without mounting the server host's exported file system. As the server's MOUNT and NFS servers may enforce some file system attributes (such as read-only), this allows the attacker to bypass file system controls; as the file handle is a capability, the attacker can use that handle to perform unauthorized actions such as deleting system files. The MOUNT server or the NFS server will send out all file handles obtained in response to the MOUNT, LOOKUP, CREATE, and MKDIR requests. The system must record all such file handles (L1). On any request using a file handle, the system must also record that file handle (L2). Then by correlating these logs, (A2) allows the auditor to detect guessed file handles, and hence this attack.

Change to parent directory not in exported file system. The fourth attack exploits a bug in an early implementation of NFS. Under certain conditions[9] an attacker can set the process' current working directory to the server host's mount point of an exported file system, and change to the parent directory, thereby obtaining a file handle of an unexported directory. Given this handle, the attacker can roam through all subordinate directories whether or not they are exported. The LOOKUP and MOUNT commands generate file handles for existing files and directories, and the

8. If the access control list does not name the server, the local host cannot mount the file system using NFS. It can mount it using local mounting procedures. The local server should never be in the access control list.
9. Specifically, that the exported file system's root directory is not the root directory of the physical file system.

MOUNT command can do so only for specific directories (the mount points of the exported file systems). Logging requirements and (L5) for the LOOKUP request require logging the directories and files accessed; auditing requirement (A3) requires an auditor check that the file name refer to a file or directory in the exported file system. These requirements will enable an auditor to detect the fourth attack.

Creating a device. The fifth attack exploits a feature allowing the creator of a file to create any type of file, including a device file. This allows an attacker to gain access to, and alter, the client system's privileged memory. The attacker simply requests that the NFS server create a device corresponding to kernel memory. By logging the type of file created, and by checking that none are device files, requirements (L3) and (A6) enable the auditor to detect such an attack.

UID truncation to 0. Finally, many NFS implementations support 32-bit UIDs, but many UNIX systems support only 16-bit UIDs. The client will pass a 32-bit value to the server, which checks for UID 0; if the UID is not 0, the UID is passed to the kernel. As the kernel treats UIDs as 16 bits, the upper 16 bits are discarded. In short, only the low-order 16 bits of the UID matter. The attacker generates a UID that is a multiple of 2^{16}, and the kernel will allow access to system files (as UID 0 has no access restrictions). As all file operations log the UID ((L1), (L2), (L3)) and have audit requirements to check that the UID is authorized to manipulate or access the file, the auditor need only check that access is properly denied. If access is not properly denied, or if the kernel audits file accesses, the mapping of a non-root UID to the root UID will be obvious.

6. Conclusion

This paper presented a methodology to design a logging mechanism as a "stepwise refinement" of the goals of the auditing, and demonstrated its effectiveness by analyzing a protocol against which several attacks are known; the system built using this methodology detects these attacks.

The Turing machine model leads to an interesting result with respect to intrusion analysis. Intrusion analysis is the art of analyzing an attacked system to determine what sequence of actions put the system into a vulnerable state, and what steps occurred once the system was in such a state. Because the range of sigma is a state, and the domain a pair with a state and a symbol, the state transitions are invertible only if there is a single symbol; otherwise, the function is not injec-

tive and hence at least some elements of the range have multiple corresponding elements of the domain. So, for realistic systems, such reconstruction will never have entropy 0 unless the log contains enough information to derive prior states. In that case, the analyst must work from the logs and not by reconstructing actions directly from the system.

The significance of this paper lies in its unique approach to the audit problem. In the past, scant attention has been paid to the mechanics of developing a theory of how to determine what should be logged; the focus has instead been on the mechanisms. Future research topics in this area include the types of security violations that auditors in the commercial and government world look for, since that will lead to logging and auditing mechanisms that are useful in practice. The issues discussed also suggest applying planning technology to this problem; that is another topic for future research.

Acknowledgments: Thanks to John Gregg and David Day of the UC Davis Office of Internal Audit for useful discussions about non-computer oriented auditing involving the analysis of computer systems, and to Becky Bace, Biswaroop Guha, James Hoagland, and Karl Levitt for useful discussions. Doug Mansur and Harry Breustle not only supported and encouraged the development of the techniques but also provided valuable insight into several of the problems. Lawrence Livermore National Laboratory supported this work through a contract to the University of California at Davis on behalf of the U.S. Department of Energy under contract no. W-7405-Eng-48. The Office of INFOSEC Computer Security funded some of the preliminary research through the University Research Program under contract MDA904-92-C-5139 to Dartmouth College (and later transferred to the University of California at Davis).

7. References

[1] James P. Anderson, "Computer Security Threat Monitoring and Surveillance," James P. Anderson Co., Fort Washington, PA (1980).

[2] D. Banning, G. Ellingwood, C. Franklin, C. Muckenhirn, and D. Price, "Auditing of Distributed Systems," *14th National Computer Security Conference Proceedings* pp. 59-68 (1991).

[3] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," Technical Report M74-244, The MITRE Corporation, Bedford, MA 01730 (1974).

[4] S. Bellovin, "Security Problems in the TCP/IP Protocol Suite", *Computer Communication Review* **19**(2) (1989).

[5] Matt Bishop, "A Model of Security Monitoring," *Proceedings of the Fifth Annual Computer*

Security Applications Conference pp. 46–52 (1989).

[6] David Bonyun, "The Role of a Well-Defined Auditing Process in the Enforcement of Privacy Policy and Data Security," *Proceedings of the 1981 IEEE Symposium on Security and Privacy* pp. 19-26 (1981).

[7] D. F. C. Brewer and M. J. Nash, "The Chinese Wall Security Policy," *Proceedings of the 1989 IEEE Symposium on Security and Privacy* pp. 206-214 (1989).

[8] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proceedings of the 1987 IEEE Symposium on Security and Privacy* pp. 184-194 (1987).

[9] P. T. Cummings, D. A. Fullam, M. J. Goldstein, M. J. Gosselin, J. Picciotto, J. P. L. Woodward, and J. Wynn, "Compartmented Mode Workstation: Results Through Prototyping," *Proceedings of the 1987 IEEE Symposium on Security and Privacy* pp. 2-12 (1987).

[10] Leendert van Doorn, "nfsbug.c," ftp://ftp.cs.vu.nl/pub/leendert/nfsbug.shar (April 1994).

[11] R. J. Feiertag and P. G. Neumann, "The Foundations of a Provably Secure Operating System (PSOS)," *Proceedings of the National Computer Conference* **48** pp. 329–334 (1979).

[12] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., Reading, MA (1989).

[13] Steven B. Lipner, "Non-Discretionary Controls for Commercial Applications," *Proceedings of the 1982 IEEE Symposium on Security and Privacy* pp. 2-10 (1982).

[14] J. Picciotto, "The Design of an Effective Auditing Subsystem," *Proceedings of the 1987 IEEE Symposium on Security and Privacy* pp. 13-22 (1987).

[15] J. Postel, "User Datagram Protocol," RFC 768 (Aug. 1980).

[16] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) pp. 365-374 (1974).

[17] Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE* **63**(9) pp. 1278–1308 (Sep. 1975).

[18] K. F. Seiden and J. P. Melanson, "The Auditing Facility for a VMM Security Kernel," *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy* pp. 262-277 (1992).

[19] W. Olin Sibert, "Auditing in a Distributed System: Secure SunOS Audit Trails," *11th National Computer Security Conference* pp. 81-91 (1988).

[20] Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," RFC 1094 (Mar. 1989).

[21] Sun Microsystems, Inc., "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1050 (June 1988).

[22] Niklaus Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM* **14**(4) pp. 221–227 (Apr. 1971).