UNIVERSITY OF CALIFORNIA
Santa Barbara

USTAT
A Real-time Intrusion Detection System for UNIX

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Koral Ilgun

Committee in charge:

Professor Richard A. Kemmerer, Chairperson

Professor Amr El-Abbadi

Professor Jianwen Su

November 1992

This page is intentionally left blank

The thesis of Koral Ilgun
is approved:

Committee Chairperson

November 1992

This page is intentionally left blank

This page is intentionally left blank

ABSTRACT

USTAT
A Real-time Intrusion Detection System
for UNIX

by

Koral Ilgun

This thesis presents the design and implementation of a real-time intrusion detection tool called USTAT, a State Transition Analysis Tool for UNIX. The original design was first developed by Phillip A. Porras and presented in [Porr91] as STAT, a State Transition Analysis Tool. STAT is a new model for representing computer penetrations, and the model is applied to the development of a real-time intrusion detection tool. In STAT, a penetration is identified as a sequence of state changes that take the computer system from some initial state to a target compromised state.

In this document, the development of the first USTAT prototype, which is for SunOS 4.1.1, is described. USTAT makes use of the audit trails that are collected by the C2 Basic Security Module of SunOS, and it keeps track of only those critical actions that must occur for the successful completion of the penetration. This approach differs from other rule-based penetration identification tools that pattern match sequences of audit records.

This page is intentionally left blank

Acknowledgments

Many people deserve acknowledgment for their support and assistance during the development of this thesis. First, I wish to thank to my advisor, Prof. Richard A. Kemmerer, for presenting this project to me, for his keen interest and support in this thesis and for proofreading this document. Each meeting with him added invaluable aspects to the implementation and broadened my perspective. I also thank Phillip A. Porras, who developed the State Transition Analysis Tool and provided many helpful comments on the implementation of USTAT. Many thanks to Dave Probert, especially for providing megabytes of disk space for audit collection. Thanks to Joe Rollins and Zeki Basbuyuk as well, for helping me in proofreading this thesis. Finally, I wish to dedicate this thesis to my parents, without whose support I would not have been able to attain this degree.

This page is intentionally left blank

Contents

List of Figures

List of Tables

This page is intentionally left blank

Chapter 1

Introduction

This page is intentionally left blank

1 Introduction

Despite the undeniable progress in the area of computer security that has taken place over the past decade there is still much to be done to improve security of today's computer systems.

The following adage has been taken from the *frequently asked questions* of the *Alt.security* [1] newsgroup.

> *"The only system that is truly secure is one that is switched off and unplugged, locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it."*

A system that is known to be secure to an outside attack by preventing access from outside can still be vulnerable to inside attacks accomplished by abusive usage of authorized users. Detecting such abusive usage not only provides information on damage assessment, but also helps to prevent future attacks. These attacks are usually detected by tools referred to as *Intrusion Detection Systems*.

The most popular and well-known data for an intrusion detection system is the *audit data*. An *audit trail* refers to the (audit) records of all activities on a system kept in chronological order. Since there exists a record for each activity (which may correspond to one system call) on the system, theoretically it is possible to manually analyze the source data and detect any abnormal activity on the system. However, the vastness of the audit data provided by an audit collection system often makes manual analysis impractical. Therefore, an automated audit data analysis tool is the only solution. An intrusion detection system is a possibly enhanced version of such an analysis tool.

The State Transition Analysis Tool (STAT) presented in [Porr91] introduces a novel idea to represent computer penetrations and provides an expert system model to detect compromises. STAT makes use of the audit trails that are provided by the audit collection mechanisms of the target systems.

USTAT, which is a State Transition Analysis Tool for UNIX [2], takes the

---

[1] The *frequently asked questions* is maintained by Alec Muffet with contributions from numerous others.

[2] UNIX is a registered trademark of AT&T

STAT effort one step further by implementing a prototype of an intrusion detection system for UNIX. USTAT uses the audit collection mechanism that exists as an add-on package to SunOS 4.1.1[3], which is called C2-BSM (Basic Security Module).

USTAT's design gives the system administrator or the site security officer the opportunity to monitor, detect and possibly preempt certain activities that would be considered illicit or that would cause a security risk for the system.

The following chapter gives an overview of STAT without getting into the specifics of USTAT. Chapter 3 gives the design issues of USTAT and lists major changes that have been made to implement USTAT that differ from the design that was proposed in [Porr91]. Chapter 4 presents a detailed discussion of each component of USTAT. It discusses some of the ideas that were formed during the development of STAT, analyses the problems encountered in implementing the USTAT prototype, presents the changes that were made to the original design and gives the algorithms that were used to implement the task of each component. Chapter 5 shows the results of testing the USTAT implementation. Finally, chapter 6 provides direction for future research and gives conclusions about the research reported in this thesis.

---

[3]SunOS is a trademark of Sun Microsystems

Chapter 2

Overview of STAT

This page is intentionally left blank

2 Overview of STAT

In [Porr91] Porras introduces a new model for representing computer penetrations and applies the model to the development of a real-time intrusion detection tool. The model is called Penetration State Transition Analysis and the application tool is referred to as STAT (a State Transition Analysis Tool). In this model a penetration is viewed as a sequence of signature actions and a corresponding sequence of state changes that lead the computer from some initial state to a target compromised state.

In the STAT approach penetrations are represented by state transition diagrams [Porr91]. STAT uses these representations and the audit trails of the target system to keep track of the critical actions that must occur for the successful completion of a penetration. This approach of matching critical actions differs from other rule-based penetration identification tools that pattern match sequences of audit records.

This chapter gives a brief overview of the STAT design. The interested reader should refer to [Porr91] for further information.

2.1 Introduction to STAT

We can characterize STAT by its three basic properties:

- STAT is a real-time expert system intrusion detection tool,
- it employs rule-based analysis on the audit trails of multi-user computer systems, and
- it searches for known penetrations.

2.1.1 Auditing Users

Intrusion detection systems need some source of input to analyze. This might be the accounting information on the target system or similar other resources. Currently the most prevalent source of data used for intrusion detection is the audit data. Auditing users on a system provides valuable data for the detection of intrusions, yet the amount of data makes the manual analysis in most cases impractical, if not impossible. This emphasizes the need for an automated analysis tool. Automated analysis is perhaps the most common feature of current intrusion detection systems. The next section gives a brief review of the different approaches to intrusion detection as well as issues to be considered with regard to particular computer environments.

2.1.2 Issues in the Development of Intrusion Detection Systems

There are many issues to be considered while implementing an intrusion detection system. Among the most important are the topology of the target system (network vs. stand-alone) and the analysis mode (batch vs. real-time analysis).

2.1.2.1 Network vs. Stand-alone

By a stand-alone system we mean a system configuration consisting of a host computer with several terminals attached to it. The auditing process is done on the host computer. By a network we mean a collection of possibly heterogeneous hosts serving many terminals. To audit the network, each host computer needs to be audited.

In [Scha91] Schaen lists many issues that exist when auditing in a networked environment. The audit collection and storage issues listed in [Scha91] are: what data to store, where to store it, how to reduce its volume, and how long to keep it. The answer to these questions is not easy even for a stand-alone system and it becomes more complex when the target system is a network. Schaen emphasizes that there are more protection requirements for a network environment than for a stand-alone system because of the multiplicity of the components, the multiplicity of administrative domains, the need for one component to authenticate itself to another, and the need to protect communications. Schaen also lists the issues regarding the integrity of audit data, such as:

> *"A network audit must be able to integrate the audit trails collected by different components. This could be done at collection or analysis time."*

Finally, Schaen discusses the issues that arise in using audit data that has been collected, and he tries to answer questions of how to integrate multiple audit trails, how to trace transactions across multiple components, and how to analyze the vast amounts of data collected.

2.1.2.2 Batch vs. Real-time

Another important issue in audit data analysis is whether it has to be done in real-time or in batch mode. Batch mode analysis has a key advantage: analysis can be done during low periods of cpu usage and/or at another computing facility. However, the disadvantage is that an impending compromise cannot be preempted, hence the damage will be detected after it has been done.

Real-time analysis can also be done at a remote site, but then a high-speed reliable data channel is needed between the target system and the analysis system. Manual analysis of audit data is impractical and real-time manual analysis is absolutely infeasible.

For a computer system with high security requirements, where any loss or disclosure of information cannot be afforded, real-time analysis is the only solution. In this case, software and hardware performance is a major issue. We refer to these considerations when we discuss USTAT's analysis of audit data.

2.1.3 Approaches to Intrusion Detection

Different approaches are used by different types of detection tools:

- Statistical anomaly detection tools use statistical analyses to measure variations in the amount and type of audit data. There are two different techniques to be discussed under this category:
  - Threshold Detection: In this approach, each occurrence of a specific event is recorded. The idea is that an unusually high number of occurrences within a specified period may indicate the presence of an intruder. The difficulty with this technique is to identify the threshold number, and the interval of analysis time. NADIR [Hubb90] and MIDAS [Sebr88] employ threshold detection components in their systems.
  - Profile-Based: This approach uses statistical measures to identify expected behavior of users or user groups. Masqueraders and mis-

feasors can be detected by monitoring a system's audit log for user activity that deviates from established patterns of usage. The main advantage of profile-based anomaly detection is that it doesn't need any prior knowledge about the security flaws of the target system. The issues that may hinder the effectiveness of these systems are false positive/negative rates and gradual misbehaviors.

- Rule-based Anomaly Detection: The major difference between rule-based and statistical anomaly detection tools is that rule-based anomaly detectors use sets of rules to represent and store the usage patterns, whereas statistical anomaly detectors use statistical formulas to identify usage patterns in audit data. This technique shares the same advantage and disadvantages with the statistical anomaly detection techniques. Examples of this approach are W&S [Vacc89] and TIM [Chen90].

- Rule-based Penetration Identification: These tools are characterized by their expert system properties that fire rules when audit information indicates illegal activities. Most of the current intrusion detection tools supplement their anomaly detection components with rule-based expert system components. For example, IDES [Lunt92], NADIR, and W&S.

All these approaches and their corresponding tools are discussed in detail in [Porr91].

2.1.4 Features of STAT

In the last section we categorized different intrusion detection systems. STAT falls in the category of rule-based penetration identification systems.

Current rule-based penetration identification tools have several weaknesses that STAT aims to improve on. One major weakness is that they use the audit records to represent a penetration scenario and try to pattern match their rules to the audit records. The representation of scenarios using audit records is very non-intuitive. This process requires a person who is experienced in the particular intrusion detection system and who has in-depth knowledge of the underlying audit collection mechanism. Pattern matching rules to audit records gives no flexibility to the representation of penetrations. For the same scenario several different audit record sequences might exist and those minor variations might slip unnoticed. STAT overcomes this problem by using a higher-level audit record independent representation of penetration scenarios and by supporting permutable rule sequences.

Garvey and Lunt [Lunt92] also proposed a new intrusion detection approach called Model-Based Intrusion Detection. With this approach they address the above problems and provide an audit record independent technique to represent intrusion scenarios.

STAT is designed to be a real-time system. One of its main features is to attempt to preempt an attack before any damage is done to the system. This preemption is only possible with real-time analysis. STAT does not audit unsuccessful events. These type of events, especially failed login or failed read attempts are addressed by anomaly detectors. By means of high-level representation of the scenarios, creating and updating the rule-base becomes much easier.

Among other features and strengths of STAT that are not addressed by the other rule-based penetration identification tools are the ability to detect cooperative attacks, the ability to detect attacks that span more than one user session, and the ability to foresee an impending compromise and attempt to preempt it.

STAT is not intended to detect all threats. The following is a list of threats that are out of scope of STAT analysis.

- Manipulation of components outside the system's execution domain, e.g., wiretapping.
- Denial of service attacks, e.g., excessive CPU utilization.
- Variations from normal patterns of use, e.g., user A logs on as user B and accesses files that user B doesn't usually access.
- Failed login attempts and failed file access attempts.

The above list consists of those attacks that are either not recorded by the audit collection mechanism, or that cannot be represented by using a state transition diagram. However, using state transition diagrams we are able to detect the abusive usage of legitimate users, e.g., account break-ins and the unauthorized reference, modification or deletion of data.

2.1.5 STAT in a Larger Intrusion Detection System

No intrusion detection tool is meant to be a catch-all tool for intrusions. They all have weaknesses and strengths. STAT is designed to be supplemented by an anomaly detector. These two tools complement each other; that is the weakness of one is the strength of the other.

Figure 2.1, which is taken from [Porr91], shows STAT in the context of a larger intrusion detection system. Here, the audit record preprocessor provides both systems with their own prerequisite audit record formats. Both of the tools send their results to the Site Security Officer (SSO) Interface, where the SSO can keep track of the current state of the intrusion detection system and where he/she can be informed of potential intrusions.

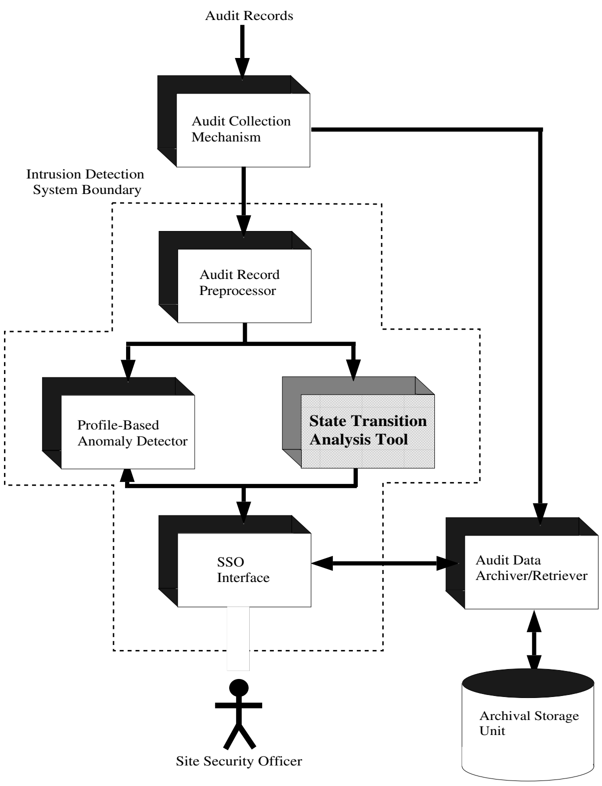


Figure 2.1 STAT in a larger Intrusion Detection System

We define the SSO as a trusted individual who has the role of setting up the auditing system, monitoring user activities, and analyzing and managing audit trails. The SSO might be the same person as the system administrator. Throughout this document we refer to him/her as the SSO.

2.2 State Transition Analysis

2.2.1 State and State Transitions

As its name implies, STAT analyzes the audit data by keeping track of the state changes on the system. Porras [Porr91] defines the state as follows.

> *"State is the collection of all volatile, permanent and semi-permanent data stores of the system at a specific time."*

However, for a given instant in time, it is infeasible to determine the value of all the data stores on the system. A close observation of the penetration scenarios reveals that we actually need only a fraction of the data stores to represent the scenarios. The attributes we are interested in depend on the particular scenario.

During all the discussions about state transitions one question remained unanswered: how do we define a compromised state? In other words, can we suspend the system at a specific instant of time, take a snapshot of it by dumping all of the volatile, permanent and semi-permanent data values and tell whether these values represent a compromised state? So far, the answer is NO. For instance, we can define one compromised state as follows. "A user (non-root) is running an interactive shell with an effective user-id equal to root." So, the user can talk to the shell while having root privileges. That clearly defines a compromised state. One might ask whether we can trivially identify this state by looking at the audit records. The audit records can tell us that the user has run a shell, and now has an effective user-id of root. However, that information is not sufficient to identify whether he/she can talk to the shell. He/she may be running a setuid-to-root program that doesn't give an interactive shell to the user.

Currently, there is no audit record format that informs the analyzer in such a descriptive manner. Therefore, we try to obtain more information by looking at the history of the "compromised" state. We try to see how we have gotten there, and we try to answer two basic questions: which actions caused the user to gain an interactive shell with root privileges, and how are the system attributes affected by these actions? The observations made by answering these questions make the state transition analysis a very effective tool for describing and detecting penetration scenarios. Since the actions are as significant as the states of the system, the tool is called the state *transition* analysis tool rather than state analysis tool.

2.2.2 Representing Penetration Scenarios: State Transition Diagrams

In STAT a very intuitive method of representing penetration scenarios, called State Transition Diagrams, is used. A state transition diagram is the graphical representation of a penetration scenario. Figure 2.2 shows two major components of a state transition diagram: nodes represent the states and arcs represent the actions.

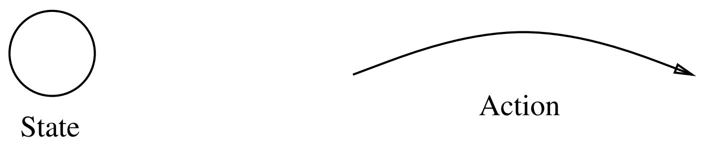


Figure 2.2 State and Action

The idea of the state transition diagrams stems from the observation of a feature that is common to all penetrations: all intruders start with limited access to a system with limited privileges (= Initial state). After performing some illicit actions they gain some previously unheld ability (= Final state). This is illustrated in Figure 2.3.

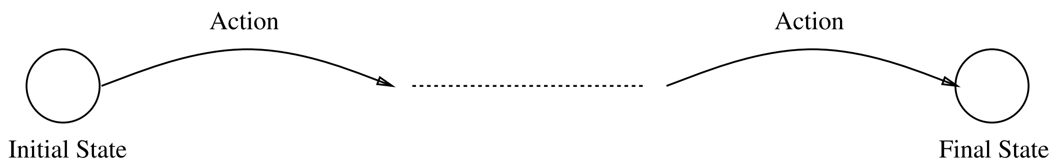


Figure 2.3 Initial and Final States

So, we can view a penetration as a sequence of actions that lead from an initial limited access state to a final compromised state. When we construct a state transition diagram we use only the key activities; that is those that make a state change on the system, that lead to the final state, and that best represent the penetration. It is easy to observe the first two, but the last is quite intuitive. In fact, there is no clear cut procedure that can define the construction of state transition diagrams, and it is possible that two different persons can come up with different state transition diagrams that represent the same penetration scenario. Which one is the best is difficult to answer.

Each state consists of one or more state assertions. If there are no state assertions for a given state, then it is called a NULL state. A NULL state can only occur at the beginning of a state transition diagram implying that there are no prerequisite state assertions to represent the penetration scenario. A NULL state cannot occur as an intermediate state of a diagram, simply because otherwise it would mean that the last action that lead to this state didn't make any state change on the system, which contradicts with our definition of state transition diagrams. The example given in the following section best illustrates the construction of state transition diagrams.

2.2.3 An Example Penetration and its State Transition Diagram

In this section we give an example penetration scenario from UNIX and go over the process of creating the state transition diagram that corresponds to it.

In the following example the target file, called *target* is a setuid shell script with the #!/bin/sh mechanism and is owned by root. (For information about setuid shell scripts, refer to Appendix). The file that is linked to target starts with a dash '-'.

```
% ln target -x
% -x
```

The steps of this process can be explained as follows.

Step-1. The attacker creates a hardlink starting with a dash '-' to root's setuid shell script containing the #!/bin/sh mechanism.

Step-2. The attacker executes '-x'.

Insight: Whenever a hardlink is created, a new directory entry is created with the target's original permissions and ownership information. The new entry uses the same inode as the target. The target can be identically accessed via any link to it. Also, executing a shell script containing the #!/bin/sh mechanism invokes a sub-shell. If the name of the shell [4] starts with a '-', this subshell becomes interactive, meaning that the user invoking the shell can talk to it. Since in this case the user is executing a setuid file owned by root, he/she receives an interactive shell with root privileges.

There are two steps involved in this penetration and each is necessary for the successful completion of the attack. First we can identify the signature actions: the first one corresponds to the 'CREATE' action among STAT's action types and the second is 'EXECUTE'. So far we have the incomplete diagram of Figure 2.4.

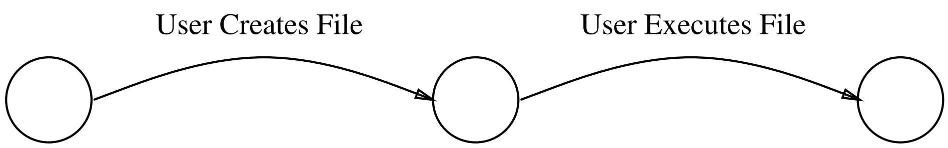


Figure 2.4 An Incomplete State Transition Diagram

Next we identify the state changes accomplished by these calls. There is no prerequisite state to execute the first signature action. The attacker doesn't need any special access to the system before the execution of the first command. Therefore, the first state can be identified as NULL. In the final state the attacker gains some previously unheld ability: he/she has an effective user id of root. So, the final state can be identified as:

```
euid (user) = root
```

To complete the diagram we should identify the intermediate state. In fact, we should include all the information that is used in the first step of the penetration explained above. The new filename should start with a dash, followed by any characters:

[4]Not all such names will work though. For more information, see chapter 5, "Testing Scenario 1."

```
name (file) = -*
```

The new file is hardlinked to a root owned setuid file. So, the new file's ownership should also indicate root, as given in the following state assertion.

```
owner (file) = root
```

The file must be a setuid shell script:

```
access (file, suid) = TRUE
shell_script (file) = TRUE
```

Finally, the file must be a link:

```
link (file) = TRUE
```

With these additions the state transition diagram is completed and it is shown in Figure 2.5.

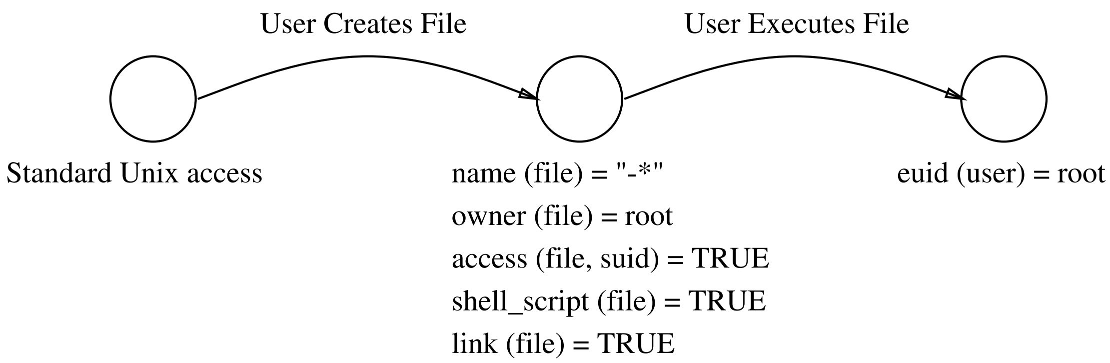


Figure 2.5 Final State Transition Diagram

The state assertions used in this section are meant to be descriptive rather than complete or accurate at this point. When we discuss the UNIX prototype implementation we give a complete list of necessary state assertions that may not have the same format as is presented here.

2.3 Components of STAT

The previous sections described how a penetration scenario can be represented using state transition diagrams. In this section, we see how STAT uses state transition diagrams to detect penetrations. We illustrate this by examining each component of STAT and by showing how these components interact with each other. STAT consists of the following components.

- The Preprocessor
- The Knowledge-base
  - The Fact-base
    - The Fact-base Initializer
    - The Fact-base Updater
  - The Rule-base
    - The State Description Table
    - The Signature Action Table
- The Inference Engine
- The Decision Engine

Except for the preprocessor, these components characterize a typical expert system. Martin and Oxman [Mart88] emphasize the modularity and independence of the expert system components in the following way. If an expert system component needs to be built or modified, it should not affect the other components. The major task of building an expert system will be the development of the knowledge-base. STAT provides modularity by designing each of these components separately and interfacing them together. The independency of the system is provided by the rule-base (a major part of the knowledge-base), which can be changed without interfering with the program code. Figure 2.6 illustrates the components of STAT. In the following subsections we look at each of these components.

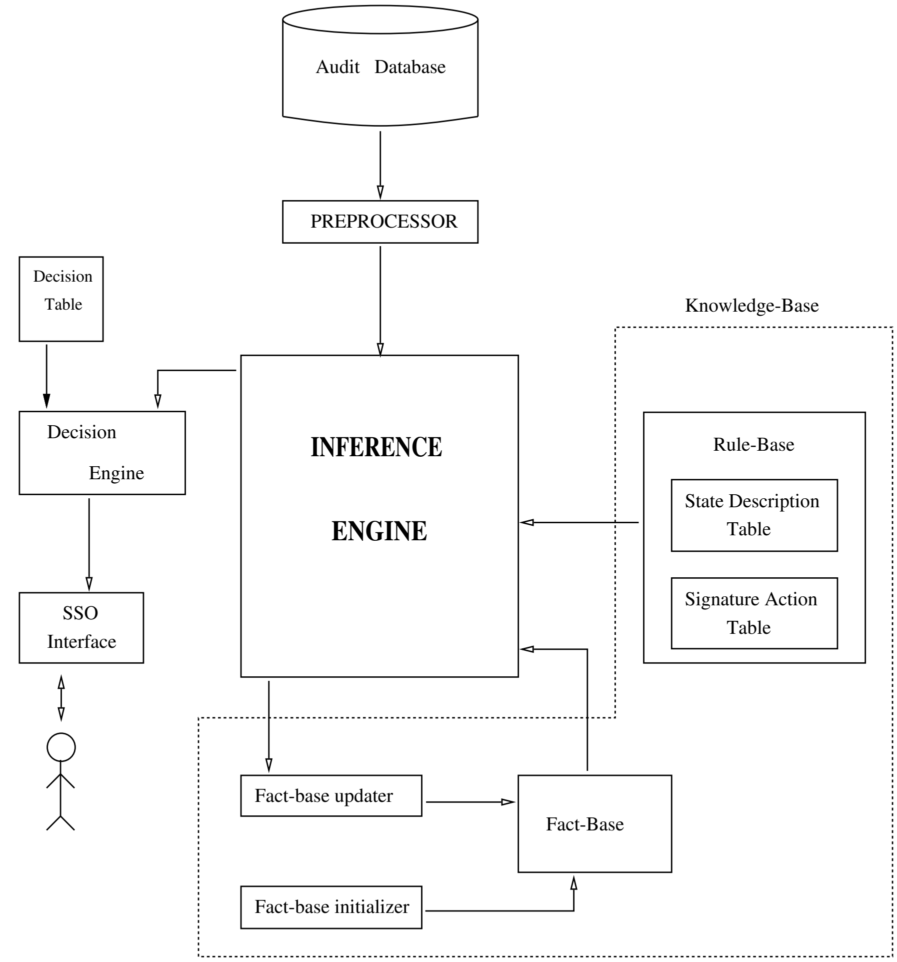


Figure 2.6 STAT Components

2.3.1 The Preprocessor

The preprocessor is responsible for reading, processing, and filtering the audit data to be used by the inference engine. The audit record format for STAT is defined by the six-tuple:

$$< \text{Subject Id, Subject Perm, Action, Object Id, Object Owner, Object Perm} >$$

- Subject Id : The unique identifier of the subject on whose behalf the audit record was generated.

- Subject Permissions : The access privileges of the subject responsible for the audit record, e.g., security level, effective user id, group membership, etc.
- Action : The action performed by the subject. Possible actions are: Read, Write, Create, Delete, Execute, Exit, Modify_Obj_Owner, Modify_Obj_Perm, Modify_Sub_Perm, and Session_Start.
- Object Id : The unique identifier or name of the object whose access was recorded.
- Object Owner : The owner of the object indicated within the previous field.
- Object Permissions : The access permissions associated with the object.

The preprocessor maps the event types of the audit system to the various action names defined above. If an event has no mapping, it is simply discarded. The events that indicate failure are also discarded, as they are expected to cause no state change on the system. [5] The objects of the target system are its files.

Each formatted audit record corresponds to at most one signature action. The fields in the audit record are used to detect the changes on the subject or object attributes.

2.3.2 The Knowledge-base

In [Fors84] Forsyth describes a knowledge-base as a collection of information structures for encoding expertise, which is usually elicited from a human specialist and reformulated as a collection of rules. Forsyth also distinguishes a knowledge-base from a database by the fact that a knowledge-base is more active and contains rules for deducing facts that are not stored explicitly. Expert systems should separate their knowledge-base from the inference engine and decision engine to provide modularity. In addition, the rule-base should be easily modifiable without having to change the inference engine structure.

[5] This is contrary to the necessary data for anomaly detection systems where a certain number of failures indicate suspicious actions.

STAT's knowledge-base consists of two major components: the fact-base contains information about the objects of the system, and the rule-base is the rule representation of the state transition diagrams. Figure 2.6 shows the knowledge-base components within the dotted lines. Having a rule-base as a separate component adds further modularity to STAT. This is particularly good since the penetration scenarios are system specific and often version specific.

2.3.2.1 The Fact-base

STAT's fact-base consists of filesets, each of which lists files that share some characteristics that make them vulnerable to certain attacks, e.g., a file-set can be characterized as: "the names of all files that are located in the /usr/spool/mail directory." Filesets increase the generality of the rule-base by enabling the rules to reference a set of files rather than one specific file. In [Porr91] Porras suggests eight different filesets for the prototype implementation (See Table 2.1).

Filesets	**Characteristics**
Restricted-read files	/dev/mem, /dev/kmem
Restricted-write files	/etc/*, /bin/*, /.*, /usr/etc/yp*, /usr/ucb/*, files referenced in /usr/lib/crontab
Fileset #1	All setuid/setgid scripts that contain the #!/bin/sh mechanism
Fileset #2	All binary executable files that are setuid/setgid enabled
Fileset #3	All setuid/setgid enabled files
Fileset #4	All designated mail files in /var/spool/mail
Fileset #5	All utilities authorized to reference restricted-read files
Fileset #6	All utilities authorized to reference restricted-write files

Table 2.1 Filesets in STAT and their characteristics

The fact-base of STAT is created by the fact-base initializer and maintained by the fact-base updater. The initializer creates the filesets for use by the inference engine. Whenever an audit record references a file from a fileset, the following actions are performed by the fact-base updater.

1. If the reference is a modification, the updater checks to see if the file still qualifies for membership in a fileset, and if necessary, removes or adds it to a fileset.
2. If the reference is a create, the updater checks to see if the new file qualifies for membership in a fileset, and if necessary, adds it to the fileset.
3. If the reference is a delete, the updater deletes the file from the filesets it currently belongs to.

2.3.2.2 The Rule-base

The rule-base defines the steps that are taken by the inference engine to infer new information based on the contents of the fact-base and the input data. The input used is the preprocessed audit records.

State transition diagrams provide a visual, easy-to-understand and easy-to-create representation of the state changes that occur during a penetration. However, they are not an efficient representation for storing such information in a computer. Therefore, STAT provides a structure, called the State Description Table to store them in a processable format. This table has a row for each penetration and a column for each state (one step) of a penetration. Figure 2.7 shows the first two rows of a state description table.

	S_1	S_2	S_3
P_1	name(file1) = "-*" owner(file1)<>user suid(file1) = TRUE shell_script(file1) = TRUE executable(file1) = TRUE	euid(user)<>user	
P_2	owner(file)=user not suid(file) name(file)= "/usr/spool/mail/*"	owner(file)=user suid(file) = TRUE name(file)= "/usr/spool/mail/*"	owner(file)<>user suid(file) = TRUE name(file)= "/usr/spool/mail/*"

Figure 2.7 A Partial State Description Table

Each rule in the rule-base is composed of the following fields.

< state_description   signature_action   [rule_dependence] >

State_description is the (row, column) coordinate in the state description table. *Signature_action* corresponds to the action that is anticipated after the states in state_description held. The *rule_dependence* field is optional, it is used to support permutable state transitions, which are not addressed in this thesis.

Each step of a state transition diagram corresponds to one rule in the rule-base, e.g., $\mathrm{RULE}_{j,i}$ (Step i of Penetration j) will look like this:

$$\mathrm{RULE}_{j,i} : \mathrm{SDT}(\mathrm{P}_j, \mathrm{S}_i) \quad \text{Sig\_Action}$$

The interpretation of this rule by the inference engine is explained in the next section.

2.3.3 The Inference Engine

The inference engine forms the heart of the control mechanism. It consists of a collection of algorithms that make use of the other components of an expert system in search of new information. The inference engine does not "know" what rules and facts should be or could be in the knowledge-base. For any given inference step, the inference engine blindly uses all the relevant rules and facts that are available to it at that time.

In [Mart88], Martin and Oxman list several problem solving activities that are supported by expert systems. STAT's activity is best described by *monitoring*, which is defined as follows.

> *"Monitoring involves observing and checking for some specific purpose. To monitor a system means to keep track of the system, to observe the behavior of the system and compare the observations with planned or designed behaviors. Monitoring systems usually have set limits within which the behavior of the system being monitored can vary without any action on the part of the monitoring system. However, if observed system behaviors exceed these limits, the monitoring system is usually designed to notify some system or operator."*

STAT's inference engine uses a forward chaining inference scheme [Mart88]. The forward-chaining scheme comes from the well-known logic rule called *modus ponens*. This can be described as follows.

```
p => q
p
------
q
```

The rule is given as "if p implies q and p holds, then we can infer that q holds."

STAT's inference engine uses a generalization of modus ponens. The inference engine starts with a set of facts and input, and fires rules based on this information to determine if a goal is met. Consider the rule example in the previous section, which gives the general format of all the rules.

$$\mathrm{RULE}_{j,i} : \mathrm{SDT}(\mathrm{P}_j,\mathrm{S}_i) \quad \text{Sig\_Action}$$

This rule expresses two things:

1. If $\mathrm{RULE}_{j,i-1}$ has fired and $\mathrm{SDT}(\mathrm{P}_j,\mathrm{S}_i)$ holds

   Then

   Sig_Action will be anticipated, and

2. If $\mathrm{RULE}_{j,i-1}$ has fired and Sig_Action occurs and as a result $\mathrm{SDT}(\mathrm{P}_j,\mathrm{S}_{i+1})$ holds

   Then

   $\mathrm{RULE}_{j,i}$ fires.

The inference engine uses the first one to determine which actions to anticipate and the second to keep track of the state transitions. We talk more about the inference engine algorithms when we discuss it specifically for USTAT. For now, all we need to know is that the inference engine interprets the rules in the rule-base and based on the available facts tries to infer new information for detecting compromises. In the next section we explain how the findings of the inference engine are passed to the SSO.

2.3.4 The Decision Engine

The basic responsibility of the decision engine is to send information to the SSO about how close a compromise is to being achieved or whether a compromise has been achieved. This gives the SSO the ability to monitor the progress of the inference engine.

For each state transition of each penetration there is a message kept in a structure called the *Decision Table*. This table adds further modularity to STAT by allowing the administrator to modify STAT's responses to each fired rule. The following is a list of possible actions that can be taken by the decision engine.

1. Generate messages for the SSO whenever a compromise is achieved.
2. Generate warning messages for the SSO whenever a rule fires.
3. Suggest possible actions to the SSO to preempt the next state transition and/or to recover from the damage, if any.
4. Take preemptive action, such as disabling the terminal connection that is being used for the penetration, logging out the attacker, etc.

In [Porr91] Porras suggests that the decision engine might also be configured to be more sensitive to certain subjects than to others and this could be combined with the operation of a profile-based anomaly detection component, which would increase the sensitivity of the decision engine to subjects whose activities generate a high number of anomaly reports. The decision engine can be implemented or configured in a variety of ways depending on the requirements. The modularity of STAT's design enables the decision engine to be highly complex, without interfering with the rest of the STAT components.

This page is intentionally left blank

Chapter 3

USTAT's Approach to Intrusion Detection

This page is intentionally left blank

3 USTAT's Approach to Intrusion Detection

Porras [Porr91] suggests an example prototype implementation for STAT. The target system chosen for the prototype is UNIX, which was chosen for the following reasons.

1. Availability of documentation.
2. Vulnerability of UNIX, and the availability of security flaws and penetration scenarios.
3. Availability of UNIX for implementation and test in an academic environment.

The high-level discussion of STAT's components gives us some insight into the feasibility of such an implementation. The work reported in this thesis takes all of the effort done for STAT one step further by implementing a UNIX prototype of STAT. This prototype of STAT is referred to as USTAT: UNIX State Transition Analysis Tool. The suggestions given in [Porr91] about the prototype were invaluable, yet some changes to the ideas and even to the design of the components of STAT were inevitable due to unforeseen constraints. In the remainder of this document we present a thorough discussion of the implementation process. We also point out the difficulties that have been encountered and that resulted in many design changes.

3.1 Major Design Changes From STAT to USTAT

Although STAT was designed to be implemented on any target system, and its documentation [Porr91] was very coherent, modifications were unavoidable. Most of these modifications were made to the suggested prototype implementation and a few were made to the original design of STAT. The following paragraphs summarize these modifications.

The preprocessor was modified to obtain all the information that is needed for the analysis. More fields were added to the audit record format to provide additional information, but the essential idea of having a $<$ Subject, Action, Object $>$ triplet was preserved.

In order to use the appropriate state assertions and signature actions we need a thorough understanding of the target operating system and its underlying audit mechanism. Sometimes it is hard to come up with the signature action that is expected to correspond to a certain command or system call and therefore it becomes puzzling when the audit collection does not work as expected. For instance, *readlink(2)* is the system call used to obtain the target file's name for a symbolic link file. However, we don't see an audit record indicating a readlink call when a file is accessed via its symbolic link. As a result, the signature actions and state assertions needed to be modified. The modifications to the signature actions are explained in detail in Section 4.1.2 of Chapter 4, and changes to state assertions are explained in Section 4.2.2 of Chapter 4.

The fact-base was changed to provide easy-to-manage and easy-to-create filesets that are more specific to the target system's operating system. Some filesets were removed and new filesets were added. The final version of filesets can be found in Section 4.2.1 of Chapter 4.

The part of the inference engine algorithm that was designed to detect cooperative attacks was changed to a better and easy-to-implement algorithm. However, the implementation of permutable state transitions was postponed until future versions.

It is difficult for the preprocessor to provide the inference engine with the audit records that indicate an action from a fixed set of action names. In USTAT's case, it was necessary to add a different action name to the possible list of signature actions since it could not be replaced with the others available and since there were penetration scenarios that include this particular action.

3.2 Issues Regarding the Application Domain

In this section we address some issues that affected and characterized the implementation of USTAT.

3.2.1 Real-Time Issues

USTAT is implemented to be run in real-time. The audit data is read and processed as soon as it is written in the audit files. The major issue is whether USTAT will be fast enough to keep up with the audit records when the user load is high. The results of the tests focusing on this issue are given in Chapter 5.

Although USTAT is implemented to be run in real-time, it would be prefer-able if it could be run in batch-mode. Batch-mode analysis, has however, a requirement that the state assertions should be evaluated only using the information in the audit records. For instance, consider the state assertion:

```
owner (file) <> user
```

which means that the owner of the file last accessed is not the user who is the subject of the last audit record. The ownership information could be obtained by issuing a stat(2v) call or else the audit record could contain the ownership information for the file being accessed. The first method may not be the correct choice for the following reasons. If USTAT is being run at a remote site that doesn't have direct access to the target system, all it can depend on is the audit data. That is, it cannot make a stat(2v) call for the target file. Even if USTAT is run on the target system or has access to the target system, there are race conditions to be considered. The result of the stat(2v) call may not reflect the state of that file at the time the last audit record was recorded. It will reflect the state of the file at the time the stat(2v) call is being made. Meanwhile, the ownership of the file might have been changed. Figure 3.1 illustrates this possibility. The audit record being parsed has time value of (1). There is a chown call at time (2). The analysis tool is operating at time (3), which is current time and it makes the stat(2v) call at this time.

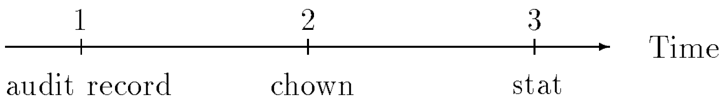


Figure 3.1 The Order of Events

Therefore, the ownership should be part of the audit record. More generally, after initializing the fact-base the audit data should be the only input source to USTAT.

For USTAT all state assertions but one can be evaluated by using the information contained in the audit records. The *shell_script* state assertion cannot be evaluated using the audit records. This assertion checks whether the file is a shell script with the #!/bin/sh mechanism. This check can only be made by opening the file and checking whether it contains "#!/bin/sh" in its first line. This dependency on the current state of the file system might cause some discrepancies especially if the analysis is done in batch mode.

3.2.2 Expert System Issues

The following key ideas of AI and expert systems are used in USTAT.

- *New ways to represent knowledge:*

  Knowledge refers to a collection of information about a specific topic that is organized to be useful. AI researchers mostly focus on the verbal and graphical aspects of knowledge rather than the more mathematical aspects. USTAT represents the knowledge of penetration scenarios in a graphical way using state transition diagrams. This provides a better understanding of scenarios and a more intuitive way of illustrating the knowledge.

- *Separating knowledge from inference and control:*

  In conventional programming, knowledge about a problem and procedures for manipulating that knowledge to solve the problem are mixed together. AI techniques separate the knowledge in a program from the procedures that manipulate the knowledge. USTAT's knowledge-base consists of different modules that can be updated without affecting the inference mechanisms. In fact, new knowledge (i.e. new penetration scenarios) can be added to the knowledge-base without modifying any line of source code.

[Harm88] lists categories of expert system tools as follows.

1. *Inductive tools*: These generate rules from examples.
2. *Simple rule-based tools*: These use if-then rules to represent knowledge. They are simpler than structured rule-based tools.
3. *Structured rule-based tools*: They offer context trees, multiple instantiation, confidence factors, and more powerful editors.
4. *Hybrid tools*: These tools use object-oriented programming techniques to represent elements of each problem that the system will work on as objects.

5. *Domain-specific tools*: Designed to be used only to develop expert systems for a particular domain and could incorporate any of the techniques listed above.

With these definitions we can characterize the expert system used by USTAT as a *domain-specific semi-structured rule-based tool.* USTAT is designed to work for a particular domain (intrusion detection), it uses simple if-then rules for its rule-base, but multi-instantiation of rules is also possible, since any rule can be fired by different users simultaneously.

3.2.3 Source Code Language Considerations

USTAT is written in the ANSI C that is available in SunOS 4.1.1. Embedding some expert system tools or high-level languages that provide AI methodologies, such as Prolog, were also considered. However, these possibilities were disregarded because of insufficient information on such tools and the uncertainty of how they would be interfaced with C. For this prototype real-time application, C was the right choice because of its reliability, portability, high-speed and availability under UNIX.

3.2.4 Target Environment

USTAT is designed to be run under SunOS 4.1.1 with the C2-BSM Module. Currently, USTAT can handle a single audit trail generated by one host computer. Auditing more machines on a network with USTAT requires some considerations. These are discussed in Section 6.3.1 of Chapter 6.

The audit files are owned by the user *audit*. This user can read and manage all the audit files. Therefore it is possible to run USTAT by the audit user if not by the root. Although USTAT is implemented to rely primarily on audit data, there are certain cases where it is desirable to have root privileges. One of the state assertions, which is used to check whether a file is a shell script containing the #!/bin/sh mechanism, can only be evaluated by opening the file. The audit user may not be able to do this since he/she does not have permission to all the files on the system. Also, as part of the fact-base, all hardlinks on the filesystem need to be determined. This cannot be accomplished completely without having root permissions. Although USTAT will run normally without root privileges, the functionality of it will be limited because of lack of data for the fact-base and for some state assertions.

During the implementation of USTAT many aspects of UNIX came across, some of which had significant impact on the resulting program. These aspects are discussed in Appendix under the header " Remarks about UNIX internals."

3.2.5 Other Issues

There are many things to be considered for an intrusion detection system and some particularly for USTAT. Not all these issues were addressed in this thesis. In particular, we did not address:

- Security of USTAT, especially the ways for an attacker to become invisible to the audit collection mechanism.
- Employment of more than one CPU to distribute the load. One CPU could be used to collect the audit data, another to preprocess the audit data and another to run the analysis part of USTAT. This would increase the overall efficiency.
- Archiving audit data, before or after being processed by the preprocessor. Although audit data takes too much disk space, in certain cases it might be desirable to archive the audit data for the analysis of other intrusion detection tools or for manual analysis.
- Knowledge Acquisition Subsystem. [Porr91] also suggested a knowledge acquisition subsystem, which would be a learning component of STAT. USTAT does not incorporate this component. If implemented it would be extremely useful to find new penetration scenarios.

Chapter 4

The Components of USTAT

This page is intentionally left blank

4 The Components of USTAT

USTAT consists of the following major components.

- The Preprocessor
- The Knowledge-base
  - The Fact-base
  - The Rule-base
- The Inference Engine
- The Decision Engine

This chapter discusses each component in detail.

4.1 The Preprocessor

In this section we first look at the audit collection system of SunOS 4.1.1. Then we describe USTAT's audit record preprocessor.

4.1.1 Audit Collection and BSM

4.1.1.1 Summary of Auditing

Operating systems with an NCSC [6] evaluation of C2 or higher are required to provide audit collection mechanisms. Each such system has its own auditing facilities with possibly different features, but with at least one common goal in mind: providing a trail that could be useful in case the system's security is breached.

There are many reasons to use auditing on a system. For instance, by monitoring user activities the individual accountability of users can be assured. Potential security breaches can also be detected by using the audit trails. Finally, auditing has a deterring effect; if users know that their actions are audited they are less likely to attempt illicit activities.

Auditing itself does not prevent security breaches. Yet, it provides a vast amount of data that can be analyzed to find how the system was breached and who was responsible. This data contains information about nearly all activities on the system. The amount of data collected depends on the particular auditing mechanism, on the amount of activities on the system and on the settings of audit flags - if any - that determine which events are to be collected.

Auditing is a costly process. An auditing system needs to have a massive amount of storage space to be able to record all activities. Depending on the configuration, megabytes of data can be collected in just minutes. One might ask whether it is worthwhile committing so much computer resources. The answer to this depends on the security requirements of the target system. If one cannot afford to have a single intruder on the system, then it should be made as secure as possible; auditing the activities on the system is one part of this process.

[6] National Computer Security Center

4.1.1.2 SunOS 4.1.1 C2-BSM

The target system that is used for this project is SunOS 4.1.1 with its C2 Basic Security Module (BSM). In the remainder of this thesis we refer to SunOS 4.1.1 C2 Basic Security Module as BSM. The BSM is designed to be compliant with the NCSC requirements for a system at the C2 classification, but at the time this document was prepared there was no information available about the evaluation of the BSM package.

The criteria for the C2 classification are given in the Orange Book [Tcse85]. The BSM provides improved security features for the standard UNIX operating systems. The following add-on features are part of the BSM: shadow password files, object reuse, device allocation/deallocation and audit collection Shadow password files add further protection to encrypted passwords by hiding them and making them readable by only root. This way, the passwords are protected against encryption attacks. For object reuse and device allocation/deallocation, the BSM prevents one user from accessing a device while it is being used by another, it also clears the data from a device when it is no longer in use. This prevents a user from accessing sensitive residual data. Our primary interest is in the BSM audit collection, which is discussed in detail in the next section.

4.1.1.5 The Auditing Process

To explain the auditing process we need to define the following terms.

- Audit Trail: A time-ordered sequence of actions that are audited on the target system. As long as the BSM is active, there exists a single audit trail for the host machine that runs the BSM.
- Audit File: Each audit trail consists of one or more audit files.
- Audit Record: An audit file consists of audit records, each of which describes the occurrence of a single audited event. Audit records are generated by users and their processes, whenever they make system calls or execute remote or local commands.
- Audit Token: Each audit record consists of a number of audit tokens that describe the fields within an audit record.

- Audit Events: Audit events are actions that are to be audited. They are also called auditable events.
- Audit Classes: Audit events that share a common feature make up an audit class. Each audit event belongs to one or more audit classes. The actions to be audited can be selected in the granularity of audit classes, but not audit events.
- Audit Preselection: This determines which audit records are to be recorded in the audit trail. The system administrator can specify the audit flags that determine the classes of events to be audited.
- Audit Postselection: By using reduction and selection tools, certain types of events in audit trails can be saved and interpreted while others are discarded.
- Audit Flags: Each event class has a short name for use in audit flags. These flags are used in two places: The *audit_control* file and the *passwd_adjunct* file. An audit flag is an indication of what to do with an event. The format is "[+|−] `class`", where a plus means to audit successful events of class, and a minus means to audit failed events. The absence of either means to audit both successful and failed events.

Figure 4.1 illustrates the hierarchy of audit trails, audit files and audit records.

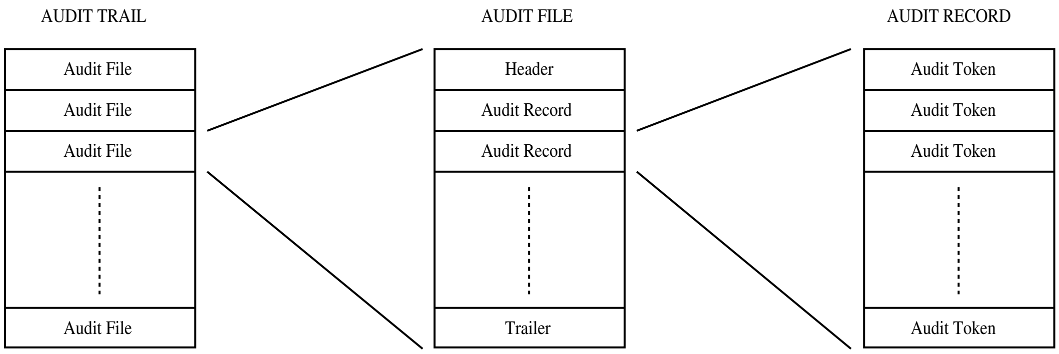


Figure 4.1 Audit Trails, Audit Files and Audit Records

There is a special user-id, audit, that is created during the installation of BSM. This is the only user-id whose actions are never audited. It is the only user, therefore, who can do something if the system came to a halt because all auditing file systems are full. The user audit does not have to have root privileges. In that way, most of the audit-related actions can still be performed by the user audit, while some still needs to be performed by root.

Audit Classes

Table 4.1 lists the different type of audit classes along with their descriptions and examples.

Name	Long Name	Description	Example
dr	data read	Open for reading, etc.	stat(2)
dw	data write	Write or modification of data	kill(2)
dc	data create	Creation or deletion of objects	link(2)
da	data access change	Change in object access	chmod(2)
lo	login logout	Login, logout, creation by *at*	login(1)
sc	spooler control	Spooler control command	lpr(1)
ad	administrative	Normal administrative operation	passwd(1)
p0	minor privilege	Privileged operation	revoke(2)
p1	major privilege	Unusual privileged operation	mount(2)
as	device assign	Device allocation	allocate(1)
au	audit user	The audituser system call	audituser(2)
other	other	Events not in any other class	su(1)

Table 4.1 List of Event Classes

Among these audit classes, dr and dw generate the highest volume of audit records. A complete list of programs or system/library calls that are audited by these flags can be found in [C2-b91].

Control files used by the audit system

Besides the audit collection data files there are two special files used by the audit system.

`/etc/security/audit/audit_control`, and
`/etc/security/audit/audit_data`

The audit_control file: The audit_control file is read by the audit daemon to determine how to audit. The following is a sample audit_control file for the machine *vladimir*.

```
dir:/etc/security/audit/vladimir/files
dir:/tmp
flags:lo,dr
minfree:10
```

The first line indicates the first file system, where the audit data should be collected. If the first file system becomes full, the second file system will be used, which is indicated on the second line. The *machine-wide audit state* is indicated on the third line as auditing all new logins and all data reads. The *minfree* level is set at 10 percent. This means that when the current audit file system gets 90 percent full, a mail message is sent to the designated audit administrator, and the audit trail is switched to the next file system. The audit_control file is created during the installation of BSM with the values and parameters specified by the administrator.

The audit_data file: The following is a sample for the contents of the *audit_data* file.

86:/etc/security/audit/vladimir/files/19920605154739.not_terminated.vladimir

This indicates the full path of the audit data file in which current audit information is being added. The contents of this file are changed automatically by the audit system whenever the audit file that is currently in use gets changed.

Changing the audit state

Occasionally, it becomes necessary to change what is being audited on a given machine or for a particular user. The *system audit state* defines the audit classes to be audited for a given machine. The *user audit state* defines the event classes to be audited for a given user.

To change the system audit state, the *flags* line should be edited in the audit_control file. After changing this line, the audit daemon needs to be instructed to reread the audit_control file. This is done by executing the following command.

```
audit -s
```

The login sessions that were already open at the time this command is issued will not be affected. The new audit information will be collected only from login sessions opened after the execution of this command.

The user audit state can be changed in two ways: Permanent (takes effect at login time) or immediate (takes effect after login time, and only for the duration of the session). For a permanent change the appropriate entry in the *passwd* file should be edited, whereas for an immediate change the *audit* program should be executed with *-u* option. Details of these can be found in [C2-b91].

Audit data files and audit records

This section describes the structure of the audit data files and the structure of the audit records contained in these files.

Audit data files: The audit data files are created in the file system specified in the audit_control file. The current audit file name and path are given in the audit_data file. The contents of the audit_data file might look like the following.

/etc/security/audit/vladimir/files/19920605154739.not_terminated.vladimir

---

The first portion of the filename *(19920605154739)* indicates the date and time when this file was created. It is used as an arbitration mechanism. For instance, the filename above was created on *June 05, 1992*, at *15:47:39*. The second portion of the filename (not_terminated) denotes that this file is currently being used for the collection of audit records and not yet terminated. If the file is terminated as the result of an *audit -s* command or when auditing switches to the next file system then this portion indicates the date and time this file was closed. Finally the last portion (vladimir) is the machine name for which the audit data is collected. The structure of each audit file is given in Figure 4.2. Each audit file starts with a *header* record, which is followed by audit records. The file is terminated by a *trailer* record. The header and trailer records provide a chain through the audit data files. Each audit data file has a unique header indicating the preceding file name, and similarly a unique trailer indicating the succeeding filename. There are two obvious exceptions to this: the first audit file (which is created when the C2 security system is first installed) does not have a preceding filename and the last audit file (which is currently being used by the audit system) does not have a succeeding filename. Hence it is not yet terminated. The fields in the header and trailer token are given in Figure 4.3.

Audit Record Structure: Each audit data file consists of many audit records, the amount of which depends on the system audit state and the number and load of the users logged on the system. Each audit record corresponds to the occurrence of a single audited event. Each event generates several audit tokens that make up the audit record. Figure 4.2 illustrates the audit file and audit record structure. Each audit record contains a possibly different combination of the tokens. Header, process, return and trailer tokens are always present in an audit record. The tokens that are marked with '*' are the ones used by USTAT. Figure 4.3 illustrates the structures of tokens. Each audit event type has a possibly different sequence of tokens. The number of tokens may vary from one event type to another.

AUDIT FILE

File Header	
Audit record	variable length
Audit record	variable length
⋮	
File Trailer	

AUDIT RECORD

Header	*	First token of an audit record
Process	*	Information taken from the per process audit data structure
[Argument]	*	Arguments from a system call
[Attribute]	*	Attributes of a vnode
[Data]		Allows tha auditing of data in various formats and types
[In_addr]		Internet address
[Ip]		The first 20 bytes of the IP header
[Ipc_perm]		IPC access structure
[Ipc]		IPC id
[Iport]		Internet port ID
[Path]	*	Current root, current working directory and absolute path
[Text]		Text string
[Groups]		Group entries from the processes credential
[Opaque]		Just a string of bytes
Return	*	Result of the request (e.g. the system call)
Trailer		Final token of an audit record

Figure 4.2 BSM Audit File and Audit Record Structure

HEADER TOKEN

token ID		1 char
record size		1 short
event type	*	1 short
time of queue	*	2 longs

PROCESS TOKEN

token ID		1 char
audit ID	*	1 short
user ID	*	1 short
real user ID		1 short
real group ID	*	1 short
process ID	*	1 short

ARGUMENT TOKEN

token ID		1 char
argument ID		1 char
argument value	*	1 long
string length		1 short
text		X chars

RETURN TOKEN

token ID		1 char
user error		1 char
return value	*	1 long

PATH TOKEN

token ID		1 char
size of root		1 short
current root		X chars
size of dir		1 short
current dir		Y chars
size of path		1 short
path argument	*	Z chars

ATTRIBUTE TOKEN

token ID		1 char
vnode mode	*	1 short
vnode uid	*	1 short
vnode gid	*	1 short
vnode fsid	*	1 long
vnode nodeid	*	1 long
vnode rdev	*	1 short

TRAILER TOKEN

token ID		1 char
magic number		1 short
record size		1 short

Figure 4.3   BSM Audit Record Tokens

The following is the beginning portion of the octal dump of an audit file. This might be useful to describe how the audit records can be extracted for use by other programs, such as USTAT. For readability, two outputs of octal dump (*od -c and od -a*) are manually merged to show only the text fields in ASCII format.

```
0000000  021 052 057 214 233 000 012 071 227 000 127   /   e   t   c   /
0000020    s   e   c   u   r   i   t   y   /   a   u   d   i   t   /   a
0000040    u   d   i   t   _   t   r   a   i   l   s   /   v   l   a   d
0000060    i   m   i   r   /   f   i   l   e   s   /   1   9   9   2   0
0000100    6   0   5   1   5   4   7   1   1   .   1   9   9   2   0   6
0000120    0   5   1   5   4   7   3   9   .   v   l   a   d   i   m   i
0000140    r 000 022 000 171 000 110 052 057 214 242 000 000 047 020 043
0000160  000 002   / 000 000 031   /   u   s   r   /   g   s   l   /   h
0000200    o   m   e   /   g   r   a   d   /   k   o   r   a   l 000 000
0000220  043   /   /   /   u   s   r   /   g   s   l   /   h   o   m   e
0000240    /   g   r   a   d   /   k   o   r   a   l   /   .   l   o   g
0000260    o   u   t 000 061 201 240 025 176 000 030 000 000 202 006 000
0000300  001 076 320 261 100 046 025 176 025 176 025 176 000 030 000 242
0000320  047 000 000 000 000 000 023 261 005 000 171 022 000 065 000 031
```

Table 4.2 gives the analysis of the above data by incorporating the information of the structure of individual tokens. This analysis is done throughout the run of USTAT to preprocess and filter the audit records.

BYTE	MEANING
021	This record starts a new file
(8 bytes)	2 longs denote the time when this file is created.
000 127	The length of the following string in octal form.
(87 bytes)	The name of the previous audit file in the audit trail.
000	String is terminated.
022	Starts a new record.
000 171	Size of this record.
000 110	Event type, open for read.
(8 bytes)	Time of the event.
043	Type of the next token (PATH).
000 002	Size of root.
/	Current root.
000	Current root string terminated.
000 031	Size of directory.
(25 bytes)	Current directory = /usr/gsl/home/grad/koral
000	Current directory terminated.
000 043	Size of path.
(35 bytes)	Path argument = ///usr/gsl/home/grad/koral/.logout
000	Path argument terminated.
061	Type of the next token (ATTRIBUTE).
201 240	Mode of the object.
025 176	Owner id.
000 030	Group owner id.
000 000 202 006	File system id.
000 001 076 320	Node id (inode).
261 100	Device id.
046	Type of the next token (PROCESS).
025 176	Audited user.
025 176	Effective user-id.
025 175	Real user-id.
000 030	Real group-id.
000 242	Process-id.
047	Type of the next token (RETURN).
000	User error value.
000 000 000 000	Return value.
023	Type of the next token (TRAILER).
261 005	Magic number.
000 171	Length of this record.
022	Starts a new record and so on.

Table 4.2 Analysis of two audit records

BSM provides a program called *praudit* that generates an output of audit trails in human readable form. The following is the output of the praudit command for the same audit file and for the first two records in the file.

```
file,Fri Jun  5 08:47:39 1992, + 670103 msec,
/etc/security/audit/vladimir/files/19920605154711.19920605154739.vladimir
header,121,open(2): read,Fri Jun  5 08:47:46 1992, + 10000 msec
path,/,/usr/gsl/home/grad/koral,/usr/gsl/home/grad/koral/.logout
attribute,100640,koral,grad,33286,81616,45376
process,koral,koral,koral,grad,162
return,Error 0,0
trailer,121
```

As we see, praudit provides a readable format of audit records, but the outrageous amount of audit records manifests the necessity for an automated analysis tool (such as USTAT).

4.1.2 The Audit Record Preprocessor

4.1.2.1 The Need for a Preprocessor

The audit record preprocessor is responsible for reading, filtering, mapping and finally passing the BSM audit records to the inference engine in the required format. There is no set standard for the audit record data structure. Each target system may have its own audit collection system and its own audit data structure. For instance, in the previous section we analyzed a specific audit record structure: SunOS 4.1.1 C2-BSM audit record structure. One goal of STAT was to provide system independence. This can be accomplished by requiring a special audit record format and employing an audit record preprocessor. If we need to port the intrusion detection system to another target system, we will not need to make major changes to the code of the inference engine, but we will need to change the preprocessor.

In this section we give the prerequisite format of audit records for USTAT and then give a detailed description of how the audit records of BSM are mapped onto those of USTAT. We refer to these records as BSM audit records and USTAT audit records, respectively.

4.1.2.2 Requirements for the System Audit State

For USTAT to be fully functional, the system audit state has to meet some minimum criteria. That is, all of the events that affect USTAT's analysis should be collected so that we don't miss any event that might change the fact-base or that might be a signature action.

The BSM can be configured to collect successful or unsuccessful events or both. This can be done in the granularity of the event classes. Therefore we should inform the BSM to collect all the event classes of the events that USTAT is interested in. These classes are data read, data write, data create and data access change. Setting the BSM to collect these is quite easy. We need to edit the audit_control file and have the line that starts with flags to indicate at least: +dr, +dw, +dc, +da. This means, all successful data reads, data writes, create-deletes and access mode changes have to be collected by the audit mechanism. The failed events and the event classes "lo, sc, ad, p0, p1, as, au, other" need not be collected. The functionality of USTAT will not be affected if these need to be collected for other analysis tools; however, the performance may be degraded depending on the amount of data that is collected through these event classes.

4.1.2.3 USTAT Audit Record Structure

Not all of the fields in USTAT audit record structure are needed for the detection of an intrusion. Some fields are included to provide added information about the identity of the attacker, the identity of the object being attacked and the time and process number of the action. These extra fields in the structure may assist the SSO in further analysis of the penetration.

The USTAT audit record structure is defined by the triple:

$$<\text{SUBJECT, ACTION, OBJECT}>$$

meaning "SUBJECT performs the ACTION on the OBJECT."

Each of these attributes contains further fields that are used to reveal as much information as possible about the particular attribute. Figure 4.4 gives the USTAT audit record structure.

SUBJECT

The SUBJECT is identified by the triple:

<Real User Id, Effective User Id, Group Id>

The Real User Id information is required by USTAT for individual accountability. It is necessary to track a user's actions in the audit trail. Some of the consecutive signature actions might need to be performed by the same user, in which case the user id's need to be compared. The Effective User Id is required by USTAT to track the changes in user's effective user id. The Group Id provides added information, it is not currently used, but might be used in future USTAT releases.

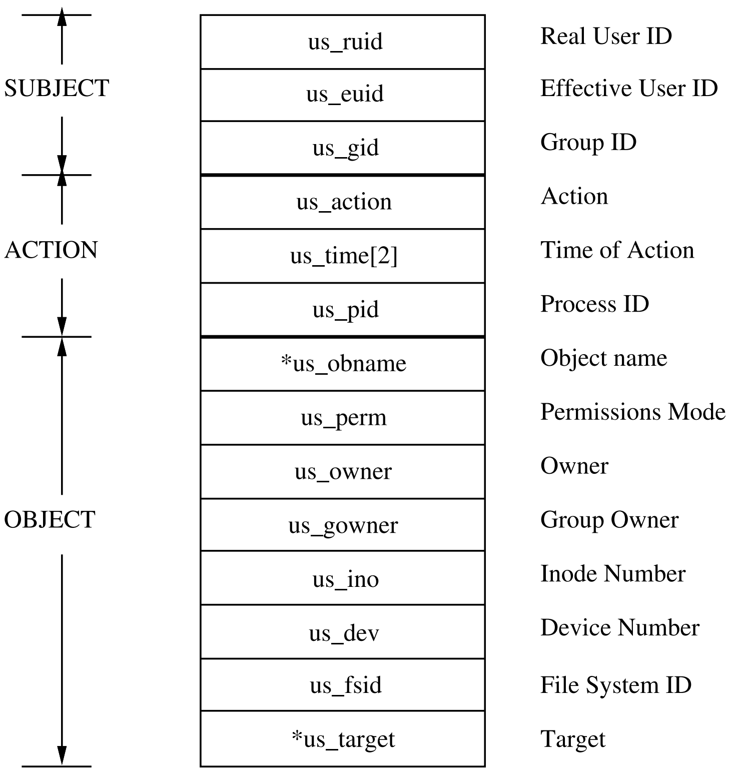


Figure 4.4 USTAT Audit Record Structure

All of the attributes in the SUBJECT triple can be obtained directly from the BSM audit record structure. The Real User Id is obtained from the audit id field of the process token. The real user id field in the BSM audit record is not used since it changes when the user issues an "su" command. However, the audit user id field in the BSM does not change throughout a login session. The Effective User Id and Group Id are also trivially obtained from the user id and real group id fields of the process token, respectively.

ACTION

The ACTION is identified by the triple:

<Action, Time, Process Id>

The ACTION is the main focus of the entire state transition analysis. USTAT's expected auditable actions consist of those actions that manipulate system objects and modify user privileges. The Action field can contain one or more of the following values: Read, Write, Create, Delete, Execute, Exit, Modify_Owner, Modify_Perm, Rename, and Hardlink.

The Action is obtained from the event type field of the Header token. Action corresponds to a list of actions connected by logical OR. Most of the time, the Action field can be directly obtained from the Event Type field, e.g., if the Event Type is dc, dw, then the Action is $Create|Write$. There are some exceptions to this, which are explained later in this chapter. The Time and Process Id fields provide valuable information for the SSO for further analysis. They are obtained from the time of queue of the Header token and process id of the Process token, respectively.

OBJECT

The OBJECT designates which object is being accessed by the SUBJECT. The OBJECT is identified by the eight tuple:

<Object Name, Permissions, Owner, Group Owner, Inode #, Device #, File System Id, Target>

The Object Name is the name of the file identified with its full path. The Object Name can be obtained from the path argument of the Path token in the

BSM audit record. The Owner, Group Owner and Permissions are required by USTAT to be able to detect changes in the file's permissions. The Inode Number, Device Number and File System Id contain valuable information for the accuracy of the fact-base. Two files having different object names can actually be the same physical file. All these fields can trivially be obtained from their corresponding fields in the Attribute token of the BSM audit record.

The Target field is effective only if the action is Hardlink or Rename. It is obvious that the permission bits, owner information and device information of a target will be the same for a file being renamed or hardlinked. Therefore, we don't need to repeat these fields. The Target is obtained from the Path Argument of the second Path token in the BSM audit record.

4.1.2.4 The Operation of the Audit Preprocessor

In this section we give the details of the operation of the audit record preprocessor. We first look at the mapping from the BSM audit record structure to the USTAT audit record structure. Then we go through the filtering process. Finally we briefly look at the program operation.

Audit record mapping from BSM to USTAT

Figure 4.5 shows the mapping of BSM audit records onto the USTAT audit records. Some extra processing needs to be done to fine tune this mapping. The extra considerations are the following.

- In a Modify_Perm action, the value of Permissions is obtained from the argument value of the Argument token, since mode of the Attribute token gives the value of permissions prior to the action.
- Similarly, in a Modify_Owner action, the value of Owner is obtained from the argument value of the Argument token, since uid of the Attribute token gives the value of owner prior to the action.
- Event Type of the Header token and return value of the Return token are the only ways to determine whether a BSM audit record needs to be completely filtered. We discuss the filtering process in the next section.

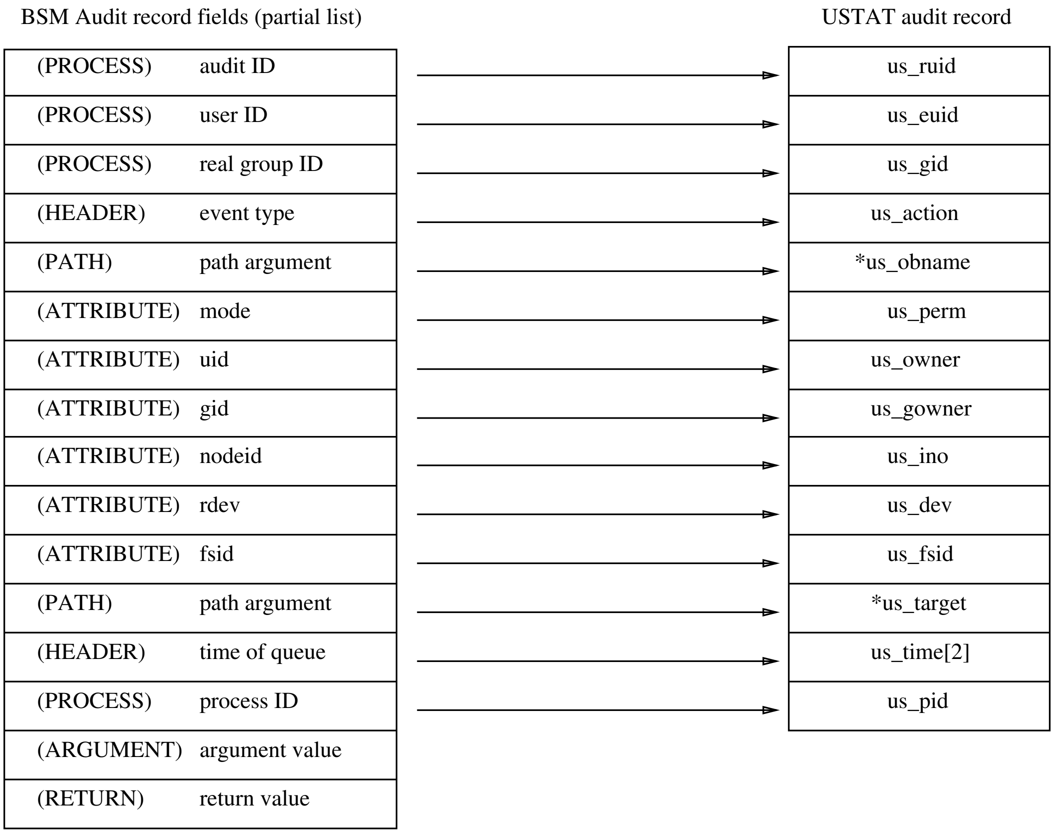


Figure 4.5 Mapping from BSM Audit Record to USTAT Audit Record

Filtering out the BSM audit records

BSM events used by USTAT: There are 239 different events that are auditable by BSM. Out of these, 28 events are used by the preprocessor and mapped onto 10 different USTAT actions. The inference engine operates using these 10 action types. Table 4.3 lists the 10 different actions of USTAT along with the BSM event types that are mapped onto them. The USTAT event types listed in the first column are covered by the following BSM event classes: Read, write, create/delete and access mode change. Note that these are the event classes that we have to specify for the system audit state (see Section 4.1.1.3, header "Changing the Audit State"). By focusing on certain event types we already filtered out 211 event types of BSM.

USTAT Action	BSM Event Types
Read	open_r, open_rc, open_rtc, open_rwc, open_rwtc, open_rt, open_rw, open_rwt
Write	truncate, ftruncate, creat, open_rwc, open_rwtc, open_rw, open_rwt, open_rt, open_rtc, open_w, open_wt, open_wc, open_wtc
Create	mkdir, creat, open_rc, open_rtc, open_rwc, open_rwtc, open_wc, open_wtc, mknod
Delete	rmdir, unlink
Execute	exec, execve
Exit	exit
Modify_Owner	chown, fchown
Modify_Perm	chmod, fchmod
Rename	rename
Hardlink	link

Table 4.3 USTAT Actions vs. BSM Event Types

Return value: The return value field of the Return token indicates the result of the event. The preprocessor also takes this return value into account. It filters out all the BSM records that indicate a Return Value of -1. This value means that the call made by the user was not finished successfully. It did not make any change to the system attributes and hence it cannot cause a state transition. One might say that reading from a file cannot make any change on the system attributes, so it should not be audited. However, a read action does change the state of the system. At least, it changes some memory locations by copying a non-volatile storage area into the volatile memory area. Therefore the preprocessor keeps the read events unless the result of the event is unsuccessful.

Note: Unlike USTAT, Statistical Anomaly Detection Systems use the return field to detect browsers who perform abnormally high numbers of unsuccessful attempts or external attackers who repeatedly fail to pass logon authentication.

More Records to Filter Out: By inspecting the BSM audit records, we also realized that a group of four audit records is recorded continuously upon execution of each command. To save some bandwidth we eliminated these records. The following records indicate read accesses to some system files.

```
READ /usr/lib/ld.so
READ /dev/zero
READ /etc/ld.so.cache
READ /usr/lib/libc.so.1.6
```

As the read events of these files are very unlikely to indicate any suspicious activity, we see no risk in discarding these events.

Short Description of Program Operation

Following is a short algorithm for the operation of the program. The static variable *audit_state* is used to keep track of the current operation of the preprocessor. When the preprocessor first starts running it is in the START state. When the preprocessor needs to open an audit data file the state indicates NEXT_FILE. While reading audit records from a file the state is READ_FILE. Finally, while closing an audit file the state indicates CLOSE_FILE.

```
While TRUE do begin
   If Audit_state = START then begin
      Allocate memory for audit record
      Allocate memory for the ptr of size BLOCK_SIZE
      Audit_state = NEXT_FILE
   Endif
   If Audit_State = NEXT_FILE then begin
      Open audit file
      Read a block from audit file to ptr
      Advance ptr to the beginning of the first record
      Obtain preceding filename
      Audit_State = READ_FILE
   Endif
   If Audit_State = READ_FILE then begin
      Attempt to read a block from audit file to ptr
      If successful then begin
         If ptr indicates AUT_OTHER_FILE then
```

```
18             Audit_State = CLOSE_FILE
19          Else begin
20             Call filter_it to filter the next record
21             If record passed through the filter then
22                Return (audit_record)
23          Endelse
24       Endif
25       Else
26          Reopen audit file
27    Endif
28    If Audit_State = CLOSE_FILE then begin
29       Obtain next audit file name
30       Close current audit file
31       Audit_State = NEXT_FILE
32    Endif
33 Endwhile
```

The preprocessor does not return to the calling routine unless it finds a record that needs to be audited by USTAT. Usually, the preprocessor will need to close and reopen the same audit file, since the audit system keeps appending records to the audit file. The preprocessor reopens the file in order to read the records that were written to the file between the time it was last opened and the time EOF is reached. When the file is terminated by the audit daemon with a trailer record, the preprocessor reads the name of the next file from the trailer record and opens the next file. If there is no activity in the system that results in adding of audit records then the preprocessor is idle and waits for the next record to be available. Line 20 in the above algorithm is the main audit processing part. It performs the mapping and filtering of audit records as defined in the previous section.

Some implementation considerations

In this section we look at some details of the implementation of the preprocessor. Although the task of the preprocessor looks quite simple, its performance, completeness, and consistency depends on many factors.

Action Types: In determining the action types for USTAT, many implementation aspects of the whole program have been considered. Although we tried to make the inference engine algorithms as independent as possible from the underlying system, we had go back and forth between different modules of USTAT to end up in a complete set of action types where each action could be disjoint from the others. Also, this set had to contain all of the actions that are needed to represent the penetration scenarios on the target system. With these guidelines it was easy to come up with the first eight action types: Read, Write, Create, Delete, Execute, Exit, Modify_Owner and Modify_Perm. There were three more specific commands (system calls) that needed to be considered: mv (rename), ln (hardlink), and ln -s (symlink). Each of these is discussed in turn.

- Rename involves two objects, one *source* and one *target*. We could have considered this call as the creation of the target and deletion of the source. That would save us one action type and we could use other action types that were already listed in our set. However, by doing that we would be neglecting the fact that the target and source are actually the same physical files. Without explicit information the inference engine had no way of detecting that these two objects were the same. Therefore, we decided to keep the rename call as an action type and added a target field to the USTAT audit record.

- Hardlink also has two objects involved. Again we could have mapped these onto two USTAT actions: read the source and create the target. The inference engine, however, had no way of determining whether the target was the same as the source. Also we realized that it was essential to keep track of all the hardlinks on the target system. The need for this and details of maintaining that information is explained later in Section 4.2.1 of Chapter 4. Therefore, the hardlink system call is kept as an action type.

- Symlink is another exception. It is similar to hardlink except that it creates another inode on the system. By scrutinizing the BSM audit collection mechanism, we realized that whenever a file is accessed via a symbolic link, the target file's identity is recorded, but not the symbolic link. The only time when a symbolic link is recorded is when the symlink system call (ln -s command) is performed. Therefore, there is no need to keep information about symlinks on the system. As a result, symlink is not audited by USTAT.

Collection of exit(2v): The collection of the exit(2v) call without some additional information is not enough for the proper operation of USTAT. USTAT also needs the identity of the object that was exited. Therefore the preprocessor keeps track of all the *exec* and *execve* system calls with their arguments. Whenever a process is exited, it checks to see whether the process number exists in the list of programs that are being executed. If it does, then the preprocessor removes the process number from the list, and records the name of the program as the object name of an EXIT action.

User sessions: The first design of STAT suggested keeping track of user sessions. Signing on and off the system were also important events to be audited. During the design and implementation of USTAT, we modified this concept. We decided to follow the objects of the system instead of the subjects. In this case, we would not need the logon and logout events. We still record the subject's identity, but we don't care if they are still logged in. As a very simple example we can consider a user executing commands after logging out by making use of the at(1) utility. Thus, there is no need to follow user sessions. In this way, penetration scenarios can span multiple sessions, or multiple users can cooperate in performing one penetration scenario. These variations can still be detected by USTAT without keeping track of user sessions.

Examples of USTAT audit records

In this section we look at some UNIX commands and their corresponding audit records in the USTAT audit record format. This should give the reader an idea of how the audit records corresponding to user commands are received by USTAT. The examples also illustrate how very similar commands can result in very different audit records.

These examples are generated by using the variations of the first state transition diagram from Section 4.2.2 of Chapter 4.

Executing a shell script: The user executes a shell script called *foo*, which is owned by another user. foo consists of a single line: "sleep 5" The first USTAT audit record corresponding to the execution of foo is the following.

```
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
```

```
PID / Time:               979 / 120000 - Fri Aug 21 12:12:49 1992
ACTION:                   READ
OBJECT:                   /tmp/foo
TARGET:                   (null)
OWNER - GOWNER - PERM:    9   0   -rwxrwxrwx
DEVICE - INODE - FSID:    130   4   1800
```

To save space, for the remainder of the records we use a shorter format indicating only PID, ACTION and OBJECT. We indicate other fields whenever needed. So, the complete set of audit records corresponding to this execution is the following.

PID	ACTION	OBJECT
979	READ	/tmp/foo
979	READ	/tmp/foo
979	EXEC	/usr/bin/sh
979	READ	/tmp/foo
980	EXEC	/usr/bin/sleep
980	EXIT	/usr/bin/sleep
979	EXIT	/usr/bin/sh

Executing a shell script with the #!/bin/sh mechanism: This time, the user executes a shell script called *foo*, which is owned by another user. foo consists of the following lines.

```
#!/bin/sh
sleep 5
```

The audit records corresponding to the execution of foo look like the following.

PID	ACTION	OBJECT
987	EXEC	/tmp/foo
987	READ	/tmp/foo
988	EXEC	/usr/bin/sleep
988	EXIT	/usr/bin/sleep
987	EXIT	/tmp/foo

Executing a setuid shell script with the #!/bin/sh mechanism: In this case, the user attempts to execute the shell script interactively. This can be accomplished by creating a hardlink that starts with a dash to the target file. For instance, the user can give the following commands.

```
% ln foo -x
% -x
```

The USTAT audit records corresponding to these commands are the following.

PID	ACTION	OBJECT
1022	EXEC	/usr/bin/ln
1022	HARDLINK	/tmp/-x
1022	EXIT	/usr/bin/ln
1023	EXEC	/tmp/-x

The line with HARDLINK above has /tmp/foo as the target. After executing -x, user gains an interactive shell with file owner's privileges, since /tmp/foo was a setuid file. (For details, see Section 4.2.2, Scenario 1).

Executing a setuid shell script with the #!/bin/sh mechanism through a symbolic link: The only difference from the previous example is that the user creates a symbolic link to the target, instead of a hardlink. For instance, the user executes the following commands.

```
% ln -s foo -a
% -a
```

The USTAT audit records corresponding to these commands are the following.

PID	ACTION	OBJECT
1024	EXEC	/usr/bin/ln
1024	EXIT	/usr/bin/ln
1025	EXEC	/tmp/foo

Again, after executing -a, user gains an interactive shell with file owner's privileges. As we see, the name of the symbolic link is not used in the last EXEC action, the name of the target file is used instead. This causes some problems for USTAT and they are discussed in Section 4.2.2 of Chapter 4.

4.1.2.5 Past Problems

There were two different auditing systems involved throughout this project: C2 auditing system for SunOS 4.1.1 and C2-BSM patch for SunOS 4.1.1. The second system differed substantially from the first one. The first version of the preprocessor [Ilgu91] was written using the first version of C2 auditing system of SunOS 4.1.1. This version of auditing had many deficiencies in providing a complete audit trail for intrusion analysis. Here is a short list of the deficiencies that existed in the previous version of the C2 auditing system.

- Delete event corresponding to the *rm* command was collected via the unlink(2V) system call. However, the audit system failed to record the owner and permissions information of a deleted file.
- Logout event was not collected.
- Successful termination of chmod, fchmod, chown, fchown system calls result in an attribute change. This attribute is either a UID or GID or permissions mode of the object. The auditing system did not record the values of those attributes prior to the modification. Hence it was impossible to determine the previous attributes by looking at the SunOS audit records.
- EXIT(2v) system call was not being recorded. In this case we were unable to identify whether a program or command was still in execution.
- Some system calls refer to a file descriptor rather than to a filename. These calls are: fchmod, fchown, and ftruncate. These calls were recorded using the file descriptor information instead of the filename. Unfortunately, determining the filename corresponding to a file descriptor is not a trivial task. Unlike a filename, a file descriptor is dynamic and it is contained in the process' u area as long as the process that opened that file descriptor is active. One solution that might overcome this problem was the following. To obtain a file descriptor, a file should be opened. This could be done by either OPEN(2V) which requires a filepath and returns a file descriptor, or FOPEN(3V) which requires a filepath and returns a pointer to the stream. In either case there would have been an open system call recorded in the audit data along with a filepath. Therefore, the preprocessor could translate the file descriptors into filenames by keeping track of the open/close system calls of a process. This part of the preprocessor would have to be quite intelligent.

> Because, it not only has to pay attention to the open/close system call, but it also has to keep track of the fork() system calls, since a child process inherits all the file descriptors from the parent process. The problem arose, since fork(2V) was not collected by the audit system. According to the Orange Book, a C2 system has to record objects' identities. However, the auditing system did not record objects consistently.

Fortunately, these problems no longer exist in the release of C2-BSM Patch of SunOS 4.1.1. A detailed explanation of the previous release and the USTAT preprocessor that was written for the previous release can be found in [Ilgu91].

4.1.2.6 Current Problems

The C2-BSM was quite successful in solving our previous problems. During the implementation of USTAT it has been realized that some of the previous problems are no longer of any concern. For example, by the nature of USTAT, we don't need the previous ownership or previous permissions mode of an object. All we care about is whether the modification to those attributes has been done successfully and with proper authorization. Therefore, although the previous values of those attributes are provided by BSM, they are not being used by USTAT, and hence the preprocessor does not need to collect these values.

However, other problems appeared with the new release. First of all, the objects' pathnames are not recorded consistently. Most of them contain repeated slashes at the beginning or worse than this some of them look like either */etc/../usr/share* or */usr/ucb//clear*, which do not uniquely identify those objects. Also, some keystrokes are not recorded by BSM although they make changes to the system attributes. For example, if Ctrl-C is pressed after executing a command, but before the termination of the command, exit(2v) for that process is not recorded, nor is any kill. Finally, accesses to symbolic links are recorded as accesses to the target. There is no way of distinguishing the direct access to an object from an access through a symbolic link.

The first one of these problems is solved by adding a filename correcting routine to the preprocessor. The second one remains as a problem. The last one causes some limitations, but it does not affect the functionality.

4.1.2.7 Concluding Remarks

The auditing system used in this project was installed on a single machine. Hence the audit data was collected from a single machine and recorded on a single file system. Future versions of the preprocessor should be capable of processing audit records from multiple machines and multiple file systems. In this case the records to be passed for state transition analysis will have to be time ordered. The fact that audit records are coming from different hosts should be transparent to the inference engine.

4.2 The Knowledge-base

USTAT's knowledge-base consists of two major components: the fact-base and the rule-base In the next two sections we present these two components in detail.

4.2.1 The Fact-base

4.2.1.1 Facts Used in the Fact-base

USTAT's fact-base consists of groups of files or directories, which we call filesets, that share certain characteristics in a typical UNIX filesystem. For example, all setuid/setgid files can be distinguished from other files in the filesystem, thereby creating a fileset.

In identifying each set for the prototype implementation, we tried to answer the following basic questions. Do these files share a common characteristic? Can these characteristics be identified easily and efficiently? Is it easy to maintain the fileset at run-time? Is the existence of the fileset significant for one or more penetration scenarios? Is there any other way of representing the same penetration scenario without referring to the fileset? The last question signifies the desire of having a minimum number of filesets; otherwise, the amount of processing time to maintain these filesets could degrade USTAT's performance.

A candidate fileset is added to USTAT's fact-base if the answer to all of the above questions but the last is YES. As a result, we made some changes to the fact-base that was suggested in [Porr91] by removing some filesets and adding others. The fact-base that was suggested in [Porr91] has been given in Table 2.1 in Section 2.3 of Chapter 2. The filesets that were removed from the suggested fact-base are filesets 1, 2, 3, and 4. The first three are removed because the characteristics of the files in these sets refer to the file attributes that can be obtained easily from audit records and that can be added to the states of a state transition diagram as simple state assertions. The fourth one has a single characteristic, which states that the files should be located in /var/spool/mail. Since this is a special case, it can easily be added as a state assertion to the state transition diagram. The final version of the fact-base, which is being used by USTAT is given in Table 4.4

FILESET	**CHARACTERISTICS**
Fileset #1	Restricted read files
Fileset #2	Restricted write setup files
Fileset #3	Files authorized to read Fileset #1
Fileset #4	Files authorized to write Fileset #2
Fileset #5	Non-writable system executables
RWSD	Non-writable system directories
HL	System hardlink information

Table 4.4 USTAT Filesets

The first five of these sets and the directory set are used directly in state transition diagrams, whereas the last one is used by the inference engine to identify variations of scenarios through references by hardlinks. In addition to the filesets that were removed from the suggested fact-base of [Porr91], some changes have been made: *NWSD* (Non-writable system directories) and *HL* are added to the fact-base, and the *restricted write files* are split into two other sets: *restricted write system setup files* and *non-writable system executables*. It is important that in case of symbolic links all files in these filesets should contain only target filenames, since accesses to symlinks are not recorded by the audit collection mechanism.

In the remainder of this section we explain each fileset in turn. We also explain how these filesets are stored and represented in memory and we list the members of them where appropriate.

Fileset #1, restricted read files

These files should not be accessed via regular utilities, as they contain sensitive information that if read by an ordinary user could violate the system security. The files that are subjected to this violation are /dev/mem, /dev/kmem, and /dev/mbmem. /dev/mem is an image of the physical memory of the computer, and /dev/kmem is an image of the kernel virtual memory of the system. In some UNIX systems these files are left readable by everyone. Discolo in [Disc85] illustrates how plaintext passwords can be obtained from /dev/kmem.

In recognition of this violation and the potential for similar ones, USTAT uses Fileset #1. These files should only be read by certain system files that are identified in Fileset #3. Files in Fileset #1 are recorded in a special text file called *rrf.set* (**r**estricted - **r**ead **f**iles). This file consists of comment lines and filepaths. Comment lines start with #. Blank lines are ignored, all other lines are recognized as filepaths. Filepaths should start with a slash ('/'). Filepaths may contain an '*' at the end as a wildcard. However, an '*' doesn't match with a '/' in filepaths. For instance, /usr/local/* does not match with the file /usr/local/etc/mapsys. The following is the current rrf.set.

```
#               RESTRICTED READ FILES

/dev/mem
/dev/mbmem
/dev/kmem

# add here any more files that should be restricted
# to be read only by certain programs
```

Since rrf.set is a text file it can be modified easily by the SSO prior to the execution of USTAT.

Fileset #2, restricted write setup files

Similar to the first set, these files should be denied write access except by the files in Fileset #4. An example to this is /etc/passwd. This file should not be overwritten by any program, except the passwd and yppasswd commands. These programs provide legitimate write access to the /etc/passwd file. Any other write to this file except by the root should be identified as a security violation.

Files in this fileset are kept in a special text file called *rwsf.set* (**r**estricted **w**rite **s**etup **f**iles). The format of this file is identical to the one for Fileset #1. The following is the current rwsf.set.

```
#        RESTRICTED WRITE SETUP FILES

# these are from COPS
/etc/*
/.*

# these are from SunOS - Administering Security
/usr/local/*
/var/spool/cron/crontabs/root

# restricted read files better be here
/dev/mem
/dev/mbmem
/dev/kmem

# add here any more system or public directories
```

Since rwsf.set is a text file it can be modified easily by the SSO prior to the execution of USTAT.

Fileset #3, files authorized to read Fileset #1

This set consists of the files that are authorized to read files in Fileset #1.

We should be very careful in creating this fileset. The meaning of "authorized" does not include "being able to." It means that if foo is a member of this set then foo needs read access to some member(s) of Fileset #1 for proper execution. For instance, although cat(1) can be used to display the contents of any file as far as permission bits allow, it should not be defined as authorized to read any file in Fileset #1.

It is impractical to automatically determine files in Fileset #3. It needs a good understanding of all system commands and the files that are accessed by them. Therefore this fileset should be created manually. Depending on the files in rrf.set the SSO should be able to determine the files in Fileset #3. Then the SSO will enter those filenames into a text file called FSET3. This process is explained in detail in Section 4.2.1.2 of Chapter 4.

Fileset #4, files authorized to write Fileset #2

This set consists of those files that are authorized to write on files in Fileset #2. The process of creating this fileset is identical to the one for Fileset #3 and it is explained in Section 4.2.1.2 of Chapter 4. The SSO will enter the filenames for this fileset in a text file called FSET4.

Again, the term "authorization" literally means needing write access to a file in Fileset #2 for proper operation. For instance, vi(1) can be used to change the contents of /etc/passwd file. But, it is not authorized to write on the passwd file unless it is used by the root. In contrast, the passwd command has legitimate write access to the /etc/passwd file, can be used by any user, and therefore it should be included in Fileset #4. Figure 4.6 illustrates the relation between first four filesets.

Fileset #5, non-writable system executables

These files consist of publicly accessible, executable system files, which are common subjects to Trojan Horse attacks. These files should not be deleted because of denial of service problem, nor should they be overwritten (except by the system administrator) because of a possible Trojan horse implantation.

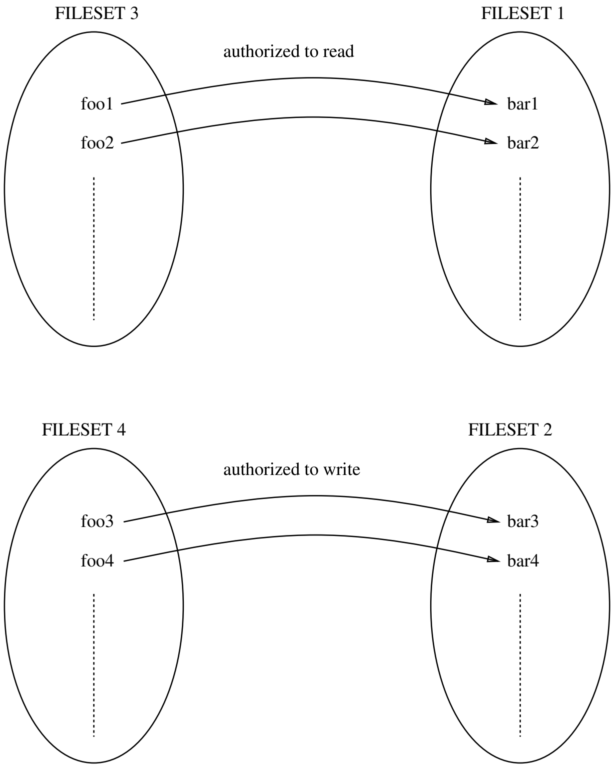


Figure 4.6 Relation between Filesets 1, 2, 3 and 4

Files in Fileset #5 are entered in a special text file called *nwse.set* (**n**on-**w**ritable **s**ystem **e**xecutables). The structure of this file is the same as rrf.set and rwsf.set. The following are the current contents of nwse.set.

```
#        NON-WRITABLE SYSTEM EXECUTABLES

# these are from COPS
/etc/*
/bin/*
/.*
/usr/etc/yp*
/usr/ucb/*

# these are from SunOS - Administering Security
```

```
/vmunix
/usr/local/*
/usr/local/bin/*

# add here any more system or public directories
```

RWSD, Non-writable system directories

The idea of these directories is similar to the idea of non-writable system executables. These directories usually consist of publicly executable files and therefore they are subject to Trojan Horse attacks. If somebody creates a fake ls program in one of these directories, it is quite possible that in the victim's path, the name of the target directory comes before the directory where actual ls is located. Therefore all these directories should be denied write access unless the access is done by the root.

The names of these directories are kept in a special text file called *nwsd.set* (**n**on-**w**ritable **s**ystem **d**irectories). The format of this file is similar to rrf.set and rwsf.set above except that the directory names may not contain an '*'. Current contents of nwsd.set are as follows.

```
#               NON-WRITABLE SYSTEM DIRECTORIES

# These directories usually exist in the search path of
# users' setup files.  If for some reason they become
# writable, a malicious user can implant a Trojan horse
# in these directories (such as another ls program).
# It is also a good practice to alias commonly used
# binaries to their full pathnames,
# e.g., alias ls /bin/ls

# According to the SunOS - Administering Security,
# the following directories should deny write permission
#    to group and others.
/
```

```
/var
/usr/bin
/dev
/etc/security
/usr/spool
/etc
/usr/etc
/usr/kvm
/usr
/usr/lib

# Add here any more directory names
# that should deny write permission.
```

HL, system hardlink information

In UNIX one physical file may have a number of pathnames associated with it. The link count of a file refers to the number of different pathnames a physical file has on a filesystem. Processes can access the file by any of these pathnames. For in-depth information about hardlinks refer to Appendix - Remarks About Unix Internals.

USTAT's fact-base keeps information about all hardlinks on the target system for the following reasons. In analyzing penetration scenarios, we noticed that variations of the scenarios can easily be accomplished by using different filenames at different steps of the penetration, while still referring to the same physical file. In this case the inference engine would fail in firing the rule of a penetration scenario, since the object in the next step was not identical to the object that it was looking for. In recognition of this fact we had two possible solutions.

1. Identify files by using their device and inode information rather than by their pathnames. In this case hardlinks will be the same (device, inode) pair, since they refer to the same physical file.

2. Keep the hardlink information of the filesystem as defined above. Whenever a file is accessed with hardlinks to it, check the hardlink informa-

tion to see whether there was any previous action that was done via the hardlinks to it and that fired a rule for some penetration scenario.

The first method might seem easier and more consistent. However, it has one major drawback that could not be overlooked. Some penetration scenarios refer to a certain file or a group of files, such as /var/spool/mail/*. The state assertions for such scenarios cannot make use of the (device, inode) pair for the following reasons. It is a costly process to determine all (device, inode) pairs for a group of files, such as for /var/spool/mail/*. Also, the (device, inode) information is dynamic. If a file in /var/spool/mail gets deleted and then gets re-created, it will have the same pathname, but not necessarily the same (device, inode) information. Therefore, the state assertion in such a scenario has to use the filepath instead of (device, inode) pair. This makes it infeasible to identify files by their (device, inode) information. As a result, the fact-base needs to keep track of all hardlinks on the filesystem as in the second solution presented above.

This set has a different structure from any of the other sets. The files in this set are not recorded in a text file. They are created directly in memory during the initialization phase of USTAT. The structure of the hardlink information is illustrated in Figure 4.7.

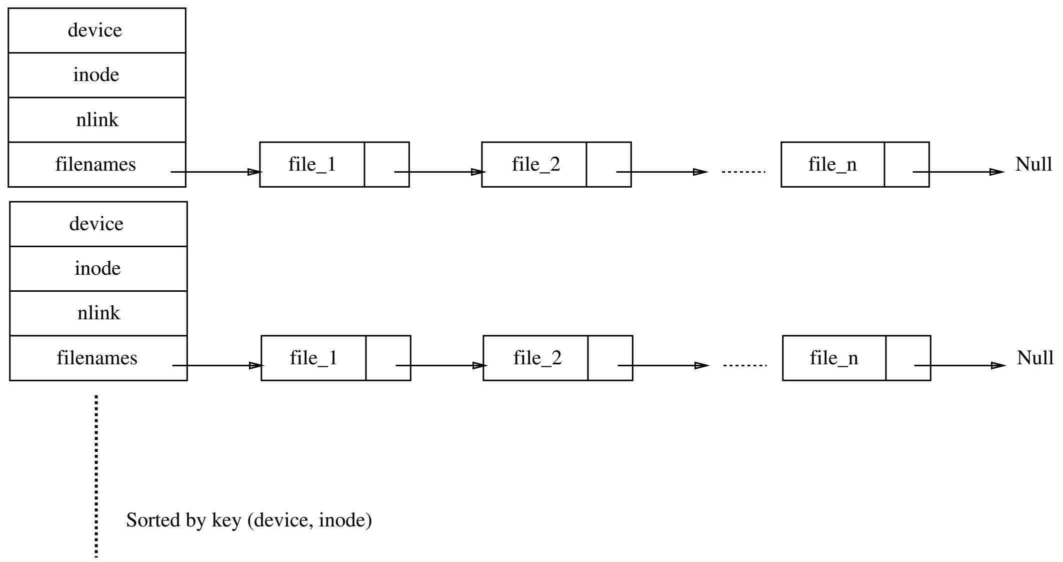


Figure 4.7 Hardlink Information Data Structure

The files are grouped in a one dimensional sorted array, where each item is a list of files that are hardlinked together and the array is sorted by the composite key of device and inode information. The creation and maintenance of the hardlink information adds some burden to the algorithms of the inference engine and the fact-base. Nevertheless, they are necessary for the sound implementation of this prototype.

Now that we identified the necessity of hardlink information for USTAT, one might ask why we don't use similar information for symbolic links. A close observation of the audit records revealed that accesses to files via symbolic links are recorded as accesses to the symbolic links; therefore, there is no use keeping that information in the fact-base.

4.2.1.2 The Fact-base Initializer

The Fact-base Initializer Module of USTAT consists of a collection of routines and some manual processes to create the filesets and to read them into memory. Because of the complexity of filesets, especially because of the difficulty of identifying the contents of some filesets, the fact-base initializer module consists of two major sections: the *pre-initialization* phase, which is performed before running USTAT, and the *post-initialization* phase, which is performed when USTAT is invoked. The pre-initialization phase needs to be performed before USTAT is run. This phase consists of three steps, two of which require manual processing by the SSO.

1. Make sure that entries in rrf.set, rwsf.set, nwse.set and nwsd.set are ready, or otherwise make necessary modifications to these files, following the guidelines that are explained in the previous section.

2. Run the program called *pre-init*, which reads and processes the first three files in the previous step. The execution of this program creates three new text files consisting of the full pathnames of the members of the first three files above and these are called FSET1, FSET2, and FSET5 respectively.

3. In this step, the members of Fileset #3 and Fileset #4 need to be determined, which correspond to files FSET3 and FSET4 respectively. Although USTAT comes with these filesets by default, depending on the needs of the system, the contents of these may need to be fine-tuned and verified. One source that can be examined in search of these filesets is the

manpages. The manpages contain information about which files access which. With this observation, USTAT provides a small tool to speed up the process of determining these filesets. This tool, called *mangrep* goes through the manual pages that are entered in the file *mandirs* and tries to extract the FILES section from the manpages. The result is written to an output file called *mangrep.out*, which can than be analyzed by the SSO with the purpose of finding the files that read/write the files in filesets 1 and 2. As a result of this step, files of FSET3 and FSET4 are ready.

The post-initialization phase is done within USTAT, as one of the first processes after invoking USTAT and consists of the following steps.

1. Sort FSET3 and FSET4, and eliminate repeated filenames. Output files are called FSET3.final and FSET4.final respectively.
2. Read the contents of rrf.set, rwsf.set, FSET3.final, FSET4.final, nwse.set, nwsd.set into memory.
3. Create the hardlink information by making a complete pass on the filesystem. This step is optional.

Fileset #1, Fileset #2 and Fileset #5 and non-writable system directories are read into memory as they existed in files *.set. Rather than having all the files in memory, the fact-base only keeps the generic pathnames in memory thereby saving space.

Fileset #3 and Fileset #4 are read into memory from files FSET3.final and FSET4.final respectively. Here, actual filenames are read into memory.

The names of the filesets, their corresponding physical files, and memory structures are listed in Table 4.5.

4.2.1.3 The Fact-base Updater

Unlike the fact-base initializer routines and manual procedures, which are processed only once, the fact-base updater routines are continuously being called during the execution of USTAT. The fact-base updater consists of those routines that maintain the fact-base for the consistent operation of the inference engine.

FILESET	PHYSICAL FILES	MEMORY STRUCTURE
Fileset #1	rrf.set, FSET1	Array of pathnames in rrf.set
Fileset #2	rwsf.set, FSET2	Array of pathnames in rwsf.set
Fileset #3	FSET3, FSET3.final	Array of filenames in FSET3.final
Fileset #4	FSET4, FSET4.final	Array of filenames in FSET4.final
Fileset #5	nwse.set	Array of pathnames in nwse.set
RWSD	nwsd.set	Array of directory names in nwsd.set
HL	(none)	Array of hardlinked filename lists

Table 4.5 Filesets, their corresponding physical files and memory structures

The only part that needs to be maintained is the hardlink information. The reason is filesets 1, 2 and 5 consist of generic pathnames rather than specific filenames. The change to files in these filesets does not affect the representation of these filesets in memory, e.g., /etc/* is included in the non-writable system executables, as indicated in nwse.set FSET5. Assume that a new executable program is added to the /etc directory. By definition of FSET5, any filename that matches with /etc/* is in this fileset. To decide whether a file is in a given fileset, USTAT uses this matching technique rather than keeping all filenames in memory and performing a search. Therefore, the maintenance of the filesets except for the hardlink information is not necessary for the proper execution of USTAT. However, the maintenance of hardlink information is necessary, because hardlinks may continuously be created on the filesystem and, given a filename there is no practical way of determining what other filenames correspond to the same physical file.

Each time an audit record is passed to the inference engine, the same record is passed to the fact-base updater routine to see if it affects the hardlink information and if it does, the hardlink information is modified accordingly. The following is the algorithm for this update routine.

```
FACT-BASE-UPDATE
Input: audit_record
Output: None

If audit_record implies a HARDLINK action
   If audit_record.target is in hardlinks
      Add audit_record.object to hardlinks
   Else begin
      Add audit_record.object to hardlinks
      Add audit_record.target to hardlinks
   Endelse
If audit_record implies a DELETE action
   If audit_record.object is in hardlinks
      Delete audit_record.object from hardlinks
If audit_record implies a RENAME action
   If audit_record.obname is in hardlinks
      Rename audit_record.obname to
      audit_record.target in hardlinks
```

4.2.2 The Rule-base

4.2.2.1 State Transition Diagrams

State transition diagrams provide a graphical representation of penetration scenarios and their effects on the system states. Obviously these diagrams are not stored in a graphical format to be used by the inference engine. Instead they are stored in two text files referred to as the State Description Table and the Signature Action Table, which store the state assertions and signature actions, respectively. These two tables form the second major component of USTAT's knowledge-base: the rule-base.

In this section we present USTAT's state transition diagrams, we analyze each diagram in turn and then we give the format of the two tables that store the state transition diagrams. Some state assertions presented here differ from the ones that were suggested in [Porr91], some being formatted differently and some being new state assertions. Also, during the implementation of the rule-base, we realized that it would be unnecessary to start from an initial state, since all initial states in our finalized state transition diagrams indicated a NULL value. That is, each penetration scenario requires an initial state with some limited privileges, but that state does not need to be added to our state transition diagrams. Therefore, all state transition diagrams start with a signature action rather than an initial state. This signature action acts like a trigger to the start of a particular scenario.

Before presenting the scenarios and their corresponding state transition diagrams we give a brief overview of state assertions. In the state transition diagrams, each state consists of one or more state assertions that are connected to each other. Each state assertion consists of one function name and zero or more arguments. The evaluation of a state assertion results in a *true* or *false* value. The *not* keyword in front of a state assertion negates the result. A short description of each type of state assertion follows.

1. **name** (*file_var*) = *file_name*
   Evaluates true if the file_var matches the filename given in the right-hand side.

2. **fullname** (*file_var*) = *full_path*
   Evaluates true if the file_var record matches the pathname given in the right-hand side.

3. **owner** (*file_var*) = *user_id*
   Evaluates true if the owner of *file_var* is the *user_id*.

4. **member** (*file_set, file_var*)
   Evaluates true if *file_var* is a member of the *file_set*.

5. **euid** = *user_id*
   Evaluates true if the effective user id of the subject of the audit record being processed equals the *user_id*.

6. **gid** = *group_id*
   Evaluates true if the group id of the subject of the audit record being processed equals the *group_id*.

7. **permitted** (*perm, file_var*)
   Evaluates true if the permission bit given as *perm* is set in *file_var*'s permission bits.

8. **located** (RWDS, *file_var*)
   Evaluates true, if *file_var* is located in any of the directories listed in the file *nwds.set*.

9. **same_user**
   Evaluates true if the subjects of the last two signature actions are the same.

10. **same_pid**
    Evaluates true if the process id's of the last two signature actions are the same.

11. **shell_script** (*file_var*)
    Evaluates true if the *file_var* is a *shell_script* with the #!/bin/sh mechanism.

A complete description of the State Description Table and the state assertions is given later in this chapter.

Scenario 1

The first scenario that we look at exploits a flaw in the shell mechanism. The attack consists of two steps:

Step 1.The attacker creates a link to somebody else's setuid shell script and gives the link a name that starts with a dash '-'.

Step 2. In this step the attacker executes the link thereby receiving an interactive shell with an effective user id, set to the file owner's user id.

In the command line this can be done in two ways:

```
% ln target -x
% -x
```

or

```
% ln -s target -x
% -x
```

The first one of these creates a hardlink, whereas the second one creates a symbolic link. The results of both of the attacks are the same. If target is a setuid shell script with the #!/bin/sh mechanism the attacker will obtain a shell running with the target owner's privileges. If target is owned by root, the attacker will receive root privileges. The details of this scenario were given in Chapter 2. Here we finalize the state description diagram in the way that is used by USTAT. We first refer to the case where the attacker creates the hardlink and we describe the other case later.

The diagram consists of two steps, which correspond to the two actions of the attacker. When we fill the signature actions we have the diagram of Figure 4.8.

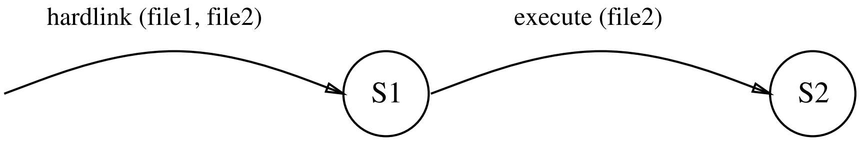


Figure 4.8 Partial State Transition Diagram

Next, we can identify the final compromised state, which is achieved after executing the second statement. In this state the user gains some previously unheld ability. His/her effective user id changes. Therefore we need a state assertion that compares the user's effective user id with the user's real user id. Fortunately these values are recorded in the audit records. So, the state assertion we use for this case is:

```
not euid = USER
```

euid acts like a function call, which returns the effective user id of the user whose identity is recorded in the last audit record. This state assertion implies that the value returned by euid has to be compared with the user id (=real user id) and then the result of this comparison has to be negated to obtain the final truth value for this state assertion.

One might ask whether it is enough to use this state assertion as the final state of our diagram. There might be many cases where a user executes a program that changes his/her effective user id to another id. Many system programs do that, such as passwd(1), or at(1). These are setuid to root programs, but they don't provide an interactive shell to the invoker. The problem is how to distinguish the execution of these programs from the case of the above attack. This is one of the major reasons why we use state transition diagrams, and the idea behind this also constitutes the importance of the state transition analysis tool. It is the state transition from the previous state to this final state that identifies it as a compromised state. Without identifying the previous state it would raise too many false alarms for the SSO.

Now, we should examine the result of the first signature action and identify the state assertions of the next state that is essential for the successful completion of both the scenario and this state transition diagram. After the creation of the hardlink, we expect the following state assertions to hold. If one of them fails to hold, then the state transition will not occur.

1. The name of the file that is just created should start with a 'dash'. The rest of the characters are irrelevant. (See "comments" in Chapter 5, Testing Scenario 1). The following state assertion accomplishes this.

   ```
   name (file2) = "-*"
   ```

   This function returns true, if the name of file2 starts with a dash. It makes the comparison against the filename instead of the full pathname.

2. The second requirement is that the owner of these files (since they are hardlinks, ownership should be the same) differ from the attacker. If the owner is the same as the attacker, then the attacker is not gaining any previously unheld ability. This is stated as follows.

   ```
   not owner (file1) = USER
   ```

   owner function in this state assertion returns true if the owner of file1 is the same as USER (subject of the audit record being processed). The not keyword at the beginning of this assertion negates the result.

3. Another requirement is that the file should be setuid enabled. Again, without this bit enabled, the penetration cannot occur.

   ```
   permitted (SUID, file1)
   ```

This state assertion returns true if file1 has the setuid bit set.

4. The file should be a shell script starting with the #!/bin/sh mechanism. Otherwise the attacker cannot receive an interactive shell after the second step.

```
shell_script (file1)
```

This returns true if file1 is a shell script with the #!/bin/sh mechanism. This is the only state assertion that cannot be validated by using the audit record. Hence a real-time check is necessary.

5. Finally, the file should give execute permission to users other than the owner, as in the following.

```
permitted (XGRP, file1) or
permitted (XOTH, file1)
```

With these state assertions we can complete our state transition diagram, which is given in Figure 4.9.

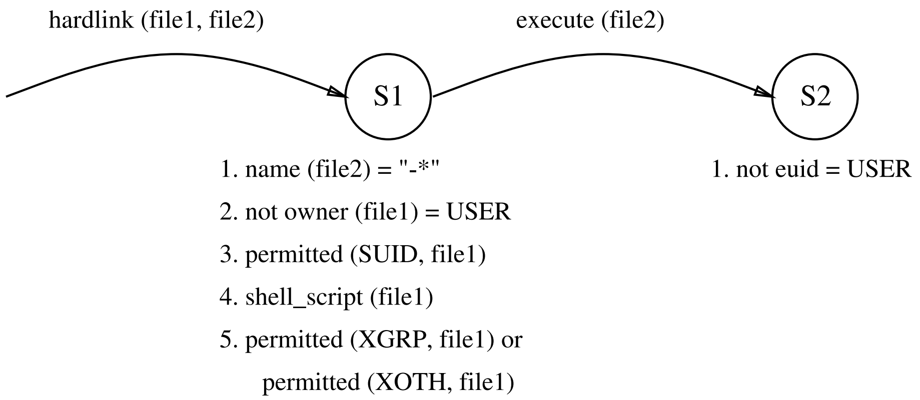


Figure 4.9 State Transition Diagram - 1

One major criterion is that the state assertions in one state should reference only one file, which is used in the last signature action (the last audit record). The audit records contain all the relevant information to these state assertions (except the shell_script state assertion). Each audit record contains information about only one file. Therefore, the state assertions can only reference the file in the last audit record. Although this factor might seem limiting it is necessary in order to have a consistent set of state assertions and a sound state transition diagram, e.g., a state assertion that looks like 'owner (file1) = USER' which comes after signature action 'create (file2)' doesn't make any sense since there is no action in 'create (file2)' that might affect the ownership of file1.

The variation of the above scenario can be done with symbolic links, the command sequence of which has been given before. However, with symbolic links we have a small problem. As mentioned in Section 4.1.2 of Chapter 4 the auditing mechanism records the accesses to symbolic links as accesses to the target. So, we have no way of distinguishing an access to a symbolic link from an access to its target. Therefore, the second signature action of the above scenario will not work for this situation. A more general case of the symlink attack is covered with the third state transition diagram, which is explained later in this section. Also note that, this scenario does not work if the shell involved is csh or tcsh.

Scenario 2

This scenario consists of one part of an attack scenario that existed in BSD 4.2 UNIX. The original scenario exploits a flaw in the mail program. The steps of the scenario as they existed in BSD 4.2 are as follows.

```
% cp /bin/sh /var/spool/mail/root
% chmod 4777 /var/spool/mail/root
% touch x
% mail root < x
% /usr/spool/mail/root
```

If these commands are executed successfully, the attacker will receive an interactive shell with root privileges.

On UNIX systems the mail directory, */var/spool/mail*, contains files under the names of the users on the system, e.g., root mail file is called */var/spool/mail/root*. The mail directory is writable by everyone, but the sticky bit is set, so that nobody can delete somebody else's mail file. There seems to be no problem with this configuration. However, whenever some user has no mail waiting, there is no file under his/her name in the mail directory.

As in the above scenario, if root has no mail waiting a malicious user can create a file with root's name.

Step 1. The attacker copies the shell under the name of root mail file. At this point the root mail file looks similar to the following.

```
-rw-------  1 (attacker) 147456 Jun 29 03:34 /var/spool/mail/root
```

Step 2. At this step the attacker sets the setuid bit of root mail file and makes it world readable/writable/executable

```
-rwsrwxrwx  1 (attacker) 147456 Jun 29 03:34 /var/spool/mail/root
```

Step 3. This step is not vital for the successful completion of the penetration. The attacker just needs a bogus file to send to root as a mail message.

Step 4. The mail program changes the ownership of root mail file to root, but fails to reset any other permission bits of the file.

```
-rwsrwxrwx  1 root       148613 Jun 29 03:34 /var/spool/mail/root
```

The result is a file owned by root, setuid enabled, and executable by everyone. In addition, it's a shell waiting to be executed. The bogus message at the end of the shell will be treated as the symbol table of the file and will not affect the proper execution of the shell.

Step 5. Result: The attacker has penetrated the system.

We are a bit fortunate with this scenario in SunOS 4.1.1. At step 4, the new mail program of SunOS 4.1.1 changes the ownership and also resets the setuid bit for the counterfeit mail file. However, the file remains world readable/writable/executable. That means, anybody can read or write this mail file. Therefore part of the above scenario works and it needs to be handled by USTAT. We handle this partial scenario this way: whenever somebody's mail file becomes readable/writable by group or everyone, it becomes vulnerable to read/write attacks. The execution of the file causes no risk on SunOS 4.1.1, since the setuid bit gets reset. Therefore, we don't need to check for the copying of the shell on the counterfeit mail file. The state transition diagram looks like the one in Figure 4.10.

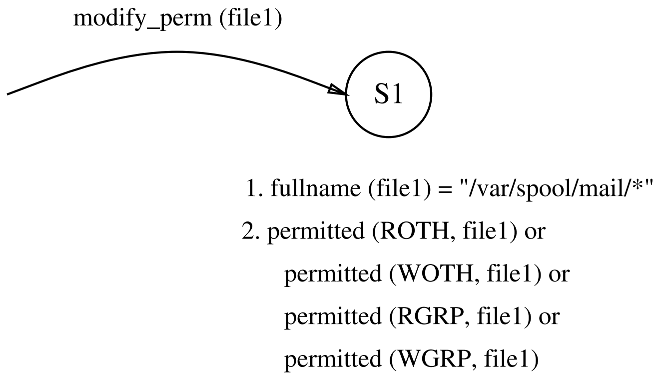


Figure 4.10 State Transition Diagram - 2

fullname is a different state assertion than *name*. name makes the comparison by just looking at the filename, whereas fullname considers the whole path of the file in the comparison.

Scenario 3

In this scenario the attacker exploits another flaw of setuid shell scripts. *file1* is a setuid shell script with the #!/bin/sh mechanism owned by any user except the attacker. The following is the contents of file1.

```
#!/bin/sh
...
file2
...
```

file2 is possibly a public executable program that exists in most users' paths. The attacker first creates another shell script called file2 in his/her current directory. The following is the one line content of this file.

```
exec /bin/sh
```

Next the attacker executes file1. If the attacker has "." as the first item in his/her path, the shell will search the current directory for the execution of file2. In this case, file2 is a shell script that executes another shell. Hence the attacker will receive an interactive shell running with file1's owner's privileges.

If file1 is owned by root, the attacker will obtain root privileges. The state transition diagram given in Figure 4.11 can be used to detect this penetration scenario.

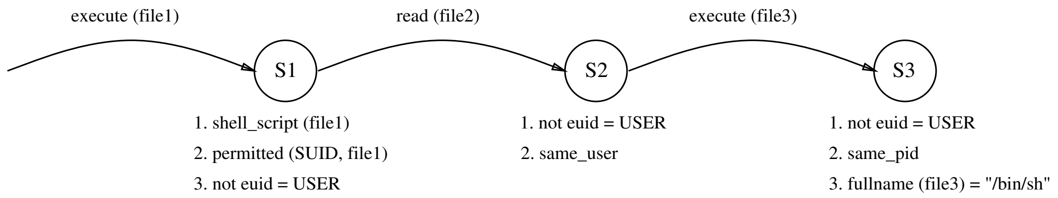


Figure 4.11 State Transition Diagram - 3

After the first signature action the attacker receives a shell that is not yet interactive. This shell executes the counterfeit public program in the second step, which in turn invokes another shell. The execution of the counterfeit public program is recorded as a read of file2 by the auditing system. Finally *file3* corresponds to the shell that the attacker is executing.

Scenario 4

This scenario illustrates a flaw in the *lpr* printing utility that allows an attacker to delete any file even if he/she does not have the authorization to do so. lpr is a "setuid to root" program, which has a "-r" option that allows the user to delete a file after the file is printed. However, the program fails to properly check the user's access rights against the target file. This way, the attacker can easily delete any file (that he/she has read access to, otherwise lpr cannot print it and hence will not be able to delete it) on the system. The attacker can also delete the password file to prevent anyone from logging in. This penetration scenario can be detected with the state transition diagram given in Figure 4.12

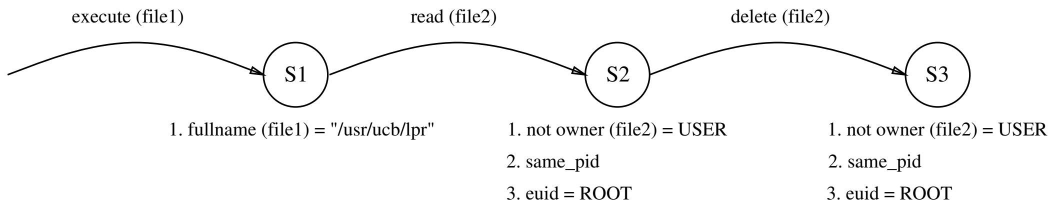


Figure 4.12 State Transition Diagram - 4

Scenario 5

Setuid shell scripts are known to be insecure because of a variety of flaws that can be exploited by using these shell scripts (See Appendix - Remarks about Unix Internals). In this scenario we want USTAT to raise an alarm whenever a user executes a setuid shell script owned by somebody else. Usually, this might be secure, but as in Scenario 1, if the execution is done through a symbolic link that starts with a dash '-' a compromise will occur. Since USTAT cannot detect executions through symbolic links (because of the way C2-BSM records the objects), the best thing USTAT can do at this point is to raise an alarm whenever a setuid shell script with the #!/bin/sh mechanism is executed.

When we look at the audit records that are generated upon execution of a shell script with the #!/bin/sh mechanism, it doesn't matter if the script is executed through a symbolic link or not, we see one key record that is being audited: the execution of the shell script. We can construct the state transition diagram with the following single step.

Sig_Act 1: `execute (file1)`

The state assertions after this step will be:

```
permitted (SUID, file1)
not owner (file1) = USER
shell_script (file1)
not euid = USER
```

describing the file's several features. The file should be setuid enabled, the owner should be different from the attacker, and the file should be a shell script with the #!/bin/sh mechanism. Also, at this point we expect that the attacker will have a different effective user id. Figure 4.13 gives the complete diagram.

The examples given in Section 4.1.2.4 of Chapter 4, (header "Examples of Audit Records") handle two different cases related to this scenario: one malicious, one legitimate case. The second example is a legitimate activity, whereas the fourth example is a malicious activity. The audit records generated upon these actions have no distinguishing feature that can be detected by USTAT, and therefore both of them will cause a warning to be generated.

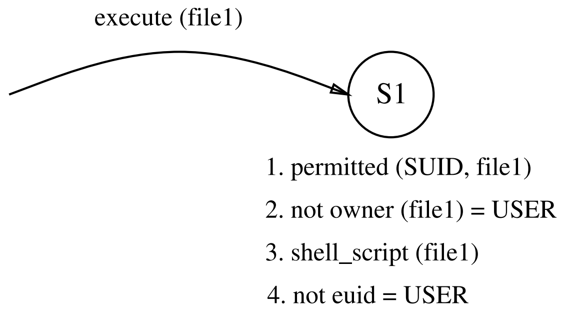


Figure 4.13 State Transition Diagram - 5

Scenario 6

In Section 4.2.1 of Chapter 4 we talked about the fact-base, which consists of filesets. One of these filesets, Fileset #1, was called restricted-read files. With this scenario we try to detect unauthorized reads of any file from this fileset. Fileset #3 consists of the files that are allowed to read some file in Fileset #1. Therefore, we raise an alarm whenever a file from Fileset #1 is read without executing a file from Fileset #3. The possible attack consists of the following two steps.

```
execute (file1)
read (file2)
```

The attacker executes a file in the first step, which allows him/her to have read access to the second file, which is one of the restricted-read files, Fileset #1. In these two steps, we require the first file to be a member of Fileset #3, or otherwise a compromise is detected. The first state consists of the following single state assertion, which follows the first signature action (execute file1).

```
not member (FS3, file1)
```

member returns true, if its second argument exists in the fileset passed in the first argument. Similarly, the second state should indicate that file2 is a member of the first fileset.

```
member (FS1, file2)
```

This state transition diagram states that, if a user executes a file that is not from Fileset #3 and reads another file that is from Fileset #1, then a compromise will be detected.

There is a slight detail missing. A user might complete this scenario and be detected as an attacker without doing any malicious activity. He/she might be reading file2 by some legitimate means, which happens to occur after execution of file1. In this case we get a false alarm. To avoid this type of false alarm, we need to find a relation between the two steps that should state that *"the reading of file2 is done during and via execution of file1."* For this purpose, we introduced a new state assertion, called *same_pid*, which returns true if the two signature actions that occurred prior to this state assertion have the same process id. With this addition the state transition diagram looks like the one in Figure 4.14.

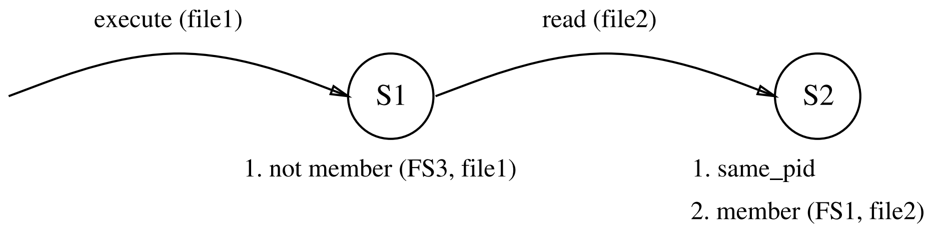


Figure 4.14 State Transition Diagram - 6

Scenario 7

Another fileset introduced in Section 4.2.1 of Chapter 4 was Fileset #2, which was called restricted-write files. Similar to the previous scenario, with this one we try to cover all unauthorized writes to Fileset #2. Fileset #4 consists of the files that are allowed to write on some file in Fileset #2. Therefore, whenever a file from Fileset #2 is written without executing a file from Fileset #4, an alarm will be generated. The possible attack consists of the following two steps.

```
execute (file1)
write (file2)
```

The attacker executes a file in the first step, which allows him/her to have write access to the second file, which is one of the restricted-write files. In these two steps, we require the first file to be a member of Fileset #4 or otherwise a compromise is detected. The first state consists of the following single state assertion that comes after the first signature action.

```
not member (FS4, file1)
```

member returns true if its second argument exists in the fileset passed as the first argument. Similarly, the second state should indicate that file2 is a member of the second fileset and as in the previous scenario that these two actions have the same process id's:

```
member (FS2, file2)
same_pid
```

This state transition diagram states that, if a user executes a file that is not from Fileset #4 and writes onto another file that is from Fileset #2 then a compromise will be detected. The state transition diagram looks like the one in Figure 4.15

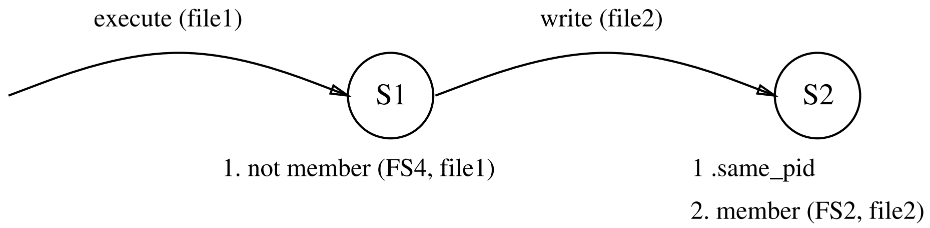


Figure 4.15 State Transition Diagram - 7

Scenario 8

Not all of the state transition diagrams necessarily indicate a compromised state. They can also be used to point out potential dangers on the system or to enforce some site security policy. For example, an alarm can be raised whenever a user writes on somebody else's file. This can be done simply by using the state transition diagram given in Figure 4.16.

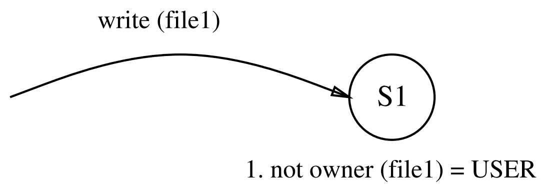


Figure 4.16 State Transition Diagram - 8

Scenario 9

Similar to the above scenario, the creation of setuid files can also be detected and a warning can be generated if that occurs. This can be done by the state transition diagram given in Figure 4.17.

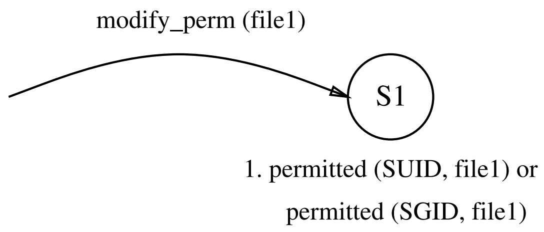


Figure 4.17 State Transition Diagram - 9

Scenario 10

Scenario 8 becomes more dangerous if the write action is done onto a file that is a member of Fileset #5. So, we add another state assertion to scenario 8 to obtain the diagram given in Figure 4.18.

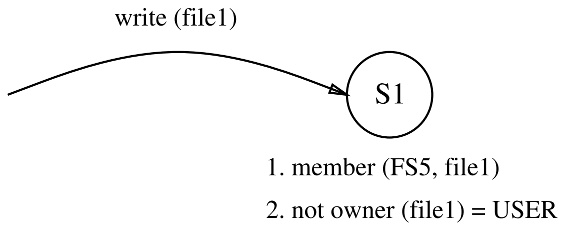


Figure 4.18 State Transition Diagram - 10

Scenario 11

There exists potential danger if somebody creates a file in a system directory. That can be a Trojan Horse implantation. We identified the directories that are vulnerable to these type of attacks as the *non-writable system directories* (NWSD). With the following state assertion USTAT can determine whether a file is located in a directory that is listed in NWSD.

```
located (RWDS, file1)
```

The state transition diagram in Figure 4.19 notifies the SSO, whenever a file is created in a non-writable system directory.

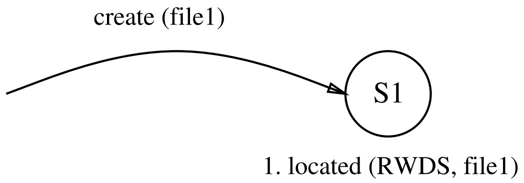


Figure 4.19 State Transition Diagram - 11

Scenario 12

A security policy may also state that: “no user shall delete other user’s files.” If that occurs, it can be detected by the state transition diagram given in Figure 4.20.

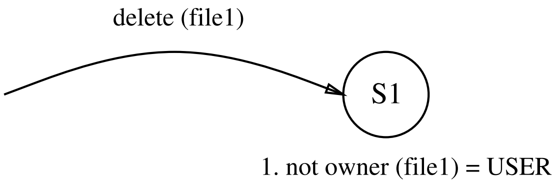


Figure 4.20 State Transition Diagram - 12

4.2.2.2 State Description Table

As we mentioned earlier in this chapter, state transition diagrams are stored in two text files, one of which is the State Description Table (SDT); the other is the Signature Action Table (SAT). These two tables form the rule-base.

The format of the state description table

The SDT is a text file, which consists of the state assertions of the states of individual state transition diagrams. In this text file, the lines starting with a pound sign '#' are comment lines, and they are not processed by the Rule-base Reader. For each state transition diagram there is a set of state descriptions. The specification of a set of state descriptions is the following.

```
SDn:
state-list.
state-list.
...
```

Each set of state descriptions starts with a header, *SDn*, with *n* identifying the state transition diagram number. The header is followed by one or more lines of state-lists. Each line is terminated with a period and an *EOLN*. Each state-list represents a set of state assertions that should be evaluated to true by the inference engine in order to anticipate the next signature action or to detect the final compromised state. States in a state-list are separated by using '&' for logical AND and '|' for logical OR. Each state-list should be written in Conjunctive Normal Form (CNF). No parentheses are necessary. A state description of the form (A | B) & (C | D | E) is written as the following.

```
A | B & C | D | E
```

where A, B, C, D, and E represent individual state assertions.

State assertions

Each state assertion consists of one function name and zero or more arguments. The functions are built into USTAT's rule-base. The arguments that are italicized should be replaced by either a constant value or by a built-in keyword. The following list describes how the arguments should be used.

- *file_var* always starts with identifier *file* and must be followed by an integer of range 1 to 5. That means, up to 5 different file variables may be involved in the state transition diagram. *file_var* in the signature action should match the *file_var* in the next state assertion.
- *file_set* refers to one of the pre-defined filesets. It can be any one of FS1, FS2, ..., FS5.
- *full_path* is a string enclosed in double quotes. The string should start with a slash '/' indicating a full path. It may have an '*' as a wildcard at the end.
- *file_name* is a string enclosed in double quotes. The string should not include any slashes. It may have an '*' as a wildcard at the end.
- *user_id* is either an integer corresponding to a user id or one of the following keywords: USER, ROOT. The keyword USER corresponds to the user who is responsible for the audit record that is being processed.
- *perm* is one of the keywords in Table 4.6.

Keyword	Meaning
RUSR	readable by owner
WUSR	writable by owner
XUSR	executable by owner
RGRP	readable by group
WGRP	writable by group
XGRP	executable by group
ROTH	readable by others
WOTH	writable by others
XOTH	executable by others
SUID	setuid enabled
SGID	setgid enabled

Table 4.6 Keywords for perm argument

Now, we look at each type of state assertion in turn.

1. **name** (*file_var*) = *file_name*
   This assertion evaluates true if the name of the file whose access is being recorded by the current audit record matches the filename given in the right-hand side.

2. **fullname** (*file_var*) = *full_path*
   This assertion evaluates true if the name of the file whose access is being recorded by the current audit record matches the path given in the right-hand side.

3. **owner** (*file_var*) = *user_id*
   Evaluates true if the owner user id of the *file_var* is equal to *user_id*.

4. **member** (*file_set, file_var*)
   Evaluates true if *file_var* belongs to the given fileset.

5. **euid** = *user_id*
   Evaluates true if the effective user id of the subject of the audit record being processed equals *user_id*.

6. **gid** = *group_id*
   Evaluates true if the group id of the subject of the audit record being processed equals *group_id*.

7. **permitted** (*perm, file_var*)
   Evaluates true if the permission bit given as *perm* is set in *file_var*'s permission bits.

8. **located** (RWDS, *file_var*)
   Evaluates true, if *file_var* is located in any of the directories listed in the file RWDS.

9. **same_user**
   Evaluates true if the subjects of the last two signature actions are the same.

10. **same_pid**
    Evaluates true if the process id's of the last two signature actions are the same.

11. **shell_script** (*file_var*)
    Evaluates true if the *file_var* is a *shell_script* with the #!/bin/sh mechanism.

It is also possible to negate the truth value of a state assertion. The keyword *not* added to the beginning of an assertion negates the return value, e.g.:

```
not owner (file1) = USER
```

returns true if the owner of file1 is different from USER.

The current SDT for USTAT

The following is the state description table that is currently being used by USTAT.

```
#           S T A T E   D E S C R I P T I O N   T A B L E
#
```

```
#                                    for
#
#                                 U S T A T
#
#-----------------------------------------------------------------------
# Unauthorized access to user privileges
# State Description-1
#
SD1:
name (file1) = "-*" & not owner (file1) = USER & permitted (SUID,file1)
     & shell_script (file1) & permitted (XGRP,file1)
     | permitted (XOTH,file1).
not euid = USER.
#
#-----------------------------------------------------------------------
# Unauthorized access (read/write) to user mail file
# State Description-2
#
SD2:
fullname (file1) = "/var/spool/mail/*" & permitted (ROTH,file1)
     | permitted (WOTH,file1) | permitted (RGRP,file1)
     | permitted (WGRP,file1).
#
#-----------------------------------------------------------------------
# Unauthorized access to user privileges
# State Description-3
#
SD3:
shell_script (file1) & permitted (SUID, file1) & not euid = USER.
not euid = USER & same_user.
not euid = USER & same_pid & fullname (file3) = "/bin/sh".
#
#-----------------------------------------------------------------------
# Unauthorized deletion of files
# State Description-4
#
SD4:
fullname (file1) = "/usr/ucb/lpr".
not owner (file2) = USER & same_pid & euid = ROOT.
not owner (file2) = USER & same_pid & euid = ROOT.
#
#-----------------------------------------------------------------------
# Unauthorized access to user privileges
# State Description-5
#
SD5:
permitted (SUID, file1) & not owner (file1) = USER &
     shell_script (file1) & not euid = USER.
```

```
#
#----------------------------------------------------------------------
# Restricted read violation
# State Description-6
#
SD6:
not member (FS3,file1).
same_pid & member (FS1,file2).
#
#----------------------------------------------------------------------
# Restricted write violation
# State Description-7
#
SD7:
not member (FS4,file1).
same_pid & member (FS2,file2).
#
#----------------------------------------------------------------------
# Potential trojan horse implantation or unauthorized modification of data
# State Description-8
#
SD8:
not owner (file1) = USER.
#
#----------------------------------------------------------------------
# Exposure to trojan horse attack
# State Description-9
#
SD9:
permitted (SUID,file1) | permitted (SGID,file1) &
     permitted (WOTH,file1) | permitted (WGRP,file1).
#
#----------------------------------------------------------------------
# Denial of service, possible TH implantation
# State Description-10
#
SD10:
member (FS5,file1) & not owner (file1) = USER.
#
#----------------------------------------------------------------------
# Denial of service, possible TH implantation
# State Description-11
#
SD11:
located (RWDS,file1).
#
#----------------------------------------------------------------------
# Unauthorized deletion of user files
```

```
# State Description-12
#
SD12:
not owner (file1) = USER.
#
#----------------------------------------------------------------------
# END OF STATE DESCRIPTION TABLE
```

4.2.2.3 Signature Action Table

In this section we discuss the second part of the rule-base for USTAT: the Signature Action Table (SAT).

The format of the signature action table

The SAT is a text file that consists of the signature actions of individual state transition diagrams. The format of this file is similar to the format of the SDT. Again, the lines that start with a pound '#' are comment lines, and they are not processed by the rule-base reader. For each state transition diagram there is a set of signature actions. The specification of a set of signature actions is the following.

```
SIGn:
action.
action.
...
```

Each set of signature actions starts with a header, *SIGn*, with *n* identifying the state transition diagram number. The header is followed by one or more lines of actions. Each line is terminated with a period and an *EOLN*. An action can be one of the ten action names that exist in the ACTION field of the audit record format of USTAT. The actions used in the SAT are as follows.

1. **read** (*file_var*)
2. **write** (*file_var*)
3. **create** (*file_var*)

4. **execute** (*file_var*)
5. **exit** (*file_var*)
6. **delete** (*file_var*)
7. **modify_owner** (*file_var*)
8. **modify_perm** (*file_var*)
9. **hardlink** (*file_var, file_var*)
10. **rename** (*file_var, file_var*)

The use of *file_var* is the same as the *file_var* in the SDT. The order of the file variables in hardlink and rename is important. The first variable is the "from" variable, which already existed prior to the call. The second one is the "to" variable. In other words, the first one is the target, the second one is the (new) object of that action.

Relation between the SDT and the SAT

For each set of state descriptions in the SDT there is a corresponding set of signature actions in the SAT. If there are n state descriptions in a particular state transition diagram, then there are n signature actions. The *i*.th signature action in a set of signature actions represents the signature action that is anticipated before the *i*.th state description. Figure 4.21 illustrates the correspondence between these two tables.

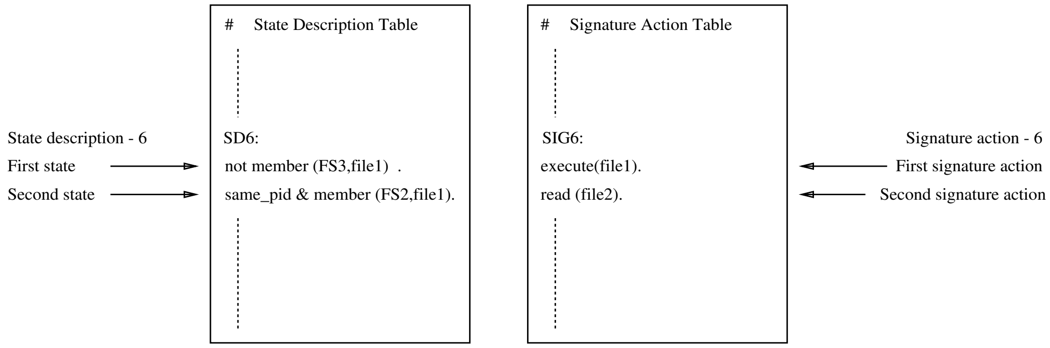


Figure 4.21 The SDT and the SAT

The current SAT for USTAT

The following is the signature action table that is currently being used by USTAT.

```
#           S I G N A T U R E   A C T I O N S   T A B L E
#
#                               for
#
#                            U S T A T
#
#----------------------------------------------------------------------
# Unauthorized access to user privileges
# Signature Actions-1
#
SIG1:
hardlink (file1, file2).
execute (file1).
#
#----------------------------------------------------------------------
# Unauthorized access to user privileges
# Signature Actions-2
#
SIG2:
modify_perm (file1).
#
#----------------------------------------------------------------------
# Unauthorized access to user privileges
# Signature Actions-3
#
SIG3:
execute (file1).
read (file2).
execute (file3).
#
#----------------------------------------------------------------------
# Unauthorized deletion of files
# Signature Actions-4
#
SIG4:
execute (file1).
read (file2).
delete (file2).
#
#----------------------------------------------------------------------
```

```
# Unauthorized access to user privileges
# Signature Actions-5
#
SIG5:
execute (file1).
#
#-----------------------------------------------------------------------
# Restricted read violation
# Signature Actions-6
#
SIG6:
execute (file1).
read (file2).
#
#-----------------------------------------------------------------------
# Restricted write violation
# Signature Actions-7
#
SIG7:
execute (file1).
write (file2).
#
#-----------------------------------------------------------------------
# Potential trojan horse implantation or unauthorized modification of data
# Signature Actions-8
#
SIG8:
write (file1).
#
#-----------------------------------------------------------------------
# Exposure to trojan horse attack
# Signature Actions-9
#
SIG9:
modify_perm (file1).
#
#-----------------------------------------------------------------------
# Denial of service, possible TH implantation
# Signature Actions-10
#
SIG10:
write (file1).
#
#-----------------------------------------------------------------------
# Denial of service, possible TH implantation
# Signature Actions-10
#
SIG11:
```

```
create (file1).
#
#-----------------------------------------------------------------------
# Unauthorized deletion of user files
# Signature Actions-12
#
SIG12:
delete (file1).
#
#-----------------------------------------------------------------------
#  END OF SIGNATURE ACTIONS TABLE
```

4.2.2.4 The Rule-base Reader and Data Structures

The rule-base reader is a program module that is responsible for reading the two tables, the SDT and the SAT into memory, checking their syntax, and verifying their consistency. If there is any error in these two tables, further operation of USTAT will not be possible. The SDT is read into a memory structure as shown in Figure 4.22

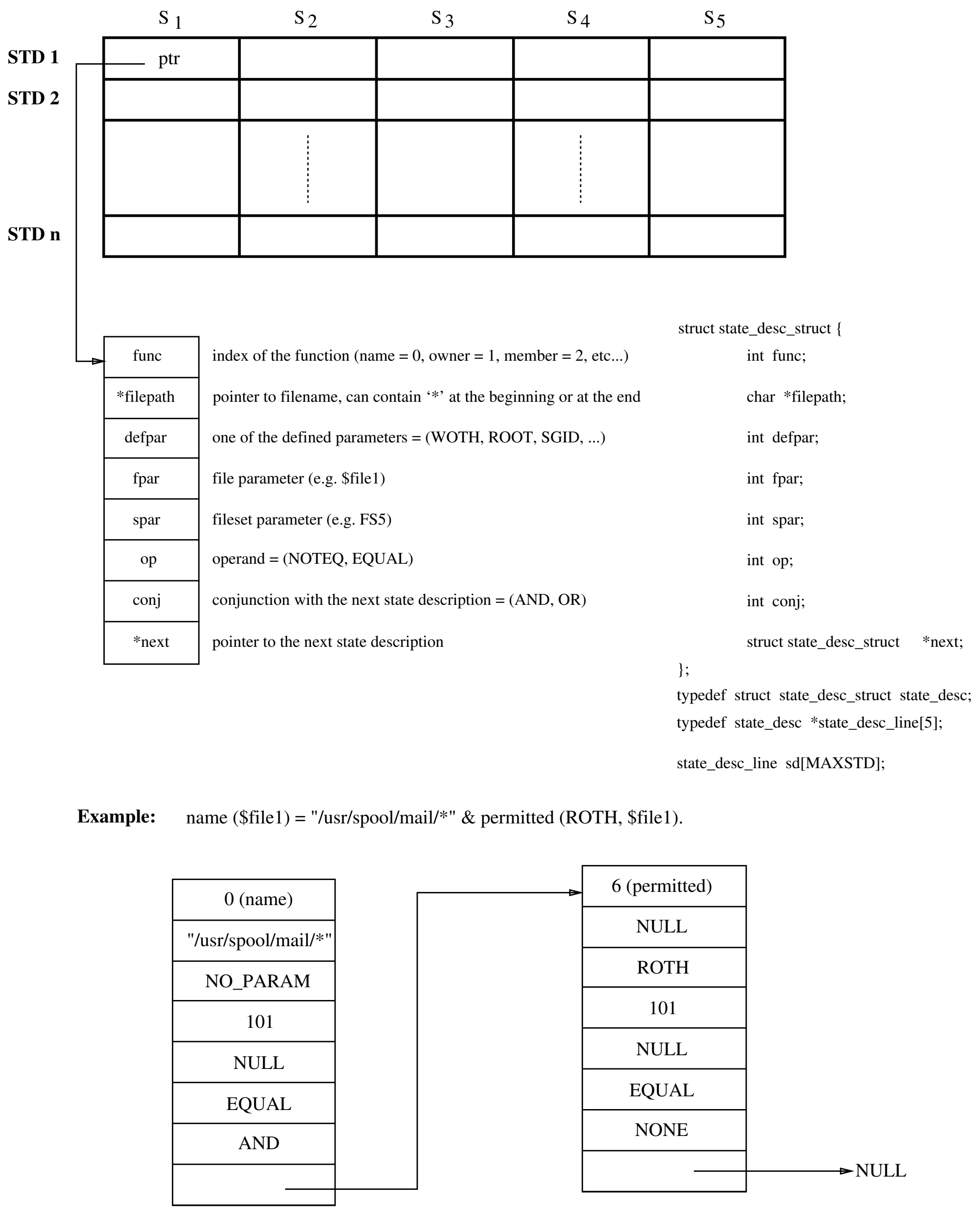
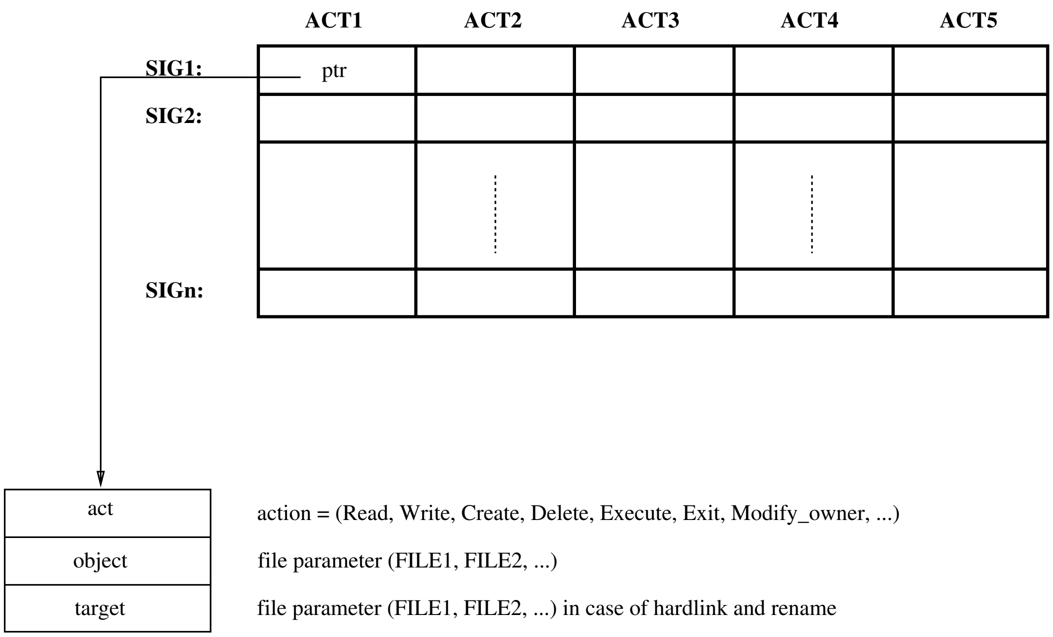


Figure 4.22 State Description Table Data Structures

Similarly, the SAT is read into a memory structure as shown in Figure 4.23. Both memory areas are used by the inference engine for the rest of the program operation.




```
struct  action_desc_struct {
            int  act;
            int  object;
            int  target;
};
typedef  action_desc_struct  action_desc;
typedef  action_desc  *action_desc_line[MAXLINES];
action_desc_line  ad[MAXSTD];
```

Example:

SIG1:
hardlink (file1, file2).
read (file1).
execute (file3).

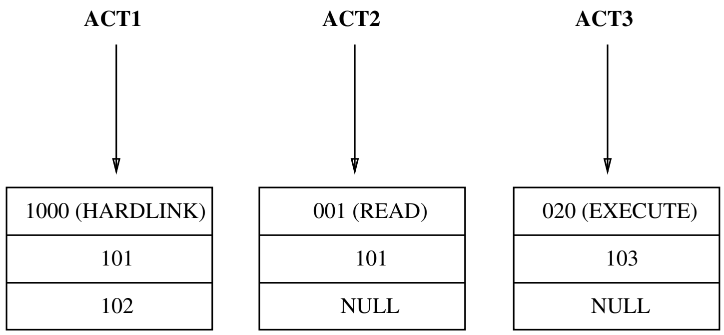


Figure 4.23 Signature Action Table Data Structures

4.3 The Inference Engine

4.3.1 Data Structures

In this section we give the data structures that are used by the inference engine. We also give the motivation for why we need these data structures. We use the term scenario to refer to a particular state transition diagram and the term instantiation (of a scenario) to refer to different instances of the same scenario.

4.3.1.1 Inference Engine Table

The inference engine table consists of the snapshots of penetration scenario instances (instantiations) that are not yet completed on the target system. Here, the notion of "table" comes from the way that the inference engine operates, rather than from the data structure itself. The imaginary table of the inference engine is presented in Table 4.7.

	S_1	S_2	S_3	S_4	S_5
1					
2					
3					
4					
. . .					
n					

Table 4.7 Inference Engine Table

Each row represents one possible instance of a penetration scenario that has been activated on the system. Each column depicts how far a compromise is from being achieved. Whenever the inference engine detects an audit record that matches the next action and satisfies the next state of a state transition diagram, it duplicates the row and "marks" the corresponding cell on the duplicated row. The reason for duplicating the row is that the original row can still represent part of another instantiation.

To illustrate the manipulation of the table we give an example that uses a hypothetical state transition diagram, called STD_h. This diagram consists of three signature actions. Suppose that the first signature action is creating a file that possesses certain characteristics, which are given in the first state of the diagram (State Assertions-1). Also, suppose that the second signature action is the execution of the file created in the first step and is followed by the third signature action, which indicates reading another file on the system. The state that follows the last signature action depicts the final compromised state. This is illustrated in Figure 4.24. From this diagram one can observe that there may be many executions of the same file by possibly different users once the file has been created.

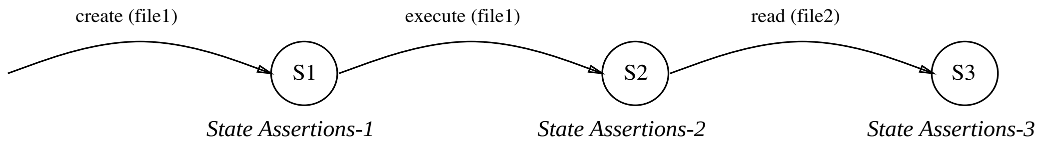


Figure 4.24 A Hypothetical State Transition Diagram

The initial table for the inference engine looks like Table 4.8. n is the number of different state transition diagrams entered into the state description table (SDT). There is one row for each scenario, meaning the initial signature action of each diagram is being anticipated by the inference engine. The hypothetical state transition diagram (STD_h) corresponds to row h.

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n					

Table 4.8 Initial Inference Engine Table

Action 1: User A creates file1 and this satisfies the state assertions in S_1.

Result: Row h is duplicated and S_1 is marked on the new row (See Table 4.9).

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n+1	X				

Table 4.9 Inference Engine Table after Action 1

At this point there are two different signature actions (related to STD_h) that are being anticipated by the inference engine. These are the following.

1. The creation of another file.
2. The execution of file1.

These two signature actions correspond to the two rows (h) and ($n+1$) of the above table, respectively.

Action 2: User B executes file1 and this satisfies S_2 of STD_h.

Result: Row $n+1$ is duplicated and S_2 is marked on the new row. But, this result should not affect row $n+1$ since the same file can be executed by another user. Row $n+1$ is deleted only when file1 is deleted or when the state assertions in S_1 of STD_h no longer hold for row $n+1$. As a result of the second action the table will look like Table 4.10.

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n					
n+1	X				
n+2	X	X			

Table 4.10 Inference Engine Table after Action 2

Action 3: User B reads file2 and this satisfies S_3 of STD_h.

Result: Since S_3 is the final state of STD_h a compromise has been achieved. No new row is added to the table and row $n+2$ remains in the table. So, as a result the table will remain as Table 4.10.

The details of how the inference engine works on the table is given in detail later in this chapter. As we mentioned earlier, the term "table" and the tables above are used only to illustrate the approach of the inference engine. The data structure of the table is in fact different from the above pictures, but serves the same purpose. The following gives more information about the table and its fields.

```
ie_table:
        std_no                  1 byte
        *filelist[5]            5 char pointers
        last                    1 byte
        *act[5]                 5 audit record pointers
        *next                   pointer to next row

*TBL    pointer to ie_table
```

Each record of the type *ie_table* represents one row of the table. TBL is the variable that we use throughout the program. TBL points to the first row in the table and the rows are maintained in a linked list structure. We also use the term TBL in this chapter to refer to the inference engine table.

std_no is used to record the state transition diagram that a particular instantiation refers to.

filelist [5] consists of the name of the files that are involved in this instantiation. The same names also exist in *act[5]*, but this field makes it easy to keep track of the filenames as they are numbered in the state transition diagrams.

last denotes the last state that has been satisfied on that particular row. When this number reaches the number of the last state of the particular state transition diagram, a compromise is detected.

act [5] corresponds to the audit records that are involved in this instantiation. This field might be necessary for the SSO to further analyze the history of the compromise.

next points to the next row in the table.

Unfortunately, this structure is not enough for the efficient operation of the inference engine. To maintain the inference engine table and to speed up the lookup process on this table the inference engine makes use of two more data structures: *Filename reverse pointers* and *expected signature action reverse pointers*, which we call FREV and ESAREV, respectively.

4.3.1.2 Filename Reverse Pointers

Whenever a file is involved in a penetration scenario its name is recorded in the filelist field of the inference engine table. For each such filename, a linked list of pointers is kept. Each pointer of this list points to the row of the table in which the filename exists. We need a linked list of pointers since the same file might be involved in more than one instantiation or in more than one scenario. The data structure is shown in Figure 4.25.

So, for each filename there is a linked list of reverse pointers, which point to the rows of the inference engine table. For faster lookup, the list of files

(FREV) is sorted by filename. This list is necessary to keep the inference engine table up-to-date. For instance, whenever a file in FREV is deleted, the rows that involve this filename can be found easily by following the pointers and these rows can be deleted from the table, as there is no reason to keep them. There are other instances where FREV is necessary. We discuss these in detail in Section 4.3.2 of Chapter 4.

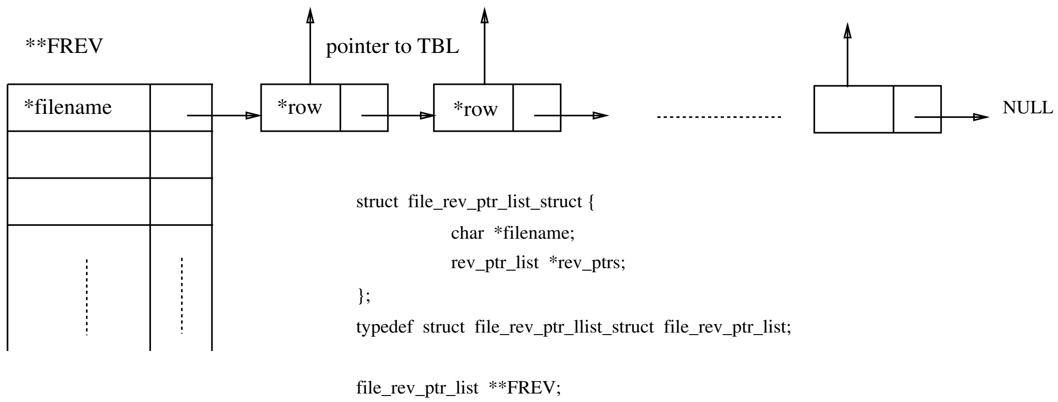


Figure 4.25 Data Structure for FREV

4.3.1.3 Expected Signature Action Reverse Pointers

An expected signature action refers to an action on a state transition diagram, the previous state of which had been satisfied and is still satisfied. For example, for the state transition diagram shown in Figure 4.26, initially Sig_Act-1 is an expected signature action. When this action occurs and the inference engine decides that the first state holds, then the next signature action becomes an expected signature action.

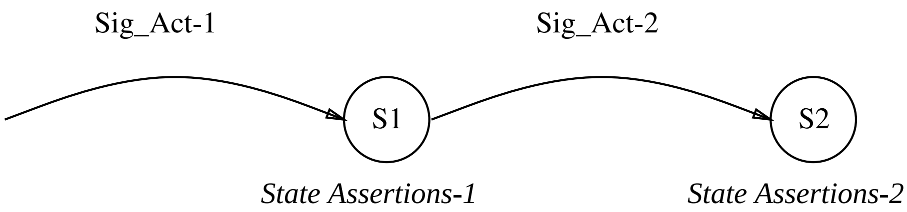


Figure 4.26 Sample State Transition Diagram

For each instantiation, and therefore for each row of the inference engine table USTAT has one expected signature action, such as CREATE, WRITE, EXECUTE, etc. Whenever an audit record is passed by the audit record preprocessor, the inference engine needs to check whether the action in the last audit record is being expected by any of the rows in the inference engine table. Rather than going through all of the rows in the inference engine table, a table of expected signature actions is employed, each entry of which consists of a list of pointers to the rows. Since there are 10 actions in the audit records, there are 10 rows on this table. In each row there is a list of pointers that point to the rows that expect this action for the next state transition. This table is called ESAREV.

The data structure is similar to the structure of FREV, except that the length of the ESAREV table is constant (10), whereas the length of the FREV table is dynamic. Figure 4.27 depicts the data structure for ESAREV. The processing of the expected signature actions is explained further in the next section.

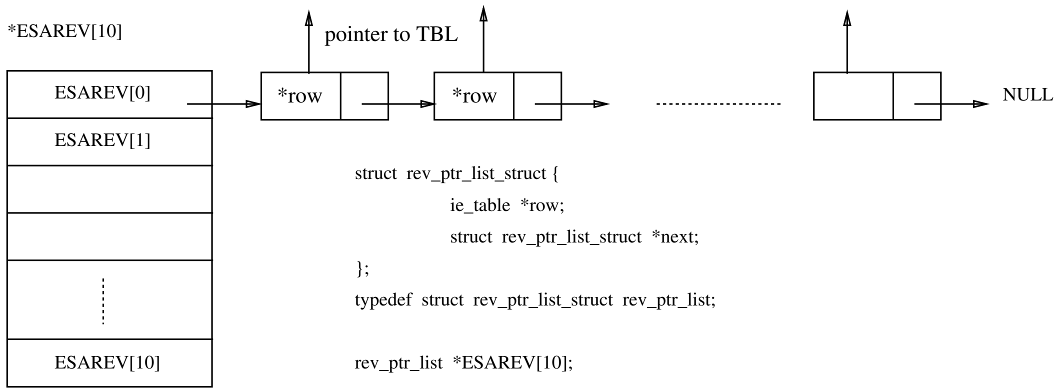


Figure 4.27 Data Structure for ESAREV

4.3.2 Program Operation

Below is a very short algorithm for the operation of the inference engine. We discuss each step of this in detail in the following subsections.

```
1  For each audit record passed by the preprocessor do begin
2     Call the fact-base updater
3     Process expected signature actions
4     If action in audit record is RENAME then
5        Rename files in FREV and TBL
6     Process file reverse pointers
7  Endfor
```

When there is no new audit record passed by the preprocessor, the inference engine stays idle until one is available. The operation of the fact-base updater (line 2) has been discussed in Section 4.2.1 of Chapter 4. In the remainder of this section we discuss the remainder of the algorithm.

4.3.2.1 Processing Expected Signature Actions

In this section we discuss how ESAREV is maintained and how it is used to add new rows to the inference engine table and to detect compromises. Remember there were 10 different actions in ESAREV, each of which points to a list of reverse pointers as shown in Figure 4.27. When an audit record is passed to the inference engine to process ESAREV, the first thing the inference engine has to do is to match the action in the audit record to the action in ESAREV. Once the expected signature action is located, the list of reverse pointers is used to check whether any new states in the rows that are pointed to are satisfied as a result of the audit record. The algorithm for this process is given below.

```
Locate the action of the audit record in ESAREV
For each row pointer in the linked list do begin
  If the next state of the row is satisfied then begin
    Duplicate the row into a new row
    Update the new row by inserting the new audit record
    Call the decision engine with the new row
    as an argument
    If the new row achieved a compromised state then begin
      Save the row to a file for future reference
      Free the memory allocated by the new row
```

```
10       Endif
11       Else begin
12         Add the new row to the inference engine table
13         Add the next expected signature action to ESAREV
14         Add the files involved in the new row to FREV
15       Endelse
16     Endif
16   Endfor
```

The process that decides whether a state is satisfied (step 3), is discussed later in Section 4.3.2.4 of Chapter 4. The details of step 6 are discussed in Section 4.4 of Chapter 4, "The Decision Engine."

When a compromised state is achieved by the row (step 7), it is not added to the inference engine table since no further processing needs to be done for that row. In step 11, the achieved state is not a final compromised state. Therefore, the new row is added to several data structures. First, it is added to the inference engine table. To be able to access this row later, the address of this row is added to the list of pointers in ESAREV that correspond to the next signature action of the row. Furthermore, for each filename that appears in the filelist of the new row, the address of the new row is added to the corresponding list of addresses in FREV. If the file does not exist in FREV a new entry for the file is created and the new row's address becomes the only pointer in the list. This adding process is illustrated in Figure 4.28.

4.3.2.2 Processing File Reverse Pointers

When an audit record is passed by the preprocessor, the record may indicate an action that is not among the expected signature actions. That means the list of the rows in ESAREV for that particular action might be empty. However, the action can still affect the state assertions that have been satisfied before the audit record that is being processed. As a very simple example, if the file in one of the rows of the inference engine table gets deleted, there is no way to finish the scenario. The following algorithm is used by the inference engine to process the file reverse pointers to keep the inference engine table up-to-date.

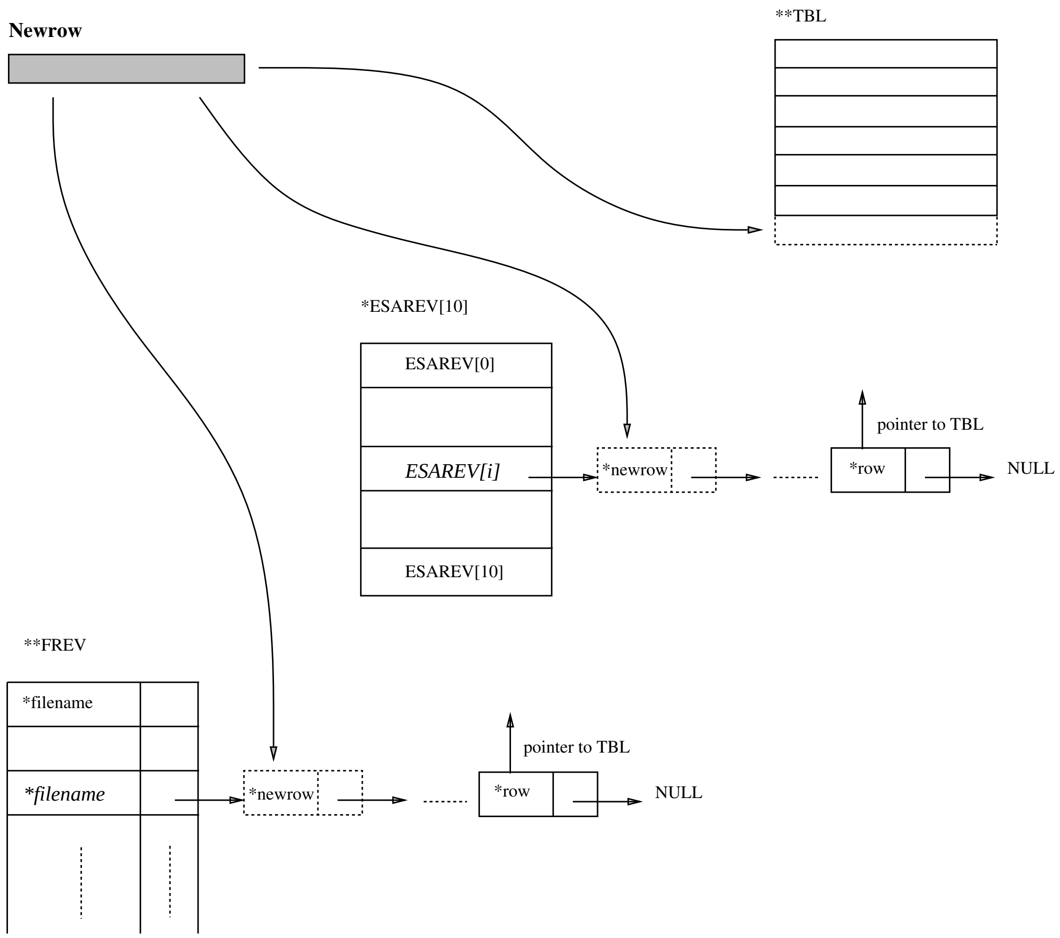


Figure 4.28 Insertion of a new row to the various data structures

```
1   If audit_record.action is one of (MODIFY_PERM or
    MODIFY_OWNER or DELETE or RENAME or EXIT) then begin
2     Binary search FREV for audit_record.object
3     If found then begin
4       If audit_record.action = RENAME then
5         Rename filename in the inference engine table
6       For each row corresponding to audit_record.object
        in FREV do begin
```

```
 7         If (object in the last action of the row) =
           audit_record.object then begin
 8           If (audit_record.action = DELETE) or
                (last state is no longer satisfied) or
                (audit record indicates an EXIT for the
                 row's last EXECUTE action) then begin
 9              Delete the row's address from ESAREV
10              Delete the row from the inference engine table
11              Delete the row's address from FREV
12           Endif
13         Endif
14       Endfor
15     Endif
16   Endif
```

In step 1, the inference engine checks to see whether the audit record indicates an action that might affect the inference engine table. Remember that this process is done after the expected signature actions are processed. No Read, Write, Create, Execute, or Hardlink action can further affect the inference engine table. Therefore, FREV is processed only if the action differs from these.

In step 4, the inference engine checks to see if the audit record indicates a Rename action, in which case all the filenames in the inference engine table that match the object name in the audit record are renamed. This process is explained in the next section.

In step 8, the inference engine tries to resatisfy the last state of the rows that have the object of audit_record in their last action. The row and all its traces are discarded from ESAREV and FREV if any of the following three cases holds.

1. If audit_record.action is a Delete, meaning the object is deleted.
2. If the last state of the row can no longer be satisfied (due to a change in file attributes, etc.)
3. If the last action in the row was an Execute that is Exit'ed by the last audit record.

The third case is necessary since Execute'd programs cannot be kept in the inference engine table after they terminate; otherwise, the inference engine table would grow indefinitely. Therefore, as soon as a program is Exit'ed, the row that has an Execute as the last action is deleted.

This also implies that scenarios that have *EXECUTE* $file_x$ as an action also include an internal state assertion in the next state by default: $file_x$ *has not yet terminated*.

4.3.2.3 Renaming Files

In USTAT the objects (files) in the target system are identified by their filenames. In all the data structures discussed in previous sections, files are recorded by using their filenames. However, it is very likely that files can be renamed and penetration scenarios can occur by using one filename during the first step, then changing the name before the second step and performing the second step by using the new name. The following example illustrates a variation from the original scenario by renaming the file. The original scenario as it appears in the state transition diagram is the following.

Step	Action	Physical File	Filename
1	Create	File1	foo
2	Execute	File1	foo

The variation of the above scenario by renaming the file could not be detected by the inference engine under normal circumstances, since USTAT follows the filename rather than the physical file. The variation looks like the following.

Step	Action	Physical File	Filename
1	Create	File1	foo
2	Rename	File1	foo bar
3	Execute	File1	bar

For this reason, the inference engine anticipates the Rename actions passed by the preprocessor and whenever a Rename appears, it attempts to change the filenames in the inference engine table and filename reverse pointers. The Rename action in an audit record has two arguments, one *target* and one *object*. The object refers to the previous name of the file and target is the new name. The algorithm for the renaming process of the inference engine is given below. Using this algorithm the inference engine easily detects the variations of penetration scenarios that are accomplished by renaming objects.

```
If audit_record.action = RENAME then begin
  Binary search FREV for audit_record.object
  If found then begin
    For each row pointer in the list of pointers do
      Rename target to audit_record.target in row.filelist
    Rename audit_record.object to audit_record.target in FREV
  Endif
Endif
```

There is another data structure where the files need to be renamed: the hardlink information of the system. This renaming is done by the fact-base updater as discussed in Section 4.2.1 of Chapter 4.

4.3.2.4 Satisfying the States of a State Transition Diagram

So far, we know how the inference engine table, TBL, the expected signature action reverse pointers, ESAREV, and the filename reverse pointers, FREV, are used by the inference engine. Using all these data structures, the most important function of the inference engine is to evaluate the state assertions. Step 3 of the algorithm given in Section 4.3.2.1 of Chapter 4 and step 8 of the algorithm given in Section 4.3.2.2 of Chapter 4 handle the process of satisfying the state assertions.

To check the truth value of a state of a state transition diagram two inputs are needed: the list of state assertions for that state and the last audit record that is passed by the preprocessor. Each state assertion in that state needs to be evaluated. The state assertions are combined with AND, and OR connectives, which need to be considered. The audit record is necessary since it reflects the last state change in the target system.

During the process of satisfying the state assertions of a state transition diagram, a file variable in the diagram, such as file1 or file2, may be either *bound* or *unbound*. A file variable in a signature action is bound, if the same variable has been used in a previous signature action of the same state transition diagram. A file variable in a signature action is unbound (or free), if it is being used for the first time in that signature action. Examples for both follow.

File1 in the transition diagram in Figure 4.29 is unbound prior to the first signature action, but it becomes bound after the first state is satisfied, prior to the second signature action. In the second signature action, the name of the file being executed should match the name of the file in the first signature action.

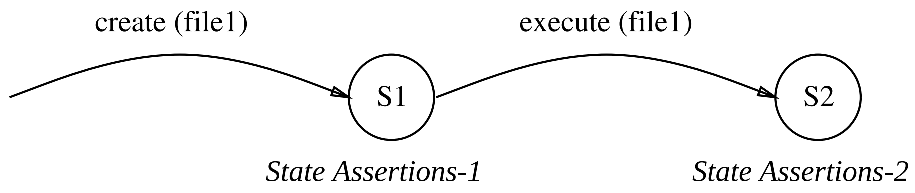


Figure 4.29 Bound and Unbound File Variables

In the diagram in Figure 4.30, file2 is an unbound variable prior to the second signature action since its name doesn't have to match the name of file1.

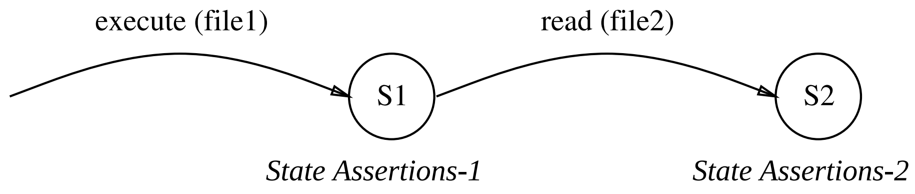


Figure 4.30 Unbound File Variable

The algorithm for satisfying the next state of a row in TBL is the following.

```
Inputs: row, audit_record

 1  Determine the next state to be satisfied for the row
 2  If (the filename in the state is unbound) or
    (it is bound and matches audit_record.object) then begin
 3     Holds = TRUE
 4     While (Holds) and (there are more state assertions
       to be satisfied) do begin
 5        If next state assertion is satisfied then
 6           Holds = True
 7        Else
 8           Holds = False
 9        If (not Holds) and
             (the next assertion's connective is an OR) then
10           Holds = True
11        Else if (Holds) and (the next assertion's
          connective is an OR) then
12           Skip the assertions until the connective is an AND
13     Endwhile
14  Endif
```

Step 5 of the above algorithm checks the truth value of a single state assertion. Possible functions for state assertions and how these assertions are evaluated were given in Section 4.2.2 of Chapter 4, "State Assertions."

4.3.2.5 Using the Hardlink Information

The necessity for hardlink information was given in Section 4.2.1 of Chapter 4. Hardlinks in the target filesystem constitute a major part of the fact-base for USTAT and they play a crucial role for the proper operation of the inference engine. Hardlinks can be used by an attacker to perform variations on a penetration scenario. These variations are similar to the variations obtained by renaming the file in a scenario. The files that are hardlinked to each other correspond to the same physical file. Therefore an access to one of them can be considered as an access to any of the other(s). As an example, consider two files, *foo* and *bar*, which are hardlinked. That means, they have the same inode and device number, only their names as directory entries are different. Now consider the scenario illustrated in Figure 4.31 in which the files in the first signature action and the second are the same.

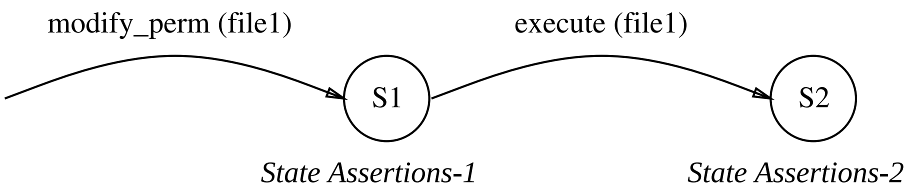


Figure 4.31 Variations with Hardlinks

By "same" we literally mean that they are the same physical files. Hence this scenario can be performed in the following way, by using two different filenames, but the same physical files.

1. Modify_Perm foo

2. Execute bar

For this reason, the inference engine uses the hardlink information along with a special purpose name matching routine. The result of name matching of foo and bar will return true, if foo and bar are in the same group of hardlinked files in the hardlink information. For example, for the two steps of the above scenario, foo and bar belong to the same group of files in the hardlink information as shown in Figure 4.32.

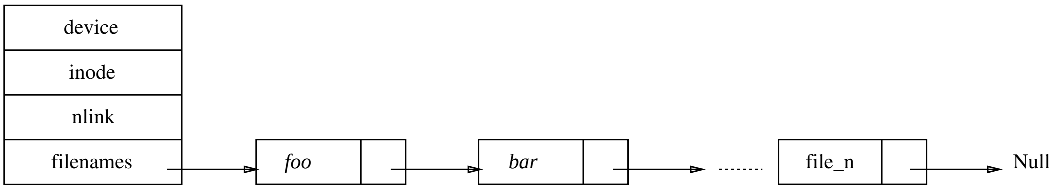


Figure 4.32 A sample hardlink entry in hardlink information structure

After the first signature action has occurred and after the first state has been satisfied, the inference engine anticipates an EXECUTE action. If the inference engine receives an audit record with an EXECUTE action, it first checks whether the filename in the audit record matches file1 in the scenario. At this point file1 is a bound file variable. If this first string comparison is not successful, it then tries to see, whether these files are in the same group of the hardlink information. With this approach, the inference engine covers all the variations to penetration scenarios that are accomplished using hardlinks.

4.3.3 Some Restrictions

There exist some problems that affect the proper operation of the inference engine. The following is a list of them along with the reasons.

1. The DELETE action can only be used as the last signature action in a state transition diagram. If it is used as an intermediate action and if it occurs and satisfies the following state, that row can never get deleted from the inference engine table. Hence the inference engine table can grow indefinitely.

2. A similar problem exists with the EXIT signature action. For the same reasons the table can grow indefinitely.

3. If a process is killed with the kill command or by using Ctrl-C, the termination of the process is not recorded by the auditing system. So, there is no way for the inference engine to decide whether a process is still active. Therefore, the table can grow indefinitely.

The first two problems may be eliminated in future releases of USTAT, the last one is an undesirable feature of C2-BSM.

4.4 The Decision Engine

4.4.1 Overview

The decision engine is responsible for informing the SSO about the results of the inference engine activities. This part of USTAT is flexible and can be implemented to print a very simple message on the screen or to actually interfere with the penetration scenario by taking the appropriate preemptive action. In this prototype implementation we focused more on the functionality of detecting compromises and on the accuracy of how well we could define scenarios using state description diagrams. Therefore, we spent little effort on the complexity of the decision engine. The output of the decision engine could be one or more of the following ranking from simplest to more complicated.

- Inform the SSO that a compromise has been achieved.
- Inform the SSO whenever a state of any instance of the scenarios has been satisfied.
- Suggest possible action to the SSO to preempt a state transition that can lead to a compromised state.
- Play an active role in preempting the attack.

The decision engine implemented for USTAT performs the first three of the above. The last one will be considered for future versions.

The inference engine notifies the decision engine whenever a new state of a penetration scenario is satisfied. The address of the row for which a new state is satisfied is all that the decision engine needs to draw some conclusion and notify the SSO. Each row in the inference engine table contains the necessary information to be able to look at the history of a penetration instance and decide which actions caused the state transition and who is or who are the responsible parties in this instantiation.

Each penetration scenario has a particular attack concept behind it. When we come up with state transition diagrams we know what kind of attack we are dealing with. We also know that each step of that scenario leads to a distinct state on the system that can be identified by the inference engine. By incorporating those ideas with the knowledge of the target operating system we can come up with specific messages for each step of each distinct scenario. This gives us the flexibility to provide the SSO with more rational messages and suggestions for preemptive actions rather than just saying "Warning: Somebody is attacking the system."

4.4.2 Data Structures

A simple data structure, which we call the decision table, is being used for the decision engine. This table is identical to the state description table except that the entries in it are the warning messages and suggestive actions instead of state assertions. For each state in a state transition diagram there is a (possibly empty) message in the decision table. The decision for some states is NULL, since some signature actions are likely to occur very frequently even under legitimate circumstances. In these cases we prefer not to have any message displayed.

The message strings in the decision table are kept in a two dimensional array of strings, and the size of the array is the same as the size of the state description table. The data structure for the decision table is shown in Figure 4.33. These strings are read into memory from a text file upon initialization of USTAT. The format of this text file is explained in the following section.

	S_1	S_2	S_3	S_4	S_5
DEC 1	message [1,1]				message[5,1]
DEC 2	message [1,2]				
DEC n	message [1,n]				message [5,n]

Figure 4.33 Decision Table Data Structure

4.4.3 The Decision Table for USTAT

The decision table is kept in a text file called *DT*. The format of this file is similar to the format of the state description table and the signature action table. The lines that start with a pound symbol '#' are comment lines, and they are not processed. For each state transition diagram there is a set of messages where the *i*.th message corresponds to the *i*.th state of the state transition diagram. The specification of a set of messages is as follows:

```
DECn:
message.
message.
...
```

DECn identifies the scenario. Each message line is terminated with a period and EOLN. An empty message is indicated by NULL, meaning that no warning needs to be generated upon achieving the corresponding state. The following are the contents of the current decision table DT for USTAT:

```
#                   D E C I S I O N    T A B L E
#
#-----------------------------------------------------------------------
# Unauthorized access to user privileges
# Decision-1
#
DEC1:
Suspicious link FILE1 created by user to setuid file FILE2.
If executed, FILE1 will invoke an interactive setuid shell with
another user's privileges.

Compromise detected: Unauthorized access to user privileges. Shell invoked
interactively via FILE1 with another user's privileges.
#
#-----------------------------------------------------------------------
# Unauthorized access (read/write) to user mail file
# Decision-2
#
DEC2:
Compromise detected: Unauthorized access (read/write) to user mail file.
#
#-----------------------------------------------------------------------
# Unauthorized access to user privileges
# Decision-3
#
DEC3:
Warning: User executing somebody else's SETUID shell script (FILE1).
NULL.
```

```
Compromise detected: Unauthorized access to user privileges. Shell invoked
interactively via FILE1 with another user's privileges.
#
#-----------------------------------------------------------------------
# Unauthorized deletion of file
# Decision-4
#
DEC4:
NULL.
Warning: User executed lpr to print FILE2 owned by somebody else.
If lpr invoked with -r FILE2 will be removed illegally.

Compromise detected: FILE2 got deleted illegally using lpr -r.
#
#-----------------------------------------------------------------------
# Possible unauthorized access to user privileges
# Decision-5
#
DEC5:
Warning: User executing somebody else's SETUID shell script.
#
#-----------------------------------------------------------------------
# Restricted read violation
# Decision-6
#
DEC6:
NULL.
Compromise detected: Restricted read violation. FILE2 referenced
illegally via FILE1.
#
#-----------------------------------------------------------------------
# Restricted write violation
# Decision-7
#
DEC7:
NULL.
Compromise detected: Restricted write violation. FILE2 modified
illegally via FILE1.
#
#-----------------------------------------------------------------------
# Potential trojan horse implantation or unauthorized modification of data
# Decision-8
#
DEC8:
Compromise detected: Potential Trojan horse implantation.
User has overwritten somebody else's file.
#
#-----------------------------------------------------------------------
```

```
# Exposure to SUID attacks
# Decision-9
#
DEC9:
Compromise imminent: Exposure to SUID attack.
User has a file setuid enabled.
#
#-------------------------------------------------------------------------
# Denial of service, possible TH implantation
# Decision-10
#
DEC10:
Compromise detected: Possible TH implantation. User has overwritten
one of the files in the non-writable system executables.
#
#-------------------------------------------------------------------------
# Denial of service, possible TH implantation
# Decision-11
#
DEC11:
Potential compromise detected:
User has created a file in a non-writable system directory.
#
#-------------------------------------------------------------------------
# Unauthorized deletion of user files
# Decision-12
#
DEC12:
Compromise detected: Unauthorized deletion. User has deleted somebody
else's file.
#
#-------------------------------------------------------------------------
#                          END OF DECISION TABLE
```

4.4.4 Operation of the Decision Engine

Whenever the decision engine is notified by the inference engine about a state change it displays various information to the console of the machine running USTAT for the SSO.

- It displays the number of the scenario for which this state change has occurred.

- It displays the message in the decision table that corresponds to the last satisfied state.
- It displays all the filenames that were involved in this instance of the scenario.
- It displays the real user id and the effective user id of the user who performed the last signature action for this instantiation.
- It informs the SSO whenever a row gets deleted because of a state change on the system.

Besides these, the decision engine provides a report file, named *ustat.report*, to which it records all the above results along with the relevant audit records. Two sample records of the report file follow.

```
                       ********** STD#4 **********
Warning: User executed lpr to print FILE2 owned by somebody else.
If lpr invoked with -r FILE2 will be removed illegally.

FILE1: /usr/ucb/lpr
FILE2: /etc/ttytab
USER: 5502 EUID: 0
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4643 / 700000 - Tue Jul 28 23:51:11 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/lpr
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   1   -rws--s--x
DEVICE - INODE - FSID:   7392   3080   1798
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4643 / 20000 - Tue Jul 28 23:51:12 1992
ACTION:                  READ
OBJECT:                  /etc/ttytab
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   10   -rw-r--r--
DEVICE - INODE - FSID:   208   31   1792

==================================================================

 STD#4 The following threat is no longer active
```

```
FILE1: /usr/ucb/lpr
FILE2: /etc/passwd
USER: 5502 EUID: 0
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4643 / 700000 - Tue Jul 28 23:51:11 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/lpr
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   1   -rws--s--x
DEVICE - INODE - FSID:   7392   3080   1798
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4643 / 120000 - Tue Jul 28 23:51:12 1992
ACTION:                  READ
OBJECT:                  /etc/passwd
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   0   -rw-rw-r--
DEVICE - INODE - FSID:   1001   101   1792
```

With this data, the SSO has enough information to either take preemptive action, or to take precautions to prevent further attacks.

This page is intentionally left blank

Chapter 5

Testing USTAT

This page is intentionally left blank

5 Testing USTAT

In this part of the document we give the test results of USTAT. In the first section, we explain the directory structure of USTAT and describe how to run USTAT. In the next section, we give some information about the testing environment. In the third section we list the results of the functional tests. The final section gives the results of the performance tests.

5.1 Running USTAT

5.1.1 USTAT Directory Structure

In the USTAT root directory there are two files and three subdirectories, as illustrated in Figure 5.1. One of the files is USTAT's executable code, *Ustat*, and the other is the mangrep utility, which is used to aid the user in creating filesets 3 and 4. The usage of mangrep was explained in Section 4.2.1 of Chapter 4, "Fact-base Initializer." In the following subsections we look at the contents of the subdirectories.

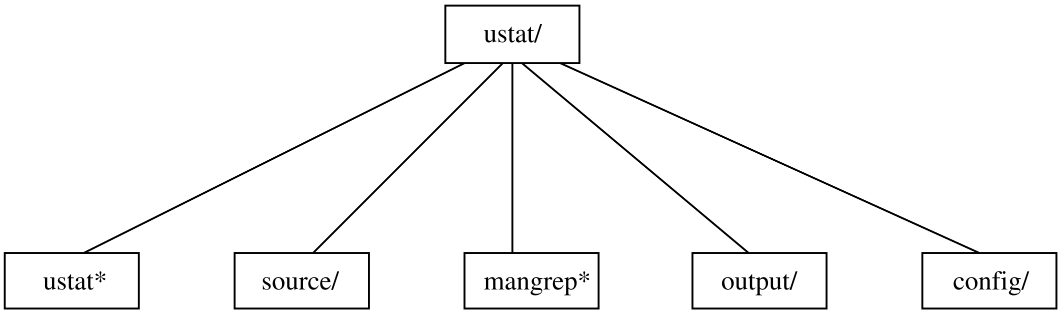


Figure 5.1 USTAT Directory Structure

5.1.1.1 The Source Files

The source files are located in the *source* subdirectory of the USTAT root directory. A makefile is provided along with the sources in case recompilation is needed. Table 5.1 lists the contents of this subdirectory, grouped by the different modules of USTAT.

Module	Source file	Explanation
The preprocessor	pre_pro.c	Reads audit trails
	pre_filter.c	Filters audit records
	pre_disp.c	Displays USTAT audit records
	pre_pro.h	Header file, data structures
The fact-base	f_bi.c	The fact-base initializer
	f_bu.c	The fact-base updater
	f_hl.c	Maintains the hardlink information
	f_b.h	Header file, data structures
	mangrep.c	A separate source file
The rule-base	rb.c	SDT and SAT reader
	rb.h	Header file, data structures
The inference engine	i_eng.c	Inference engine routines
	i_sat.c	Satisfying state assertions
	i_eng.h	Header file, data structures
The decision engine	d_eng.c	Reads and processes the decision table
Others	common.c	Routines and macros common to most modules
	common.h	Common include files and define statements
	menu.c	The menu routine provided majorly for testing
	pre-init.c	A utility to aid in creation of Filesets 3 and 4

Table 5.1 USTAT modules and related source files

5.1.1.2 The Configuration and Input Files

USTAT needs some configuration and input files to be able to run. These files are located in the *config* subdirectory of the USTAT root directory. Table 5.2 lists the configuration files.

The file named *ignore.these* was added during the test runs to reduce the high number of false alarms. In this file, each line that starts with # is ignored; these are used as comment lines. Blank lines are also ignored. For each file to be ignored, the full pathname of the file is given followed by an action name. Only the specified action is ignored for the file except for the action name ALL, in which case all actions on the filename are ignored.

Filename	Short Explanation and Purpose	Details, section:
rrf.set	The restricted-read files Fileset #1	4.2.1
rwsf.set	The restricted-write setup files Fileset #2	4.2.1
FSET3	The files authorized to read FSET1 Fileset #3	4.2.1
FSET4	The files authorized to write FSET2 Fileset #4	4.2.1
nwse.set	The non-writable system executables Fileset #5	4.2.1
nwsd.set	The non-writable system directories RWD of the fact-base	4.2.1
SDT	The state description table The rule-base	4.2.2
SAT	The signature action table The rule-base	4.2.2
DT	The decision table The decision engine	4.4
ignore.these	The filenames to be ignored Reduce false alarms	5.1.1
mangrep.dirs	Directory names for man pages Fileset #2 and #4	4.2.1

Table 5.2 USTAT Configuration Files

The following gives the contents of the current ignore_these.

```
# This file contains the name of the files and directories
# that can be ignored by the preprocessor.
# So, whenever the preprocessor encounters an audit record
# with one of these files it will ignore that record.
#
#
/usr/lib/ld.so                    read
/dev/zero                         read
```

```
/etc/ld.so.cache                 read
/usr/lib/libc.so.1.6             read
/dev/null                        ALL
/etc/utmp                        write
/etc/utmp                        read
/var/yp/binding/ucsb-cmpsci.2    read
/usr/include/bsm/audit_event.h   read
/etc/ttytab                      read
/etc/printcap                    read
/usr/lib/libbsm.so.1.1           read
```

This adds to the filtering job of the preprocessor, but reduces the number of audit records that the inference engine needs to deal with.

5.1.1.3 The Output Files

There are three output files of USTAT, which are included in the *output* subdirectory. *ustat.errors* is a second copy of the error messages that are displayed on the console of the machine that is running USTAT. *ustat.report* is a second copy of all the messages that are displayed on the console, but it also contains extensive information about the compromises that are impending and/or achieved. Examples of the contents of this file can be found in Section 5.3 of Chapter 5, "Functional Testing." *mangrep.out* is provided to aid in fact-base initialization.

5.1.1.4 Initiating USTAT

The synopsis of running USTAT is the following.

```
ustat audit_filename
```

USTAT cannot start processing audit data if one of the following holds.

- USTAT cannot open/read audit_filename.
- USTAT cannot open/read one of the configuration files.
- The format of SDT, SAT, or DT is invalid.

- There is a mismatch between any two of SDT, SAT, or DT.
- USTAT cannot open report files in the output directory.
- USTAT cannot open console and tty outputs for displaying messages.

In addition, USTAT will terminate abnormally if there is not enough memory to allocate for the inference engine data structures, or if USTAT cannot open/read next audit file. When USTAT starts it displays the following messages.

```
config/rrf.set is read into memory
config/rwsf.set is read into memory
config/FSET3.final is read into memory
config/FSET4.final is read into memory
config/nwse.set is read into memory
config/nwsd.set is read into memory

      USTAT is about to create hardlink information for the
      filesystem. This process might take hours to finish.
      Last chance to abort gracefully.

      Continue (Y/N)?
```

If *Y* is given as an answer, USTAT will go through the filesystem and create the hardlink information for the fact-base. This can be a very long process depending on the size of the filesystem. If the answer is *N*, USTAT will skip this process and use only the new hardlinks, the creation of which are recorded in audit files. The following illustrates the responses for each case.

```
      Continue (Y/N)? y

      Executing 'find / -print' to create hardlink info.
      Please be patient...


      Continue (Y/N)? n

      Current hardlinks will not be considered for
      the program execution.
```

After these messages USTAT displays a menu such as the following for the rest of the program execution. The menu is prepared primarily for testing purposes. But, it is also useful to understand the internal structure of the inference engine.

```
p - Process next audit record
i - Inference engine table
f - File reverse pointers
e - ESA reverse pointers
a - Audit record
h - Hardlink information
s - Stop when something is detected
n - Non-stop run
c - Non-stop run and continuous display of audit records

    Choice ->
```

Depending on the choice, USTAT either displays part of its current working data, or it processes new audit records. Table 5.3 illustrates the purpose of each choice.

Choice	**Function**
p	Processes the next audit record passed by the preprocessor.
i	Displays the inference engine table.
f	Displays the file reverse pointers.
e	Displays the ESA reverse pointers.
a	Displays the current audit record.
h	Display the hardlink information.
s	Keeps processing the audit records until a state transition occurs.
n	Continuously processes the audit records. Never returns to the menu.
c	Same as 'n', but also displays the audit records that are being processed.

Table 5.3 USTAT Menu Choices

To give an indication of the amount of the audit records being processed, USTAT displays a period for each audit record. When USTAT finishes processing one audit file it displays the name of the next one and continues processing. When USTAT catches up with the audit records and reaches the end of a non-terminated audit file it displays a message indicating that it is idle and waiting for more audit records. Whenever USTAT detects a compromise or part of a compromise, the decision engine generates a short message that is displayed on the console of the machine running USTAT, and a more detailed explanation of the attack, recorded in a log file named ustat.report. In Section 5.3 of Chapter 5 we illustrate the format of these outputs for various test runs.

5.2 Test Environment

All tests were run on a *SPARCstation 1*. The following is the output of the *showrev* command executed on the test machine.

```
***************  showrev version 1.7  *****************
* Hostname: "vladimir"
* Hostid: 5100920f
* Kernel Arch: "sun4c"
* Application Arch: "sun4"
* Kernel Revision:
  4.1.1 (BSM) #1: Wed Dec 11 14:09:22 PST 1991
* Release: 4.1.1
*******************************************************
```

Initializing the fact-base, especially creating the hardlink information will take different amounts of time on different target filesystems. On our filesystem we ran USTAT with regular user privileges and found 4,983 hardlinks that correspond to 1,106 physical files. This process took 37 minutes. Initializing the hardlink information makes more sense if USTAT is run in real-time. Otherwise, discrepancies might exist in the hardlink information.

To illustrate the speed and amount of audit data collected by the audit daemon, we ran three different processes. The processes we chose were audit intensive [7] CPU intensive processes create less audit records, as will be seen in Section 5.4 of Chapter 5. Table 5.4 gives the results of these. Time to finish gives the amount of time to complete the process while being audited. The next column gives the amount of time to finish without being audited. The difference gives roughly the amount of time the audit daemon used to collect this data. The last column indicates the ratio of the fourth column to

[7] These processes continuously perform a large amount of system calls that keep the audit daemon and hence the disk where the audit records are written, continuously busy.

the third column. This ratio shows that the processes listed in the table run approximately four times slower when being audited.

Event	Data Kbytes	Time to finish (audited) hr:min:sec	Time to finish (not audited) hr:min:sec	Difference hr:min:sec	Time (not audited)/ Time (audited)
Starting X-windows	161	00:03:38	00:00:47	00:02:51	0.22
Compilation of USTAT	143	00:03:58	00:01:00	00:02:58	0.25
Running COPS	4,570	01:20:45	00:18:50	01:01:55	0.23

Table 5.4 Audit Data Collection Examples

The following lists some factors that might affect the speed of USTAT.

1. Size of hardlink information.
2. Size of the filesystem (affects initialization time).
3. Number of rules in the rule-base.
4. Cpu load on the machine where USTAT is running: other users, processes, the audit daemon, etc. (If the audit daemon is run on the same machine, than USTAT should be run below the auditing mechanism, e.g., user audit might run USTAT. Otherwise, there will be a circular, infinite execution.)
5. Load of the disk on which the audit trails are recorded.
6. The network traffic load if the audit data is sent over the network.

5.3 Functional Testing

To perform the functional tests we examined three different features of USTAT.

1. The rules for the state transition diagrams. Do all of the rules work when their penetration scenarios are performed?

2. The hardlink information. Do the rules work when the attack is performed using hardlinks? For instance, the first part of a scenario can be performed by one file, whereas the second part can be performed by a hardlink to the first file.

3. Cooperative attacks. Do the rules work if parts of a scenario are performed by more than one attacker?

In the following subsections we give the results of each of these functional test cases.

5.3.1 Testing the State Transition Diagrams

In this section we give the results of testing each of the state transition diagrams, which were explained in Section 4.2.2 of Chapter 4. For each test we give a short reminder of the attack scenario, we give one possible command or command sequence that can be used to perform the attack. The commands are listed as actions. However, sometimes one command corresponds to more than one signature action and vice versa. We also give the output of USTAT in response to these actions. The output consists of two parts. The first part is a short message that is displayed on the console of the machine running USTAT. The second part is a more detailed explanation of the attack that consists of the filenames and the user-id's involved in the scenario along with the audit records that are used by the inference engine to detect the compromise. The second part is recorded in the log file named ustat.report. When we give the output of USTAT, we only include those responses that are relevant to the particular scenario being tested.

5.3.1.1 Testing Scenario 1

Description: The victim (user-id: 9) has a setuid shell script, located in */tmp* and named *foo*. The attacker (user-id: 5502) creates a hardlink to this shell script and then executes the shell script through the new name, which starts with a '-'.

Action-1: The attacker creates the hardlink in the same directory as the target file.

```
% cd /tmp
```

```
% ln foo -x
```

Response-1:

Console:

```
STD#1
Suspicious link FILE1 created by user to setuid file FILE2.
If executed, FILE1 will invoke an interactive setuid shell
with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-a
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#1 **********
Suspicious link FILE1 created by user to setuid file FILE2.
If executed, FILE1 will invoke an interactive setuid shell
with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-a
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              1290 / 920000 - Fri Aug 21 09:53:47 1992
ACTION:                  HARDLINK
OBJECT:                  /tmp/foo
TARGET:                  /tmp/-a
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
```

Action-2: Now the attacker executes the hardlink thereby gaining an interactive shell with victim's privileges.

```
% -x
```

Response-2:

Console:

```
STD#1
Compromise detected: Unauthorized access to user privileges. Shell
invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-a
```

```
USER: 5502 EUID: 9
```

Report File:

```
                  ********** STD#1 **********
Compromise detected: Unauthorized access to user privileges. Shell
invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-a
USER: 5502 EUID: 9
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              1290 / 920000 - Fri Aug 21 09:53:47 1992
ACTION:                  HARDLINK
OBJECT:                  /tmp/foo
TARGET:                  /tmp/-a
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              1295 / 420000 - Fri Aug 21 09:55:22 1992
ACTION:                  EXEC
OBJECT:                  /tmp/-a
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
```

Comments: This attack scenario works only with certain letters that follow the '-'. The letters that are applicable to this scenario are (primarily) the ones that can be given as options when executing the sh command. The synopsis for sh in the man pages is the following.

```
sh [ -acefhiknstuvx ] [ arguments ]
```

Some combinations of these letters may also work, but this detail is not important for the functionality of the first state transition diagram.

5.3.1.2 Testing Scenario 2

Description: The attacker creates a counterfeit mail file in the /var/spool/mail directory for the victim (user-id: 9). The attacker then makes the mail file readable by everyone.

Action-1:

```
% touch /var/spool/mail/audit
% chmod o+r /var/spool/mail/audit
```

Response-1:

Console:

```
STD#2
Compromise detected: User mail file became vulnerable to read/write attacks.

FILE1: /var/spool/mail/audit
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#2 **********
Compromise detected: User mail file became vulnerable to read/write attacks.

FILE1: /var/spool/mail/audit
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              1713 / 120000 - Fri Aug 21 11:32:03 1992
ACTION:                  MODIFY_PERM
OBJECT:                  /var/spool/mail/audit
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rw----r--
DEVICE - INODE - FSID:   0   16   33286
```

Comments: This rule will not work if the attack is done without using the chmod command. The counterfeit, readable mail file can also be created by first changing the umask to 0 and then "touch"ing the mail file. This way the attack would slip unnoticed. However, it is easy to overcome this problem. We can add another state transition diagram to the rule-base that looks exactly like this one except that the signature action shows a create action.

Another result of this test revealed an important issue in creating the state transition diagrams. The pathnames used in the states of a state transition diagram should always indicate direct paths, that is, there should not be any symbolic link along the pathname, since the auditing system records the pathname using the direct paths, even if the file is accessed through a symbolic link.

5.3.1.3 Testing Scenario 3

Description: The attacker (user-id: 5502) examines the victim's (user-id: 9) world readable/executable setuid shell script foo, and notices that the script invokes an executable file called bar. The attacker creates a counterfeit, world executable bar which consists of a single line (exec /bin/sh). The attacker also changes his/her path to search the directory first where the counterfeit bar is located. By executing foo the attacker receives an interactive shell with the victim's privileges.

Action-1:

```
% cat /tmp/foo
#!/bin/sh
bar
% ls -al foo
-rwsr-xr-x 16 audit          14 Aug 25 10:13 foo*
% cat > bar
exec /bin/sh
^D
% chmod 755 bar
% set path=(. $path)
% /tmp/foo
$
```

Response-1:

Console:

```
STD#3
Warning: User executing somebody else's SETUID shell script.

FILE1: /tmp/foo
USER: 5502 EUID: 9
```

Report File:

```
              ********** STD#3 **********
Warning: User executing somebody else's SETUID shell script.

FILE1: /tmp/foo
```

```
USER: 5502 EUID: 9
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              1950 / 640000 - Fri Aug 21 12:12:49 1992
ACTION:                  EXEC
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
```

Action-2: There is no other action for this scenario. The execution of /tmp/foo continues and a read event of the counterfeit bar is recorded by the auditing system. This event causes another state transition.

Response-2:

None.

Action-3: At this point, the counterfeit bar invokes a shell, which causes a state transition to the final state of this state transition diagram.

Response-3:

Console:

```
STD#3
Compromise detected: Unauthorized access to user privileges. Shell
invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/a
FILE3: /usr/bin/sh
USER: 5502 EUID: 9
```

Report File:

```
                    ********** STD#3 **********
Compromise detected: Unauthorized access to user privileges. Shell
invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/a
FILE3: /usr/bin/sh
USER: 5502 EUID: 9
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              1950 / 640000 - Fri Aug 21 12:12:49 1992
ACTION:                  EXEC
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
```

```
DEVICE - INODE - FSID:   129   4   1800
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              1951 / 720000 - Fri Aug 21 12:12:49 1992
ACTION:                  READ
OBJECT:                  /a/galaxy/fs.real/gsl/home/grad/koral/a
TARGET:                  (null)
OWNER - GOWNER - PERM:   5502   24   -rwxr-xr-x
DEVICE - INODE - FSID:   46591   81678   33283
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              1951 / 750000 - Fri Aug 21 12:12:49 1992
ACTION:                  EXEC
OBJECT:                  /usr/bin/sh
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   10   -rwxr-xr-x
DEVICE - INODE - FSID:   3888   1545   1798
```

5.3.1.4 Testing Scenario 4

Description: The attacker (user-id: 5502) executes the lpr command using the -r option, which removes the file after it is printed. The attacker attacks the victim's (user-id: 9) file called /tmp/foo.

Action-1:

```
% lpr -r /tmp/foo
```

Response-1:

None.

Action-2: lpr command continues and reads the file to be printed.

Response-2:

Console:

```
STD#4
Warning: User executed lpr to print FILE2 owned by somebody else.
If lpr invoked with -r FILE2 will be removed illegally.

FILE1: /usr/ucb/lpr
FILE2: /tmp/foo
```

```
USER: 5502 EUID: 0
```

Report File:

```
                    ********** STD#4 **********
Warning: User executed lpr to print FILE2 owned by somebody else.
If lpr invoked with -r FILE2 will be removed illegally.

FILE1: /usr/ucb/lpr
FILE2: /tmp/foo
USER: 5502 EUID: 0
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              2054 / 360000 - Fri Aug 21 13:15:53 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/lpr
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   1   -rws--s--x
DEVICE - INODE - FSID:   7392   3080   1798
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              2054 / 920000 - Fri Aug 21 13:15:53 1992
ACTION:                  READ
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwxr--r--
DEVICE - INODE - FSID:   130   5   1800
```

Action-3: At this point the file that is printed gets deleted, since lpr was invoked with the -r option.

Response-3:

Console:

```
STD#4
Compromise detected: FILE2 got deleted illegally using lpr -r.

FILE1: /usr/ucb/lpr
FILE2: /tmp/foo
USER: 5502 EUID: 0
```

Report File:

```
                    ********** STD#4 **********
Compromise detected: FILE2 got deleted illegally using lpr -r.

FILE1: /usr/ucb/lpr
FILE2: /tmp/foo
```

```
USER: 5502 EUID: 0
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              2054 / 360000 - Fri Aug 21 13:15:53 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/lpr
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   1   -rws--s--x
DEVICE - INODE - FSID:   7392   3080   1798
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              2054 / 920000 - Fri Aug 21 13:15:53 1992
ACTION:                  READ
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwxr--r--
DEVICE - INODE - FSID:   130   5   1800
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              2054 / 130000 - Fri Aug 21 13:15:54 1992
ACTION:                  DELETE
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwxr--r--
DEVICE - INODE - FSID:   130   5   1800
```

Comments:

- The execution of lpr causes the creation of some root owned, temporary files. In our case these files are created in /var/spool/lpd/rsl directory. These files get deleted once the print job is finished. Although the audit records corresponding to these deletions indicate normal activities, they raise many false alarms since they look like the above scenario. To prevent these false alarms, one more state description can be added to the second state of this state transition diagram, the contents of which will depend on the target filesystem, e.g.,

  ```
  not fullname (file2) = "/var/spool/lpd/rsl/*"
  ```

  By adding this state assertion, the deletion of the temporary files will be ignored by the inference engine.

- Fortunately, in SunOS 4.1.1 lpr makes some user access checks before deleting the file. When a user tries to delete somebody else's file with lpr -r, lpr gives the following message:

  ```
  lpr: foo: is not removable by you
  ```

However, lpr fails to perform a proper access check if the file is located in /tmp directory (or in any other directory with the same attributes). /tmp directory is setgid to wheel and has sticky bit set. The sticky bit prevents unauthorized deletion of files via rm. However, lpr -r can delete them.

5.3.1.5 Testing Scenario 5

Description: In this scenario the attacker (user-id: 5502) executes the victim's (user-id: 9) setuid shell script /tmp/foo. The action may not necessarily indicate a malicious action, but we decided to have the decision engine give a warning since there are various attacks involved with setuid programs.

Action-1:

```
% /tmp/foo
```

Response-1:

Console:

```
STD#5
Warning: User executing somebody else's SETUID shell script.

FILE1: /tmp/foo
USER: 5502 EUID: 9
```

Report File:

```
                  ********** STD#5 **********
Warning: User executing somebody else's SETUID shell script.

FILE1: /tmp/foo
USER: 5502 EUID: 9
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:              2110 / 730000 - Fri Aug 21 13:35:23 1992
ACTION:                  EXEC
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
```

Comments: Having this warning, the SSO can analyze the contents of the shell script and determine whether it can be involved in a malicious action.

5.3.1.6 Testing Scenario 6

Description: To be able to test this scenario, we removed /usr/kvm/ps temporarily from Fileset #3 (files that are allowed to read the files in Fileset #1). ps command accesses /dev/mem and /dev/kmem, which are not readable by anyone except root and members of group kmem. The "attacker" executes ps and raises alarms.

Action-1:

```
% ps
```

Response-1:

None.

Action-2: As a result of the first action ps accesses several restricted-read files.

Response-2:

Console:

```
STD#6
Compromise detected: Restricted read violation.
FILE2 referenced illegally via FILE1.

FILE1: /usr/kvm/ps
FILE2: /dev/kmem
USER: 5502 EUID: 5502

-------------------------------------
STD#6
Compromise detected: Restricted read violation.
FILE2 referenced illegally via FILE1.

FILE1: /usr/kvm/ps
FILE2: /dev/mem
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#6 **********
```

```
Compromise detected: Restricted read violation.
FILE2 referenced illegally via FILE1.

FILE1: /usr/kvm/ps
FILE2: /dev/kmem
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              2137 / 180000 - Fri Aug 21 13:46:46 1992
ACTION:                  EXEC
OBJECT:                  /usr/kvm/ps
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   2   -rwxr-sr-x
DEVICE - INODE - FSID:   41008   18515   1798
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              2137 / 720000 - Fri Aug 21 13:46:46 1992
ACTION:                  READ
OBJECT:                  /dev/kmem
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   2   -rw-r-----
DEVICE - INODE - FSID:   769   2327   1792
```

5.3.1.7 Testing Scenario 7

Description: Similar to the modification made to Fileset #3 for the above scenario, we added a user's .cshrc file to the restricted-write files, Fileset #1, so that a flag will be raised if the file gets overwritten. The attacker modifies the .cshrc file with vi and then saves it. Since vi is not listed in Fileset #4, this action raises a flag.

Action-1:

```
% vi .cshrc
```

Response-1:

None.

Action-2: The attacker exits vi by typing *:wq*.

Response-2:

Console:

```
STD#7
Compromise detected: Restricted write violation.
FILE2 modified illegally via FILE1.
```

```
FILE1: /usr/ucb/vi
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/.cshrc
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#7 **********
Compromise detected: Restricted write violation.
FILE2 modified illegally via FILE1.

FILE1: /usr/ucb/vi
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/.cshrc
USER: 5502 EUID: 5502
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              2299 / 680000 - Fri Aug 21 14:14:20 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/vi
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   10   -rwxr-xr-x
DEVICE - INODE - FSID:   5720   1678   1798
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              2299 / 150000 - Fri Aug 21 14:14:22 1992
ACTION:                  CREATE WRITE
OBJECT:                  /a/galaxy/fs.real/gsl/home/grad/koral/.cshrc
TARGET:                  (null)
OWNER - GOWNER - PERM:   5502   24   -rw-r--r--
DEVICE - INODE - FSID:   0   81556   33283
```

5.3.1.8 Testing Scenario 8

Description: The attacker (user-id: 5502) overwrites the victim's (user-id: 9) file called /tmp/foo.

Action-1:

```
% vi /tmp/foo
```

The attacker exits vi by typing *:wq*.

Response-1:

Console:

```
STD#8
Compromise detected: Unauthorized modification of data.
```

```
User has overwritten somebody else's file.

FILE1: /tmp/foo
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#8 **********
Compromise detected: Unauthorized modification of data.
User has overwritten somebody else's file.

FILE1: /tmp/foo
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              4321 / 210000 - Mon Aug 24 11:08:08 1992
ACTION:                  CREATE WRITE
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwsr-xrwx
DEVICE - INODE - FSID:   0   4   1800
```

5.3.1.9 Testing Scenario 9

Description: In this scenario a user enables the setuid bit of his/her file. There is no attacker involved, but a new setuid file might be a new threat to the system security.

Action-1:

```
% chmod 4755 p
```

Response-1:

Console:

```
STD#9
Compromise imminent: Exposure to SUID attack.
User has a file setuid enabled.

FILE1: /a/galaxy/fs.real/gsl/home/grad/koral/bin/p
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#9 **********
Compromise imminent: Exposure to SUID attack.
User has a file setuid enabled.

FILE1: /a/galaxy/fs.real/gsl/home/grad/koral/bin/p
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              4378 / 830000 - Mon Aug 24 11:23:29 1992
ACTION:                  MODIFY_PERM
OBJECT:                  /a/galaxy/fs.real/gsl/home/grad/koral/bin/p
TARGET:                  (null)
OWNER - GOWNER - PERM:   5502   24   -rwsr-xr-x
DEVICE - INODE - FSID:   32087   75401   33283
```

5.3.1.10 Testing Scenario 10

Description: To test this scenario /tmp/* is temporarily added to the non-writable system executables (Fileset #5). The attacker performs this scenario by overwriting a file, /tmp/foo, which is now in Fileset #5. The attacker uses vi to alter the file and then quits by typing *:wq*.

Action-1:

```
% vi /tmp/foo
```

Response-1:

Console:

```
STD#10
Compromise detected: Possible TH implantation. User has overwritten
one of the files in the non-writable system executables.

FILE1: /tmp/foo
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#10 **********
Compromise detected: Possible TH implantation. User has overwritten
one of the files in the non-writable system executables.

FILE1: /tmp/bar
```

```
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              4409 / 320000 - Mon Aug 24 11:28:27 1992
ACTION:                  CREATE WRITE
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   0   -rwxr-xrwx
DEVICE - INODE - FSID:   0   4   1800
```

5.3.1.11 Testing Scenario 11

Description: Again temporarily, the /tmp directory is added to the non-writable system directories. The attacker creates a file /tmp/foo.

Action-1:

```
% touch /tmp/foo
```

Response-1:

Console:

```
STD#11
Potential compromise detected:
User has created a file in a non-writable system directory.

FILE1: /tmp/foo
USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#11 **********
Potential compromise detected:
User has created a file in a non-writable system directory.

FILE1: /tmp/foo
USER: 5502 EUID: 5502
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              4460 / 500000 - Mon Aug 24 11:33:25 1992
ACTION:                  CREATE WRITE
OBJECT:                  /tmp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   5502   0   -rw-------
DEVICE - INODE - FSID:   0   5   1800
```

5.3.1.12 Testing Scenario 12

Description: The attacker (user-id: 5502) deletes a file owned by the victim (user-id: 9). Although the file has permission to be deleted (the directory is writable by anyone), it violates the rule: "Nobody shall delete somebody else's file."

Action-1:

```
% rm foo
```

Response-1:

Console:

```
STD#12
Compromise detected: Unauthorized deletion.
User has deleted somebody else's file.

FILE1: /a/galaxy/fs.real/gsl/home/grad/koral/temp/foo

USER: 5502 EUID: 5502
```

Report File:

```
                    ********** STD#12 **********
Compromise detected: Unauthorized deletion.
User has deleted somebody else's file.

FILE1: /a/galaxy/fs.real/gsl/home/grad/koral/temp/foo
USER: 5502 EUID: 5502
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              4483 / 960000 - Mon Aug 24 11:37:19 1992
ACTION:                  DELETE
OBJECT:                  /a/galaxy/fs.real/gsl/home/grad/koral/temp/foo
TARGET:                  (null)
OWNER - GOWNER - PERM:   9   24   -rw-------
DEVICE - INODE - FSID:   35875   76807   33283
```

Comments: During the execution of most system programs many root owned temporary files are created and then deleted upon termination of the programs. This results in many false alarms. Since most of these deletions are done while the user's effective user-id is set to root, the following state assertion is added to the final state of this scenario to prevent false alarms.

```
not euid = ROOT
```

5.3.2 Testing the Functionality of the Hardlink Information

In this part we apply another type of test to USTAT to make sure that the hardlink information that is maintained throughout the run of USTAT is used correctly for detecting compromises that involve hardlinks. For instance, consider the following situation. Assume that file foo is a member of the restricted-read fileset (Fileset #1). That is, any read access to this file except the accesses performed through the members of Fileset #3 (files authorized to read Fileset #1) should be detected as a compromise. The detection is accomplished by using the sixth state transition diagram. A variation to this scenario can be achieved by using hardlinks. First, the attacker creates a hardlink to foo and then makes an illegitimate access to foo through the hardlink. Since the hardlinks to foo are not listed in Fileset #1, this compromise would not be noticed under normal circumstances. However, USTAT makes use of the hardlink information to detect such variations. For more information about USTAT's hardlink information structure, refer to Section 4.2.1 of Chapter 4.

The following illustrates the test run used to check the functionality of the hardlink information.

Description: To be able to create a simple test case we add a new file,

```
/a/galaxy/fs.real/gsl/home/grad/koral/foo
```

to the restricted-read files set. Any read access to this file except the accesses done by a program in Fileset #3 should be detected as a compromise according to the sixth state transition diagram. The attacker (user-id: 5502) creates a hardlink to this file and then accesses the original file through the new filename.

Action-1:

```
% ln foo bar
% cat bar
```

Response-1:

None.

Action-1 for this scenario corresponds to the execution of cat that doesn't cause an alarm condition.

Action-2:

Action-2 corresponds to the read access to bar.

Response-2:

Console:

```
STD#6
Compromise detected: Restricted read violation.
FILE2 referenced illegally via FILE1.

FILE1: /usr/bin/cat
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/bar
USER: 5502 EUID: 5502
```

Report File:

```
                        ********** STD#6 **********
Compromise detected: Restricted read violation.
FILE2 referenced illegally via FILE1.

FILE1: /usr/bin/cat
FILE2: /a/galaxy/fs.real/gsl/home/grad/koral/bar
USER: 5502 EUID: 5502
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              6565 / 340000 - Tue Sep  1 11:49:10 1992
ACTION:                  EXEC
OBJECT:                  /usr/bin/cat
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   10   -rwxr-xr-x
DEVICE - INODE - FSID:   6136   1706   1798
-------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 5502 - 24
PID / Time:              6565 / 400000 - Tue Sep  1 11:49:10 1992
ACTION:                  READ
OBJECT:                  /a/galaxy/fs.real/gsl/home/grad/koral/bar
TARGET:                  (null)
OWNER - GOWNER - PERM:   5502   24   -rw-------
DEVICE - INODE - FSID:   44472   81409   33283
```

5.3.3 Testing Cooperative Attackers

Not all of the scenarios can be performed by the cooperation of more than one attacker. In fact, scenario 1 is the only one that can allow two attackers to cooperate. To test this scenario with cooperative attackers, we need at least three subjects: One victim, two attackers.

The victim (user-id: 9) has a setuid shell script, called foo, located in /tmp. The first attacker (user-id: 659) logs on to the target host and creates a hardlink to foo. Later, the second attacker logs on and executes the hardlink, and thereby receives an interactive shell with victim's privileges.

The first attacker logs on, executes the following and logs out.

Action-1:

```
% cd /tmp
% ln foo -x
```

Response-1:

Console:

```
STD#1
Suspicious link FILE1 created by user to setuid file FILE2.
If executed, FILE1 will invoke an interactive setuid shell
with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-x
USER: 659 EUID: 659
```

Report File:

```
                        ********** STD#1 **********
Suspicious link FILE1 created by user to setuid file FILE2.
If executed, FILE1 will invoke an interactive setuid shell
with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-x
USER: 659 EUID: 659
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 659 - 659 - 24
PID / Time:              27072 / 160000 - Tue Sep 27 11:45:29 1992
ACTION:                  HARDLINK
OBJECT:                  /tmp/foo
TARGET:                  /tmp/-x
OWNER - GOWNER - PERM:   9   0   -rwsr-xr-x
DEVICE - INODE - FSID:   129   4   1800
```

Action-2: The second attacker logs on, executes the following and receives the shell.

```
% cd /tmp
% -x
$
```

Response-2:

Console:

```
STD#1
Compromise detected: Unauthorized access to user privileges.
Shell invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-x


USER: 5502 EUID: 9
```

Report File:

```
                        ********** STD#1 **********
Compromise detected: Unauthorized access to user privileges.
Shell invoked interactively via FILE1 with another user's privileges.

FILE1: /tmp/foo
FILE2: /tmp/-x
USER: 5502 EUID: 9
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 659 - 659 - 24
PID / Time:             27072 / 160000 - Tue Sep 27 11:45:29 1992
ACTION:                 HARDLINK
OBJECT:                 /tmp/foo
TARGET:                 /tmp/-x
OWNER - GOWNER - PERM:  9   0   -rwsr-xr-x
DEVICE - INODE - FSID:  129   4   1800
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 9 - 24
PID / Time:             27085 / 70000 - Tue Sep 29 10:51:31 1992
ACTION:                 EXEC
OBJECT:                 /tmp/-x
TARGET:                 (null)
OWNER - GOWNER - PERM:  9   0   -rwsr-xr-x
DEVICE - INODE - FSID:  129   4   1800
```

For USTAT it makes no difference whether this attack is performed by only one user or two users. Therefore, the response is the same as the response in test case 1, except for the user-id for the first action.

5.4 Performance Testing

In this section, we test USTAT's performance first while it is being run as the only process on the system, and then while other processes coexist on the system. The auditing process, the audit daemon, is always active. During these tests all processes except USTAT are audited by the audit daemon. USTAT itself is not audited since otherwise it would result in a circular action.

5.4.1 Benchmarks

The tests listed in Table 5.5 were performed on various audit data sizes using 12 rules in the rule-base, without initializing the hardlink information and with no other major cpu load on the machine that was running USTAT. The table is sorted by the increasing values of U/A.

Test indicates the test number.
Data is the size of the total audit record file processed in kilobytes.
Audit is the number of audit records in the test data.
Ustat is the number of audit records that are used by USTAT.
U/A gives the ratio between Ustat and Audit.
Time is the amount of time elapsed to process these audit records.
Kbps is the number of kilobytes processed per second.
Aps is the number of audit records processed per second.
Ups is the number of USTAT audit records processed per second.
Max is the maximum number of rows in the inference engine table reached during processing.
Curr is the number of rows in the inference engine table at the end of processing.

Data (Kb)	Audit	USTAT	U/A	Time mm:ss	Kbps	Aps	Ups	Max	Curr
2,758	25,623	4,351	0.170	2:06	21.9	203	35		
3,442	30,780	5,537	0.180	2:34	22.4	200	36	70	64
4,174	38,921	7,585	0.195	3:51	18.1	169	33		
2,925	27,017	5,442	0.201	2:35	18.9	174	35		
7,370	68,249	14,150	0.207	7:00	17.6	163	34		
11,536	100,139	22,710	0.227	11:13	17.2	149	34	239	239
1,526	13,916	3,245	0.233	1:26	17.8	162	38		
912	8,707	2,196	0.252	0:58	15.7	150	38	12	12
2,223	19,142	4,886	0.255	2:15	16.5	142	36	110	110
5,127	47,669	13,865	0.291	6:27	13.3	123	36	76	39
10,525	94,033	28576	0.304	14:20	12.5	231	34	77	39
3,567	30,879	9,974	0.323	5:55	10.1	87	28		
6,146	53,114	19,302	0.363	9:59	10.3	89	32		
2,041	16,963	6,729	0.397	4:25	7.7	64	25		
16,687	156,016	63,208	0.405	20.36	13.5	126	51	94	84
677	5,584	2,288	0.409	0:59	11.5	95	39	62	56
12,512	115,603	47,954	0.415	17:47	11.7	108	45		
51,259	480,643	213,965	0.445	1:03:41	13.4	126	56	150	140
7,445	71,203	34,701	0.487	10:03	12.4	118	58		
8,334	76,636	40,337	0.526	13:37	10.2	94	49		
13,559	122,830	69,540	0.570	23:30	9.6	87	49		
4,906	46,987	30,818	0.656	8:53	9.2	88	58	30	18
4,570	44,186	29,259	0.662	7:17	10.5	101	67		
4,027	83,977	59,033	0.703	27:46	2.4	50	35		
893	7,343	5,249	0.715	1:46	8.4	69	50	60	44

Table 5.5 Test runs on various data sizes

Kbps gives us an idea about the speed of processing the audit data. However, as we see from the table, this number varies depending on the test. We expect that the larger the percentage of USTAT audit records in the audit data, the Kbps should decrease, since USTAT needs to process relatively more audit data. To justify this hypothesis we drew a graph relating the Ustat to total audit ratio (U/A) with Kbps. On the graph of Figure 5.2 the horizontal axis corresponds to U/A values and the vertical axis corresponds to Kbps values. On this graph we see that if U/A increases, Kbps decreases and vice versa. The graph also indicates major fluctuations, especially the two low spikes of the graph. Although it is hard to prove, these fluctuations might be caused by any of the factors that are listed as the "factors that might affect the speed of USTAT" in Section 5.2 of Chapter 5.

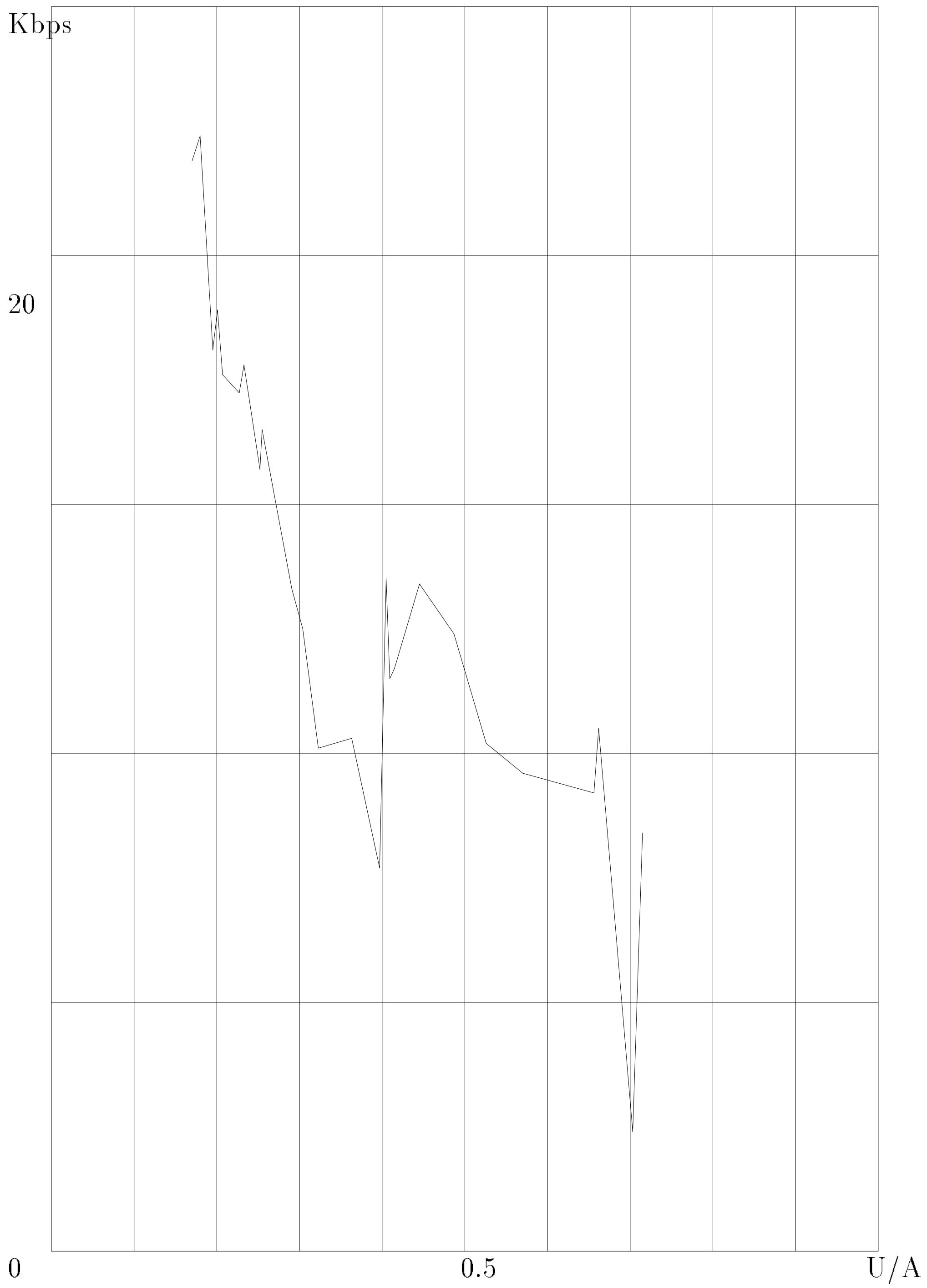


Figure 5.2 Graph of U/A vs Kbps

5.4.2 Various Test Cases

In this section we report on tests where USTAT was run with a combination of several other I/O intensive and cpu intensive processes. To collect data, ps was run periodically to obtain cpu and I/O values. It is obvious that ps also uses some cpu cycles and therefore affects the test results.

In these tests, USTAT was using the audit data that had been previously collected. It can be considered a batch-mode execution, but it becomes real-time when USTAT catches up with the current audit data. The following are the processes that were used in the test.

- USTAT: Both cpu and I/O (primarily input) intensive, unless it is waiting for audit data.
- Crack: A cpu intensive program that runs encryption routines to guess passwords.
- find: An I/O intensive (primarily input) system program.
- Database simulation (DSIM): A cpu intensive database simulation program written in SIMSCRIPT.
- Physics simulation (PSIM): Another CPU intensive simulation program used for hybrid molecular dynamics simulations of Kogut-Susskind fermions.
- Audit daemon: I/O intensive (primarily output). The audit daemon was in the test by default, since USTAT and the audit daemon were running on the same machine.

5.4.2.1 Test Case 1: USTAT only

In this test, we tested USTAT's performance while no other major processes were running. USTAT read audit data for about 43 minutes. After this point it became idle and waited for more audit data. The graph in Figure 5.4 displays USTAT's cpu usage over time. The lowest and highest values for cpu usage are 34.8 % and 51.2 %, respectively.

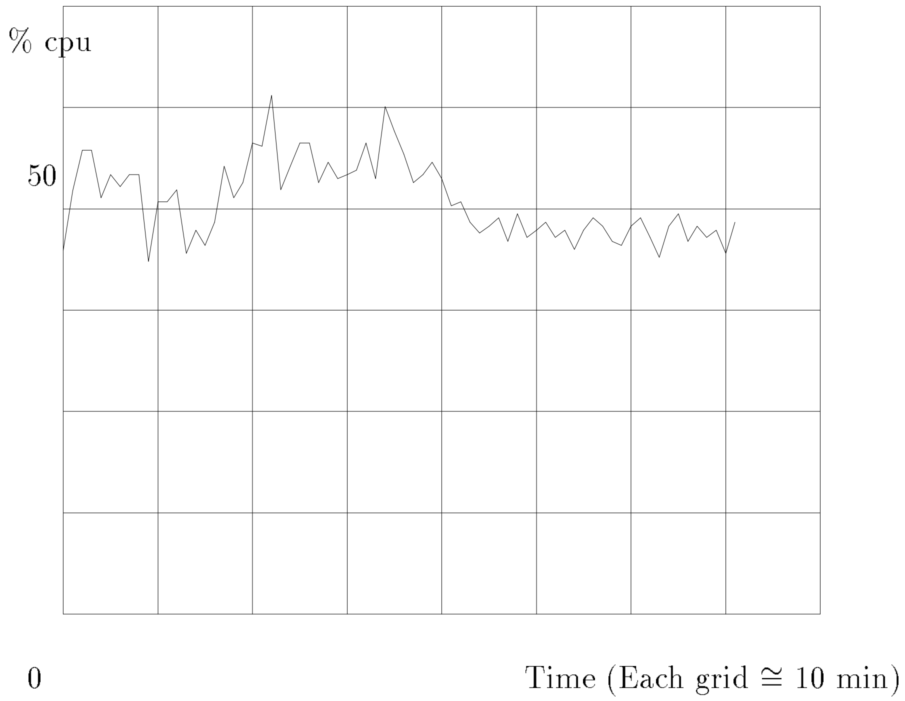


Figure 5.3 USTAT cpu usage in running USTAT

Each vertical grid corresponds to approximately 10 minutes of runtime. USTAT used about 42 % of cpu during processing and 38 % when idle. The cpu utilization during this idle period seems high.

More than 30 minutes (i.e. more than 50 %) of cpu time was idle even during USTAT's processing (See Table 5.6). This indicates that for half its runtime USTAT was waiting for the I/O to be completed.

Process	**Cpu time**	**Percentage**
Auditd	0:05	0.1
USTAT	32:17	45
Idle	38:56	55
Total	1:11:18	100

Table 5.6 Cpu usages [8] of Auditd and USTAT

[8]Idle includes other processes

5.4.2.2 Test Case 2: USTAT and Crack

In this test run, the cpu was rarely idle. Because both are user processes, USTAT and crack competed for cpu time. Compared to the previous test case, in this one crack seemed to fill the unused cpu cycles (see crack's cpu usage in Table 5.7). Cpu utilization of USTAT dropped by 5 %.

Figures 5.4 and 5.5 display the cpu usages of USTAT and crack over time.

Process	**Cpu time**	**Percentage**
Auditd	0:06	0.3
USTAT	14:51	43
Crack	14:01	40
Idle	5:58	17
Total	34:56	100

Table 5.7 Cpu usages of Auditd, USTAT and Crack

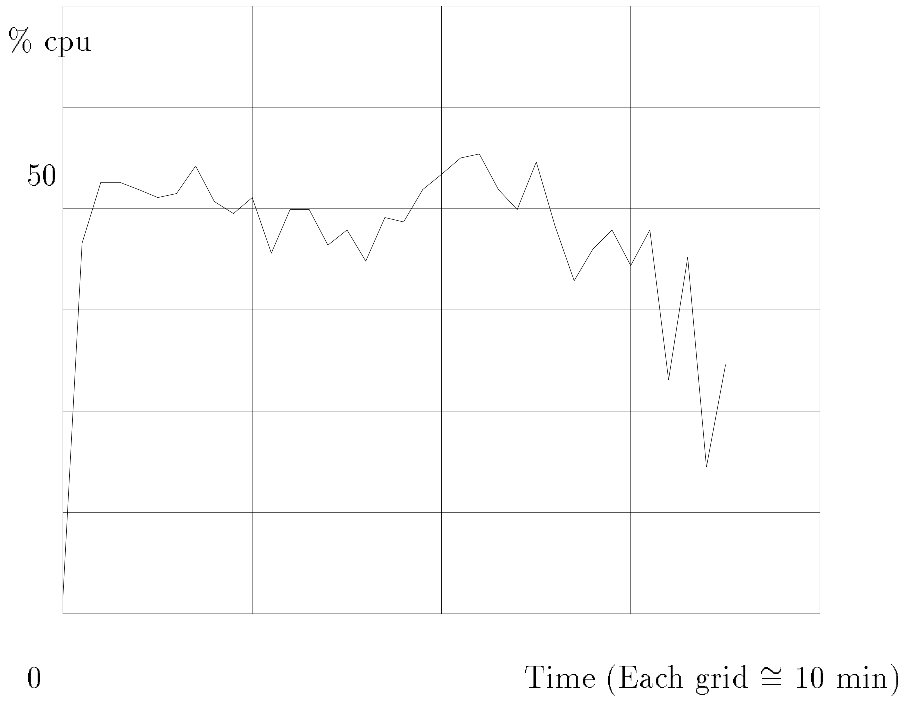


Figure 5.5 USTAT cpu usage in running USTAT and Crack

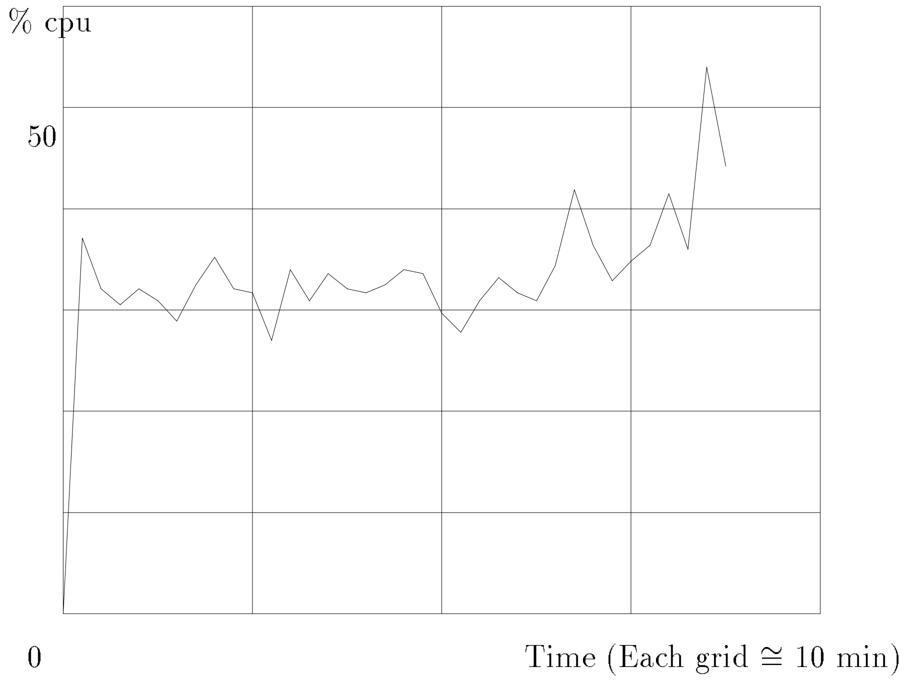


Figure 5.6 Crack cpu usage in running USTAT and Crack

5.4.2.3 Test Case 3: USTAT and find

Being an audit intensive process, find made a tremendous amount of calls that push the audit daemon to its limit. Since the audit daemon runs as a root process it caused a considerable slow-down of the user processes. The processes that required disk I/O from the same disk that was used by the audit daemon, were very likely to hang. For instance, an ls command on the audit data directory hung. Since USTAT is also a user process requiring disk I/O from the audit disk, it hung after a while. (See Graph of Figure 5.10). The only solution to this hang was to terminate the execution of find, thereby relieving the audit daemon and giving USTAT some opportunity to read the audit data. In this test, USTAT indicated 19 percent of cpu usage. This is more than a 50 % slow-down in the processing speed. Without terminating the "find" command, USTAT's cpu utilization would approach zero (See Table 5.8). More than 60 % of cpu time was idle, since USTAT didn't have much opportunity to process audit data.

Process	**Cpu time**	**Percentage**
Auditd	6:16	12
USTAT	10:24	19
find	2:36	5
Idle	34:14	64
Total	53:30	100

Table 5.8 Cpu usages of Auditd, USTAT and find

When we look at the run-time priorities of the audit daemon and USTAT by using "ps -l" we see that the audit daemon's priority remains 1 all the time, whereas USTAT's priority value continuously increases (i.e. decrease in actual priority). One solution to the hang-up of USTAT might be to run it as a root process and give a better or equal priority.

Figures 5.7 through 5.9 display the cpu usages of auditd, find and USTAT over time. In these three figures each vertical grid equals approximately 10 minutes.

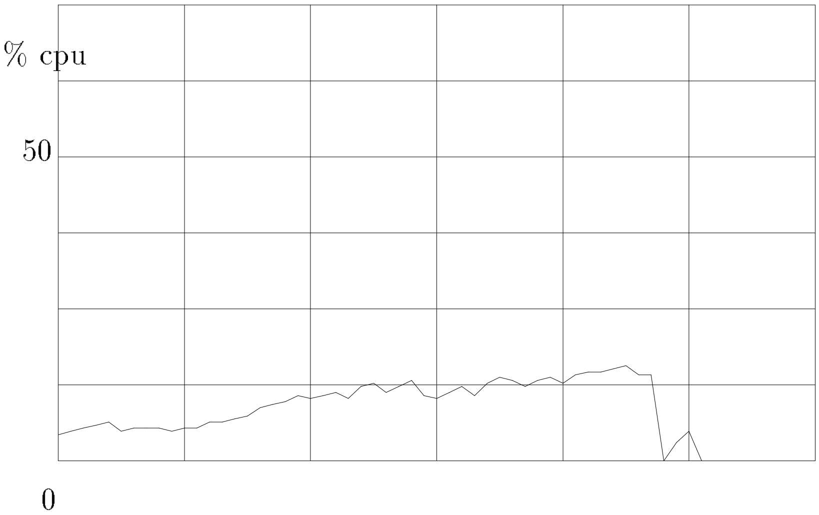


Figure 5.7 Auditd cpu usage in running USTAT and find

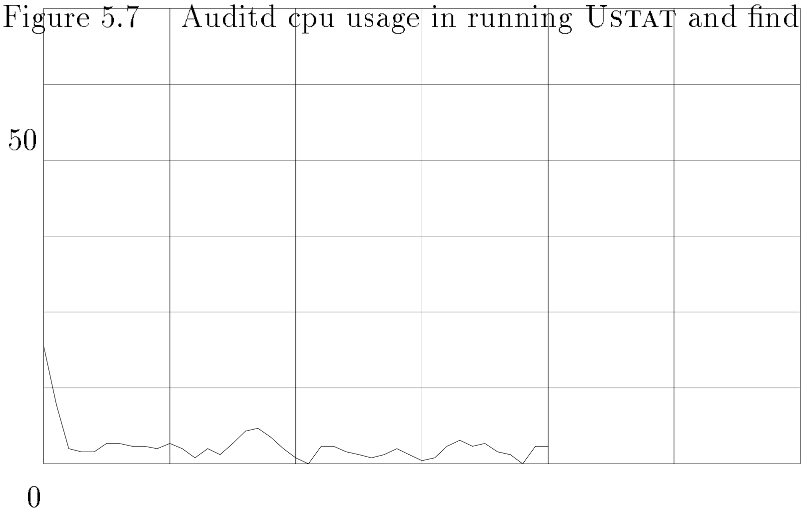


Figure 5.8 Find cpu usage in running USTAT and find

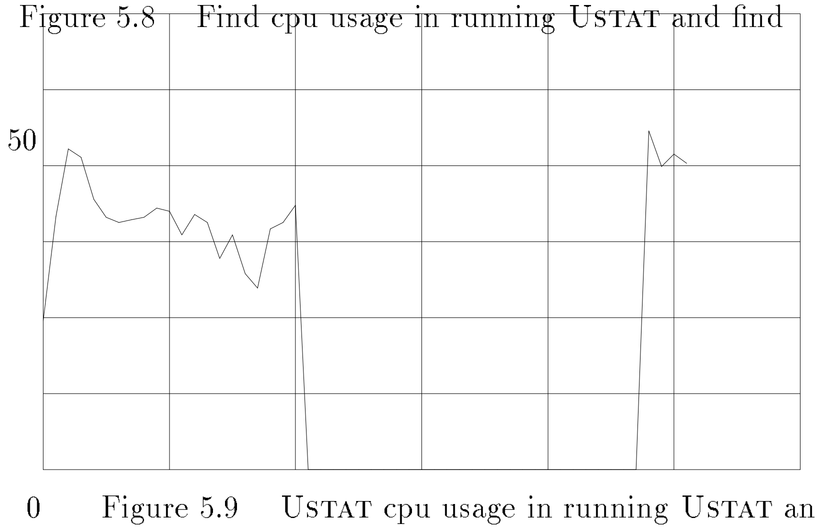


Figure 5.9 USTAT cpu usage in running USTAT and find

5.4.2.4 Test Case 4: USTAT, Crack and find

Similar to the result of Test Case 3, USTAT hung after about 15 minutes of runtime. At this point we killed the process that was running find. In about 5 minutes the audit daemon caught up with real time, released the disk, and USTAT continued processing. Crack meanwhile, utilized the extra cpu cycles, while USTAT was waiting for the disk I/O. Table 5.9 lists the cpu usages of the processes involved in this test case. Figures 5.10 through 5.13 display cpu usages of the four different processes involved in this test.

Process	Cpu time	Percentage
Auditd	2:36	10
USTAT	7:23	29
Crack	10:42	42
find	1:40	7
Idle	3:15	12
Total	25:36	100

Table 5.9 Cpu usages of Auditd, USTAT, Crack and find

5.4.2.5 Test Case 5: USTAT and a DSIM

The results of this test were almost identical to the results of Test Case 2. That is, cpu was barely idle. USTAT and DSIM competed for cpu time. Table 5.10 lists the cpu usages of the processes involved in this test case. Figures 5.14 and 5.15 display cpu usages of USTAT and DSIM.

Process	Cpu time	Percentage
Auditd	0:01	0
USTAT	10:20	44
DSIM	11:10	48
Idle	1:52	8
Total	23:22	100

Table 5.10 Cpu usages of Auditd, USTAT and a DSIM

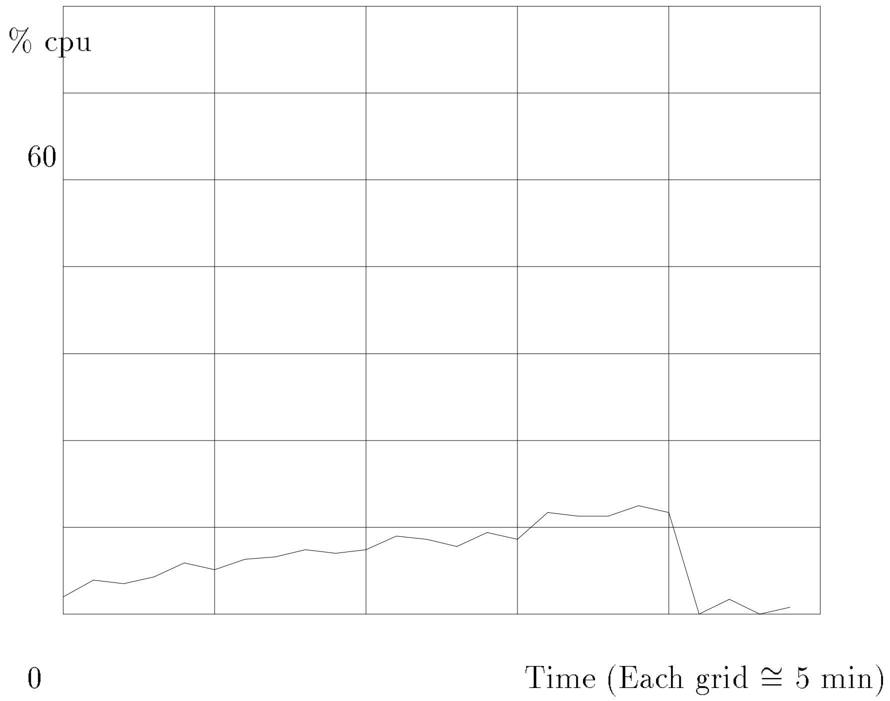


Figure 5.10 Auditd cpu usage in running USTAT, Crack and find

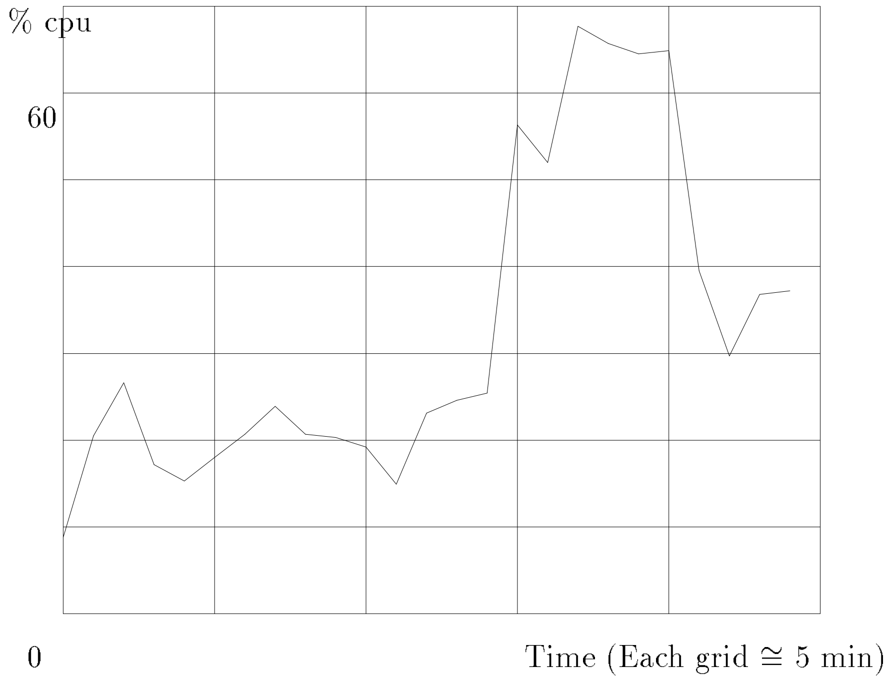


Figure 5.11 Crack cpu usage in running USTAT, Crack and find

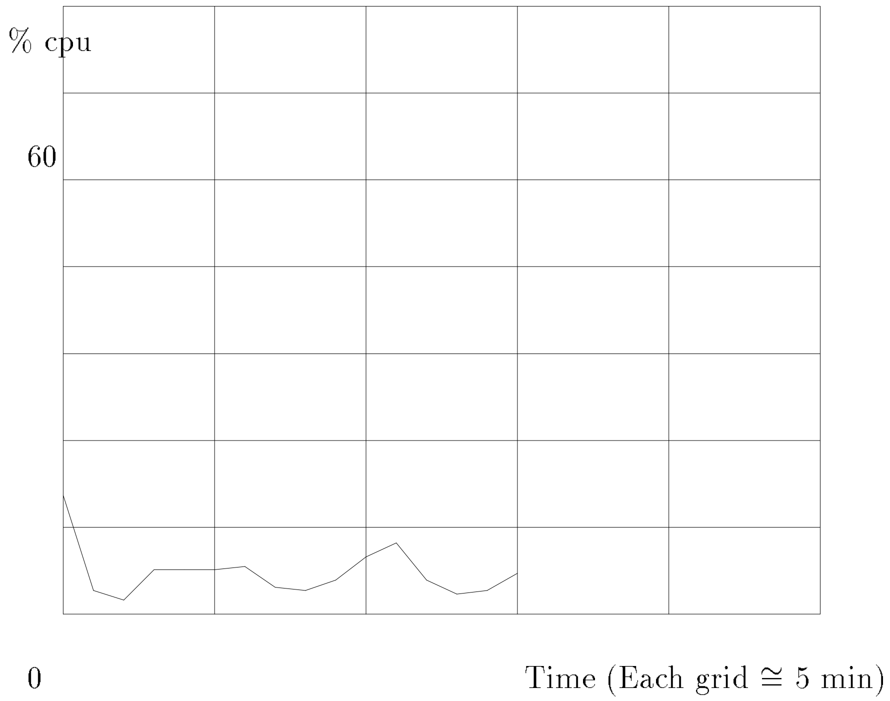


Figure 5.12 Find cpu usage in running Ustat, Crack and find

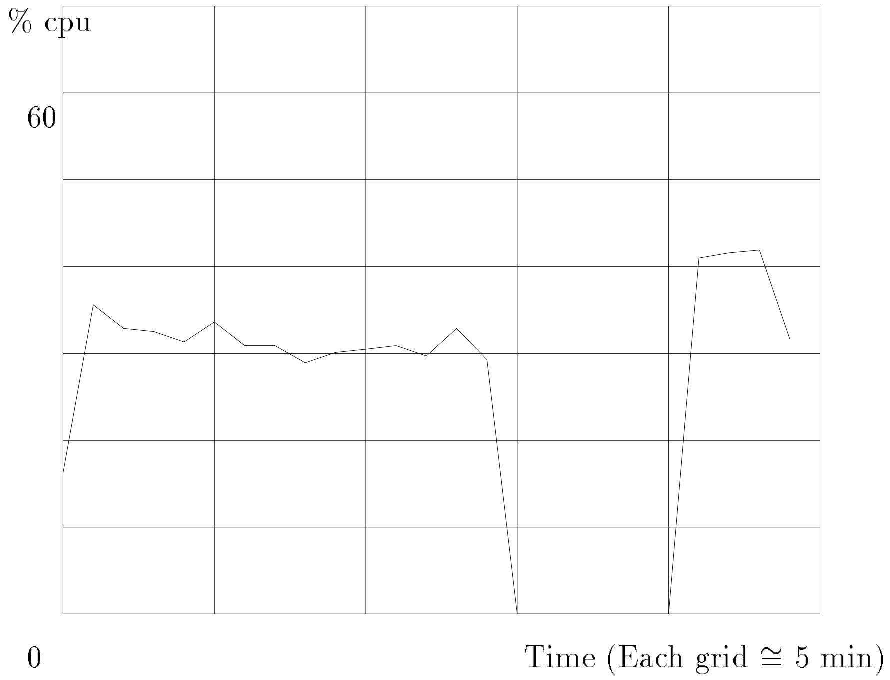


Figure 5.13 Ustat cpu usage in running Ustat, Crack and find

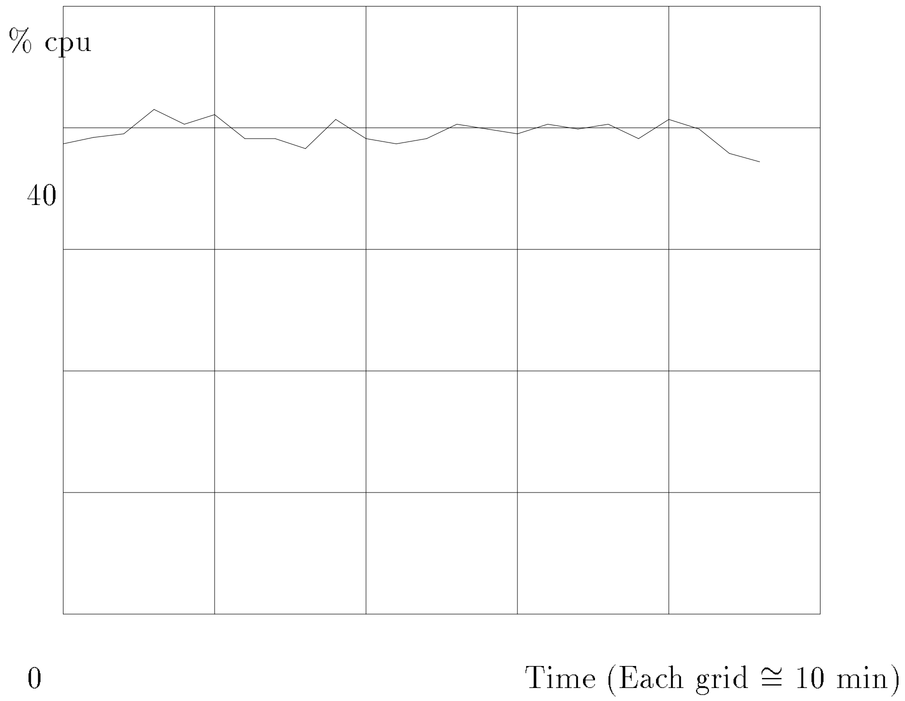


Figure 5.14 DSIM cpu usage in running Ustat and DSIM

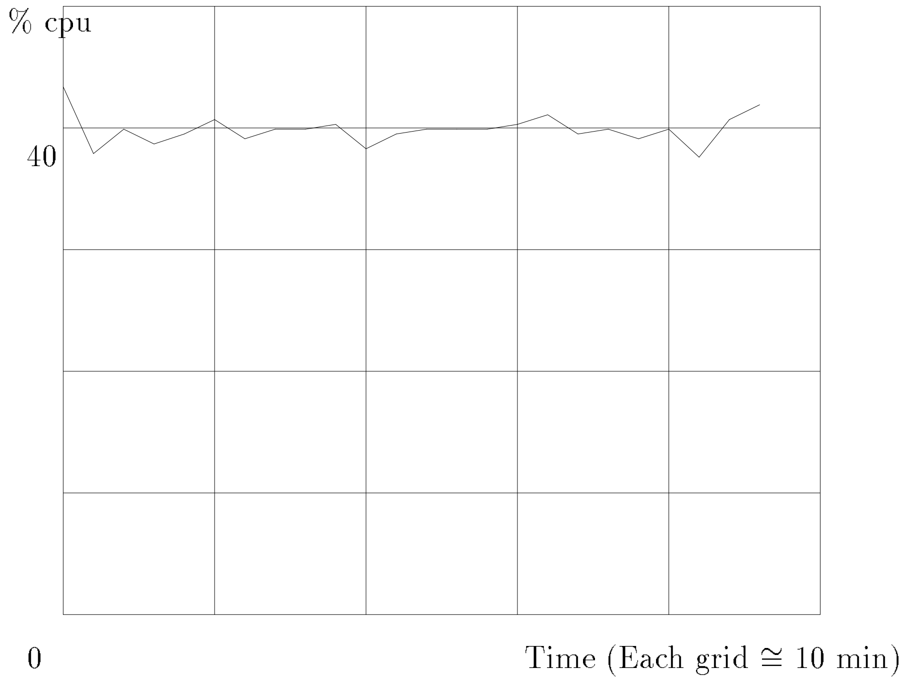


Figure 5.15 Ustat cpu usage in running Ustat and DSIM

5.4.2.6 Test Case 6: USTAT, DSIM and find

The results of this test were almost identical to the results of Test Case 4. That is, Ustat hung after 18 minutes of processing, since find was keeping the audit daemon busy. DSIM was utilizing the extra cpu cycles. Table 5.11 lists the cpu usages of the processes involved in this test case.

Process	Cpu time	Percentage
Auditd	2:56	8
USTAT	10:32	27
DSIM	18:44	49
find	1:57	5
Idle	4:06	11
Total	38:15	100

Table 5.11 Cpu usages of Auditd, USTAT, DSIM and find

Figures 5.16 through 5.19 display cpu usages of the four different processes involved in this test.

5.4.2.7 Test Case 7: USTAT and PSIM

The final test run we made was running USTAT along with another cpu intensive process. The results of this test were almost identical to the results of the test cases 2 and 5, which showed that cpu intensive processes have minor effect on the performance of USTAT. Table 5.12 lists the cpu usages of the processes involved in this test case. Figures 5.20 and 5.21 display the cpu usages of USTAT and PSIM over time.

Process	Cpu time	Percentage
Auditd	0:02	0
USTAT	12:35	45
PSIM	12:51	46
Idle	2:25	9
Total	27:51	100

Table 5.12 Cpu usages of Auditd, USTAT and PSIM

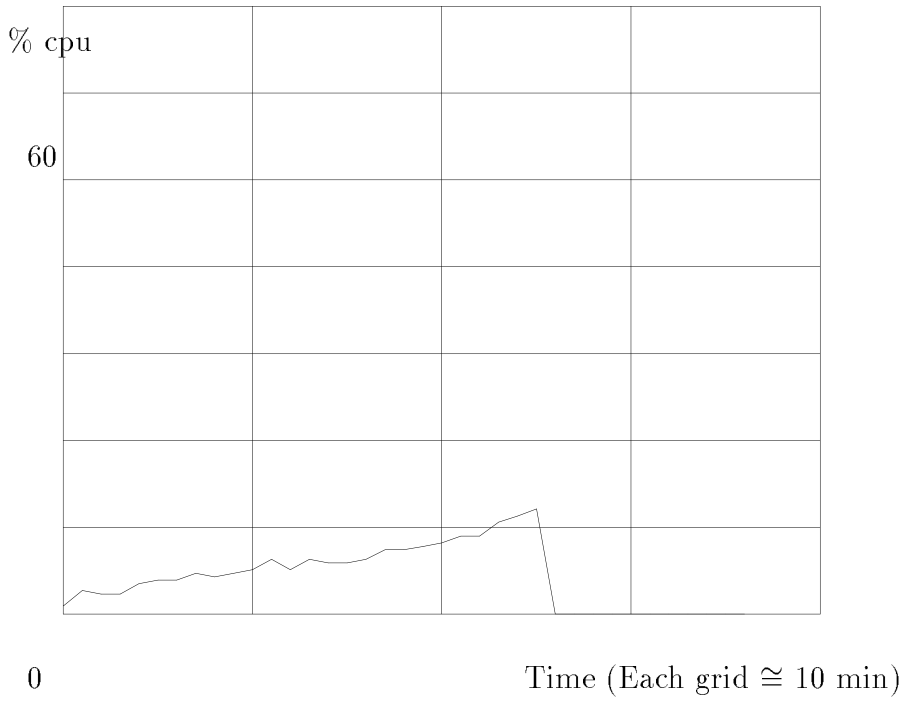


Figure 5.13 Auditd cpu usage in running Ustat, DSIM and find

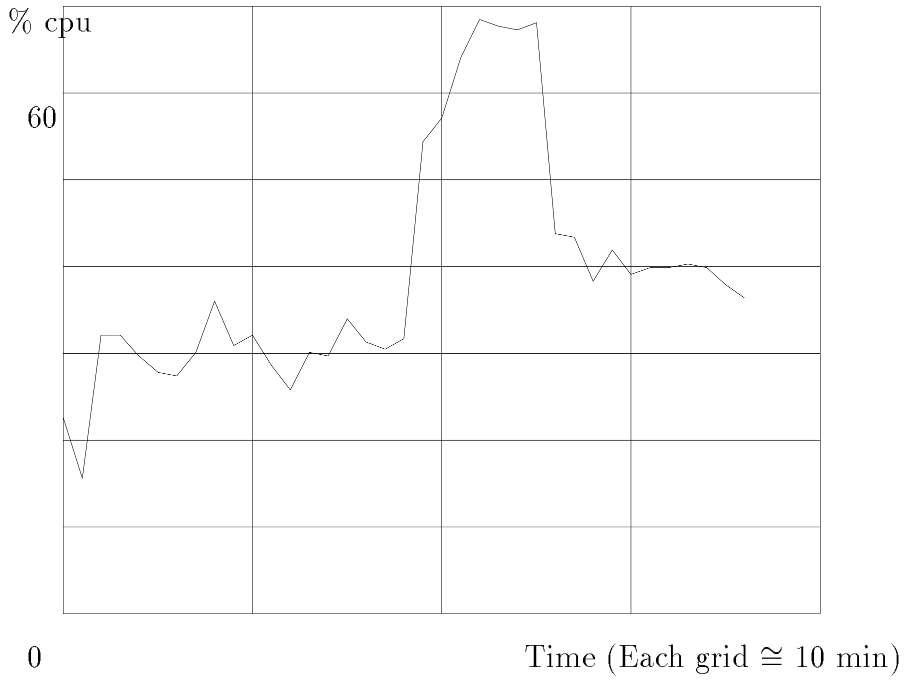


Figure 5.14 DSIM cpu usage in running Ustat, DSIM, and find

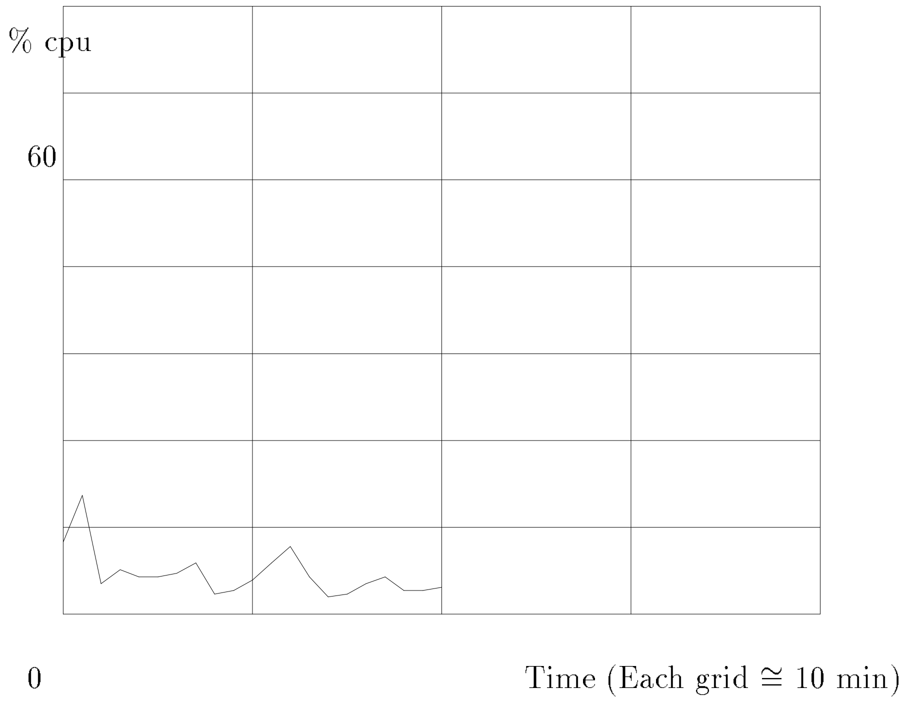


Figure 5.15 Find cpu usage in running USTAT, DSIM and find

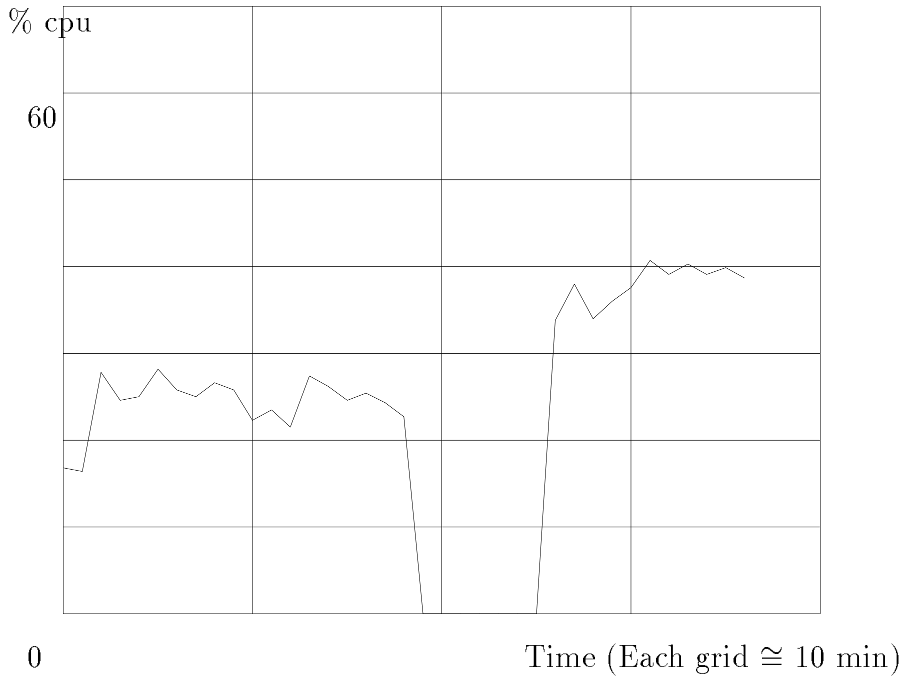


Figure 5.16 USTAT cpu usage in running USTAT, DSIM and find

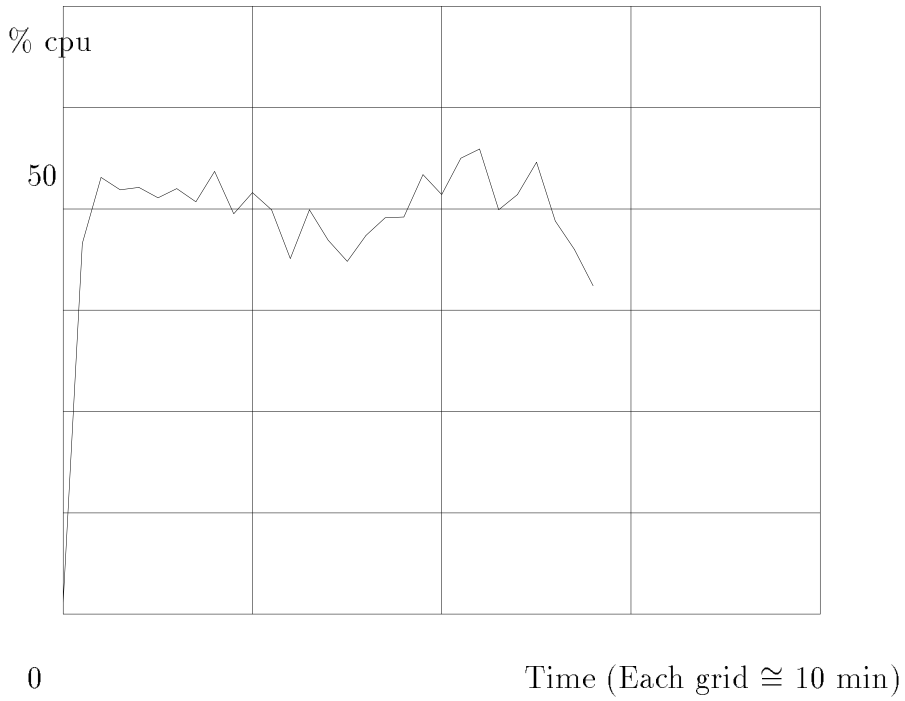


Figure 5.17 USTAT cpu usage in running USTAT and PSIM

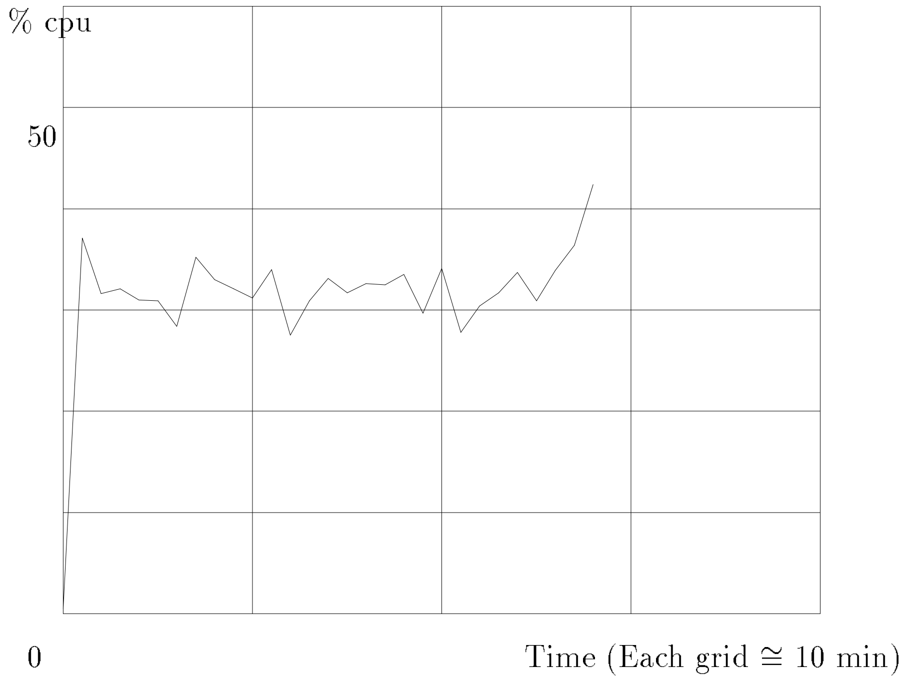


Figure 5.18 PSIM cpu usage in running USTAT and PSIM

This page is intentionally left blank

Chapter 6

Conclusion and Future Work

This page is intentionally left blank

6 Conclusion and Future Work

In this final chapter, we first give some comments about the overall results of the test runs. Then, we illustrate one more scenario that could not be added to the previous chapters. Next, we give some topics to work on in the future to improve the efficiency of USTAT and broaden its scope. Lastly, we summarize our final thoughts.

6.1 Remarks About the Tests

The tests that were presented in the previous chapter were meant to be preliminary tests rather than exhaustive. In fact, the actual performance of USTAT and the audit daemon will be more apparent after running them on different target systems with different configurations. The massive amount of data that was collected by the audit daemon kept us from performing extensive tests spanning multiple machines of a network. We could not afford the slowdown of the computers, which was mainly caused by the extensive disk I/O. So, instead we chose to limit our tests to a single host machine that was running the audit daemon, USTAT, and some other user processes. The disk that was used as an audit data repository was not dedicated to the audit daemon. It was shared by other users, but performing the tests during the summer, when there were not many users accessing the system, prevented the denial of service possibilities.

So far, the limiting factor seems to be the transfer rate of the disk that is extensively used by USTAT and the audit daemon. Cpu intensive processes that run on the same machine as USTAT have little effect on the performance of USTAT. The tests showed that if the cpu intensive process is a user process (not a root process), approximately a 5 % slowdown is experienced in USTAT's processing speed (see Test Case 2 in Section 5.4 of Chapter 5).

Among the possibilities to increase the performance of USTAT are:

- Running USTAT on a dedicated machine (IDES [Lunt92] does this),
- A periodic audit filesystem switch for the audit daemon, so that USTAT will not starve while waiting for the disk I/O (see Test Case 3, in Section 5.4 of Chapter 5).
- Running USTAT and the audit daemon on a system with better perfor-

mance, (faster disk drives, etc.).

- Running USTAT as a higher priority process, if USTAT and the audit daemon need to be run on the same computer. This would speed up USTAT's processing, but there is a problem associated with this. As in test case 3 in Section 5.4 of Chapter 5, the audit daemon is already falling behind real-time when there are many events to be recorded. Increasing USTAT's priority might cause the audit daemon to fall even further behind real-time.

As indicated by test case 1 in Section 5.4 of Chapter 5, the cpu utilization of USTAT when it is idle (waiting for more audit records) is not much different from when it is processing. This is because USTAT is continuously reopening the audit data file and trying to read new data. Instead of making this process continuous, we could add a couple of seconds delay between each reopen, so that we don't experience a high slow-down in the speed of other processes.

In Section 5.3.3 of Chapter 5 we tested the functionality of USTAT with regard to cooperative attacks. The results showed that USTAT is capable of detecting these attacks, since there is no difference for USTAT between two signature actions performed by one user or by two users. This being the case, signature actions can also span multiple user sessions and still be detected by USTAT. By recording the subject of each such action, USTAT is also able to tell which parties are involved in a cooperative attack.

6.2 An Additional Scenario

There is another scenario that was not included in the previous chapters, since it was discovered during the test runs, and we did not want to change the configuration files, which would have affected the rest of the test runs. This scenario is worthwhile including here, since it exploits a very serious flaw in remote logins.

The .rhosts file in a user's home directory contains entries that list trusted hosts and users (see hosts.equiv(5) in man pages for more information).

If the entry has the form,

```
hostname username
```

then the specified user from the specified host can access the system as the owner of the .rhosts file, e.g., if one of the entries in koral/.netrc shows:

```
orion phil
```

and assuming that user koral has access to a machine called vladimir, then user phil from host orion can logon to vladimir as user koral by issuing the following command.

```
rlogin -l koral vladimir
```

The problem arises if a user's .rhosts file becomes publicly writable. Then, anybody can alter the contents of this file and add a line containing his/her username and hostname.

USTAT can easily detect if a user's .rhosts file becomes public or group writable. The state transition diagram in Figure 6.1 can be used to detect this change.

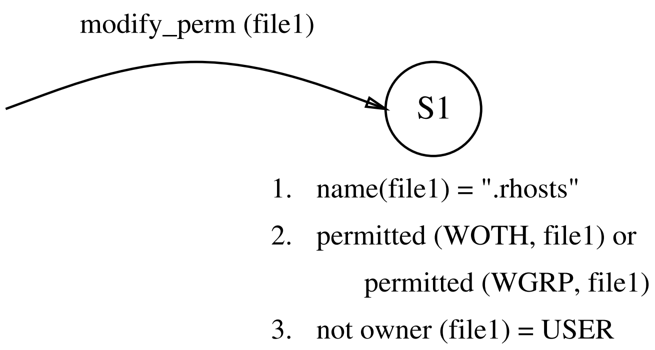


Figure 6.1 State Transition Diagram for .rhosts flaw

6.3 Future Work

6.3.1 Multiple hosts

One of our next efforts will be to run USTAT on the audit data collected by several hosts. The following lists various issues to be considered when running

USTAT for the audit data collected by multiple hosts.

- When C2-BSM is installed on multiple hosts, each host generates its own audit trail. USTAT's preprocessor is designed to operate on a single, time-ordered audit trail. If there are multiple audit trails the preprocessor needs to be redesigned and recoded to merge multiple audit trails and to provide the inference engine with a single, time-ordered audit trail.

- Instead of recoding the preprocessor, the *auditreduce* command installed with C2-BSM can be used to merge the audit trails. auditreduce takes one or more filenames (audit trails) as input and produces a single time-ordered audit trail. This output can be used as a single trail input to USTAT's preprocessor. (See [C2-b91] or the auditreduce(8) man pages for more information about auditreduce).

- There exist some real-time related problems with both of the options above. There is almost always a delay involved between the time an event has occurred and the time the event is recorded. The merging process needs to know the maximum possible amount of delay. Otherwise, the events in the resulting audit trail might not be time-ordered. Figure 6.2 illustrates this situation. Two hosts are recording their events on two different audit trails. Event_1 at Host B takes place at time t_1, and it is recorded at time t_4. Event_2 at Host A takes place at time t_2, and it is recorded at time t_3. If the merging process adds Event_2 to the merged audit trail before adding Event_1, the events will be out of order in the resulting audit trail. Therefore, the merging process must also take into account the maximum possible delay in recording. Let d be the maximum possible delay. Then, the merging process at real time t_3 should wait at least d time units before adding Event 2 to the merged output. If d time units have passed after t_3 and no record from Host A has been encountered then the merging process can add Event_2 to the merged audit trail. Otherwise, if the merging process encounters an event from Host A then it can compare the time stamps on Event_1 and Event_2, and perform the merge accordingly.

  The clock synchronization of multiple hosts must be taken into consideration even when merging data in batch-mode.

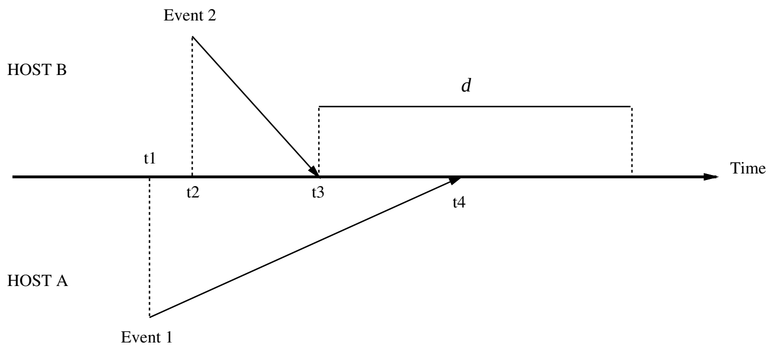


Figure 6.2 Delay problems in merging

- All hosts to be audited must mount their filesystems at the same mount-points to assure consistency in filenames. However, there will be some files that will have identical names, but in fact correspond to different physical files, such as the files in /usr/bin. In our analysis, we used the full pathnames for the identity of files. When performing the analysis for multiple hosts, device and inode numbers of files also need to be considered.
- USTAT can be run on one of the hosts that is audited, or it can be run on a dedicated machine to improve performance.

6.3.2 Interactive interface

Currently, once the program is started, there is no other input to USTAT except the audit data. Another module can be added to USTAT to provide more interaction between the SSO and USTAT. One desirable feature of this interface would be the capability of adding and removing rules to the rule-base while USTAT is running. The addition or removal request could be taken by the interface and processed after the processing of the current audit record is finished.

Another desirable feature would be the addition of new state assertions. All state assertions are system specific. It is difficult to have a complete list of state assertions that cover all of the state transition diagrams of current penetration scenarios for the target system and that will cover the state transition diagrams of future scenarios. We might need a new state assertion that cannot be expressed with the ones we already have. The solution to this is not trivial and it was not addressed in [Porr91]. Currently, all the state assertions of USTAT are coded-in. The rule-base can only use these pre-determined state assertions. To add modularity, the SSO should be allowed to define new state assertions by writing small programs once the interface is defined. This feature is expected to be added to later versions of USTAT.

6.3.3 Permutable state transitions

STAT was originally designed to support permutable state transitions. In this thesis, each scenario corresponds to a single, non-permutable rule sequence. Permutable state transitions were not necessary for the scenarios that we encountered during the development of USTAT. Therefore, we chose not to implement it.

To incorporate permutable state transitions into USTAT, some source code changes will be necessary. First, the data structures for the rule-base need to be modified to include a rule-dependence field. In addition, the syntax of the state description table and signature action table need to be modified to allow the SSO to indicate the rule-dependence of the steps of a scenario. Finally, the inference engine routines that record the expected signature actions need to be modified to include signature actions of alternative paths in a permutable state transition diagram.

6.3.4 Trusted users

USTAT could also incorporate a configuration file consisting of the user id's of the trusted users. The actions of these users would still be audited and passed by the preprocessor to the inference engine. However, the inference engine would use these actions only to report to the fact-base updater and after that they would be ignored. This would increase the processing speed of USTAT by skipping the actions of trusted users.

6.3.5 USTAT's security

The security of USTAT and the audit collection mechanism is not addressed in this thesis. The following issues come to mind regarding the security of

USTAT. Is there any way for an attacker to become a clandestine user? Can the attacker disable the audit mechanism? For example, in all the test cases that involved processes that were being audited heavily (such as find), we encountered a considerable amount of delay in the audit daemon's recording time. How far back in time can this delay go? If there is a delay, that means the audit daemon is keeping the events to be recorded in memory. Is it possible that the audit daemon skips some events or crashes because of running out of memory? It might also be desirable to have dedicated audit disks so that the attacker does not fill the disk space or does not keep the audit disk busy.

6.3.6 Maintenance of Inference Engine Data Structures

Several state transition diagrams of USTAT involve execute actions. Whenever an execute event on the system satisfies the next state assertion in a scenario, the next signature action, if any, is added to the list of expected signature actions. That means, for each execute event on the system it may be necessary to demand additional memory to record the changes to the inference engine data structures. This additional memory space can be freed only when the process that corresponds to the execute event is exited. Unfortunately, not all the process exits are recorded by the audit mechanism. For example, if a process terminates because of a Ctrl-C or kill, there is no exit event recorded by the audit mechanism. This results in continuously increasing the memory usage of USTAT.

To avoid a memory overflow, a ps-like program can be run to see which processes are still alive and according to the results a periodic clean-up can be applied to the inference engine data structures to remove instantiations that are impossible to complete. There are some restrictions on this clean-up process. First, USTAT needs to run real-time. Secondly, it needs to run on the target machine, where auditd is running. The periodic clean-up solution would be highly impractical when multiple hosts are audited.

6.4 Final Comments

It is worthwhile repeating the following remark, which was presented in [Porr91].

> *"The importance of choosing the correct target environment in which to implement STAT is key. In low security-critical computing*

environments, the added security provided by STAT may not outweigh the potential performance degradation. On the other hand, in high security-critical environments performance will not be the key issue."

One of the most important results that is obtained by the implementation of USTAT is the confirmation of the above statement. The tests showed that the performance degradation caused by the audit daemon and USTAT were inevitable. However, there is always a trade-off between system performance and security. The amount of each will depend on the target environment.

During the implementation of USTAT many problems were encountered. Many changes were made to the original design of STAT mostly to overcome these problems. All these problems and modifications are discussed in detail in Chapter 4, "The Components of USTAT." The following are some of the key points that are worth repeating.

- The development of the preprocessor was not simply limited to the meaning of preprocessing. This development included the consideration of hundreds of event types and the selection of only a fraction of those that were mapped onto the 10 different event types of USTAT.

- The original design of STAT suggested eight different filesets. Many changes have been made to the contents of these filesets. Also, due to the efficiency constraints, most of the filesets in the fact-base remained at the conceptual level. We realized that it was easier and more efficient to keep the properties of a fileset in working memory, rather than keeping each member of a fileset. The exceptions to these were filesets 3 and 4, the creation of which required manual processing and each member had to be kept in memory. As a result, the maintenance of the fact-base was limited to the hardlink information. The process of obtaining the hardlinks to a given file was found to be impossible unless the filesystem was completely scanned.

- The state transition diagrams were central to this thesis. More care needs to be taken in constructing multi-step state transition diagrams, since they are subject to more constraints in signature actions, e.g., an instantiation of a state transition diagram that includes the execute signature action can be active as long as the process that triggered the signature action is active.

- The discovery of the inference engine table was of crucial importance for the sound implementation of the inference engine. By working on that idea and developing other structures around it, the resulting implementation of the inference engine was quite efficient.

- Because we did not want to overload the purpose of the decision engine, the decision engine was the easiest USTAT module to implement. In the future, it can be expanded to have a more active role in the program, such as taking preemptive action once a compromise is about to be achieved.

The experience with UNIX internals gained throughout the development of this thesis were of substantial value. Part of this experience is shared with the reader by presenting many issues of UNIX internals in the appendix.

Finally, by implementing USTAT the conceptual soundness and functional capabilities of STAT as a model have been validated. The author hopes that more research in the area of intrusion detection will follow.

This page is intentionally left blank

References

[C2-b91] Sun Microsystems, Inc., *SunOS Release 4.1.1. C2-BSM Patch, Revision A*, 2550 Garcia Ave, Mountain View, CA 94043, Dec. 15, 1991.

[Chen90] K. Chen, S.C. Lu and H.S. Teng, "Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns," presented at the Fifth Intrusion Detection Workshop, SRI International, Menlo Park, CA, May 1990.

[Disc85] Anthony V. Discolo, "4.2B SD UNIX Security," Computer Science Dept., University of California, Santa Barbara, Apr. 1985.

[Fors84] Richard Forsyth, *Expert Systems: Principles and Case Studies*, New York, NY, Methuen, 1984.

[Harm88] P. Harmon, R. Maus, W. Morrissey, *Expert Systems Tools and Applications*, 605 Third Avenue, New York, N.Y., John Wiley & Sons, Inc., 1988.

[Hubb90] B. Hubbard, T. Haley, N. McAuliffe, L. Schaefer, N. Kelem, D. Wolcott, R. Feiertag and M. Schaeffer, "Computer System Intrusion Detection," Trusted Information Systems, Inc., RADC-TR-90-413, Final Technical Report, Dec. 1990.

[Ilgu91] K. Ilgun, "Implementation of USTAT Audit Data Preprocessor," Term Paper, Computer Science Dept., University of California, Santa Barbara, Dec. 1991.

[Lunt92] Teresa F. Lunt et al. "A Real-time Intrusion Detection Expert System (IDES)," SRI Technical Report, February 28, 1992.

[Mart88] James Martin, Steven Oxman. *Building Expert Systems: A Tutorial*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

[Porr91] P. A. Porras. "STAT: A State Transition Analysis Tool for Intrusion Detection,", Master's Thesis, Computer Science Dept., University of California, Santa Barbara, July 1992.

[Scha91] S. I. Schaen, B. W. McKenney "Research, Standards, and Policy Directions for Network Auditing," presented at the Fifth Intrusion Detection Workshop, SRI International, Menlo Park, California, May 10-11, 1990.

[Sebr88] M. M. Sebring, E. Shellhouse, M.E. Hanna, R.A. Whitehurst, "Expert System in Intrusion Detection: A Case Study," Proceedings of the 11th National Computer Security Conference, Baltimore, MD, Oct. 1988.

[Tcse85] National Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD, DoD 5200.28-STD, Dec. 1985.

[Vacc89] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, pp. 280-289, May 1989.

Appendix

This page is intentionally left blank

Remarks About UNIX Internals

In this section we point out many aspects of UNIX internals. These were encountered while working on this project and most of them had direct effects on USTAT's implementation. These features are especially effective in SunOS 4.1.1, but they might differ on other versions.

1 User Id's

- *newgrp* command enables the user to obtain a new effective group id, provided that the user belongs to more than one group. It starts up a new shell with the new effective group id.
- When a setuid program that is owned by somebody else is copied, the setuid bit remains set, although the ownership changes. When the group ownership of a program is changed by using *chgrp*, however, the setuid bit gets turned off. In general, setgid programs are more secure than setuid programs, because group permissions are usually, but not necessarily a subset of user permissions.
- Why shouldn't we use setuid shell scripts?

  There are a variety of reasons why we should not use setuid shell scripts, mostly involving bugs in the Unix kernel. The ones that are still effective on SunOS 4.1.1 are the following.

  1. If the script begins "#!/bin/sh" and a symbolic or hard link can be made to it with a name starting with "-", an interactive shell can be obtained by executing the link. (See Section 4.2.2 of Chapter 4)
  2. By including "." as the first directory in path and creating a malicious program (or even script) that is named after any command name that is included in the original setuid shell script, one can do things that he/she is normally not authorized to do. (See Section 6.2 of Chapter 6)
  3. Many kernels suffer from a race condition, which can allow one to exchange the shellscript for another executable of his/her choice between the times that the newly exec()ed process goes setuid, and when the command interpreter gets started. If one is persistent enough, in theory he/she could get the kernel to run any program he/she wants.

If one really must write scripts to be setuid, he/she must use a scripting language like Perl, which has a safe setuid facility.

2 Links

- There are two types of link structures: hardlinks and symbolic links. The latter are simply called symlinks.

- When a hardlink is created a new name is added to the directory structure, creating a new directory entry for an existing inode. The file system contains a path name for each link the file has, and processes can access the file by any of the pathnames. The kernel does not know which name was the original file name, so no file name is treated specially. Creation of a hardlink increases the inode count of the inode by 1. Deletion of a file decreases the inode count by 1. If the inode count becomes 0, the inode is deleted. The inode count for any file can be examined by using ls command with the -i option.

- Symlinks serve the same purpose, but their physical structure is different. They add the capability of linking files across different filesystems. A new type indicator, l, specifies a symbolic link file; the data of the file is the path name of the file to which it is linked. ls -l command can be used to display the file permissions. An l in the first bit of the permissions indicates a symbolic link, and the string that comes after the symlink filename and starts with an arrow indicates the target file or directory name, e.g.,

```
lrwxrwxrwx  1 koral 17 Feb 13 14:18 ps -> /usr/local/ps/lib/
```

- Hardlinks are created using ln command, symbolic links are created using the same command with the -s option.

- Symbolic links create a new inode, hardlinks do not.

- A symbolic link can be linked to another symbolic link, which can be linked to another one, and so on. The target for a symbolic link is always the last file in such a linked list. There cannot be more than 20 consecutive links. Circular links also cause an error condition.

```
% ln -s foo bar
% ln -s bar foo
```

Execution of this command sequence will create a circular link. If one tries to access any of these files, an error will be returned indicating *too many levels of symbolic links*.

- For hardlinks the target is the inode since there may be any number of pathnames that are linked to that inode.
- A symbolic link can be created with a non-existing target filename, but a hardlink cannot.
- The stat(2v) system call with a symbolic link as its argument gives the target file's permissions. lstat(2v) system call with a symbolic link as its argument gives the symbolic link's permissions, which are always lrwxrwxrwx.
- Hardlinks to symlinks:

  ```
  % ln -s target foo
  % ln foo bar
  ```

  After executing these commands, bar becomes a hardlink to target, so they will refer to the same inode.
- Yet another symbolic link:

  ```
  % touch foo
  % ln -s foo bar
  % mv bar ..
  ```

  In the last step bar is moved to another directory. Since bar was linked to foo without using the full pathname of foo, bar has lost its link to foo. Similarly, even if full pathname is used for linking, if the target is moved to another directory or renamed, the link to it will get lost. Therefore, it is possible to have many dangling pointers in the filesystem if care is not taken in using symbolic links.
- The link count of a setuid/setgid program should be checked before removing it. If it is greater than 1, the file should be removed after changing its mode to 000, so that the remaining link will be harmless.

3 Permission Bits

Permission Bits on Files

- When a file is copied with cp, the permissions of the source file are duplicated and then umask'ed [9] if the destination file doesn't already exist. If the destination file already exists, its original permissions are retained. In both cases, setuid and setgid bits are preserved, ownership and group ownership are changed to the user making the copy. The following example illustrates the case where the destination file did not exist prior to cp command.

```
~ > ls -lg /bin/ypchsh
-rwsr-xr-x  5 root  staff 32768 Oct 11  1990 /bin/ypchsh*
~ > cp /bin/ypchsh .
~ > ls -lg ypchsh
-rws------  1 koral grad  32768 Mar 16 17:04 ypchsh*
```

- Setuid bit gets reset when a file is overwritten.

Permission Bits on Directories

- The deletion of a file is considered to be a write access on the directory, not on the file. So, user A can delete a file owned by user B as long as the file is located in a directory to which user A has write access. The exception to this is the sticky bit *t*. If this bit is set for a directory then users can delete only the files that are owned by them. The sticky bit is useful for shared directories, such as /tmp or /var/spool/mail. Everybody has write access to the mail directory, so that they can manipulate their mail files, but nobody can delete somebody else's mail file. The sticky bit can be set by executing the following command.

```
chmod +t filename
```

- Read permission to a directory is equivalent to the ability to read only the names of the files that are located in that directory.

---

[9] Permission bits are created depending on the value of the umask environment variable

- Execute permission to a directory is equivalent to the ability to change the current directory to the target directory.

- Other attributes of the files, their permission bits, ownership, etc. in a directory can be obtained only if the execute and read permissions for the directory are set at the same time.

- The following is a group of examples for a user trying to access somebody else's files and/or directories. The interesting part is if user A gives write permission for one of his/her directories to user B, user B can create a directory and a file under that directory and user A won't be able to delete it. (User A = koral, User B = audit).

  - Directory temp, owned by koral, has no permissions for others. User audit is executing the command:

    ```
    drwx------  2 koral      512 Mar 10 15:01 temp/

         % ls temp
         temp unreadable
    ```

  - Directory temp, owned by koral, has read permission for others. User audit is executing the commands:

    ```
    drwxr--r--  2 koral      512 Mar 10 15:01 temp/

         % cd temp
         temp: No such file or directory
         % ls temp
         ls: temp/mbox: Permission denied
         ls: temp/mm: Permission denied
         % ls -al temp/mbox
         ls: temp/mbox: Permission denied
    ```

  - temp has read and execute permissions for others. User audit is executing the commands:

    ```
    drwxr-xr-x  2 koral      512 Mar 10 15:01 temp/

         % ls -l temp
         total 96
         -rw-r--r--  1 koral   48521 Mar 10 15:01 mbox
    ```

```
-rw-------  1 koral   48521 Mar 10 15:01 mm
% cd temp
% ls
mbox    mm
```

– temp has execute permission for others. User audit is executing the commands:

```
drwx--x--x  2 koral      512 Mar 10 15:01 temp/
```

```
% cd temp
% ls
. unreadable
% ls -al mbox
-rw----r--  1 koral   32773 Mar 16 16:55 mbox
% ls -al mm
-rw-------  1 koral   32773 Mar 16 16:57 mm
% cat mm
cat: mm: Permission denied
% head mbox
From @uccvma.ucop.edu:TEL@USCVM.BITNET Fri Nov
...
```

– temp has read, write, and execute permission for others. User audit is executing the commands:

```
drwxrwxrwx  2 koral      512 Mar 10 15:01 temp/
```

```
% cd temp
% mkdir mine
% ls -al
total 100
drwxrwxrwx  3 koral     512 Mar 10 15:08 ./
drwxr-xr-x 11 koral    1536 Mar 10 15:01 ../
-rw-r--r--  1 koral   48521 Mar 10 15:01 mbox
drwx------  2 audit     512 Mar 10 15:08 mine/
-rw-------  1 koral   48521 Mar 10 15:01 mm
% cp mbox mine
% ls -al mbox
-rw-r--r--  1 koral   48521 Mar 10 15:01 mbox
% ls -al mine
```

```
total 50
drwx------  2 audit     512 Mar 10 15:09 ./
drwxrwxrwx  3 koral     512 Mar 10 15:09 ../
-rw-------  1 audit   48521 Mar 10 15:09 mbox
```

– This time, user koral executes the following.

```
~/temp > ls -al
total 100
drwxrwxrwx  3 koral     512 Mar 10 15:09 ./
drwxr-xr-x 11 koral    1536 Mar 10 15:01 ../
-rw-r--r--  1 koral   48521 Mar 10 15:01 mbox
drwx------  2 audit     512 Mar 10 15:09 mine/
-rw-------  1 koral   48521 Mar 10 15:01 mm
~/temp > rmdir mine
rmdir: mine: Directory not empty
~/temp > rm mine/*
mine/: Permission denied
```

4 Ownerships

- Prior to Release 4.0, files took the group of their parent directory since their creator might be a member of more than one group. With 4.1, files created on file systems not mounted with the *grpid* option obey System V semantics. That is, their group id is set to the effective group id of the creating process. grpid option uses 4.2 BSD semantics.

- The owner of a file and the superuser can change the group that owns a particular file by using chgrp command.

- Only the superuser can use chown.

- The following information is obtained from the frequently asked questions list of Alt.security newsgroup.

  Question: Is there a security problem of having /etc directory owned by bin?

  ```
  drwxr-sr-x  9 bin   staff     2048 Jun 19 13:17 etc/
  ```

Answer:

YES. If you're in an NFS environment, it is recommended that the owner of /etc be the root. The reason is that if someone cracks root on a machine that your machine "trusts" (because of /etc/hosts.equiv or for some other reason), that someone can say "su bin" and then your system will believe he should be treated as "bin" on your system, and he'll have the power to install anything he wants into the /etc/ directory.

But "root" is special: NFS, rsh, etc, treat root on some other system as "nobody" on your system, so that if someone cracks root on a system you trust, he's still only a generic normal user on your system. This'll at least slow a good cracker down a bit.

Conclusion: if you use NFS, rsh, rlogin, rcp, etc., then all system-critical files and directories (/etc, /bin, /usr/bin, etc, and everything in them) should be owned by root and writable only by root. The habit of having "bin" own things, or having a special "staff"/"wheel" group that could modify things, comes from pre-NFS days and it's no longer safe.

- Another question from Alt.security:

Question: Is there a security risk of having an ascii file owned by root world writable?

Answer: YES. This file can be used to limit system availability by using up the filesystem. If the file is on the root filesystem, crashes may occur.