EXECUTION MONITORING OF SECURITY-CRITICAL PROGRAMS IN A DISTRIBUTED SYSTEM: A SPECIFICATION-BASED APPROACH

By

Calvin Cheuk Wang Ko
B.S. (University of Hong Kong) 1990
M.S. (University of California, Davis) 1993

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Karl N. Levitt, Chair

Manfred Ruschitzka

Matthew Bishop

Committee in Charge

1996

Acknowledgements

I am deeply indebted to my advisors, Karl Levitt and Manfred Ruschitzka. Without their help and guidance, I was not able to finish this work. Special thanks to Karl for his precious time and energy, and his patience in teaching me not only how to perform research, but also how to write proposals, deal with sponsors, give presentations, and survive as a researcher in the security field. I would like to thank Manfred for the time and effort he has spent working with me and polishing the dissertation. His ideas and insightful suggestions have greatly enriched the technical content of this work; his persistent pursuit of clarity has greatly improved its presentation.

I would also like to thank the other member of my committee, Matt Bishop, for his valuable comments on this dissertation. I also thank the fellow students and researchers in the security group for their comments and feedbacks on my research. Thank Jim Hoagland, Steven Cheung, and Raymond Yip for proof-reading drafts of this dissertation.

I would like to thank my family and friends for their support and constant encouragement, especially my wife Mandy, for her patience and understanding and also for her great assistance in proof-reading early drafts of this dissertation.

Finally, I would like to thank God (my heavenly Father, my beloved Jesus, and the Holy Spirit) for His eternal love, encouragement and constant nourishment. His grace has always been sufficient for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computer systems are used increasingly for processing critical information and performing critical jobs, our society heavily depends on the secure operation of these systems. Lack of security in these systems could result in loss of money, manpower, national competitiveness, and even threaten lives.

1.1 Computer Security

Generally speaking, computer security is concerned with protecting the computing resources and the information stored in computer systems. Confidentiality, integrity, and availability are the three typical high-level requirements of a secure computer system. Confidentiality is concerned with preventing unauthorized disclosure of information and unauthorized accesses to information. Integrity is concerned with maintaining consistency and correctness of data in the system. Availability is concerned with preventing exhaustion of resources and denial of services to authorized users.

Most computer systems employ access controls [33, 34, 44] as the first line of defense to protect resources and information in the systems. Access controls ensure that all direct accesses by subjects (e.g., users or processes) to objects (e.g., files and records) are authorized. Access controls regulate direct accesses to objects, but not what subjects might do with information contained in these objects [14]. In order to achieve confidentiality, information flow controls [5, 6] are needed to regulate dissemination of information. Besides access controls and flow controls, authentication is also needed in a computer system to assure the identity of users. Also, it is important to assure that the security mechanisms work properly.

Both formal [40, 43] and informal [22, 35] techniques for assuring the security of a system have been investigated.

Despite the advance in computer security and software engineering, current computer systems are still vulnerable to attacks. Many existing computer systems contain vulnerabilities [49, 47, 18] that enable attackers to penetrate the systems. The proliferation of computer networking further exacerbates the problem, as computer systems are not just accessible by insiders who are presumably trusted, but by virtually anyone on the network. An insecure computer system connected to the Internet is opened to attackers around the world. As redesign is not a feasible and cost-effective solution to cope with the current situation, retrofit approaches that require only minor changes to systems are needed to enhance the security of current computer systems.

Intrusion detection has been proposed as an approach to cope with current security problems. The goal of intrusion detection is to detect activities in a system that violate the security policy or compromise a system's security. It is an after-the-fact approach to security as oppose to the preventive approaches used traditionally. As intrusion detection requires as its data source only a log of the activities of the system, it can be applied to most existing systems. In addition, it can enhance the security of even a very secure system, for example, to detect insiders who accidentally or maliciously exceed their privileges. In the next three sections, we describe current approaches to intrusion detection, including configuration analysis, anomaly detection, and misuse detection. We evaluate the approaches and then describe a novel approach to intrusion detection, the specification-based approach.

1.2 Configuration Analysis

The goal of configuration analysis (also known as static analysis) is to check whether a system has been or could be compromised. Static analysis involves examination of the current system configuration such as the content of system files and system tables. The term *static* refers to the static characteristics of the system, not the system activity. The rationale behind configuration analysis is twofold: a security compromise may leave behind a residue that can be detected by checking the state of the system, and system administrators and users often make mistakes in setting up a system, enabling attackers to compromise a system's security. As configuration analysis can reveal potential problems before they occur, it is also a preventive approach to security.

A widely used configuration checking tool is the Computer Oracle Password and Security System (COPS) [19]. COPS enforces a predefined and fixed configuration policy represented in a set of shell scripts that check for common misconfigurations in a Unix system. For example, it checks the permission mode of security-relevant files and directories such as the password file, `/etc/passwd`, the group file, `/etc/group`, and the `/etc` directory to ascertain that only the superuser can modify these files.

COPS uses a set of *rules-of-thumb* to determine whether a system's configuration is correct. An example of a rule-of-thumb is that no *setuid root* [1] files should be writable by all users. Other rules deal with search paths, system initialization files, and programming conventions. This approach simplifies the task of securing a system's configuration, but excludes alternative configurations set up by individual administrators which are useful.

The U-Kuang system [4] is a rule-based expert system for checking the security of a Unix file system's configuration. Instead of a predefined configuration policy, U-Kuang enforces a high-level, user-specified policy which defines the set of privileges available to each user. It enumerates the set of operations directly or indirectly accessible to each user.

The rule base describes the behavior of the security kernel, privileged programs, and the ways an attacker can extend his privileges. One example rule is "*if a user can write to a directory, then the user can replace any file inside the directory*". In Unix, a user can delete a file if he has write permission on the directory containing the file, and the rule follows because the user can first delete the file that he wants to replace, and subsequently recreate the file with the desired content. Another example rule is "*a user who can replace the password file is able to acquire superuser privileges*". The U-Kuang system exhaustively searches for all possible plans of actions that enable a user to acquire privileges inconsistent with the specified policy. One limitation of U-Kuang is that the policy can specify only user and group privileges that can be acquired by a user.

The Miro' security constraint file checking system [23] checks a file system against a set of user-specified security constraints that defines the legal configurations of the system.

[1] A file is *setuid root* if it is owned by root and has the setuid bit on. When a setuid root program is executed, the resulting process possesses superuser privileges instead of the privileges of the invoker.

Security constraints are specified graphically as pictures in the Miro' *constraint* language. A system administrator uses a graphical editor to create and modify constraint pictures, which are subsequently fed to the *constraint checker* to verify the current file system configuration. The use of visual notations enhances the understandability and readability of the security constraints, which are otherwise similar to the rules-of-thumb in COPS.

Tripwire [30] is a file integrity tool that aids system administrators and users in monitoring a designated set of files for any changes. Attackers tend to leave behind backdoors (e.g., in the login program) or Trojan horses after they penetrate a system that enable them to re-enter the system even after the exploited vulnerability is removed. Moreover, an attacker might install a sniffer program and replace utility programs (e.g., *ls*, *ps*) that can reveal information about the sniffer program [10]. This technique makes it difficult to discover the compromise. In Tripwire, each file to be protected is provided with a precomputed cryptographic checksum which is theoretically unbreakable without the key. The file is then periodically checked for any modifications by comparing the current checksum with the precomputed checksum.

1.2.1 Evaluation of Configuration Analysis

A significant percentage of computer break-ins is caused by improper system configurations. Even in operating systems with strong protection mechanisms, their incorrect use leads to security flaws. System administrators and users make mistakes because of carelessness and misunderstanding the complexity of protection systems. In most current operating systems (e.g., Unix), a user does not set up an access policy directly, but by setting the permission bits of files or by other low-level mechanisms. Configuration analysis aids in ascertaining that the configuration is consistent with a standard security policy.

Configuration analysis provides a reasonable solution to many security problems. However, it is not a complete solution. Even if a system's configuration is correct, it may still contain potential vulnerabilities such as errors in trusted programs that facilitate penetrations. Also, insiders could misuse their privileges in a correctly configured system. To further enhance a system's security, activities in a system should be monitored to detect potential compromises.

1.3 Anomaly Detection

Anomaly detection, first proposed by Anderson [2] to detect intrusions in computer systems, involves monitoring activities in a system using audit trails. Later, Denning [15, 16] presented a model of an anomaly-based intrusion detection system. The basic premise behind anomaly detection is that exploitation of a system's vulnerabilities involves abnormal use of the system; security violations can, therefore, be detected from abnormal patterns of system usage. This technique can potentially reveal a masquerader or a legitimate user abusing his/her privileges [2]. The basic idea is to establish normal behavior patterns of subjects (e.g., individual users, groups of users, hosts, etc.) by observing audit trails over a duration of time. An audit trail that deviates from the subject's established behavior pattern is regarded as an indication of an intrusion.

Current anomaly detection systems use statistical or rule-based profiles. A profile is a description of a subject's normal behavior in terms of intrusion-detection measures or features such as login time, login location, CPU time used, and file accesses. A statistical profile consists of statistics such as frequencies, means, and variances of the features. In a statistical profile-based anomaly detector, statistical methods are used to determine whether an audit trail deviates from the norm significantly. A rule-based profile is a set of rules that specify the legal values of features which may depend on the values of other features. Rules can, in principle, be generated automatically from the past behavior. However, selecting features that best delineate normal behavior from intrusive behavior is a difficult problem [17, 20].

1.3.1 Statistical Profile-based Approaches

SRI International's Next Generation Real-Time Intrusion-Detection Expert System (NIDES, formerly IDES) [37, 38, 27] is a statistical profile-based anomaly detection systems. The anomaly detector observes audit data and adaptively draws conclusions about the normal behavior of subjects, which can be individual users, groups, remote hosts and the overall system. It uses multivariate methods for profiling normal behavior and identifying deviations. It maintains a statistical knowledge base of subjects and audited activity is described by a vector of intrusion-detection variables, corresponding to the features recorded in the profiles.

As an audit record arrives, the relevant profiles are retrieved from the knowledge base and compared with the vector of intrusion-detection variables. If the point in N-space defined by the vector of intrusion-detection variables is sufficiently far from the point defined by the values stored in the profiles, the record is considered anomalous. Also, as subjects alter their behavior, their corresponding profiles change.

The NIDES statistical component is also used to monitor activities of application programs in the SAFEGUARD project [1]. Its main goal is to detect unauthorized use of applications or application classes on restricted data, but the methodology is useful in the detection of Trojan horses and masquerading applications too.

1.3.2 Rule-based Approaches

Wisdom and Sense (W & S) [53] is a rule-based anomaly detection system. It uses a tree-structured rule forest to describe historical behavior patterns that are statistically significant. The rules specify normal feature values conditioned on the values of other features. Rules can overlap in specificity due to incomplete information in the history. Rule pruning occurs if there are too many normal values for a feature, too few historical values, the rule is too deep, or a rule is conditioned on an anomalous value. The rule base tends to be very large (10^4 to 10^6 rules). An expert can add to or modify the rule base using an English-like syntax.

The Time-based Inductive Machine (TIM) [52] is also a rule-based anomaly detection system. TIM uses an inductive method to generate rules, which are modified dynamically during the learning phase. Rules remain in the rule-base only if they are highly predictive or confirmed by many observations. Prediction is calculated using an entropy model. The user must specify the behavior TIM is to predict. An example of a rule generated by TIM is

$$E_1 \to E_2 \to E_3 \Rightarrow (E_4 = 95\%, E_5 = 5\%)$$

where E_1 through E_5 are security events. This rule indicates that after the sequence of events E_1, E_2, and E_3, the probability of E_4 occurring is 95% and that of E_5 is 5%. This rule is generated based on previously observed data. In TIM, rules are modified over the lifetime of a system in order to keep the rule set representative and manageable.

1.3.3 Evaluation of Anomaly Detection

Anomaly detection provides a method to detect penetrations without requiring specific knowledge about the operating system or its security flaws. It is also the only viable technique to detect masqueraders.

Nevertheless, it is difficult to establish behavior patterns for users in many environments and to determine the threshold values that signal anomaly. In addition, anomaly detection alone cannot detect all kinds of intrusions, since not all penetrations produce an identifiable anomaly. Moreover, an experienced attacker can change his behavior slowly to avoid detection by an anomaly detection system.

1.4 Misuse Detection

Misuse detection is concerned with the detection of user actions that are suspicious, that resemble known intrusions, that exploit known vulnerabilities in the system, or that are in direct violation of the security policy. The goal of misuse detection is to identify these suspicious actions (or misuse signatures) and check for the occurrences of these actions in audit trails. Misuse signatures are often crafted by security experts who are familiar with the vulnerabilities in the target system, the internals of the operating system, and known intrusions. To date, there is no systematic way to identify misuse signatures, and misuse detection is highly driven by known intrusion scenarios and known system vulnerabilities.

There are many misuse detection systems being developed [38, 32, 24]. They differ in the way attack signatures are represented and the mechanisms used for checking occurrences of signatures in the audit trails. They can be classified into four approaches: expert systems (if-then-else), state transition analysis, pattern matching, and model-based intrusion detection. We discuss these approaches, focusing on the expressive power of the signatures as well as on the efficiency of the detection mechanisms.

1.4.1 Expert Systems

Most early misuse detection systems employ expert systems to detect penetrations. NIDES [38], W & S [53], MIDAS [45], and NADIR [26] have expert-system components that supplement their anomaly detectors. In these systems, penetrations are encoded as expert system rules. A rule has the form "if *condition* then *action* " where *condition* specifies constraints on individual fields of audit records and *action* specifies the actions to be taken when the rule is fired. An action can be the assertion of new facts, or the raising of a user's suspicion rating. The rules may recognize single auditable events or a sequence of events that represent a penetration scenario. An example of an expert rule of NIDES is shown in Figure 1.1.

```
1.    rule[SimuLogon(#1;*):
2.        [+tr:transaction]
3.        [+se:session|userid == tr.userid]
4.        [?|se.terminal != tr.terminal]
5.    ==>
6.        [!|printf("SimuLogon: user %s at terminals %s, %s\n",
                    tr.userid, tr.terminal, se.terminal)]
```

Figure 1.1: An Expert Rule of NIDES

This rule, *SimuLogon*, is written in the expert-system specification language PBEST [38]. It detects a user logging in on a terminal while already logged in somewhere else. When a user logs in, a transaction fact is asserted, and a session fact is added to the fact-base of the expert system if the login is successful. The rule checks each *transaction* fact to determine if there is any *session* fact with the same userid field (Lines 2 and 3). If such a session exists, it compares the terminals of the transaction fact with the session fact, and fires the rule if they are different. When the rule is fired, a message is written to the terminal of the security officer (Line 6).

One advantage of using expert-system rules to represent intrusions is that the mechanism for checking audit trails is provided automatically – the expert system interprets audit records as facts and determines whether they satisfy the rules. However, using expert-system rules to represent a sequence of actions is nonintuitive, and updating is difficult for all but experienced knowledge-base programmers. Also, it is difficult to use expert system rules to detect cooperating attacks. Additionally, expert systems are general tools, and as such, are less efficient than a monitoring system handcrafted for audit analysis.

The Distributed Intrusion Detection System (DIDS)

The Distributed Intrusion Detection System (DIDS) [46] monitors hosts in a local area network for intrusions. It has a signature analysis component that uses a different way to represent intrusions. An attack signature consists of a key event and the context in which an occurrence of the event is to be considered an intrusion.

Subject	userid=dragyn
Object	NULL
Action	rlogin
Context:	The account was fingered within the last 5 seconds

Table 1.1: An Example Signature

The attack signature in Table 1.1 captures what an attacker might do when he logs in as another user, assuming he has the password of the victim. The attacker first executes the *finger* command to check whether the victim is on the system. If not, he executes the *rlogin* command to enter the system as the victim. Therefore, the signature defines that an *rlogin* action is suspicious in the context where the account was fingered within the last 5 seconds.

In general, the context is an abstraction of all previous events in the system. An attribute of context may be the origination of a session: whether it is from the console or from a host outside the domain. The notion of context is important in identifying penetrations, as an action could be perfectly legitimate in one context, but indicate an intrusion in another context. The signature-analysis mechanism recognizes a signature based on both the actions and the context. It also maintains the current context of the system and of each user session.

Tracking Users' Movement Across the Network

Recognizing the importance of aggregating users' activities in a distributed environment, DIDS pays particular attention to tracking user movements across the network. DIDS implemented the Distributed Recognition and Accountability algorithm (DRA) [31] which assigns a unique *Network Identifier* (NID) to each session and maps remotely created sessions to the originating sessions (sessions initiated from physical devices or from hosts outside of the monitored domain). The algorithm observes audit data generated by all monitored hosts and the network monitor to keep track of user movements across the network. It uses a directed graph to relate remote sessions to the originating sessions. The unique NID of a session is considered part of the context of a session. Two sessions with the same NID are considered to be owned by the same individual so that the activities of the sessions are aggregated together.

In DIDS, both the prototype signature analysis mechanism and the DRA algorithm are implemented using the Clips [39] expert-system shell. DIDS consists of a centralized data analyzer that analyzes the audit data collected from the hosts in a local area network. It is also the first intrusion detection system designed to monitor the hosts on a local area network. However, as the analysis is centralized, DIDS as it stands, cannot be used in a large network consisting of many hosts owing to performance limitations.

1.4.2 State-Transition Analysis

The state-transition analysis approach [41] was implemented in USTAT [24, 25]. USTAT is a rule-based expert system for detecting intrusions in real-time based on state-transition analysis.

In state-transition analysis, an intrusion is a sequence of actions performed by the attacker that leads from some initial states to a compromised state. A state is a snapshot of the system representing the values of all volatile, semi-permanent and permanent memory locations on the system. It is represented by the values of a set of system attributes. The initial state corresponds to the state of the system just before the start of the intrusion, and the compromised state corresponds to the state resulting from the completion of the intrusion. One or more intermediate state transitions may occur between the initial and the compromised state. After identifying the initial and compromised states, the main step is to identify the key actions, called signature actions that, if omitted, would prevent the intrusion from completing successfully. This information is represented graphically using state-transition diagrams, and used for generating the expert rules for detecting the intrusion.

State-transition analysis focuses on the effects that the individual steps of an intrusion have on the state of the computer system. The rule base is independent of audit records and thus is easier to read. It also has the ability to detect cooperating attackers and attacks across user sessions. However, this model can represent an intrusion only as a totally ordered sequence of signature actions; it does not allow more complex ways of specifying penetrations such as partially ordered actions. Also, there is no general-purpose mechanism to prune partial matches of attacks other than through assertion primitives built into the model.

1.4.3 Pattern Matching

The pattern matching approach [32] by Sandeep for detecting intrusions deals with the four types of intrusions below; they are based on the signatures used for their detection.

1. Existence: The existence of an event in the audit trail indicates an intrusion.

2. Sequence: The occurrence of a sequence of events in the audit trail indicates an intrusion.

3. Partial order: The occurrence of a partially ordered sequence of events in the audit trail indicates an intrusion.

4. Duration and interval: The occurrence of a sequence of events within a certain duration or time interval in the audit trail indicates an intrusion.

In this approach, an attack signature is a Colored Petri Network (CPN) [28]. A CPN is a directed graph in which nodes represent states and edges represent transitions. Optional guards with expressions can be placed at transitions. These expressions permit assignment to the token local variables that flow past the transition. There can be several start states, but only one final state. At the start of a match, a token is placed in each initial state. A CPN may have a set of variables associated with it, representing the context.

The model has the expressive power to specify partially ordered actions and the timing constraints among actions. The objective is to translate the intrusion detection problem into the pattern matching problem: the audit trail is an abstracted event stream, and the detector is a pattern matcher. An advantage of using the pattern-matching approach is that it has been extensively studied. It is amenable to several optimizations that can make a system built around it practical and efficient. Therefore, using pattern matching to detect penetrations is more efficient than using expert systems.

1.4.4 Model-based Intrusion Detection

Model-based intrusion detection [21] deals with intrusions at a higher level of abstraction than audit records. In model-based detection, scenario models that represent the characteristic behavior of intrusions are built. Scenario models allow administrators to generate penetrations in an abstract manner. Model-based techniques thus differ from rule-based techniques, which simply match audit records to expert rules.

1.4.5 Evaluation of Misuse Detection

A significant advantage of misuse detection is that it can guarantee detection of known intrusions if the signatures of the intrusions are included in the database of the misuse detector. However, current approaches to identifying misuse signatures are mostly driven by previous intrusions and known vulnerabilities. Although it is relatively easy to include an intrusion into the misuse database to detect subsequent occurrences of the intrusion, it is difficult to capture all the variants of an intrusion. We illustrate the difficulties involved by an example presented in Chapter 3.

The fundamental limitation of current misuse detection approaches is their inability to deal with novel attacks and unknown system vulnerabilities: it is infeasible to identify all possible undesired behaviors. Also, it is not intuitive to enforce a security policy using a misuse detection system, as the policy has to be expressed in terms of signatures of prohibited actions.

1.5 Specification-based Monitoring

This dissertation describes a new approach to intrusion detection, called specification-based monitoring. The idea is to write security specifications for the security-critical programs (e.g., privileged programs) that describe their desirable behavior, and to monitor the execution of these programs for violations with respect to the specifications. A specification is driven by the functionality of the programs and the system security policy.

In more detail, a specification captures the valid operation sequences of the execution of one or more programs. A sequence of operations performed during an execution of the program that is outside the specification are considered a security violation. As the specification determines whether an execution trace of a program is valid or not, it is called a *trace policy.* Audit trails are used to monitor the execution of a program.

Grammars are used as specifications of valid operation sequences of programs, where the alphabets of the grammars are system operations. We developed a novel type of grammar, *parallel environment grammars* (PE-grammars), for specifying trace policies. A PE-grammar can describe parametric and context-sensitive languages. It can describe many different classes of trace policies that are important to security. Besides, it can describe traces of a concurrent program execution in which the concurrent processes are synchronized. Also, a PE-grammar serves as a design specification for the development of a parser that determines whether a trace of a program execution is a sentence of the grammar. Intrusion detection thus becomes parsing.

We designed a specification-based monitoring system for a distributed system. The monitoring system consists of *analyzers* distributed over multiple hosts in the distributed system. Each analyzer monitors a subject, which can be a single program execution, multiple program executions, a user, multiple users, etc. The system combines distributed data collection and filtering as well as decentralized data analysis. We developed a prototype execution monitor for a Unix system and the trace policies for setuid root programs in Unix. The prototype detects intrusions that exploit vulnerabilities in these programs in real time.

Because of the way these programs are specified, their specifications are independent of the vulnerabilities in these programs. Thus, specification-based monitoring has the potential to detect attacks that exploit unknown vulnerabilities in these programs.

1.5.1 Terminology

This section explains the terms used throughout the dissertation.

- *Attack*: An attack is any set of actions whose purpose is to compromise the integrity, confidentiality, or availability of a resource. The set of actions may be performed by a single attacker or by a group of cooperating attackers. An attacker exploits vulnerabilities in a system to gain necessary privileges to achieve his/her goal.

- *Exploitation*: An exploitation is a set of actions that result in a violation of the security policy of a computer system. Intruders exploit system vulnerabilities or flaws to gain unauthorized access to the system.

- *Flaw*: A flaw is an error of commission omission or oversight in a system that allows protection mechanisms to be bypassed [36]. A program flaw is an error in a program, either in design or implementation, that causes the program to malfunction. We use vulnerabilities and flaws synonymously.

- *Intrusions*: An intrusion is an attack. Attacks and intrusions are used synonymously in this dissertation.
- *Security policy*: A security policy is a set of laws, rules, and practices that regulate how an organization manages, protects and distributes sensitive information.
- *Privileged program*: A privileged program is a program that is executed with special privileges, enabling it to bypass the system security mechanism in order to accomplish its task. In Unix, examples are those programs that are owned by root and have the setuid bit on. They are refered to as *setuid root* programs throughout the dissertation.
- *Privileges*: Privileges are capabilities given to subjects so that they can perform operations that are not accorded normal users.
- *Vulnerabilities*: A vulnerability is a weakness in automated system security procedures, administrative controls, or internal controls that could be exploited by a thread to gain unauthorized access to information or to disrupt critical processing [36].

1.5.2 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 presents a few attack scenarios to motivate our work. These intrusions exploit the vulnerabilities of privileged programs in Unix to gain unauthorized access to the system. The goal is to illustrate the subtleties of the vulnerabilities and of the intrusions that exploit the vulnerabilities. Chapter 3 describes the terms and concepts that are used later in the dissertation. Chapter 4 describes a language for specifying trace policies. It first gives the formal definition of the language in a general setting and then describes how it is used to specify trace policies. Chapter 5 presents several example trace policies for the problematic programs described in Chapter 2. Chapter 6 presents the design of a distributed monitoring system and the implementation of a prototype. It also describes our experience with the prototype. Chapter 7 discusses various issues of our work and suggests future research.

Chapter 2

Intrusion Scenarios

In this chapter we present several vulnerabilities in Unix privileged programs that can be exploited to gain unauthorized access to the system. We also include several intrusion scenarios to illustrate how these vulnerabilities are exploited. We focus on the fundamental problems behind the vulnerabilities as well as the relevant system actions associated with an exploitation. The goal is to give the reader a better understanding of the vulnerabilities, the ways they are exploited, and the ways the exploitations can be detected by monitoring the system actions.

2.1 Rdist

Rdist [48] (Remote File Distribution Program) is a Unix utility for maintaining identical copies of files over multiple hosts. Since its release in 4.3 BSD Unix, *Rdist* has been widely used by system administrators and users on almost every major Unix Platform.

Rdist is normally invoked by a user in the host where the master copies reside to update the copies of files in remote hosts. It preserves the content, the owner, the group, the mode and the modification time of the master. *Rdist* employs the client-server approach to perform file updates on the remote hosts. For each host whose copies of files are to be updated, *rdist* invokes the *rdist server* [1] (hereafter referred to as *rdistd*) in the remote host using the *rcmd(3)* interface [2]. After that, a connection is established between *rdistd* and

[1] In most Unix systems, the single executable file of *rdist* (e.g., */usr/ucb/rdist*) contains the code of the *rdist* client and the *rdist* server. The *rdist* server is invoked with a special flag, -Server [48].

[2] The *rcmd(3)* interface is also used by the *rsh(1c)* command.

rdist through which commands are sent to the server to perform the file update. *Rdist* is a setuid root program because root privileges are required in order to use the *rcmd* routine in which privileged ports are used for authentication. Therefore, *rdist* and *rdistd* run with root privileges that enable them to perform operations inaccessible by normal users – this is the basis for the exploitation.

Rdist has a race-condition flaw, [3] which enables an attacker to acquire root privileges illegally. The flaw relates to the way rdist updates a file as well as to the semantics of the *chown* and *chmod* system calls in dealing with symbolic links. Specifically, the flaw enables a nonprivileged user to change the permission mode of any file in the system. It has been exploited by attackers to set the setuid bit of a system shell (e.g., */bin/sh*), resulting in a setuid root shell that is publicly executable.

The flaw exists in the server portion of the program. When *rdistd* is instructed by the client to update a file, say $F1$, *rdistd* does not overwrite the file directly; instead, it creates a temporary file, writes the new data to the temporary, changes the ownership and the permission mode of the temporary to correspond to the master, and if necessary, renames the temporary file to $F1$.

Step	System Call
1.	`fd = creat("/ko/rdista768");`
2.	`write(fd, ...);`
3.	`close(fd);`
4.	`chown("/ko/rdista768", owner);`
5.	`chmod("/ko/rdista768", pmode);`
6.	`rename("/ko/rdista768", "/ko/data");`

Table 2.1: System Call Sequence of *Rdist*

Table 2.1 summarizes the system calls that *rdistd* makes to update the file */ko/data*. In Step 1, *rdistd* creates a temporary file (*rdista768*) in the directory where */ko/data* resides using the *creat* system call. In Steps 2 and 3, it writes the new contents of the file to the temporary file and closes the file. In Steps 4 and 5, it uses *chown* and *chmod* to change the owner and the mode of the temporary file. Lastly, in Step 6, it renames the temporary file */ko/rdista768* to */ko/data* using the *rename* call. Both *chown* and *chmod* take a symbolic

[3] The flaw exists in 4.3 BSD Unix and other variants of Unix.

path name as parameters. If the path name is a symbolic link, *chown* changes the ownership of the symbolic link itself, while *chmod* changes the permission mode of the file to which the symbolic link points. The sequence of system calls looks perfectly legitimate at first glance. The effect is that the content and the permission mode of the file named */ko/data* are set to that of the master. However, the effect of the sequence of operation is correct only when the sequence of operations is done atomically, without any intervening operations.

We describe an intrusion scenario in which an attacker exploits the sequence of system calls made by *rdistd* to change the permission mode of a system shell (*/bin/sh*). In the exploitation, the attacker performs operations that change the meaning of the temporary file name while *rdistd* is updating the file.

Step	System calls done by rdistd	System calls done by the attacker
0'.		`execve("/usr/ucb/rdist");`
1'.	`fd = creat("/ko/rdista768");`	
2'.	`write(fd, ...);`	
3'.	`close(fd);`	
4'.		`rename("/ko/rdista768", "/ko/tmp");`
5'.		`symlink("/bin/sh", "/ko/rdista768");`
6'.	`chown("/ko/rdista768", owner);`	
7'.	`chmod("/ko/rdista768", pmode);`	
8'.	`rename("/ko/rdista768", "/ko/data");`	

Table 2.2: System Calls done by *Rdist* and the Attacker

Table 2.2 describes the steps in the exploitation. The attacker, who has acquired normal user access to the system, invokes *rdist* with appropriate commands (Step 0') to update a file (*/ko/data*) in the local host, in effect causing *rdistd* to update the file. *Rdistd* follows the normal procedure as described to update the file. In Step 1', *rdistd* creates the temporary file. In Step 2' and 3', *rdistd* writes the new data to the temporary file and closes the file. Meanwhile, the attacker renames the temporary file (Step 4'), creates a symbolic link and names it after the temporary file (Step 5'). The symbolic link points to the target file, the permission of which he desires to change (*/bin/sh*) [4]. Then, in Steps 6' and 7', *rdistd* performs the *chown* and *chmod* system calls to change the permission mode of the

[4] An attacker can have better control of the timing and the final permission mode of the target file if he invokes the *rdist* server directly and gives internal commands to the *rdist* server to update a file.

(a) Normal Sequence of Operations

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...

Step 1. rdist: fd = creat("/ko/rdista768");

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	**534**	**534**	**srwxr-xr-x**	**ko**	

Step 2 & 3. rdist: write(fd, ...); close(fd);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	534	534	srwxr-xr-x	ko	**aiejkdkb..**

Step 4. rdist: chown("/ko/rdista768", ko);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	534	534	-rwxr-xr-x	**ko**	aiejkdkb...

Step 5. rdist: chmod("/ko/rdista768", 04755);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	534	534	**srwxr-xr-x**	ko	aiejkdkb...

(b) Attack Sequence of Operations

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...

Step 1'. rdist: fd = creat("/tmp/rdista768");

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	**534**	**534**	**srwxr-xr-x**	**ko**	

Step 2' & 3'. rdist: write(fd, ...); close(fd);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	534	534	srwxr-xr-x	ko	**aiejkdkb...**

Step 4'. Attacker: rename("/ko/rdista768", "/ko/dummy");

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/dummy	534	534	srwxr-xr-x	ko	aiejkdkb...

Step 5'. Attacker: symlink("/bin/sh", "/ko/rdista768);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/master	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/dummy	534	534	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	**537**	**537**	**lrwxrwxrwx**	**ko**	**/bin/sh**

Step 6'. rdist: chown("/ko/rdista768", ko);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	-rwxr-xr-x	root	^A\043^A...
/ko/data	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/dummy	534	534	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	537	537	lrwxrwxrwx	**ko**	/bin/sh

Step 7'. rdist: chmod("/ko/rdista768", 04755);

Pathname	Inode	Inode	perm	owner	content
/bin/sh	3867	3867	**srwxr-xr-x**	root	^A\043^A...
/ko/data	4234	4234	srwxr-xr-x	ko	aiejkdkb...
/ko/dummy	534	534	srwxr-xr-x	ko	aiejkdkb...
/ko/rdista768	537	537	lrwxrwxrwx	ko	/bin/sh

Figure 2.1: Operation Sequence of *rdist*

temporary file */ko/rdista768*. However, since the path name */ko/rdista768* was changed to a symbolic link pointing to */bin/sh*, *chown* changes the owner of the symbolic link itself in Step 6'. In Step 7', *chmod* changes the permission mode of */bin/sh* to set the setuid bit. Therefore, the attacker can obtain root privileges by invoking */bin/sh*.

Figure 2.1 compares the system operations occurring in a normal execution of rdist (shown in Table 2.1) and in the exploitation (shown in Table 2.2). Also, it shows the relevant portion of the file system state and highlights the changes to the state of the file system caused by each operation. In both cases, the file system starts off in the same state, in which the setuid bit of */bin/sh* is off and the setuid bit of master file (*/ko/master*) is on. Also, *rdistd* performs the same sequence of system calls with the same parameters in both situations.

After the first three steps, the resulting states are the same in both cases. After Step 1 (1'), a new file (inode = 534) is created and a new path name */tmp/rdista768* is created to refer to inode 534. After Steps 2 and 3 (2' and 3') , the content of the file identified by inode 534 is modified to that of the master. In Steps 4 and 5 of a normal execution, *chown* and *chmod* change the owner and permission mode of the file of inode 534, which is the the temporary file created in Step 1. However, in the exploitation, the attacker performed Step 5' and 6', which change the file system so that the path name */ko/rdista768* now refers to a symbolic link pointing to */bin/sh* instead of the temporary file created in Step 1'. Therefore, in Step 7', *chmod* changes the permission mode of */bin/sh* to that of the master file */ko/master*, in effect setting the setuid bit of */bin/sh*.

We can detect this intrusion in two ways. First, we can detect it by just looking at the operations performed by *rdistd*. Although the sequence of calls and parameters performed by *rdistd* are the same in both cases, the actual physical files modified by the calls are different. Table 2.3 shows the audit records generated by a SUN Solaris BSM audit system for *rdistd* under normal execution and under the exploitation. In Unix, each physical file is identified by a unique *inode* number. In the normal sequence, *chmod* changes the file corresponding to the *inode* of the temporary file created. However, in the sequence of audit records of *rdistd* under the exploitation, *chmod* changes the file whose *inode* is different from that of the temporary file created. Thus, by checking the *inode* of the files *chown* and *chmod* change, which are usually available in the audit trail, we can detect occurrences of this intrusion.

Second, we can detect the intrusion by checking whether operations that change the

Audit Records in a Normal Execution			Audit Records in an Intrusion		
Event	path	inode	Event	path	inode
`creat`	`"/ko/rdista768"`	`534`	`creat`	`"/ko/rdista768"`	`534`
`chown`	`"/ko/rdista768"`	`534`	`chown`	`"/ko/rdista768"`	`534`
`chmod`	`"/ko/rdista768"`	`534`	`chmod`	`"/ko/rdista768"`	`3867`

Table 2.3: A Comparison of the Audit Records in a Normal Execution and in an Intrusion

meaning of the temporary file name are performed by other processes between the time *rdistd* makes the *creat* call and the *chown* call. Some ways of specifying the valid order among these operations are needed in order to detect the intrusion in that way.

In the exploitation, *rdist* is made to perform operations outside the behavior intended for it. Specifically, *rdist* is used by a user to update his files in a host; therefore, when invoked by a user, *rdist* should update only the file owned by the user. In addition, *rdist* should change the permission mode and the ownership of only the files it creates, not other files. By specifying the valid operations that are intended to be performed by *rdist*, the exploitation can be detected.

2.2 Fingerd

Our second example is the finger daemon (*fingerd*), which provides to remote clients information on system status and individual users. *Fingerd* listens for connections on the TCP port 79, also known as the finger port. Once connected, it reads a single line containing the request and invokes the finger program with the request as parameter, which in turn collects the information from various status files in the system and reports the information back to the requestor. *Fingerd* is executed with root privileges because it listens to the finger port, which is a privileged port in Unix.

The finger daemon versions before 1989 contain a well-known buffer-overflow bug which has been exploited to cause *fingerd* to execute any arbitrary program. Specifically, *fingerd* reads a finger request using the *gets() C* library call, which does not restrict the size of the input. As a result, when fingerd is given a very long request message, the data could overwrite the memory location beyond the buffer, causing unexpected behavior.

Figure 2.2 shows an excerpt of the fingerd program that is related to the flaw and the

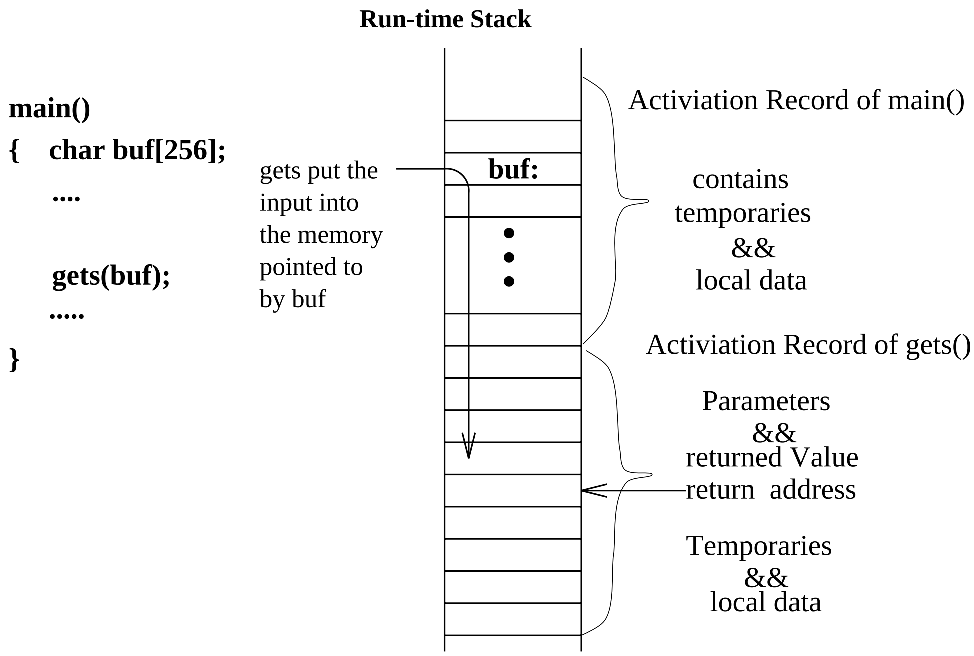

Figure 2.2: Fingerd Attack

content of the run-time stack when *gets()* is invoked. When *gets()* is called, the run-time stack consists of the activation record of the main function followed by the activation record of *gets()*, which consists of the parameters, the return address, and local variables. When an attacker sends a long request message consisting of code to the finger daemon, *gets()* fills the read buffer and beyond with the message so that the buffer now contains the injected code and the return address is overwritten with an address pointing to the buffer. When the subroutine returns, it branches into the buffer and executes the attacker's code.

Obviously, we can detect attacks that exploit this vulnerability by checking the data *fingerd* reads. However, most auditing systems do not record the content of read/write operations because of the huge volume of data associated with these operations. In fact, most auditing systems do not even record the *read* and *write* system calls; they just record the *open* and *close* system calls. Therefore, it is not possible to tell whether from the audit trails, *fingerd* is given a long request message or just a normal-length message. However, if we look at the operations performed by *fingerd*, we can detect the intrusion. Table 2.4 compares the operations performed by fingerd in a normal execution and under an attack.

Note that the finger daemon should only execute the finger program (*/usr/bin/finger*) after it reads the request message. Any other operation performed by *fingerd* is an indication of a possible intrusion. Therefore, by specifying the valid operations that can be performed by the finger daemon, we can detect intrusions that exploit the vulnerability.

Step	Normal Operation Sequence	Attack Operation Sequence
1.	`read(stdin)`	`read(stdin)`
2.	`execve("/usr/ucb/finger")`	`open("/tmp/worm")`
3.		`fork()`

Table 2.4: System Call Sequence of Fingerd

The buffer-overflow bug is not unique to fingerd, but exists in many other privileged programs, for example, httpd, sendmail, syslog, etc.. In fact, it still exists in recent releases of the Unix operating system [11, 12].

2.3 Mail

This section describes a vulnerability in the 4.2 BSD UNIX *mail* utility and several related intrusions that exploit this and similar vulnerabilities. The goal here is to illustrate the difficulties in identifying all variants of an intrusion that exploit the same vulnerability.

The vulnerability relates to the permission mode of root's mail-box file and the way the *mail* utility delivers mail: it changes the owner of root's mail-box file to root without checking the permission mode of the file. The vulnerability in the *mail* utility actually exists in the back-end mailer program, *binmail*. *Binmail* is a setuid root program since it needs to append to the mail-box of any user. When a user invokes the *mail* utility program to send a message, it invokes the mail intermediary and passes it the message for delivery. The mail intermediary (*sendmail* in most systems), after determining that the mail message is to be delivered locally, invokes the back-end mailer (usually *binmail*) to delivery the message. To deliver a message to a user, *binmail* appends the message directly to a user's mail-box file, which is in the mail spool directory. Last, it changes the owner of the mail-box file to the user. However, it does not check the permission mode of the mail-box file before it changes the owner of the file; it assumes a mail-box file should be readable and writeable only by the owner and has the setuid bit off. If an attacker can create a root's mail-box file that is

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

Table 2.5: An Example Intrusion of *Mail*

publicly executable and has the setuid bit on when *binmail* is executed, *binmail* will change the owner of root's mail-box file to root, resulting in a setuid file which is owned by root and publicly executable.

We describe an intrusion that exploits the *mail* utility to illegally acquire root privileges. Table 2.5 summarizes the steps of the intrusion. In step 1, the attacker creates a copy of *csh*(1) and names it after root's mail-box file (*/usr/spool/mail/root*). In general, the mail-box of a user is the file named "*/var/spool/mail/¡username¿*", which should be owned by "*username*" and readable and writeable only by the owner. The attacker can do that only when there is no unread mail of root so that the mail-box file of root does not exist. In Step 2, the attacker enables the setuid bit of the fake mail-box file. In steps 3 and 4, the attacker creates and sends a message to root using the *mail* utility. When the mail subsystem delivers the message, it changes the owner of the file to root, resulting in a setuid root shell which is publicly executable. As a result, the attacker is able to obtain a root shell by executing the publicly executable shell program.

It is straightforward to encode this sequence of operations as expert rules in a rule-based misuse detection system. However, it is not trivial to identify all variants of this intrusion. For example, steps 3 and 4 can be replaced by a single "mail root" command in which the attacker keys in the content of the mail interactively. Also, some steps can be omitted and the order of them can be permuted.

Table 2.6 presents two variations of the intrusion, both of which create a counterfeit root's mail-box file that is publicly executable and has the setuid bit on before the *mail* utility is invoked to delivery a mail message to root. In the first variant, the attacker creates a copy of *csh*(1) and names it after root's mail-box file (*/usr/spool/mail/root*) in step one. In

Variant No. 1		
Step	Command	Comment
1.	`% cp /bin/csh /usr/spool/mail/root`	- create a counterfeit mail file of root
2.	`% ln -s /tmp/mroot /usr/spool/mail/root`	- create an alias of the mail file
3.	`% chmod 4777 /tmp/mroot`	- make the mail file setuid
4.	`% mail root`	- mail root and give input interactively
5.	`% /usr/spool/mail/root`	
6.	`root%`	

Variant No. 2		
Step	Command	Comment
1.	`% cp /bin/csh /usr/spool/mail/tmp`	- creat a temporary copy of C shell
2.	`% chmod 4777 /usr/spool/mail/tmp`	- make the copy setuid
3.	`% mv /usr/spool/mail/tmp /usr/spool/mail/root`	- rename the temp copy
4.	`other% mail root`	- another user send mail root
5.	`% /usr/spool/mail/root`	
6.	`root%`	

Table 2.6: Two Variants of the *Mail* Intrusion

step 2, the attacker creates an alias of the mail-box file by making a symbolic link pointing to the file. In step 3, instead of changing the permission mode of the counterfeit mail-box file directly, the attacker invokes *chmod* with the symbolic link as parameters, in effect changing the permission mode of the mail-box file to setuid and publicly executable. In step 4, the attacker invokes *mail* and interactively keys in an empty message.

In the second variant, the attacker first creates a temporary copy of *csh* in the *mailbox* directory, */usr/spool/mail*. In step 2, the attacker enables the setuid bit of the copy of *csh* and makes it publicly executable. In step 3, the attacker renames the temporary copy to the mail-box file of root, in effect creating a counterfeit mail-box file of root. Then the attacker does not send mail to root, but waits until another user sends mail to root. After a user sends mail to root using *mail*, the attacker executes the setuid mail-box file of root to obtain root accesses.

The state transition analysis approach [25] touches on this issue. It attempts to identify the key actions in an intrusion without which the intrusion will not succeed. However, this approach can only identify variants of the intrusion that consist of the same sequence of key actions. There could be other variants of the intrusion using actions other than the key actions. For example, in [25], the authors identify three key actions: 1) the attacker creates a counterfeit mail-box file of root, 2) the attacker changes the permission of the counterfeit file, and 3) the attacker changes the ownership of the mail-box file. However, these three signature actions cannot represent all possible intrusions that exploit this vulnerability of *mail*. For instance, the signature fails to represent the second variant of the intrusion we presented above because the attacker uses *mv* to create the counterfeit mail-box file.

It is not straightforward to identify all possible sequences of actions that exploit a vulnerability. One needs to know the intrinsic problem behind the vulnerability. The flaw presented in this section allow the system to move into a compromised state if the mail-box file of root is setuid and publicly executable when *binmail* is run, or more exactly, when *binmail* performs the *chown* call to change the owner of root's mail-box file. Therefore, in order to detect exploitations of this vulnerability, we want to 1) detect the actions by the attacker that result in a counterfeit mail-box file of root that is setuid and publicly executable, and 2) detect when *binmail* changes the owner of the file */var/spool/mail/root* to root while the mode of the file is setuid and publicly executable.

2.4 Binmail

This section describes a flaw in *binmail* that enables a normal user to replace any file in the system. *Binmail* is part of the mail system in Unix. It is responsible for directly appending a mail message to users' mail-box files. *Binmail* is a setuid root program because it has to append to the mail-box file of any user (including root). It is not intended to be executed by users directly, but by *sendmail* to deliver mail locally.

The vulnerability is concerned with the way in which *binmail* creates a temporary file to hold a user's mail-box. When invoked to deliver a mail message, *binmail* first creates a temporary file in the */tmp* directory. It uses the *mktemp* library call to obtain a file name for the temporary file. It deletes any existing temporary file with this name. Then it creates the temporary file and copies the current mail-box file of the recipient to the temporary file.

Table 2.7 summaries the operations *binmail* performs. In Step 1, *binmail* makes the library call *mktemp()* to obtain a file name for the temporary file. In Step 2, it performs a

Step	Operation
1.	`s = mktemp()`
2.	`stat(s)`
3.	`if exist(s) unlink(s)`
4.	`fd = open(s, CREAT | TRUNC | WRITE | READ)`
5.	`write(fd, ...)`

Table 2.7: Operation Sequence of *Binmail*

stat call to check the existence of a file named by the temporary file name. It deletes the file if it exists using *unlink* (Step 3). In Step 4, *binmail* uses the *open* system call to create the temporary file. It calls *open* with the *TRUNC* and *CREAT* flags, which causes *open* to create the file if it does not exist, or truncate the file if the file already exists. Also, *open* follows symbolic links, that is, if the path name is a symbolic link, *open* operates on the file to which the symbolic link points.

We describe an intrusion in which an attacker causes *binmail* modify the password file in the system. An attacker first invokes *binmail* directly to deliver an empty message to himself. Knowing the file name of the temporary file that *binmail* will create, the attacker renames the temporary file, creates a symbolic link pointing to the file he wants to replace (in this case, */etc/passwd*) and gives it the original temporary file name. If the *symlink* operation is scheduled to execute after *binmail* performed Step 2 and before it performs Step 4, the temporary file name is actually a symbolic link to the password file when *binmail* performs the *open* call in Step 4. Therefore, *open* replaces the content of the password file.

2.5 Modifying the Password File

This section presents a scenario resulting in security failure because of improper synchronization between programs that access the password file. In Unix, the password file (*/etc/passwd*) contains the encrypted passwords of and important information about users. The password file is configured to be writeable only by the superuser; however, a normal user can change his/her password, login shell, as well as his/her full name recorded in the password file using the *passwd* program. Therefore, the password file could be accessed by multiple processes at the same time. For example, two administrators could edit the password file simultaneously using the *vi* editor program.

In this scenario, a user invokes the *passwd* program to change his password while an administrator is editing the password file. The two programs modify the password file simultaneously, leaving it with an incorrect content. The scenario begins when an administrator invokes *vi* to modify the password file. When being invoked, *vi* first creates a temporary file and copies the data from the password file into the temporary. Then, it displays the data on the terminal and lets the administrator edit the file. The new contents of the file is kept in the temporary file, which will be copied back to the actual password file when the administrator issues a write command. Meanwhile, a user invokes the *passwd* program to change his password. The *passwd* program prompts the user for the old password and a new password. It then updates the password file to reflect the changes of the password. At the same time, the administrator finishes editing the passwd file and issues a write command, causing *vi* to replace the password file with the new content.

Step	System Call	System Call
	vi	passwd
1.	`fd = open("/etc/passwd", RD);`	
2.	`fd1 = open("/tmp/ex0072", WR);`	
3.	`read(fd); write(fd1);`	
4.	`close(fd); close(fd1);`	
	EDITING	
5.		`fd = open("/etc/passwd", RD);`
6.		`fd1 = creat("/etc/ptmp");`
7.	`fd = open("/tmp/ex0072", RD);`	
8.	`fd1 = open("/etc/passwd", WR & TR);`	
9.	`read(fd); write(fd1);`	
10.		`read("/etc/passwd");`
11.		`write("/etc/ptmp");`
12.		`close(fd); close(fd1);`
13.	`read(fd); write(fd1);`	
14.	`close(fd); close(fd1);`	
15.		`rename("/etc/ptmp", "/etc/passwd")`

Table 2.8: Timing of the Events

Figure 2.8 summarizes the relevant system calls occurring in this scenario. The left column denotes the sequence of system calls made by *vi*. The right column denotes the sequence of system calls made by *passwd* to update the password file. When the administrator invokes *vi* to edit the password file, it first opens the password file for reading (Step 1) and creates a temporary file for writing (Step 2). It then copies the data of the password file into the temporary file via *read* and *write* calls [5] (Step 3). After it finishes, it closes the two opened files (Step 4). Then, context switches and *passwd* is scheduled to execute, which is about to update the password file. The *passwd* program opens the password file for reading (Step 5) and creates a temporary file for writing (Step 6). Context switches and *vi* opens the temporary file for reading (Step 7). Then, it opens the passwd file for writing (Step 8) with the TRUNC flag, causing the file to be truncated. After that, it performs a *read* system call to read the first block of the temporary file and a *write* system call to write the first block to the password file (Step 9). Context switches and *passwd* makes a *read* call which reads the entire file (Step 10). Since *vi* writes back only the first block of the original data to the password file, *passwd* reads only the first block of the file. Then, *passwd* updates the password entry of the user (which happens to be in the first block of the file) and writes the data to the temporary file */etc/ptmp*. After that, it closes the two opened files. Context switches again and *vi* writes the remaining data to the password file with another *write* call (Step 13) and closes the file (Step 14). Now, the password file contains the content updated by the administrator. However, at last, *passwd* renames the temporary file */etc/ptmp* to the password file, causing the current password file to contain only the first block of the original.

In fact, a normal user can invoke *passwd* and stop the passwd program when the program is in the process of updating the password file (e.g., between steps 10 and 11). After an administrator changes the password file, the user can continue the program (by sending a continue signal) so that *passwd* will perform the *rename* call and replace the password file with an old version (a version without the administrator's changes).

The detection of multiple processes that are modifying the password file requires the identification of all programs that can modify the password file. Moreover, some constraints on their relative order must be specified.

[5] The number of *read* and *write* calls *vi* uses to copy the file depends on the size of the file. The *vi* program reads and writes a file using a block size of 1024 bytes.

Chapter 3

The Model

3.1 System Model

A distributed system consists of a number of hosts that are connected by a network. The basic entities that perform operations on objects in the system (e.g., files) are processes. An event denotes an execution of an operation in the system, and is attributed to the process that performs the operation. Events happening in the system can be totally ordered, and the history of the system is the sequence of events that occurred since the system started.

Definition 3.1 (System Traces) The execution of a distributed system S produces a sequence of events

$$v_1, v_2 \ \cdots, \ v_i, v_{i+1}, \ \cdots,$$

which is called a *system trace* of the system. Each event v_i has an occurrence time, denoted by $C(v_i)$. Events are totally ordered, that is, $C(v_i) < C(v_{i+1})$ for all $i \geq 1$. A sequence of events $v_{l_0}, v_{l_1}, \ \cdots, \ v_{l_k}$ is a *subtrace* of a system trace if $l_0, l_1, \cdots, l_k$ is a subsequence of $1, 2, \cdots,$. Also two subtraces $v_1^a, v_2^a, \cdots, v_k^a$ and $v_1^b, v_2^b, \cdots, v_l^b$ are said to be *distinct* if and only if $v_i^a \neq v_j^b$ for all $1 \leq i \leq k$, $1 \leq j \leq l$.

The execution of a sequential process (or just process) p_l in the system produces a sequence of events

$$v_1^l, v_2^l, \ \cdots, \ v_i^l, v_{i+1}^l, \ \cdots, \ .$$

The sequence of events is called a *process trace*, and denotes the sequence of operations performed by the process from the time it starts to the time it terminates. A process trace is a subtrace of the system trace.

Definition 3.2 (Merge of Traces) Given two distinct subtraces V_1 and V_2 of V, the *merge* of the two traces is a subtrace of V, denoted by $V_1 \oplus V_2$, and is defined by $V_1 \oplus V_2 = v_1, v_2, v_3, v_4, \cdots, v_{m+n}$ if and only if there exists two subsequences of $1, 2, \cdots, m+n$, $i_1, \cdots, i_m$ and $j_1, \cdots, j_n$, s.t. $V_1 = v_{i_1}, v_{i_2}, \cdots, v_{i_m}$, $V_2 = v_{j_1}, v_{j_2}, \cdots, v_{j_n}$, and $C(v_i) < C(v_{i+1})$.

Definition 3.3 (Filter of Traces) A filter ∇_p is a function that maps a trace $V = v_1, v_2, \cdots, v_n$ to another trace V_s, a subtrace of V, where p is a predicate on the set of possible event attributes, in which V_s is obtained from V by removing all events v_i $(i \geq 0)$ in V s.t. $p(v_i) = false$. For example, given a predicate q s.t. $q(v)$ is *true* if only if the event e describes an operation performed by the user *ko*, $\nabla_q(V)$ is a subtrace of V that consists of events in which the operations are performed by *ko*.

3.2 Monitoring Programs

A program is a passive entity. To monitor a program means to monitor the executions of the program. An execution of a program is a *distributed process* $dp = \{p_1, p_2, \cdots, p_n\}$, $n \geq 1$, which consists of one or more processes. For instance, an execution of a sequential program is a distributed process consisting of a single process, while an execution of a distributed program or a concurrent program is a distributed process which consists of multiple processes.

Figure 3.1 depicts the distributed process associated with a *sendmail* execution in Unix, which consists of 5 processes. Assume that *sendmail* is invoked to deliver a mail message to two users. The *sendmail* program creates a number of processes to do the job. When executed, the top-level main process (6772) does not deliver mail directly, but creates a second-level child process. The child process creates a record in the mail queue directory, and then it creates a third-level child process. The third-level child process then creates a fourth-level child process, which looks at the record and performs the delivery. It creates a

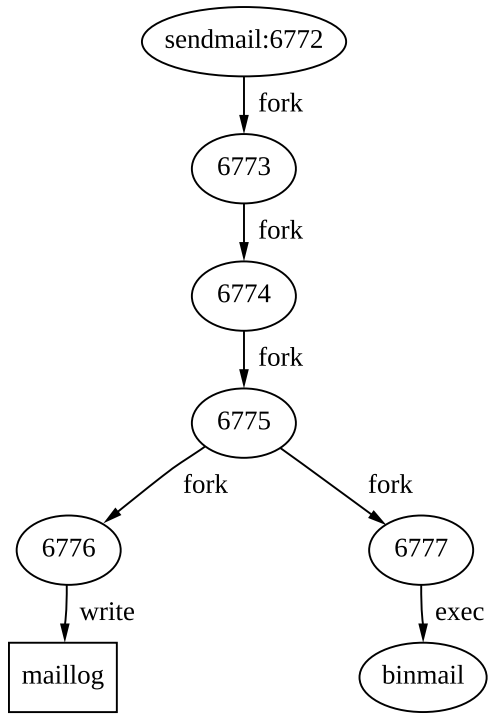

Figure 3.1: The Process Hierarchy of a Sendmail Execution

process (6776) to deliver mail to files, and another process (6777) to deliver mail to a user's mailbox.

The trace of a program execution is the sequence of events corresponding to the operations performed by the distributed process, which is the merge of the individual traces of the processes forming the distributed process. With $dp = \{p_1, p_2, \cdots, p_n\}$ denoting the distributed process, the execution trace is

$$V_{dp} = V_{p_1} \oplus V_{p_2} \oplus \cdots \oplus V_{p_n},$$

where V_{p_i} $(1 \leq i \leq n)$ is the execution trace of process p_i. V_{dp} is a subtrace of the system trace V.

A single program could have a number of executions existing at the same time. For example, two users can execute the same program at the same time, resulting in two distributed processes both running the program under the operating system. In some situations, we want to monitor one execution at a time. In other situations, we want to monitor all executions of a program. For example, in order to detect the synchronization failure in Section 2.5, we need to monitor all executions of *vi* by the administrator (i.e., root) and all executions of *passwd*. In this case, the input to the monitor is the merge of the filtered traces corresponding to all executions of *vi* (predicate: user root) and the traces of all executions of *passwd*.

3.3 The Subject of Monitoring

A trace policy describes the valid operation sequences of a single program execution, multiple program executions, a user, a group of users, a host, etc. The entity or entities are collectively called a *monitored subject*, or simply a *subject*. Therefore, a subject could be a distributed process, a group of distributed processes, a user, a group of users, or a host. The basic components of a subject are processes. A host refers to all processes running on the host. A user refers to all processes that are owned by the user.

Monitoring a subject means analyzing the sequence of operations performed by the subject. The execution trace of a subject is the time-ordered sequence of operations performed by the processes forming the subject. Each process has its own trace, and the subject trace is the merge of the individual traces. Figure 3.2 depicts the situation of monitoring a program execution which involves three processes. Each process produces an execution trace. The process traces are *merged* to form the single trace of the three processes. The latter trace is then the input to the monitor (the parser is driven by a specific grammar).

3.4 Characteristics of a Program Execution

A program execution is characterized by attributes on which its desirable behavior depends, including the user associated with it and the host on which it takes place.

3.4.1 The User of a Program

We associate with each program execution a user who is accountable for the actions of the program. In general, a program is a high-level tool used to manipulate data in the system; the services provided by the kernel are often too primitive for users to use directly. Users

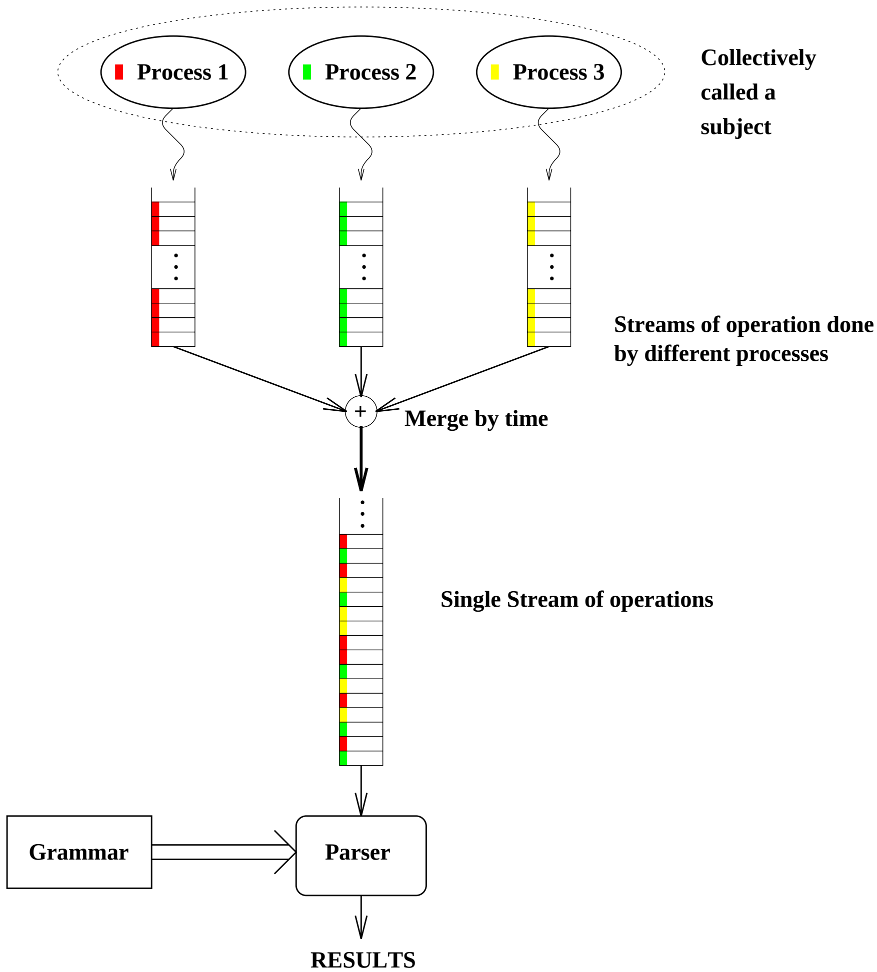

Figure 3.2: The Monitoring Model

invoke programs to accomplish a job on their behalf. Therefore, users should be accountable for the execution of programs. The valid execution trace of a program execution is generally a function of the user associated with the execution.

The idea is similar to the integrity policy of Clark and Wilson [9] in which high-integrity data can only be accessed by authorized users using a particular program. The concept of a user or a process as a subject in an operating system is well understood. Here, we employ both users and programs as subject attributes.

3.5 Security-Relevant Aspects of Program Behavior

In this section, we discuss and identify aspects of program behavior that are relevant to security. The goal is to motivate the specification language for specifying trace policies. The following four aspects of program behavior are security-relevant considerations:

1. Accesses of system objects (e.g., files).

2. Sequencing, should do operations in some particular order.

3. Synchronization among processes of a parallel program or among processes of related programs, access to shared data objects.

4. Race conditions.

3.5.1 Accesses

A simple but intuitive aspect of the behavior of a program is the set of objects being accessed. In this case, a program is treated as the subject of access control. It is necessary to restrict the accesses of a privileged program. As in the examples shown in Chapter 2, a privileged program could be forced by an attacker with normal user privileges to release system privileges or expose the system to the attacker. Therefore, by specifying the valid accesses of a privileged program, it is possible to detect attacks that trick the privileged program into performing accesses outside the specification.

In addition, one can specify the access policy of a suspect program so that even if the program is a Trojan horse, any access to files outside the specification can be detected. For this kind of access policy, the specification is just the sequence of actions that are allowed to appear in the traces of the program.

3.5.2 Sequencing

In some situations, not only the set of operations performed by a program is of concern, but also the order of these operations. For example, one may want to make sure that the *login* program reads the password file before it lets a user enter the system by executing a shell program and passing the control to the user. Thus, we want to make sure that the operation of reading the password file appears prior to the execution of a shell in the trace of a *login* program. As another example, when a process locks a file for exclusive access, we may want to make sure that the process removes the lock on the file before it exits in order to prevent the file from being locked permanently.

Therefore, we should be able to specify the sequence of operations a program is allowed to perform.

3.5.3 Synchronization

In a distributed system, a security failure can result from improper synchronization of programs. As illustrated in Section 2.5, concurrent access to the password file can leave the file inconsistent. It is necessary to specify the desirable interactions between programs that access shared files. In addition, synchronization problems occur in a parallel program, whose execution consists of multiple processes.

Therefore, the trace-policy language should be able to describe the traces of multiple programs that are synchronized in a desirable way. There are two main problems concerned with synchronization: Mutual exclusion and Precedence.

> **Mutual Exclusion**: In some cases, two processes cannot access a file in the system simultaneously. When a process is accessing a file, the other one has to wait until the former process finished.
>
> **Precedence**: There may be a precedence relation between operations of different processes. For example, process B should not read file C until process A finishes writing it.

3.5.4 Race Conditions

A race condition is a special case of the synchronization problem. If a program has a race-condition flaw, an attacker can affect the operations of the program by performing certain operations during the execution of the program. To monitor exploitations of a race condition, we need to monitor the operation sequence of the program during execution as well as all other relevant processes in the system.

We have identified four different characteristics of traces that are relevant to security. In the next chapter, we describe a language framework that is capable of describing these characteristics of traces.

Chapter 4

Specification Language

This chapter presents a language framework for specifying trace policies. We attempt to achieve, as much as possible, two conflicting goals: expressibility and detection efficiency. The language should be capable of describing all possible interesting trace policies, yet have an efficient algorithm for determining whether an execution trace satisfies a specified policy. In addition, compactness and readability of a policy specification are also considerations.

We use a grammar as the specification of a trace policy. A grammar defines a language, which is a set of sentences. In our problem domain, the set of alphabets is the set of operations, and a sentence is a sequence of operations. A trace policy for a program is a grammar, and an execution trace of a program satisfies the policy if it is a sentence of the language specified by the grammar. Using grammars as specifications has the advantage that formal languages are a mature discipline and many results can readily be applied.

From a grammatical point of view, describing valid traces of a program involves a parameterized and context-sensitive language. For example, the valid operation sequences of a program execution could be a function of the configuration of the system (e.g., the name of the mail spool directory or the name of the host) and the characteristics of the process (e.g., the owner or the process ID). We treat these characteristics as parameters of a trace policy. In addition, one common policy for a program is that it is allowed to change the permission mode of only the files it creates. That is, a *change permission mode operation on a file X* is allowed if a *create file X* operation occurs earlier in the program trace. Also, describing valid traces of a concurrent program whose execution consists of set of concurrent processes involves a language that describes the merge of the traces of these processes.

For programming languages, a context-free grammar typically defines the underlying language structure and additional rules impose the context-sensitive constraints. The formalization of such rules leads to a variety of extensions of context-free grammars, such as attribute grammars [7] and affix grammars [13]. Environment grammars [42] permit the parameterization of the language syntax and feature so-called environment variables which aid in parsing efficiency, parameterization, the linking of semantic and syntactic information and clarity of presentation. They enable the specification of a language in a compact form and have an efficient parsing algorithm in all practically important cases. Environment grammars have been employed in the translation and transfer of relational data in real time.

We developed a novel type of grammar, a parallel environment grammar, for specifying trace policies of programs. Based on environment grammars, a parallel environment grammar is able to specify traces of concurrent processes. We define parallel environment grammars in Section 4.2 and discuss their properties in Section 4.3. Then, we present a parallel parsing method that is efficient for a restricted class of languages in Section 4.4. Last, we describe how parallel environment grammars are used to specify trace policies of programs and illustrate the use of parallel environment grammars by an example in Section 4.5.

4.1 Notation

This section defines the terminology used throughout this chapter.

- $\emptyset$ and ε denote the empty set and the empty string, respectively.
- [and] bracket optional syntactic quantities.
- A^* denotes the set of strings whose components are elements of the set A.
- A^+ denotes $A^* - \{\varepsilon\}$.
- $C \backslash B$ $(C \in A^*, B \subset A)$ denotes string C with all elements of B removed.
- $T_1 \oplus T_2$, denotes the ordered merge of the two traces T_1 and T_2 as defined in Chapter 3.

4.2 Definition of Parallel Environment Grammars

The concepts on which parallel environment grammars are based are closely related to those of the environment grammars described in [42].

Definition 4.1 A parallel environment grammar (PE-grammar) is an ordered 6-tuple

$$(E,\ P,\ T,\ I_E,\ R,\ S)$$

where

- E is the set of environment variables, whose ordered set of values is called the environment e. Each environment variable in E is either *global* or *local*. E_G (E_L) denotes the set of environment variables E that are global (local), $E = E_L \cup E_G$ and $E_L \cap E_G = \emptyset$.

- P is a finite set of protovariables, where $E \cap P = \emptyset$.

- T is a finite set of terminals or tokens, on which the occurrence time function C is defined.

- I_E is a finite set of initial environment assignments of the form $R_X \quad X = Y$, where $X \in \mathcal{E}$, $Y \in R_X$, and R_X is the value range of X (I_E results in the initial composite environment e_0).

- R is a finite set of hyperrules of the form

  $$X_0 \ \rightarrow\ Z\,[B] \quad \text{or} \quad X_0 \ \rightarrow\ X_1\ X_2\ \cdots\ X_m\ [B]$$

  where $X_0 \in N$, $Z \in T$, $X_i \in N$ for $1 \leq i \leq m$, $m \geq 0$,

  $$N = \{< x_1, x_2, \cdots, x_n > \ |\ x_j \in (E \cup P^+),\ 1 \leq j \leq n,\ n \geq 1\}$$

  is the finite set of hypernotion, $B \subset A$ is an optional set of attached actions, $A = A_S \cup A_E$, A_S is the set of semantic actions, A_E is the set of environment assignments of the form $Y = Z$ where $Y \in E$ and Z is an expression formed from elements of E, P, and any auxiliary variables maintained for semantic actions.

- S is the parallel start expression of the form

$$s_1 \parallel s_2 \parallel \cdots \parallel s_n,$$

where $n \geq 1$, and $s_i \in \{< x > \ | \ x \in (E \cup P^+)\}$ is called a start notion.

The sets of environment variables E, protovariables P, terminals T, initial environment assignments I_E, and hyperrules R above are analogous to those of environment grammars [42]. However, in a parallel environment grammar, there are no metavariables and the start expression which consists of one or more start notions replaces the start notion in an environment grammar.

The parallel aspect of a parallel environment grammar arises from the start expression, which is a parallel expression of one or more start notions. Each start notion s_i together with the other 5 elements of G actually defines an environment grammar, referred to as *sub-grammar* G_i. Therefore, a parallel environment grammar consists of one or more sub-grammars. Intuitively, a sentence of a parallel environment grammar is the merge of the sentences generated by each sub-grammar. In a parallel derivation, each s_i derives a sentence x_i in sub-grammar G_i, and the sentence derived from the start expression is the *ordered merge* $x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

The environment variables E serve to parameterize the grammar. Their values are initialized by the initial environment assignments I_E, and may be changed by environment assignment A_E attached to a rule. Semantic actions may also be attached to a hyperrule. The set E is divided into the set of global environment variables E_G and the set environment variable E_L which are local to a sub-grammar. This concept is analogous to the concept of global and local variables in programming languages. A sub-grammar has its own copy of the local environment variables E_L and shares the same set of global environment variables E_G with all other sub-grammars. Therefore, the environment seen by two sub-grammars can be different. From a global perspective, the composite environment is the ordered set of values of the local environment variables of all sub-grammars and the values of the global environment variables. The initial environment assignment I_E initializes the local environment variables by initializing the values of all copies of the local environment variables.

Definition 4.2 Given a PE-grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$ with a hyperrule r_H in R,

$$X_0 \;\rightarrow\; Z\,[B] \quad \text{or} \quad X_0 \;\rightarrow\; X_1\; X_2\; \cdots\; X_m\,[B]$$

which contains the environment variable $V_h \in E(1 \leq h \leq l)$, the production rule $r_p(r_H, e)$ corresponding to the hyperrule r_H in the current environment e is

$$X_0 \;\rightarrow\; Z\,[B] \quad \text{or} \quad Y_0 \;\rightarrow\; Y_1\; Y_2\; \cdots\; Y_m\,[B]$$

where $Y_i(0 \leq i \leq m)$ are obtained from X_i by the following *uniform replacement*: each occurrence of the environment variable V_h in X_i is replaced by the current value of V_h. Hence, $Y_0 \in F$, $Y_i \in (T \cup F)$, $1 \leq i \leq m$, where

$$F = \{< y_1, y_2, \cdots, y_n > \;|\; y_j \in P^*,\; 1 \leq j \leq n,\; n \geq 1\}$$

is called the set of protonotions.

Each hyperrule $r_H \in R$ gives rise to a production rule $r_p(r_H, e)$ that depends on the environment e. Therefore, the set of production rules for the current environment e is $R_p(e) = \{r_p(r_H, e) \mid r_H \in R\}$.

A hyperrule serves as a template for the replacement which causes the left-hand side of the resulting production rule to be composed of a single protonotion and the right-hand side of protonotions and terminals. Hence, production rules are equivalent to production rules of context-free grammars (protonotions are nonterminals) and their applications correspond to derivation steps in context-free grammars.

However, the set of production rules $R_p(e)$ is not constant; it varies with the environment e as a derivation progresses. A change in the environment occurs when an environment assignment in B is executed upon successful application of a production rule. Note that the uniform replacement does not affect the environment variables in B; they are evaluated when an environment assignment in B is executed (see below).

Definition 4.3 The language specified by a PE-grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$, $S = s_1 \parallel s_2 \parallel \cdots \parallel s_n$, is $L(G)$, which is the set

$$\{x_1 \oplus \cdots \oplus x_n \mid (s_1 \parallel \cdots \parallel s_n,\ e_0) \stackrel{p}{\Longrightarrow}^* (x_1 \parallel \cdots \parallel x_n,\ e_f), x_i \in T^*(1 \leq i \leq n)\},$$

where e_0 is the initial composite environment defined by the initial environment variable assignment I_E, e_f is the final composite environment, and the symbol $\overset{p}{\Longrightarrow}{}^*$ is the reflexive and transitive closure of the relation $\overset{p}{\Longrightarrow}$ defined below. The symbol $\overset{p}{\Longrightarrow}{}^*$ is called a parallel derivation as every start symbol derives a sentence of tokens, and $\overset{p}{\Longrightarrow}$ is called a parallel derivation step.

A parallel derivation step corresponds to an application of a production rule to one of the sentential form in the parallel expression. Exactly one sub-grammar is involved in a parallel derivation step. A parallel derivation step transforms a sentential form in the parallel expression into another and keeps the other sentential forms in the parallel expression the same. We define $\overset{p}{\Longrightarrow}$ from the perspective of a sub-grammar separately in a) and b) depending on whether the right-hand side of the applied production rule contains protonotions or only a terminal and environment assignments. In the latter case, optional environment assignments may have to be executed.

a) Definition of a derivation step without environment changes:

> $(X, e) \overset{p}{\Longrightarrow} (X', e)$ if and only if there exist $P \in T^*$, $Q \in (F \cup T \cup A)^*$, and $W' \notin (T \cup A)^*$ such that $X = PWQ$, $X' = PW'Q$, and $(W \to W') \in R_p(e)$

b) Definition of a derivation step with possible environment changes:

> $(X, e) \overset{p}{\Longrightarrow} (X', e')$ if and only if there exist $P \in T^*$, $Q \in (F \cup T \cup A)^*$, $W' \in (T \cup A)^*$, $U \in (T \cup A)^*$, and $V \in (F \cup \{\text{right end marker of } X\})$ such that $X = PWUVQ$, $X' = P(W' \backslash A)(U \backslash A)VQ$, $(W \to W') \in R_p(e)$, $e' = (W'U \backslash T)(e)$, i.e., e' is obtained by starting with environment e and executing the environment assignments in U in left-to-right order. Note that the environment assignments change both the global part of the composite environment and the part that is local to the sub-grammar.

From the perspective of a sub-grammar, a (leftmost) derivation step transforms one sentential form (X) of protonotions, terminals, environment assignments, and semantic actions into another (X'). Environment assignments and semantic actions that are attached to a production rule are elements of the sentential form. They are discarded from a sentential form only after they have been executed. This happens as soon as the elements of the sentential form to the left of the environment assignment or semantic action consists solely of terminals. Also, the environments e and e' are the environments from the perspective of sub-grammar G_i before and after the derivation step, not the composite environments. A derivation step does not changes the local variables that are not private to the sub-grammar G_i. In our definition, the derivation steps in different sub-grammars are done sequentially. However, we will illustrate in Section 4.X that the parsing of the sentences of different sub-grammars can be done concurrently.

A parallel derivation can be thought of as n individual derivations. The derivation of x_i consists of a sequence of (leftmost) derivation steps $\overset{p}{\Longrightarrow} \cdots \overset{p}{\Longrightarrow}$, corresponding to s_i derives x_i in sub-grammar G_i $(1 \leq i \leq n)$. The parallel derivation is driven by the occurrence time of the tokens. That is, the time associated with the tokens generated in subsequent parallel derivation steps is greater than the time associated with the token generated in the current parallel derivation step.

We illustrate a parallel derivation using a very simple parallel environment grammar shown in Fig 4.1. The langauge defined by the PE-grammar is { *open_A close_A open_B close_B*, *open_B close_B open_A close_A* }.

```
    Environment Variables
1.  int E = 0;

    Start Expression
2.  <progA> || <progB>

    Hyperrules
3.  <progA> -> <writeA, E>.
4.  <writeA, 0> -> <openA> <closeA> { E = E - 1;}.
5.  <open> -> open_A { E = E + 1; }.
6.  <close> -> close_A.

7.  <progB> -> <writeB, E>.
8.  <writeB, 0> -> <openB> <closeB> { E = E - 1;}.
9.  <openB> -> open_B { E = E + 1; }.
10. <closeB> -> close_B.
```

Figure 4.1: An Example Parallel Environment Grammar

Table 4.1 depicts a parallel derivation of *open_A close_A open_B close_B*, which consists of 8 parallel derivation steps [1] (steps 1-8). Sentential forms labeled with †represent inter-

[1]The subscripts t_1 to t_4 denote the time associated with the tokens.

Step	Sub-grammar 1	Sub-grammar 2	E
	<progA>	<progB>	0
1. (I)	<writeA, 0>		0
2. (II)	<openA> <closeA> $\{E = E - 1\}$		0
3. (III)	†$open\_A_{t_1}$ $\{E = E + 1\}$ <closeA> $\{E...\}$		0
	$open\_A_{t_1}$ <closeA> $\{E = E - 1\}$		1
4. (IV)	$open\_A_{t_1}$ $close\_A_{t_2}$		0
5. (i)		<writeB, 0>	0
6. (ii)		<openB> <closeB> $\{E = E - 1\}$	0
7. (iii)		†$open\_B_{t_3}$ $\{E = E + 1\}$ <closeB> $\{E...\}$	0
		$open\_B_{t_3}$ <closeB> $\{E = E - 1\}$	1
8. (iv)		$open\_B_{t_3}$ $close\_B_{t_4}$	0
Result: open_A close_A open_B close_B			

Table 4.1: An Example Parallel Derivation

mediate results of derivation steps that involve executions of environment assignments or semantic actions. The parallel derivation can be thought of as two derivations, corresponding to the derivation of *open_A close_A* in sub-grammar 1 (steps I-IV) and the derivation of *open_B close_B* in sub-grammar 2 (steps i-iv). In derivation step 1, the production rule $<progA> \rightarrow <writeA, 0>$, which is obtained from hyperrule 3 at the initial environment ($E = 0$), is applied to the first sentential form, changing it to $< writeA, 0 >$. In step 3, $< open\_A >$ changes to *open_A*, and the environment assignment changes E to 1. Note that the parallel derivation is driven by the occurrence time of the tokens. The first few derivation steps expand the first sentential form until the first token *open_A* which has the earilest time is generated. The next two derivation steps also expand the first sentential form as the second token is *close_B*. Also, *open_A open_B close_A close_B* is not a sentence of the grammar because after step III, $E = 1$, progB derives $< write, 1 >$, which does not match any left-hand sides of the production rules.

4.3 Properties of Parallel Environment Grammars

For a grammar to be useful in practice, parsing must be unambiguous and efficient. In this section, we discuss several properties of parallel environment grammars that are related to ambiguity and parsing efficiency. Many of the properties of environment grammars described in [42] also apply to parallel environment grammars.

Definition 4.4 A PE-grammar G is called ambiguous if for some $x \in L(G)$ there is more than one parallel derivation of x from S in G. Otherwise, G is called unambiguous.

The languages arising in practical applications tend to be unambiguous. Given an ambiguous grammar of such a language, techniques such as left-factoring or precedence grouping may often be applied to disambiguate it.

Definition 4.5 In a PE-grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$ a pair of distinct hyperrules r_H and r'_H with left-hand sides X_0 and X'_0, respectively, $r_H \neq r'_H$ and $r_H,\ r'_H \in R_H$, is said to be *left-disjoint* if the following two constraints hold.

(a) $A \neq C$ if $X_0 \neq X'_0,\ (A \rightarrow B) = R_p(r_H, e),$ and $(C \rightarrow D) = R_p(r'_H, e)$;

(b) $B \neq D$ if $X_0 = X'_0,\ (A \rightarrow B) = R_p(r_H, e),$ and $(C \rightarrow D) = R_p(r'_H, e)$;

A PE-grammar $G = (\mathcal{E},\ \mathcal{P},\ \mathcal{T},\ \mathcal{I}_{\mathcal{E}},\ \mathcal{R},\ \mathcal{S})$ is said to be *left-disjoint* if all its hyperrules are pairwise left-disjoint.

In a left-disjoint PE-grammar no two production rules have the same left-hand side unless they have been obtained from two hyperrules with the same left-hand sides. In the latter case, the right-hand sides of the two production rules differ. Thus, $R_p(r_H, e) \neq R_p(r'_H, e)$, i.e., each production rule can be obtained from only one hyperrule. This property is useful not only for parsing but also when environment assignments or semantic actions are attached to a hyperrule.

Definition 4.6 A hyperrule r'_H with the left-hand side hypernotion $< x'_1, x'_2, \cdots, x'_n >$ is called a *reference* of a protonotion $< y_1, y_2, \cdots, y_n >$ on the right-hand side of a production rule $r_p(r_H, e)$ if the uniform replacement for the hypernotion $< x'_1, x'_2, \cdots, x'_n >$ is $< y_1, y_2, \cdots, y_n >$.

In a derivation step, the right-hand side of the applied production rule contains terminals and protonotions. The references of the latter specify the hyperrules from which the production rules of subsequent derivations steps will be obtained.

Lemma 4.7 In a left-disjoint PE-grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$ all references of a protonotion $< y_1, y_2, \cdots, y_n >$ have the same hypernotion on the left-hand side, and this hypernotion can be uniquely identified.

Proof: It follows from left-disjointness that two or more references of the same protonotion have the same left-hand side. Now consider only those hyperrules whose left-hand sides

$< x'_1, x'_2, \cdots, x'_n >$ consist of exactly n elements; the others cannot generate a production rule whose left-hand side equals the given protonotion. The first hyperrule with a left-hand side for which the uniform replacement is $< y_1, y_2, \cdots y_n >$ determines the unique left-hand side of the reference. If no such hyperrule exists, the protonotion does not have a reference.

Definition 4.8 In a parallel environment grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$, $S = s_1 \parallel s_2 \parallel \cdots \parallel s_n$, G is said to be *token-disjoint* if and only if $\exists T_1, T_2, \cdots, T_n \subset T$ such that $T = T_1 \cup T_2 \cup \cdots \cup T_n$, $T_i \cap T_j = \emptyset\ \forall i \neq j$, and any tokens generated from s_i must belong to T_i

A sentence of a token-disjoint PE-grammar thus can be split into n sentences based on the sets $T_1, T_2, \cdots, T_n$. The token-disjoint property of a PE-grammar is important for parsing the generated language efficiently and unambiguously.

Definition 4.9 In a parallel environment grammar $G = (E,\ P,\ T,\ I_E,\ R,\ S)$, a hypernotion $X =< x_1,\ x_2,\ \cdots,\ x_n >$ is said to be *local* if and only if it does not contain any global environment variables, $\forall i\ x_i \notin \mathcal{E}_{\}}$.

A hyperrule $X_0 \to X_1 X_2 \cdots X_m[B]$, where X_0 is a hypernotion, X_i $(1 \leq i \leq m)$ can be a hypernotion or a terminal, is said to be *local* if and only if each hypernotion X_i in the hyperrule is local. For a local hyperrule r_H, the production rule $R_p(r_H, e)$ is independent of the global environment, i.e. $R_p(r, e) = R_p(r_H, e')$ if and only if the local portions of the environments e and e' are the same.

A block of environment assignments B is said to be *local* if and only if it does not contain any global environment variables. That is, execution of B does not refer to or modify global environment variables. For a local block of environment assignments B, the local portion of any environment e is equal to the local portion of B(e). Also, if the local portions of e and (e') are the same, so are the local portions of $B(e)$ and $B(e')$.

4.4 Parallel Hyperparsers

This section describes a parsing method for recognizing sentences generated from a parallel environment grammar. A parallel hyperparser Hp is a top-down parser for recognizing sentences of $L(G)$ where $G = (E,\ P,\ T,\ I_E,\ R,\ S)$, is a left-disjoint, token-disjoint, and unambiguous PE-grammar.

Figure 4.2 depicts the structure of a parallel hyperparser. A parallel hyperparser consists of a hyperdispatcher, several sequential hyperparsers(or hyperparser), a set E_G of global environment variables shared by all hyperparsers, and an initialization procedure for performing the initial environment assignments to the global environment variables. Each hyperparser is a top-down parser, and is very similar to the hyperparser described in [42].

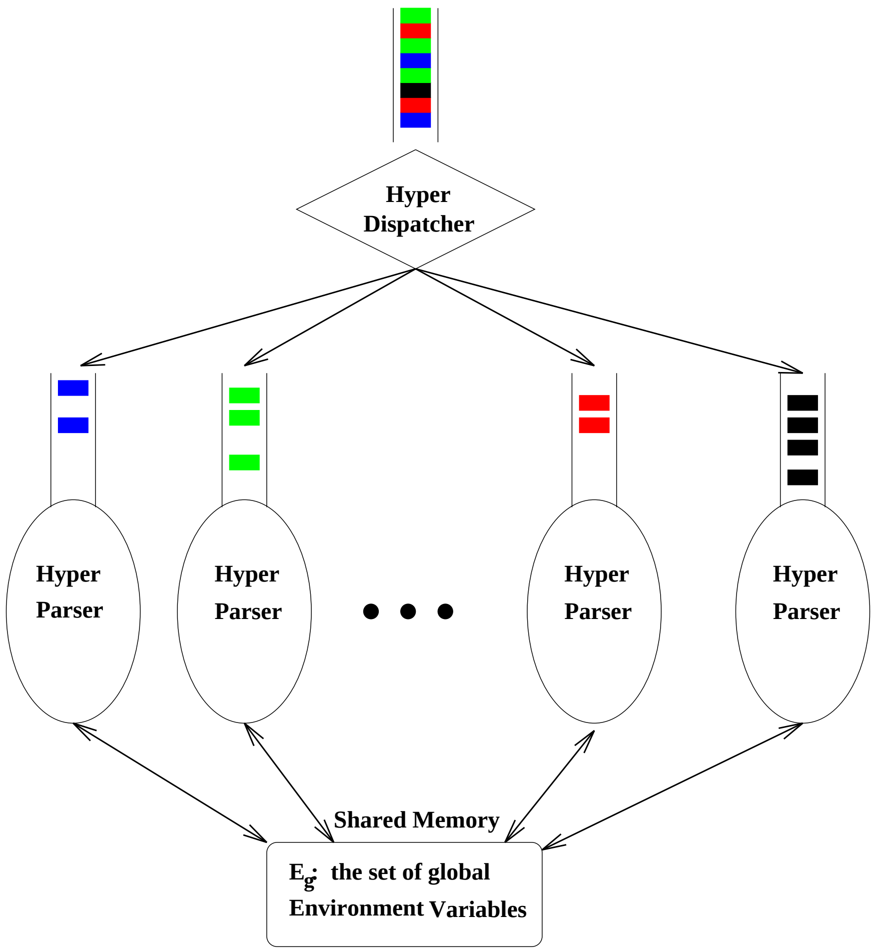

Figure 4.2: A Parallel Hyperparser

4.4.1 Hyperparsers

There is a hyperparser associated with each s_i in S. Each hyperparser contains a set E_L of local environment variables private to the parser, a local initialization procedure for performing the initial environment assignments on the local environment variables, a set of lexical procedures for reading and recognizing the terminals T, and a set of (possibly recursive) hyperprocedures for generating and applying production rules. There is one hyperprocedure per hyperrule, where a hyperrule may contain alternative right-hand sides. Also, each thread hyperparser has a local variable *current_token* which holds the token it is processing.

Hyperprocedures

The hyperprocedures form the core of a hyperparser. Let $HP\_X_0()$ be the hyperprocedure corresponding to the hyperrule $X_0 \rightarrow X_1X_2 \cdots X_m[B]$. When called, it first inspects the next token in the input queue and sets *current_token* to that token. If the input queue is empty, it waits until the next token arrives. Then, it uses the environment variables to generate the current production rule by obtaining the protonotions (or terminals) Y_i from the hypernotions (or terminals) X_i on the right-hand side of the hyperrule. The hyperprocedures for the references of the protonotions Y_i (or lexical procedures in case the latter are terminals) are first identified and subsequently called in sequence. If all m procedures return successfully, any attached environment assignments or semantic actions B are performed and the hyperprocedure returns indicating success.

If $HP\_X_0()$ represents a set of hyperrules, the order in which they are processed is determined by consulting the next input token. Processing is as above, but a failure indication by a called procedure causes the thread hyperparser to backtrack and start the processing of the next alternative rather than to immediately return with a failure indication.

To parse a given sentence, the hyperdispatcher first performs the global initialization procedures to initialize the global environment variables, and starts all thread hyperparsers. It then reads the tokens one by one and dispatches them into the input queue of the appropriate hyperparsers. The hyperdispatcher does not wait until the hyperparser accepts the token before it dispatches the next token. Therefore, the input queue of a thread hyperparser could consist of several tokens.

When a hyperparser starts, it first performs the local initialization procedure to initialize its local environment variables. It then calls the hyperprocedure $HP\_s_i()$ for the hyperrule with the start symbol s_i on the left hand side. The execution of each hyperparser is driven by the input token. A thread hyperparser either pauses in a lexical procedure or at the beginning of a hyperprocedure when the next token is not available, i.e., the input queue is empty. For example, the hyperprocedure $HP\_S_i()$ stops if the next input token is not available when the hyperparser starts.

The hyperparsers run in parallel to accept the sequence of tokens placed in their input queues. The purpose of a hyperparser is to parse the sequence of tokens in its input queue and report any errors. When the hyperdispatcher finishes reading and dispatching tokens to the hyperparsers and each hyperparser parses the sequence of tokens in its input queue without any errors, the parsing is considered to be successful, i.e., the sentence is recognized by the parallel hyperparser. Otherwise, the input sentence is not a sentence of the PE-grammar.

4.4.2 Synchronization among Hyperparsers

In parsing an input sentence, the sentence is split into n disjoint sentences, each of which is fed into one of the hyperparsers. Nevertheless, the parsing operations of the hyperparsers are not totally independent. A thread hyperparser occasionally synchronizes with other thread hyperparsers. We describe the synchronization conditions and the additional procedures for handling synchronization in this subsection.

Definition 4.10 The execution of a thread hyperparser can be divided into *execution steps*. Each execution step corresponds to the generation of a production rule from a hyperrule and application of the rule or to an execution of a block of environment assignments. An execution step is *local* if the hyperrule is local or the block of environment assignments is local. Otherwise, the execution step is *nonlocal*.

In general, when a thread hyperparser needs to read from or write to global environment variables, it synchronizes with other thread hyperparsers. Precisely, the synchronization condition (SC) is

> When a thread hyperparser executes a nonlocal execution step in processing a token x, it cannot start until all execution steps processing with tokens that have

a time earlier than that of x have finished.

A *waittoken(t)* procedure is used for synchronization, where t is the timestamp of *current_token*. The procedure returns when the timestamp of the invoking procedure is earlier than the timestamps of *current_token* of all other hyperparsers. At that time, all tokens preceeding *current_token* have been processed.

There are two situations where *waittoken()* will be called: In generally, *waittoken()* is called when the hyperparser needs to access (read or write) global environment variables. Specifically,

- In a hyperprocedure $HF\_X_0()$ that corresponds to a nonlocal hyperrule $X_0 \rightarrow X_1X_2\cdots X_m[B]$, the hyperprocedure calls $waittoken(t)$ before it uses the environment variables to generate the protonotions and terminals of the production rule.

- When a hyperprocedure is about to perform an action B, it calls $waittoken(t)$ before it performs the environment assignments in B if B is not local, i.e., if the environment assignments in B have references to global environment variables.

Definition 4.11 A strongly synchronized parallel hyperparser Hp' is a parallel hyperparser in which the thread hyperparsers synchronize themselves for every token. A strongly synchronized parallel hyperparser is exactly the same as a parallel hyperparser except the token dispatcher passes a token from the input into a hyperparser and waits until the hyperparser finished processing the token before it feeds the next token to another hyperparser. Although it consists of multiple hyperparsers running in parallel, it can be considered as a sequential machine. It is because at any single time, only one hyperparser is in action, all the others are blocked waiting for the next token.

Figure 4.3 compares the execution of a strongly synchronized parallel hyperparser and a parallel hyperparser consisting of 4 thread hyperparsers. The execution steps in a strongly synchronized parallel hyperparser occur in sequence while the execution steps in a parallel hyperparser could overlap and occur in a different order. In the example in (Fig 4.3, execution step s_2 starts before execution step s_1 completes. Nevertheless, no two execution steps that are both nonlocal overlap. It is because the two thread hyperparserss executing the two steps will both call $waittoken()$ and one of them has to wait for the other to finish before it can proceed. In general, we can name the execution steps in a strongly synchronized parallel hyperparser by their order of occurrences as $s_1, s_2, \cdots, s_n$.

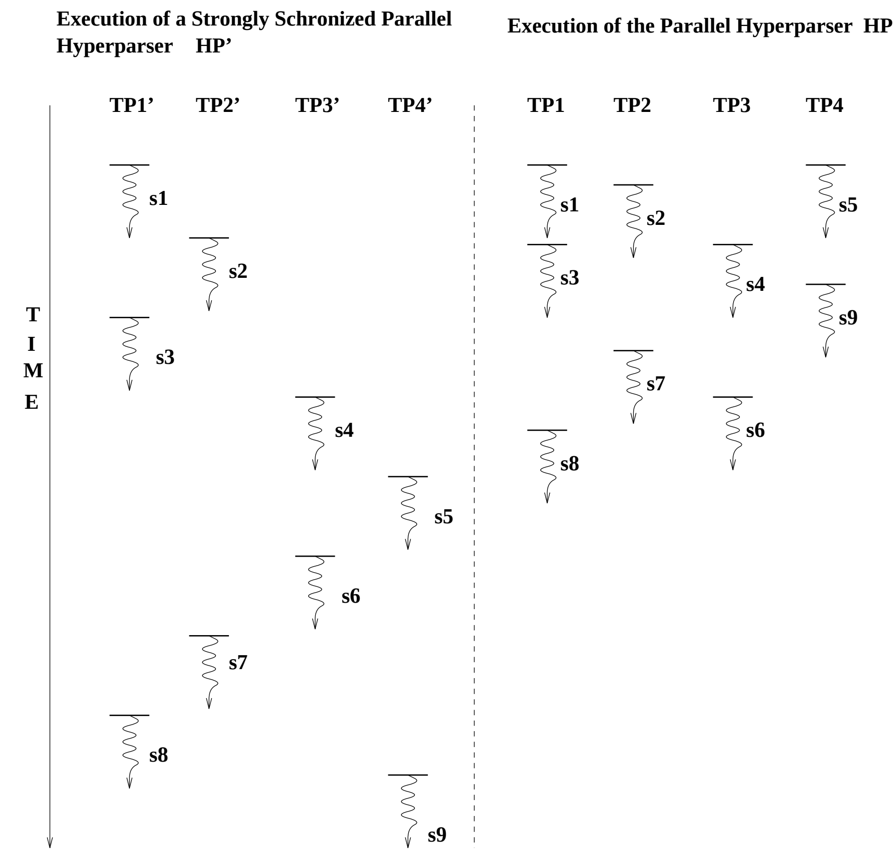

Figure 4.3: Execution of Two Hyperparsers

Lemma 4.12 A strongly synchronized parallel hyperparser recognizes sentences of $L(G)$ where $G = (E,\ P,\ T,\ I_E,\ R,\ S)$, $S = s_1 \parallel s_2 \parallel \cdots \parallel s_n$, is a left-disjoint, token-disjoint and unambiguous PE-grammar.

Proof: A strongly synchronized parallel hyperparser mirrors a parallel derivation which is unique since G is unambiguous.

The token-disjoint property assures that a token is fed into the input queue of the right thread hyperparser, i.e., if s_i derives x_i, then x_i is fed into the input queue of thread hyperparser i. Therefore, each thread hyperparser is given the right sentence to parse.

Lemma 4.7 (the property of left-disjointness) assures that the parse proceeds exactly as the derivation in each thread hyperparser. Since the input is a trace, tokens are fed one by one in increasing time. Thus, the parsing is driven by the occurrence time of the tokens as in the parallel derivation. The thread hyperparsers simulate a parallel derivation which is driven by the occurrence time of the tokens. Therefore a strongly synchronized parallel hyperparser accepts the language generated by the parallel environment grammar G. □

Theorem 4.13 A parallel hyperparser recognizes sentences of $L(G)$, where $G = (E,\ P,\ T,\ I_E,\ R,\ S)$ is a left-disjoint, token-disjoint and unambiguous PE-grammar.

We prove the theorem by showing that a strongly synchronized parallel hyperparser Hp' and a parallel hyperparser Hp accept the same language. By Lemma 4.11, the theorem follows. Before we present the proof, we first describe the symbols used.

- $p_1, p_2, \cdots, p_m$ denote the execution steps occurring in the execution of Hp and Hp'. The steps are named according to their occurrence order in Hp'.
- Tp_m and Tp'_m denote the m^{th} thread hyperparser of Hp and Hp'.
- $Q_{m,i}$ ($Q'_{m,i}$) denotes the local state of Tp_m (Tp'_m) at the time it just finished execution step p_i and Δ (Δ') denotes the global state of Hp (Hp') at the time p_i is finished.

If Tp_i and Tp'_i both perform an execution step at the same starting local state, they behave the same and end in the same local state. Also, the execution step does not affect the global state (4.1). In addition, when a thread hyperparser performs a nonlocal execution step at a state (Q, Δ), where Q is the local state of the thread hyperparserand Δ is the global state, if there is no other global execution steps overlapping with the step, it always behaves the same and transits to the same new state (4.2).

Proof: We want to prove that given an input sentence X, each thread hyperparser Tp_i in Hp performs the same execution steps as the corresponding thread hyperparser Tp'_i in Hp'. First, the sequences of tokens being dispatched to the input queue Tp_i and Tp'_i are the same for all i. The proof is by induction on the number of execution steps started in Hp using the following induction assumption:

Let p_{l_i} by the i^{th} execution step starts in Hp. (Note that $l_i = i$ for Hp', but it is not necessarily true for Hp). For any $(1 \leq i \leq k)$, let p_{l_i} is performed by Tp_m; when s_{i_i} is completed, we have

- Tp_m and Tp'_m behavior the same in this execution step.

- $Q_{m,l_i} = Q'_{m,l_i}$, and
- $\Delta_{l_i} = \Delta_{l_i}$ if p_{l_i} is not a local step.

Base case: (n = 1) Let the first execution step in Hp, p_{l_1} and finishes at t_1. If $i_1 = 1$, then only local execution steps can start before t_2, hence by (4.2), the induction assumption holds. If $i_1 \neq 1$, then p_{l_1} must be a local step, otherwise p_{l_1} cannot start. By (4.1), the induction assumption also holds.

Induction Step: Assume the induction assumption holds when $k = n - 1$. Let pl_n be the n^{th} execution starts in Hp, and is performed by thread hyperparser m and finishes at t_{l_n}.

Case 1: p_{l_n} is a local step. Let the last step performed by Tp_m be p_i, by the induction assumption, we have $Q_{m,i} = Q'_{m,i}$. That is, the local state of Tp_m at the time Tp_m starts p_{l_n} is equal to the local state of Tp'_m at the time Tp'_m starts p_{l_n}. Since p_{l_n} is a local step, by (4.1), the induction assumption holds for $k = n$.

Case 2: p_{l_n} is a not a local step. Let the last step performed by Tp_m be p_i, by the induction assumption, we have $Q_{m,i} = Q'_{m,i}$ - (a). In addition, the steps $s_1, s_2, \cdots s_{i_n - 1}$ must have been completed (because of the synchronization constraints). Let p_j be the last step in $s_1, s_2, \cdots s_{i_k - 1}$ which is nonlocal. We have $\Delta_j = \Delta'_j$ by the induction assumption. Also $\Delta'_{l_n} = \Delta'_j$ because of the definition of p_j. As $s_j, \cdots, s_{i_k - 1}$ are local steps, and any step that start between t_j and t_{l_n} must be local steps, $\Delta_{i_k - 1} = \Delta_j = \Delta'_j = \Delta'_{l_n}$ - (b). Therefore, by (a), (b) and (4.1), the induction assumption hold for $n = k$.

4.4.3 Tokens

The tokens of a PE-grammar may be audit records and are defined in the lexical description. An audit record describes an occurrence of a system event (or an audit event), which corresponds to the execution of a system operation (e.g., a system call). An audit record consists of an event type field denoting the type of the event and other data fields describing other information pertained to the event, such as information about the subject that performed the action, information about the objects being accessed , the occurrence time, and other relevant information. The format of an audit record varies from system to system. In general, we model the set of audit records as a set of tuples

$$\{(f_0,\ f_1,\ f_2,\ f_3,\ \cdots,\ f_N)\}$$

where

- f_0 is the audit event type, $f \in Ev$, the set of audit events
- $f_i(1 \leq i \leq N)$ are the values of the i^{th} data field of the audit record, $f_i \in R_i$, the range of the i^{th} data field. Each data field has a name N_i.

4.4.4 Lexical Procedures

A token is recognized by a lexical procedure which may consider only a subset of event attributes. A lexical procedure is constructed based on a token definition. A token definition has the form

$$name_1\text{-}name_2\text{-}\cdots\text{-}name_m \quad : \quad [\sim] \quad (T_1 \mid T_2 \mid \cdots \mid T_n)$$

where the right-hand side symbol is called a token-name template, $name_i$ could be a name or a variable W_i, and T_i is token expression of the form

$$(t, e_1, e_2, \cdots, e_n), 0 \leq n,$$

$t \in (E_v \cup *)$, e_i, $1 \leq i \leq n$, is field expression of the form $a = expr$, $a \neq expr$, $a =\sim expr$, $a\ !\sim expr$, $a \in expr$, $a \notin expr$, $a \in V_t$, $expr$ is an expression formed from W_i, $1 \leq i \leq m$, N_i, $1 \leq i \leq N$.

In a lexical description, the first name in each token-name template must be different. In this way, it is obvious to identify from a given token name, the token-name template (the reference) that defines the meaning of the token name. Given a token name str-y_1-$\cdots$-y_2, which reference is the token-name template str-X_1-X_2-$\cdots$-X_n. The set of tokens represented by the token name is

$$\begin{array}{lcl} \{\ (t, v_1, v_2, \cdots, v_k) \mid \exists i(1 \leq i \leq m)\ s.t.\ T_i' = true\} & : & \text{if the definition does not start with } \sim \\ \{\ (t, v_1, v_2, \cdots, v_k) \mid \forall i(1 \leq i \leq m)\ s.t.\ T_i' = false\} & : & \text{if the definition starts with } \sim \end{array}$$

Shown below are the definitions of several token-name templates.

```
open_r_passwd : [(open_r, path == ``/etc/passwd´´)].
not_open_r_passwd : ~[(open_r, path == ``/etc/passwd´´)].
open_r-P1 : [(open_r, path == ``P1´´)].
```

The first template `open_r_passwd` refers to all audit records describing an *open_r* event on a path equal to */etc/passwd*. The second template refers to all audit records that do not describe an *open_r* event on a path equal to */etc/passwd*. It refers to all other audit records that are not referred to by the first template. The last template consists of a variable $P1$. The tokens it refers to depend on the value of the variable. Basically, it refers to all audit records with an *open_r* event on a path equal to the value of $P1$.

Let TP_tk be the lexical procedure corresponding to the token-name template *tk*-X_1-X_2-$\cdots$-X_m, whose definition is $(T_1 \mid T_2 \mid \cdots \mid T_n)$. To recognize a token of a particular token type *tk*-y_1-y_2-$\cdots$-y_m, a hyperparser calls a lexical procedure TP_tk with actual parameters $y_1, y_2, \cdots, y_n$. When called with actual parameters $y_1, y_2, \cdots, y_n$, the lexical procedure LP_tk evaluates the first token expression T_i with the current token. To evaluate a token expression, it substitutes the field names with the values in the current token, and evaluates the resulting truth value for each field expression. If the event matches the event of the current audit record, and the values of all field expressions $expr'_i$, $1 \leq i \leq n$ are *true*, it returns success. Otherwise, it evaluates the next alternative token expression and so on. If all token expressions are evaluated to *false*, it returns with a failure indication. In case the definition is of the form $\sim (T_1 \mid T_2 \mid \cdots \mid T_n)$, the lexical procedure evaluates the truth value of every token expression, and returns success if all of them are *false*. Otherwise, it returns with a failure indication.

4.5 Illustration of the Use of PE-grammars for Specifying Trace Policies

In this section we illustrate the use of PE-grammars for specifying the valid execution traces of programs. We give an example of a PE-grammar that describes the valid execution trace of two programs. In addition to the notation introduced above, we shall adhere to the following conventions in our examples of parallel environment grammars.

- CAPITAL LETTERS are used for environment variables
- Small letters are used for token types or protovariables.
- $< x_1\ x_2\ \cdots\ x_n >$ will be written for hypernotions instead of $< x_1, x_2, \cdots, x_n >$, i.e., significant blanks are used to separate elements instead of commas.
- | denotes alternatives in hyperrules.
- { } is used to enclose environment assignments or semantic actions.
- ; is used to terminate initial environment assignments.
- . is used to terminate hyperrules.

The illustrative PE-grammar describes the valid execution traces of two Unix programs, program A and program B. Both programs modify the password file during execution. They perform the following sequence of operations during their executions. Note that the actual *read* and *write* operations are omitted.

Steps of Program A

1. *open_r*(*file*1)
2. *close*(*file*1)
3. *open_r*(*passwd*)
4. *close*(*passwd*)
5. *open_w*(*passwd*)
6. *close*(*passwd*)

Steps of Program B

1. *open_r*(*file*1)
2. *close*(*file*1)
3. *open_r*(*passwd*)
4. *close*(*passwd*)
5. *open_w*(*ptmp*)
6. *close*(*ptmp*)
7. *rename*(*passwd*, *ptmp*)

Program A first reads `file1` in steps 1-2 and then modifies the password file in steps 3-6. Program B first reads `file1` in steps 1-2 and then modifies the password file in steps 3-7. When the two programs execute simultaneously, they must not modify the password file at the same time. Therefore, the synchronization constraint is that steps 3-6 of program A cannot overlap with steps 3-7 of program B.

Figure 4.4 shows a PE-grammar that defines the valid operation sequence of the execution of the two programs with respect to the stated synchronization constraint. The start expression on line 2 consists of two start notions <progA> and <progB>; <prog A>

describes the operations performed by program A and <progB> describes the operations performed by program B.

```
    Environment Variables
1.  LOCAL int PID = getpid();
2.  ENV int CS = 0;

    Start Expression
3.  <progA> || <progB>

4.  <progA>  -> <init> <modify >
5.  <init>   -> <open_r file1> <close file1>
6.  <modify> -> <open_r passwd>
                {   if CS != 0 then violation();
                    CS = CS + 1;                 }
                <close passwd>
                <open_w passwd>
                <close passwd>
                {   CS = CS - 1;  }

7.   <open_r file1> -> open_r_file1-PID.
8.   <close file1>  -> close_file1-PID.
9.   <open_r passwd> -> open_r_passwd-PID.
10.   <open_w passwd> -> open_w_passwd-PID.
11.  <close passwd> -> close_passwd-PID.

12.  <progB>  -> <look> <change>
13.  <look>   -> <open_r file1> <close file1>
14.  <change> -> <open_r passwd>
                {   if CS != 0 then violation();
                    CS = CS + 1;                 }
                <close passwd>
                <open_w ptmp> <close ptmp>
                <rename ptmp passwd>
                {   CS = CS - 1;  }

15.  <open_w ptmp> -> open_w_ptmp-PID.
16.  <close ptmp> -> close_ptmp-PID.
17.  <rename ptmp passwd> -> rename_ptmp_passwd-PID.
```

Figure 4.4: An Illustration of a PE-grammar

Lines 1-2 show the initial environment assignment. The local environment variable PID stores the process ID of the running program. The copy of PID local to <progA> (<progB>) is assigned to the process ID of the process running program A (B) during the local environment initialization. It gets the value from the function *getpid()*. CS stores the number of processes that are accessing the password file. It is initialized to 0 and is increased by 1 in the environment assignment attached to the hyperrules on lines 6 and 14.

This happens when program A or B opens the password file for reading, which is the first step it uses to modify the password file. When the hyperrules on lines 6 and 14 recognize the last hypernotions, `CS` is decreased by 1.

The hyperrules on lines 4-11 describe the execution of program A. The hyperrule on line 3 specifies the execution of program A as the concatenation of hypernotions <`init`> and <`modify`>, which describe the operations program A performs to read */home/file1* (steps 1 and 2) and the operations program A performs to modify the password file. The hyperrule on line 5 recognizes the operations in steps 1 and 2. The hyperrule on line 6 describes the operations performed by program A to modify the password file, which correspond to the operations in steps 3-6. The first operation is represented by the token `open_r_passwd-PID`. There are two actions after the first operation. The first one raises a violation if `CS` is not zero, i.e., another process is modifying the password file; the second action increases `CS` by 1, indicating the process is now modifing the password file. There is an environment assignment attached to the end of the hyperrule on line 6, which decreases `CS` by 1 after program A performed the last operation in modifying the password file. The hyperrules on lines 7-11 accept tokens `open_r_file1-PID`, `close_file1-PID`, `open_r_passwd-PID`, `open_w_passwd-PID`, and `close_passwd-PID`. The tokens they accept depend on the value of `PID`. For the first sub-grammar (the one associated with `progA`), `PID` contains the ID of the process executing program A. Therefore, the hyperrules accept the tokens corresponding to the operations of the process executing program A.

For the second sub-grammar, the hyperrules accept the tokens corresponding to the operations done by the process executing program B.

The hyperrules on lines 12-17 describe the execution of program B. The hyperrule on line 12 specifies the execution of program B as the operations for reading *file1*, which is <`look`> followed by the operations for modifying the password file, <`change`>. The hyperrule on line 13 describes the operations for reading *file1* (steps 1-2) . The hyperrule on line 14 describes the operations performed by program B to modify the password file. When program B performs the first operation (represented by <`open_r passwd`>) the semantic action raises a violation if `CS` is not zero and increases `CS` by 1. Therefore, the PE-grammar prohibits steps 3-6 of program A from overlapping with steps 3-7 of program B in their execution. Lastly, the environment assignment attached at the end of the hyperule on line

14 decreases `CS` by 1 after program B finished modifying the password file.

The tokens to be recognized by the lexical procedure are `open_r_file1-X`, `close_file1-X`, `open_r_passwd-X`, `open_w_passwd-X`, `close_passwd-X`, `open_w_ptmp-X`, `close_ptmp-X`, `rename_ptmp_passwd-X` for $1 \leq X \leq MaxPid$. Note that environment variables may be used in the formation of token names.

For illustration purposes, consider the two traces of the execution of two programs shown in Figure 4.5. A trace (Def. 3.1) is a sequence of audit records corresponding to the operations performed by the two programs during the execution. A simplified audit record may be denoted by a 4-tuple (*event*, *file*, *pid*, *time*), where *event* denotes the operation, *file* denotes the file involved in the operaton, *pid* denote the process ID, and *time* denote the occurrence time of the operation. In the first trace, program B does not start until program A finishes. Therefore, it satisfies the synchronization requirement. In the second trace, program A and program B modify the password file simultaneously, thus violating the synchronization requirement.

Trace 1	Trace 2
$(open\_r,\ /home/file1,\ 23,\ t_1)$	$(open\_r,\ /home/file1,\ 23,\ t_1)$
$(close,\ /home/file1,\ 23,\ t_2)$	$(close,\ /home/file1,\ 23,\ t_2)$
$(open\_r,\ /etc/passwd,\ 23,\ t_3)$	$(open_r,\ /home/file1,\ 40,\ t_3)$
$(close,\ /etc/passwd,\ 23,\ t_4)$	$(open\_r,\ /etc/passwd,\ 23,\ t_4)$
$(open\_w,\ /etc/passwd,\ 23,\ t_5)$	$(close,\ passwd,\ 23,\ t_5)$
$(close,\ /etc/passwd,\ 23,\ t_6)$	$(close,\ file1,\ 40,\ t_6)$
$(open\_r,\ /home/file1,\ 40,\ t_7)$	$(open\_w,\ passwd,\ 23,\ t_7)$
$(close,\ file1,\ 40,\ t_8)$	$(open\_r,\ passwd,\ 40,\ t_8)$
$(open\_r,\ passwd,\ 40,\ t_9)$	$(close,\ passwd,\ 40,\ t_9)$
$(close,\ passwd,\ 40,\ t_{10})$	$(open\_w,\ ptmp,\ 40,\ t_{10})$
$(open\_w,\ ptmp,\ 40,\ t_{11})$	$(close,\ ptmp,\ 40,\ t_{11})$
$(close,\ ptmp,\ 40,\ t_{12})$	$(close,\ passwd,\ 23,\ t_{12})$
$(rename,\ /etc/ptmp,\ /etc/passwd,\ 40,\ t_{13})$	$(rename,\ /etc/ptmp,\ /etc/passwd,\ 40,\ t_{13})$

Figure 4.5: Execution Traces of Program A and Program B.

Tables 4.2 and 4.3 show the parallel derivations of traces 1 and 2. Because of space limitation, we use the following abbreviations in the tables. *open_r*, *close*, *open_w*, and *rename* are written as R, C, W, and RN respectively; $/etc/passwd$, $/home/file1$, $/etc/ptmp$ are written as pw, $f1$, and $ptmp$ respectively, and an audit record $(X_1, X_2, \cdots, X_n)$ is written as $(X_1\ X_2\ \cdots X_n)$.

Each parallel derivation consists of two derivations, corresponding to the derivations starting from < *progA* > and < *progB* >. There is a step number associated with each step, indicating the order of the steps in the parallel derivation. The derivation of trace 1 is successful while the derivation of trace 2 is unsuccessful as the *violation()* function is executed in step 14.

Step	<progA> || <progB>	CS
	<progA>	0
1	<init> <modify>	0
2	<open_r file1 > <close file1> <modify>	0
3	(R f1 23 t_1) <close file1> <modify>	0
4	(R f1 23 t_1) (C f1 23 t_2) <modify>	0
5	(R f1 23 t_1) (C f1 23 t_2) <open_r passwd> { if CS ... } <close pw> <open_w pw> <close passwd> { CS ... }	0
6	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_Q) <close passwd> <open_w pw> <close pw> { CS ... }	1
7	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_3) (C pw 23 t_4) <open_w pw> <close pw> { CS ... }	1
8	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_3) (C pw 23 t_4) (W pw 23 t_5) <close pw> { CS ... }	1
9	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_3) (C pw 23 t_4) (W pw 23 t_5) (C pw 23 t_6)	0
	<progB>	0
10	<look> <change>	0
11	<open_r f1> <close f1> <change>	0
12	(R f1 40 t_7) <close f1> <change>	0
13	(R f1 40 t_7) (C f1 40 t_8) <change>	0
14	(R f1 40 t_7) (C f1 40 t_8) <open_r passwd> {if CS ...} <close passwd> ... <rename ptmp passwd> {CS ...}	0
15	(R f1 40 t_7) (C f1 40 t_8) (R pw 40 t_9) <close passwd> <open_w ptmp> ... <rename ptmp passwd> {CS ...}	1
16	(R f1 40 t_7) ... (R pw 40 t_9) (C pw 40 $t_1$0) <open_w ptmp> <close ptmp> <rename ptmp passwd> {CS ...}	1
17	(R f1 40 t_7) ... (R pw 40 t_9) (C pw 40 $t_1$0) (W ptmp 40 t_{11}) <close ptmp> <rename ptmp passwd> {CS ...}	1
18	(R f1 40 t_7) ... (R pw 40 t_9) (C pw 40 $t_1$0) (W ptmp 40 t_{11}) (C ptmp 40 t_{12}) <rename ptmp passwd> {CS ...}	1
19	(R f1 40 t_7) ... (R pw 40 t_9) (C pw 40 $t_1$0) (W ptmp 40 t_{11}) (C ptmp 40 t_{12}) (RN ptmp pw 40 t_{13})	0

Table 4.2: A Parallel Derivation of Trace 1

Step	<progA> || <progB>	CS
1 (A)	<init> <modify>	0
2 (A)	<open_r f1 > <close file1> <modify>	0
3 (A)	(R f1 23 t_1) <close file1> <modify>	0
4 (A)	(R f1 23 t_1) (C f1 23 t_2) <modify>	0
5 (B)	<look> <change>	0
6 (B)	<open_r f1> <close file1> <change>	0
7 (B)	(R f1 40 t_3) <close file1> <change>	0
8 (A)	(R f1 23 t_1) (C f1 23 t_2) <open_r passwd> { if CS ... } <close pw> <open_w pw> <close pw> { CS ... }	0
9 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) <close passwd> <open_w passwd> <close passwd> { CS ... }	1
10 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) <open_w passwd> <close passwd> { CS ... }	1
11 (B)	(R f1 40 t_3) (C f1 40 t_6) <change>	1
12 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) (W pw 23 t_7) <close passwd> { CS ... }	1
13 (B)	(R f1 40 t_3) (C f1 40 t_6) <open_r passwd> {if CS ...} <close passwd> ... <rename ptmp pw> {CS ...}	1
14 (B)	(R f1 40 t_3) (C ...) (R pw 40 t_8) <close passwd> <open_w ptmp> ... <rename ptmp passwd> {CS ...}	2 V!!!
15 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) <open_w ptmp> <close ptmp> <rename ptmp passwd> {CS ...}	2
16 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) (W ptmp 40 t_{10}) <close ptmp> <rename ptmp passwd> {CS ...}	2
17 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) (W ptmp 40 t_{10}) (C ptmp 40 t_{11}) <rename ptmp passwd> {CS ...}	2
18 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) (W pw 23 t_7) (C pw 23 t_{12})	1
19 (B)	(R f1 40 t_3) (C f1 40 t_6) (R pw 40 t_8) (C pw 40 t_9) (W ptmp 40 t_{10}) (C ptmp 40 t_{11}) (RN ptmp pw 40 t_{13})	0

Table 4.3: An Unsuccessful Parallel Derivation of Trace 2

In Table 4.2, steps 1-9 correspond to the derivation steps in sub-grammar 1, and steps 10-19 correspond to the derivation steps in sub-grammar 2. In Table 4.3, steps 1-4, 8-10, and 18 (those followed by an A) correspond to the derivation steps in sub-grammar 1, and steps 5-7, 11, 13-17, and 19 (those followed by a B) correspond to the derivation steps in sub-grammar 2. Table 4.3 shows the derivation steps in the order they occur in the parallel derivation. To aid visualizing the derivation in each sub-grammar, Table 4.4 depicts the derivation steps from the perspective of sub-grammars 1 & 2.

Step	<progA> || <progB>	CS
	<progA>	0
1 (A)	<init> <modify>	0
2 (A)	<open_r f1 > <close file1> <modify>	0
3 (A)	(R f1 23 t_1) <close file1> <modify>	0
4 (A)	(R f1 23 t_1) (C f1 23 t_2) <modify>	0
8 (A)	(R f1 23 t_1) (C f1 23 t_2) <open_r passwd> { if CS ... } <close pw> <open_w pw> <close pw> { CS ... }	0
9 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) <close passwd> <open_w passwd> <close passwd> { CS ... }	1
10 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) <open_w passwd> <close passwd> { CS ... }	1
12 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) (W pw 23 t_7) <close passwd> { CS ... }	1
18 (A)	(R f1 23 t_1) (C f1 23 t_2) (R pw 23 t_4) (C pw 23 t_5) (W pw 23 t_7) (C pw 23 t_{12})	1
	<progB>	0
5 (B)	<look> <change>	0
6 (B)	<open_r f1> <close file1> <change>	0
7 (B)	(R f1 40 t_3) <close file1> <change>	0
11 (B)	(R f1 40 t_3) (C f1 40 t_6) <change>	1
13 (B)	(R f1 40 t_3) (C f1 40 t_6) <open_r passwd> {if CS ...} <close passwd> ... <rename ptmp pw> {CS ...}	1
14 (B)	(R f1 40 t_3) (C ...) (R pw 40 t_8) <close passwd> <open_w ptmp> ... <rename ptmp passwd> {CS ...}	2 V!!!
15 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) <open_w ptmp> <close ptmp> <rename ptmp passwd> {CS ...}	2
16 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) (W ptmp 40 t_{10}) <close ptmp> <rename ptmp passwd> {CS ...}	2
17 (B)	(R f1 40 t_3) (C ...) (R ...) (C pw 40 t_9) (W ptmp 40 t_{10}) (C ptmp 40 t_{11}) <rename ptmp passwd> {CS ...}	2
19 (B)	(R f1 40 t_3) (C f1 40 t_6) (R pw 40 t_8) (C pw 40 t_9) (W ptmp 40 t_{10}) (C ptmp 40 t_{11}) (RN ptmp pw 40 t_{13})	0

Table 4.4: An Unsuccessful Parallel Derivation of Trace 2

4.6 Summary

In this chapter we described a language framework for specifying trace policies. The language is based on a novel type of grammar, a parallel environment grammar (PE-grammar). We applied the formal framework to the specification of valid execution sequences of programs. We also presented an efficient parsing method for recognizing valid execution trace of subjects in a distributed setting.

Chapter 5

Example Trace Policies

In this chapter we present several example trace policies of the Unix programs described in Chapter 2. The trace policies can be used to detect exploitations of the vulnerabilities in these programs. Our goal is to illustrate the usefulness of our specification-based approach and the expressiveness of parallel environment grammars. We show that parallel environment grammars can specify many different kinds of trace policies, and we comment on the approaches to their development.

Many approaches can be taken to develop trace policies for programs. One approach is to identify what operations a program needs to perform in accomplishing its function and to allow in the policy only these operations. This approach actually applies the *least privileges* principle [44], and is particularily useful for specifying programs that are designed to performed specific functions (e.g., privileged programs). Another approach is to focus on some important properties of a program such as synchronization and specify the desirable behavior with respect to these properties. Also, one can develop a trace policy based on some suspected or existing weaknesses of the program.

5.1 Rdist

We present two trace policies for *rdist* that can be used to detect the exploitation of *rdist* described in Chapter 2. The first trace policy specifies the valid accesses of *rdist* and *rdist* server. It is developed based on what operations *rdist* and *rdistd* need to do in order to accomplish their functions. The second policy describes exactly the sequence of operations performed by an *rdist* server during execution.

When invoked, an *rdist* client reads the specified *distfile*, executes *rdistd* on the remote machine, and sends commands to *rdistd* to perform the remote updates. Therefore, the client should read only files belonging to the invoker, and should not write to any file. For the server, the main function is to listen to the client commands and perform the file updates. To update a file F, *rdistd* creates a temporary file, writes the new contents to the temporary, and renames the temporary file to F. The rename operation essentially replaces the contents of the file F. It also changes the ownership and the permission mode of the file to correspond to the original file when necessary. Therefore, *rdistd* is allowed to create a temporary file, change the owner and the permission mode of the file it creates, and rename the temporary file to the target file. We cannot specify exactly what files *rdist* will update because they are specified in the input *distfile* and are not recorded in the audit trails. However, *rdist* should update only the files owned by the invoker in the remote machine. Therefore, *rdist* should be allowed to rename the temporary file to a pathname that is under the home directory of the invoker.

Figure 5.1 shows the first trace policy for *rdist*. The PE-grammar describes the valid operations of a single *rdist* execution. Line 1 shows the header of the specification which consists of a *selection expression* indicating the subjects with which the specification is concerned. The meaning of a selection expression is described in Chapter 6. For this specification, it describes the operation sequence of a single execution of *rdist* on the host *blanc*. Lines 2-6 show the initial environment assignment. `U` is initialized to the user associated with the execution, which is returned from *getuser()*. `PID` contains the process ID (obtained from *getpid()*) of the process corresponding to the execution. `FILECD` (`PATHCD`) is an associative array for storing the *inode* numbers (pathnames) of the files created by the program execution. They are both initialized to empty and are changed during parsing when a create-file operation is recognized. `HOMEDIR` is initialized to the home directory of the invoker, which is */export/home/¡Username¿*. Its value will not be changed thereafter.

On line 7, the start expression contains only the start notion `<rdist>`, which implies that the input is described by the hypernotion `<rdist>`. Hyperrule 8 recursively defines the input as a repetition of the valid operations specified by `<valid_op>`; the second alternative represents the termination condition.

Hyperrule 9 describes the operations *rdist* is allowed to perform. It has 9 alternative

```
1.  SPEC rdist  <rdist(I), U, blanc>
2.        ENV User U = getuser();
3.        ENV int  PID = getpid();
4.        ENV int FILECD[int];
5.        ENV int PATHCD[str];
6.        ENV str HOMEDIR = "/export/home/U.name";

7.        SE : <rdist>
8.        <rdist> -> <valid_op> <rdist> | .
9.        <valid_op> -> open_r_worldread
                |  open_r_not_worldread
                   {    if !Created(F) then
                              violation();   fi
                   }
                |  open_rw
                   {    if !(Dev(F)) then violation(); fi;   }
                |  creat_file
                   {    if !(Inside(P, "/tmp") || Inside(P, HOMEDIR)) then
                              violation();   fi
                        FILECD[F.nodeid] = 1;
                        PATHCD[P] = F.nodeid;
                   }
                |  creat_dir
                   {    if !(Inside(P, "/tmp") || Inside(P, HOMEDIR)) then
                              violation();   fi
                   }
                |  symlink
                   {    if !(Inside(P, "/tmp") || Inside(P, HOMEDIR)) then
                              violation();   fi
                   }
                |  chown
                   {     if !(Created(F) and T.newowner = U)) then
                              violation();   fi
                   }
                |  chmod
                   {     if !(Created(F)) then violation(); fi   }
                |  rename
                   {    if !(PathCreated(P) && Inside(P1, HOMEDIR)) then
                              violation()   fi;
                   }.
10.  END;
```

Figure 5.1: A Parallel Environment Grammar for Monitoring *Rdist*

right-hand sides. The first alternative contains just a terminal, and each of the remaining eight alternatives contains a terminal followed by one or more semantic actions. The tokens recognized by the hyperrule are of types `open_r_worldread`, `open_r_not_worldread`, `open_rw`, `creat_file`, `creat_dir` `symlink`, `chown`, `chmod`, and `rename`.

The semantic actions in various alternatives of the hyperrule check the attributes (such as the pathname and the *inode* of the file) of the recognized operation to determine whether the operation is valid. They raise a violation for an invalid operation by calling the function *violation()*. Two attributes of operations are referenced in the semantic actions: F denotes characteristics of the process and the file associated with the recognized operation, and P denotes the pathname of the file associated with the recognized operation. See Appendix C for a full listing of the attributes of the operations for a Unix system.

```
1.   #define WorldReadable(F)   ((F.pmode & 004) != 0)
2.   #define Inside(A, B)       (A ~= B/*)
3.   #define Created(F)         (FILECD[F.inode] == 1)
4.   #define PathCreated(P)     (PATHCD[P] == 1)
5.   #define Dev(F)             ((F.pmode & P_DEV) != 0)
```

Figure 5.2: Definitions of the Macros

Macros are used in the semantic action. Their definitions are shown in Figure 5.2. Their meanings are as follows.

- *WorldReadable(F)* takes a file as parameters and returns *true* if the file is publicly readable.
- *Inside(A, B)* takes two pathnames as parameters and returns *true* if the first pathname is inside the second pathname. It uses regular pattern matching to determine the result.
- *Creat(F)* is used in conjunction with the environment variable `FILECD`, which stores all the files created by the process. It returns *true* if the given file is defined in the associative array *FILECD*, i.e., the file is created by the program execution.
- *Pathcreat(P)* is used in conjunction with the environment variable `PATHCD`, which stores all the pathnames created by the process. It returns *true* if the given pathname is defined in the associative array *PATHCD*, i.e., the pathname is created by the program execution.
- *Dev(F)* returns *true* if the given file is a device file.

The semantic action in the second alternative raises a violation if the file is not created by the process. The semantic action in the third alternative raises a violation if the recognized *open_rw* operation is not associated with a device file. There are three semantic actions following *creat_file* in the fourth alternative. The first semantic action raises a violation if the file associated with the operation is not inside the */tmp* directory or the home directory (specified by the environment variable `HOMEDIR`). The second and third semantic actions update the environment variable `FILECD` and `PATHCD` to indicate that a new file and a new path has been created. The semantic action following *symlink* raises a violation if the file is not inside the */tmp* directory or the home directory. The semantic actions following *chown* and *chmod* raise a violation if the file is not created by the process. The semantic action following *rename* raises a violation if old pathname associated with the rename operation is not created by the program or the new pathname is not inside the home directory of the invoker.

To summarize, the policy specifies that *rdist* can (1) open a publicly readable file for reading, (2) open a file that is created by itself for reading, (3) open a device file for both reading and writing, (4) create a new file, directory, or symbolic link that is inside the */tmp* directory or the home directory of the invoker, (5) change the permission mode and the ownership of a file that is created by the program execution itself, and (6) rename a file whose name is created by the program and inside the */tmp* directory.

The exploitation of *rdist* described in Chapter 2 cause *rdist* change the permission of */bin/sh*. Therefore, the exploitation makes *rdist* violate this trace policy and hence will be detected.

5.1.1 Sequence of Operations

Figure 5.3 shows the second PE-grammar, which describes the sequence of operations performed by an *rdist* server during execution. In order to develop with this specification, a fairly good knowledge about *rdistd* is required.

Line 1 shows the header of the specification, which indicates that this specification is concerned with a single execution of *rdist*. Lines 2-3 show the initial environment assignment. *U* is initialized to the user associated with the process. *NODEID* is initialized to 0. It is used to hold the *inode* number of the file associated with the operation recognized by hyperules 7, 9 and 10 temporarily. Hyperrule 5 describes the execution of *rdistd* as a repetition of the hypernotion <`command`>, which describes the operations *rdistd* performs

```
1.    SPEC rdist (<?, rdist, *, *>)
2.         ENV User U = getuser();
3.         ENV int NODEID = 0;

4.         SE:  <rdist>
5.        <rdist> -> <command> <rdist> |.
6.        <command>  -> <up_file> | <up_slink> | <up_dir>.
7.        <up_file> -> creat {  NODEID = F.nodeid; }
                       <opt_chmod>
                       <opt_chown>
                       <efile>.
8.        <efile> -> rename-NODEID | unlink-NODEID.
9.        <up_slink> -> symlink {  NODEID = F.nodeid; }
                        <efile>.
10.       <makedir> -> (mkdir)  { NODEID = F.nodeid; }
                       <opt_chmod>
                       <opt_chown>.
11.       <opt_chmod> -> chmod-NODEID.
12.       <opt_chown> -> chown-NODEID-U |.
13.   END;
```

Figure 5.3: A Parallel Environment Grammar for Monitoring *Rdistd*

in a command. The hyperrule models the way an *rdist* server works: it repeatedly waits for the next command and performs the command. Hyperrule 6 describes <`command`> as either <`file`>, <`slink`>, or <`dir`>, which represents the sequence of operations *rdist* performs in an update-file command, an update-symbolic-link command, or an update-directory command. Hyperrules 7-8 and 11-12 describe the operation sequence in an update-file command. Hyperrule 7 specifies a update-file command as a concatenation of the terminals and hypernotions `creat`, <`opt_chmod`>, <`opt_chown`>, and <`efile`>. When hyperrule 7 recognizes `creat` (a create operation), its attached environment assignments set `NODEID` to the *inode* of the newly created temporary file. Hyperrules 11 and 12 recognize tokens of types `chmod-NODEID` and `chown-NODEID-U`, which represent *chmod* operations on a file whose *inode* is equal to `NODEID`, as well as *chown* operations on a file to a new owner `UID` whose *inode* is equal to `NODEID`. Therefore, the two hyperrules recognize *chmod* and *chown* operations on files that are created by the program. Hence, the intrusion [1] described in Section 2.1 can be detected using this trace policy. Hyperrule 8 specifies the last operation in the sequence, which could be a *rename* or *unlink* operation depending on whether the update is successful.

Similarly, hyperrule 9 describes the operation sequence in an update-symbolic com-

[1]The intrusion exploits *rdist* to change the permission of */bin/sh* to enable the setuid bit of the file.

mand. *rdist* creates a temporary symbolic link and then renames it to the target. Lastly, hyperrule 10 describes the operation sequence in a update-directory command.

The tokens recognized by the lexical procedures are `creat`, `rename-`i, $1 \leq i \leq MaxInode$, `unlink-`i, $1 \leq i \leq MaxInode$, `chmod-i` $1 \leq i \leq MaxInode$, and `chown-`i-j, $1 \leq i \leq MaxInode$, $1 \leq j \leq MaxUid$. `creat` represents all *creat* operations; `rename-i`, `unlink-`i, and `chmod-`i represent all *rename*, *unlink*, and *chmod* operations on a file whose *inode* is equal to i; `chown` represents all *chown* operations on a file whose *inode* is i to a new owner j.

5.2 Fingerd

We describe a PE-grammar that specifies the valid operation sequence of the finger daemon, including the main *fingerd* process as well as the child process. The goal is to illustrate a PE-grammar that describes the execution trace of two or more processes.

In current Unix systems, the finger daemon does not run continuously in the background waiting for incoming finger requests. It is invoked by the internet daemon (*inetd*) when a finger request arrives at the finger port. When invoked, it forks a child to serve the finger request. The child process reads a single command line containing the request and invokes the finger program with the request as parameters. The finger program then collects the information from various status files in the system and reports back to the requestor. Basically, *fingerd* needs to create a child process and read some status files. The child process needs to execute the finger program, which reads system status files.

Figure 5.4 shows the PE-grammar. The start expression consists of the start notion <`fingerd`> and <`finger`>; <`fingerd`> describes the valid operation sequences of the parent *fingerd* process and <`finger`> describes the valid operation sequences of the child process. The PE-grammar describes any trace of the two processes, which consists of two subtraces described by <`fingerd`> and <`finger`>.

Hyperrules 5-6 specify the valid operation sequence for the main process, which is a repetition of the valid operations described by <`valid_op`>. According to Hyperrule 6, <`valid_op`> refers to tokens of types `open_r_worldread` or `fork`. The former represents *open_r* operations on a publicly readable file, and the latter represents *fork* operations. The

```
1.  SPEC fingerd <?, fingerd, *, *>

2.  LOCAL ENV user U = getuser();
3.  LOCAL ENV int PID = getpid();

4.  SE:   <fingerd> || <finger>

5.        <fingerd> -> <valid_op> <fingerd> |.
6.        <valid_op> -> open_r_worldread
                   |  fork
                      {    printf("Child pid = %d\n", M.chpid);   }.

7.        <finger> -> <valid_op1> <finger> |.
8.        <valid_op1>   -> open_r_worldread
                   |  exec_finger
                   |  open_rw_dev.

9.  END;
```

Figure 5.4: A Parallel Environment Grammar for Monitoring *Fingerd*

second alternative of hyperrule 6 has a semantic action which prints out the process ID of the child process. All other operations of the main process will not be recognized by hyperrule 6 and are considered violations.

Hyperrules 7-8 specify the operation sequence for the child process, which is a repetition of the operations described by <`valid_op1`>. Hyperrule 8 defines the valid operations of the child, which are represented by the three tokens `open_r_worldread`, `exec_finger`, and `open_rw_dev`. It specifies that the child process can read any publicly readable file, execute the finger program, and write to the terminal file.

As mentioned in Section 2.2, the finger daemon contains a vulnerability that enables a remote attacker to inject his own code and make the daemon execute the code. This vulnerability was exploited by the Internet Worm [47, 18] to execute a copy of the worm in hosts that provide the finger service. The worm attack obviously violates the trace policy since *fingerd* is allowed to execute only the finger program.

5.3 Race Condition: *Binmail*

The *binmail* example illustrates how to specify a trace policy that can be used to detect exploitation of a race-condition flaw in a program using a PE-grammar. The program selected for this example is the backend mail delivery program in Unix, *binmail*, which contains a race-condition flaw.

```
1.  SPEC <(?, binmail, U, H)>

2.     ENV int CREATTMP = 0;
3.     ENV int PID = getpid();

4.     SE: <binmail> || <other>
5.     <binmail> -> <init> <mktemp> <rest>.
6.     <init> -> <not_mktemp> <init> | Nil.
7.     <reat> -> any_op <rest> | Nil.

8.     <mktemp> ->  open_tmpfile-PID {  CREATTMP = 1; }.

9.     <not_mktemp> -> not_open_tmpfile-PID

10.     <other> -> <vop, CREATTMP> <other> | Nil.
11.     <vop, 0> -> not_chgtmp.
12.     <vop, 1> -> any_op.

13.   END;
```

Figure 5.5: A Parallel Environment Grammar for Monitoring *Binmail*.

As mentioned in Chapter 2, an attack can user *binmail* to compromise a system. When invoked, *binmail* creates a temporary file using the *open_rwtc* system call. If an attacker process performs an operation that changes the binding of the name of the temporary file (of the form "/tmp/ma?PID") to a file he wants to replace before *binmail* creates the temporary file, *binmail* will overwrite the file. In order to check whether the execution of *binmail* is secure, the execution of *binmail* itself, as well as all other processes in the system need to be monitored.

Figure 5.5 shows a PE-grammar that prohibits traces in which the race-condition flaw in *binmail* is exploited. On line 2, the environment variable `CREATTMP` is initialized to 0, indicating *binmail* has not created the temporary file. It is set to 1 when hyperrule 5 recognizes an operation which creates a temporary file of the form */tmp/ma?PID*. On line 3, `PID` is initialized to the *pid* of the process running *binmail*.

Hyperrules 5-9 describe the operation sequence of *binmail* as the initial sequence `<init>` followed by the create-temporary-file operation `<mktemp>` and the rest of the operations, `<rest>`. Hyperrule 5 describes the initial sequence, which consists of any operations except the operations that create a temporary file. Hyperrule 6 describes the `<rest>` as a repetition of any operation, represented by the token `any_op`. Hyperrule 8 recognizes an operation that creates the temporary file, which is represented by the token `open_tmpfile-PID`. The token `open_tmpfile-PID` represents any *open_rwtc* or *open_rwt* operations on a file whose pathname matches the regular expression */tmp/ma?PID*, where the environment variable PID contains the process ID of the process executing *binmail.* There is a semantic action attached, which changes CREATTMP to 1, indicating *binmail* has created the temporary file. Hyperrule 9 recognizes operations of type `not_open_tmpfile-PID`, which are all operations except those described by `open_tmpfile-PID`.

Hyperrules 10-13 describe the operation sequence of all other processes. Hyperrule 10 describes a valid operation sequence as the repetition of the valid operations described by `<vop, CREATTMP>`. The operation represented by `<vop, CREATTMP>` depends on the value of the environment variable `CREATTMP`. If `CREATTMP` equals 0, `<vop, CREATTMP>` is described by hyperrule 11 as any operations except *link* or *symlink* on the temporary file. If `CREATTMP` equals 1, `<vop, CREATTMP>` is described by hyperrule 12 as any operations. If a process performs a *link* or *symlink* operation that changes the binding of the temporary file when CREATTMP is 0, the parsing will, therefore, fail. The PE-grammar thus describes a trace in which *binmail* is not exploited.

5.4 Concurrent Access to the Password File

This section presents a PE-grammar that describes valid execution sequences of the *passwd* and *vi* programs in Solaris 2.X with respect to synchronization. The PE-grammar disallows any concurrent accesses to the shadow password file */etc/shadow.* The input trace is the merge of traces of processes that are executing *passwd* as well as processes that are executing *vi* and owned by root.

Figure 5.6 shows the parallel environment grammar. Line 1 indicates that the trace policy is concerned with all executions of *vi* and *passwd.* Lines 2-3 show the initial environment assignment of the two environment variables in the PE-grammar. The global environment variable `MUTEX` holds the number of program executions (or processes) that are modifying the shadow password file; it is initialized to 0, indicating that no process is modi-

```
1.    SPEC (<*, passwd, *, H>, <*, vi, root, H>)
2.        ENV int MUTEX = 0;
3.        LOCAL ENV int FD = 0;

4.        SE is <passwd>* || <vi>*

5.        <passwd> ->  <begin> <update> <end>.
6.        <begin>  ->  not_lock <begin> | Nil.
7.        <end>    ->  any_op <end> | Nil.
8.        <update> ->  <Lock> <Rd_shpw> <Wr_shpw> <Unlock>.
9.        <Lock>   ->  fcntl_lock.

10.       <Rd_shpw>    -> open_r_shadow
                          {   if MUTEX > 0 then violation();
                              MUTEX++;                        }
                       |  not_open_r_shadow <Rd_shpw>.

11.       <Wr_shpw>   ->  rename_shadow_stmp
                          {   MUTEX--;        }
                       |  not_rename_shadow_stmp <Wr_shpw>.

12.       <Unlock> ->  not_fcntl_unlock <Unlock> | fcntl_unlock.

13.       <vi>   -> <initseq> <mod_shpw> <rest>

14.        <initseq> ->   not_open_r_shadow <initseq> | Nil.
15.        <mod_shpw> ->  <read_shpw> <pad> <write_shpw> | Nil .
16.        <read_shpw> -> open_r_shadow
                          {   if MUTEX > 0 then violation(); fi;
                              MUTEX++;
                          }.
17.        <write_shpw> ->  creat_shadow   { FD == F.fd; }
                            close_shadow-FD
                            {   MUTEX--;        }
                         |   Nil.

18.        <pad>     ->   not_creat_shadow <pad> | Nil.

19.        <rest>    ->   any_op <rest> | Nil.
```

Figure 5.6: A Trace Policy for Accesses to the Password file.

fying the shadow password file at the beginning. The local environment variable `FD` holds a file descriptor temporarily after hyperrule 16 recognizes `creat_shadow`, which represents a *create* operation on the file */etc/shadow*. It is used later in the token `close_shadow-FD` for recognizing the *close* operation on the same file descriptor.

The start expression is <`passwd` > ∗ || <`vi`> ∗, which indicates that the input trace consists of any number of *passwd* executions and *vi* executions. The symbol ∗ applied to a start notion means any number of the start notions in parallel.

Hyperrules 5-12 describe the trace of a *passwd* execution, specified by < *passwd* >. The hyperrules do not model the execution of *passwd* in detail. Instead, only operations that are related to modification of the shadow password file are modeled. Hyperrule 5 specifies a *passwd* execution as the concatenation of the hypernotions <`begin`>, <`update`>, and <`end`>, which specify the initial sequence of operations, the sequence of operations *passwd* performs to modify the shadow password file, and the remaining sequence of operations. Hyperrule 6 describes the initial sequence of operations by recursively recognizing operations other than the `lock` operations (specified by `not_lock`); the second alternative serves as the termination condition of the first alternative. Hyperrule 7 describes the end sequence, which is a sequence of any operations, represented by `any_op`.

Hyperrule 8 describes the update sequence, which models the way that *passwd* modifies the shadow password file. It locks the lock file */tmp/.pwd.lock*, opens the shadow password file for reading, renames the temporary file into the shadow password file, and unlocks the lock file. Hyperrule 9 recognizes the first lock operation, represented by the token `fcntl_lock`. Hyperrule 10 describes the open-read operation on the shadow password file, `open_r_shadow`. It also accepts other operations before it accepts the open-read operation. It is because *passwd* may perform other operations between the lock operation and the open-read operation. Similarly, Hyperrule 11 and 12 describe the rename-shadow operation (`open_r_shadow`) and the unlock operation (`open_shadow_stmp`).

Hyperrule 10 has two attached actions; the first of these checks `MUTEX` and issues a violation if the value is not zero. The second increases the value of `MUTEX` by 1, indicating that the process is in the midst of modifying the shadow password file. Hyperrule 12 has an attached environment assignment which decreases `MUTEX` by 1, indicating that it has finished modifying the shadow password file.

Hyperrules 13-19 specify the execution of *vi*. Hyperrules 13 specifies the trace as the concatenation of the initial sequence <`initseq`>, the update sequence <`mod_shpw`>, and the rest of the sequence <`rest`>. Hyperrule 14 describes <`initseq`> recursively, which is a sequence of operations other then the *open_r* operation on */etc/shadow*. Hyperrule 15-18 describes the update sequence, which consists of an read operation followed by a sequence of operations and then by a write operation, or just an empty sequence, which indicates that an execution of *vi* may or may not modify the shadow password file. Hyperrule 15 recognizes the read operation as *open_r* on */etc/shadow*. It has two environment assignments attached; the first of these checks the value of `MUTEX` and reports an error if the value is greater than 0; the second increases the value of `MUTEX`. Hyperrule 16 recognizes the write operation to the shadow password file, *open_w(/etc/shadow)* and *close*(). The attached environment assignment decrements `MUTEX` by 1. Hyperrule 17 recognizes the operations between the read and write operation. Hyperrule 18 recognizes the rest of the operations after the modify sequence.

5.5 Other Policies

This section illustrates how commonly used policies can be specified using PE-grammars. We deal with the Bell-LaPadula Policy and the Clark-Wilson Policy.

5.5.1 Bell-LaPadula Policy

Figure 5.7 shows the Multi-level Security (MLS) Policy [5] for a Unix system written as a PE-grammar. The MLS policy requires that a user read only files whose security level is not greater than his current clearance level (Basic Security Property) and write only to files whose security level is not less than his current clearance level (*-Property).

The trace policy is concerned with the execution traces of all non-privileged programs in the system, that is, those programs that must not violate the MLS policy. Lines 2-3 describe the environment variables. `SL` is an array for storing the security level of every physical file (uniquely identified by an *inode* number) in the system. `CL` is an array which stores the clearance level of each user in the system. The initial environment assignment initializes the two arrays to correspond to the current security state of the system.

Line 4 defines the start expression as the hypernotion <`opns`>. Hyperrule 5 specifies the input as a repetition of valid operations represented by <`opn`>, where the second alternative serves as the termination of the first recursive alternative. Hyperrules 5-9 define valid operations. Hyperrule 6 recognizes four different types of operations. The first alternative recognizes read operations `read`; the attached semantic action requires that the clearance level of the user be not less than the security level of the file. The second alternative recognizes write operations `write`; the attached semantic action requires that the clearance

```
1. SPEC <*, NonPriv, *, *)>

2.     ENV Slevel SL[Inode];
3.     ENV Clevel CL[User];

4.     SE : <opns>
5.     <opns> -> <opn> <opns> | .
6.     <opn>  -> read  {   if CL[U] < SL[F] then
                                 violation();
                       }
              |  write {   if CL[U] > SL[F] then
                                 violation();
                       }
              |  creat {   SL[F] = CL[U];
                       }
              |  rdwr  {   if CL[U] != SL[O] then
                                violation();
                       }.

7. END;
```

Figure 5.7: An PE-grammar for a Multi-Level Security Policy.

level of the user should not exceed than the security level of the file. The third alternative recognizes create operations `create`; the attached semantic action sets the security level of the file to be the clearance level of the user. The fourth alternative recognizes operations (`rdwr`) that give both read and write access on the file to the process; the attached semantic action requires that the clearance level of the user must be equal to the security level of the file as it has to satisfy both the basic security property and the *-property.

5.5.2 Clark-Wilson Policy

A Clark-Wilson Policy [9] requires that not only a file (or a piece of data) should be accessed by authorized users, but also using authorized programs (or transformation procedures). That is, an access to an object is regulated based on the user and the program the user is using. We present a Clark-Wilson Policy written as a parallel environment grammar in Figure 5.8.

Line 1 shows the specification header, which indicates that the policy is concerned with every program in the system. Line 2 shows the environment variable `CWMatrix`, which is the Clark-Wilson Access Matrix defining the accesses that the ordered pair (user, program) has over objects. If a user U using a program P is authorized to access a file F, then CWMatrix[U,

```
1.    SPEC <*(*), U, *)>

2.    ENV Bool: CWMatrix[ProgId, User, Inode];

3.        <opns> -> <opn> <opns> | nil.
4.        <opn>  -> modify  {   if CWMatrix[S.prog, S.ruid, F.NodeId] != 1
                                       violation();    fi;
                           }.

5.    END
```

Figure 5.8: An PE-grammar for a Clark-Wilson Policy.

P, F] is 1, otherwise, it is 0. We assume that `CWMatrix` is initialized to correspond to the current integrity policy.

The specification describes the accesses a program has over objects. On line 4, when a program modifies a file, the attached action raises a violation if the modification is not authorized. It consults `CWMatrix` to check whether a program used by a user is authorized to modify a file.

Chapter 6

Design and Implementation Overview

Chapter 5 presented a language for specifying the trace policies and a method for checking whether an execution trace conforms to a trace policy. Using the language, we are able to specify trace policies for security-relevant programs to further restrict the behavior of these programs in a system. In this chapter we present the design of a monitoring system, *Distributed Program Execution Monitor* (DPEM), which monitors executions of programs in a distributed system to detect behavior inconsistent with their trace policies. We also describe a prototype implementation of the DPEM and share our experiences in using DPEM to detect intrusions in computer systems.

6.1 Design of DPEM

The target platform is a distributed system which consists of several hosts connected by a local area network. Each host in the system collects audit trails about the system operations that occur in the host, which should include all system calls.

DPEM consists of a *director*, a *specification manager*, *trace dispatchers*, *trace senders*, and *analyzers* situated in various hosts in the distributed system. Our design combines distributed data collection and data reduction with decentralized analysis. Our system is the first that enable data analysis to be carried out concurrently on multiple hosts. Also, each component is designed to be as simple as possible, and the amount of audit data that

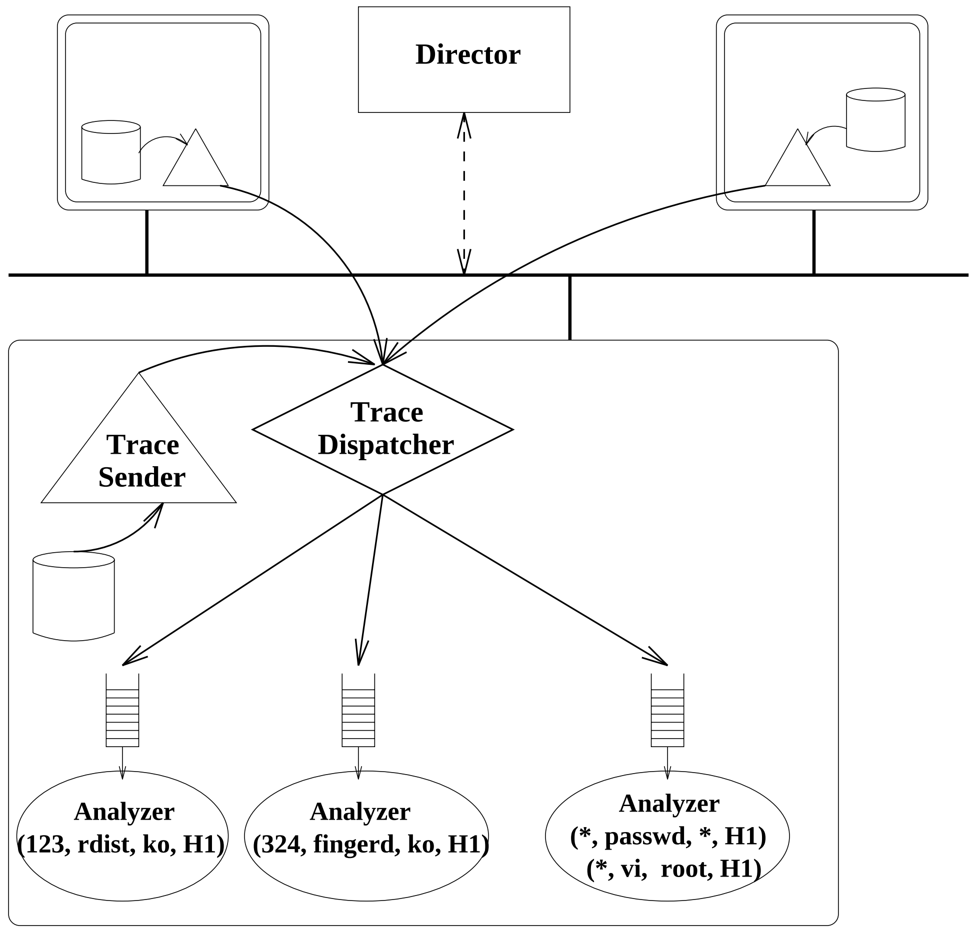

Figure 6.1: Architecture of the Execution Monitor from the Perspective of a Host

needs to be transferred across the network is minimized.

Figure 6.1 depicts the architecture of DPEM from the perspective of a host in the distributed system. It also shows the data flow among various components. Analyzers are the components that perform the monitoring. An analyzer checks the execution trace of a subject for violations with respect to a trace policy, where the policy is the basis for the analyzer. Thus, an analyzer can be thought of as the runtime counterpart of a policy specification. The number of analyzers running in the system can change dynamically. In Figure 6.1, three analyzers are running, and each of them is associated with a different trace policy. The first analyzer is associated with a trace policy concerning the behavior of a single execution of *rdist*. The second analyzer is associated with a trace policy concerning the behavior of a single execution of *fingerd*. The third analyzer monitors all executions of *passwd* and *vi*. Additional analyzers will be executed when a program that needs to be monitored is executed, and running analyzers terminate when the subjects being monitored exit. For instance, if a user executes *rdist*, an analyzer associated with the trace policy concerned with *rdist* will be executed to monitor that execution of *rdist*. An analyzer reports any erroneous behavior of the monitored subject to the director, which carries out the appropriate response for the incident, such as notifying the system security officers and firing up additional analyzers. An analyzer can run on any host in the system. Therefore, the analysis of audit data is distributed among multiple hosts in the distributed system.

A trace dispatcher must be presented on each host where analyzers are running. It is responsible for sending the execution traces of the subjects to the analyzers running on the host. There is one message queue per analyzer.

A trace sender runs on each host where an audit trail resides. It fetches the audit records directly from the audit repository and sends the records to the trace dispatchers (possibly situated on different hosts) that request the records. It performs filtering in that only the records requested by the dispatcher will be sent, thus minimizing the network bandwidth used by the monitoring system.

The specification manager enables the system administrator to manage the security specifications. An administrator can add, modify, or delete the security specifications in the system through the specification manager interface. The specification manager starts up analyzers to monitor program executions when programs that need to be monitored are executed.

6.1.1 The Specification Manager

The specification manger keeps all the trace policies in a specification database. Each trace policy is associated with a selection expression indicating the subjects with which the policy is concerned. At a high level, a subject could be one or more program executions, one or more users, and one or more hosts. At the system level, a subject consists of one or more distributed processes. When a new distributed process is created, i.e., a program is executed, the specification manager checks the selection expressions to see whether the distributed process belongs to any of the monitored subjects. When necessary, it invokes the corresponding analyzer to monitor the new process.

The elements in a selection expression are derived from the characteristics of the distributed system, which include the set of programs P, the set of users U, and the set of hosts H. Also, two special symbols $*$ and ? are used. Intuitively, they mean "all" and "any". A selection expression is a list of selectors $(s_1, s_2, \cdots, s_n)$ where s_i $(1 \leq i \leq n)$ is a selector of the form

$$< \; PID,\; PS,\; US,\; HS >,$$

where

- PID is the distributed process selector, $PID \in \{* \;,\; ?\}$,
- PS is the program, $PS \in P \cup \{* \;,\; ?\}$,
- US is the user, $US \in U \cup \{* \;,\; ?\}$, and
- HS is the host, $HS \in H \cup \{* \;,\; ?\}$.

A selection expression identifies one or more subjects. A selection expression without the ? symbol defines a single subject; a selection expression consisting of ? defines a set of subjects. The symbol ? in a selection expression is similar to a variable in a generic template, which is given a value in an instantiation. For instance, $< *, rdist, ko, ? >$ defines a set of subjects which contain the subjects $< *, rdist, ko, k2 >$, $< *, rdist, ko, blanc >$, and so on, with ? replaced by each of the hosts in the distributed system.

We give several selection expressions and the subjects they refer to below.

1. $<?, rdist, *, * >$ defines a set of subjects. A subject here is a distributed process executing the *rdist* program, and an analyzer monitors one execution of *rdist*. When there are multiple executions of *rdist*, multiple analyzers will be executed, and each monitors one execution.

2. $< *, passwd, *, blanc >$ *and* $< *, vi, root, blanc >$ defines all executions of *passwd* and all executions of *vi* by *root* on the host *blanc*.

3. $< *, ?, ko, * >$ defines all executions of a program by user ko on any host.

4. $< *, *, ko, blanc >$ defines the user ko on the host *blanc*.

5. $< *, *, *, k2 >$: the host *k2*, i.e., all processes on *k2*.

Recall from Chapter 3 that an execution of a program is a distributed process. A distributed process is identified by a process ID (pid), the user it is representing, the program it is executing, and the host on which it is running. Given a selection expression and a distributed process identified by its *pid*, user, program and host, a simple matching can determine whether the distributed process belongs to any of the monitored subjects specified by a selection expression.

When the specification manager receives an audit record indicating a process pid associated with a user u who executes a program p on a host h, it matches (pid, p, u, h) with the selection expression of each trace policy. If the selection expression of a trace policy matches the audit record, it is instantiated to a subject. A subject is defined by a subject expression, a selection expression that does not contain any ? symbol. For example, the selection expression $<?, rdist, *, * >$ of a trace policy P is instantiated to the subject $< 456, rdist, *, * >$ when a user executes *rdist*, and the *pid* of the process created is 456. If there is no analyzer associated with P monitoring the subject, an analyzer will be started to monitor the subject. In general, no two analyzers which are associated with the same trace policy and are monitoring the same subject must exist simultaneously.

The part of a specification manager that checks the trace and executes an analyzer can be thought of as an analyzer, which monitors all program-execution operations in the system.

6.1.2 Trace Dispatchers

A trace dispatcher is responsible for providing the execution traces needed by the attached analyzers. Conceptually, it reads the audit trace from each of the hosts in the system, merges them to form a single time-ordered audit trace of the whole distributed system, and filters the trace to obtain a subtrace required an analyzer.

The trace dispatcher keeps a process set for each analyzer it serves. The process set contains the *pid* of processes with which the subject expression of the analyzer is associated. The trace dispatcher updates the process sets as it processes the audit records.

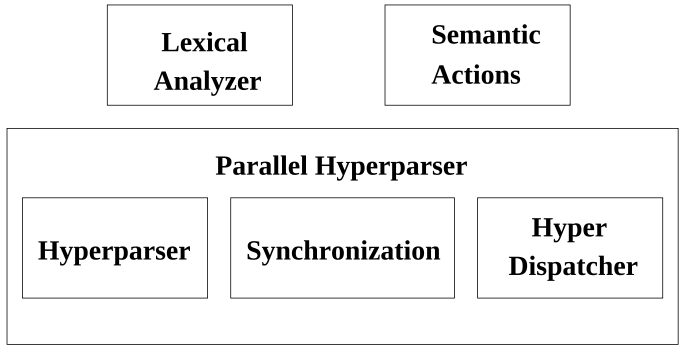

Figure 6.2: Structure of an Analyzer

When it reads an audit record, it first identifies the analyzers whose process sets contain the *pid* associated with the audit record. It then dispatches the record to the input queue of each of the identified analyzers. If the record denotes a create-process operation, it adds the process ID of the new process to each process set which contains the parent process. If the record denotes a program-execution operation, it checks the subject expression of each analyzer and adds the *pid* associated with the record to the process set of the analyzer if it is not in the set already. When a process exits, it removes the process from all the process sets.

A trace dispatcher gets audit records from the trace senders situated in various hosts. Depending on the subjects monitored by the attached analyzers, a trace dispatcher may or may not request audit records from the trace sender in a host. It identifies the trace senders from which audit records are needed and requests audit records from them only when necessary.

6.1.3 Analyzers

An analyzer monitors the execution of a subject with respect to a trace policy. A different analyzer is constructed for each trace policy and is invoked when a subject that is to be monitored starts. Each running analyzer has a subject expression which defines the subject in the system it monitors.

Figure 6.2 shows the modules of an analyzer. When an analyzer starts, it first calls the initialization procedure of the parallel hyperparser with information on the monitored

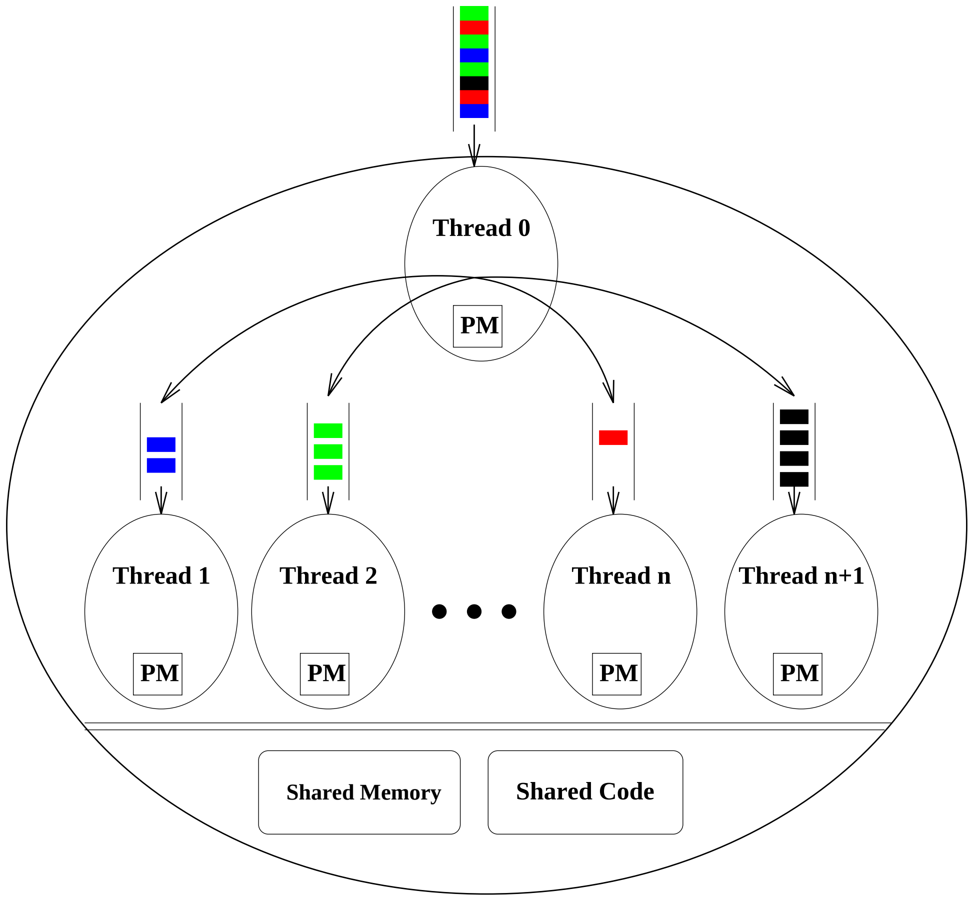

Figure 6.3: Architecture of an Analyzer

subject to initialize the environment variables. It also calls the initialization procedure of the lexical analyzer, the dispatcher module, and the semantic action module with information on the subject it monitors. After the initialization, it passes control to the parallel hyperparser which performs the audit data analysis with respect to the trace policy.

The design of the parallel hyperparser is based on the parallel environment grammar in the trace policy, following the methodology presented in Section 4.4. A parallel hyper-parser consists of a hyperdispatcher and several hyperparsers. The hyperdispatcher and the hyperparsers are threads of the analyzer process. The address space of the process is shared by all threads, which means that they share the same set of global variables. In addition, they also share the same set of open files, timers, signals, etc. Nevertheless, each thread has its own private variables [51].

Figure 6.3 depicts the run-time structure of a parallel hyperparser. The set of hyperprocedures is shared by all threads. When control is passed to the parallel hyperparser, the analyzer process consists of only the hyperdispatcher thread. The hyperdispatcher reads the audit records from the input queue, examines the records, and passes the records to the appropriate hyperparser threads.

The hyperdispatcher creates hyperparser threads on demand. Consider an analyzer corresponding to the trace policy for *vi* and *passwd* presented in Section 5.4. When a new execution of *vi/passwd* starts, the hyperdispatcher creates a hyperparser thread to parse the records associated with the process. When a hyperparser thread starts, it first calls the initialization procedure to initialize its local environment variables. Then it reads its input from a message queue.

We illustrate the structure of a parallel hyperparser written in the C programming language using the PE-grammar in Section 5.4. The parallel hyperparser consists of a set of lexical procedures and a set of (possibly recursive) hyperprocedures.

The parallel hyperparser contains 13 functions, corresponding to the hypernotions <`passwd`>, <`begin`>, <`update`>, <`end`>, <`Rd_shpw`>, <`Wr_shpw`>, <`vi`>, <`init`>, <`mod_shpw`>, <`rest`>, <`read_shpw`>, <`write_shpw`>, and <`pad`>.

Lines 1-4 show the variable declarations for the environment variables used in the PE-grammar. Global environment variables are declared as global variables in the program. Local environment variables are declared as parameters in each hyperprocedure and are passed in every subsequent call [1]. In this way, each hyperparser thread can have its own local environment variables.

```
1.   typedef struct {
2.       int FD;
3.   } LOC_ENV;
4.   int MUTEX = 0;
```

Lines 5-19 show the starting function and the hyperprocedure for the hyperrule

<`passwd`> → <`begin`> <`update`> <`optupdate`> <`end`>,

where <`passwd`> is a start notion. When the dispatcher creates a new hyperparser thread corresponding to <`passwd`>, the newly created thread calls the *Threadparser* function with a

[1] All local environment variables are place into a single structure, for convenience.

pointer to the hyperprocedure corresponding to the start notion $<passwd>$. The *Threadparser* function has a local variable *lenv*, a structure containing all the local environment variables. When invoked with a pointer to a hyperprocedure, *Threadparser* in turns calls the given hyperprocedure with *lenv* as the parameter, which is passed in the subsequent calls so that the hyperprocedures subsequently called can access the local environment variables.

Since the hyperrule does not contain any environment assignments or semantic actions, it simply calls the four hyperprocedures corresponding to the four hypernotions on the right-hand side of the rule.

```
5.    Threadparser(startHP)
6.    int *(startHP)();

7.    {
8.         LOC_ENV lenv;
9.         int i;
10.        i = (*startHP)(&lenv)
11.   }

12.   HP_passwd(lenvp)
13.   Local_Env *lenvp;

14.   {
15.          HP_begin(lenvp);
16.          HP_update(lenvp);
17.          HP_optupdate(lenvp);
18.          HP_end(lenvp);
19.   };
```

Lines 20-28 show the hyperprocedure corresponding to the hyperrule

<write_shpw> → creat_shadow { FD == $F.fd } close_shadow_FD { MUTEX-- }

The hyperprocedure first calls the lexical procedure corresponding to the token `creat_shadow`. It then executes the environment assignment on Line 24. It calls the lexical procedure corresponding to the token `close_shadow_FD`, which recognizes the audit record whose event is *close*, whose path field is */etc/shadow*, and whose *fd* field (the file descriptor) is equal to the environment variable FD.

```
20.   HP_write_shpw(lenvp)
21.   Local_Env *lenvp;

22.   {
23.          LEX_creat_shadow(&token);
24.          lenvp->FD = token->fd;
25.          LEX_close_shadow(&token, lenvp->FD)
26.          MUTEX--;
27.          return(SUCCESS);
28.   };
```

Synchronization Module

The synchronization module provides support for synchronization among the hyperparser threads. There is a global array CT for storing the time stamp of the current token each hyperparser is processing; $CT[i]$ stores the current token of hyperparser i. When thread j needs to wait, it checks all times CT. If all of them are greater than the current token time of j, it proceeds. Otherwise, it sends a message to the synchronization thread with the time of the current token and sleeps.

The synchronization thread maintains a synchronization structure for each waiting hyperparser. The structure contains the current token time of the waiting hyperparser and an array of bits for each hyperparser thread. For the waiting hyperparser i, bit j indicates whether hyperparser j is processing a token whose time is greater than the current token time of hyperparser i. It checks CT periodically and clears bit j if thread hyperparser j is processing a token whose time is greater than the current token time of hyperparser i. When all bits are clear, it wakes up hyperparser i.

The lexical analyzer scans the source input for tokens of the parallel environment grammar. The hyperparser invokes the lexical analyzer with a specification of the expected token type, and the lexical analyzer returns a success/failure indication together with the token value in case of success. The lexical analyzer is constructed based on the lexical description in the trace policy, following the approach shown in Section 4.4.4. The semantic action module contains the necessary procedures for performing the semantic actions. It includes procedure *violation()* which reports a violation and pertinent information to the security officer. The function does not terminate the parser; it allows parsing to continue. In general, users can code up their own procedures for use in semantic actions.

6.2 Implementation Overview: A Unix Prototype

We built a prototype execution monitor for a single host based on the design above. It serves as a proof of concept implementation for our approach.

The prototype is written in the C programming language [29]. The C programming language was chosen because of its wide-spread use, free availability, and portability across different Unix platforms. The prototype runs under the Solaris 2.4 operating system and uses the auditing services provided by the Sun BSM audit subsystem [50].

The BSM audit subsystem provides a log of the activities that occur in the system. It records the sequence of system events in the order of occurrence. Thus, the audit trails contain a trace of the system. An audit record contains information such as the process id (*pid*) and the user id (*uid*) of the process involved as well as the path name, the *inode*, and the permission mode of the files being accessed. However, it does not contain information about the program the process is running. Therefore, additional preprocessing is needed to associate a program with each audit record. The audit subsystem lets administrators decide what subjects to audit, and manage the audit data. We configured the audit system to audit all successful operations for all users in the system, including pseudo users [2]. Appendix A provides a description of the BSM audit subsystem.

The prototype consists of a trace dispatcher and a number of analyzers. The trace dispatcher reads audit records directly from the audit files and dispatches the records to the appropriate analyzers.

6.2.1 The Audit Record Preprocessor

The audit record preprocessor is part of the trace sender in our design. It serves two purposes. First, it filters audit records that are irrelevant to the monitoring system. Appendix B lists the audit records used by the prototype. Second, it translates the BSM audit records into the format required by the monitoring system. Our design requires that each audit record should be associated with the program the process is executing. The audit record preprocessor associates a program with an audit record as follows.

The audit record preprocessor keeps an array pg that holds the program each process is currently executing. When it reads an audit record associated with process ID x, it associates the program in $pg[x]$ with that record. It monitors all *exec* and *fork* calls and updates the array accordingly:

- *exec*: when a process invokes a program P, the program associated with the new process becomes P.

[2] In Unix, pseudo users are user accounts that are not associated with any real users. They are created for administrative purposes. For instance, many daemons (e.g., *inetd*) are running as pseudo users.

- *fork*: when a new process is created, the program associated with the new process is that of its parent.

When the execution monitor is started, several processes are typically executing in the system. It is impossible to trace back how the processes got created from the audit trail as the audit records could have been modified or deleted. To obtain the program each existing process is executing, the audit record preprocessor inspects the kernel memory and initializes the array *pg* when it starts up.

6.2.2 The Dispatcher

The dispatcher contains all the trace policies. It monitors all *fork* and *exec* calls. When a new process is created, it checks the selection expressions of the trace policies to determine if a new analyzer should be executed. It checks all subject expressions to determine whether to dispatch future records of the process to the analyzers.

6.2.3 Analyzers

In our prototype, each analyzer is a separate process. Each specification is translated to a C program equivalent to a parallel hyperparser. When the dispatcher needs to start up an analyzer, it forks and lets the child process execute the C program.

Our prototype maintains a table of registered programs. Each entry contains the name of a program, as well as the pathname, and the *inode* of the physical file that contains the program. In the identification of a program started by an *execve* event, the *inode* number in the record is used as the key because a pathname is not a unique identifier of a program. For example, an attacker can create a hard link to a setuid root program and execute the hard link, in effect executing the setuid root program. Yet the pathname of the *execve* record would not indicate an execution of a setuid root program.

6.3 Experience

This section describes our experience with the prototype. We have written trace policies for many *setuid root* programs and servers in Unix, including the programs described in

Chapter 2. Using these trace policies, we tested the prototype with three different kinds of intrusions. These intrusions exploit vulnerabilities in *rdist*, *sendmail*, and *binmail* described in Chapter 2.

The experiments described below were performed on a Sun SPARCstation 5 with 32MB of memory running Solaris 2.4. Auditing was enabled with the default configuration, which logs all successful events, and all network daemons, including *inetd*, *fingerd*, *rlogind*, and *telnetd*. The original version of *rdist* and *sendmail* were replaced by the SUN 4.3 versions, as their vulnerabilities have been removed in Solaris 2.4. The execution monitor runs continuously. It analyzes the audit data generated by the audit system in real time and reports any violations to the security specification.

The first intrusion was simulated using a Perl [54] script. The script essentially performs the operations that exploit the vulnerability in *rdist* described in Section 2.1. The script was executed when the execution monitor was running. Figure 6.4 shows the report generated by the execution monitor after the program was attacked. The intrusion simulated by the script actually produces two violations. The first is that *rdist* changes the ownership of a file not created by the program. The second is that *rdist* change the permission mode of */usr/bin/exsh*. The report shows the time each violation was detected and the time the violation occurred. The time of detection was obtained by the *gettimeofday()* library call when the hyperparser executed a semantics action that called the *violation()* function. The time of the operation was obtained from the audit records. The time elapsed between the occurrence of the violation and the detection was approximately 0.06 second. Also, no noticeable degradation of performance was noted.

The second and third intrusions were simulated using a Perl script and a C program respectively . The second intrusion exploits a vulnerability in the sendmail program that causes *sendmail* create a setuid shell in the */tmp* directory that is owned by root and is publicly executable. The third intrusion causes *binmail* overwrite the shadow password file with the contents the attacker desires. The execution monitor detected these intrusions within 0.1 second.

We also tested the prototype with the scenario that involves simultaneous modification of the password file. We used the trace policy concerned with *passwd* and *vi* described in Section 5.4. The specification requires that only one execution of *passwd* or *vi* can modify

```
% rdistattack /bin/sh
% /bin/sh

VIOLATION detected Tue May 14 16:52:20 1996 + 0.894236000 sec
------------------------------------------------------------
Tue May 14 16:52:20 1996 + 0.820003000 sec
lchown, (5456, 5000, 0, rdist), nodeid: 27, path: /tmp/rdista05456

VIOLATION detected Tue May 14 16:52:20 1996 + 0.972195000 sec
------------------------------------------------------------
Tue May 14 16:52:20 1996 + 0.830008500 sec
chmod, (5456, 5000, 0, rdist), nodeid: 4149, path: /usr/bin/exsh, mode: 4777
```

Figure 6.4: A Report Generated by the Execution Monitor

the password file. The scenario was simulated manually. On one window, the attacker logged on as the superuser and executed the command *vi /etc/shadow* to modify the shadow password file. Meanwhile, the author logged on as a normal user on another winder and executed *passwd* to change the password. After *passwd* obtained the old password and the new password and started modifying the password file, the execution monitor detected the violation.

Figure 6.5 shows the report generated by the execution monitor.

```
VIOLATION detected Tue May 14 16:47:27 1996 + 0.884425000 sec
------------------------------------------------------------
Tue May 14 16:47:27 1996 + 0.830000500 sec
open_r, (5379, 768, 0, passwd), nodeid: 4251, path: /etc/shadow
```

Figure 6.5: A Report of the Synchronization Violation

The execution monitor detected the violation approximately 0.05 seconds after *passwd* opened the shadow password file.

Chapter 7

Discussion and Future Work

7.1 Discussion

We presented a new approach to intrusion detection: specification-based monitoring. We identified aspects of program behavior that are security-relevant and developed a formal policy language for describing the desirable behavior of programs. In addition, we developed a prototype specification-based monitoring system that is able to detect attacks exploiting the vulnerabilities of privileged programs in Unix. Below we discuss our work and compare it with other approaches.

7.1.1 Limitations

Similar to other monitoring approaches using audit trails, the capability of our approach is limited by the information contained in the audit trails. In current auditing systems, audit trails contain only limited information about events that occur. For example, an audit record typically indicates an open operation on a file for writing, but not the data written to the file. The limited information restricts the capabilities of the monitoring system. For instance, a simple but effective trace for the *passwd* program is that a particular *passwd* invocation should modify only the password data corresponding to the invoker in the password file. However, our execution monitor cannot enforce this trace policy because it cannot obtain the information solely from the Solaris BSM audit trails. Nevertheless, active diagnosis of the system, such as reading the contents of the password file after it is being modified, may be used to obtain supplementary information. The active diagnosis can be encoded as semantic actions in the grammar rule in an on-line analysis.

The violations that can be detected by the execution monitor depend on the set of trace policies used. In our approach, the set of trace policies restrict the behavior of programs beyond that enforced by the protection mechanisms built in the operating system. The trace policies we specified for setuid root programs in Unix are able to catch all attacks to such programs known to us. Obviously, one cannot guarantee that the trace policies can catch all possible attacks to these programs. Further research is needed to identify a methodology for determining whether a given set of trace policies is adequate for a system. On the other hand, we can write trace policies of programs that can be used to detect all known attacks that exploit their vulnerabilities.

Similar to other intrusion detection approaches, our approach is not a panacea to the intrusion problem. Attacks that do not produce state changes (e.g., passive wiretapping) or that require massive behavioral analysis can not be detected. This approach also assumes integrity of the audit data. Thus, attacks that involve spoofing, which produce the same audit trail but from a different source, may not be detected.

Last, our approach is a detection approach, which raises an alarm when an intrusion occurs. Our approach can, at best, detect security violations, and it is up to the security officer to deal with each detected violation.

7.1.2 Comparison to Misuse Detection

In misuse detection, the goal is to identify actions (or misuse signatures) that represent intrusive activities and to check for occurrences of these actions in the audit trails. Misuse signatures are described by expert-system rules, state-transition diagrams, and patterns in Petri networks.

The specification-based approach can be thought of as the dual of misuse detection. A misuse signature describes undesired behavior in a system while a trace policy describes the desirable behavior of a subject. In particular, our approach focuses on the desirable behavior of security-critical programs (e.g., privileged programs) in a system. On way to specify the desirable behavior of a program is to enumerate the operations the program needs to perform in order to accomplish its function.

A misuse detector matches a signature with the whole system trace to identify intrusions while an analyzer in a specification-based execution monitor parses the trace of a subject to determine whether the subject conforms to a trace policy. Although matching of different signatures can be distributed over multiple hosts, each misuse detector requires the whole system trace. In a distributed system with many hosts, the whole system trace would be huge and cannot be processed by a misuse detector in real time. In our approach, an analyzer monitors the execution of a particular subject, only the audit records associated with the subject are needed by the analyzer

In misuse detection, signatures are mostly driven by previous attacks or known vulnerabilities. Although possible, it is not intuitive to encode a policy as misuse signatures. Our approach is more policy-oriented; a trace policy for a subject is specified based on the functionality of the subject and the system security policy. Therefore, it can succeed in catching attacks that exploit unknown vulnerabilities in programs. In general, given an intrusion represented by a sequence of actions, we can write a PE-grammar that rejects the traces of a subject that contain this sequence of actions.

7.1.3 Comparison to Type Enforcement

In the type-enforcement approach [8], accesses to objects by a subject are restricted by a type-enforcement policy based on the domain of the subject and the type of the object. Each subject is running in a domain and each object is assigned to a fixed type when it is created.

The Domain and Type Enforcement (DTE) approach [3] applies type enforcement to a Unix system. It takes the process hierarchy and the file hierarchy of current systems into consideration. The type enforcement policy is specified in a DTE language. Each domain is associated with one or more entrance programs, when executed by a subject/process switches, will move the subject/process to that domain. In effect, the type enforcement policy restricts the access of a process based on the program it is executing.

The DTE approach is similar to our approach as it further restricts the accesses of a program. In general, a DTE policy can be specified by a set of trace policies in our approach. A trace policy can specify the valid accesses of a program, but also the valid ordering of the accesses. Therefore, a trace policy is more expressive than a DTE policy regarding the specification of the behavior of a program.

One difference between DTE and our approach is that DTE is a preventive approach.

Operations performed by a program during execution that violate the DTE policy are denied by the DTE subsystem, while ours is a detection approach that raises a warning when a violation occurs. Nevertheless, one can incorporate our parsing mechanism into a reference monitor that prohibits any operations that are not accepted by the parser (i.e., those operations that are in violation of a trace policy).

7.1.4 Experience in Specifying Trace Policies

We specified trace policies for approximately 15 privileged programs in Unix. These programs include setuid root programs and server programs normally executed by root. Our experience in specifying the trace policies of these programs are summarized below.

Identifying the legal operations of a program is in general a good approach to specifying a trace policy for a program. It is not difficult to enumerate the operations that a privileged program needs to perform in order to accomplish its function and capture valid sequences of these operations in a trace policy. Since privileged programs are designed to accomplish specific functions, their expected behaviors are normally simple. Most known attacks that exploit vulnerabilities in privileged programs cause the programs to perform operations other than those intended. However, this approach does not work for all privileged programs. For some programs, the allowable operations may be a function of the input to the program, such as the command line parameters, interactive input, or files accessed by the program. With the exception of the command line parameters, the data read by a program is not generally available from the audit trails, or it is too application-specific to be dealt with generally. For programs such as *login* and command interpreters, it is not feasible to specify just the system calls and files accessed by these programs. For instance, the *login* program authenticates a user and gives him a shell. We cannot know from the accesses performed by the program whether the authentication is performed correctly, for example, the login program could contain a Trojan horse such that certain users are incorrectly authenticated. Our system will not detect such a vulnerability as it would not involved accesses inconsistent with the specification we would write for this program.

A Trace Policy for a program may be site-specific. The desirable behavior of a program at a particular site can be different from that intended by the designer. Sometimes, the differences are minor, such as the locations of specific files. In this case, the same trace policy can be used for different sites, and the environment variables in the policy parameterize the policy for different sites. Sometimes, the differences can be significant. For example, the *http daemon* (or *httpd*) is allowed to access the files corresponding to the locations specified by the browser connected to it. However, a system administrator may want to restrict the accesses of his site's daemon. Therefore, two different trace policies for the same program may be used at two different sites.

7.2 Conclusions

This dissertation described a new approach to security monitoring. The main idea is to specify the desirable behavior of security-critical programs in a system and to monitor their executions for behavior inconsistent with the specifications. The idea of attempting to specify the desirable behavior of programs (the *specification-based* approach) is unique among other intrusion-detection approaches. It is a more policy-driven and systematic approach to monitoring the security of a system than current approaches.

We use grammars as specifications of the valid traces of programs, which has the advantage that formal languages are a mature discipline and many results can be readily applied. We developed a language framework, *parallel environment grammars*, for specifying trace policies. Such a grammar is able to describe a parametric, context sensitive language (a trace policy), yet has an efficient algorithm for recognizing the generated sentences in practically important cases. Our language framework permits specifications of many different kinds of trace policies in a precise and compact way. We presented a parallel parsing method for recognizing the generated traces efficiently based on recursive-decent parsing.

We presented a distributed design of a monitoring system that combines decentralized analysis, distributed audit collection, and data reduction. Our design minimizes the amount of data that needs to be transferred across the network. With the language, we specified the trace policies for a number of programs (privileged and non-privileged) in Unix. We are able to detect known attacks that exploit vulnerabilities in privileged programs. The monitoring system also has the potential to detect attacks that exploit unknown vulnerabilities in such programs.

7.2.1 Contributions

The work in this dissertation makes two major contributions. The first is the approach of specification-based monitoring, a novel approach to intrusion detection. The approach pinpoints one of the major security problems – compromises caused by security flaws in security-critical programs. A major breakthrough of this approach is its potential to detect unseen attacks in computer systems. The approach leads itself naturally to a decentralized analysis system, which enhances the scalability of the monitoring system. It is also the first monitoring approach that addresses security problems due to synchronization in concurrent programs. Second, the parallel environment grammar developed in this dissertation is the first grammar that is able to describe merges of multiple streams. The grammar can describe valid synchronization behavior among concurrent processes. We envision the use of parallel environment grammars as a specification tool for describing parallel computations.

7.3 Future Work

This section discusses future research our dissertation suggests.

7.3.1 On the Specification-based Approach

Our execution monitor uses system audit trails to monitor the activities of programs. As discussed in Section 7.1, current audit trails do not provide sufficient information about the data being read or written in an operation. In many cases, this limitation affects the capabilities of a monitoring system that uses the audit trails. Therefore, other data sources are useful for monitoring the activities of a system. Network audit trails as well as application-level audit trails should be considered. For example, the presence of a system call does not reveal the details of a command that a server receives from a client. Therefore, information from a network audit trail may be valuable. To use and trust application-level audit trails, further research is needed on protecting and managing them. Also, we need to combine the information obtained from different audit trails to form a single operation sequence.

The specification-based detection approach can also be applied to monitor network components or network services that are relevant to security, such as domain name services (DNS), network file systems, and routers. For instance, the specification of the desirable security-relevant behavior of domain name servers can be written and the actual behavior of these servers can be monitored for violations of this specification. For example, data received by a DNS server must be consistent with the data provided by the authoritative servers; the security-relevant behavior is related to the content of the messages transferred between the servers. To monitor these components, network audit trails are needed because system audit trails normally do not reveal the content of a message being sent or received from a server process.

7.3.2 On the Implementation and Testing

We presented a distributed design of an execution monitor, but we implemented a prototype for only a single host. Also, the analyzers are developed manually from the corresponding security specifications following the approach described in Section 6.1.3, a tedious and time-consuming process when the security specifications are lengthy. Nevertheless, the implementation serves as a proof of concept of our approach.

One future effort is to implement a real distributed monitoring system based on our design. The distributed monitoring should work on a large distributed system and have efficient communication mechanisms. It should have a parser generator that translates a parallel environment grammar automatically into the corresponding parallel hyperparser.

In addition, more testing of execution monitoring is needed. In particular, it is important to evaluate the performance of the monitoring system in a very large distributed system. The test should clarify how large a system the monitoring system can handle. The performance evaluation should include CPU time, memory requirements, network bandwidth consumed by the system, and the response time of normal users. Criteria for distributing the load of the analyzers among different machines are also needed.

7.3.3 On Applications to Other Areas

We developed a language framework for specifying trace policies of programs. Although we employed a monitoring approach to enforce the trace policies, preventive approaches can be taken to enforce the trace policies as well.

Parallel environment grammars can be used to specify the behavior of parallel processes. We envision that these grammars can evolve as tool for functionally specifying distributed programs.

7.3.4 On Reasoning about the Security of the System

Our approach enables a system administrator to specify trace policies for subjects (e.g., users or programs) and monitor their executions for violations of the policies. Subjects in the system are further restricted by the trace policies, in addition to the basic policy enforced by the built-in protection mechanisms. Given a set of trace policies for subjects, the overall policy of the system could be investigated. Such research can guide us in the specification of the trace policies for different subjects and in determining the degree of security in a system.

A Unix security model that permits reasoning about the security should be developed. The model should capture the relevant states of a Unix system, the basic protection mechanisms, (i.e., the permission bits), and the trace policies of subjects. It should also permit the determination of whether security can be compromised, and of the ways in which it can be compromised.

Bibliography

[1] D. Anderson, "Safeguard final report: Detecting unusual program behavior using the NIDES statistical component," Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[2] J. P. Anderson, "Computer security threat monitoring and surveilance," Technical report, James P. Anderson Co., Fort Washington, PA, April 1980.

[3] L. Badger *et al.*, "Practical domain and type enforcement for UNIX," in *Proceedings of the 1995 Symposium on Security and Privacy*, (Oakland, CA), May 8-10, 1995, pp. 66–77.

[4] R. W. Baldwin, "Rule based analysis of computer security," Technical report MIT/LCS/TR-401, Laboratory for Computer Science, Mass. Inst. of Tech., Cambridge, MA, March 1988.

[5] D. Bell and L. LaPadula, "Secure computer systems: Mathematical foundations and model," Technical report M74-244, The MITRE Corp., Bedford, MA, 1973.

[6] K. Biba, "Integrity considerations for secure computer systems," Technical report ESD-TR-76-372, Air Force Electronic Systems Division, Hanscom AFB, MA, 1977.

[7] G. V. Bochmann, "Semantic evaluation from left to right," *Communications of the ACM*, vol. 19, pp. 55–62, Feburary 1976.

[8] W. E. Boebert and R. Y. Kain, "A practial alternative to hierarchical integrity policies," in *Proceedings of the 8th National Computer Security Conference*, (Gaithersburg, MD), October 1985, pp. 18–27.

[9] D. Clark and D. Wilson, "A comparsion of commercial and military computer security policies," in *Proceedings of the 1987 Symposium on Security and Privacy*, (Oakland, CA), April 27-29, 1987, pp. 184–194.

[10] Computer Emergency Response Team (CERT), Pittsburgh, PA, *CA:94-01 Ongoing Network Monitoring Attacks*, Feburary 1994. Available from ftp://info.cert.org/pub/cert_advisories/.

[11] Computer Emergency Response Team (CERT), Pittsburgh, PA, *CA:95-04 NCSA HTTP Daemon for UNIX Vulnerability*, February 1995. Available from ftp://info.cert.org/pub/cert_advisories/.

[12] Computer Emergency Response Team (CERT), Pittsburgh, PA, *CA:95-13 Syslog Vulnerability - A Workaround for Sendmail*, October 1995. Available from ftp://info.cert.org/pub/cert_advisories/.

[13] D. Crowe, "Generating parsers for affix grammars," *Communications of the ACM*, vol. 15, no. 8, pp. 728–734, 1972.

[14] D. Denning, *Cryptography and Data Security.* Menlo Park, CA: Addison-Wesley Publishing Company, 1982.

[15] D. E. Denning, "An intrusion-detection model," in *Proceedings of the 1986 Symposium on Security and Privacy*, (Oakland, CA), April 7-9, 1986, pp. 118–131.

[16] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[17] J. Doak, "The application of feature selection: A comparision of algorithms, and the application of a wide area network analyzer," Master's thesis, Dept. of Comp. Sci., University of California, Davis, 1992.

[18] M. W. Eichin and J. A. Rochlis, "With microscope and tweezers: An analysis of the internet virus of novermber 1988," in *Proceedings of the 1989 Symposium on Security and Privacy*, (Oakland, CA), May 1-3, 1989, pp. 326–343.

[19] D. Farmer and E. Spafford, "The COPS Security Checker System," in *Summer USENIX Conference*, (Anaheim, CA), June 11-15, 1990, pp. 165–170.

[20] J. Frank, "Artificial intelligence and intrusion detection: Current and future directions," in *Proceedings of the 17th National Computer Security Conference*, (Baltimore, MD), October 11-14, 1994.

[21] T. D. Garvey and T. F. Lunt, "Model based intrusion detection," in *Proceedings of the 14th National Computer Security Conference*, (Washington, D.C.), October 1-4, 1991, pp. 372–385.

[22] B. Hebbard *et al.*, "A penetration analysis of the michigan terminal system," *ACM Operating System Review*, vol. 14, no. 1, pp. 7–20, 1980.

[23] A. Heydon, "Specifying and checking Unix security constraints," in *Proceedings of the 3rd USENIX Unix Security Symposium*, (Baltimore, MD), September 14-16, 1992, pp. 211–226.

[24] K. Ilgun, "USTAT: A real-time intrusion detection system for Unix," in *Proceedings of the 1993 Symposium on Security and Privacy*, (Oakland, CA), May 24-26, 1993, pp. 16–28.

[25] K. Ilgun, R. Kermmerer, and P. Porras, "State transition analysis: A rule-based intrusion detection approach," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, 1995.

[26] K. Jackson, D. DuBois, and C. Stallings, "An expert system application for network intrusion detection," in *Proceedings of the 14th National Computer Security Conference*, (Washington, D.C.), October 1-4, 1991.

[27] H. S. Javitz and A. Valdes, "The NIDES statistical component description and justification," Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1994.

[28] K. Jensen, *Coloured Petri Nets – Basic concepts I.* New York: Springer Verlag, 1992.

[29] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice Hall, 1978.

[30] J. Kim and E. Spafford, "The design of a system integrity monitor: Tripwire," Master's thesis, Department of Computer Science, Purdue University, 1993.

[31] C. Ko *et al.*, "Analyzis of an Algorithm for Distributed Recognition and Accountability," in *1st ACM Conference on Computer and Communication Security*, (Fairfax, MD), November 3-5, 1993.

[32] S. Kumar, *Classification and Detection of Computer Intrusions.* PhD thesis, Department of Computer Science, Purdue University, August 1995.

[33] B. W. Lampson, "Dynamic protection structures," in *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 35, (Montvale, NJ), AFIPS Press, 1969, pp. 27–38.

[34] B. W. Lampson, "Protection," in *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems*, 1971, pp. 437–443. Reprinted in *Operating System Review 8, 1* (Jan. 1974), 18-24.

[35] R. R. Linde, "Operating system pentration," in *National Computer Conference*, vol. 44, (Montvale, NJ), AFIPS Press, 1975.

[36] D. Longley and M. Shain, *Data and Computer Security: Dictionary of Standards, Concepts, and terms.* New York: Stockton Press, 1987.

[37] T. Lunt *et al.*, "A Real-time Intrusion Detection Expert System (IDES)," Technical report, Computer Science Laboratory, SRI International, May 1990.

[38] T. Lunt *et al.*, "A real-time intrusion detection expert system (IDES) - final technical report," Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Feburary 1992.

[39] NASA, Lyndon B. Johnson Space Center, Information System Directorate, Software Technology Branch, *Clips Version 5.1 User's Guide*, March 1992.

[40] P. Neumann *et al.*, "A provably secure operating system: The system, its applications, and proofs," Technical report CSL-116, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1980.

[41] P. Porras and R. Kemmerer, "Penetration state transitin analysis: A rule-based intrusion detection approach," in *Proceedings of the 8th Computer Security Application Conference*, (San Antonio, TX), Novermber 30 - December 4, 1992, pp. 220–229.

[42] M. Ruschitzka and L. Clevenger, "Heterogeneous data translations based on environment grammars," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1236–1251, 1989.

[43] J. M. Rushby, "Design and verification of secure systems," *Proceeding of the 8th Symposium on Operation System Principles, ACM Operating System Review*, vol. 15, December 1981.

[44] J. D. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, 1975.

[45] M. Sebring *et al.*, "Expert systems in intrusion detection: A case study," in *Proceedings of the 11th National Computer Security Conference*, October 1988, pp. 74–81.

[46] S. Snapp *et al.*, "DIDS (distributed intrusion detection system)–motivation, architecture and an early prototype," in *Proceedings of the 14th National Computer Security Conference*, (Washington, D.C.), October 1-4, 1992.

[47] E. H. Spafford, "The internet worm program: An analysis," *ACM SIGCOM*, vol. 19, no. 1, 1989.

[48] Sun Microsystem, *Man Pages: Rdist - remote file distribution program.*

[49] Sun Microsystems, *Sun Security Bulletin #120 - 135.*

[50] SunSoft, Mountain View, California, *Solaris SHIELD Basic Security Module*, August 1994.

[51] A. Tanenbaum, *Modern Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992.

[52] H. Teng, K. Chen, and S. Lu, "Adaptive real-time anomaly detection using inductively generated sequential patterns," in *Proceedings of the 1990 Symposium on Security and Privacy*, (Oakland, CA), May 7-9, 90, pp. 278–284.

[53] H. Vaccaro and G. Liepins, "Detection of anomalous computer session activity," in *Proceedings of the 1989 Symposium on Security and Privacy*, (Oakland, CA), May 1-3, 1989, pp. 280–289.

[54] L. Wall and R. L. Schwartz, *Programming Perl.* Sepastopol, CA: O'Reilly and Associates, Inc., 1992.

Appendix A

A Brief Note on the Audit Subsystem in Solaris

This chapter describes the audit subsystem in the SUN Solaris 2.4 Operating System (henceforth Solaris). The audit mechanism enables logging of security-relevant events that have occurred in the system. The log, or the audit trail, provides a history of system execution that enables an administrator to review the cause of a security violation, and to trace it back to the user accountable. The audit subsystem allows an administrator to select which activities to monitor. The selection can be finely tuned to select what to audit for each individual user. It provides utilities for administrators to inspect the audit trail and filter irrelevant data. All administration commands, e.g., enabling and disabling auditing, that are associated with auditing can be performed only by the superuser.

A.1 Basics

Auditing is enabled and disabled by starting and terminating the audit daemon. The audit daemon is a daemon process that runs in the background. It is responsible for reading audit data from the kernel and writing the data into audit files. The audit daemon runs as root. All files it creates are owned by root and readable only by root. The audit daemon itself is not being audited.

The system actions that are auditable are defined as audit events, specified in the audit_event file. Each audit event is identified by a unique audit event ID. There are two categories of audit events: kernel events and application-level events. Kernel events are generated by the kernel when kernel-level system actions such as system calls occur. Application-level events are events generated by applications, which represent the occurrence of abstract actions within an application. Kernel events have event IDs from 0 to 2047, while application events have event IDs from 2048 to 65535. There are 215 kernel events and 16 application level events configured. The definition of kernel events cannot be modified without changes to the kernel. Additional application events can be defined by modifying the file *audit_event*.

In general, most audit events are attributable to individual users who are accounted for their actions. However, some events are not attributable because they occur at the kernel interrupt level or before a user is identified and authenticated (e.g., the process running the login program or daemon processes).

Each audit event is also defined as belonging to an audit class or classes. When naming a class, one can simultaneously address all events belonging to the class, for instance, in selecting audit events to audit. The system allows a maximum of 32 audit classes to be defined, and 19 of them are predefined, including a class *null* which contains no events and a class *all* which contains all defined events. The mapping of audit events to classes is specified in *audit_event*. The class definition is kept in the file *audit_class*. New classes can be defined and existing classes can be redefined by modifying the file.

The activities to audit are preselected by setting up the system-wide audit flags in the file *audit_control* and audit flags of individual users in the file *audit_user*. Audit flags specify classes of event to audit. Specifically, one can specify in a class to audit both successful events and unsuccessful events, only successful events, or only unsuccessful events. The system-wide audit flags specify the classes of events to audit for all users in the system. By setting up a single system-wide audit flag, an administrator can control what to audit for all users. Yet, the administrator can modify what gets audited for individual users by setting up the user's audit flags.

Audit data generated are saved by the audit daemon in audit files. The system allows an administrator to specify a list of locations (actually directories) in which to store audit files. When the audit daemon starts up, it chooses the first directory in the list for the new audit file. When the current directory is full, the audit daemon closes the current audit file and creates a new audit file in the next directory in the list.

An audit-file name has the form starttime.endtime.hostname, indicating the period of time the audit data covers. The name of the current audit file is stored in the file *audit_data* and has the form starttime.nonterminated.hostname.

An audit file consists of a sequence of audit records ordered in increasing time of occurrence. An audit record describes comprehensively the occurrence of a single audited event. The information in the audit record is sufficient to identify who performed the action, which files (including the *inode*, and a full pathname) were affected, what action was attempted and whether it succeeded or not, as well as where and when it occurred. The type of information saved for each audit event is defined as a set of audit tokens. Depending on the nature of the event, an audit record can contains some or all the tokens defined.

A.2 Internal Components of the Audit Subsystem

This subsection describes the internal components of the audit subsystem that is related to our work. The components that are concerned with auditing are the kernel and the audit daemon.

The kernel generates audit data when an audit event belonging to preselected event classes occur. In addition, it also receives audit data from application programs (via system call *audit*). The audit data generated and received is inserted into an internal audit queue, which will be fetched and removed by the audit daemon. In normal situations, the audit daemon should be able to keep up with the generation of audit data. However, when the queue is full, the kernel may discard the audit data generated, or suspend the processes that are associated with the data, depending on the current audit policy. Internally, the kernel has an audit status flag, which indicates whether auditing is enabled. The kernel will generate audit data only if the flag is set. This flag should be set or reset only by the audit daemon.

The audit daemon is started by executing the audit daemon program. When invoked, the audit daemon first checks whether an audit daemon is already running to avoid having multiple daemons running at any one time. It then changes the kernel audit status to on, so that the kernel will generate audit data when security-relevant audit events occur. The daemon runs with the effective user ID set to *root*. The actions of the daemon are never audited.

Each process in Solaris has an audit user ID and an audit preselection mask. The audit user ID is not associated with any other user IDs (e.g., real user ID). A process acquires its audit ID at login time, and this audit ID is inherited by all child processes. Even if a user changes identity (by using su(1M)), all actions performed are tracked with the same audit ID. A process can read and modify its audit user ID using the *getauid* and *setauid* system calls, which are allowed only if the process's effective user ID is root. Normally, only the process that is running login programs should change its audit user ID.

The audit preselection mask is used to determine whether an event caused by a process is to generate audit records. When a user logs in, the login program combines the system-wide audit flags with the user-specific audit flags (if any) to establish the process preselection mask for the user's process. The process preselection mask is inherited by child processes. The process preselection mask enables efficient checking of whether an audit event associated with the process is to be recorded. (The checking consists of the logical *and* of the process preselection mask and the class mask of the event.) The preselection mask of a process can be modified by any root process using system call *auditon*. The superuser can change what to audit for a process by changing the preselection mask of the process.

A.2.1 Static and Dynamic Configuration

The set of configuration files specifies the static configuration of the audit system. The kernel internally stores the audit configuration, referred to as the dynamic configuration. At system startup, the kernel initializes itself from the configuration files. Changes to the configuration file require changing the dynamic configuration.

For example, if an administrator wants to audit every action by a user A who is not audited initially, he has to change the audit flags of user A as well as the audit preselection masks of all A's existing processes explicitly. Since the audit preselection mask of a process is initialized at login time, changes of the audit flags will affect the audit preselection mask of a user process that will be created at the next login.

A.2.2 Auditing Daemons

By default, system daemons (e.g., *inetd*, *rcpbind*, *nfsd*) are not audited because they are not associated with any user. These daemons are started by *Init*, and are usually run with audit

ID -2. In addition, they are normally started up before the audit daemon starts.

System daemons cannot be audited just changing the static audit configuration files. However, the actions of these daemons can be audited by explicitly changing the preselection masks of the daemon processes using system call *auditon*.

Appendix B

Audit Events Used by the Prototype

The following table describes the audit events used by the prototype execution monitoring described in Chapter 6. There are 250 audit events defined in Sun Solaris BSM Audit Subsystem, and 36 of them are used by the prototype.

AUDIT EVENTS		
AUE_EXECVE	AUE_EXIT	AUE_KILL
AUE_VFORK	AUE_FORK	AUE_FORK1
AUE_FCHOWN	AUE_CHOWN	AUE_SETREUID
AUE_FCNTL	AUE_IOCTL	AUE_SETREGID
AUE_FCHMOD	AUE_CHMOD	AUE_RENAME
AUE_STAT	AUE_LSTAT	AUE_FSTAT
AUE_OPEN_R	AUE_OPEN_RC	AUE_OPEN_RT
AUE_OPEN_RTC	AUE_OPEN_W	AUE_OPEN_WC
AUE_OPEN_WT	AUE_OPEN_WTC	AUE_OPEN_RW
AUE_OPEN_RWC	AUE_OPEN_RWT	AUE_OPEN_RWTC
AUE_CREAT	AUE_CLOSE	AUE_MKNOD
AUE_LINK	AUE_UNLINK	AUE_SYMLINK

Appendix C

Attributes of Operations

This appendix lists the attributes associated with the system operations in Unix in our design. For convenience, the attributes are grouped into 5 structures: A, S, F, P, and M. The attribute A denotes some general characteristics of the event.

- time : time of the operations
- event : event Id
- rval : return value

The attribute S denotes attributes of the subject involved in the event.

- prog : the program associated with the process
- ruid : the real user Id of the process
- pid : the process Id
- euid : the effective user Id of the process
- gid : the real group user Id of the process

The attribute F denotes attributes of the file involved in the event.

- ouid : the user Id of the owner of the file
- ogid : the group Id of the owner of the file
- pmode : the permission mode of the file
- nodeid: the inode number of the file

- fsid : the Id of the file system in which the file is situated

The attribute M denotes miscellaneous attributes of the operation.

- newowner : the new owner (for chown or fchown)
- newmode : the new permission mode (for chmod or fchmod)
- newpath : the newpath (for rename, link, or symlink)
- chpid : the child process Id (for fork)