# Rule Based Analysis of Computer Security

by

Robert W. Baldwin

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

# Rule Based Analysis of Computer Security

by

Robert W. Baldwin

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

# Rule Based Analysis of Computer Security

by
Robert W. Baldwin

Submitted to the
Department of Electrical Engineering and Computer Science
on May 1, 1987 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

## Abstract

Computers are rarely as secure as they could be. Users are lax or inconsistent in the ways they configure a computer's protection system, and these user mistakes often lead to serious security holes. For example, a privileged user might accidentally make his login initialization file publicly writable and that mistake could allow ordinary users to acquire super-user privileges. This sort of *operational security* problem is not caused by software bugs. It can happen even if all the computer's trusted programs behave according to their specifications. Operational security problems arise from complex interactions between the pieces of a computer's protection system.

This report describes a tool for improving the operational security of discretionary access control systems. The tool is a rule based system that knows about the behavior of the computer's software and the tricks used by attackers. The tool uses this knowledge to deduce the set of privileges directly or indirectly accessible to each user. Once the set of accessible privileges has been deduced, that set can be compared against a site specific access policy and any differences can be reported.

A prototype of this tool has been used at MIT to improve the security of its Unix computers. About twice each month the prototype identifies a database entry or file access mode that has been changed incorrectly and accidentally allows untrusted users to acquire super-user privileges.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Key Words and Phrases: computer security, expert systems, rule based systems, security analysis, operational security, Unix, Kuang

# Acknowledgments

I would like to thank my thesis advisor, Professor Stephen A. Ward, for providing an outstanding environment in which to complete this research. Professor Jerome H. Saltzer and Dr. George A. Michael provided guidance and encouragement which was very important during the early stages of research. Professor David K. Gifford helped me define the scope of this thesis. My thanks also go to Professors Randall Davis and Robert H. Halstead Jr. for their comments and advice as thesis readers.

This research project grew out of an attempt to display the structure of security systems using the innovative imagery found in William Gibson's book *Neuromancer* [9]. Turning this idea into a finished piece of research required many skills, most of which I learned while working with Dr. David D. Clark and Professor Fernado J. Corbato. I greatly appreciate the experience of working with them.

Noel Chiappa has my special thanks for teaching me the art of building large software systems and for being a friend. I want to thank Cliff Neuman, Tim Shepard, and those who prefer to remain nameless for many conversations on this project and security in general.

For providing diversions, my thanks go to the folks at East Campus.

I want to thank my parents for their love and support. I am deeply grateful for the patience, encouragement, and love of my wife, Maureen Baldwin.

# Table of Contents

# Table of Figures

# Table of Tables

# Chapter One

# Operational Security Problems

Computer systems are rarely as secure as they could be. Even when the operating system provides good protection mechanisms users may be lax or inconsistent in their use of the protection mechanisms leading to security holes. People make mistakes that cause to *operational security* problems. These problems are not caused by software bugs; they are caused by user errors. An operating system that has been certified to behave as expected (e.g., A1 certification [3]) can still have operational security problems. A certified system will do exactly what it is told to do, but the overall effect may not be what the users desired.

This thesis focuses on the computer aspect of operational security problems as opposed to the physical or administrative aspects of operational security. A typical operational security problem arises when one user allows a group of other users to have write access to his or her home directory. A user might grant this kind of access to create a project directory for a team of people, or it might be granted accidentally as part of making some file readable to the group. On many computer systems, granting this access allows all the members of the group to bootstrap their privileges to include the privileges available to the original user.

On Unix, write access to a user's home directory grants access to that user's privileges because of an interaction between the file system kernel and the interactive command interpreter (i.e., the shell or executive). The Unix file system interprets write access to a directory to mean that a user can delete and create files in that directory. When a user logs in, the Unix command interpreter automatically executes commands from an initialization file stored in the user's home directory. An attacker who has write access to the user's directory can delete the original login initialization file and replace it with a file that executes any desired commands. Those commands will be executed with the user's privileges next time that user logs in. Typically an attacker would choose commands that make it easier for him to get into that users account. For example, the commands might make all the user's files publically readable and writable.

This example illustrates the central focus of this research. It shows how the interactions between trusted programs can lead to operational security problems. The goal of this research is to develop a rule based system that can systematically analyze these interactions and report on any undesirable interactions that are possible on a particular machine.

Experience from this research shows that operational security problems are common and serious. One system with a sophisticated user community and a security conscious staff developed serious security holes about twice a month. The thesis of this dissertation is that rule based systems can be used to analyze the operational security of computer systems and identify protection decisions (e.g., the settings of the access controls on files and directories, or the contents of system tables) that lead to inconsistencies between the desired access policies and the actual access policies.

The major contribution of this research is an abstract model for representing the security relevant behavior of trusted programs (e.g., operating systems, command interpreters, mail systems, archive daemons, etc.). The model has enough detail that it can represent many of the sources of operational security problems, but not so much detail that exhaustively analyzing the interactions between trusted programs is infeasible. This thesis refers to security checkers based on this model as *Kuang Systems*[1].

The best way to show that Kuang systems can improve the security of information systems would be to build Kuang systems for several different information systems and test each one for its effectiveness at finding security holes. That would take a long time. This thesis takes a less time consuming approach that supports a weaker statement. The approach is to show that a Kuang system can solve both the easy and the hard problems which are known to arise in one information system. The philosophy behind this approach is that one learns more about a technique by finding its limits than by showing how it can solve a wide range of easy problems.

I chose to build a Kuang system for Unix. The advantage of choosing Unix is that there are published works ( [10], [25]) that identify the operational security problems which can plague a Unix system. If the Unix checker, called U-Kuang can find all the problems that have been published, then that is a good indication that Kuang systems can significantly improve the security of information systems for which the problems are known in advance.

The bulk of this thesis describes U-Kuang and the insights gained from running it on MIT's Project Athena computers. U-Kuang was able to detect all the problems that have been published, and by running it I learned that those problems are common and recurring. U-Kuang almost always found a problem when it was first run on a computer. After a learning period, the rate at which

[1] Kuang rhymes with twang. This research project was inspired by William Gibson's book *Neuromancer*, which won the 1984 Nebula and Hugo awards for best science fiction novel. Gibson's book describes innovative ways to visualize the structure of information systems. In particular, it describes a program called a Kuang Grade Mark 11 Ice Breaker (*ice* refers to the mechanisms used to protect access to information). The program described here is a greatly simplified version of that ice breaker program.

holes were created settled down to a few serious security holes per month (most holes granted super-user privileges to all users). The error rate was highest on systems where the staff did not care about security.

## 1.1 Existing Solutions to Operational Security Problems

Operational security problems appear to be caused by the complexity of modern protection systems. Modern information systems contain numerous trusted programs that implement the automatic management and flexible sharing mechanisms desired by computer users (e.g., deleting old scratch files, or invoking programs to process incoming mail). The kind of flexibility and automatic management desired by computer users can be seen by comparing a locked bookcase to the publishing office of the New York Times. Both systems have a library of material that must be available to the users, but it should be obvious why the access control solution which works for a bookcase does not generalize to solve the problems of controlling access within the publishing office. The publishing office is going to have complexity problems that do not exist with a bookcase. The fundamental functional requirements for computers have changed in much the same way that the functional requirements of a newspaper publishing house differ from that of a bookcase. Condensing, distributing and otherwise adding value to information is the main job of the publishing house. Storing and retrieving information is only a small part of the job. The publishing house has many different kinds of resources that must be controlled including the private notes of reporters, the long distance phone services, the staff in the research library, the printing presses, the personnel data, the photo library, the janitors that clean offices, the editors who finalize articles, and the money that pays for it all. Protecting diverse resources requires diverse protection mechanisms that can interact in complex ways. If users want a computer system that is like a publishing office, they will have to cope with complexity problems that did not exist in older computer systems.

One way to solve the complexity problem is to build new kinds of protection systems that are easier to understand. This approach is discussed in the background section (1.4). So far, no one has proposed a system that is easy to understand, easy to analyze, and flexible enough to meet the needs of modern computer users[2].

Currently system administrators cope with the complexity of their protection systems by enforcing a large number of rules-of-thumb. One such rule-of-thumb on Unix is to make sure that home directories can only be written by the person who logs into that directory. Other Unix injunctions deal with search paths, system tables, and programming conventions.

This rule-of-thumb approach helps improve security but it is not systematic and it often sacrifices many of the desirable features of the computer system. Another problem is that these rules are ad hoc. Rules are generated as administrators notice problems. They are not generated from a deep understanding of the interactions between trusted program and thus tend to be incomplete. There is no assurance that they cover even a small fraction of the possible security problems.

The rule-of-thumb technique for solving operational security problems tends to be severe. It often sacrifices the flexibility that initially attracted the users to a particular computer system. In order to simplify the rules, a wide range of desirable forms of sharing or automatic system management are forbidden. For example, a common rule for Unix is to insist that only the superuser has write access to the `/etc` directory because anyone who can write that directory can replace the password file, `/etc/passwd`. This rule throws away the possibility of having a group of trusted users who could use that access to

[2] Notice that two of the three goals can easily be achieved by just turning your computer off.

perform system maintenance. Without the restrictive rule-of-thumb, the administrator would need to check to see if untrusted users could indirectly acquire super-user privileges due to a security mistake made by the members of the trusted group. In order to avoid that time consuming task, the administrator enforces the strict rule-of-thumb.

The security checking tool described in this thesis helps users cope with complexity without sacrificing flexible sharing or automatic system management. The idea is to build a knowledged based model of each trusted program and then use a general inference algorithm to analyze the interactions between these programs which can occur on a particular machine. To the extent that the model is accurate and complete, the tool can compute a list of the privileges accessible to each user of this particular computer. That list can be compared to the desired accesses and any mistakes can be identified and corrected. This approach is tailored to the security requirements of the computer's users. They only need to sacrifice as much flexibility as is required to meet their specific security policy. They do not need to follow rules-of-thumb that were designed for the most conservative situation.

## 1.2 Kuang: Rule Based Security Checking

A computer system is operationally secure if it behaves as its users expect it to behave. All the desired operations should be allowed, and all the undesired operations should be disallowed[3]. The task of checking operational security can be divided into an analysis phase which determines the set of privileges available to each user and a comparison phase which checks the accessible operations against the desired operations.

[3] Not allowing desired operations is just as much a problem as allowing undesired access. In fact, my experience is that security holes are often created when users are trying to make some particular piece of information available to others. They end up granting more access than they intended.

This research focuses on the analysis phase, so a very simple language is used to specify the desired access policy. The policy specification is table that shows which users should be able to access each privilege. A privilege is any ticket that the operating system checks to decide whether to grant access to a protected resource. On Unix there are two kinds of privileges called user-ids and group-ids. The details of which privileges should be required to access each file or directory are not stated. Only the distribution of privileges is specified. The purpose of the analysis is to determine which users can directly or indirectly access each privilege.

The rule based security checker described here works by simulating the execution of a computer from power-up to shutdown and at each step of the simulation the attacker tricks are consulted to see if that step can be subverted to grant extra privileges to any user. U-Kuang starts with the knowledge that when a Unix system is booted, it executes the program `init` with super-user privileges. The checker knows that `init` executes commands found in the file `/etc/rc`. When that finishes `init` sets up login servers for the terminal devices specified in another file. U-Kuang knows that privileges are inherited by default when one program executes another, so U-Kuang examines the contents of the `/etc/rc` file on the computer being checked to determine which other programs will be run with super-user privileges. As the simulation continues, all system daemons that are activated via the `/etc/rc` file will be simulated. The logging in (and logging out) of all the users is handled as part of simulating the login server. Finally, the Unix `shutdown` program is simulated.

At each step of the simulation U-Kuang examines the sources of information that control the behavior of the simulated processes. Each source of information provides a toehold for a user to acquire additional privileges. The analysis then applies its knowledge about attacker tricks to determine which users can exploit these toeholds. The level of detail of the simulation must expose the toeholds without including so much information that an exhaustive analysis is infeasible. Chapter 3 describes the models for expressing information about the behavior of programs and the tricks attackers can use to exploit toeholds.

In summary, the approach described in this thesis for detecting operational security problems is to build a rule based system that embodies knowledge about the behavior of trusted programs and knowledge about privilege bootstrapping tricks. The inference engine for the system deduces the set of privileges that are actually accessible to each user of a particular system. The accessible privileges can then be compared against the desired access. The constructive nature of this analysis makes it easy to identify the mistakes that lead to violations of the access policy.

This knowledge based approach to checking operational security is more systematic and extensive than the rule-of-thumb approach currently used. This approach will make it easier for computer users to have both good system security and good tools for flexible sharing and automated system management.

## 1.3 Outline of Thesis

This chapter has introduced the general problem of operational security and pointed out that the focus of this work is on the analysis of online security problems (as opposed to physical or administrative aspects of operational security). The advantages of using knowledge based security checkers were presented, and the basic workings of a Kuang system were described.

The second chapter describes the range of security problems that Kuang systems can and cannot check. The particular problems addressed by the prototype security checker are described in that chapter. For example, Kuang

Systems could check for source integrity problems (e.g., verifying that only authorized users can modify the source files for programs that will run with super-user privileges), but the prototype, U-Kuang, does not check for this kind of operational security problem.

Chapter three presents the abstract model that is used to analyze the behavior of trusted programs. The model was created by generalizing the Unix protection system, but as discussed in chapter six, the model appears to be suitable for analyzing the major features of the TOPS-20 and the VAX/VMS operating systems. The abstract model highlights the toeholds that allow users to bootstrap their privileges. The model contains a simulation component that focuses on the sources of information that control processes, and a component that focuses on how a user can subvert those sources of information. Knowledge about trusted programs (e.g., the security kernel, the system daemons, the command interpreters, etc.) and knowledge about attacker tricks (e.g., how a user can extend his or her privileges) are embedded in both components of the model. The model includes high level abstractions like privileges, processes and files, and low level abstractions like search paths, disk partitions and swap spaces.

The fourth chapter describes how the model is used by U-Kuang to check the operational security of Unix systems. The major modules of U-Kuang are described, and the specific knowledge it has about trusted programs and attacker tricks is presented.

The experience gained by running U-Kuang is presented in chapter five. This chapter describes the experiments that lead to this thesis' conclusion that even security conscious users make mistakes that cause serious security holes. This chapter describes some tools that would make it easier to keep a Unix system secure. This chapter is not meant to be a criticism of Unix security. I believe any complex protection system will have similar problems.

Chapter six discusses the limitations and extensions of the rule based approach to security analysis. An important result presented in this chapter is that a rule based model can be used to answer general questions about a security system. For example one question that can be answered is whether conspiracy is a problem on Unix computers. That is, can two users working together acquire greater privileges than the union of the privileges they can acquire working alone? If they can, then the checker must analyze the implications of different conspiracies of untrusted users. Consider the protection system of bank vaults. Vaults cannot be opened without conspiracy. Two people must work together to open the vault. Neither person can achieve access to the vault working alone. Within U-Kuang's model of Unix, conspiracy cannot lead to privileges that would not be accessible to a single user.

The last chapter presents the general conclusions of this research: that operational security problems are serious and that rule based systems are a promising framework for solving these problems.

## 1.4 Intellectual Background

This research project applies expert systems technology to solve the long standing problem in computer security of ensuring the operational security of a computer. The existing solutions to this problem were discussed in section 1.1. This section discusses the intellectual background that helped shape the solution presented in this thesis.

This research draws on results in the fields of computer security and artificial intelligence. Work on protection mechanisms and on security models is used to design a language for describing the behavior of pieces of a protection system. Early AI research on planning and searching systems is used to develop the system for analyzing the interactions within a complex protection system.

Later work on knowledge based systems provide the syntax and algorithms for representing and manipulating knowledge about security systems.

The expert system used in this project is a goal-directed backward-chaining planning system. The planning system is not novel, it is much like NOAH [21]. The novel aspect of the planning system is organizing information about the behavior of the pieces of a protection system around attacker-oriented goals. The hard part of any rule based system is developing the correct abstractions for expressing knowledge in the problem domain. Thus the main intellectual contribution of this thesis is the abstract model of computer security systems that is presented in chapter 3.

Within the computer security field this research focuses on analyzing the behavior of large protection systems. It is assumed that the security system of a flexible computer system will consist of a large number of trusted programs outside of the operating system. Further, this project assumes that each piece behaves according to its specification, whereas much of the current work in computer security focuses on ensuring that a piece of a security system conforms to its specification. Given these assumptions, the remaining question is whether the protection system as a whole has been told to behave in a manner that is consistent with the users' security requirements.

Finally, this project explores a new meaning for security requirements. Most work defines the security requirements as a few general statements that govern information flow and information integrity. This project explores the advantages of fine grain security requirements that can be specified by the users of each computer.

### 1.4.1 Protection Models and Analysis

Models for protection systems enter this research at two levels. First, the models provide a way of viewing knowledge about protection systems and thus they suggest natural ways to represent the rules describing the behavior of pieces of a protection system. Second, they suggest a language for expressing fine grain security requirements.

The most general discretionary protection model is the access matrix described in [13], which generalizes the access control list model and the capability model [23]. The problem with all discretionary models is the difficulty of enforcing global policies such as restricting the flow of information between different users. To deal with this problem, both the access matrix model and the capability model have been extended to embody information flow requirements resulting in the model proposed by Bell & LaPadula [1] and the lattice model of Denning [6]. The Bell & LaPadula model has been further extended to incorporate information integrity requirements [7]. Unfortunately these extended models restrict the flexible information sharing mechanisms that were present in the original models. The approach explored in this thesis attempts to achieve the flexibility of the original models as well as the global policy features of the extended models.

Very little research has been done on analyzing the configuration of protection systems. Early work by Harrison, et al. [11] presented a method for modeling protection systems and showed that deciding whether an attacker could gain access to a particular object is like deciding whether a grammar is unambiguous. No single procedure can decide the accessibility question for all protection systems that can be expressed within their model, and there are protection systems that can emulate Turing machines so for those systems no decision procedure exists. However, their paper presents one model which does have a decision procedure. They state that this model is too simple to be of interest, but in fact the attacker-oriented rules used in this research generate an example of this simple model.

The attacker-oriented rules can be viewed as defining a set of commands for an abstract protection system. Each command examines the state of an access matrix to decide if it can be applied, and if so, it adds at most one token to the access matrix. The commands never add subjects (users) or objects (files) to the abstract access matrix, even though in a real Unix system applying the rule might involve creating a new file that takes the place of an existing one. Since the access matrix cannot grow, a simple counting argument shows that a decision procedure must exist.

The problem of indirect access to objects has been studied with the take-grant model for capability systems as in [23]. A variant of the take-grant model, called the schematic send-receive model, can be analyzed in linear time if ordinary users cannot create other users with greater privileges [22]. Neither of these models embody the complexity of protection systems that consist of several pieces of trusted software, so the results have limited applications to the real computer systems considered in this research.

### 1.4.2 Protection Mechanisms

One goal of this research is to describe how to build rule based systems that can analyze complex protection systems. For that reason knowledge about existing computer protection systems is relevant. One family of real computers has tried to meet the needs of the military computing environment and they tend to have a rich set of protection mechanisms. This includes ADEPT-50 [24], Multics [18], and most recently the Honeywell SCOMP [12]. This family of systems can enforce some global security requirements such as information flow control, but they do not address the problem of ensuring a match between the settings of the security configuration and the detailed security requirements of each site.

The classic example of a capability system is Hydra [26]. It has a general model that allows ordinary users to create capabilities to access objects and capabilities to create capabilities. A system like Unix could be viewed as a capability system, but it lacks the generality present in Hydra.

The range of commercial computer protection systems studied in this project has Unix [10] at one end representing simple access control list security, and VMS [5] at the other representing a complicated cross between list and capability protection. The VMS model also includes objects with diverse abstraction such as files and page tables. Due to lack of availability, I ignored retrofitted security systems like RACF for IBM systems.

### 1.4.3 Rule Based Systems

This project draws on research in the area of expert systems to build a security checker. Expert systems have used several different methods for representing knowledge and for drawing inferences from that knowledge. Levesque and Brachman have summarized these different approaches and pointed out a fundamental tradeoff between the representation of the knowledge base and the kinds of inferences that can be easily made from the knowledge base [14]. The prototype security checker for Unix demonstrates that simple goal-oriented rules and a backward-chaining planning system can analyze an interesting range of security problems.

Early AI research developed hierarchical planning systems like NOAH [21] which are general enough to analyze a protection system once the system has been properly described. More recent AI research on rule based systems has developed different mechanisms for representing facts and procedures for reasoning with facts. For example, an early Unix security checker had twelve rules concerning programs controlled by databases; the final U-Kuang system had just two general rules about how databases can control programs and then it had several facts about what programs are controlled by which databases. Separating facts from the techniques for using those facts lead to a simpler knowledge base that was easier to understand and extend. The information structuring features of U-Kuang's rule based system were selected from several modern expert system shells (IKE [19], KRL [2], and KEE [8]).

# Chapter Two

# Scope of the Solution

Operational security covers a wide range of areas so it is important to define the specific goals and problem areas addressed in this research. This chapter presents the problem areas that U-Kuang addresses by describing the general areas, and within those areas, which problems are handled by U-Kuang. The first section lists the kinds of operational security holes that Kuang systems could find and the particular problems that U-Kuang finds. The second section presents a functional specification of Kuang systems in general and U-Kuang specifically. The third section describes different modes of operations for security checkers and the particular mode of operation for U-Kuang. The last section discusses policy specification languages and presents the simple language used by U-Kuang.

## 2.1 Operational Security Problems

Operational security problems are quite different from the problems that arise from design or implementation mistakes. Operational problems arise from the incorrect operation of the computer. They can exist even if the design and implementation have been certified to conform to each each and to some global model of the user's security requirements. This study of operational security problems assumes that the trusted programs behave as expected. The question it asks is whether the security system has been configured to enforce the desired security policy. That is, are there unexpected interactions between the pieces of the security system that allow users to perform undesired operations?

This section describes different kinds of operational security problems and identifies the ones that are handled by the prototype security checker. In general a Kuang system can check for any security hole that involves changing a file or table on the computer system. A list of operational security problems that do not fit this paradigm are presented in figure 2-1. The kinds of problems that do fit this paradigm are listed in figure 2-2. The extent to which U-Kuang checks for these problems is discussed below.

- Wire tapping. Communication channels are secure.
- Unlocked machine. Physical access to the system is well controlled.
- Bad passwords. The users have chosen good passwords.
- Authentication Integrity. The databases and programs that are used to authenticate users have not been tampered with.
- Installation Integrity. The people and data sources that update the system are trusted.

**Figure 2-1:** Unhandled Operational Security Problems

Of the problems listed in figure 2-2, U-Kuang checks for indirect access, privilege bootstrapping, and resolution integrity. It handles a limited form of authentication integrity. It can report on all the users who could modify the authentication system, but it cannot tell if the system has been modified. It does not check for source integrity problems in general, but it does check one special case where a database that controls the mail delivery program is converted from text to binary form. Object integrity and installation integrity are not checked, and in general I believe it is hard for Kuang systems to check for these kinds of operational problems because they require knowledge that is hard to represent in the abstract model presented in Chapter 3.

- Indirect Access. Given the operating system's rule for changing access controls, be sure that only desired users can access an object.

- Privilege Bootstrapping. Prevent users from acquiring additional privileges by changing the controlling databases of trusted programs.

- Resolution Integrity. When a program is invoked make sure the desired executable object is used. This category includes trojan horses and search path attacks.

- Object Integrity. The executable image of a program corresponds to the correct source code.

- Source Integrity. Only authorized persons can change the source code or the object libraries that form trusted programs.

**Figure 2-2:** Operational Security Problems

It is worth emphasizing that this security analysis system does not and cannot find software bugs. For example, version 4.2 of Berkeley Unix had a bug in the implementation of shared code segments that made it possible for anyone to acquire super-user privileges. The security checker could not find this problem. One could build a rule based system that incorporates knowledge about software bugs but it would not be Kuang system since it would not use the abstract model presented in Chapter 3.

## 2.2 Functional Specification of U-Kuang

The simplest Kuang system can be viewed as a boolean function of two arguments. The function examines a *Security Configuration* and an *Access Policy Specification* and returns `True` if the security configuration is consistent with the access policy. It returns `True` if and only if the computer is configured to allow all the specified access and disallow all the unspecified access.

A computer's security configuration is the sum of all the information that controls access to information. This includes the file and directory protection modes which are used by the kernel and all the system tables and configuration files examined by trusted programs. For example, the login program runs with super-user privileges, so the databases it reads, like the password file and the directory containing the password file, are part of the security configuration. The contents of each user's login initialization file are part of the security configuration, since those commands are automatically executed with the privileges of that user.

U-Kuang has a very simple policy specification language. Section 2.4 discusses the general issues. The goal of the simple language is to allow a system administrator to specify the set of privileges (i.e., group-ids and user-ids) that can be accessed by each user. The specification is exhaustive; any access not explicitly granted should be inaccessible.

This simple language does not address the problem of specifying which privileges should be necessary to access each file. A more general language would allow an administrator to specify which users are allowed access to each file or group of files. The simple language lets an administrator specify the distribution of privileges among the users, but there is no way to make sure that files containing sensitive information are protected by the correct privileges.

The primary inputs to the U-Kuang program are the security configuration and the desired access policy for the computer being analyzed. These inputs are different for each computer analyzed. The facts and rules that describe the privileged programs running on the computer (including the operating system) form a secondary input to U-Kuang. The secondary input would only change when the security model for Unix was changed or refined.

The output of U-Kuang is a list of plans that describe how the security configuration allows users to violate the access policy. If the list is empty, then the site is secure against the attacks that U-Kuang detects. To the extent that U-Kuang has an accurate and complete model of the trusted programs that run on Unix machines, an empty list means that the site does not have any operational security holes. Only (and all of) the specified privileges are accessible to the specified users.

## 2.3 Modes of Operation

A Kuang system could be used as a stand-alone auditing program, or it could be integrated into the operating system. U-Kuang is a stand-alone security checker. It is a program that can be run periodically to detect human errors that lead to operational security holes. U-Kuang takes less than a minute to run, so it could be performed whenever a privileged user makes changes to system tables. The experiments performed for this research involved running U-Kuang weekly.[4]

The auditing program does not prevent operational security holes; it just detects them. To get better security, a Kuang system could be integrated into the security kernel. Changes to the security configuration would be grouped into atomic transactions that are reviewed by the Kuang system and only transactions that leave the system in an acceptable state would be applied. This approach would only be feasible if the rule based system was fast enough.

[4] On one undergraduate machine the program was run every Friday morning so holes could be fixed before the weekend. Eventually it was discovered that system crackers were running the program every Thursday evening.

## 2.4 Policy Specification Languages

The basic question answered by a Kuang system is whether a particular site is in a 'safe' state. The characteristics of a safe state are determined on a per site basis, so there must be a way to specify such a state. Basically, a safe state is one in which all the desired accesses to information are allowed, while all the undesired accesses are disallowed. The specification can be viewed as a statement of the policies for acceptable accesses. This section discusses policy languages and describes the language used by U-Kuang.

The policy specification can be viewed as a virtual access matrix. Like a conventional access control matrix [13], the matrix that has a row for each user and a column for each protected resource (e.g., files, mail queues, networks, and devices). Each cell of the matrix defines the set of operations that the user can perform on the resource. This access matrix is virtual in the sense that it does not include details about how each resource is named nor does it list the related operations that are required to perform a specified operation. For example, on Unix if a user should have `read` access to a file, he must also have `search` access to the directory containing the file. These details are left out of the virtual access matrix.

When a Kuang system compares a policy to a security configuration there are several possible outcomes. The policy itself might be inconsistent. In that case the checker should describe how the policy statements contradict themselves. A checker that did not generate a constructive statement of the inconsistency would not help the user identify the source of the inconsistency. Recall that security checkers are motivated by the desire to help users deal with the complexity of protection systems. Another outcome could be that the policy is not realizable on the given operating system. In that case, the Kuang system should point out rules in the protection model that contradict statements in the policy. For example, if the policy says that some directory owned by user `alice` should not be writable by `alice`, the checker should point out that on Unix the owner of a directory can always gain write access because the owner can change the access controls on the directory. Finally, a the system must say whether the site is configured to meet the policy and if not, it should describe a sequence of actions that a user could perform to violate the policy specification. U-Kuang only performs this last function.

There is a trade-off between the expressiveness of the policy language and the difficulty of detecting inconsistencies (both within a policy specification and between a specification and a security configuration). The focus of this research is security analysis, so U-Kuang's policy specification was chosen to be very simple. The specification is a table very much like the table in the Unix file `/etc/groups`. The table lists for each privilege (user-ids and group-ids) the users who are allowed direct or indirect access to that privilege. For example, a user who has access to super-user privileges can indirectly access all user-ids or group-ids and that must be explicitly specified.[5]

A great deal of research is possible in the area of policy specification languages. For example, it would be interesting to formalize the specification languages of different systems and then compare them in terms of expressiveness or completeness. Another project would be to find out what policies are required to solve security problems in different environments (schools, corporations, computer centers). Policies about default protections are particularly interesting because they express constraints on objects which do not yet exist. It is not clear how a rule based system could check a policy like "all files created by the `foobar` program in this directory should only be readable by `bob`." Operating systems are unlikely to help users implement detailed policies like that one. At least a security checking system could detect any existing violations of the policy.

[5] In practice the policy table is generated by macros from a slightly more convenient language.

The only protected resources considered in U-Kuang's policy specification language are Unix user-ids and group-ids. The specification identifies the users who should have access to each privilege[6]. In this simple language files are not considered protected resources. The set of privileges required to access each file is not part of the specification. Only the distribution of privileges to users is specified.

U-Kuang's policy specification is called a *Privilege Access Table.* The BNF for this table is given in figure 2-3. Terminal symbols are underlined. The table has a row for each privilege. A Unix system can have a user and group id with the same name, so the symbols `u.` and `g.` are used to distinguish them. Each row lists the users who should be able to directly or indirectly acquire that privilege. This specification is exhaustive. Any access that is not explicitly granted is forbidden. Further, all user and group privileges must be listed. It is a violation of the policy for the computer to have privileges that are not listed in the table.

```
pol-spec    ::= priv-spec   |  priv-spec pol-spec
priv-spec   ::= priv : usernames
usernames   ::= username   |  usernames , usernames
priv        ::= U. username   |  G. groupname
username    ::= string
groupname   ::= string
```

**Figure 2-3:** BNF for the Privilege Access Table

A sample specification is given in figure 2-4. Notice that there is a user-id and a group-id named `daemon` as well as a user named `daemon`. The super-user,

[6] A privilege is any ticket checked by the operating system to determine whether to grant access to a protected resource.

`root`, has access to all privileges. The users `alex` and `alice` can use each others user-ids. For example, their entries in the password file might state that they have the same user-id even though they have different home directories. Unix does not enforce a one-to-one correspondence between users and user-ids. The user `bob` has allowed `tom` to use his account. For example, `bob` might have put `tom` in his `.rhosts` file which allows `tom` to log into `bob`'s account without supplying a password. The details of how each form of access is grant are not important. The privilege access table just specifies the set of user who can acquire each privilege.

```
user.daemon : daemon, root
user.alice : alice, alex, root
user.alex : alex, alice, root
user.bob : bob, tom, root
user.tom : tom, root
group.guest : alex, alice, bob, daemon, tom, root
group.staff : alex, alice, root
group.daemon : daemon, root
```

**Figure 2-4:** Sample Policy Specification

# Chapter Three

# An Abstract Security Model

The hard part of designing any rule based system is choosing the vocabulary and structures that will be used to represent facts in the problem domain. The goal is to pick a simple representation that facilitates answering the desired questions. In the case of a Kuang system, the important question is which users can indirectly access each privilege (e.g., user-id or group-id). To answer this question the system must create an abstract model of a computer's security system that clearly identifies the interactions each user could exploit to achieve greater privileges.

The abstract model is the major intellectual contribution of this thesis. Once the model is understood, it is easy to see how operational security problems can be checked systematically. The model identifies the essential causes of these problems and describes how the analysis problem can be decomposed into simple steps.

The model was developed as a generalization of the Unix protection system, but it can be applied to other systems. Section 3.2 describes the relationship between concepts in the abstract model and concepts in both the Unix and VMS operating systems. By presenting the model explicitly, this chapter makes it easier to decide whether this approach to security analysis can be applied to other systems. To the extent that the model can represent the significant features of a system, this work can be applied directly.

The key idea of this model is to focus on the sources of information that influence the behavior of a process (a running program). If an attacker can change one of the sources of information, then the attacker can gain control of the process and thus acquire the privileges available to the process. This idea unifies the tricks that attackers use to extend their privileges. Every trick is an example of modifying a file, table, or database that crucially influences the behavior of some process. For example, adding commands to a user's login initialization file can be viewed as modifying one of the sources of controlling information for the process that executes that user's command interpreter.

The first section of this chapter describes how the analysis problem can be decomposed into three simple steps. That section describes the information passed between each step. Sections two, three, and four describe those three steps in greater deal and present the models that are used to represent knowledge about trusted programs and attacker tricks. The chapter ends with a summary of the models.

## 3.1 Decomposition of the Analysis Problem

The general problem handled by a Kuang system is checking the configuration of a security system to ensure that the desired access policy is being enforced. The system must analyze the configuration to produce a table called the *Privilege Access Table* (PAT) that shows which users can indirectly access each privilege. This table has the same format as the one that specifies the access policy (see section 2.4) so it is easy to check the security.

The PAT is computed in three steps. The first step uses knowledge about trusted programs and facts about the configuration of the system being analyzed to deduce the set of files and tables that allow an attacker to directly acquire each privilege. That is, the first step examines the behavior of the trusted programs that will run on the computer being analyzed to produce a list of toeholds for an attacker. The second step applies knowledge of attacker tricks to deduce all the techniques that could be used to directly exploit these toeholds. The last step performs a transitive closure operation to identify all the indirect relationships. The PAT is extracted from the matrix of indirect relationships. This decomposition is expressed in figure 3-1.

```
Analysis Steps                       Examples

    Model of      <--- Rules,        Facts
Trusted Programs       Facts         - /etc/init forks shell to
       |                               execute commands in /etc/rc
       |
       |  Table of                   Privilege Controlling Operations
       |  Privilege                  u.root: {modify(/etc/rc) ...}
       |  Controlling                g.staff: {modify(/etc/group) ...}
       |  Operations                 u.tom: {modify(/usr/tom/.login) ...}
       |
       V
    Model of      <--- Rules,        Facts
Attacker Tricks        Facts         - /etc/rc writable by {u.root}
       |                             - /etc writable by {u.root, g.staff}
       |
       |                             Op. Grant Matrix     1 2 3 4 5
       |  Operation                  acquire(u.root)   1| T     T T
       |  Grant                      acquire(g.staff)  2|   T     T
       |  Matrix                     acquire(u.tom)    3|   T T
       |                             modify(/etc/rc)   4| T     T
       |                             modify(/etc)      5|       T T
       |
       V
   Transitive     <--- Rules,        Facts
    Closure            Facts         - user tom has {u.tom, g.guest}
       |
       |                             Privilege Access Table
       |  Privilege                  u.root: { root, tom }
       |  Access                     g.staff: { root, tom }
       |  Table                      u.tom: { root, tom }
       |
       V
```

**Figure 3-1:** Steps in Security Analysis

The first two steps use different knowledge bases and they have different models for representing facts and rules. The details of these models are explained in sections 3.2 and 3.3. To highlight the purpose and function of each step the remainder of this section describes the interfaces between the steps. For U-Kuang, additional information is passed between the steps because it answers a more detailed question. Not only does it need to figure out which users can indirectly access each privilege, it must be able to describe how each privilege could be acquired. The descriptions make it easier to identify the cause of an operational security problem, but they do not add any new complications to the analysis process.

### 3.1.1 Privilege Controlling Operations

A key assumption of this research is that operational security holes are exploited by modifying files, tables, or other information that controls a process. This assumption defines the kind of security problems that are considered in this thesis. For example, one problem which is not considered is the problem of leaving the computer in an unlocked room. An attacker with physical access to the machine can often acquire privileges by using the console terminal to interrupt the normal power-up sequence of the machine. This sort of problem does not fit the key assumption. However, the problem of an attacker modifying the files or tables that specify the commands that will be executed when the machine reboots does fit the key assumption. Those commands are often executed with super-user privileges (i.e., on Unix these are a user-id privilege called `u.root` and a group-id privilege called `g.wheel`), and by changing these commands an attacker could create new accounts with any desired privileges.

The key assumption suggests that one important concept is the notion of a *Controlling-File* (CF). A CF for a process is any file, directory, table, program, etc. that can be manipulated to allow an attacker to acquire the privileges available to that process. A CF is a generalization that encompasses all the sources of information that can influence the behavior of a process. The concept of a *Controlling-Operation* (CO) for a process specifies both the CF and the kind of operation on the CF which is necessary to manipulate the process. For example, the Unix login program runs with super-user privileges and it is controlled by information in the password file, **/etc/passwd**. The operation **write(/etc/passwd)** is a CO for the login process and thus an attacker who can perform that operation can acquire super-user privileges.

In general there are read and modify COs. A specific operating system might have different forms of these operations like **append** or **delete**, and the available operations might depend on the type of controlling-file. The operations on directories could be different than the operations on files. U-Kuang models Unix CO's with three operations: **read(cf)**, **write(cf)**, and **replace(cf)**. The **read** operation has the conventional meaning for files. For directories it means that the attacker has the ability to search the directory for a given file name. The **write** operation means that the contents of the CF can be modified without changing the ownership of the file or directory. The ownership can be changed as part of a **replace** operation. For example, a attacker can perform the **replace(cf)** operation by deleting the original file and creating a new one to take its place. This trick usually changes both the ownership and contents of the CF.

The purpose of the first step in the analysis is to build a table of all the COs that grant access to each privilege. The table of *Privilege Controlling Operations* (PCO) is constructed using knowledge about the behavior of trusted programs which has been compiled into the security checker and facts about the contents of command files and tables which are read from the machine being analyzed. Basically, the execution of all the trusted programs is simulated and as these programs reference controlling information, COs are recorded for all the privileges available to the process. A sample PCO table is shown in figure 3-2.

A table of PCOs has a row for each privilege. Each row lists the set of COs that allow and attacker to control some process that runs with the given privilege. In figure 3-2, access to the super-user privilege, `u.root`[7], is controlled by the operations of writing the password file or reading the kernel memory (via the special device `/dev/kmem`). The password file is a controlling file for the login program, which runs in a process that has super-user privileges. The kernel memory is a controlling-file for all processes. In particular, it includes the terminal buffers for all users, so read access allows an attacker to watch other users type their passwords when they login.

```
u.root: {modify(/etc/passwd), read(/dev/kmem)...}
g.staff: {modify(/etc/group)...}
u.tom: {modify(/usr/tom/.login)...}
```

**Figure 3-2:** Sample Table of Privilege Controlling Operations

### 3.1.2 Operation Grant Matrix

The first step of the analysis process produced a list of the operations that would allow an attacker to directly acquire each privilege. For example, the first step used its knowledge about how a Unix system boots to deduce that a process with super-user privileges executes a program called `/etc/init` so it would add an entry to the table of privilege controlling operations stating that `modify(/etc/init)` grants access to the super-user privilege, `u.root`. The second step of the analysis applies its knowledge of attacker tricks to find all the ways that an attacker could perform controlling operations.

[7] Unix has user and group privileges. To distinguish a user privilege named `daemon` from a group privilege of the same name, the former is written as `u.daemon` while the later is written `g.daemon`.

The PCO table can be viewed as a list of goals that would help an attacker increase his privileges. The second step builds a list of subgoals that can be used to achieve those goals, and recursively, a list of sub-subgoals to achieve the subgoals, etc. The details of this inference process are described in section 3.3, while a description of the output of the process is described below.

The output of the second step is a boolean matrix that describes which goals are directly granted by each goal. The goals can be represented as operations that an attacker can perform on CFs or privileges. That is, the goals are `modify(cf)`, `read(cf)`, and `acquire(priv)`. The first two goals are exactly the same as the controlling operations that were introduced earlier. The third goal means that the attacker can run an arbitrary program in a process that has access to the `priv` privilege.

The notation was chosen to encourage the reader to think of `acquire(priv)` as the ability to perform the `acquire` operation on an object of type *privilege*. This perspective unifies privileges and CFs into a single framework of performing operations on objects. Knowledge about the relationships between privileges and files can be expressed in the same terms as knowledge about relationships between COs. For example, one relationship between COs that exists in Unix is that for all files, `f`, in the directory, `d`, the operation `modify(d)` grants the operation `modify(f)`. A similar relationship between privileges is that for each user-id privilege, `u`, that is a member of the group-id privilege, `g`, the operation `acquire(u)` grants `acquire(g)`.

In terms of these general operations, the purpose of the second step of analysis is to build a boolean matrix that describes what other operations are directly granted by each operation. The *Operation Grant Matrix* (OGM) defines a relation between operations that is reflexive (an operation directly grants itself), but not necessarily symmetric (operation `o1` directly grants `o2` does not imply that o2 directly grants o1), nor transitive (o1 directly grants o2 and o2 directly grants o3 does not imply that o1 directly grants o3). The OGM only defines direct granting relationships between operations. The third step computes the transitive closure of the OGM to produce the indirect relationships.

A sample OGM is shown in figure 3-3. The OGM is always square, so the labels for the $i^{th}$ column is the same as the label of the $i^{th}$ row. The rows and columns are numbered to facilitate reading labels. The symbol, T, is placed in the cell at the intersection of a row and a column if the row operation directly grants the column operation. An empty cell means that there is not a direct grant relationship between the operations. The first row of the OGM expresses the fact that the super-user privilege, u.root, directly grants itself, acquire(u.root), and directly grants the modify operation for all CFs (/etc/rc and /etc).[8] The last cell of the second row expresses the fact that the staff group-id privilege has write access to the /etc directory, so it grants the modify(/etc) operation. The third row indicates that anyone who can acquire the user-id privilege, tom, can acquire the group-id privilege, staff. The transitive closure operation will deduce that acquire(u.tom) indirectly grants acquire(u.root).

The information in the OGM could also be expressed by a directed graph. The nodes would be the operations. An edge would lead from one node to another if the first operation granted the second. In fact, U-Kuang used a graph to represent the OGM. Similarly the table of PCOs could be viewed as a graph showing which COs grant the acquire operation for each privilege. Thus both the input and output of the second step of analysis can be viewed as a graph describing the direct grant relationship between operations. The second step uses

[8] It is also true that the super-user privilege directly grants access to all other privileges by using the /bin/su command, but that knowledge is not included in this simple example.

```
                      1  2  3  4  5
acquire(u.root)    1| T        T  T
acquire(g.staff)   2|    T        T
acquire(u.tom)     3|    T  T
modify(/etc/rc)    4| T        T
modify(/etc)       5|          T  T
```

**Figure 3-3:** Sample Operation Grant Matrix

its knowledge of attacker tricks to expand the direct granting relationships that were deduced in the first step.

### 3.1.3 Privilege Access Table

The goal of a Kuang system is to compare the operations that each user can directly or indirectly perform against a specified access policy. In the case of U-Kuang the only operations that are specified are operations that acquire privileges. The output of the third and final step of the analysis is a *Privilege Access Table* (PAT), which has the same format as the table specifying the desired access policy. The table has a row for each privilege, and each row lists the users that should be able to access that privilege. Any access that is not explicitly permitted should be forbidden (see section 2.4 for details).

A sample PAT is shown in figure 3-4. There are two user-id privileges, `u.root` and `u.tom`, and one group-id privilege, `g.staff`. The two users, `root` and `tom`, have access to all three privileges. It is no surprise that the super-user, `root`, has access to all privileges. However, the system deduced that `tom` could acquire `u.root`, so he too can access all three privileges. The PAT does not describe the plan by which `tom` can acquire super-user privileges. In order to build plans, each step of the analysis needs to keep track of extra information, but the nature of the analysis is not changed.

```
u.root: { root, tom }
g.staff: { root, tom }
u.tom: { root, tom }
```

**Figure 3-4:** Sample Privilege Access Table

The third step computes the transitive closure of the OGM and extracts just the information that deals with privileges. The computed matrix defines the indirectly grants relationship between privileges. The desired result is a table that defines the set of users who can acquire each privilege. To compute that result, the third step needs to consult facts about which privileges are initially accessible to each user. These facts come from the computer's authorization database, which on Unix is kept in the files `/etc/passwd` and `/etc/group`.

## 3.2 Model for Trusted Programs

The model of trusted programs defines the vocabulary and structures that are used to express facts and rules about the behavior of programs. The focus of the model is on the sources of information that an attacker could manipulate to gain control of the processes that execute these programs. By controlling a process, an attacker can acquire all the privileges available to that process. This section describes how knowledge about programs is represented and how that knowledge is used to deduce the set of operations that directly grant access to each privilege.

The algorithm for making these deductions is similar to an algorithm for simulating the execution of the programs. The algorithm uses compiled-in knowledge about the behavior of programs (e.g., the fact that the `init` program executes commands in the file `/etc/rc`) and facts about the computer being analyzed (i.e., list of commands found in that computer's /etc/rc file) to deduce the new processes and programs that will run on the computer. This is not a true simulation since the order of execution does not need to be maintained.

At each step of the simulation, a small set of rules are used to generate a list of COs that could change the outcome of that step. These COs are added to the set of COs for all the privileges currently available to the process. These sets form the PCO table.

To motivate the model which is explained in section 3.2.1, table 3-1 lists the kinds of facts that must be represented by the model. The first fact in the table describes how Unix systems start execution. The important information is that the future action of a process with super-user privileges is determined by information found in a file named /etc/init. The second fact provides an explicit description of the important actions taken by the init program. In general programs that are compiled into machine instructions must be simulated using compiled-in facts about their behavior. Programs which are written as a series of user-level commands (e.g., /etc/rc) can be parsed directly by U-Kuang, so their behavior does not need to be represented by compiled-in facts.

The third fact is about the privileges available to a process. When a new process is created, it inherits the privileges of its parent. Combining this fact with the first two, the system can deduce that all the commands mentioned in /etc/rc will be executed with super-user privileges.

The fourth fact describes how one program, sh, converts a program name into the full name of the file that contains the instructions for the program. The details of how program names are resolved into file names are very important to the operational security of a computer. In particular, the fifth fact states that

1. When Unix boots it create a process with super-user privileges that executes the program in `/etc/init`.

2. The program `/etc/init` creates a new process running the program `/bin/sh` to execute commands in the file `/etc/rc`.

3. By default, a new process has the same privileges as its parent process.

4. The program `/bin/sh` uses a search path to resolve command names into executable files.

5. Any directory searched to find a file that provides instructions for a process is a CF for that process.

6. Any file that provides instructions for a process is a CF for that process.

**Table 3-1:** Sample Knowledge About Trusted Programs

any directory search to find a program is a CF for the process. If an attacker can modify that directory, he can substitute his own program for the intended one and thus can acquire all the privileges available to the process. Similarly, the sixth fact says that if an attacker can modify the file that contains the instructions, then he can gain control of the process.

### 3.2.1 Formal Model for Trusted Programs

The key feature of the model for trusted programs is the *process* abstraction. This abstraction is similar to the conventional operating system notion of a process. It represents the instantaneous state of a program. As the security checker simulates the behavior of trusted programs it creates process objects and modifies the state of these objects to model the behavior of the corresponding real processes.

A process object has three state components. The behavior of each

**Process**

- **Privilege state** - List of privileges available to the process.
- **Naming state** - Context for resolving names.
- **Control state** - Current actions plus list of remaining action.

**Figure 3-5:** Attributes of a Process Object

process is specified by a *control state.* The privileges available to the process are specified by a *privilege state,* and the naming environment used to convert names into disk addresses is specified by the *naming state.* As the analysis proceeds, the state of each process object is changed according to the list of actions specified in the control state. New processes are created and old processes are simulated to completion. The analysis ends when all processes have been simulated. To define the model precisely, the concepts used to define the state of a process are defined below. Figure 3-6 relates these concepts to those found in the Unix operating system. Figure 3-7 illustrates the same correspondence for the VMS operating system.

A *Privilege* is any ticket which grants the ability to perform operations on protected resources. For example, the only privileges on Unix systems are user-ids and group-ids. These grant access to all the files, directories and devices that exist in Unix. The VMS system has group and user ids but it also has 'privilege bits' which allow a user to perform operations like submitting batch jobs, and change the page-table (virtual memory map) for a process. All of these are privileges. The privilege state of a process is a list of privileges that a program could use.

The model does not specify the restriction on how the privilege state can change. For example, on Unix only a process with super-user privileges can change its privilege state independently from changing the program that it is

| Unix | Model for Trusted Program |
|---|---|
| Process | <privilege state, naming state, control state> |
| user-ids | Privileges |
| group-ids | |
| current-directory | Naming state |
| search-paths | |
| shell-variables | |
| programs | Scripts = lists of actions |
| shell-scripts | |
| control-tables | |
| create-process | Actions |
| resolve-file-name | |
| execute-command-file | |
| execute-program | |
| change-privileges | |
| change-directory | |
| change-shell-variable | |
| copy-file | |

**Figure 3-6:** Unix and the Trusted Program Model

executing. The model does not enforce this restriction. This detail about the behavior of Unix is expressed by the actions specified by programs. The model must be able to determine the privilege state of a process at each step, but it does not need include information about forbidden changes to the privilege state.

The *Naming state* is a table that describes how file and program names can be resolved into disk addresses. This abstraction covers concepts like the current directory of a process, the list of directories search to find a file, and logical names (e.g., $HOME or SYS$SRC). When a process takes an action that must resolve a file or program name, the current naming state of the process is examined to determine the outcome.

The *Control State* of a process is a current action plus a list of remaining actions call the *remaining script* for the process. Each step of the simulation

| VMS | Model for Trusted Program |
|---|---|
| Process | <privilege state, naming state, control state> |
| Batch-job | |
| user-names | Privileges |
| group-names | |
| privilege-bits | |
| current-directory | Naming state |
| search-paths | |
| logical-names | |
| programs | Scripts = lists of actions |
| command-files | |
| control-tables | |
| create-process | Actions |
| resolve-file-name | |
| execute-command-file | |
| execute-program | |
| set-privileges | |
| change-directory | |
| define-logical-name | |
| copy-file | |

**Figure 3-7:** VMS and the Trusted Program Model

performs the current action, and then extracts the next action from the head of the remaining script. A process is fully analyzed when its remaining script is empty.

The possible actions are listed in figure 3-8. They define the activities of a process that could lead to operational security holes. The first two actions allow a process to change its privilege and naming state. Again, the model does not incorporate details about which kinds of changes are allowed. The compiled-in actions or the actions parsed from files express these restrictions. The third action describes the creation of new processes. It includes the restriction that the initial privilege and naming states are the inherited from the parent process. If that state needs to change, the change must be modeled by explicit actions. For example, the first action in the script for a set-id program on Unix, or a VMS program installed with privileges, must be an action that changes the privilege state.

1. Change the privilege state according to a constant in the action.
2. Change the naming state according to a constant in the action.
3. Create a new process with the same privilege state and naming state as this process, but a new instruction state as specified by a constant in the action.
4. Append or insert scripts to the list of remaining scripts. The scripts are specified by either 1) a constant in the action, or 2) a list of scripts parsed from a specified controlling-file.
5. Copy one controlling-file to another.

**Figure 3-8:** Possible Actions for a Process

The fourth action in figure 3-8 describes how a process can add actions to its script of remaining actions. The new actions can be specified by a constant or they can be read from a controlling-file (e.g., program or shell-script). This feature gives the model general Turing capabilities and the possibility of infinite loops, but in practice, trusted programs have a straight-line behavior in terms of actions that are relevant to operational security holes.

### 3.2.2 Representing Facts About Programs

This section briefly describes how the model can be used to represent the knowledge about trusted programs that were presented in table 3-1. Factual knowledge is represented by assigning value to the attributes associated with objects. For example, there will be an object that represents the file `/etc/init`, and it will have a field that holds the list of privileges that grant direct write access to that file. Each privilege is itself represented by an object. The facts in knowledge base can be thought of as pre-initialized objects. Facts that are deduced are recorded by creating new objects or by filling in the attributes of existing objects. Other knowledge describes how to compute the values of attributes. This knowledge is represented by rules. The antecedent of each rule selects one or more objects with particular properties, and the consequent fills in the value of some attribute. To avoid including syntax details the representations are described in english.

The first fact in table 3-1 is that Unix systems boot by running the program `/etc/init` in a process with super-user privileges. This is represented by a compiled-in process object that starts off the simulation. The privilege state includes the super-user privileges, the naming state is empty, and the control state specifies a single action which is to read a script from `/etc/init` and add it to the list of remaining actions.

The script for `/etc/init` is expressed as a compile-in fact. This fact is stored in the knowledge base as a property of the object that represents the file `/etc/init`.

The third fact states that new processes inherit the privilege and naming state of the parent process. This fact is represented by the rules which create new process objects. There is a general operation to create new process objects in the knowledge base given the three components of its state, and each rule that calls this operation passes it the privilege and naming state of the parent process.

The next fact states that the command interpreter, `/bin/sh`, uses a search path to resolve the names of programs into the files that contain the code for those programs. This fact is represented by an attribute in the naming state of a process. The routine that reads and parses commands from files checks this flag to build the appropriate scripts. The resulting scripts identify both the program name and the label for the attribute that holds the search path used to resolve the program name.

The fifth fact states that any directory searched to find a file that provides new actions for a process is a controlling-file for the process. This fact is expressed by a rule that examines the current action of each process and if that action involves reading a script from a file, then the appropriate directories of the search path (if any) are added to the table of process controlling operations.

The last fact states that the file that provides a new script is also a CF for the process. This fact is also expressed by a rule that examines each action taken by a process.

## 3.3 Model for Attacker Tricks

The first step of the security analysis produces a table of privilege controlling operations. If an attacker can perform one of these operations, he can gain control of a process that is running with the specified privilege. An alternative view of the PCO is that for each privilege, `p`, the PCO specifies the CF operations that directly grant `acquire(p)`. Taking this viewpoint, the PCO table can be used to fill in the initial rows and column of the operation grant matrix.

The second step in the analysis uses knowledge of attacker tricks and facts about the computer being analyzed to add additional rows to the OGM and to fill in additional granting relations between the operations. The entries in the PCO can be viewed as a list of goals that an attacker would want to achieve, and the purpose of the second step is to find all the sub-goals that would help achieve the initial goals. The analysis is recursively applied to the sub-goals, and it stops when there are no new goals. For example, the PCO table may say that `modify(/etc/init)` grants access to the `u.root` privilege. The attacker model would be used to find all other operations that grant that operation. For example, one attacker trick states that modify access to a directory grants modify access to all the files in that directory, so the `modify(/etc)` operation would be added to the OGM. The analysis would then look for ways to achieve this new operation.

The model for attacker tricks defines the vocabulary and relationships that are used to express attacker tricks. The examples in table 3-2 will clarify the kinds of knowledge that this model must be able to represent. The first two facts are information about specific files that exist on a specific machine. The analysis program must be able to read this information from the machine's file system. Similarly, the third fact describes a property of the `g.staff` privilege on this particular machine.

1. The file `/etc/rc` is directly writable using the `u.root` privilege.

2. The directory `/etc` is directly writable using the `u.root` or `g.staff` privilege.

3. The user-id privileges, `u.tom` and `u.alice`, have direct access to the group-id privilege, `g.staff`.

4. Any privilege that has direct write access to a CF grants modify access to that CF.

5. Modify access to a directory grants modify access to all the CFs in that directory.

**Table 3-2:** Sample Facts for The Model of Attacker Tricks

The key feature of the model for attacker tricks is the controlling-file abstraction. The CFs objects have attributes that represent the basic access control information. For example, a file object would have attributes listing the privileges that grant direct read and write access to that file. Thus the attacker model uses a general access control list to represent the security mechanisms of the computer being analyzed. Unlike Unix, the model does not restrict the number or type of privileges that can be listed in these CF attributes. The Unix protection system restricts the list of writers to include at most one user-id privilege, one group-id privilege, and one special group-id privilege called `other` or `world`. The model only needs to know which privilege directly grant the write operation for each file. The detailed restrictions are not modeled.

The fourth fact is a rule about how an attacker can use facts about the direct writers of a file. It states that for each privilege, `p`, in the list of writers of a file `f`, the `acquire(p)` operation grants the `modify(f)` operation. The last fact is similar. It says that for each file, `f`, in directory `d`, the operation `modify(d)` grants `modify(f)`.

From these facts, the analysis can deduce that `acquire(g.staff)` grants `modify(/etc)` which in turns grants `modify(/etc/init)`. The fact that `modify(/etc/init)` grants `acquire(u.root)` was deduced using the model of trusted programs.

### 3.3.1 Formal Model for Attacker Tricks

These examples point out that the model of attacker tricks has two basic objects: privileges and controlling-files. A Kuang system can read and record properties of these objects. For example, file objects will have attributes like the list of privileges that grant write and read access and a reference to one or more directories that contain this file. Figure 3-9 lists the general properties of these two objects. For a particular system like Unix there will be multiple types of CFs and privileges. The attributes for these specific objects will also vary from one operating system to another.

- **Controlling-Files**  
  writers: list of privileges  
  readers: list of privileges  
  parent-directory: controlling-file

- **Privileges**  
  members: list of privileges

**Figure 3-9:** Object Properties in the Model of Attacker Tricks

The two general operations on CFs are **`modify`** and **`read`**. Usually, the read operation does not help an attacker gain control of a process, but it might be used to find out security relevant information like the passwords for users or files. The one general operation on privileges is **`acquire`**. A specific operating system may have more than one instance of these general operations. For example, Unix has two kinds of modify operations. The **`replace`** operation allows changing the ownership of the file (e.g., this might happen if the file was deleted and a new file was created to replace it). The **`write`** operation does not allow the ownership to change, so if the file is deleted the file that takes it place must have the same owner. The object operations are summarized in figure 3-10.

The model uses rules to describe attacker tricks. Each rule describes the conditions that allow one operation to grant another. The conditions are expressed as a predicate on the values of one or more attributes for selected objects. If the conditions are true, then the deduced grant relationship is added to the OGM.

- **Controlling-Files**
  modify(cf): Attacker can change information read from file.
  read(cf): Attacker can view information in the file.

- **Privileges**
  acquire(p): Attacker can run any program in a process that has access to the privilege, p.

**Figure 3-10:** Object Operations in the Model of Attacker Tricks

### 3.3.2 Representing Attacker Tricks

This section briefly describes how the model can be used to represent the facts about attacker tricks that were presented in table 3-2. To avoid including syntax details the representations are described in english.

The first two facts describe the privileges that have direct write access to /etc/rc and /etc. These facts are represented by the values of the writers attribute of CF objects associated those files. These values are computed by reading the protection information for the machine being analyzed.

The third fact states that the u.tom and u.alice privileges have direct access to the group-id privilege g.staff. This fact is represented by the value of the members attribute of the object associated with the g.staff privilege.

The next fact relates the value of a writers attribute of a CF to the ability to perform the modify operation on the CF. It is represented by a rule that says the operation modify(f) is granted by the operation acquire(p) for all privileges, p, in the writers list of the file object for f. Similarly, the last fact is represented by a rule that says modify(f) is granted by modify(d) where d is the parent directory of the file f.

In summary, facts about CFs and privileges are read from the protection information of the computer being analyzed and stored as the values of attributes associated with those CFs and privileges. The deductions about the relationship between operations are recorded in the OGM, which is also the output of this step of the analysis. The operations can be thought of as attacker goals, and in this sense the rules which express attacker tricks describe how goals can be achieved in terms of properties of objects and the ability to achieve subgoals (i.e., other operations).

## 3.4 Transitive Closure Step

The only complication in the third step of security analysis is that the transitive closure algorithm must handle all the knowledge about attacker tricks that use more than one operation to grant the desired operation. For example, one attacker trick for Unix requires write access to multiple directories and that may require performing `acquire` operations on several privileges. On VMS systems there is a trick for acquiring super-user privileges that requires both the privilege to submit batch jobs and the privilege to set the privileges of a job. A regular transitive closure algorithm cannot handle this.

After computing the transitive closure of the OGM, the third step extracts all the relationships between privilege operations and builds the privilege access table. To build this table, the program needs to know about the initial distribute of privilege to users. This information is read from the machine's authorization database (e.g., `/etc/passwd` and `/etc/group`) and stored in an attribute of the objects that represent each user.

## 3.5 Summary of Knowledge Model

The knowledge model that a Kuang system uses to check computer security has one component that understands the behavior of trusted programs and another that understands tricks a user could use to extend his privileges. Each model has been presented by defining a number of objects that represent the facts deduced by that model. The attributes of these objects express facts about the configuration of the security system (e.g., the list of privileges that grant write access to a file, or the list of commands execute when the computer boots). Facts about the relationships between objects are recorded in separate tables. For example, the fact that modify access to the directory `/etc` grants modify access to the file `/etc/init` is recorded in the operation grant matrix. Tables 3-3 and 3-4 list the key objects and tables for the two models.

New deductions are made by applying rules to the attributes of existing objects. Each rule has an antecedent and a consequent. The antecedent is a predicate that can select objects and test the values of attributes. If the predicate is true, the consequent can create new objects, set the values of existing attributes, or make entries in the tables. Conceptually, all antecedents are tested after any change to an object. In practice only a small number of antecedents need to be tested. The analysis ends when there are no new deductions.

- **Process** - Represents instantaneous state of a program.
- **Privilege** - Represents the tickets of the protection system.
- **Script** - List of actions a process will perform.
- PCO - Table of operations that can control a process. Organized by the privileges available to the process.

**Table 3-3:** Objects in the Model of Trusted Programs

- **Privilege** - Represents the tickets of the protection system. Includes information about its relation to other privileges.
- **Controlling-File** - Any source of information that controls a process including programs, data files and directories.
- **Controlling-Operation** - A goal meaningful to an attacker.
- OGM - Boolean matrix listing the COs directly granted by each CO.

**Table 3-4:** Objects in the Model of Attacker Tricks

# Chapter Four

# Description of U-Kuang

This chapter presents information about the Unix (BSD 4.2) security analysis system. The previous chapter presented an abstract model that could be used by several Kuang systems, this chapter illustrates how the model is tailored to a particular computer system. The first section describes the major components of U-Kuang and the interfaces between those components. The second section presents the functionality and features of its rule based system. The last section lists the knowledge that it has about trusted programs and attacker tricks.

## 4.1 Structure of U-Kuang

The program is implemented in three layers as shown in figure 4-1. The lowest layer is a database for querying the security configuration of the computer being analyzed. The database supports simple queries to examine file protection information, and complex queries that involve parsing the contents of files. For example, the database can parse the password file and return a list of all the users. The Instance Database contains all the knowledge about the format of files on Unix.

The rule based system (RBS) builds a graph that describes how access to each privilege can lead to access to other privileges. The RBS layer is described in detail in section 4.2. Its rules describe the behavior of privileged programs and the tricks that can be used to acquire privileges. The RBS uses the rules and the information in the instance database to build a model of the security relevant

```
Access Policy --> Policy Checker
                        ^
                        | Privilege Graph
Abstractions,           |
Facts, Rules  --> Rule Based System <==> Domain
                        ^                 Model
                        | Queries, Parsers
Security                |
Configuration --> Instance Database
```

**Figure 4-1:** Structure of U-Kuang

operations that each user can perform. The model is built in a database module called the Domain Model.

The top layer of the program checks for and reports on any violations of the specified access policy. The final result of the RBS is a directed multi-graph that describes how the privileges are connected. The nodes of the graph are privileges. Each plan for acquiring a privilege P1 using a privilege P2 is represented by an arc from P2 to P1. The policy checker takes the transitive closure of this graph to compute the list the privileges accessible to each user. If the computed list differs from the list specified by the access policy, the checker scans the graph to identify the arcs that cause the violations. The plans corresponding to these arcs are displayed to help the user debug the security configuration.

## 4.2 U-Kuang's RBS

The rule based system used by U-Kuang is similar to IKE [19]. The main difference is that U-Kuang's RBS queries a database instead of a person when it needs information about the world being analyzed. The RBS is implemented in a dialect of lisp called C-Scheme [20]. This dialect is well suited to a Unix analysis tool because it allows lisp programs to call functions written in the C programming language. A large amount of code for parsing Unix command files and system tables already exists in C, so I did not need to re-implement it in lisp.

U-Kuang reasons about a model of the Unix security system that is represented by objects. Each object has a number of attributes, and it is the values of these attributes which represent facts about the security system. An object's type determines the set of attributes it possesses and the type of values that can be bound to each attribute. The values can be atomic objects (e.g., numbers, strings, lists) or references to other objects.

Object types are defined by an *abstraction tree.* The abstraction tree defines a rooted hierarchy of types. Each type defines a set of attributes that are inherited by all its subtypes. The abstraction tree makes it possible to write a rule that applies to all subtypes of a given type. For example, many rules apply to both directories and files, so they are both subtypes of a database type. The abstraction tree for U-Kuang is shown in table 4-2. Details about some of these abstractions are presented in section 4.3. Objects are instances of the leaf types of the abstraction tree. As objects are created, they are placed in a database called the *domain model.*

The inference engine contains both a planning system and a value-finding system. The rules for the value-finding system describe how unknown attribute values can be deduced from known attribute values. The rules for the planning system describe how goals can be achieved from subgoals. The two inference systems are unified by associating goals with attributes and by making plans be one of the atomic types supported by the RBS. In this framework a single backward-chaining inference algorithm can handle both planning and value-finding.

1. Object-Root. Root of the tree.

2. Process-Segment. One piece of the behavior of a process.

3. FNE. Context for resolving file names. A table mapping strings to strings.

4. Script. A list of actions a process-segment can take.

5. Privilege. A security capability.

   a. User-Priv. Access to a user ID.

   b. Group-Priv. Access to a group ID.

6. Controlling-file. The basic abstraction of the Unix file system.

   a. File. Holder for directories, data files, and programs.

   b. Swap-Space. Device that grant access to the raw file storage.

   c. Partition. Device that grant access to the raw file storage.

7. Path-Name. Abstracts canonical file and directory names.

8. Program. Information about scripts that come from binary files or shell scripts.

9. User. Information about a user.

**Figure 4-2:** U-Kuang's Abstraction Tree

Each rule has an antecedent and a consequent. The antecedent is a predicate on the state of objects found in the domain model. This predicate calls lisp functions to lookup or create new objects in the domain model. The consequent sets the value of an attribute for some object. If the attribute holds a list value, the consequent can append elements to the list. For example, a consequence can add a plan for achieving a particular goal to an attribute which lists existing plans.

When a backward-chaining inference algorithm needs to determine an unknown attribute, it searches its list of rules for the ones that might define the attribute. The antecedents of each of these rules are evaluated and then one or more consequents will assign a value to the attribute. This process is recursive because evaluating an antecedent may require determining another unknown attribute.

The special object, `model`, has pointers to all the objects created in the domain model. It has attributes for each type of object and the values of these attributes are the lists of objects of each type. For example, the value of `model.inode` is the list of all the objects of type `inode` or any subtype of `inode`. One of the clauses that can appear in the antecedent of a rule tells the inference engine to apply the rule to each element of a list value.

The initial contents of the domain model is described by a number of *facts*. Each fact creates an object and initializes its attributes. As the U-Kuang runs, it creates new objects and determines in the attributes for old and new objects.

## 4.3 Example of Security Analysis

This section presents a detailed example of how the rules and objects are used to uncover an operational security hole. Only a small part of the abstraction tree and rule base is presented in detail. The entire knowledge base is described in section 4.4.

The security hole considered in this section allows a user with access to the `u.tom` privilege to acquire access to the `u.alice` privilege. Presumably this access is contrary to the computer's access policy. The plan for exploiting this hole is illustrated in figure 4-3. Access to the `u.tom` privilege grants access to the `g.guest` privilege because Tom is a member of the guest group. The `g.guest` privilege has direct write access to the file `/usr/alice/.login`. That file is a controlling-file (CF) for a process that runs with the `u.alice` privilege. Specifically, when the user Alice logs into the computer, her shell (command interpreter), `/bin/sh`, reads a list of initialization commands from that file. An attacker who can add a command to this file can create a copy of the shell that sets it's user-id to the `u.alice` privilege. The attacker could then use this special shell to execute any desired program and that program would inherit access to the `u.alice` privilege from the special shell[9]. The rules and objects used to create this plan are explained below.

To uncover this hole, U-Kuang uses knowledge about the behavior of the `/bin/sh` program and knowledge about how someone could modify the file `/usr/alice/.login`. The facts associated with this knowledge are represented by the values of attributes of four types of objects: process-segment, privilege, script, and controlling-file. The attributes for each of these objects are listed in figure 4-4 and described below.

The main difference between the abstract model presented in chapter 3 and the model used by the Unix checker is that the checker must build detailed plans describing how each user can acquire the privileges that are accessible to that user. In the abstract model, the first step in the analysis just records the

[9] There are many ways an attacker can install a back-door that allows later access to Alice's privileges. Some people believe that the set-id feature is the root of all Unix security problems. This is wrong. The feature is just the simplest way to install a back-door. Alternatively, the attacker could change Alice's search path to include one of his directories. The attacker would install programs in this directory that had the side-effect of executing his commands as well as the commands that Alice intended.

```
                     Rule Z
acquire(u.tom)     --------->   acquire(g.guest)
                                        |
                                        |   Rule Y
                                        |
                                        V
                                 write(/usr/alice/.login)
                                        |
                                        |   Rule X
                                        |
                     Rule W             V
acquire(u.alice)   <---------   control(/bin/csh for alice)
```

**Figure 4-3:** Plan to Exploit an Operation Security Hole

controlling-operations that provide toeholds for each privilege. The Unix checker must identify the precise step in the behavior the process that provides the toehold. For example, on Unix the same process executes the login program and the shell for a user. Initially the process has access to super-user privileges, but later it just has the privileges available to the user. An attacker would prefer to control the process while it had super-user privileges. To keep track of the changing privilege state, the abstract process is modeled by a series of *process segments* (PSs). Each segment has a fixed privilege and naming state. Actions that would change either component of the state, create new PSs.

Another difference is that the abstract model has separate tables for recording the relationships between operations whereas U-Kuang records these facts in the attributes of the objects that represent process segments, privileges and controlling-files. As shown in figure 4-4, PS objects have an attribute that lists all the plans for controlling the PS. If an operation provides a toehold for controlling a particular segment of a process, then that operation is expressed as a plan and it is added to the list of plans in the `controllable` attribute for that segment. The set of privileges granted by this operation can be deduced from the `privileges` attribute of the PS. In this way, the `privileges` and

**Process-Segment**

- privileges - Fixed list of privileges available to this PS.
- naming - Fixed table of naming information.
- current-action - Current action taken by this PS (mutable).
- script - List of remaining actions (mutable).
- controlling-files - Deduced list of controlling-files for this PS.
- forks - Deduced list of programs forked by this PS.
- sources - Deduced list of files that provide scripts.
- controllable - Deduced list of plans to control this PS.

**Privilege**

- kind - Fixed value one of Group or User.
- id - Fixed integer.
- members - Fixed list of user-privs with access to this privilege.
- processes - Deduced list of process-segments that have this privilege.
- accessible - Deduced list of plans to acquire this privilege.

**Script**

- new-naming - Specification of changes to naming state.
- new-priv - Specification of changes to privilege state.
- to-fork - Specification of programs to fork. The name of the program file is resolved using the naming state of the PS.
- to-source - Specification of files to parse to get scripts. Alternatively this can specify script constants.

**Controlling-File**

- writers - Fixed list of privileges that grant direct write access.
- readers - Fixed list of privileges that grant direct read access.
- parent-directory - Fixed list of directories that contain this CF.
- writable - Deduced list of plans for to achieve `write(cf)`.
- replaceable - Deduced list of plans for to achieve `replace(cf)`.

**Figure 4-4:** Object Attributes for Analysis Example

**Rule 1:** *Any file sourced by a process is a controlling file for that process.*

```
Foreach ps in model.process-segments
  Bind action to ps.current-action
  Foreach cf in action.to-source
Then
        set ps.controlling-files includes cf
```

**Rule W:** *A privilege can be acquired by gaining control of one of the processes that runs with that privilege.*

```
Foreach priv in model.privileges
  Foreach ps in priv.processes
    If ps.controllable
Then
        achieve priv.accessible
```

**Rule X:** *A process can be controlled by writing one of its controlling-files.*

```
Foreach ps in model.process-segments
  Foreach cf in ps.controlling-files
    If cf.writable
Then
        achieve ps.controllable
```

**Rule Y:** *A controlling-file can be written using any of the privileges that have direct write access to that file.*

```
Foreach cf in model.controlling-files
  Foreach priv in cf.writers
    If priv.accessible
Then
        achieve cf.writable
```

**Rule Z:** *A privilege can be directly acquired by all of the users who are 'members' of that privilege group.*

```
Foreach priv in model.privileges
  Foreach user-priv in priv.members
    If user-priv.accessible
Then
        achieve priv.accessible
```

**Figure 4-5:** Rules for Analysis Example

controllable attributes replace the table of privilege controlling operation that was described in chapter 3.

In a similar way, the write(cf) and replace(cf) operations are represented by the values of the writable and replaceable attributes of the controlling-file object. A fact like write(/etc) grants replace(/etc/init) is expressed by a plan stored in the replaceable attribute of the /etc/init object. By representing the operation grant matrix by the values of attributes it is possible to express attacker tricks as rules for finding the values of attributes. In this case, the rule describes how to fill in the replaceable attribute for a file object from the value of the parent-directory attribute for this object.

As the RBS runs it creates new PSs and simulates the actions specified in the script for each PS. At some point, the simulation will model the logging in of the user Alice. A PS will be created to execute the /bin/csh program and its privileges attribute will include u.alice.

When the PS for Alice's shell is created, Rule W of figure 4-5 is triggered. This rule expresses one of the attacker tricks. It says that the acquire(priv) operation can be achieved by controlling any PS that includes priv in its list of privileges. This information is encoded as follows: The foreach clause on the first line of Rule W tells the inference engine to apply this rule to all the privileges that are created in the domain model. The notation model.privileges selects the privileges attribute of the model object. Recall that the special object model records all the objects of each type. One at a time, each privilege is bound to the label priv. Associated with each privilege is a list of the PSs that have this privilege in their privilege state[10]. The second line of Rule W binds the label ps to each one of those PSs. The remaining lines

[10] A different rule updates this attribute as new PSs are created.

of the rule construct a plan relating the ability to control the PS, **ps**, to the ability to access the privilege, **priv**.

The plan created by Rule W can be thought of as adding a grant relationship to the OGM. Equivalently the plan defines an arc in figure 4-3, which can be viewed as an operation grant graph. The **accessible** attribute of a privilege object represents the ability to perform the **acquire(priv)** operation. Likewise the **controllable** operation represent the **control(ps)** operation. The plan built by Rule W indicates that **control(ps)** grants **acquire(priv)**.

The behavior of the Alice's shell is represented by a compiled-in script. That script is held in an object of type *program* and it is extracted from the knowledge base using its canonical pathname (**/bin/csh**). One of the actions in that scribe tells the process to source (read and execute) commands from the file **/usr/alice/.login**. When this action becomes the **current-action** of the PS, Rule 1 of figure 4-5 will trigger.

Rule 1 records a piece of knowledge about the behavior of trusted programs. It states that any file that provides commands for a process is a CF for that process. The first line of the rule binds **ps** to each PS in the domain model. The second line binds the label **action** to the current action of that PS. The effect of these two line is to cause this rule to be examined each time the current action changes for each PS created. The third line iterates over the list of files sourced by the current action. If such files exist, the consequent of the rule is executed. The consequent adds the name of the sourced file, in this case **/usr/alice/.login**, to the list of CFs for this process segment.

Skipping the details of syntax, Rule X examines the list of CFs for each PS and constructs a plan relating the **writable** attribute for each CF to the `controllable` attribute of the PS. This plan adds the arc in figure 4-3 between the `write(/usr/alice/.login)` operation and the `control(/bin/sh for alice)` operation.

Other rules examine the `controlling-files` attribute and according to information found in the naming state of the PS, those rules add plans to the `controllable` attribute of the PS. This extra level of indirection makes it easy to express knowledge about how names are resolved. If a CF is specified by a search path, then plans to control the PS are added for each directory searched before the CF was found.

Each time a new CF object is created, Rule Y is triggered. This rule binds the label `priv` to each privilege listed in the `writers` attribute of the CF. The value of that attribute is computed by a lisp function in the instance database. The function examines the security configuration of the computer being analyzed to see which privileges have direct write access to the CF. The last three lines of Rule Y construct a plan linking the `writable` attribute of this CF to the `accessible` attribute of the privilege identified by `priv`.

The final arc in figure 4-3 is filled in by Rule Z. That rule is applied to each privilege in the domain model. It binds the label, `user-priv`, to each of the privileges listed in the `members` attribute. This list is empty for user-priv objects, but for group-priv objects it lists the user-priv objects that have direct access to this group-priv. In this example the privileges `u.tom` and `u.alice` have direct access to `g.staff`. The consequence of Rule Z builds a plan that says `acquire(u.tom)` grants `acquire(g.staff)`.

Viewing figure 4-3 as an operation grant graph, the last step in the analysis is to take the transitive closure of the graph. The plans that allows an attacker to go from `acquire(u.tom)` to `acquire(u.alice)` are built by concatenating the plans for traversing each arc between those two node.

## 4.4 Knowledge about Unix Security

The purpose of U-Kuang's model of Unix is to identify all the ways that trusted programs can be manipulated to grant users additional privileges. The model assumes that all programs function as expected and it includes all the attacks published by researchers at Bell Laboratories ( [25] and [10]). U-Kuang is not trying to find or exploit bugs in the software. The goal is to analyze the interactions between the kernel and the privileged program to see how access to one privilege can lead to access to other privileges. To carry out the analysis, the RBS needs knowledge about the tricks for extending privileges, and knowledge about the behavior of trusted programs. Basically, the RBS simulates the activity of the programs and looks for actions that can be tricked into extending the privileges available to some user.

U-Kuang's model of the Unix security system contains abstractions and rules. The abstractions define object types and thus they define the organization of the facts that the RBS will deduce about Unix. The rules are organized around goals that are meaningful to an attacker of the system. The four goals in U-Kuang's model are `acquire privilege`, `control process`, `write file`, and `replace file`. The `write` and `replace` goals are both aimed at changing the information found when a process opens a file for reading. The difference is that the `replace` goal allows changing the ownership of the file[11], whereas the `write` goal requires that the ownership of the file does not change. Basically, the goal of acquiring a privilege is achieved by controlling a process that runs with the desired privilege. Control is achieved by writing or replacing a file that is critical to the behavior of that process. Completing the loop, writing a file is achieved by acquiring access to a privilege.

[11] For example, deleting the original file and creating a new one in its place is an acceptable way to `replace` a file.

This chapter presents the contents of U-Kuang's knowledge base without going into the details of how that knowledge is represented. Knowledge about attacker tricks will be presented first.

### 4.4.1 Attacker Tricks

Knowledge about attacker tricks is organized by the four attacker goals: `Acquire privilege`, `Control process`, `Replace file`, and `Write file`.

#### Acquire Privilege

The goal of acquiring a privilege means that the attacker can execute an arbitrary program in a process with the effective user-id or group-id set to the desired privilege. U-Kuang has two tricks for acquiring a privilege.

1. Find a user account that has direct access to the desired privilege, and does not have a password.

2. Find a process segment that runs with the desired privilege and is controllable.

Other possibilities include guessing at passwords or taking active measures to intercept a password. These were not modeled.

#### Control Process

A process is controllable if an attacker can cause it to execute (or create another process to execute) a program chosen by the attacker.

The general trick for controlling a process is modifying a database that controls the process. For example, the file containing the executable code for a process is one of the controlling databases. Other controlling databases include directories that are searched or data files that influence the forking of processes. The knowledge about trusted programs and the simulation of those programs produces the list of databases that control each process that runs with a particular privilege. The attacker trick for exploiting controlling databases has two forms depending on whether the ownership of the database is allowed to change. They are shown in figure 4-6.

1. Find a process segment for this process that has a controlling database that is replaceable.
2. Find a process segment for this process that has a controlling database that is writable (i.e., the ownership of the database file may not change).

**Figure 4-6:** Rules to Control a Process

In general, an attacker might need to modify several files in order to gain control of a process. Surprisingly, this research did not uncover any cases where more than a single file needed to be modified. The closest the model comes to the double-file situation is the workings of the `rlogin` program (see section 4.4.2). If a user has a non-empty `.rhosts` file, then an attacker who can modify the host name table can masquerade as the user authorized in the `.rhosts`. This plan assumes that the attacker has super-user access on some computer attached via a network. This trick can only be used if the `.rhosts` file is not empty and the `/etc/hosts` file is replaceable. It does not depend on the replaceablity of both files.

### Replace File

A file can be replaced if the attacker can change the information obtained when a process opens and reads from that file. The ownership of the file is allowed to change. The tricks for achieving this are shown in figure 4-7.

The last rule handles the recursive nature of the `replace` goal. Replacing a directory file can be achieved by replacing the directory which contains the original directory, etc. Making a list of all the directories that are examined

1. Achieve the goal `write file`.

2. Achieve the goal `replace dir` for the parent directory of the file. This trick works by deleting the original file and replacing it one that has the desired contents.

3. Achieve the goal `replace dir` for any directory searched before the desired file was found.

**Figure 4-7:** Rules to Replace a File

when a file name is resolved is complicated by search paths and indirect file names (soft-links). A lisp function is used to handle these complexities.

### Write File

For attackers to be able to write a file, they must be able to change the contents of the file without changing the ownership of the file. The tricks for achieving this are shown in figure 4-8

1. Acquire super-user privileges. This rule expresses the importance of super-user access.

2. Acquire access to any privilege that has direct write access to the file.

3. Acquire access to the privileges of the file's owner. The owner can always change the protection modes of the file to grant herself write access.

4. If the directory containing the file is writable, find a writable file on the same disk partition that has the same owner.

**Figure 4-8:** Rules to Write a File

The last rule works because the original file can be deleted and some other file with the same owner can take its place. This attack depends on the feature that the Unix move (rename) system call does not change the ownership as long as the file moves within the same disk partition.

### 4.4.2 Trusted Programs

U-Kuang's knowledge about trusted programs is organized on a per-program basis. The behavior of each program is specified separately. Since the underlying model was presented in the last chapter, and the knowledge inference techniques were presented section 4.2, the knowledge base can be presented by listing the facts that are known about each trusted program.

Advocates of rule based systems claim that knowledge can be added in a modular fashion. Within limits, this statement is true. As long as the basic model that underlies the RBS does not need to change, information can be added easily. For example, adding information about another program that uses search paths is not difficult. However, adding the first program that used search paths was difficult. An early version of U-Kuang did not have the notion of a File Naming Environment. It could not represent attacks based on search paths. Adding the FNE required changing almost all of the rules.

#### Init

The `init` program drives the session cycle (login, shell, logout). Before any sessions are started it forks (i.e., creates a new process to execute) a shell, `/bin/sh`, to execute the commands found in `/etc/rc`. When that completes, `init` examines the database in `/etc/ttys` to determine which program to fork for each tty device. The default is to fork the program `/bin/getty`. The database can specify other programs to fork.

#### Getty

The program `getty` waits for a terminal device to become active, prints a greeting banner, and requests a username. By default it passes the username to the login program, but an alternative program can be specified in the file `/etc/gettytab`. The list of possible programs forked is computed by a script parsing function that examines `/etc/gettytab`.

### Inetd

The TCP/IP daemon, `inetd`, mediates all network connections. It behaves like `getty` for remote logins via a TCP/IP network. The mail daemon and file transfer daemon are forked by `inetd`. One of the first actions of this program is to change its user and group IDs to `daemon`.

### Login

The login program forks a shell (or other program) for each user that can log in. The file `/etc/ftpusers` lists the users that are not allowed to login. The `ftpusers` database allows anonymous file and mail transfer without allowing anonymous login. The login program is modeled by a process that changes its privileges and forks a shell for each user in the file `/etc/passwd` that is allowed to login. The password file is treated as the source of a script for the login program. Any account that does not have a password can be accessed by all users via the `login` program. They are also accessible via the `su` program.

### Shells

Shell programs are the Unix command interpreters. When a shell is started to read commands from a user's terminal (as opposed to executing commands from a file), it begins by reading commands from an initialization file. For example, the shell, `sh`, reads commands from the file `.profile` in the user's home directory. Some shells read command files on logout as wells as on login (e.g., `csh`), and others read initiation files that are shared by several users (e.g., all the students taking a particular class). The behavior of each shell is represented by an initial script of actions for that shell.

When a shell is invoked in non-interactive mode, it skips reading some of its initialization files. U-Kuang models this by treating the non-interactive shells as different programs that have different initial scripts.

Currently, U-Kuang's simulation does not include the fact that most user's invoke a text editor, or any of a number of other common programs. It would be easy to add this. The parser for the shell initialization file would be extended to append a list of programs which the given user executes. This information could be extracted from the system accounting records. To take advantage of this new information, the program should also be extended to include knowledge about the behavior of the common programs (e.g., `gnuemacs`, `vi`, and `cc`).

These improvements are not necessary to check for the possibility that an attacker has created a hostile program that will be invoked when the user thinks he is invoking a friendly program (i.e., a trojan horse). This sort of attack is based on the fact that a search path is used to resolve command names. All the commands in the login initialization will be resolved with the aid of that search path (including commands to change the search path), so most initialization files will exercise the search path sufficiently to turn up any possibilities for trojan horses.

Shell programs resolve command names in a complex way that refers to *shell variables* which are passed from one program to the next as part of a program's *environment*. For example, the command search path is stored in one shell variable whereas the editor library search path is held in a different variable. One shell, `sh`, uses a shell variable, `IFS`, to set the inter-field separator characters (normally blank and tab). U-Kuang knows about attacks that exploit this feature. U-Kuang has procedural knowledge about how shell variables are used to resolve file and command names. Each trusted program states which procedure is used to resolve each name.

### Cron

The `cron` program executes commands periodically and it is the central feature of the automatic management system provided by Unix. `Cron` actually runs continuously, but U-Kuang models it as a program that runs once performing all the actions that are specified in `/usr/lib/crontab`. That file is treated as the source of `cron`'s script. In general, U-Kuang models system daemons as running once and performing all possible actions. The fact that `cron` runs with root privileges is deduced from the fact that it is forked by a command in `/etc/rc` which is read by the `init` program.

### Atrun

The `atrun` program allows users to request that commands be executed at a later time. `Atrun` provides the features of `cron` to ordinary users. Ordinary users are not allowed to write the `crontab` file because any program mentioned in `crontab` will be run with super-user privileges. The `atrun` program determines the privileges it should use when executing a command by looking at the ownership of the request file in the directory `/usr/spool/at`. Anyone who can create a writable file in that directory with a given ownership can acquire privileges available to that owner. The attacker trick of moving files within a disk partition makes this easy, so many Unix sites do not run the `atrun` program.

### Remote Execution

The `rsh`, `rlogin`, and `rlogin` programs allow remote operations. They allow a system administrator to state the list of all the hosts which are trusted by all users. This information is kept in the file `/etc/hosts.equiv`, and the host names in that database are resolved with the aid of a database stored in `/etc/hosts`.[12] Each user can list additional hosts and users that he trusts by specifying them in a file, `.rhosts`, in his home directory. The server end of these programs (`rshd`, `rlogind`, and `rcpd`) insist that the `.rhosts` file is owned by the user whose account is being accessed.

[12] This research predates the conversion to a distributed nameserver system.

### Syslog

The **syslog** program records system activity and sends messages about critical system events. It appends messages passed to a Unix system call into one or more files as specified in **/etc/syslog.conf**. Since **syslog** runs with root privileges, replace access to **syslog.conf** grants append access to all files (like the password file).

### Sendmail

The heart of the Unix mail system is the **sendmail** program. It is invoked by any user who sends mail, and it is run with **daemon** privileges when it is invoked by **inetd** on inbound messages. It reads several databases any one of which can be used to gain access to the privileges that **sendmail** is running with. The databases in the **/usr/lib** directory are: **aliases**, **aliases.pag**, **aliases.dir**, **sendmail.cf**, and **sendmail.fc**. It also reads a file in each user's home directory, **.forward**. An essential ingredient in all attacks based on these files is that the mailer can be told to invoke an arbitrary program in addition to appending a message to some mailbox file.

The Unix mail system has several security problems, but these were the only ones that were modeled by U-Kuang.

### ... and so forth

A complete model of Unix would need to include several additional programs. All programs that run with super-user privileges should be modeled. Currently U-Kuang just prints warning messages identifying the program it doesn't know about. The programs it does know about were chosen to illustrate the range of interactions that are possible between trusted programs.

# Chapter Five

# Experience Running U-Kuang

An early version of U-Kuang has been run occasionally on about 30 computers at MIT. The first time it is run, it almost always finds a hole that allows all users to acquire super-user privileges. Most of the problems are simple mistakes like group writable initialization files in the home directories of privileged users. After U-Kuang has been run a few times users learn about the interactions between different privileged programs and the simple errors disappear. In many cases system administrators were surprised to learn that there was no difference between the privileges they have been calling `staff`, `operator`, and `wheel`. They all could be used to acquire super-user privileges[13].

U-Kuang has been run regularly on three machines that try to be secure and on these machines it finds approximately two new problems each month. One source of problems is the fact that critical databases for super-user processes exist in several directories. There are numerous reasons for allowing users to have write access to one of those directories, and that access can be used to acquire super-user privileges. Another problem arises from a common procedure for updating system tables. The old file is renamed to preserve the information about its last write date, then a new copy is installed and edited. Unfortunately the new copy is created with the default protection settings of the current

[13] As discussed in the chapter on limitations and extensions, it may be feasible to build a monitoring system that notices when a lower privilege like `staff` is being used to acquire super-user privileges.

process, not the settings of the original file. The new settings usually create a security hole. A general tool for updating any system file would be easy to implement and would help eliminate this kind of error. Many system administrators use such a tool (vipw) to update the password file but they need a tool that can handle all system tables.

Security holes that did not involve system tables usually involved two or three levels of indirection. Many users have personal bin directories which hold customized versions of their favorite programs. These directories appear first in the users' search path, and thus are good places to plant trojan horses. On one system, which was intended to be secure, U-Kuang found an operational security hole that allowed any user to acquire super-user privileges by going through three users, alice, bob and charles[14]. Alice philosophically objected to computer security, so she allowed anyone to write her home directory. Anyone could acquire her privileges by replacing her login initialization file. In particular, she was a member of the games group. Bob's personal bin directory was writable by anyone in the games group; anyone who could get access to the games privilege could plant a trojan horse in /usr/bob/bin and acquire bob's privileges. Bob was not a trusted user, but he was a member of the friends group which charles had set up for his friends. The last connection was that charles, who had operator privileges, had a program in his bin directory which was writable by anyone in the friends group. The program had been written by another member of the friends group, and charles had moved it to his bin directory without changing its protection modes. By indirecting through alice's and bob's accounts, anyone could plant a trojan horse in that program and thus they could acquire operator privileges from charles. As mentioned earlier, operator privileges are usually sufficient to gain super-user privileges. It would be quite hard for a person to check for holes that involved three levels of indirection but U-Kuang has no trouble finding them.

[14] Not their real usernames.

The running time of U-Kuang depends primarily on the number of group privileges, not on the number of users. It takes U-Kuang about 15 seconds to analyze a system with 100 users and 10 groups, and about 40 seconds for a system with 1100 users and 30 groups. More precisely, the running time of U-Kuang is proportional to the number of edges in the *privilege graph.* The privilege graph is a multi-graph whose nodes are privileges and files. The number of edges between any two nodes depends on the number of rules which describe plans for using access to the first node to grant access to the second node. There are very few rules that specify how an attacker can go from a user privilege to a group privilege, whereas there are many rules specifying how to go between group privileges. Thus the running time depends primarily on the number of group privileges, rather than the number of user privileges (i.e., the number of users).

U-Kuang has also been used to find back-doors[15] left by someone who cracked into a system. U-Kuang quickly finds any file protection modes that have been changed by the cracker. In one case it found an extra device file for the disk partition that contained the `/bin` directory. The cracker had a program that could use write access to that disk partition to create a shell with super-user privileges. One blind spot of U-Kuang is that it does not check for extra programs that have the set-user-id feature. Crackers often create such programs with the super-user or operator privileges. This blind spot could be corrected by adding a few facts to the knowledge base.

Another weakness is that the program does not check for extra accounts that the cracker might have created, or unused accounts that now have passwords known to the cracker. It is not clear whether a rule based approach

[15] A back-door is any modification that makes it easier for a system cracker to get back into a system.

could systematically search for this kind of back-door. It may be that the best way to check for back-doors is to compare the system to an earlier snapshot of itself as suggested in [25].

This experience does not imply that the Unix security system is flawed. The correct conclusion is that people make mistakes when they change the configuration of complex security systems. U-Kuang finds these mistakes.

# Chapter Six

# Limitations and Extensions

Preceding chapters have described a rule based system that can analyze Unix systems and identify serious operational security holes. This chapter discusses the limitations and extensions of a rule based approached to security problems. The first section addresses the question of building Kuang systems for other operating systems. The second section points out that it is possible to answer interesting questions about a security system by examining properties of the rule based model for that system. Section three identifies two obstacles that make it difficult to synthesize protection decisions from specifications of the access policy. The last section discusses the possibility of using a rule based approach to detecting inappropriate system activity.

## 6.1 Other Operating Systems

Many features of computer systems can be simulated using the model presented in Chapter 3. The question is whether there are important features of a system which are not captured by the model. For example, DEC's VAX/VMS system has a flexible file protection mechanism that includes both group restrictions and access control lists. This turns out to be easy to convert into the abstract model because the model uses general access control lists. However, this does not prove the the model can handle all the security relevant features of VMS.

A Kuang system for VMS would need to know about third-party software like the TCP/IP network facilities. In general, third-party software does not use the protection facilities as well as the software which was developed by DEC. It would be important for the Kuang system to know about these packages and know how to tell if they are being used.

## 6.2 Analyzing the Rules

The rules that describe a particular operating system form a representation of that system, and as with all representations there will be questions which are easier to answer using that representation than others. The rule based description can provide new insights into the nature of the system, or provide an easier way to analyze a suggested change to the system.

One interesting question to ask about a security system is whether conspiracy between users can yield surprising results. For example, most bank vaults cannot be opened unless two employees conspire to open it. More precisely, the question is whether the set of privileges achievable by two users working together can be larger than the union of the privileges achievable by each user operating alone. If conspiracy yields greater gains than the union of each conspirator's privileges, then the security checker's job becomes more complex. It must consider the privileges accessible to each group of conspirators. The policy specification language become correspondingly more complex since it must specify the acceptable and unacceptable forms of conspiracy.

The conspiracy question can be answered by examining the rules describing a security system. If the antecedent of each rule requires at most one privilege, then conspiracy does not yield surprising results. U-Kuang's rules have this property, so conspiracy is not a problem within its model of the Unix security system.

There is nothing inherent in the Unix operating system that eliminates conspiracy. It would be easy to build an trusted program that could not be subverted unless two files could be written. If those files require different privileges for writing then a conspiracy might be required to subvert the process. Notice that if one of the two files required super-user privileges to change, then conspiracy is not necessary. The program can be subverted if and only if the attacker can acquire super-user privileges. For some reason, programs like this do not exist on current Unix systems.

## 6.3 Synthesizing Protection Decisions

This project focused on the analysis of security systems, but an obvious extension is a system that can synthesize protection decisions (e.g., the ownership and protection modes for a file or directory) from a specified access policy. The system would need to handle new objects added to the file system and new rules added to the policy specification. The access policy would have to categorize objects which will be created in the future and then specify the access restrictions for each category. For example, a simple policy language might allow objects to be categorized by the program that created them or the directory that will contain them. Once the rule based system knows what restrictions apply to the new object, it must pick access controls that are consistent with all existing access policies. That is, it must be sure that the new object does not allow an attacker to acquire some new privilege. That part is easy; U-Kuang already performs that computation. The hard part is picking the correct access restrictions efficiently. Presumably the system would have rules that would guide this choice.

The second problem with synthesis is changing or amending the policy specification. It is easy to imagine that a new restriction could force a large number of protection decisions to change. It is also possible to add a rule which makes the policy specification inconsistent or unrealizable. If the policy specification language is complex, the questions about whether a policy is consistent or realizable becomes undecidable. However, the fact that some policies lead to undecidablities does not force us to give up hope. A single decision procedure may exist that can make the decision in bounded time for most policy specifications.

## 6.4 Computer Security Monitoring

Often users have the capability to perform some operation, but they are expected to refrain from performing it. If a computer had a truly flexible and convenient protection system, this sort of excess capability would not exist. However, modern protection systems are inflexible and hard to use, so this case arises frequently. The administrators of such systems would like to monitor these trusted users to determine whether they have become untrustworthy or whether someone else is using their account to crack into the system.

Most Unix sites have excess capability associated with the `staff` or `operator` group. Either of these privileges is usually sufficient to acquire super-user privileges. To the extent that U-Kuang's model is complete and accurate, it can construct all the plans that use those privileges to acquire super-user access. Thus U-Kuang could be an important piece of a Unix security monitor. It would be the piece that knows how an attacker might bootstrap his privileges. The rest of the monitor would have the job of examining the system activity (or the logs of system activity) to see if any of the plans are being performed. A general rule based system could encompass both the plan generation and the activity checking.

# Chapter Seven

# Conclusions

This thesis presents a novel solution to a long standing problem in the field of computer security. The problem is that a security system can be no better than the people who use it. Even if a computer's protection mechanism are free from software bugs, users can be lax or inconsistent in the ways that they use the protection mechanisms, and this leads to security holes. The computer will be doing exactly what it was told to do, but the overall effect could allow an attacker to acquire undesired access. Such an effect is called an operational security problem to distinguish it from a problem due to the design or implementation of the system. This thesis presents a general model for analyzing the operational security of computers and it describes how that model was used to implement a rule based security checker for Unix. The checker, U-Kuang, was run on several of MIT's undergraduate machines and it frequently found significant security holes.

## 7.1 Highlights of the Problem

A computer is operationally secure if the collection of operations that each user can perform is consistent with the security policy for that computer. Two approaches have been used to ensure the operational security of computers. The first approach is to choose a restrictive language for expressing security policies and then build protection mechanisms that directly read and enforce the policy specification. Operating systems aimed at military users have taken this approach. Such systems include ADEPT-50 [24], Multics [18], and SCOMP [12].

Unfortunately the range of policies that these systems can express is limited, and it is still possible for severe operational security problems to arise. For example, the system administrator might leave his login initialization file publicly writable and this could allow other users to execute commands using the administrator privileges. A serious problem with these restrictive languages is their assumption that the major function of a computer is to store and retrieve information. Basically, computers are treated as glorified bookcases. It is not clear how these languages can be extended to handle an information system that is more like newspaper publishing house than a bookcase.

The other technique for ensuring operational security requires that the computer's administrator periodically audits the configuration of the security system. The security configuration includes the protection information of all files, directories, and programs. The full security configuration is quite large. To simplify the audit, administrators enforce restrictive policies that reduce the complexity of the audit. However these policies may be hard to enforce, and if they are enforced, they may sacrifice the flexibility that initially attracted the users to the computer system. For example, Unix has an information sharing mechanism that allows a user to specify a program that the system should invoke when electronic mail is delivered to that user. Some sites turn off this feature to avoid complex interactions with other programs in the security system. No matter what the policies are, they are only checked when the administrator has time to check them. There is little guarantee that the system will stay in a secure state after the audit.

As systems implement new mechanism for flexible information sharing and automatic system management, the complexity of the security system increases. Additional features lead to additional programs that must be considered part of the security system. A few examples should convince the reader that the number of trusted programs has already increased as computers have become more useful. The trusted programs in Unix include: the login server, the command interpreter, the program that executes batch commands, the program that executes commands on other computers, and the program that automatically installs new software that has been released over a trusted network connection. The possible interactions between these programs makes security checking a complex problem.

## 7.2 Overview of the Solution

This thesis presents a rule based approach to security checking and describes a program that can automatically audit the operational security of Unix computers. The primary benefit of this new approach is that it is systematic and thorough. It harnesses the power of the computer to help users cope with the complexity of the security system. A rule based system can use knowledge about the behavior of trusted programs and knowledge about the tricks attackers use to extend their privileges to exhaustively analyze the ways that each user could achieve additional privileges. By dealing with the complexity of the auditing problem, this approach provides a new way to resolve the conflicting requirements of security and flexibility.

The major intellectual contribution of this work is a model for representing and reasoning about security systems. Chapter 3 presents the general model and discusses how the model represents the major concepts found in the Unix and VMS operating system. Chapter 4 describes the implementation of a Unix security check based on the general model.

The unifying principle of the model is that operational security problems arise when an attacker can modify one of the sources of information that controls a trusted program. For example, the search path that is used to convert a program name into a file name defines a security critical database for the command interpreter. If an attacker can modify part of this database, then he can gain control of the privileges available to the command interpreter by substituting his own program for the expected one.

The unifying principle leads to the concept of a controlling-file (CF). This concept includes programs, data files, directories, system tables and logical names. Any source of information that allows an attacker to gain control of a process is considered a controlling-file for that process. The principle also helps to identify the key information that must be known about programs. The only facts that must be represented about the behavior of a program are those that influence the choice of controlling-files and the set of privileges available to the program.

Chapter 3 describes a simple model that is used to express the tricks that attackers use to exploit CFs. The model has two types of objects: privileges and CFs. The privilege object represents any ticket that the operating system checks to decide whether to allow access to a protected resource. For example a Unix user-id is a privilege. Both types of objects have attributes that express facts about the security configuration. For example, a privilege object would have a `members` attribute listing the other privileges that grant direct access to this privilege. A CF object would include a `writers` attribute listing the privilege objects that grant direct write access to the CF. The CF object for a file would include a `parent-directory` attribute that refers to the CF object representing the directory that contains this file.

The goals of an attacker are expressed in terms of abstract operations on privilege and CF objects. The basic goals are `acquire(priv)` which means an attacker can gain control of a process that runs with the privilege `priv`, and `modify(cf)` which means that an attacker can change the information that a program would find if it accessed `cf`. The purpose of a security analysis is to compute the list of goals that each user can achieve. This list can then be compared against a policy specification and any differences can be reported.

An attacker trick is represent by a rule that describes the conditions under which one goal can be achieved by another goal. The conditions are expressed as a predicate on the the values of attributes. For example, if modify access to a directory grants modify access to all the files in that directory (e.g., all the files can be deleted and recreated with the desired contents), this trick can be expressed by the rule:

```
if d = f.parent-directory
then
     modify(d) grants modify(f)
```

To find out which users can acquire each privilege, the knowledge about trusted programs is used to compute the list of CFs that grant access to each privilege, and then the knowledge about attacker tricks is used to find all the ways that those CFs could be modified by each user. Section 4.3 describes this analysis process in detail.

## 7.3 Conclusions

The experience related in chapter 5 shows that the Unix security checker can uncover serious operation security problems. These security holes are created by user mistakes, not by software bugs. It appears that even a security conscious user community will make serious mistakes on a monthly basis. Most of these mistakes involved three or more levels of indirection, so they would be very hard to find manually. The experience with U-Kuang demonstrates that a rule based system can analyze the complexities of a modern security system.

One benefit of an automated security checker is that the frequency of security audits can be increased. A manual audit takes a long time so it is done rarely. The automated audits take less than a minute on Unix so they can be done frequently. These audits do not prevent operational security holes, but they do detect them quickly. Of course quick detection aids both the attacker and the administrator of the system. It may be possible to prevent holes by integrating a checker into the operating system. The checker would examine a sequence of changes to the security configuration and only apply the sequence if it leaves the system in an acceptable state. Additional benefits and limitations of rule based security checkers will become apparent after they have been implemented for other operating systems.

This research touched on two areas that are ripe for further research. Users need to express their desired access policy, but languages for expressing these policies do not exist. It is clear that a vocabulary based on read and write operations is not sufficiently expressive to meet even existing needs. It is not clear what would make a better vocabulary or what would serve as appropriate operators for combining pieces of a policy specification. Another area to explored is the use of rule based systems to synthesize security configurations from policy specifications. This area is has several computability problems, but carefully chosen restrictions might avoid these problems in practice.

# References

**1.** Bell, D.E. and LaPadula, L.J. Computer Security Model: Unified Exposition and Multics Interpretation. ESD-TR-75-306, The MITRE Corporation, Bedford, MA, June, 1975. .

**2.** Bobrow, D.G. and Winograd, T. "An Overview of KRL, A Knowledge Representation Language". *Cognitive Science 1*, 1 (1977), 3-46.

**3.** Brand, S. (Editor). Department of Defense Trusted Computer System Evaluation Criteria. CSC-STD-001-83, DOD Computer Security Center, 1983.

**4.** Davis, R., Buchanan B. and Shorliffe E. "Production Rules as a Representation for a Knowledge-Based Consultation Program". *Artificial Intelligence 8*, 1 (1977), 15-45.

**5.** DEC. VMS Security Manual. Digital Equipment Corp., 1981.

**6.** Denning, D.E. "Certification of Programs for Secure Information Flow". *CACM 20*, 7 (July 1977), 504-513.

**7.** Dion, L.C. A Complete Protection Model. Proc. Symposium on Security and Privacy, IEEE Computer Society, New York, NY, 1981, pp. 49-55.

**8.** Fikes, R. and Kehler T. "The Role of Frame-Based Representation in Reasoning". *CACM 28*, 9 (September 1985), 904-920.

**9.** Gibson, W.. *Neuromancer*. ACE Books, New York, NY, 1984.

**10.** Grampp, F.T. and Morris, R.H. "Unix Operating System Security". *Bell Laboratories Technical Journal 63*, 8 (October 1984), 1649-1671.

**11.** Harrison, M.A., Ruzzo, W.L. and Ullman J.D. On Protection in Operating Systems. Proc. 5th Symposium on Operating System Principles, ACM, Nov, 1975.

**12.** Honeywell Federal Systems Division. SCOMP Trusted Computer Base. Honeywell Information Systems, Inc., McLean, VA, July, 1984.

**13.** Lampson, B.W. "Protection". *Operating Systems Review 8*, 1 (January 1974), 18-24. .

**14.** Levesque, H.J. and Brachman R.J. A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version). Proc. CSCSI-84, London, Ontario, 1984, pp. 141-152.

**15.** Lipton, R.J. and Snyder, L. "A Linear Time Algorithm for Deciding Subject Security". *JACM 24*, 3 (December. 1977), 455-464. .

**16.** Lockman, A. and Minsky, N. "Unidirectional Transport Rights and Take-Grant Control". *IEEE Trans Soft. Eng. 8*, 6 (November 1982), 597-604.

**17.** McDermott, J. "R1: The Formative Years". *AI Magazine* , 2 (1980), 21-29.

**18.** Organick E.. *The Multics System: An Examination of its Structure.* MIT Press, Cambridge, MA, 1971.

**19.** Raphals, L.A. and Chassell, R.J.. *IKE 1.0 Users Guide.* LISP Machine Inc., Cambridge, MA, 1986.

**20.** Rees, J. and Clinger W. (Editors). Revised$^3$ Report on the Algorithmic Language Scheme. Massachusetts Institute Technology, 1985.

**21.** Sacerdoti, E. *Nets of Action Hierarchies.* Ph.D. Th., Stanford Univ., 1975.

**22.** Sandhu R.S. Analysis of Acyclic Attenuating Systems for the SSR Protection Model. Proceedings of the 1985 Symposium on Security and Privacy, IEEE Computer Society, Silver Spring, MD, 1985, pp. 197-206.

**23.** Snyder, L. "Formal Models of Capability-Based Protection Systems". *IEEE Trans. Comp. C-30*, 3 (March 1981), 172-181.

**24.** Weissman C. Security Controls in the ADEPT-50 Time-Sharing System. Proc. AFIPS 1969 FJCC, AFIPS Press, Montvale, NJ, 1969, pp. 119-133.

**25.** Wood and Kochran. *System Library Series.* Volume :*Unix System Security.* Hayden Books, 1986.

**26.** Wulf, W., et al. "HYDRA: The Kernel of a Multi-processor System". *CACM 17*, 6 (June 1974), 337-345.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution is unlimited. |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>MIT/LCS/TR-401 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>DARPA/DOD N00014-83-K-0125 |
| 6a. NAME OF PERFORMING ORGANIZATION<br>MIT Lab for Computer Science<br>6b. OFFICE SYMBOL *(If applicable)* | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research/Dept. of Navy |
| 6c. ADDRESS *(City, State, and ZIP Code)*<br>545 Technology Square<br>Cambridge, MA 02139 | 7b. ADDRESS *(City, State, and ZIP Code)*<br>Information Systems Program<br>Arlington, VA 22217 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>DARPA/DOD<br>8b. OFFICE SYMBOL *(If applicable)* | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS *(City, State, and ZIP Code)*<br>1400 Wilson Blvd.<br>Arlington, VA 22217 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE *(Include Security Classification)*

RULE BASED ANALYSIS OF COMPUTER SECURITY

12. PERSONAL AUTHOR(S)

Baldwin, Robert W.

| | | | |
|---|---|---|---|
| 13a. TYPE OF REPORT<br>Technical | 13b. TIME COVERED<br>FROM ________ TO ______ | 14. DATE OF REPORT *(Year, Month, Day)*<br>1988 March | 15. PAGE COUNT<br>92 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)* |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | computer security, expert systems, rule based systems, security analysis, operational security, Unix, Kuang |
| | | | |
| | | | |

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Computers are rarely as secure as they could be. Users are lax or inconsistent in the ways they configure a computer's protection system, and these user mistakes often lead to serious security holes. For example, a privileged user might accidentally make his login initialization file publicly writable and that mistake could allow ordinary users to acquire super-user privileges. This sort of operational security problem is not caused by software bugs. It can happen even if all the computer's trusted programs behave according to their specifications. Operational security problems arise from complex interactions between the pieces of a computer's protection system.

This report describes a tool for improving the operational security of discretionary access control systems. The tool is a rule based system that knows about the behavior of the computer's software and the tricks used by attackers. The tool uses this knowledge to deduce the set of privileges directly or indirectly accessible to each user. Once the set of accessible privileges has been deduced, that set can be compared against a site

| | | |
|---|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Judy Little | 22b. TELEPHONE *(Include Area Code)*<br>(617) 253-5894 | 22c. OFFICE SYMBOL |

83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

★U.S. Government Printing Office: 1985—507-047

Unclassified

19. specific access policy and any differences can be reported.

A prototype of this tool has been used at MIT to improve the security of its Unix computers. About twice each month the prototype identifies a database entry or file access mode that has been changed incorrectly and and accidentally allows untrusted users to acquire super-user privileges.