CERIAS Tech Report 2009-37
Analysis of access control policies in operating systems
by Hong Chen
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Hong Chen

Entitled

Analysis of Access Control Policies in Operating Systems

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Ninghui Li
Chair

Elisa Bertino

Dongyan Xu

Xiangyu Zhang

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Ninghui Li

Approved by: Aditya Mathur / William J. Gorman 23 Nov, 2009
Head of the Graduate Program Date

Graduate School Form 20
(Revised 6/09)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Analysis of Access Control Policies in Operating Systems

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Hong Chen
Printed Name and Signature of Candidate

11/29/2009
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

ANALYSIS OF ACCESS CONTROL POLICIES IN OPERATING SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Hong Chen

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2009

Purdue University

West Lafayette, Indiana

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Dedicated to my family for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Ninghui Li for his continuous guidance and encouragement. Prof. Li taught me how to do scientific research and helped me to identify important and interesting research problems. He sets an example for me by a researcher's quest for innovation, quality and efficiency.

I would like to thank Prof. Elisa Bertino, Prof. Cristina Nita-Rotaru, Prof. Dongyan Xu and Prof. Xiangyu Zhang for serving in my thesis committee and/or my preliminary exam committee. Their insightful suggestions improved the quality of the dissertation. I also thank the faculty members of the Department of Computer Science, for the courses and seminars they offer that help me to lay a solid foundation for the works in this thesis.

I would like to thank Dr. Crispin Cowan for being the shepherd of our paper which was published in NDSS'09. We are grateful to the anonymous reviewers of the papers that we submitted or published for their careful reading of the papers and helpful comments.

I would like to thank my fellow graduate students, Jing Dong, Chris Gates, Tiancheng Li, Zhiqiang Lin, Ziqing Mao, Ian Molloy, Qun Ni, Wahbeh Qardaji, Qihua Wang, Hao Yuan, among others, for the discussions we had on research that inspired new ideas and for the help on technical issues. I also thank my fellow graduate students for making my stay in Purdue a blissful and memorable one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Chen, Hong Ph.D., Purdue University, December 2009. Analysis of Access Control Policies in Operating Systems . Major Professor: Ninghui Li.

Operating systems rely heavily on access control mechanisms to achieve security goals and defend against remote and local attacks. The complexities of modern access control mechanisms and the scale of policy configurations are often overwhelming to system administrators and software developers. Therefore, mis-configurations are common, and the security consequences are serious. It is critical to have models and tools to analyze thoroughly the effectiveness of access control policies in operating systems and to eliminate configuration errors.

In this dissertation, we propose an approach to systematically analyze access control policies in operating systems. The effectiveness of a policy can be evaluated under attack scenarios. An attack scenario consists of the initial resources an attacker has and the attacker's objective. Attacks under an attack scenario are encoded in a host attack graph. Compared to existing solutions, our approach is more comprehensive and does not rely on manually defined attack patterns.

Based on the model, a tool called VulSAN is implemented to analyze policies in Linux systems, and a tool called WACCA is implemented to analyze policies in Windows systems. We analyze policies in Ubuntu, Fedora, SUSE Linux and Windows Vista. We discuss the results and show the possibilities to improve the quality of protection. The results are also used to compare the effectiveness of SELinux and AppArmor policies in a version of Ubuntu Linux.

1 INTRODUCTION

Access control is a critical part of a secure operating system which "provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system" [1]. Many of the security goals of operating systems are related to the CIA triad of information security: *confidentiality*, *integrity* and *availability*. For example, when a computer is shared by multiple users, the operating system needs to protect data privacy (confidentiality) of each individual user such that other users cannot view the user's data without her consent. An operating system usually needs to protect the integrity of its critical system files such that all changes to these files are desirable. A Linux server shared by multiple remotely connected users may want to ensure that a user cannot disrupt the availability of the computing resources, e.g., a non-administrator cannot shut down the server.

Three important concepts in access control are *subject*, *object* and *operation*. In operating systems, subjects are usually *processes*, which are instances of running programs. Typical objects include files, sockets, etc. For each type of objects, there are a number of operations defined upon them, e.g., a process can read, write or execute a file. The access control *mechanism* in an operating system specifies a framework to authorize requests by processes to perform operations on objects. For example, a simplified mechanism to protect system integrity might be: every process has an integrity label with value low or high, and every object has an integrity label with value low or high; a "high" process can access all objects and a "low" process can only access "low" objects; a process's label is defined by the label of the program it runs. Under this mechanism, the integrity of "high" objects are protected against "low" processes. An access control *policy* (or *configuration*) in an operating system consists of information that can be used by the mechanism to make authorization decisions. In the above mechanism, a policy comprises the values of all the labels of processes and objects. The access control mechanisms of all computers that use

a same operating system product are the same. Usually the policy on each computer can be *configured* and policies on different computers are different. Given a security goal, an access control mechanism decides the maximum protection an access control system can provide, which is offered by the "best" configuration of the access control policy. The actual level of protection a system receives is determined by the policy deployed in the system.

The access control *enforcement component* in an operating system enforces access control policies. To ensure the effectiveness of the enforcement, it is critical that all access requests are received by the enforcement component and all the authorizations are made by the enforcement component. This principle is referred to as *complete mediation* [2].

Conceptually, an access control policy describes a Lampson access matrix [3]. The rows of the matrix represent processes, and the columns of the matrix represent objects. The element of row i and column j contains the allowed operations by process i to object j. When a process requests to access an object, the enforcement component authorizes the access if the requested operation is in the corresponding element of the matrix. There are two ways to implement an access matrix. The first one is based on *capabilities* [4]. In capability-based systems, the subjects (processes) hold *credentials* which enable the access to various objects. When a request is made, the access control enforcement component examines the credentials of the subject to make the decision. The second approach is based on *access control lists* (ACL). In this approach the information is stored on the objects. Each object is associated with a list which specifies what subjects can access it. When a request is made, the access control enforcement component examines the ACL of the object to make the decision. It is natural to implement *Discretionary Access Control* (DAC) using access control lists. In DAC, each object has an owner and the owner of an object decides who can access the object. In other words, the accesses to the object is at the discretion of the object owner. To implement DAC with ACL approach, the access control enforcement component can give the permission to the owner of an object to modify the ACL of the object. Most of today's commercial off-the-shelf operating systems, e.g., Microsoft Win-

dows, Linux and Mac OS, adopt the access control list approach as the main mechanism. DAC is usually a major part of the access control mechanisms of these operating systems.

An access control policy is configured by different parties. When an operating system is newly installed, the access control policy is the *default policy* that is shipped with the operating system. The vendor of the operating system configures the default policy. The default policy protects the integrity of critical system objects and provides a framework for further configurations. When software products are installed in the system, the access control policy is usually augmented by the software vendor to control the access to newly installed applications. For example, a newly installed browser can set up critical executables to be modified only by system administrators, and set up per-user data (cookies, browser cache, etc) to be accessed only by the owner of the data. System administrators may configure the access control policy to achieve various goals, e.g., to share files among a group of users. Normal users can configure the policy of their own files. For example, they can change the configuration to share files with others.

In the DAC mechanisms in operating systems, a process is usually associated with an account. When a user executes a program, a process is created and is associated with the user's account. The user account is a *security principal*. The process thus acts on behalf of the user, and the process has all the privileges of the user when it accesses objects. The scenario is well supported by DAC, since the ACL of each object specifies who can access the object. This scenario works perfectly if the user fully understands the functionalities of the program and the program reflects the user's intention faithfully. However this assumption does not always hold. DAC is vulnerable to Trojan horse attacks. A Trojan horse (or trojan) is a computer program that appears to be benign and perform useful functions, but actually contains some hidden and usually malicious actions. When a trojan is executed by a user, the trojan program runs with the user's privileges. It can do a variety of malicious things, e.g., to steal the user's bank accounts. It is a common practice for some desktop users to always use an administrator account for daily tasks [5]. If a system administrator executes a trojan program, the trojan can compromise the integrity of the whole system. Another attack to DAC is by exploitations of software vulnerabilities. When a program contains

vulnerabilities, an attacker may exploit the program and control the process which runs the program. The exploitation essentially turns the program into a Trojan horse. Given the fact that many processes that are associated with high privileged accounts, such as some Linux daemon programs and Windows service programs, are connected to the network and receive remote data, remote exploitations can incur great damage.

To protect host systems from Trojan horse attacks and exploitations of buggy software, and to strengthen the DAC mechanisms in general, many security enhancement mechanisms have been proposed. The enhancement mechanisms include Windows User Account Control [6], Security-Enhanced Linux [7], AppArmor [8], Systrace [9], etc. The mechanisms improve the security of operating systems and usually increase the complexity of the protection systems.

Access control in operating systems faces new challenges today, when a large number of new applications emerge, computers become highly connected, and attacks become more sophisticated. These challenges, especially the ones to analyze access control policies, motivate the works in this dissertation.

1.1 Problems and Challenges

The threat model of access control in operating systems is dynamic and constantly changing. The attack surfaces of systems constantly become larger. Intuitively, a system's attack surface denotes "the set of ways in which an adversary can attack the system. Hence the larger the attack surface, the more insecure the system" [10]. One source of enlarged attack surfaces is the ever-growing complexity of web browsers. A web browser has become a powerful application, with the ability to execute code of several scripting languages, with a large number of available browser plugins and with many browser extensions. A web browser itself also becomes complicated to support various features. Several types of attacks can be based on browsers. Attackers can use malicious scripts to initiate attacks, e.g., to trick the users into downloading Trojan horses. They can compromise a browser plugin, such as a multimedia player, by providing malicious content and exploiting a bug in the

plugin. They can distribute malicious browser extensions to steal sensitive information in web applications. They can even directly exploit browse vulnerabilities. Given the fact that many end users use web browsers intensively for everyday task and the existence of several ways to attack, web browsers have become a major target of attackers. New access control mechanisms are applied to strengthen web browser security. One of the mechanisms is the Mandatory Integrity Control (MIC) [11], which is introduced in Windows Vista and Windows Server 2008. In MIC, each process is assigned an access level and each securable object is assigned a protection level. There are four access or protection levels: low, medium, high and system (from lower ones to higher ones). By configuration, policies can restrict the accesses from processes with lower levels to objects with higher levels. For example, processes with "low" level may be restricted from reading, writing, or executing files with "medium" level, hence the confidentiality and integrity of "medium" level files can be preserved. In Windows Vista, Internet Explorer can run in protected mode, in which the browser process is labeled with "low" integrity level, therefore the damage of a compromised browser can be mitigated.

Besides using new mechanisms, operating systems also fight threats by fine-grained policy configuration. In Windows versions before Windows Server 2003, many service programs run under Local System account which is highly privileged. Windows services are objects that are used to manage long-running programs. Service programs usually run with privileged accounts and interact with network traffic or local processes. For example, an ftp/http server program or an antivirus scanner can be configured as a service program. While it is convenient to give these programs a privileged account so they can access any object in the system, this practice violates the principle of least privilege [2] because many such programs do not need the Local System account. Many of these programs receive network traffic. An attacker can take over a host completely by remotely compromising one such service program. To strengthen security of service programs, account Network Service and Local Service are introduced in Window Server 2003 [12]. The two accounts are assigned less privileges than Local System. Several service programs run under these two

accounts in Windows Server 2003 and later versions of Windows. If the programs are compromised, the attacker only gets restricted privileges.

When access control mechanisms and policies constantly expand according to the threat model, they also become more complicated. Another reason that the policies are complicated is due to the typical large number of applications installed in an operating system and the dynamic security goals they want to achieve. For example, in Windows there are about 15 types of securable objects [13], and each type is associated with up to 32 access rights which correspond to different operations to access a type of objects [14]. Virtually every object in the system, such as a file, a registry key, a service, a process, a pipe, etc, is a securable object and is guarded by a security descriptor [15]. The security descriptor of an object contains an access control list which defines what accounts can access the object and by what operations. Understanding the effects of a security descriptor is nontrivial [16, 17], since different entries in the access control list may conflict with each other. The policy of the operating system is comprised of security descriptors of all securable objects in the system, which is a large number in a typical Windows host. Therefore the number of possible configurations of the policy is usually large.

It is important to know what protection can the mechanisms and policies provide. For example, one question might be asked is what are the actual effects by the introduction of NetworkService and LocalService to reduce the attack surface. However, the complexity often makes the mechanisms and policies difficult to understand by normal users, system administrators and software developers. Given the scale of possible configurations and the difficulty to thoroughly understand the mechanisms, host systems can easily be mis-configured [18]. Since access control is a major mechanism to provide user isolation and to protect operating systems against local privilege escalation attacks and remote attacks, access control mis-configurations could lead to serious security consequences. An attack to Windows services is described in [18]. As discussed before, Windows services are objects that are used to manage long-running programs. Services can be started, stopped, paused, queried, configured, etc. In the attack, a Windows service is mis-configured such that unprivileged users have the "SERVICE_CHANGE_CONFIG" access right over the service.

This access right is required to call the ChangeServiceConfig function, which changes the configuration parameters of a service. An unprivileged user, e.g., a *guest* user, can therefore get control of a privileged process by providing his own executable and configuring the service to run the executable under accounts such as *Local System*.

A number of studies have been conducted to analyze the effectiveness of access control policies in operating system, for both Linux systems [19–25] and Windows systems [5, 18, 26–32]. Most of the existing works first formalize and identify a security goal of the policy, and then use various techniques to verify whether a policy complies with the goal. The security goals and the models of access control mechanisms are usually targeted on a particular platform and mechanism. The lack of a generic approach makes it difficult to generalize existing solutions and to compare policies of different access control mechanisms. For example, there are several approaches that aim to strengthen the traditional Linux access control [8, 9, 33], and there are debates on which mechanism is better [34]. However, such debates often center on the mechanisms and lack the actual comparison of the securities offered by the policies shipped with the protection systems. Some existing solutions provide policy analysis over several mechanisms. However, to analyze the policy one usually needs not only to manually define a high level security goal, but also to manually define the details of a violation of the goal, e.g., by giving the pattern of a type of attacks.

1.2 An Approach to Analyze Access Control Policies

In this dissertation, we provide a generic approach to analyze access control policies in operating systems. Intuitively, we try to find attacks to answer questions such as: What does it take for an attacker to penetrate the defense of the system, e.g., to install a rootkit on the host? Can the attacker leave a Trojan horse program on the host such that when the program is later accidentally executed by a user, the host is taken over by the attacker? We analyze the policies in the context of *attack scenarios*. An attack scenario is defined by an attack objective and the attacker's initial resources. For example, "remote to full control" is

an attack scenario in which a remote attacker wants to fully control the system. A "remote" attacker can access the host through network. "Full control" of a system means the attacker can obtain the privileges of an administrator. Other attack scenarios can be "remote to leaving a trojan", "local to full control", etc.

In the analyses, we assume the assurance of access control systems. We assume that the access control systems faithfully enforce the access control policies. In general software assurance is another open and challenging problem. Particularly in access control, some systems employ the approach called *reference monitor* [35] to improve assurance. A reference monitor is a module in the system that is responsible to control all accesses in the system. The module is usually small enough so it is relatively easy to test, verify and protect the module. An implicit part of our assurance assumption is complete mediation, by which all relevant accesses are intercepted by the access control enforcement component.

Given an attack scenario, an *attack* consists of a series of *actions* that can realize the attack scenario. In other words, when an attacker has the initial resources, the attack objective can be achieved if the actions can be taken. In the previously discussed attack to Windows services, there are two actions: configure service and start service. To automate the reasoning of attack discovery, we model the effects of actions as the change of *system states*. The system state represents the attacker's control of the system. For example, in SELinux $\mathsf{proc}(\mathsf{uid}, \mathsf{gid}, \mathsf{domain})$ denotes that the attacker controls a process that has user id uid, group id gid and domain domain. By taking actions, an attacker can change the current state. For example, by launching utility tool insmod, an attack can change the process state from $\mathsf{proc}(0, 51, \mathsf{sendmail\_t})$ to $\mathsf{proc}(0, 51, \mathsf{procmail\_t})$ (if defined by the policy). At some states, the attacker's objective is achieved. For example, in SELinux the attacker may want to control a process with the root user id, group id and the unconfined domain, which can be represented as $\mathsf{proc}(0, 0, \mathsf{unconfined\_t})$.

An action can be taken when all the *pre-conditions* of the actions are satisfied. A pre-condition can be a system state or a *system fact*. A system fact can be a policy rule or some information of the system. The pre-conditions to configure a service are as follows: (1) the attacker controls a certain process (a system state) and (2) there exists a service

(system information) such that (3) the ACL of the service allows the process to configure the service.

After manually defining the form of system states, the pre-conditions to carry out each action, and the effects of each action, we propose a methodology to automatically generate all possible attacks under an attack scenario. We use a *host attack graph* to represent all the attacks found. A host attack graph is a directed graph. Each node of the graph represents a system state. If the attacker can take an action to change a state a to another state b, there is an edge from a to b. The edge is marked by the corresponding action. The attacker's initial resources correspond to some nodes (called *initial nodes*) that do not have incoming edges. The attacker's objective corresponds to some nodes (called *objective nodes*) that do not have outgoing edges. An attack corresponds to a path from an initial node to an objective node. Such a path is called an *attack path*.

After building a host attack graph, we can perform a variety of analyses on the graph. For example, we can find all the (minimal) attack paths in the graph. We can also extract all *attack patterns* in the graph. An attack pattern represents a set of similar attacks. These attacks consist of a same sequence of actions, but actions in different attacks target on different objects.

Based on the model, we build the Vulnerability Surface ANalyzer (VulSAN) to analyze Linux systems, and Windows Access Control Configuration Analyzer (WACCA) to analyze Windows systems. Both tools have similar components: a component to collect access control policy and system information, a component to generate the host attack graph, and a component to analyze the host attack graph. VulSAN has been used to analyze SELinux and AppArmor policies in several Linux distributions (including Ubuntu, RedHat and SUSE Linux). We analyze the standard policies shipped with these Linux distributions, and the analysis shows possibilities to improve the policies. The model we use is generic such that we can compare the policies between different mechanisms. We compare the policies of SELinux and AppArmor in a version of Ubuntu Linux. WACCA is used to analyze a Windows Vista system. WACCA categorizes attacks found in a configuration into different attack patterns. Several attack patterns are found in the analyses.

1.3 Structure of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses basic access control mechanisms and enhancements under Unix systems and Windows systems. DAC in Unix systems and Windows systems, User Account Control, SELinux and AppArmor are discussed. Chapter 3 presents related work. Some approaches are solutions to analyze the effectiveness of access control policies, some are proposals of various access control mechanisms. Chapter 4 gives details of the design, implementation and evaluation of VulSAN, a tool to analyze Linux policies. Chapter 5 presents the design, implementation and case studies of WACCA, a tool to analyze Windows policies. Chapter 6 summarizes the dissertation.

2 OPERATING SYSTEM ACCESS CONTROL MECHANISMS

In this chapter, we discuss some state-of-the-art access control mechanisms and enhancements under Unix systems and Windows systems.

2.1 Unix Access Control

One of the best known and most widely used access control mechanisms in operating systems is the file system access control in Unix-based systems (including Linux, Solaris, OpenBSD, and others). It implements a type of discretionary access control (DAC). Besides the simplicity of its basic form, this mechanism also has many intricate details.

Three most important concepts in Unix access control are users, subjects, and objects. From the computer's point of view, each account is a user. Each user is uniquely identified by a numerical identifier called *user id*. A user can correspond to a human user who has an account on the machine. A user account can also be create for an application. For example, a common practice to configure Apache on Debian/Ubuntu Linux is to run the web server as user www-data. The account is given privileges to access web server related resources. There is a special user called root or *superuser* which has user id 0. The root user is for administrative purposes and has full privileges in the system. A *group* contains a set of users. A user can be a member of several groups. Every group has a numerical identifier called group id. Each subject is a process, which is an instance of a running program. A process is a basic unit to issue requests to access resources. Many protected resources are modeled as files, which we call objects. As it is a DAC system, each object has a owner. The owner of an object can specify which users are allowed to access the files by what operations. For non-file privileges, the access policy is generally fixed by the system. In most systems, only the root user has these privileges. In the descriptions below, we use subjects and processes interchangeably, and objects and files interchangeably. We dissect

the Unix access control into two components: the policy specification component and the enforcement component.

The policy component determines which users are allowed to access what objects. Each object has an owner (which is a user) and an associated group. In addition, each object has 12 permission bits. These bits are:

- three bits for determining whether the owner of the file can read/write/execute the file,
- three bits for determining whether users in the associated group (other than the owner) can read/write/execute the file,
- three bits for determining whether all other users can read/write/execute the file,
- the SUID (set user id) bit, the SGID (set group id) bit, and the sticky bit. Only the SUID bit is relevant for our discussions below.

The file owner, group and permission bits are part of the metadata of a file, and are stored on the inode of a file. The owner of a file and the root user can update these permission bits, which is the discretionary feature. In most modern Unix-based systems, only the root can change the owner of a file. The *DAC policy* of a Unix-based system is defined as the collection of the owner, associated group and permission bits of all the objects in the system.

The policy component specifies only which users are authorized, whereas the actual requests are generated by subjects (processes) but not users. The enforcement component fills in this gap. It tries to determine on which users' behalf a process is executing. Each process has an effective user id (euid), which determines the access privileges of the process. The first process in the system has euid 0 (root). When a user logs in, the process's euid is set to the id corresponding to the user. When a process loads a binary through the execve system call, the new euid is unchanged except when the binary file has the SUID bit set, in which case the new euid is the owner of the file.

The SUID bit is needed for two reasons. First, the granularity of access control using files is not fined-grained enough. For example, the password information of all the users

is stored in the file /etc/shadow. A user should be allowed to change her password. However, we cannot allow normal users to write the shadow file, as they can then change the passwords of other users as well. To solve this problem, the system has a special utility program (passwd), through which a user can change her password. The program is owned by the root user and has its SUID bit set. When a user runs the passwd program, the new process has euid root and can change /etc/shadow to update the user's password. Note that this process (initiated by a user) can perform anything the root user is authorized to do. By setting the SUID bit on the passwd program, the system administrator (i.e., the root user) trusts that the program performs only the legitimate operations. That is, it will authenticate a user and change only that user's password in /etc/shadow. Any program with SUID bit set must therefore be carefully written so that they do not contain vulnerabilities to be exploited. Second, non-root users often need to use the non-file privileges. For example, a user may need to mount a CD, which requires a non-file privilege that is available only to the root user. This requires the mount utility program to be owned by root and has the SUID bit set. For historical reasons, there are different versions of system calls related to SUID bit, and the semantics of them are sometimes conflicting and confusing [36].

2.2 Windows Access Control

In Windows, there are a variety of objects, such as files, services, registry keys, pipes, etc. A process can access a type of objects in a number of ways. For example, a process can read/write/execute a file, and a process can start/stop/configure a service. The permissions that are needed to carry out the accesses are called *access rights*. For example, FILE_WRITE_DATA is the access right to write a file and SERVICE_CHANGE_CONFIG is the access right to configure a service. Some access rights are common to all types of objects, and these permissions are called *standard access rights*. For example, the DELETE standard access right, which is required to delete an object, is applicable to most types of objects. Some access rights are applicable to a specific type of objects, e.g., the

PROCESS_VM_READ access right is required to read memory in a process, hence it is a process-specific access right.

Each process runs on behalf a user, and each user may be a member of several groups. For example, a user account john may be a member of groups Administrators, Everyone, AuthenticatedUsers, etc. Each user or group corresponds to a unique identifier called *Security Identifier*, or *SID*. For example, the group Administrators corresponds to the SID S-1-5-32-544. Users and groups are represented by SIDs in internal data structures.

Each process is assigned an *Access Token*, which is issued and maintained by the operating system. The access rights of a process depend on its access token. An access token contains the SID of the process user, the SIDs of the groups that the user is a member of, and a list of *privileges* held either by the user or by the groups. Privileges enable the process to perform some system operations, e.g, SeDebugPrivilege is required to debug other processes and SeShutdownPrivilege is required to shut down the machine.

The basic Windows access control mechanism is a type of discretionary access control (DAC). The owner of an object decides which accounts can access the object. The access information of an object is stored in the object's *security descriptor*. A security descriptor contains the owner's SID, a *discretionary access control list* (DACL) and a *system access control list* (SACL). DACL is used to guard the accesses to an object. It consists of a number of *access control entries* (ACEs), each of which either grants or denies a set of access rights to an SID. For example, the security descriptor may contain an ACE that grants the BackupOperators group the access rights FILE_READ_DATA, FILE_WRITE_DATA, FILE_APPEND_DATA, etc. The owner of an object by default has READ_CONTROL and WRITE_DAC access rights to the object, which allows the owner to read/write the DACL. Any access right is implicitly denied to any SID, if there is no ACE that grants the right to the SID. When an allowing ACE and a denying ACE conflict with each other, the denying ACE overrides the allowing one. Similar to a DACL, an SACL consists of a list of ACEs, each of which triggers the logging system to audit a set of accesses carried out by an SID.

To summarize, when a process wants to access an object, the operating system decides whether to grant the access by (1) the object type (2) the access rights that are requested

(3) the access token of the process (4) the security descriptor of the object. The operating system checks if the requested rights are allowed by the security descriptor to the access token.

A common practice makes it difficult to secure a Windows system using above mechanism. Given the popularity of windows system in end-user desktops, the users are often the administrators of their system. In order to make it easy to perform tasks such as installing software or changing system settings, users often log in using an administrative account [5]. As a result, if an attacker is able to launch a Trojan horse attack, or exploit some programs of the user, the attacker is able to acquire full privileges. The attacker can thus change the critical parts of the operating system and take over the system silently.

In order to provide enhanced security for users, Windows Vista features an access control mechanism called User Access Control (UAC) [6]. In UAC, when an administrator logs in, the user's processes are granted two access control tokens instead of just one: an administrator access token that has full privileges and a standard user access token. When the user performs normal tasks, e.g., browsing the web, reading emails, only the standard user access token is involved in access control decisions. When the user wants to perform some administrative tasks, e.g., install a program or a driver, Vista will prompt to ask for user's consent and the administrator access token is used afterwards. Therefore the user is alerted when the privileges are used, and a Trojan cannot take over the system silently. This mechanism requires a user to be able to correctly tell whether extra privileges is required for a particular activity and not to blindly click through (or enter the password) each time. While securing the system, this feature also introduces usability problems as a user is often required to confirm critical operations.

2.3 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a security mechanism in Linux that has been developed to support a wide range of security policies. SELinux is an enhancement to the DAC mechanism in Linux. When SELinux is enabled, a request of access is granted

only when both DAC and SELinux authorize the access. SELinux was first implemented as kernel patches and currently it is implemented using the Linux Security Module (LSM) framework. The architecture of SELinux separates policy decision-making logic from the policy enforcement logic. SELinux policies include features as Type Enforcement, Role-Based Access Control, Multi-Level Security, etc. We discuss the architecture of SELinux and some policies as follows.

In SELinux, every subject (process) and object (resources like files, sockets, etc) is given a *security context*, which includes a *type*. Every process has a *domain*, which is a type, and every object (e.g., files) has a type. All processes with a same domain are treated identically. The objects (resources) are categorized into *object security classes*. Each object security class represents a particular kind of objects, e.g., regular files, folders, TCP sockets, etc. For every object security class, there is a set of access operations that can be performed, e.g., the operations to a file include read/write/execute, lock, create, rename, getattr/setattr, link/unlink, etc. All objects with the same type and the same class are treated identically. When an access attempt is made by a process, the enforcement part will decide whether to grant the access based on the security context of the process, the security context of the object being accessed, and the object security class of the resource.

An SELinux policy consists of following types of rules (TE is short for Type Enforcement):

- A TE access vector rule defines what operations a process with a particular domain can perform on an object of a particular security class with a particular a type. For example, the rule

  allow sshd_t sshd_exec_t:file read execute entrypint;

  says a process with the sshd_t domain can perform three operations on a file with the sshd_exec_t type: read, execute, and entrypoint. The meaning of read and execute is the same as in the DAC policies of Unix. The entrypoint permission means that the sshd_exec_t type is a legitimate entrypoint for the sshd_t domain, i.e., a process is allowed to enter sshd_t domain by executing a file with sshd_exec_t type.

An access vector can also start with auditallow, auditdeny, or dontaudit, in addition to allow. Operations are denied unless there is an explicit allow rule. When an access request is granted, it is not logged unless it is authorized by an auditallow. And when an access request is denied, it is logged unless it is authorized by a dontaudit rule.

- A TE transition rule for a process defines which new domain a process should enter after executing a program, based on the current process domain and the type of the program. For example, the following rule

  type_transition initrc_t sshd_exec_t:process sshd_t;

  says when a process with the initrc_t domain executes a program with the sshd_exec_t type, the process should transit to the sshd_t domain. The domain transition can occur only if two additional conditions are met: (1) the initrc_t domain is allowed to execute files of the sshd_exec_t type; (2) the sshd_exec_t type is a legitimate entrypoint for the sshd_t domain. Both conditions are specified by TE access vector rules.

- A TE transition rule for an object defines the type of a newly created object. For example, the rule

  type_transition sshd_t tmp_t:dir file sshd_tmp_t;

  says when a process with the sshd_t domain creates a file or a directory in a directory with the tmp_t type, the newly created file or directory should be with the sshd_tmp_t type.

Current SELinux policies are complicated. There are 42 kernel object classes , 14 userland object classes, hundreds of possible permissions [37]. A typical policy contains thousands of policy rules. To reduce the size of policies, macros are used to make templates for frequent used patterns that group permissions together.

2.4 AppArmor

AppArmor is an access control enhancement to Linux that confines the access permissions on a per program basis. It follows the principle of least privilege. Similar to SELinux, AppArmor works on parallel with the Linux DAC. For every protected program, AppArmor defines a list of permitted accesses, including file accesses and capabilities. The list for a program is called the program's profile. The profiles of all protected programs constitute an AppArmor policy. A program's profile contains all possible file reads, file writes, file executions and capabilities that may be performed by a protected program. Under AppArmor, a process that executes a protected program can only perform accesses in the program's profile.

By following the principle of least privilege, AppArmor makes local and remote exploits more difficult. Consider a system that runs an FTP server using the root account. If an attacker exploits a vulnerability in the server and injects her own code, under normal Linux DAC protection, the attacker is able to gain full privileges in the system. The attacker can, e.g., install a rootkit by loading a kernel module. However, if the system is protected by AppArmor, there will not be a kernel module loading capability in the FTP server's profile because a FTP server wouldn't need that. Therefore even if the attacker controls the server process, she cannot directly install a rootkit.

An excerpt of a profile for passwd [38] is shown in Figure 2.1. The profile confines the program such that if a local user exploits this setuid root program, the user cannot get full privileges of root. In the profile, there are 2 rules for capabilities (line 4 and 5) and 11 rules for file accesses (line 6 through line 16). A file rule consists of a file name and one or more permitted access modes. There are 9 access modes in total: read mode, write mode, link mode, and six other modes for executing the file. For details of these access modes, please refer to [8].

A profile can be created using AppArmor utilities. One can run a program in the "learning mode". In this mode, all the requests of a program are permitted and logged. The user makes the program perform as many accesses as possible. Later the user can use the logs

```
1. ...
2. /usr/bin/passwd {
3. ...
4. capability chown,
5. capability sys_resource,
6. /etc/.pwd.lock w,
7. /etc/pwdutils/logging r,
8. /etc/shadow rwl,
9. /etc/shadow.old rwl,
10. /etc/shadow.tmp rwl,
11. /usr/bin/passwd mr,
12. /usr/lib/pwdutils/lib*.so* mr,
13. /usr/lib64/pwdutils/lib*.so* mr,
14. /usr/share/cracklib/pw_dict.hwm r,
15. /usr/share/cracklib/pw_dict.pwd r,
16. /usr/share/cracklib/pw_dict.pwi r,
17. }
```

Figure 2.1. A Sample AppArmor Policy for /usr/bin/passwd

to create the profile of the program. For each access, an AppArmor utility asks the user whether to allow the access; and if the access is a file access, the user can choose to generalize the access by using wildcards in the permitted filename (globbing).

AppArmor also provides finer-grain access control than process level, by the "ChangeHat" feature. ChangeHat-aware programs can use this feature to have part of a program use a different profile.

AppArmor identifies a number of programs that could be major targets of attackers. These programs are confined and protected by profiles. If a program has no policy associated with it, then it is by default not confined. If a program has a policy, then it can access only the objects specified in the policy. This approach remains vulnerable to Trojan horse attacks. As most programs, such as shells, obtained through normal usage channels are unconfined, a user would mostly operate in an unconfined environment, and the execution of a Trojan horse program will not be under the control of any policy rules.

3 RELATED WORK

To tackle the challenges to configure access control policies, there are generally two types of approaches. The first type of approaches aim to provide better models and tools to analyze existing policies, especially ones that are widely used. The second type of approaches create access mechanisms other than the prevailing DAC mechanisms, aiming to offer better usability, better protection, etc. In this chapter we present related works of the two types of approaches.

3.1 Analyses of Access Control Policies

The access control of Unix-like systems has been studied in many works. Existing approaches for analyzing SELinux security policies include Gokyo [21, 22], SLAT [23], PAL [39], APOL [20, 40], SELAC [41], NETRA [26], and PALMS [24]. Gokyo [21, 22] identifies a set of domains and types as the implicit Trusted Computing Base (TCB) of a SELinux policy. Integrity of the TCB holds if no type in it can be written by a domain outside the TCB. SLAT [23] verifies if a SELinux policy satisfies certain information flow goals. It answers questions such as: Is it true that all information flow paths in a system from a starting security context to a final security context go through a series of specific steps? PAL [39] provides similar functionalities to SLAT. It differs in that it is implemented in XSB, a logic programming system. This enables PAL to handle other kinds of queries. APOL [20] is a tool to analyze the relationships between domains and types in a SELinux policy. In [40] the authors augment APOL to find paths from susceptible domains to security sensitive domains. The selection of susceptible and security sensitive domains is manually done. The query language is less flexible than SLAT or PAL, but it provides a graphical user interface to display the results. SELAC [41] is a formal model to describe the semantics of a SELinux policy. The authors develop an algorithm based on SELAC

to verify if a given subject can access a given object in a given mode. NETRA [26] is a another tool for analyzing explicit information flow relationships in access control configurations. It has been applied to analyze Windows XP and SELinux policies. PALMS [24] is a tool for analyzing SELinux MLS policy, and was used to verify that the SELinux MLS reference policy satisfies the simple security property and the *-property defined by Bell and LaPadula [42].

Our work is different in the following ways. First, VulSAN supports analyzing AppArmor in addition to SELinux. Second, VulSAN utilizes the current system state (such as which files exist in the system) as well as DAC policies (such as which users can write to a file according to the DAC permission bits) in addition to the MAC policies. As shown in Section 4.3.2, considering DAC is necessary to obtain accurate analysis results. Third, our goal, which is to compute the vulnerability surface under different attack scenarios, is different from that of existing tools. In particular we need to be concerned with more than just providing a policy analysis tool; we need to also come up with appropriate ways of querying the tool and analyzing the result.

Comparing the Quality of Protection (QoP) offered by different systems is challenging because different policy models are used. For example, SELinux uses Type Enforcement (TE), and AppArmor confines security-critical programs with profiles. Currently there exists no tool to compare the security of systems protected using different technologies. There is an ongoing debate about which of SELinux and AppArmor is a better system, but such debate often centers on the mechanism and lacks actual comparison of the security offered by the standard policies shipped with these protection systems. As a result, such comparison tends to become rhetoric wars. In [34] Cowan from Novell and Riek from Red Hat debated about usability, simplicity, and policy implementation (labels vs. pathnames) between AppArmor and SELinux. QoP is not discussed in details. We believe that comparisons involving actual deployed policies are necessary. It may be theoretically possible to configure a MAC system to offer very strong protection, but it is the shipped standard policy that determines the QoP in reality, since very few people change the shipped policy.

In our approach, we perform a concrete measurement of QoP for both mechanisms using shipped policies.

Windows access control has been studied in several research works [5, 18, 26–28]. Govindavajhala and Appel use a logical model to study Windows XP [18]. A scanner reads access control information from the host, and a vulnerability analysis framework MulVAL [43] is used to discover privilege escalation attacks. The two-layer structure of NETRA [26] makes it suitable to analyze different operating systems. NETRA is used to study both Windows systems and Linux systems. WACCA differs from them in that: (1) WACCA can automatically discover attack patterns, while attack scenarios are manually defined in [18, 26]; (2) our model considers potential software vulnerabilities.

Data flows between accounts that are allowed by security descriptors of objects under Windows are studied in [28]. These data flows define the trust boundaries and can be used to study potential threats and elevation paths. EON [27] is a logic-programming language and tool to study dynamic access control systems. Windows Vista security policies is studied in EON. EON is focused on the general security mechanism, and does not have a scanner to consider specific configurations of a host. Chen et al. [5] study least-privilege incompatibilities that cause many windows users to run with administrator privileges. This exposed another aspect of misconfiguration: access rights are sometimes unnecessarily given to only privileged accounts.

Attack surface [29–32] defines a metric to measure how secure a system is. It is defined by a system's interfaces and channels that are exposed to unauthorized users. Intuitively a larger attack surface indicates a less secure system. In addition to the interfaces exposed to the attacker, WACCA also studies the follow-up steps of an attacker after the initial compromises.

Attack graph [44–48] is usually used to study the vulnerabilities of network configurations. An attack comprises a set of exploitations, and there are certain dependencies between exploitations. An attack graph is used to express the dependencies and to represent possible exploit combinations. Sheyner et al. [44] propose an approach to construct attack graphs automatically using symbolic model checking. The authors present two ways

to analyze an attack graph: (1) to generate a minimal set of atomic attacks to thwart the attacker's goal (2) to determine the likelihood that the attacker will succeed based on probability. In [45], it is proved that the minimization problem of critical attacks is polynomially equivalent to the minimum hitting set problem and a greedy algorithm is presented. Markov decision process is used to model the attack graph and the probability of the intruder's success is calculated. In [46], monotonicity is assumed such that the pre-condition of a given exploit is never invalidated by the successful application of another exploit. Based on monotonicity a scalable solution with polynomial time complexity is provided. Given an attack graph, the set of hardening measures that guarantee the safety of given critical resources is computed in [47]. Logical attack graphs are proposed in [48] to provide a scalable and efficient way to generate attack graphs. Based on MulVAL [43], the generation of a logical attack graph is in quadratic time. WACCA uses attack graphs to analyze the privilege escalation attacks in operating systems, and the analysis is based on attack pattern discovery.

Expandable Grid is proposed in [16, 17] to visualize access control policies. This approach provides a graphical interface for users to better understand access control policies which are usually presented in the form of rules or texts.

3.2 Design of Access Control Mechanisms

The limitations of DAC have been discussed in many sources, e.g., [49, 50]. Traditionally, people deal with the weaknesses of DAC by replacing or enhancing it with Mandatory Access Control (MAC). There are three classes of approaches to add MAC to operating systems: confidentiality-based, confinement-based, and integrity-based.

A well known example of confidentiality-based MAC is the Bell-LaPadula (or BLP) model [42]. Systems that implement protection models similar to BLP include Trusted Solaris and IX [51]. In BLP model, each subject has a clearance label and each object has a classification label. The values of clearance or classification labels are organized in a lattice. A system is called secure if accesses satisfy security properties. The *Simple*

Security Property (no read-up) states that a subject at a certain security level should not read an object at a higher security level. The **-Property* (no write-down) states that a subject at a certain security level should not write an object at a lower security level. Some subjects are trusted such that they can violate security properties and the system is still secure. These subjects can perform tasks such as transferring data from a high classification level to a low classification level. BLP model mainly focuses on confidentiality, which is part of the security goals of a modern operating system. It is difficult to directly apply BLP model to today's commercial off-the-shelf operating systems, since it may require a lot of components to be trusted.

Confinement-based MAC systems include SELinux [33], AppArmor [8,52], systrace [9], LIDS [53], PACL [54]. These approaches develop an access control system completely separate from DAC to offer additional protection. Some of them have been discussed in Chapter 2. Jaeger et al. [21] analyzed the SELinux example policy to separate the domains and types into those in a Trusted Computing Base (TCB), i.e., high integrity, and those are not, i.e., low integrity. They found many information flow channels from low to high, due to the nature of Linux. The approaches in AppArmor, systrace, and PACL are to identify a number of programs that, when compromised, could be dangerous, and confine them by a policy. Systrace [9] defines policies for programs at a finer granularity. Instead of defining allowed accesses to files and capabilities, systrace defines allowed system calls with parameter values for each program to confine the operations of processes.

The Biba model [55] is perhaps the earliest mandatory integrity protection model. It provides five integrity policies, which offer important insights into integrity protection and contamination tracking. LOMAC [56] is based on the subject low-water mark policy in Biba. IFEDAC can be viewed as an approach that integrates Biba's integrity tracking and DAC's policy specification and enforcement. Microsoft Vista introduced a security feature called Mandatory Integrity Control (MIC) [57], which has been discussed in Chapter 1. The approach partitions files and programs into four different integrity levels: low, medium, high, and system. A program running at one level cannot update objects that are at a higher

level. A subject's integrity level is fixed. This can be viewed as a simplified version of SELinux, where there are only four types.

Another well-known integrity model is the Clark-Wilson model [58], with follow-up work by Karger [59] and Lee [60], among others. The key idea of Clark-Wilson model is well-formed transactions. A well-form transaction changes a consistent system state and results in another consistent system state. These integrity-protection approaches have not been applied to operating systems and do not support user-specific integrity, e.g., separating one user from another.

UMIP [61] and IFEDAC [62] uses information flow to enhance discretionary access control policies in operating systems. It is argued that the vulnerabilities of DAC to Trojan attacks are due to the enforcement component rather than the policy component. The reason that DAC enforcement component is incompetent is that a single principal is responsible for any request. In reality, a process is initiated and controlled by different principals. UMIP and IFEDAC use information flow tracking to infer the actual principals behind access requests. A request is granted only when all principals responsible for the request are allowed to access a resource. UMIP is an approximation of IFEDAC, where a set of principals collapse into a value of high or low. Therefore IFEDAC is able to provide better user separation while UMIP treats users alike.

Language-based information flow security has been studied extensively in the programming language context [63–65]. The CW-Lite work [66] addresses the issue of trust by explicitly analyzing source code of programs. This line of work mainly focuses on analyzing and controlling information flow within a program. It is a slightly different context from the ones that apply to operating systems. Hicks et al. [67] proposed an architecture for an operating system service that integrates a security-typed language with MAC in operating systems, and built SIESTA, an implementation of the service that handles applications developed in Jif running on SELinux.

4 ACCESS CONTROL POLICY ANALYSIS UNDER LINUX

In this chapter, we demonstrate how to apply the analysis methodology to analyze access control policies in Linux systems. The motivation to analyze the policy is to better understand the effectiveness of the policies and to compare between policies, which is important to fight against host compromise.

Host compromise is one of the most serious computer security problems today. A key reason why hosts can be easily compromised is that the Discretionary Access Control (DAC) mechanism in today's operating systems is vulnerable to Trojan horses and the exploitation of buggy software. Recognizing this limitation of existing DAC mechanisms, in the past decade there have been a number of efforts aiming at adding some form of Mandatory Access Control (MAC) to Commercial-Off-The-Shelf (COTS) operating systems. Examples include Low Water-Mark Access Control (LOMAC) [56, 68], Security Enhanced Linux (SELinux) [33], AppArmor [8, 52], and Usable Mandatory Integrity Protection (UMIP) [61]. Some of these systems have been widely deployed. For example, SELinux is supported in a number of Linux distributions, including Fedora, Debian, Gentoo, EnGarde and Ubuntu [69], and AppArmor is supported in Linux distributions including SUSE, PLD, Pardus Linux, Annvix, Ubuntu and Mandriva [70].

Given the existence of these protection systems, a natural desire is to *understand* and *compare* the quality of protection (QoP) offered by them. A system administrator would want to know the QoP offered by the MAC system he is using. Note that by an MAC system, we mean both the mechanism (e.g., SELinux or AppArmor) and the specific policy being used in the system, because the QoP is determined by both. More specifically, it would be very useful for an administrator to know: What kinds of attacks are prevented by the MAC system my host is using? What does it take for an attacker to penetrate the defense of the system, e.g., to install a rootkit on my host? Can the attacker leave a Trojan horse program on my host such that when the program is later accidentally executed by a user, my

host is taken over by the attacker? Would it be more secure if I use a competing distribution which either has a different MAC mechanism or or has different policy settings?

Based on the framework discussed in Chapter 1, we develop a tool called Vulnerability Surface ANalyzer (VulSAN) for answering these questions. We analyze the QoP by measuring the *vulnerability surface* for *attack scenarios*. An attack scenario is defined by an attack objective and the attacker's initial resources. For example, "remote to full control" is an attack scenario in which a remote attacker wants to fully control the system. Other attack scenarios can be "remote to leaving a trojan", "local to full control", etc. A vulnerability surface of a system is a list of minimal attack paths. Each attack path consists of a set of programs such that by compromising those programs the attack scenario can be realized. Vulnerability surface is related to attack surface [30] which is a concept in the Microsoft Security Development Lifecycle (SDL). Attack surface uses the resources that might be used to attack a system to measure the attackability of the system. They are different in that vulnerability surface provides potential multi-step attack paths of a system while attack surface considers potential entrypoints of attacks. VulSAN computes the vulnerability surfaces for attack scenarios under SELinux and AppArmor. To do this, VulSAN encodes the MAC policy, the DAC policy and the state of the host into Prolog facts, and generates a host attack graph for each attack scenario, from which it generates minimal attack paths which constitute the vulnerability surface.

VulSAN can be used by Linux system administrators as a system hardening tool. A system administrator can use VulSAN to compute the host attack graphs for attack scenarios that are of concern. By analyzing these graphs, the administrator can try to harden the system by tweaking the system and policy configurations. For example, the administrator can disable some network deamon programs, remove some unnecessary setuid-root programs, or tweak the MAC (SELinux or AppArmor) policies to better confine these programs. After making these changes, the system administrator can re-run the analysis to see whether it achieves the desired objective. Because VulSAN uses intermediate representation of the system state and policy, it is possible to make the changes in the representation and to perform analysis, before actually deploying the changes to the real system. Because VulSAN

can handle both SELinux and AppArmor, which are the two MAC systems used by major Linux distributions, it can be used for most enterprise Linux distributions and home user distributions.

VulSAN can also be used to compare the QoP of policies between different systems. Such comparisons help system administrators to select which Linux distributions to use. In addition, they also help system hardening. If an administrator knows that another Linux distribution with the same services does not have a particular vulnerability path, then the administrator knows that it is possible to remove such a path while providing the necessary services, and can invest the time and effort to do so.

We have applied VulSAN to analyze the QoP of several Linux distributions with SELinux and AppArmor. Comparing the default policies of SELinux and AppArmor for the same Linux distribution (namely Ubuntu 8.04 Server Edition), we find that AppArmor offers significantly smaller vulnerability surface, while the SELinux policy with Ubuntu 8.04 offers only slightly smaller vulnerability surface compared with the case when no MAC is used. More specifically, when no MAC is used, the system has seven length-1 attack paths in the scenario when a remote attacker wants to install a rootkit. They correspond to the seven network-facing daemon programs running as root, namely apache2, cupsd, nmbd, rpc.mountd, smbd, sshd, and vsftpd. Among them, the SELinux policy confines only cupsd. This shows that the often claimed strong protection of SELinux is not realized, at least in some popular Linux distributions. We also note policies in different distributions offer different levels of protection even when they use the same mechanism. For example, the SELinux policy in Fedora 8, which is a version of the targeted policy, offers tighter protection than that in Ubuntu 8.04, which is a version of the reference policy. We also observe that Ubuntu 8.04 and SUSE Linux Enterprise Server 10 expose different vulnerability surfaces when they both use AppArmor. Also, one attack scenario that neither SELinux nor AppArmor offers strong protection is when a remote attacker leaves a malicious executable program somewhere in the system and waits for it to be accidentally executed by users, at which point the process would not be confined by the MAC system. This attack is possible for two reasons. First, both SELinux and AppArmor confine only a subset of the known

programs and leave any program not explicitly identified as unconfined. Second, as neither SELinux nor AppArmor performs information flow tracking, the system cannot tell a program left by a remote attacker from one originally in the system.

4.1 Overview of the Approach

To analyze and compare the QoP of MAC systems, we need a way to define the QoP first. Lacking such a definition prevents debates about the virtues of different systems to go beyond subjective and rhetoric arguments.

The MAC systems are motivated by the threats and attacks facing today's operating systems, thus they should be evaluated by their ability to defend against these attacks. Our approach generates all possible attack paths that can lead an attacker to control of the system. We analyze the QoP under multiple attack scenarios. Each attack scenario has two aspects. One is the objective of the attacker (e.g., load a kernel module or plant a trojan horse). The other is the initial resources the attacker has (e.g., can connect to the machine from network, or has a local account). Based on the scenario, VulSAN gives all possible attack paths.

Our approach consists of following steps:

1. Establish a running server as the analysis target.

2. Translate policy rules and system state information into Prolog facts. We write parsers for SELinux and AppArmor policies. We write scripts to collect information of the file system and running services.

3. Encode what the attacker can do to break into a system and escalate privileges in one or more steps. For each security-enhanced mechanism, we define the notion of *attack states* to describe the attacker's current privileges. For each MAC system we write a library of system rules that describe how an attacker exploits a program to cause state transition under the MAC system.

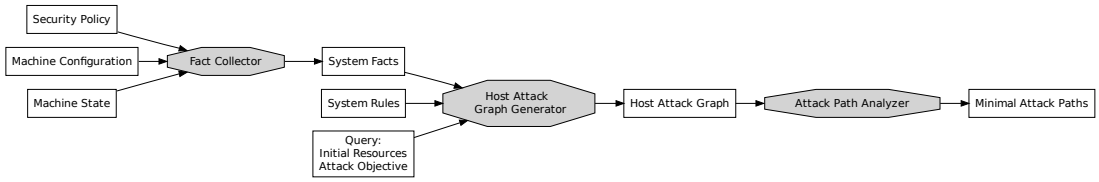


Figure 4.1. Solution Overview of VulSAN

4. Encode an attack scenario into a query, and use the query to generate the *host attack graph*. A host attack graph is a directed graph. The graph nodes are attack states, and graph edges correspond to state transitions. Edges are marked by programs, and by compromising marked programs the attacker can cause state transitions. We call the nodes of the graph that represent the attacker's initial resources *initial attack states*, and we call the nodes of the graph that represent the attack objective *goal attack states*.

5. Analyze the host attack graph. What we care about are the paths from initial attack states to goal attack states. The most interesting paths are the ones that are "minimal". VulSAN generates all the minimal attack paths.

Figure 4.1 shows the overview of our approach.

The interesting result from the host attack graph is the attack paths. An attack path is a path that starts from an initial attack state and ends with a goal attack state. Suppose there are two attack paths p_1 and p_2, and we have $V(p_1) \subset V(p_2)$ ($V(p)$ represents the set of edge labels along the path). Then we are not interested in p_2 since it is easier to realize p_1 than to realize p_2. An attack path p is desirable when there does not exist another attack path p' such that $V(p') \subset V(p)$. We call such paths *minimal paths*.

We define the *vulnerability surface* of a protection system as the set of all minimal attack paths. Each path includes the programs that must be exploited to realize the attack objective.

When we compare two protection systems A and B under the same attack scenario, we first generate the sets of all minimal attack paths of the two protection systems, called P_A and P_B. For any path $p \in P_A$, we say:

- p is a *strong* path if there exists a path $p' \in P_B$ such that $V(p) \subset V(p')$.
- p is a *weak* path if there exists a path $p' \in P_B$ such that $V(p) \supset V(p')$.
- p is a *common* path if there exists a path $p' \in P_B$ such that $V(p) = V(p')$.
- p is a *unique* path otherwise.

When comparing A and B, a common path shows a common way to exploit both systems. A strong path p of system A suggests that, if the attacker compromises the same programs in p under system B, she will need to compromise more programs to achieve the attack objective in B. A weak path p of A suggests that, compromising a subset of the programs in p under B already helps the attacker to achieve the objective in B. A unique path p of A suggests that A is more vulnerable than B because by realizing p, an attacker can compromise A but not B. By examining the strong, weak, common, and unique attack paths in details, we can better understand the differences of QoP between two systems.

There are two approaches to use the sets of minimal attack paths to compare the QoP of two systems. In one approach, one makes no assumption about whether one program is easier to compromise than another program. In this approach, one could only partially order the QoP as measured by the host vulnerability surfaces of different systems. P_A has higher QoP than P_B when all minimal attack paths for P_A are either common paths or weak paths. That is, for every minimal attack path p for P_A, either P_B has the same path, or there exists a path p' for P_B that contains a strict subset of the programs in p, which means that p' is easier to exploit than p. The strength of this approach is that the comparison result remains valid even when some programs are significantly easier to exploit than other programs. The drawback is that often times two protection systems are not directly comparable. Most of the analysis in this chapter use this approach.

In the second approach, one views each program as one unit, implicitly assuming that all programs are equal. By making this assumption, it is possible to come up with a total order among all protection systems. However, the drawback is that the validity of the assumption is questionable. In a few head-to-head comparisons in this chapter, we use this approach. Whenever we do so, we will explicitly state the assumption that all programs are considered equal.

The ideal solution is to be able to quantify the efforts needed to exploit different programs. However, this is a challenging open problem that appears unlikely to be solved anytime soon.

4.2 Implementation of VulSAN

VulSAN consists of the following components: the Fact Collector, the Host Attack Graph Generator, and the Attack Path Analyzer.

4.2.1 Fact Collector

Fact Collector retrieves information about the system state and security policy, and encodes the information as facts in Prolog.

The information about file system consists of facts of all relevant files, system users, system groups and running processes. Several sample Prolog facts are depicted in Figure 4.2. We only consider system facts that are relevant to our security analysis. Irrelevant information, like CPU/memory consumption of a process, is not considered. Whether a piece of system information is relevant to our analysis depends on the system rules (which will be discussed later), and the MAC system to be analyzed. Some facts are security-relevant under all protection mechanisms, like uid/gid of a process; while some facts are unique to a particular mechanism, like security contexts in SELinux and process profiles in AppArmor.

The encoding of Prolog facts for security policies vary for different security mechanisms. For example, in SELinux policies, there are several kinds of statements, e.g., Type

```
(1) file_info(path('/usr/bin/passwd'),
      type(regular), owner(0), group(0),
      uper(1,1,1), gper(1,0,1), oper(1,0,1),
      setuid(1), setgid(0), sticky(0),
      se_user('system_u'), se_role('object_r'),
      se_type('bin_t')).
(2) user_info('root', 0, 0).
(3) group_info('mail', 8, [dovecot]).
(4) process_running(4412, 0, 0,
      '/usr/lib/postfix/master',
      system_u, system_r, initrc_t).
(5) process_networking(4412).
```

(1) is the fact for file /usr/bin/passwd. The fact encodes the file name, type, owner, group, user/group/world permissions, setuid/setguid/sticky bit, and security context of the file. (2) is the fact for root user, which includes the user name, user id and group id. (3) is the fact for mail group, which includes the group name, group id and group members. (4) is the fact for the postfix master process. The fact contains the process id(pid), user id(uid), group id(gid), executed program, and the security context of the process. (5) is the fact for the same process as (4), denoting that the process is open to network.

Figure 4.2. Sample Facts of System State

```
(1) dom_priv('user_ssh_t', 'bin_t', 'file',
      ['ioctl', 'read', 'getattr', 'lock',
       'execute', 'execute_no_trans']).
(2) se_typetrans(old_dom('user_ssh_t'),
      new_dom('user_xauth_t'),
      type('xauth_exec_t')).
(3) se_domain('user_ssh_t').
(4) se_type('bin_t').
```

(1) says a process running under domain 'user_ssh_t' has the following permissions over a file with type 'bin_t': ioctl, read, getattr, etc. The fact is derived from a TE Access Vector Rule. (2) says if a process running under domain 'user_ssh_t' executes an executable file with type 'xauth_exec_t', the domain of the process should transition to domain 'user_xauth_t'. The fact is derived from a TE Type Transition Rule. (3) says 'user_ssh_t' is a SELinux domain. (4) says 'bin_t' is a SELinux type. Facts like (3) and (4) are used to enumerate SELinux domains and types.

Figure 4.3. Sample Facts of SELinux Policy

```
(1) aa_capability('/usr/lib/postfix/master',
      'net_bind_service').
(2) aa_access_mode('/usr/lib/postfix/master',
      '/etc/samba/smb.conf', r(1), w(0),
      ux(0), px(0), ix(0), m(0), l(0)).
```

(1) says the program /usr/lib/postfix/master has the capability of net_bind_service. (2) says the program can read samba configure file /etc/samba/smb.conf. Facts like (2) define the privileges of a program over a certain file or file pattern.

Figure 4.4. Sample Facts of AppArmor Policy

Enforcement Access Vector Rules and Type Enforcement Transition Rules. We also define all the domains and types. Figure 4.3 gives several sample Prolog facts which are generated based on a SELinux policy. Our parser for SELinux policy is based on the tool *checkpolicy*.

In AppArmor, a profile defines the privileges of a certain program. A privilege can be a capability, or a set of permissions over a file or file pattern. Figure 4.4 gives some sample Prolog facts of an AppArmor policy. Our parser for AppArmor policy is based on *apparmor_parser*.

4.2.2 Host Attack Graph Generator

Host Attack Graph Generator takes system facts, a library of system rules and the attack scenario as input, and generates the host attack graph. We first discuss how to define attack states.

In our analysis, the basic unit is a process. The attack state of a process consists of process attributes that are related to access control enforcement. Uid and gid of a process are used in Linux DAC mechanism, which is the default mechanism. MAC systems give additional process attributes. In SELinux, the current domain of a process is a security related attribute. Hence the attack state of a process is described as $\mathsf{proc}(\mathsf{uid}, \mathsf{gid}, \mathsf{domain})$. In AppArmor, an attack state is represented as $\mathsf{proc}(\mathsf{uid}, \mathsf{gid}, \mathsf{profile})$ where $\mathsf{profile}$ is the profile that confines the process.

Given the attack state of a process controlled by the attacker, the privileges available to the attacker is defined by the policy. For example, under SELinux, a process with a certain domain can only have a certain set of permissions. Permissions also depend on the uid and gid. Following are some relevant predicates to describe such enforcement:

- **dac_can_execute(Uid, Gid, Program)** : Decide if a process with certain uid and gid can execute a program.
- **dac_execve(Uid, Gid, NewUid, NewGid, Program)** : Decide the new uid and gid of a process after executing a program.
- **se_can_execute_prog(Domain, Program, NewDomain)** : Decide if a process with certain domain can execute a program, and what the new domain is after execution.
- **aa_file_privilege(Profile, File, Mode)** : Decide if a process with a certain profile can access a file with a certain mode, e.g., read, write, execute.
- **aa_new_profile(Profile, Program, NewProfile)** : Get the new profile of a process after executing a program. A profile can be 'none' meaning there is no profile confining the process.

Suppose the attacker controls a process p, she may exploit or launch a program prog to further control another attack state. We are interested in all the potential attack states that might be controlled by an attacker.

In SELinux, we represent the fact that the attacker can control a certain attack state as $\mathsf{se\_node}(\mathsf{proc}(\mathsf{uid}, \mathsf{gid}, \mathsf{domain}))$. If the attacker controls attack state s_1, and after exploiting a program prog she can control attack state s_2, the transition is represented as $\mathsf{se\_edge}(s_1, s_2, \mathsf{prog})$. Here $\mathsf{se\_node}(\cdot)$ and $\mathsf{se\_edge}(\cdot, \cdot, \cdot)$ are both dynamic predicates in Prolog. The state transition depends on the current attack state, the compromised program and the policy.

As one example of system rules, we now discuss how to encode domain transition under SELinux. The logic to decide domain transition is described in [7], and is non-

trivial. Suppose the current domain is OldDom, the type of the executable is Type and the new domain is NewDom. We summarize the logic as follows:

1. If OldDom doesn't have file execute permission on Type, the access is denied.

2. If there is a type transition rule: 'type_transition OldDom Type: process NewDom', the access is granted only when OldDom has process transition permission on Type and NewDom has file entrypoint permission on Type. Otherwise the access is denied. If the access is granted, the process runs on the domain NewDom after executing the program.

3. If there isn't such a type transition rule, the access is granted only when OldDom has file execute_no_trans permission on Type. Otherwise the access is denied. If the access is granted, the process runs on the original domain OldDom after executing the program.

Using logic programming the domain transition logic can be encoded naturally. Related Prolog code is shown in Figure 4.5.

The initial resources of the attacker can be represented as a set of initial attack states. Suppose the attacker can connect to the machine from the network, the initial attack states are encoded in Figure 4.6(a). Similarly, we use a set of goal attack states to represent the objective of the attacker. The encoding of the objective to load a kernel module is depicted in Figure 4.6(b).

Given the initial attack states and the goal attack states, we can generate the host attack graph that contains all the potential states that the attacker can control. The pseudo code is depicted in Figure 4.7.

4.2.3 Attack Path Analyzer

Attack Path Analyzer finds all the minimal attack paths in a host attack graph. Figure 4.8 describes the iterative algorithm used by Attack Path Analyzer. The algorithm repeatedly updates a set of paths for each node until all the sets are stablized.

```
se_can_execute_type(Domain, Type, NewDomain) :-
  se_typetrans(old_dom(Domain),
         new_dom(NewDomain), type(Type)),
  !,
  se_domain_privilege(domain(Domain),
         type(Type), class(file), op(execute)),
  se_domain_privilege(domain(Domain),
         type(NewDomain), class(process),
         op(transition)),
  se_domain_privilege(domain(NewDomain),
         type(Type), class(file), op(entrypoint)).

se_can_execute_type(Domain, Type, NewDomain) :-
  se_domain_privilege(domain(Domain),
         type(Type), class(file), op(execute)),
  se_domain_privilege(domain(Domain), type(Type),
         class(file), op(execute_no_trans)),
  NewDomain = Domain.
```

Figure 4.5. Rules for Domain Transition

```
net_init(proc(Uid,Gid,Domain), [Program]) :-
  process_networking(Pid),
  process_running(Pid, Uid, Gid, Program,
                           _, _, Domain).
```

(a) Initial resources: the attacker can connect to the machine from network

```
load_module_goal(proc(0, _Gid, Domain)) :-
  se_domain_privilege(domain(Domain), _,
        class(capability), op(sys_module)).
```

(b) Attack objective: to load a kernel module

Figure 4.6. Predicates for Initial Attack States and Goal Attack States

```
1: function GENERATE_GRAPH_NODE(s)
2:     if s is already a graph node then
3:         return
4:     Add s as a graph node
5:     if s is a goal attack state then
6:         return
7:     for all program prog that s can execute do
8:         s'      the attack state after executing prog
9:         Add (s, s') as a graph edge with label prog
10:        Generate_Graph_Node(s')
1: function GENERATE_HOST_ATTACK_GRAPH
2:     for all Initial attack state s do
3:         Generate_Graph_Node(s)
```

Figure 4.7. Algorithm for Host Attack Graph Generation

4.3 Evaluation

We use three attack scenarios to evaluate our approach. The first is for a remote attacker to install a rootkit. We assume the rootkit is installed by loading a kernel module. The second is for a remote attacker to plant a Trojan horse. We use two definitions of trojan attacks: (1) the attacker can create an executable in a folder on the executable search path or user's home directory (2) the attacker can create an executable in any folder such that a normal user process (with a user's uid and runs under unconfined domain in SELinux or is not confined by any profile in AppArmor) can execute. In both cases, after the trojan program is executed the process should be unconfined. We call (1) a strong trojan case and (2) a weak trojan case. The third is for a local attacker to install a rootkit.

We analyze the QoP under several configurations:

1. Ubuntu 8.04 (we use the Server Edition for all the test cases) with SELinux and Ubuntu 8.04 with AppArmor. To understand what additional protection MAC offers on top of DAC, we also evaluate Ubuntu 8.04 with DAC protection only (without MAC protection).

```
1:  function GENERATE_MINIMAL_ATTACK_PATHS
2:      V      V ∪ v_g
3:      for all goal attack state node v do
4:          add an edge from v to v_g,
5:          the exploited program for the edge is empty
6:      for all v ∈ V do
7:          MP(v)      φ
8:      for all initial attack state node v do
9:          MP(v)      {φ}
10:     repeat
11:         stable      true
12:         for all e ∈ E do
13:             for all p ∈ MP(e.v_1) do
14:                 p'      append(p, e)
15:                 if ∃p_0 ∈ MP(e.v_2) s.t. V(p') ⊂ V(p_0) then
16:                     Remove all such paths from MP(e.v_2)
17:                 if not ∃p_1 ∈ MP(e.v_2) s.t. V(p') ⊃ V(p_1) then
18:                     MP(e.v_2)      MP(e.v_2) ∪ {p'}
19:                     stable      false
20:     until stable
21:     return MP(v_g)
```

Symbols	Meaning
V	The set of host attack graph nodes
E	The set of host attack graph edges
v_g	The virtual "goal" node added such that each goal attack state has an edge to v_g
MP	MP(v) stores the set of minimal attack paths to node v
$e.v_1, e.v_2$	The starting node and ending node of an edge e
$V(p)$	The set of all exploited programs along the path p
append(p, e)	Append edge e to the end of path p

Figure 4.8. Minimal Attack Paths Generation

2. Fedora 8 with SELinux and SUSE Linux Enterprise Server 10 with AppArmor. We compare the results with Ubuntu 8.04/SELinux and Ubuntu 8.04/AppArmor to show that different distributions with the same mechanism provide different levels of protection.

3. Ubuntu 8.04 with SELinux. In the evaluation, we only analyze the SELinux policy. We use the result to show that only considering MAC policy without DAC policy and system state is not sufficient.

The active services include: sshd, vsftp, apache2, samba, mysql-server, postfix, nfsd, named, etc. In Fedora 8, the SELinux policy is the targeted policy that shipped with the distribution. In Ubuntu 8.04, the SELinux policy is the reference policy that comes with the selinux package. The AppArmor policy is the one that comes with the apparmor-profiles package.

4.3.1 SELinux vs. AppArmor vs. DAC only on Ubuntu 8.04

Ubuntu 8.04 Server Edition supports both SELinux and AppArmor. This offers an opportunity for us to compare the QoP of SELinux and AppArmor head to head. We also include the case in which only DAC is used in the comparison.

A Remote Attacker to Install a Rootkit In this attack scenario, the attacker has network access to the host, and the objective is to install a rootkit via loading a kernel module. The host attack graphs for DAC only, AppArmor and SELinux are shown in Figure 4.9, Figure 4.10 and Figure 4.11, respectively. The comparison of minimal attack paths between SELinux and AppArmor is shown in Table 4.1.

Among the three cases, AppArmor has the smallest vulnerability surface. SELinux has all the minimal attack paths AppArmor has and some additional ones. The DAC only case has all the attack paths SELinux has, and has one additional minimal attack path. More specifically, AppArmor has 3 length-1 minimal attack paths and 34 length-2 minimal attack paths. In addition to these, SELinux has 3 more length-1 minimal attack paths and 63 more length-2 minimal attack paths.

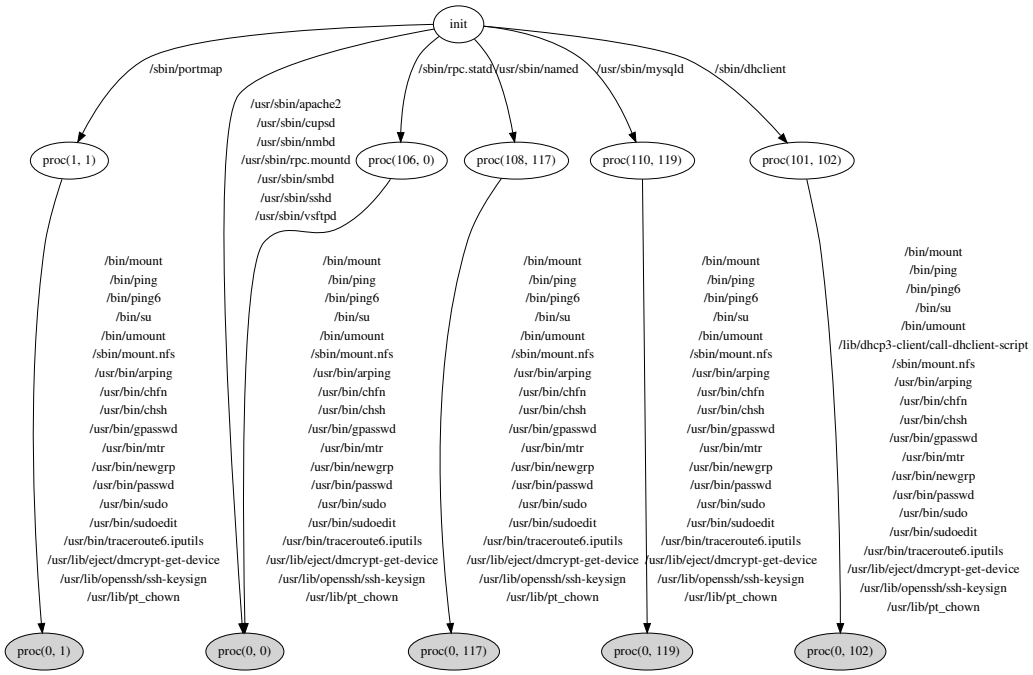


Figure 4.9. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with DAC only)

Attack paths common to all three cases are mainly due to daemon programs that run in unconfined domain under SELinux (meaning that the program is not constrained by SELinux) and are not confined by profiles under AppArmor. The length-1 paths are due to the daemon programs apache2, rpc.mountd and sshd which run as root. (Although sshd is running in sshd_t under SELinux and confined by a profile in AppArmor, the domain and the profile both allow the process to load a kernel module directly or indirectly). The length-2 paths are due to unprivileged daemon programs mysqld and named. After compromising one of them, the attacker needs to do another local privilege escalation.

The minimal attack paths that SELinux has but AppArmor doesn't have are due to three reasons: (1) Some programs are running in the unconfined_t domain under this version of SELinux policy, while AppArmor has profiles for them; these include, e.g., nmbd, smbd, vsftpd, portmap, and rpc.statd. (2) Some programs are confined by SELinux domains, but

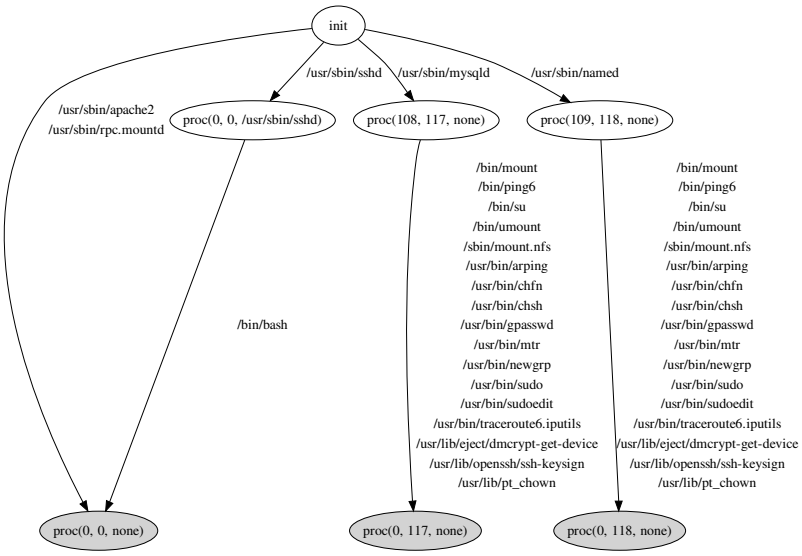


Figure 4.10. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with AppArmor)

the confinements are not as tight as corresponding AppArmor profiles. Two programs, cupsd and dhclient, fall into this category. For example, domain dhcpc_t is allowed to load a kernel module while the profile /sbin/dhclient doesn't allow kernel module loading. (3) Some programs (named and mysqld) are not confined either in SELinux or AppArmor. However, because they run with unprivileged accounts (as opposed to the root) under DAC, compromising them do not enable the attacker to load a kernel module. There are unique attack paths for SELinux because of the confinement of some setuid root programs. Ping and passwd are unconfined in SELinux but confined in AppArmor, therefore they can be used to further escalate the attackers' privileges after compromising named or mysqld.

Somewhat surprisingly, the DAC only case has only one additional (strong) length-1 minimal attack path compared to SELinux. The path is /usr/sbin/cupsd. The cupsd daemon runs as root and is confined by the cups_t domain of SELinux. When the attacker exploits cupsd with SELinux enabled, she has to additionally exploit the setuid root program /bin/unix_chkpwd to gain the privilege to install a rootkit.

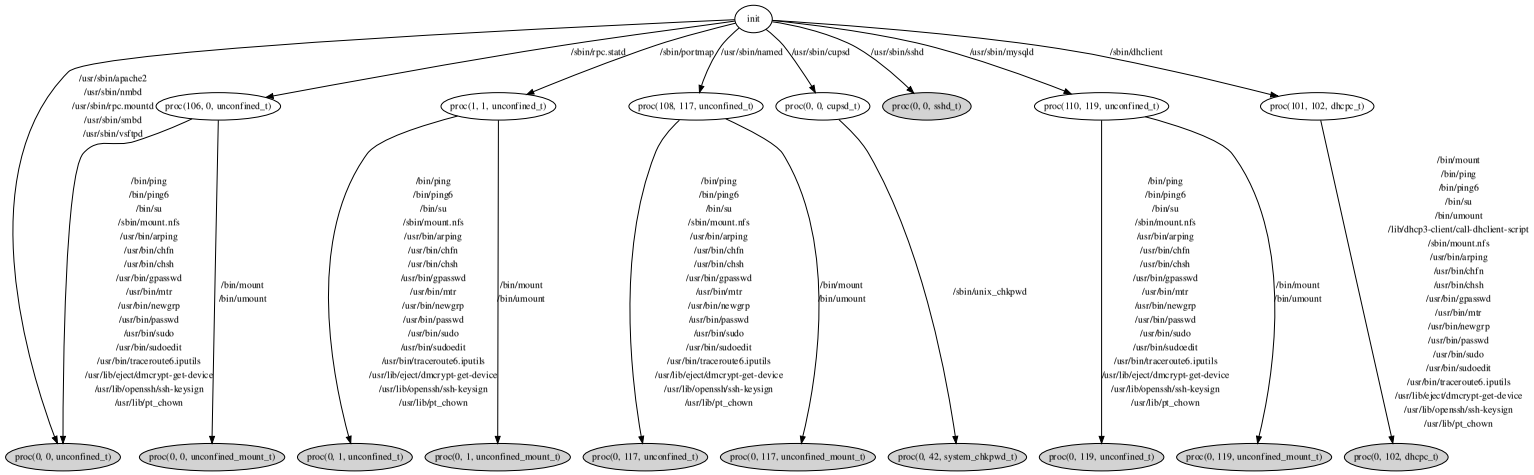


Figure 4.11. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux)

Table 4.1
Minimal Attack Paths Comparison for a Remote Attacker to Install a Rootkit

	SELinux compared to AppArmor
common	/usr/sbin/apache2 /usr/sbin/rpc.mountd /usr/sbin/named SUID* /usr/sbin/mysqld SUID* /usr/sbin/sshd
unique	/usr/sbin/nmbd /usr/sbin/smbd /usr/sbin/vsftpd /sbin/portmap SUID** /sbin/rpc.statd SUID** /usr/sbin/cupsd /sbin/unix_chkpwd /sbin/dhclient SUID** /sbin/dhclient /lib/dhcp3-client/call-dhclient-script /usr/sbin/named /bin/ping /usr/sbin/named /usr/bin/passwd /usr/sbin/mysqld /bin/ping /usr/sbin/mysqld /usr/bin/passwd

SUID* represents a set of setuid root programs:
/bin/ping6
/bin/su
/sbin/mount.nfs
/usr/bin/arping
/usr/bin/chfn
/usr/bin/chsh
/usr/bin/gpasswd
/usr/bin/mtr
/usr/bin/newgrp
/usr/bin/sudo
/usr/bin/sudoedit
/usr/bin/traceroute6.iputils
/usr/lib/eject/dmcrypt-get-device
/usr/lib/openssh/ssh-keysign
/usr/lib/pt_chown
/bin/mount
/bin/umount

SUID** includes all programs in SUID* and also /bin/ping and /usr/bin/passwd

Our analysis shows that among the seven network-facing programs running as root in Ubuntu 8.04 Server Edition, namely apache2, cupsd, nmbd, rpc.mountd, smbd, sshd, and vsftpd, only one of them is confined in any meaningful way by the SELinux policy. Hence one can argue that the additional protection provided by the SELinux reference policy in Ubuntu 8.04 is quite limited.

Remote Attacker to Leave a Trojan Horse

We consider a scenario in which the attacker is remote and wants to leave a Trojan horse. We consider both the strong Trojan horse case and the weak Trojan horse case. We observe that performing a strong trojan attack is always not more difficult than installing a kernel module.

For Ubuntu 8.04 with AppArmor, compared to loading kernel module, there is one extra attack path in strong trojan attack: /usr/sbin/smbd. For Ubuntu 8.04 with SELinux, the host attack graph is the same as the graph for a remote attacker to install a rootkit.

It's significantly easier to perform weak trojan attacks. Figure 4.12 shows the host attack graph to leave a weak trojan in Ubuntu 8.04 with SELinux. Every network faced program, if compromised, can be used directly to leave a weak Trojan horse. This is so due to two reasons. First, both SELinux and AppArmor confine only a subset of the known programs and leave any program not explicitly identified as confined. Second, as neither SELinux nor AppArmor performs information flow tracking, the system cannot tell a program left by a remote attacker from one originally in the system.

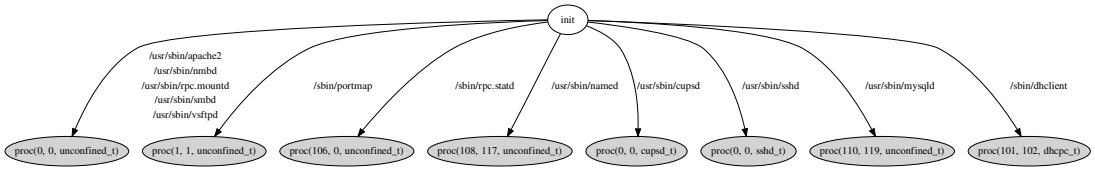


Figure 4.12. Host Attack Graph for a Remote Attacker to Leave a Weak Trojan (Ubuntu 8.04 with SELinux)

A Local Attacker to Install a Rootkit

In the third attack scenario, the attacker has a local account. The objective is to install a rootkit (load a kernel module). Figure 4.13 and Figure 4.14 shows the host attack graphs for Ubuntu 8.04 with SELinux and AppArmor, respectively.

Again, AppArmor has a smaller vulnerability surface. All minimal exploit paths in AppArmor also occur in SELinux, which has some additional exploit paths. There are 19 common minimal attack paths, they are all of length 1. They are due to 19 setuid root programs that have sufficient privileges. These programs are as follows:

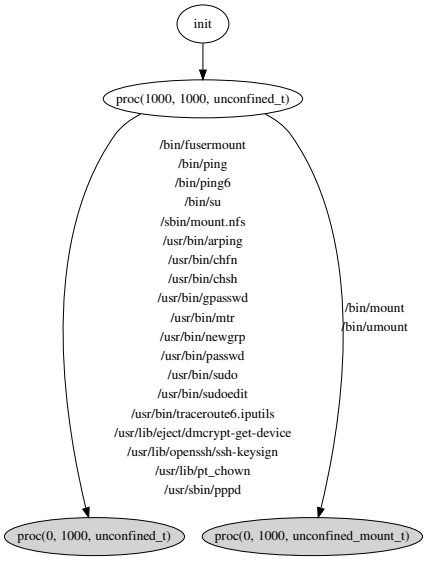


Figure 4.13. Host Attack Graph for a Local Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux)

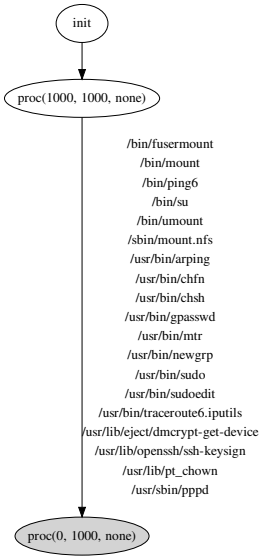


Figure 4.14. Host Attack Graph for a Local Attacker to Install a Rootkit (Ubuntu 8.04 with AppArmor)

/bin/fusermount, /bin/ping6, /bin/su, /sbin/mount.nfs, /usr/bin/arping
/usr/bin/chfn, /usr/bin/chsh, /usr/bin/gpasswd, /usr/bin/mtr, /usr/bin/newgrp
/usr/bin/sudo, /usr/bin/sudoedit, /usr/bin/traceroute6.iputils
/usr/lib/eject/dmcrypt-get-device, /usr/lib/openssh/ssh-keysign
/usr/lib/pt_chown, /usr/sbin/pppd, /bin/mount, /bin/umount

The programs in the common paths are setuid root programs. The result shows that the way for a local user to load a kernel module is to exploit one of the setuid root programs. SELinux has 2 unique minimal attack paths for SELinux: /bin/ping and /usr/bin/passwd. They are due to the same reason in the first scenario, that SELinux does not confine ping and passwd while AppArmor confines them.

4.3.2 Other Comparisons

In this subsection we compare the QoP offered by different Linux distributions with a same MAC mechanism. We also discuss why considering MAC policy alone is not enough.

Different Versions of SELinux

We have found that the SELinux policy in Fedora 8, which is the SELinux targeted policy, offers significantly better protection than the SELinux in Ubuntu 8.04 Server Edition, which uses a version of the SELinux reference policy. In addition, the most current version of the SELinux reference policy is also tighter than the policy shipped with Ubuntu 8.04.

Figure 4.15 shows the host attack graph for a remote attacker to install a rootkit in Fedora 8 with SELinux. The vulnerability surface is not directly comparable with that of Ubuntu 8.04 (shown in Figure 4.11) because each has some unique attack paths. If we assume that all programs are equal, the vulnerability surface of Fedora 8/SELinux is smaller because there is 1 length-1 minimal attack path and 13 length-2 minimal attack paths in Fedora 8/SELinux, while there are 6 length-1 minimal attack paths and 97 length-2 minimal attack paths in Ubuntu 8.04/SELinux.

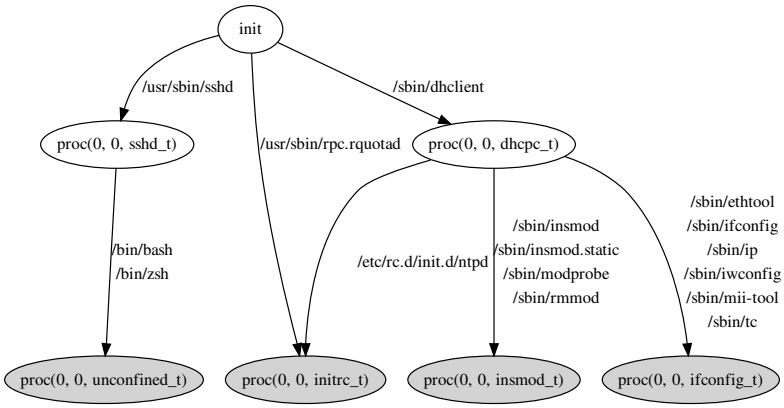


Figure 4.15. Host Attack Graph for a Remote Attacker to Install a Rootkit (Fedora 8 with SELinux)

Figure 4.16 shows the host attack graph for a remote attacker to leave a strong trojan in Fedora 8 with SELinux. Compared to the kernel module loading scenario, trojan attack scenario has three additional minimal attack paths:

/usr/sbin/rpc.mountd
/usr/sbin/smbd
/usr/sbin/sendmail /usr/bin/procmail

Two paths are related to file sharing and the other is due to sendmail. Those programs are confined, but they have privileges to write to the user's home directory or directories in the executable search path. Under the assumption that all programs are equal, the vulnerability surface of Fedora 8/SELinux is smaller than that of Ubuntu 8.04/SELinux for the remote trojan attack scenario.

Different Versions of AppArmor

We have analyzed the vulnerability surface of SUSE Linux Enterprise Server 10, or SLES 10, with AppArmor protection. To keep the services in SLES 10 the same as in Ubuntu 8.04, some services that are up by default in SLES 10 are turned off, e.g., slpd and zmd.

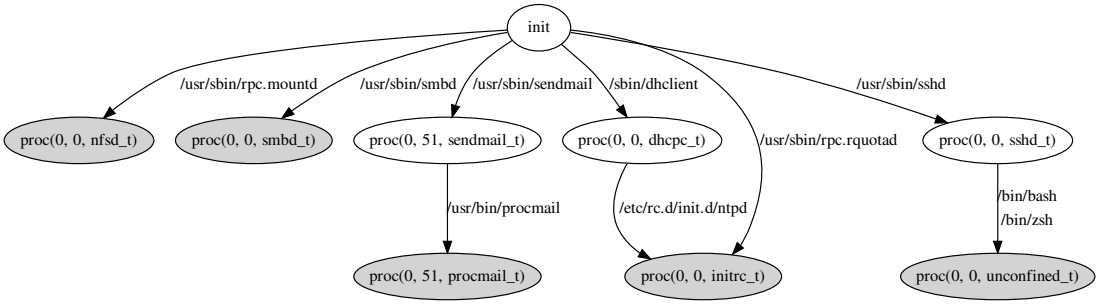


Figure 4.16. Host Attack Graph for a Remote Attacker to Leave a Strong Trojan (Fedora 8 with SELinux)

The vulnerability surface of SLES 10/AppArmor under the scenario that a remote attacker wants to install a rootkit (as shown in Figure 4.17) is not directly comparable with that of Ubuntu 8.04/AppArmor. The two distributions expose different vulnerability surfaces.

The common attack paths are through sshd and rpc.mountd (NFS mount daemon). The unique paths for Ubuntu 8.04 are through apache2, mysqld and named, due to that those programs are not confined. The unique paths for SLES 10 are through cupsd since cupsd is not confined. Sshd also contributes to some unique paths since there are more shells installed in SLES 10.

In SLES 10, the host attack graph for a remote attacker to plant a strong Trojan horse is the same as the graph for a remote attacker to install a rootkit. For a local attacker to install a rootkit, the host attack graph for SLES 10 is shown in Figure 4.18. There are 10 common attack paths due to unconfined set uid root programs. There are 9 unique attack paths for Ubuntu 8.04 and 20 unique attack paths for SLES 10.

The Need to Consider DAC Policy

Our approach considers both the MAC policy and the DAC policy. If we only consider MAC policy, e.g., SELinux policy, the result may not be accurate. Figure 4.19 shows the host attack graph for a remote attacker to install a rootkit, when we only consider

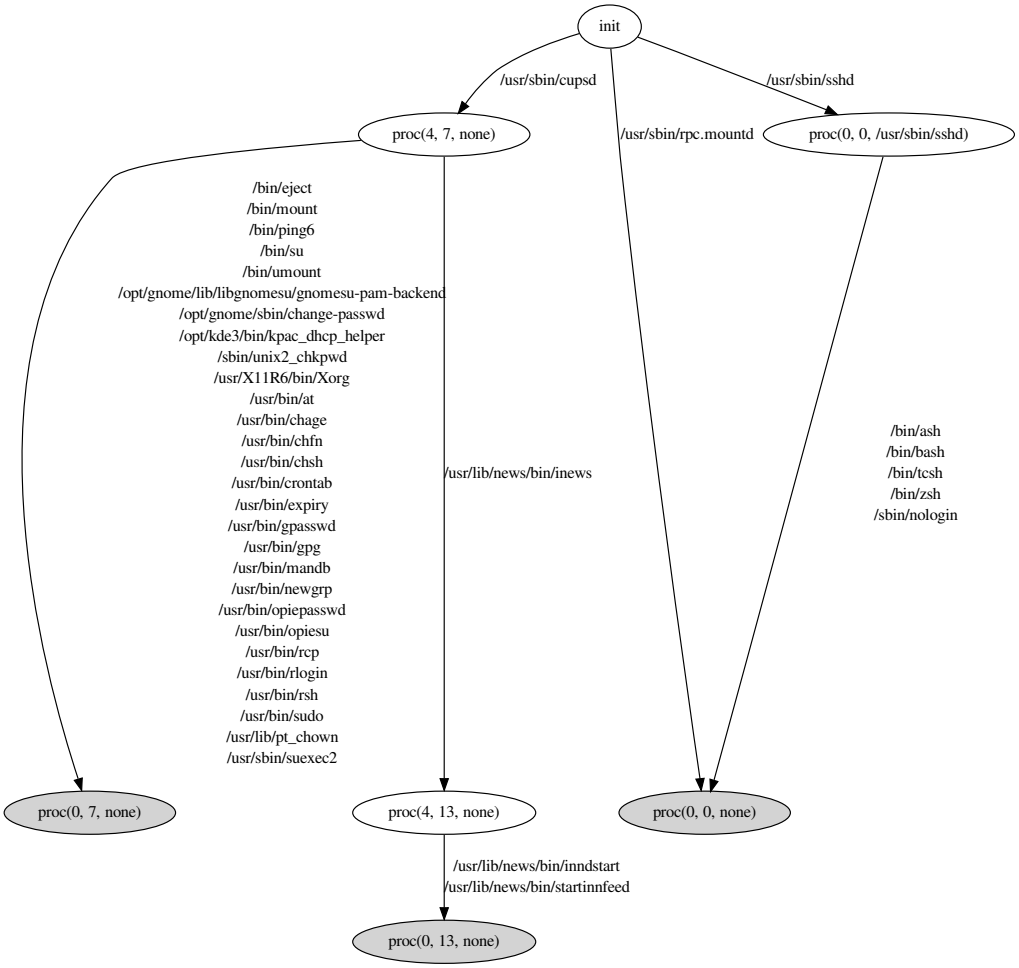


Figure 4.17. Host Attack Graph for a Remote Attacker to Install a Rootkit (SUSE Linux Enterprise Server 10 with AppArmor)

SELinux policy but not DAC policy. Compared to the host attack graph that considers both DAC and MAC policy (shown in Figure 4.11), we observe that without considering DAC policies, there are following extra length-1 attack paths: /sbin/portmap, /sbin/rpc.statd, /usr/sbin/mysqld, /usr/sbin/named, /sbin/dhclient. They are not accurate. For example, mysqld is running with uid 110 and unconfined_t. By compromising mysqld the attacker can control unconfined_t, but she still cannot load a kernel module because the uid is un-

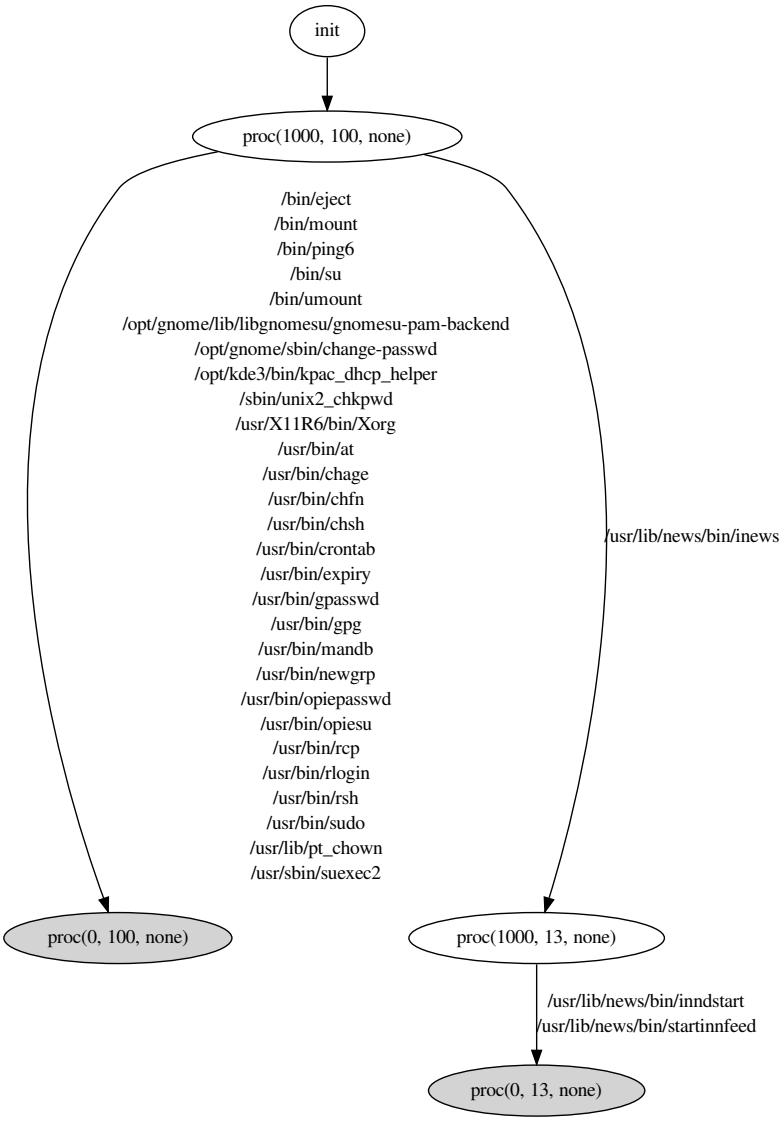


Figure 4.18. Host Attack Graph for a Local Attacker to Install a Rootkit (SUSE Linux Enterprise Server 10 with AppArmor)

privileged. To control root uid the attacker needs to do another exploit, e.g., by exploiting a setuid root program.

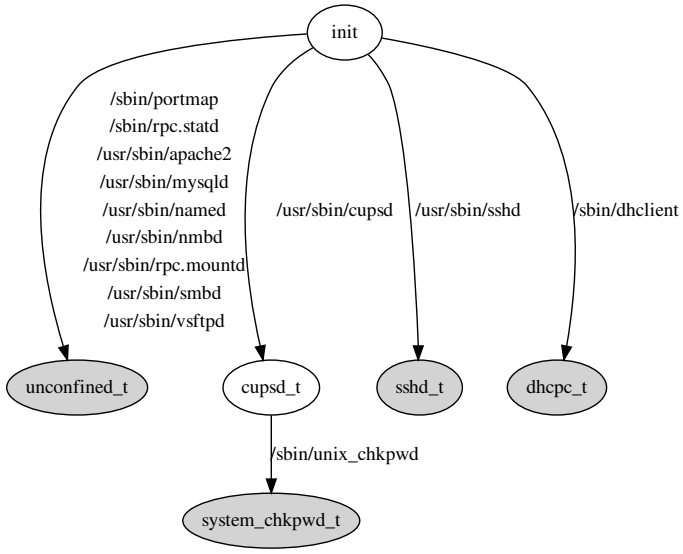


Figure 4.19. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux – only Considering SELinux Policy)

4.3.3 Performance

In our experiments, the targeted operating systems (Ubuntu, Fedora and SUSE Linux) are installed in virtual machines using VMWare. The host attack graph generation and attack path analysis are performed on a laptop with Intel(R) Pentium(R) M processor 1.80GHz and 1G memory. The Prolog engine is swi-prolog 5.6.14.

The running time for the fact collector is less than 10 minutes for every test case. The running time for the host attack graph generation and analysis is less than 10 minutes for every test case.

4.4 Summary

In this chapter, we introduce the notion of vulnerability surfaces under attack scenarios as the measurement of the QoP offered by Mandatory Access Control systems for Linux. A vulnerability surface consists the set of minimal exploit paths that are necessary and

sufficient for an attack scenario to be realized. The surface depends on the MAC mechanism, the specific MAC policy, the DAC policy settings, and the system configuration information. We have implemented a tool called VulSAN for computing such vulnerability surfaces for Linux systems with SELinux or AppArmor. VulSAN generates the host attack graph for a given attack scenario and compute the vulnerability surface from an attack graph. VulSAN can be used to compare the QoP of different Linux distributions. It can also be used as a system hardening tool, enabling the system administrator to analyze the vulnerability surface of the system and tweaking the policy to reduce it. Because VulSAN can handle both SELinux and AppArmor, which are the two MAC systems used by major Linux distributions, it can be used for most enterprise Linux distributions and home user distributions. We have evaluated VulSAN by analyzing and comparing SELinux and AppArmor in several recent Linux distributions, and showed that there are opportunities to tighten the policy settings in several popular Linux distributions.

5 ACCESS CONTROL POLICY ANALYSIS UNDER WINDOWS

Among different operating systems, the study of Windows systems is critical given its popularity and complexity. Windows access control is an example of complicated access control systems. For example, in Microsoft Windows there are about 15 types of securable objects [13], and each type is associated with up to 32 access mask bits. Virtually every object in the system, such as files, registry keys, services, processes, pipes, etc, is guarded by a security descriptor which contains an access control list and other security information. The complexity often makes the mechanisms difficult to understand by normal users, system administrators and software developers. Given the scale of possible configurations and the difficulty to thoroughly understand the mechanisms, host systems can easily be mis-configured [18].

Since access control is a major mechanism to provide user isolation and to protect operating systems against local privilege escalation attacks and remote attacks, access control mis-configurations could have serious security consequences. For example, if the right to write a critical system file is accidentally given to unprivileged users, a normal user can easily gain the privileges of a system administrator. Even if configurations are correct, an attacker might exploit software vulnerabilities to compromise the system. There are always legitimate reasons for a privileged process to interact with untrusted and potentially malicious data. For example, a privileged server program might receive network traffic. It can be compromised if, say, the input validation is not done properly.

There are a number of existing research works that study Windows access control [5, 18, 26–32]. Most of existing approaches for Windows are based on known attack patterns. With these approaches, attack scenarios are defined manually, and corresponding attacks are captured automatically. Given the complexity and the amount of possible attacks, it is difficult to manually enumerate all attack scenarios. Another issue of existing solutions is

that they usually do not consider software vulnerabilities therefore some potential attacks are ignored.

In this chapter, we present a tool called Windows Access Control Configuration Analyzer (WACCA) to automatically discover attack patterns in a Windows host. Here an attack pattern represents a set of similar attacks. An attack consists of one or several actions by the attacker. WACCA can be used to find various privilege escalation attacks through the access control subsystem in Windows. Such a tool is useful for software vendors that the configurations of software packages can be tested thoroughly. It also benefits system administrators that the potential vulnerabilities of the systems they manage can be better understood. Our contributions are as follows:

1. We propose a model to analyze Windows access control configurations. The model is centered on the abilities of the attackers over system objects. An attacker can take actions to gain additional abilities from the abilities he already has. Given the attacker's initial abilities and attack goals, an attack graph is used to represent all possible attacks by the attacker.

2. We implement WACCA based on the model. WACCA consists of a scanner to scan the configurations, an attack graph generator to generate the attack graph, and a pattern analyzer to automatically discover attack patterns and their instances. Logic programming is used to describe the interaction rules within the system, which makes the rules highly declarative and intuitive.

3. We study the configurations of a Windows Vista host. A remote attack case and a local attack case are analyzed. WACCA found 8 attack patterns totally. The attack subgraph and instances of each pattern were also generated.

5.1 Design of WACCA

We aim to develop a tool for reasoning privilege escalation attacks through the access control subsystem in Windows.

WACCA can find such privilege-escalation attacks for a Windows host. It takes inputs that include two parts: (1) system facts, and (2) specified interaction rules. The system facts include access control configurations, such as access tokens of processes, security descriptors of files, etc. The interaction rules define how an attacker can interact with the system, e.g., by writing files, launching processes, etc. The tool answers a fundamental question of all privilege escalation attacks: what can an unprivileged user do? For example, can the user acquire a privileged process which has an access token of an administrator? An attack graph is generated to answer the question.

Due to the size of a typical such attack graph, simply presenting the graph to an administrator may be overwhelming. WACCA addresses this by extracting attack patterns from the graph. An attack pattern essentially represents a particular type of attack paths in the attack graph.

5.1.1 Attacker's Abilities

Our model is centered on analyzing the attacker's abilities over objects in the system. There are different types of objects, such as processes, files, services, etc. The privileges an attacker acquires in the system are expressed by his *abilities* over objects. We consider the following abilities

- control_process_code(AccessToken). The attacker controls the code that a process runs with AccessToken being its access token. This may happen because the attacker has a local account with AccessToken, or because the attacker exploits a vulnerability of a process and hijacks the control flow of the process.

- control_network_input(PortNum). The attacker can access the host machine remotely through the port number PortNum. This happens when the machine's port number is open to the network.

- control_file_data(FileName). The attacker controls the data of the file with name FileName.

- control_file_dacl(FileName). The attacker controls the discretionary access control list of the file with the name FileName.

- control_service_config(ServiceName). The attacker controls the configuration parameters of the service with the name ServiceName.

- control_service_dacl(ServiceName). The attacker controls the discretionary access control list of the service with the name ServiceName.

To denote an object, we sometimes use the object's attributes that are relevant to access control, rather than a unique identifier. For example, if an attacker controls the code of a process, what matters to the attacker is the access token of the process.

5.1.2 Actions and Deriving Rules

To achieve privilege escalation, the attacker carries out various *actions* to gain new abilities. An attacker may perform a wide range of actions. For example, he may write a critical file to control the data of the file or he may compromise a program to control a process. Some actions are simply Win32 API function calls, e.g., ChangeServiceConfig. Some actions require other users' involvement, e.g., Trojan attacks in which the attacker replaces a binary file with a malicious program and waits for a user to launch a Trojan program. We consider two types of such Trojan attacks. The first type is represented by the action WaitExecute, which means that in the current snapshot, some process is already executing the binary file. Then we know it is very likely that in the future a process with the same access token will execute this binary file. The other type is represented by the action HopeExecute, which means that the attacker hopes that some user might execute the binary. The latter type has a lower likelihood of succeeding. Some actions require compromising some programs. The action CompromiseRead represents the action to compromise a buggy program by maliciously crafting the content of a file that is read by the program. The action CompromiseNetwork represents the action of sending maliciously crafted data to a buggy program through network.

When carrying out an action, there are pre-conditions that must be satisfied. After successfully carrying out an action, the attacker acquires new abilities. We call these *derived* abilities, and use a *deriving rule* to define the pre-conditions, the action, and the derived ability. For example, the deriving rule of file writing is as follows:

$$
\begin{aligned}
&\mathsf{control\_process\_code}(\mathsf{access\_token})\\
&\mathsf{file\_security\_descriptor}(\mathsf{filename}, \mathsf{security\_descriptor})\\
&\mathsf{access\_token\_has\_right}(\mathsf{access\_token}, \mathsf{right}, \mathsf{security\_descriptor})\\
&(\text{here } \mathsf{right} \in \{\mathsf{FILE\_APPEND\_DATA}, \mathsf{FILE\_WRITE\_DATA},\\
&\qquad \mathsf{GENERIC\_WRITE}, \mathsf{GENERIC\_ALL}\})\\
&\Longrightarrow \mathsf{WriteFile}(\mathsf{filename}) \Longrightarrow\!\leftarrow\\
&\mathsf{control\_file\_data}(\mathsf{filename})
\end{aligned}
$$

There are 3 pre-conditions in the rule: (1) The attacker controls a process with a certain access token (2) The file has a certain security descriptor (3) The security descriptor allows the access token to write to the file.

Table 5.1 shows different deriving rules and actions considered in our model. Note that the difficulties to carry out different actions vary. The action ChangeServiceConfig can be easily carried out once the pre-conditions are satisfied: the attacker just needs to call the corresponding API function. On the other hand, to exploit a bug of a network server program might require a lot more efforts. It is discussed in Section 5.2.3 how to differentiate between actions.

5.1.3 Initial Abilities and Goals

The attacker has some *initial abilities*. In our case studies, we consider two scenarios, a remote attacker and a local attacker. The remote attacker can access the host remotely through the network. Let PortNum be any valid port number of the host, the attacker's initial ability is $\mathsf{control\_network\_input}(\mathsf{PortNum})$.

Table 5.1
Deriving Rules

Ability	Pre-conditions	Action
control_service_config(·)	control_process_code(access_token) SCM_security_descriptor(sd) access_token_has_right(access_token, right, sd) right ∈ {SC_MANAGER_CREATE_SERVICE, GENERIC_WRITE, GENERIC_ALL}	CreateService(service)
control_service_config(·)	control_process_code(access_token) service_security_descriptor(service, sd) access_token_has_right(access_token, right, sd) right ∈ { SERVICE_CHANGE_CONFIG, GENERIC_WRITE, GENERIC_ALL }	ChangeServiceConfig(service)
control_service_config(·)	control_service_dacl(service)	ChangeServiceConfig(service)
control_service_dacl(·)	control_process_code(access_token) service_security_descriptor(service, sd) access_token_has_right(access_token, right, sd) right ∈ { WRITE_DAC, WRITE_OWNER, GENERIC_ALL }	SetServiceObjectSecurity(service)
control_file_dacl(filename)	control_process_code(access_token) file_security_descriptor(filename, sd) access_token_has_right(access_token, right, sd) right ∈ { WRITE_DAC, WRITE_OWNER, GENERIC_ALL }	SetNamedSecurityInfo(filename)
control_file_data(filename)	control_process_code(access_token) file_security_descriptor(filename, sd) access_token_has_right(access_token, right, sd) right ∈ { FILE_APPEND_DATA, FILE_WRITE_DATA, GENERIC_WRITE, GENERIC_ALL }	WriteFile(filename)
control_file_data(filename)	control_file_dacl(filename)	WriteFile(filename)
control_process_code(∗)	control_service_config(service)	WaitStartService(service)
control_process_code(access_token)	control_file_data(filename) process_running(pid, filename) process_token(pid, access_token)	WaitExecute(filename)
control_process_code(*)	control_file_data(filename) is_executable(filename)	HopeExecute(filename)
control_process_code(access_token)	control_file_data(filename) process_reading(pid, filename) process_running(pid, binary) process_token(pid, access_token)	CompromiseRead(binary, filename)
control_process_code(access_token)	control_network_input(port) receiving_data(pid, port) process_running(pid, binary) process_token(pid, access_token)	CompromiseNetwork(binary, port)

Two types of wildcards are used here. If a value is irrelevant to access control, it is replaced with the wildcard of $\cdot$. For example, if the attacker can control the configuration parameters of a service, the name of the service is not important. Therefore the ability is represented as control_service_config($\cdot$).

If the value can be manipulated by the attacker to any relevant value, it is replaced with the wildcard of $*$. For example, if the attacker can configure a service, she can thus manipulate the account that the service runs with. Therefore after the service is started the access token of the process is $*$.

The local attacker has an unprivileged local account TestNormalUser, and has the ability

$$\mathsf{control\_process\_code}(\mathsf{SecurityDescriptor}(\mathsf{sid}_{\mathsf{TestNormalUser}}, \ldots))$$

The attacker's *goals* can also be expressed by abilities. A typical goal is to acquire an administrator's SID. The corresponding ability is

$$\mathsf{control\_process\_code}(\mathsf{SecurityDescriptor}(\mathsf{sid}_{\mathsf{administrator}}, \ldots))$$

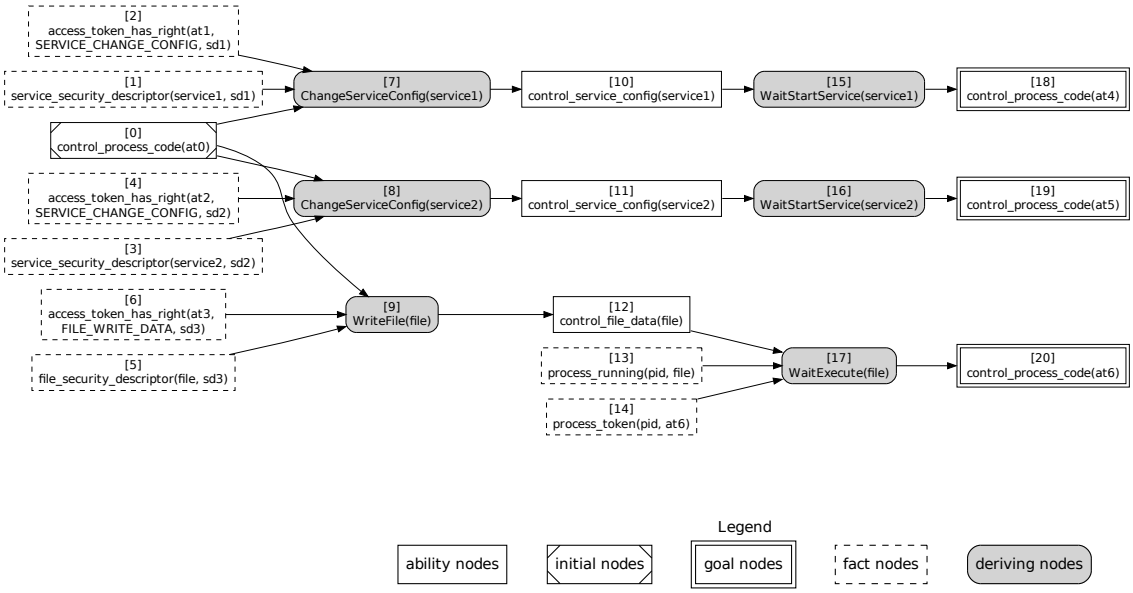
Intuitively, an attack is realized by carrying out actions (or, applying deriving rules) to augment the attacker's abilities from initial abilities to goals. We study a host configuration in the context of the attacker's initial abilities and goals.

5.1.4 Attack Graph and Attacks

An *attack graph* is used to demonstrate all possible attacks within the model. There are three types of nodes in an attack graph: ability nodes, fact nodes and deriving nodes. Each ability node represents an attacker's ability. Each fact node represents some system facts or information that is implied by system facts. Each deriving node corresponds to a deriving rule. We call ability nodes and fact nodes *graph nodes*. A deriving node is connected to several graph nodes which correspond to the pre-conditions. It is also connected to one ability node which corresponds to the derived ability. A deriving node is labeled by the action of the corresponding deriving rule.

Nodes representing initial abilities are called *initial nodes*. Nodes representing goals are called *goal nodes*. Goal nodes do not serve as pre-conditions for any deriving nodes.

Figure 5.1 is a simple example of attack graph. The attacker's initial ability is controlling an unprivileged process with access token at0. The goal is to control some processes with privileged access tokens. With at0 the attacker can change the configuration of two services, service1 and service2. By configuring the services, the attacker can wait for the services to be started, and control corresponding processes with access tokens at5 and at6. With at0 the attacker can also write to a binary file, which is executed by some process in the current snapshot. By writing to the file and controlling the data of the file, the attacker



In the graph, **at?** is an access token; **sd?** is a security descriptor; **service?** is a service name; **file** is a file name; **pid** is a process ID.

Figure 5.1. An Example of Attack Graph

can wait for the file to be executed and control the new process with access token at6. at4 − at6 are all privileged access tokens.

An *attack* can be expressed as a (minimal) subgraph of the attack graph such that (1) at least one goal node is in the subgraph (2) any ability node in the subgraph is either an initial node or can be derived from ability nodes and fact nodes in the subgraph. In Figure 5.1, three attacks can be identified. The first one is expressed by the subgraph with nodes $\{0, 1, 2, 7, 10, 15, 18\}$. The second one is expressed by the subgraph with nodes $\{0, 3, 4, 8, 11, 16, 19\}$. The third one is expressed by the subgraph with nodes $\{0, 5, 6, 9, 12, 13, 14, 17, 20\}$.

5.1.5 Attack Patterns

There could be many attacks in an attack graph. We categorize attacks into *attack patterns*. Every attack can be abstracted into an attack pattern, which represents all attacks that use the same actions. In general, an attack pattern is defined by the deriving rules involved and the structure of the deriving rules. Particularly, since all the driving rules currently considered in our model has exactly one ability in the pre-conditions, an attack pattern can be represented as a sequence of deriving rules. For example, in Figure 5.1, there are 2 attack patterns (to save space the fact nodes are elided):

$$
\begin{aligned}
P_1 =& \mathsf{control\_process\_code} \\
& \rightarrow (\mathsf{ChangeServiceConfig}) \rightarrow \mathsf{control\_service\_config} \\
& \rightarrow (\mathsf{WaitStartService}) \rightarrow \mathsf{control\_process\_code} \\
P_2 =& \mathsf{control\_process\_code} \rightarrow (\mathsf{WriteFile}) \rightarrow \mathsf{control\_file\_data} \\
& \rightarrow (\mathsf{WaitExecute}) \rightarrow \mathsf{control\_process\_code}
\end{aligned}
$$

An attack pattern represents the set of all the attacks that can be carried out by following the same deriving rules. Different attacks may use different parameters to instantiate the objects and actions in the deriving rules. Attacks of a same attack pattern are called *instances* of the attack pattern. There are two instances of P_1 and one instance of P_2 in Figure 5.1.

Attack patterns and their instances constitute an important part of the analysis results. Attack patterns give an overview of the (potential) vulnerabilities of a Windows system. All the attacks provide a comprehensive list of possible ways to compromise a system (as per the deriving rules in our model).

5.2 Implementation

WACCA consists of three components. *Fact Collector* collects system facts, *Attack Graph Generator* generates the attack graph, and *Pattern Analyzer* produces attack pat-

terns and their instances by analyzing the attack graph. Attack Graph Generator encodes deriving rules into Prolog rules and enlists the Prolog reference engine to conduct the graph generation, therefore the output of Fact Collector are in the form of Prolog facts.

The sets of attacker's abilities, attacker's actions and deriving rules are manually defined. Fact collection, attack graph generation and attack pattern analysis are performed automatically.

5.2.1 Fact Collector

The Fact Collector retrieves information that is related to our analysis. Any system fact that serves as a pre-condition of a deriving rule (or is needed to evaluate a pre-condition) in Table 5.1 needs to be collected. The facts are then encoded into Prolog facts. Table 5.2 gives an overview of the systems facts. Examples of system facts are given in Figure 5.2.

Table 5.2
System Facts Overview

Prolog Encoding	System Fact
scm_sd(SD)	The security descriptor of Service Control Manager is SD.
network(Pid, Protocol, Port)	The process with ID Pid is using a port Port with protocol Protocol.
is_executable(FileName)	The file FileName is an executable file.
service_sd(ServiceName, SD)	The security descriptor of the service with the name ServiceName is SD.
process_access_token(Pid, AT)	The process with ID Pid has the access token AT.
process_info(Pid, FileName)	The process with process ID Pid is running the executable with the name FileName.
system_handle(Pid, Handle, HType, ObjName)	The handle Handle is owned by the process with ID Pid, is of type HType and the name of the object is ObjName.
file_sd(FileName, SD)	The file with the name FileName has the security descriptor SD.
account(Sid, Account)	The user or group account Account corresponds to the SID Sid.

In the encoding, SD is a Prolog term that encodes a security descriptor and AT is a Prolog term that encodes an access token. An example of each is given in Figure 5.2. Pid, Port and Handle are numbers. Protocol $\in \{\mathsf{tcp}, \mathsf{udp}\}$. HType $\in \{\mathsf{file}, \mathsf{directory}, ...\}$. FileName, ObjName, ServiceName, Sid and Account are strings.

The Fact Collector is implemented using Python 2.6 with Windows extension [71]. Most facts are collected by calling relevant Win32 API functions. The files that processes

```
file_sd('c:\\Windows\\regedit.exe',
sd('S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464',
   'S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464',
[ace('allowed', 0,
     'S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464',
     ['DELETE', 'WRITE_OWNER', 'READ_CONTROL', 'WRITE_DAC',
                               'FILE_WRITE_EA', ...]), ...
 ace('allowed', 0, 'S-1-5-18', ['READ_CONTROL', 'FILE_READ_DATA',
                               'FILE_EXECUTE', ...]),
 ...],
[ace('audit', 192, 'S-1-1-0', [ 'DELETE', 'WRITE_OWNER', 'WRITE_DAC',
                                                    ... ])] )).
```

(a) The security descriptor of executable file regedit.exe

```
process_access_token(732, access_token(0, primary, ...,
                     'S-1-5-18', 'S-1-5-32-544',
[ ['S-1-5-32-544', enabled_by_default, enabled, owner],
  ['S-1-1-0', enabled_by_default, enabled, mandatory],
  ['S-1-5-11', enabled_by_default, enabled, mandatory], ... ],
...,
[ ['SeCreateTokenPrivilege', enabled],
  ['SeAssignPrimaryTokenPrivilege'],
  ['SeLockMemoryPrivilege', enabled_by_default, enabled], ...],
'S-1-5-18', 'S-1-16-16384', no_write_up)).
```

(b) The access token of the process which runs lsass.exe

Figure 5.2. Examples of System Facts

are currently reading are retrieved by examining all the handles in the current system snapshot. The handles are collected by the Windows Sysinternals tool *Handle* [72]. The Fact Collector consists of about 1700 lines of Python scripts and 250 lines of C++ code (including comments and blank lines).

5.2.2 Attack Graph Generator

The input of the Attack Graph Generator is a collection of system facts in the form of Prolog facts. The attacker's abilities are also encoded into Prolog facts. For example, the ability to control the data of a file is encoded into the Prolog fact

$$\mathsf{control\_file\_data}(\mathsf{file}(\mathsf{FileName}, \mathsf{SecurityDescriptor}))$$

Here FileName is the name of the file and SecurityDescriptor is the Prolog term to represent the security descriptor of the file, which is similar to the one shown in Figure 5.2(a).

The initial ability of the attacker is declared using a Prolog clause that does not have other abilities as pre-conditions. Suppose the attacker initially has an unprivileged local account named "TestNormalUser", the initial ability is declared as follows:

```
control_process_code(AccessToken) :-
    account(Sid, 'TestNormalUser'),
    sid_access_token(Sid, AccessToken).
```

The predicate account(Sid, AccountName) means that the SID Sid represents the account named AccountName. The predicate sid_access_token(Sid, AccessToken) gives an access token whose user SID is Sid.

The deriving rules can be naturally encoded using Prolog rules. Each such rule contains two parts: (1) an evaluation part that evaluates the pre-conditions (2) a logging part that logs the evaluation if the evaluation succeeds. For example, the deriving rule to compromise a program by manipulating its input file can be encoded as follows:

```
control_process_code(AccessToken) :-
  %% evaluation part
  process_reading(Pid,FileName),
  control_file_data(FileName),
  process_access_token(Pid,AccessToken),
  process_info(Pid,Binary),
  %% logging part
  get_node_id(control_process_code(AccessToken),Id0),
  get_node_id(control_file_data(FileName),Id1),
  get_node_id(process_reading(Pid,FileName),Id2),
  get_node_id(process_access_token(Pid,AccessToken), Id3),
  get_node_id(process_info(Pid,Binary),Id4),
  assert(proof_node(control_process_code_4,Id0,
                [Id1,Id2,Id3,Id4],
                [compromiseRead(Binary,FileName)])).
```

The predicate get_node_id(Term, Id) returns a unique identifier for Term. proof_node is a dynamic term. By asserting a proof_node, a deriving node is created and thus logged. The deriving node contains the information of pre-conditions, post-condition and the action. Therefore the whole attack graph can be reconstructed by logged deriving nodes. This technique is inspired by [48].

The following Prolog rule jump starts the attack graph generation:

```
search :- control_process_code(_AccessToken), fail.
```

To speed up the execution, we use tabling evaluation provided by Prolog implementation XSB [73]. All the predicates to represent the attacker's abilities are declared as tabled predicates. This also makes our Prolog program highly declarative. Note that without tabling, constructing deriving rules like the above may cause an infinite loop if a standard Prolog evaluation is used.

Attack Graph Generator consists of about 800 lines of Prolog code (including comments and blank lines). To achieve better loading and execution time, system facts are declared as dynamic predicates.

5.2.3 Pattern Analyzer

As discussed previously, the deriving rules we consider share a common property that there is exactly one ability in their pre-conditions. Our pattern analyzer is tailored according to this property. If we treat all the ability nodes in an attack graph as vertices, and deriving nodes as edges connecting the pre-condition node (ability node) and the post-condition node, the attack graph is reduced to a directed graph. An attack is reduced to a path in the graph. The pattern analyzer uses a depth-first search (DFS) to enumerate all paths from the initial ability nodes to goal nodes. Paths are categorized into different attack patterns on the fly. A pruning is used in the DFS: suppose a path contains nodes $v_1, \ldots, v_i, v_{i+1}, \ldots, v_j, \ldots$, if v_j can be derived directly from v_i, the path is pruned. This pruning avoids unnecessary actions of the attacker which make an attack not interesting.

To differentiate between different actions, one can assign a "cost" for each action. The cost of an attack pattern is the sum of the costs of all actions involved. Attack patterns can therefore be prioritized by their costs.

Another solution of attack generation is to calculate the shortest paths from initial nodes to goal nodes. This method generates a subset of attacks by the above method. We tried this approach originally and did not adopt it because the costs assigned to actions affect the attacks generated, which makes the cost assignment very tricky, e.g., some interesting paths might be ignored if they have slightly higher costs than that of the shortest paths.

5.3 Case Studies

We did two case studies on a machine with Windows Vista Ultimate installed. Some common software products are installed for evaluation, such as web browser, email client, word processor, multimedia player, photo editor, instant messenger, etc. The programs are all running when the fact collector collects system facts.

The first attack scenario is that a remote attacker who can access the targeted host through network wants to control a privileged process. The second attack scenario is that an attacker with an unprivileged local account wants to control a privileged process. The goal of both cases can be represented as the control of a process such that (1) the access token is "*", or (2) the access token contains the SID of account Administrators, or (3) the access token contains the SID of account System.

To prioritize attack patterns, the following costs are assigned to different actions:

createService	1	changeServiceConfig	1
setServiceObjectSecurity	1	setNamedSecurityInfo	1
writeFile	1	waitStartService	1
waitExecute	2	hopeExecute	5
compromiseRead	10	compromiseNetwork	10

Note that the costs only affect the order of the attack patterns. The costs assignments do not affect what attack patterns and attacks are generated.

The system facts of both cases were collected together. The fact collecting took about 10 minutes, with most of the time spent on file system scanning. For case 1, the attack graph generation took about 100 seconds, and the pattern analysis took about 15 seconds. For case 2, the attack graph generation took about 35 seconds, and the pattern analysis took about 5 seconds.

5.3.1 Case 1: Remote Attack

The initial ability is that the attacker can access the host remotely. The attack graph consists of 2469 graph nodes and 5223 deriving nodes. Figure 5.3 shows the 5 attack patterns in the attack graph. The costs and numbers of instances of attack patterns are shown in Table 5.3.

P_1 =control_network_input $\rightarrow$ (compromiseNetwork)
$\rightarrow$ control_process_code
P_2 =control_network_input $\rightarrow$ (compromiseNetwork)
$\rightarrow$ control_process_code $\rightarrow$ (writeFile) $\rightarrow$ control_file_data
$\rightarrow$ (waitExecute) $\rightarrow$ control_process_code
P_3 =control_network_input $\rightarrow$ (compromiseNetwork)
$\rightarrow$ control_process_code $\rightarrow$ (writeFile) $\rightarrow$ control_file_data
$\rightarrow$ (hopeExecute) $\rightarrow$ control_process_code
P_4 =control_network_input $\rightarrow$ (compromiseNetwork)
$\rightarrow$ control_process_code $\rightarrow$ (writeFile) $\rightarrow$ control_file_data
$\rightarrow$ (compromiseRead) $\rightarrow$ control_process_code
P_5 =control_network_input $\rightarrow$ (compromiseNetwork)
$\rightarrow$ control_process_code $\rightarrow$ (writeFile) $\rightarrow$ control_file_data
$\rightarrow$ (compromiseRead) $\rightarrow$ control_process_code
$\rightarrow$ (writeFile) $\rightarrow$ control_file_data $\rightarrow$ (compromiseRead) $\rightarrow$ control_process_code

Figure 5.3. Attack Patterns of Remote Attacks

Table 5.3
Costs and Numbers of Instances for Attack Patterns of Case 1 & 2

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
cost	10	13	16	21	32	3	6	11
number of instances	45	60	8562	211	6	4	573	14

P_1 represents attacks that compromise a privileged process which receives network traffic. Once the process is compromised, the goal is achieved. Such privileged processes receive traffic from about 40 ports in the target system. For example, several service programs run with the System account, and web browsers, instant messengers, etc, run with the user's account which belongs to the group Administrators.

P_2 represents attacks that compromise an unprivileged process, and then write to an executable file that is being executed by a privileged process at the collecting snapshot. Then the attacker can mount a Trojan attack. The unprivileged processes run with unprivileged user accounts, LocalService and NetworkService. LocalService and NetworkService were introduced from Windows XP and Windows Server 2003 to run less privileged services. Before they were introduced, services ran mostly with the System account, which means by compromising any service an attacker can take over the entire host. There are 3 executables that are running and can become Trojans. By examining the security descriptors of these executables, we found that they all grant the right of writing the executable files to AuthenticatedUsers, a SID almost every access token has.

P_3 is similar to P_2. The difference is that the Trojan attacks involve executables that are not running at the collecting snapshot. P_3 is more difficult to realize than P_2, since there is uncertainty in the executables' being executed by privileged accounts. There are 574 executables (including dlls) that can be written in this pattern. These executables are within 6 software packages from 5 vendors.

Attacks of P_4 comprise two steps: (1) similar to P_1 to P_3, to compromise an unprivileged process that receives network traffic, (2) write to a file that a process is reading and exploit a bug to compromise the process. As an example, Figure 5.4 shows the subgraph

of the attack graph that is related to P_4. Figure 5.5 shows the attack pattern in details. For better visualization, we only show ability nodes in the attack subgraph. The actions are shown as edges. From left to right, there are 4 stages and ability nodes belong to the same stage are aligned vertically. We call them stage 1 to stage 4, from left to right. There are 15 nodes on stage 1 and each represents the ability to send data to a port. After taking the action CompromiseNetwork, the attacker can control some processes and thus acquire the abilities on stage 2. There are 7 unique access tokens on stage 2. The attacker can then write to one of the 15 different files to advance to stage 3, with the ability to control the content of the corresponding file. After compromising privileged processes which read the files, the attacker achieves the goal at stage 4. There are 2 possible privileged access tokens the attacker can control at the final stage.

Attacks of P_5 comprise three steps. The first two steps are similar to P_4, the attacker compromises a process p_{net} through network and compromises another process p_{file} through file manipulation. p_{file} is unprivileged therefore the goal has not been achieved, however it has some SID that p_{net} does not have which can be used to write to another data file f. By writing to f, the attacker can finally compromise a privileged process which reads f, and this is the third step. To realize P_5 the attacker needs to exploit 3 vulnerabilities. Intuitively this makes P_5 more difficult to succeed than the other patterns. This is reflected by its highest cost among the 5 patterns.

5.3.2 Case 2: Local Attack

The initial ability is the control of an unprivileged process which runs with the account of a normal user TestNormalUser. The attack graph consists of 1970 graph nodes and 1216 deriving nodes. Figure 5.6 shows the 3 attack patterns found in the attack graph. The costs and numbers of instances of attack patterns are shown in Table 5.3.

P_6 is a Trojan attack. The attacker writes an executable and mounts a Trojan attack. P_7 is another Trojan attack, the difference is that the executables (dlls) are not running

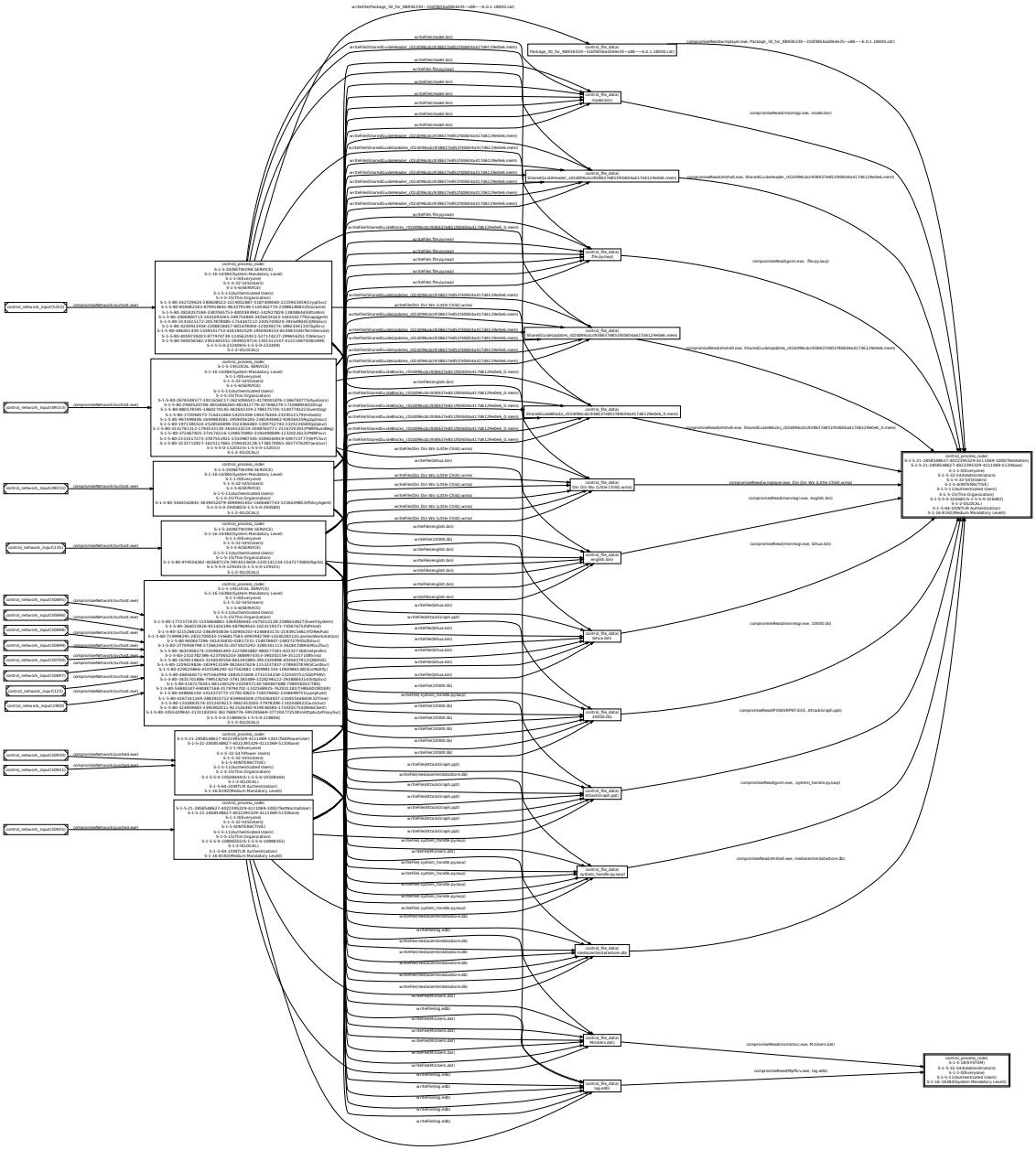


Figure 5.4. The Subgraph of the Attack Graph of Case 1 for Attack Pattern P_4

at the collecting snapshot. P_8 is realized by manipulating the input file of a process and compromising the process by exploiting some vulnerabilities.

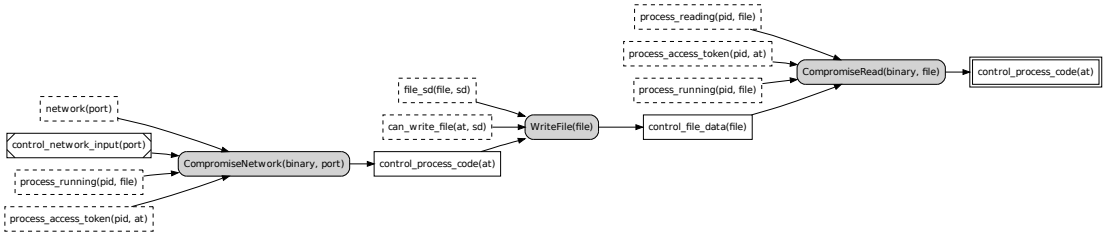


Figure 5.5. Attack Pattern P_4 with Details

$$
\begin{aligned}
P\_6 =& \mathsf{control\_process\_code} \rightarrow (\mathsf{writeFile}) \rightarrow \mathsf{control\_file\_data} \\
& \rightarrow (\mathsf{waitExecute}) \rightarrow \mathsf{control\_process\_code} \\
P\_7 =& \mathsf{control\_process\_code} \rightarrow (\mathsf{writeFile}) \rightarrow \mathsf{control\_file\_data} \\
& \rightarrow (\mathsf{hopeExecute}) \rightarrow \mathsf{control\_process\_code} \\
P\_8 =& \mathsf{control\_process\_code} \rightarrow (\mathsf{writeFile}) \rightarrow \mathsf{control\_file\_data} \\
& \rightarrow (\mathsf{compromiseRead}) \rightarrow \mathsf{control\_process\_code}
\end{aligned}
$$

Figure 5.6. Attack Patterns of Local Attacks

P_6, P_7 and P_8 share some similarities with P_2, P_3 and P_4, respectively. The difference is that in local attacks, the attacker has already controlled a local process, therefore a compromising through network is not necessary.

5.3.3 Discussions on Attacks

Since WACCA models software vulnerabilities, the success of some attacks depends on the success of identifying and exploiting vulnerabilities. The actions that require software exploitation are CompromiseRead and CompromiseNetwork. Some common practices for system administration can help reduce these attacks, e.g., only enable necessary server programs, patch software regularly to eliminate known bugs, etc.

Some attacks are Trojan attacks. The success depends on two factors (1) The attacker can write to an executable (2) Users, especially privileged users, will execute the Trojan. To prevent (1), one solution is to strengthen the security descriptors of the executables. For example, it might be unnecessary to give AuthenticatedUsers the right to write to the executables in patterns P_2 and P_6.

There are certain issues we didn't consider in our model. For example, there are several dll files that can be used in Trojan attacks of pattern P_3 and P_7. We contacted the software vendor about these files and were informed that they have employed digital signature to make sure that, even when these dll files are changed by an attacker, the malicious code will not be executed since the attacker will not be able to forge a valid digital signature.

Services are related to several deriving rules in WACCA. However in both case studies there are no attacks related to services. By examining the security descriptors of the services on the host, we found that only highly privileged accounts have critical rights on services, such as WRITE_DAC, WRITE_OWNER, SERVICE_CHANGE_CONFIG, etc. If considering other initial abilities or goals, there might be attacks related to services. Also the case studies are on the particularly configured host, there could be service-related attacks if other software products are installed, or existing software packages are configured improperly.

5.4 Discussions

WACCA can find attacks based on the abilities and deriving rules that we consider in the model. There are certain things not currently in the tool. For example, interprocess communications such as pipes and RPC are not considered. To consider more system interactions, new deriving rules should be added. More abilities will have to be included, when we have new attack goals in mind. For example, concerning data privacy, the ability to read files should be included. New objects should be included, when we want to model new types, e.g., registry keys. WACCA serves as a prototype to demonstrate the effective-

ness of our analysis model, therefore we include the most important and well known object types and attacker's abilities.

There are possibilities to further refine some abilities. For example, the ability to control the code of a process is on a 0 or 1 basis, i.e., the attacker can either completely control a process or not at all. It's possible that some vulnerability can let the attacker influence the process without completely control the process. For example, the attacker might be able to manipulate a buffer that is used for a process to write to some critical file, but he cannot command the process to execute arbitrary code.

Another issue is the completeness of the information. As discussed in Subsection 5.1.2, to consider the deriving rule of compromising a buggy program by crafting content of its input files, one needs the list of all possible input files of the target program. This might not be practical and we use the approximate approach as discussed previously. This technique is inspired by [26].

6 SUMMARY

This dissertation introduces a generic methodology to analyze access control policies in operating systems. We analyze the effectiveness of access control policies under attack scenarios. An attack scenario consists of two parts: (1) the resources that an attacker initially has (2) the attack objective of the attacker. Under an attack scenario, the effectiveness of a policy is evaluated by the possible attacks that can be carried out to achieve the attack objective. An attack is defined by a series of attack actions that change system state. We use the state of a system to represent the control an attacker has on the system. All possible attacks are encoded in a host attack graph, in which each node is a system state and each edge represents an attack action that can incur a state change. Analyses can be performed on a host attack graph, e.g., to group similar attacks into attack patterns, or to find the list of minimal attack paths.

We apply the approach to analyze access control policies in Linux systems and Windows systems. The tool under Linux is called Vulnerability Surface ANalyzer (VulSAN) and the tool under Windows is called Windows Access Control Configuration Analyzer (WACCA). Both tools have similar components: (1) a fact collector that collects system facts and policy configuration, (2) an attack graph generator that generates a host attack graph, and (3) an attack graph analyzer that analyzes the host attack graph. Both tools run automatically after we manually define the attack scenario and the rules of attacker's actions, which include the pre-conditions and effects of each action. System facts are encoded using Prolog facts, and the rules of attacker's actions are encoded using Prolog rules. The attack graph generator uses the Prolog reference engine to generate the host attack graph.

We use VulSAN to analyze the default policies that shipped with Ubuntu, Fedora and SUSE Linux. Several opportunities to strengthen the protection are found. We also use the results to compare policies of SELinux and AppArmor on a version of Ubuntu, and discuss the comparison results. We use WACCA to analyze the policy configuration on a

Windows Vista host, and find several attack patterns. The results suggest that the security configurations of several software products could be improved.

The models and tools discussed in this dissertation can help normal users, system administrators and software developers to better understand the effectivenesses and weaknesses of access control configurations in operating systems. It is especially helpful given the complexities of typical access control policies in operating systems.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Trent Jaeger. *Operating System Security, Synthesis Lectures on Information Security, Privacy, and Trust*. Morgan & Claypool Publishers, 2008.

[2] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[3] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971. Reprinted in ACM Operating Systems Review, 8(1):18-24, Jan 1974.

[4] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[5] Shuo Chen, John Dunagan, Chad Verbowski, and Yi-Min Wang. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[6] Understanding and configuring User Account Control in Windows Vista. `http://technet.microsoft.com/en-us/library/cc709628.aspx`.

[7] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical Report 01-043, NAI Labs, December 2001.

[8] AppArmor application security for Linux. `http://www.novell.com/linux/security/apparmor/`.

[9] N. Provos. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium*, pages 252–272, August 2003.

[10] Pratyusa Manadhata, Jeannette Wing, Mark Flynn, and Miles McQueen. Measuring the attack surfaces of two FTP daemons. In *Proceedings of the 2nd ACM Workshop on Quality of Protection*, pages 3–10, 2006.

[11] Mandatory Integrity Control. `http://msdn.microsoft.com/en-us/library/bb648648(VS.85).aspx`.

[12] Services and service accounts security planning guide. `http://technet.microsoft.com/en-us/library/cc170953.aspx`.

[13] Securable objects. `http://msdn.microsoft.com/en-us/library/aa379557(VS.85).aspx`.

[14] Access rights and access masks. `http://msdn.microsoft.com/en-us/library/aa374902(VS.85).aspx`.

[15] Security descriptors. http://msdn.microsoft.com/en-us/library/aa379563(VS.85).aspx.

[16] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable grids for visualizing and authoring computer security policies. In *Proceeding of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems*, pages 1473–1482, 2008.

[17] Robert W. Reeder, Patrick Gage Kelley, Aleecia M. McDonald, and Lorrie Faith Cranor. A user study of the expandable grid applied to P3P privacy policy visualization. In *Proceedings of the 7th ACM workshop on Privacy in the Electronic Society*, pages 45–54, 2008.

[18] Sudhakar Govindavajhala and Andrew W. Appel. Windows access control demystified. Technical Report TR-744-06, Department of Computer Science, Princeton University, January 2006.

[19] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, 2002.

[20] Tresys technology, SETools – Policy analysis tools for SELinux. Available at http://oss.tresys.com/projects/setools.

[21] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, August 2003.

[22] Trent Jaeger, Xiaolan Zhang, and Fidel Cacheda. Policy management using access control spaces. *ACM Transactions on Information Systems Security*, 6(3):327–364, 2003.

[23] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[24] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick Drew McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 91–100, 2007.

[25] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and comparing the protection quality of security enhanced operating systems. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS)*, 2009.

[26] Prasad Naldurg, Stefan Schwoon, Sriram K. Rajamani, and John Lambert. *NETRA*: Seeing through access control. In *Proceedings of the 4th ACM Workshop on Formal Methods in Security Engineering*, pages 55–66, 2006.

[27] Avik Chaudhuri, Prasad Naldurg, Sriram K. Rajamani, G. Ramalingam, and Lakshmisubrahmanyam Velaga. EON: Modeling and analyzing dynamic access control systems with logic programs. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 381–390, 2008.

[28] Matt Miller. Modeling the trust boundaries created by securable objects. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, pages 1–7, Berkeley, CA, USA, 2008. USENIX Association.

[29] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring relative attack surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security*, December 2003.

[30] Michael Howard. Mitigate security risks by minimizing the code you expose to untrusted users. *MSDN Magazine*, November 2004.

[31] Steve Lipner. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 2–13, 2004.

[32] Pratyusa K. Manadhata, Kymie M. C. Tan, Roy A. Maxion, and Jeannette M. Wing. An approach to measuring a system's attack surface. Technical Report CMU-CS-07-146, CMU, August 2007.

[33] NSA. Security Enhanced Linux. `http://www.nsa.gov/selinux/`.

[34] Achim Leitner. Novell and Red Hat security experts face off on AppArmor and SELinux. *Linux Magazine*, (69), 2006.

[35] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[36] Hao Chen, Drew Dean, and David Wagner. Setuid demystified. In *Proc. USENIX Security Symposium*, pages 171–190, August 2002.

[37] SELinux object classes and permissions reference. `http://oss.tresys.com/projects/refpolicy/wiki/ObjectClassesPerms`.

[38] AppArmor profiles package. `http://packages.ubuntu.com/gutsy/base/apparmor-profiles`.

[39] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for Security-Enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004.

[40] Susan Hinrichs and Prasad Naldurg. Attack-based domain transition analysis. In *Annual Security Enhanced Linux Symposium*, 2006.

[41] Giorgio Zanin and Luigi V. Mancini. Towards a formal model for security policies specification and validation in the SELinux system. In *Proceedings of ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 136–145, 2004.

[42] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, March 1976.

[43] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[44] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 273–284, 2002.

[45] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *In Proceedings of the 15th Computer Security Foundation Workshop*, pages 49–63, 2002.

[46] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224, 2002.

[47] Steven Noel, Sushil Jajodia, Brian O'Berry, and Michael Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, page 86, 2003.

[48] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 336–345, 2006.

[49] Deborah D. Downs, Jerzy R. Rub, Kenneth C. Kung, and Carole S. Jordan. Issues in discretionary access control. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 208–218, April 1985.

[50] National Computer Security Center. A guide to understanding discretionary access control in trusted systems, September 1987. NCSC-TG-003.

[51] M. D. Mcllroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software–Practice and Experience*, 22(8):673–694, August 1992.

[52] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil D. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th Conference on Systems Administration (LISA 2000)*, pages 355–368, December 2000.

[53] LIDS: Linux intrusion detection system. `http://www.lids.org/`.

[54] David R. Wichers, Douglas M. Cook, Ronald A. Olsson, John Crossley, Paul Kerchen, Karl N. Levitt, and Raymong Lo. Pacl's: An access control list approach to anti-viral security. In *Proceedings of the 13th National Computer Security Conference*, pages 340–349, October 1990.

[55] Kent J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.

[56] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2000.

[57] The advantages of running applications on Windows Vista. `http://msdn2.microsoft.com/en-us/library/bb188739.aspx`.

[58] David D. Clark and David R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, May 1987.

[59] Paul A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 130–139, 1988.

[60] Theodore M. P. Lee. Using mandatory integrity to enforce "commercial" security. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 140–146, 1988.

[61] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2007.

[62] Ziqing Mao, Ninghui Li, Hong Chen, and Xuxian Jiang. Trojan horse resistant discretionary access control. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, pages 237–246, 2009.

[63] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[64] A. C. Myers. JFlow: Practical mostly-static information-flow control. In *Proceedings of the 1999 Symposium on Principles of Programming Languages*, January 1999.

[65] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[66] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium*, February 2006.

[67] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[68] Timothy Fraser. LOMAC: MAC you can live with. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.

[69] SELinux for distributions. `http://selinux.sourceforge.net`.

[70] AppArmor development. `http://developer.novell.com/wiki/index.php/Apparmor`.

[71] Python for Windows extensions. `http://python.net/crew/mhammond/win32/`.

[72] Handle. `http://technet.microsoft.com/en-us/sysinternals/bb896655.aspx`.

[73] The XSB Research Group. The XSB programming system. `http://xsb.sourceforge.net/`.

VITA

VITA

Hong Chen received his Ph.D. and M.S. degrees in Computer Science in 2009 and 2007, from Purdue University. He received his B.E. degree in Computer Science and Technology from Tsinghua University in 2004. Hong Chen's research interests are in the areas of information security. He did research projects related to operating system security, database security, web browser security and role-based access control. He received the Frederick N. Andrews Fellowship from 2004 to 2006. Hong Chen did summer internships with Yahoo!, IBM Almaden Research Center, and Microsoft Research in 2005, 2006, and 2008, respectively.