Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems

Hong Chen   Ninghui Li   Ziqing Mao
Center for Education and Research in Information Assurance and Security
and Department of Computer Science, Purdue University
{chen131, ninghui, zmao}@cs.purdue.edu

Abstract

Host compromise is a serious computer security problem today. To better protect hosts, several Mandatory Access Control systems, such as Security Enhanced Linux (SELinux) and AppArmor, have been introduced. In this paper we propose an approach to analyze and compare the quality of protection offered by the policies of different mechanisms. We introduce the notion of vulnerability surfaces under attack scenarios as the measurement of protection quality, and implement a tool called VulSAN *for analyzing and comparing protection quality of these MAC systems for Linux. In* VulSAN, *we encode security policies, system states, and system rules using logic programs. Given an attack scenario,* VulSAN *computes a host attack graph and the vulnerability surface. We apply our approach to compare SELinux and AppArmor policies in several Linux distributions and discuss the results. Our tool can also be used by Linux system administrators as a system hardening tool. Because of its ability to analyze SELinux as well as AppArmor policies, it can be used for most enterprise Linux distributions and home user distributions.*

1 Introduction

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) [6, 7], Security Enhanced Linux (SELinux) [19], AppArmor [5, 1], and Usable Mandatory Integrity Protection (UMIP) [16]. 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 [3], and AppArmor is supported in Linux distributions including SUSE, PLD, Pardus Linux, Annvix, Ubuntu and Mandriva [2].

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 has a different MAC mechanism or different MAC policy?

In this paper, 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 [11] which is a concept in 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 (details are discussed in Section 2). 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. 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 comparison helps 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.

The rest of the paper is organized as follows: Section 2 presents the background and related work. Section 3 discusses our analysis approach. Section 4 talks about the implementation of VulSAN. Section 5 presents the results of comparing SELinux with AppArmor in several Linux distributions. Section 6 concludes the paper.

2 Background and Related Work

Security-Enhanced Linux [19] (SELinux) is a security mechanism in Linux that has been developed to support a wide range of security policies. SELinux has been integrated into Linux Kernel since 2.6. In SELinux, every process has a domain and every object has a type. Objects are categorized into object security classes, such as files, folders, sockets, etc. A set of operations are defined over each object security class (e.g., read, write, execute, lock, create, rename, etc for a file). A SELinux policy defines processes of which domains can access objects of which types with which operations. A policy also defines how to determine the domain of a process and how the domain changes when a process executes another program.

AppArmor [1] is an access control system that confines the access permissions on a per program basis. It confines programs that are likely to be attacked, e.g., server programs that face network and setuid root programs. For every protected program, AppArmor defines a program profile. A profile is a list of permitted accesses, including file accesses and capabilities. The profiles of all protected programs constitute an AppArmor policy. If a program does not have a profile, it is by default not confined. If a program has a profile, it only has permissions specified in the profile.

Previous approaches for analyzing SELinux security policies include Gokyo [14, 13], SLAT [8], PAL [21], APOL [24, 10], SELAC [25], NETRA [18], and PALMS [9]. Gokyo [14, 13] 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 [8] 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 [21] 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 [24] is a tool to analyze the relationships between domains and types in a SELinux policy. In [10] 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 [] 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 [18] 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 [9] 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 [4].

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 5.2, 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 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 [15] 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.

Attack surface is proposed as a metric to measure the attackability of a system [11, 12]: "The attack surface of an app is the union of code, interfaces, services, protocols, and practices available to all users, with a strong focus on what is accessible to unauthenticated users." The heuristic is that a larger attack surface indicates a less secure system. Reducing the attack surface is part of the Microsoft Security Development Lifecycle (SDL) [11]. In [17], Manadhata et al. propose to measure a system's attack surface in terms of three kinds of resources used in attacks on the system: methods, channels and data.

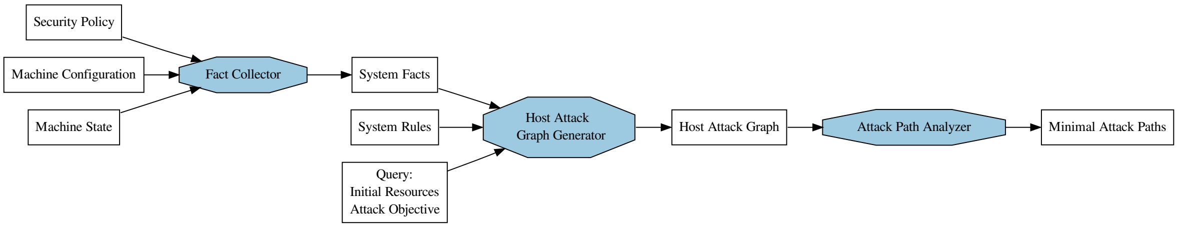

Figure 1. Solution Overview

Two IMAP and two FTP programs are evaluated using this method.

Attack graph is used to analyze the security of networks in existing works [22, 20]. Our approach also computes a graph similar to an attack graph. However, our problem space is different, as we consider control of processes under different access control restrictions, rather than control of network-connected hosts. Also, we perform additional analysis on the resulted graph to generate all minimal attack paths for analysis and comparison purposes.

3 Overview of Our 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. In this paper, we present a first attempt at coming up with a pragmatic definition.

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.

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 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 paper 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 paper, 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 Our Tool

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

4.1 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 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 Enforcement Access Vector Rules and Type Enforcement Transition Rules. We also define all the domains and types. Figure 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

```
(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/setgid/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 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 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. Sample Facts of AppArmor Policy

gives some sample Prolog facts of an AppArmor policy. Our parser for AppArmor policy is based on *apparmor_parser*.

4.2 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. Figure 5 gives some relevant predicates to describe such enforcement.

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 [23], 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

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.

Figure 5. Sample System Rules

```
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 6. Rules for Domain 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 6.

```
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 7. Predicates for Initial Attack States and Goal Attack States

```
function GENERATE_GRAPH_NODE($s$)
    if $s$ is already a graph node then
        return
    Add $s$ as a graph node
    if $s$ is a goal attack state then
        return
    for all program prog that $s$ can execute do
        $s'$ ← the attack state after executing prog
        Add $(s, s')$ as a graph edge with label prog
        Generate_Graph_Node($s'$)

function GENERATE_HOST_ATTACK_GRAPH
    for all Initial attack state $s$ do
        Generate_Graph_Node($s$)
```

Figure 8. Algorithm for Host Attack Graph Generation

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 7(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 7(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 8.

4.3 Attack Path Analyzer

Attack Path Analyzer finds all the minimal attack paths in a host attack graph. Figure 9 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.

```
function GENERATE_MINIMAL_ATTACK_PATHS
    $V \leftarrow V \cup v_g$
    for all goal attack state node $v$ do
        add an edge from $v$ to $v_g$,
        the exploited program for the edge is empty
    for all $v \in V$ do
        MP($v$) $\leftarrow \phi$
    for all initial attack state node $v$ do
        MP($v$) $\leftarrow \{\phi\}$
    repeat
        stable $\leftarrow$ true
        for all $e \in E$ do
            for all $p \in$ MP($e.v_1$) do
                $p' \leftarrow$ append($p, e$)
                if $\exists p_0 \in$ MP($e.v_2$) s.t. $V(p') \subset V(p_0)$ then
                    Remove all such paths from MP($e.v_2$)
                if not $\exists p_1 \in$ MP($e.v_2$) s.t. $V(p') \supset V(p_1)$ then
                    MP($e.v_2$) $\leftarrow$ MP($e.v_2$) $\cup \{p'\}$
                    stable $\leftarrow$ false
    until stable
    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 9. Minimal Attack Paths Generation

4.4 Tool Status

We have implemented VulSAN in Linux. VulSAN has been used to evaluate SELinux and AppArmor in several Linux distributions. We plan to further improve the tool and release it to the public in the future (possibly under the terms and conditions of the GNU General Public License (GPL)).

5 Comparing SELinux with AppArmor

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).

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.

5.1 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 10, Figure 11 and Figure 12, respectively. The comparison of minimal attack paths between SELinux and AppArmor is shown in Figure 13.

Among the three cases, AppArmor has the smallest vulnerability surface. SELinux has all the minimal attack paths AppArmor has and some additional ones.

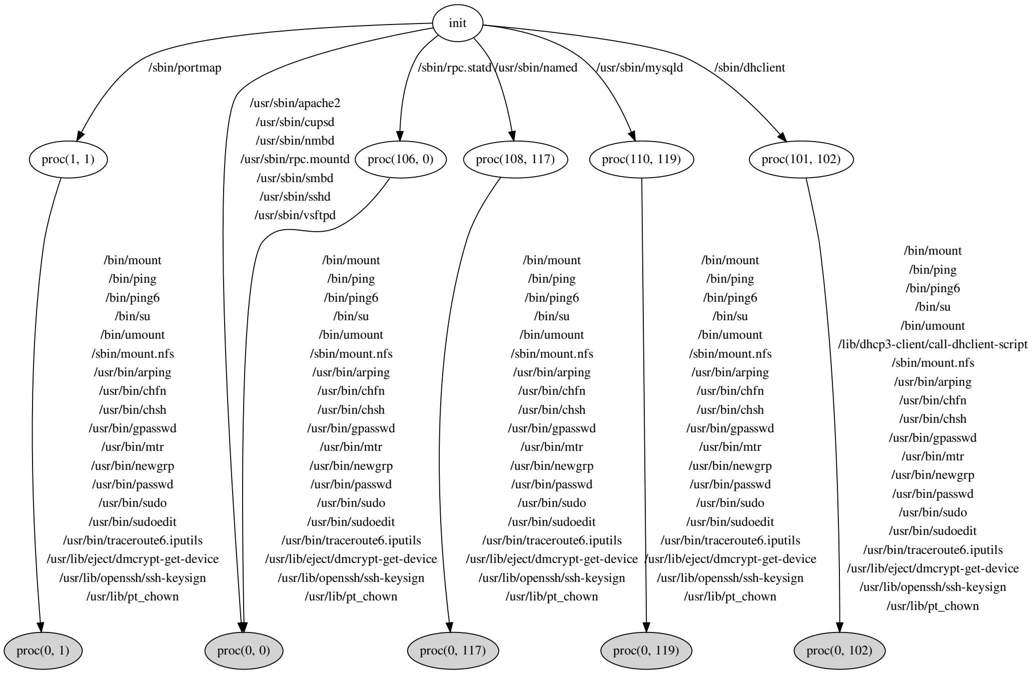

Figure 10. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with DAC only)

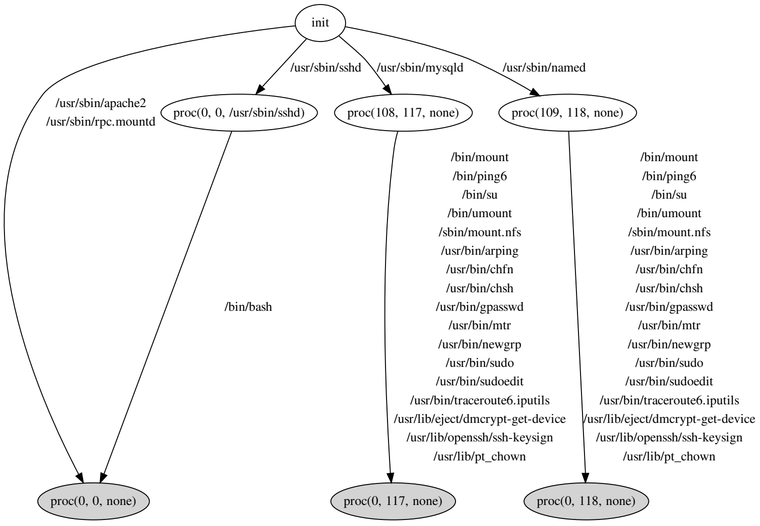

Figure 11. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with AppArmor)

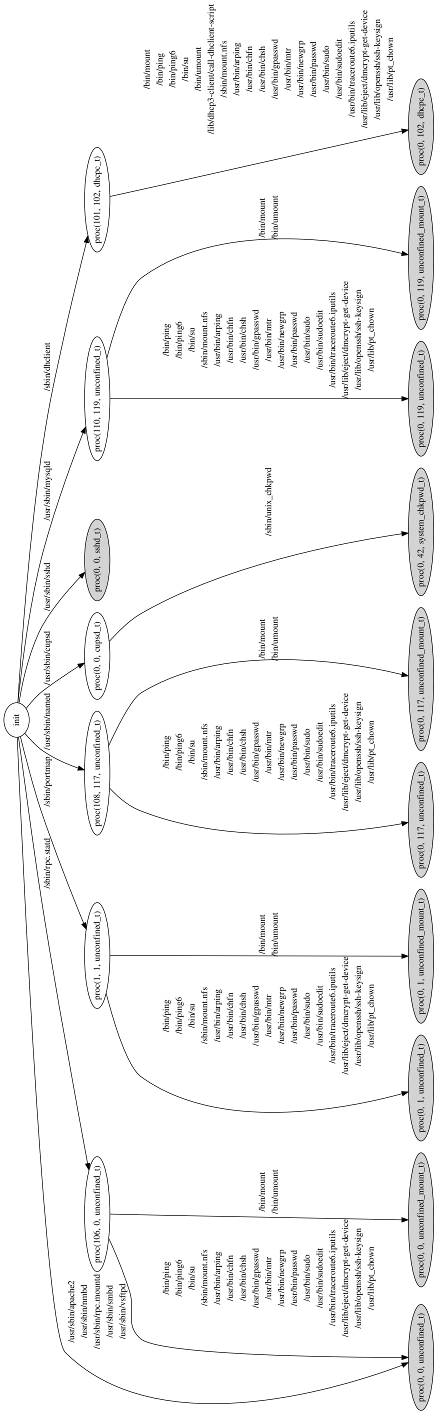

Figure 12. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux)

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.

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 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.

Our analysis shows that among the seven network-facing programs running as root in Ubuntu 8.04 Server

	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

Figure 13. Minimal Attack Paths Comparison for a Remote Attacker to Install a Rootkit

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 14 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.

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 15 and Figure 16 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 /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 at-

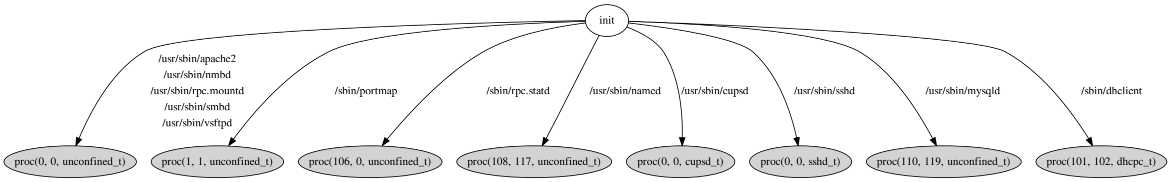

Figure 14. Host Attack Graph for a Remote Attacker to Leave a Weak Trojan (Ubuntu 8.04 with SELinux)

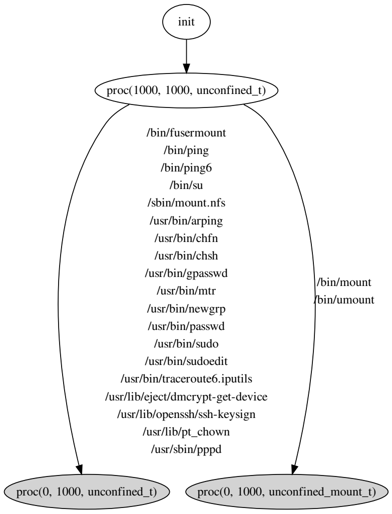

Figure 15. Host Attack Graph for a Local Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux)

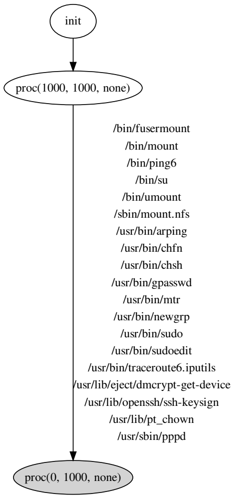

Figure 16. Host Attack Graph for a Local Attacker to Install a Rootkit (Ubuntu 8.04 with AppArmor)

tack 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.

5.2 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 17 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 12) 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.

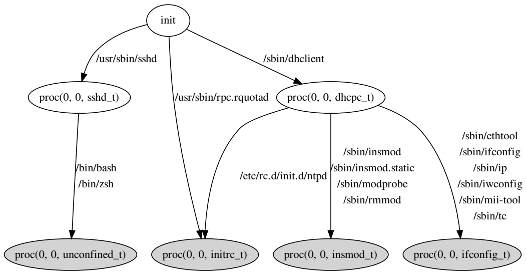

Figure 17. Host Attack Graph for a Remote Attacker to Install a Rootkit (Fedora 8 with SELinux)

Figure 18 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.

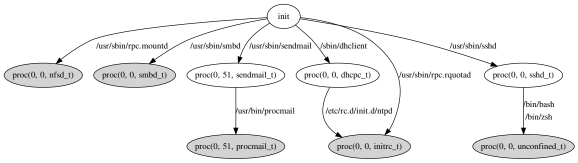

Figure 18. Host Attack Graph for a Remote Attacker to Leave a Strong Trojan (Fedora 8 with SELinux)

Different Versions of AppArmor

We have analyzed the vulnerability surface of SUSE Linux Enterprise Server 10 (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.

The vulnerability surface of SLES 10/AppArmor under the scenario that a remote attacker wants to install a rootkit (as shown in Figure 19) is not directly comparable with that of Ubuntu 8.04/AppArmor. The two distributions expose different vulnerability surfaces.

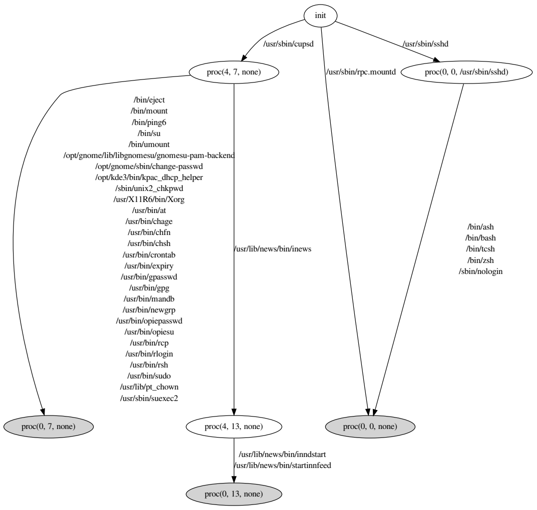

Figure 19. Host Attack Graph for a Remote Attacker to Install a Rootkit (SUSE Linux Enterprise Server 10 with AppArmor)

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 20. 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

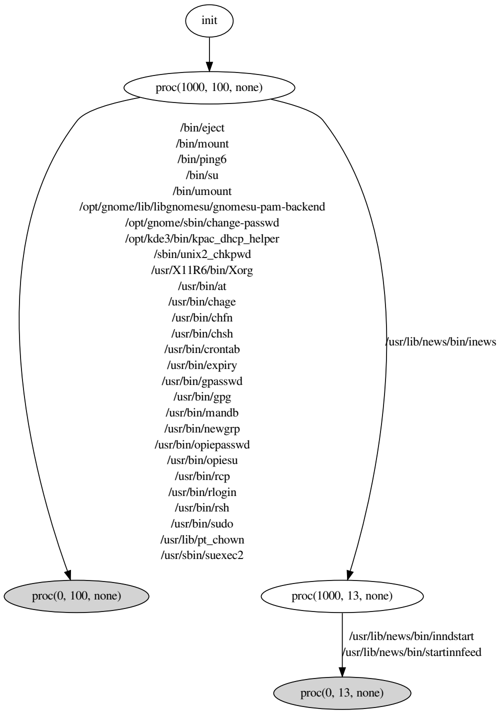

Figure 20. Host Attack Graph for a Local Attacker to Install a Rootkit (SUSE Linux Enterprise Server 10 with AppArmor)

the DAC policy. If we only consider MAC policy, e.g., SELinux policy, the result may not be accurate. Figure 21 shows the host attack graph for a remote attacker to install a rootkit, when we only consider SELinux policy but not DAC policy. Compared to the host attack graph that considers both DAC and MAC policy (shown in Figure 12), 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 unprivileged. To control root uid the attacker needs to do another exploit, e.g., by exploiting a setuid root program.

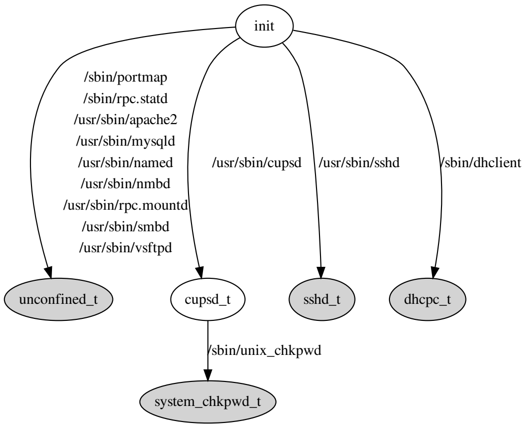

Figure 21. Host Attack Graph for a Remote Attacker to Install a Rootkit (Ubuntu 8.04 with SELinux – only Considering SELinux Policy)

5.3 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.

6 Conclusions

In this paper, we propose an approach to analyze and compare the protection quality offered by policies of different Mandatory Access Control mechanisms in security-enhanced operating systems. Our analysis is based on the security policy, system state and system configuration. We develop a tool to generate the host attack graph for a given attack scenario. We propose to use vulnerability surface to measure the protection quality of a system. We evaluate our approach by analyzing and comparing SELinux and AppArmor in several Linux distributions.

6.1 Acknowledgements

This work is supported by NSF CNS-0448204 (CAREER: Access Control Policy Verification Through Security Analysis And Insider Threat Assessment), and by sponsors of CERIAS. We also thank the anonymous reviewers for NDSS and the shepherd of our paper Crispin Cowan for valuable comments that have greatly improved the paper.

References

[1] Apparmor application security for linux. *http://www.novell.com/linux/security/apparmor/*.

[2] Apparmor development. *http://developer.novell.com/wiki/index.php/Apparmor*.

[3] Selinux for distributions. *http://selinux.sourceforge.net*.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Mar. 1976.

[5] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th Conference on Systems Administration (LISA 2000)*, pages 355–368, Dec. 2000.

[6] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proc. IEEE Symposium on Security and Privacy*, May 2000.

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

[8] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[9] B. Hicks, S. Rueda, L. S. Clair, T. Jaeger, and P. D. McDaniel. A logical specification and analysis for selinux mls policy. In *SACMAT*, pages 91–100, 2007.

[10] S. Hinrichs and P. Naldurg. Attack-based domain transition analysis. In *Annual Security Enhanced Linux Symposium*, 2006.

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

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

[13] 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.

[14] T. Jaeger, X. Zhang, and F. Cacheda. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[15] A. Leitner. Novell and red hat security experts face off on apparmor and selinux counterpoint. *Linux Magazine*, (69), 2006.

[16] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *Proc. IEEE Symposium on Security and Privacy*, May 2007.

[17] P. K. Manadhata, K. M. C. Tan, R. A. Maxion, and J. M. Wing. An approach to measuring a system's attack surface. Technical Report CMU-CS-07-146, CMU, August 2007.

[18] P. Naldurg, S. Schwoon, S. K. Rajamani, and J. Lambert. *NETRA*: : seeing through access control. In *FMSE*, pages 55–66, 2006.

[19] NSA. Security enhanced linux. *http://www.nsa.gov/selinux/*.

[20] X. Ou, W. F. Boyer, and M. A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345, New York, NY, USA, 2006. ACM.

[21] B. Sarna-Starosta and S. 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.

[22] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 273, Washington, DC, USA, 2002. IEEE Computer Society.

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

[24] Tresys technology, setools - policy analysis tools for selinux. Available at *http://oss.tresys.com/projects/setools*.

[25] G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 136–145, 2004.