Host Integrity Protection Through Usable Non-discretionary Access Control

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

November 22, 2006

Abstract

Existing non-discretionary access control systems (such as Security Enhanced Linux) are difficult to use by ordinary users. We identify several principles for designing usable access control system and introduce the Host Integrity Protection Policy (HIPP) model that adds usable non-discretionary access control to operating systems. The HIPP model is designed to defend against attacks targeting network server and client programs and to protect the system from careless mistakes users might make. It aims at not breaking existing applications or existing ways of using and administering systems. HIPP has several novel features to achieve these goals. For example, it supports several types of partially trusted programs to support common system administration practices. Furthermore, rather than requiring file labeling, it uses information in the existing discretionary access control mechanism for non-discretionary access control. We also discuss our implementation of the HIPP model for Linux using the Linux Security Modules framework, as well as our evaluation results.

1 Introduction

Host compromise is one of the most serious computer security problems today. Computer worms propagate by first compromising vulnerable hosts and then propagate to other hosts. Compromised hosts may be organized under a common command and control infrastructure, forming botnets. Botnets can then be used for carrying out attacks, such as phishing, spamming, distributed denial of service, and so on. These threats can be partially dealt with at the network level using valuable technologies such as firewalls and network intrusion detection systems. However, to effectively solve the problem, one has to also deal with the root cause of these threats, namely, the vulnerability of end hosts. Two key reasons why hosts can be easily compromised are: (1) software are buggy, and (2) the discretionary access control mechanism in today's operating systems is insufficient for protecting hosts against network-based attacks. Network-facing server and client programs may contain software vulnerabilities, and users may download and run malicious programs. Often times programs are running with system privileges, and malicious programs can abuse the privileges to take over the host.

There are a lot of research efforts on making computer systems more secure by adding non-discretionary access control to operating systems, e.g., Janus [10], DTE Unix [3, 2], Linux Intrusion Detection System (LIDS) [11], LOMAC [8], systrace [17], AppArmor [1], and Security Enhanced Linux (SELinux) [15]. Systems such as systrace and SELinux are flexible and powerful. Through proper configuration, they could provide highly-secure systems. However, they are also complex and intimidating for normal users. They are designed *by the experts* and *for the experts*. For example, policies for systrace take the form of conditions on system call

parameters. Understanding and configuring policies require intimate knowledge about the operating system programming interface. As another example, SELinux has 29 different classes of objects, hundreds of possible operations, and thousands of policy rules for a typical system. The SELinux policy interface is daunting even for security experts. While SELinux makes sense in a setting where the systems run similar applications, and sophisticated security expertise is available, its applicability to systems used by ordinary users is unclear. While SELinux has been included in Redhat's Fedora Core Linux distribution since 2004, it is shipped with the Targeted Policy, which confines several dozens of well-known daemon programs. This approach, however, violates the fail-safe defaults principle [20], as a daemon program with no policy will by default run unconfined. To protect more programs, system administrators still need to understand and configure the policy. While there is ongoing research on developing tools to help the administration process, easy-to-use and effective tools are still not available. There have been several books devoted to SELinux, e.g., [14, 13], and an annual SELinux symposium since 2004. Many companies are offering consulting and training services for SELinux. While these facts speak to the importance of adding non-discretionary access control mechanisms to operating systems and the popularity of SELinux, it also shows that SELinux is complicated and difficult to use.

Our view is that in order to use a non-discretionary access control technology to deal with threats posed by botnets, the technology must be usable by ordinary users with limited knowledge in systems security, as it is the hosts used by these users that are more likely to be compromised and taken over. In this paper, we tackle the problem of designing and implementing a usable non-discretionary access control[1] system to protect end hosts. In this paper we introduce a policy model that we call the Host Integrity Protection Policy (HIPP) model. The objective of the HIPP model is to defend against attacks targeting network-facing servers and network client programs (such as web browsers) and careless mistakes users might make. While HIPP is designed for UNIX-based system, we believe that some ideas can be applied to other operating systems. HIPP classifies processes into high-integrity and low-integrity, and aims at minimizing the channels that an attacker could get control of a high-integrity process by exploiting program vulnerabilities or human engineering attacks. One novel feature is that, unlike previous NonDAC systems, HIPP does not require labeling of files, and use existing DAC information instead. For example, a low-integrity process (even if running as root) by default is forbidden from writing to any file that is not world-writable according to DAC (i.e., the file is only owner-writable or group-writable). Thus HIPP does not require changes to file system implementation and can reuse valuable information in existing DAC setting, avoiding the process of labeling files, which can be both time-consuming and error-prone. While HIPP focuses primarily on system integrity, it also has confidentiality protection, as it forbids a low-integrity process from reading any file owned by a system account (such as root, bin, etc) and is not readable by the world. This prevents attackers from reading files such as /etc/shadow. The design of HIPP is very much practical-oriented. It aims at not breaking existing applications or existing ways of using and administering systems. To achieve this, HIPP has several interesting features distinguishing from previous integrity models. For example, while by default a process drops its integrity when receiving any remote network traffic, certain programs (such as sshd) can be declared to be remote administration points, so that processes running them do not drop integrity upon receiving network traffic. The policy model requires only a relatively small number of exceptions and settings to be specified to support the needs of an individual host. The specification is done through a policy file that uses concepts that ordinary system administrators are familiar with.

We have implemented HIPP for Linux using the Linux Security Modules (LSM) framework [22], and have been using evolving prototypes of the HIPP system within our group for a few months. We discuss and evaluate our implementation in this paper. We plan to release the code to the open-source community in near future.

The contributions of this paper are three-fold.

1. We identify several design principles for designing usable access control mechanisms. Not all of these

[1] We use non-discretionary access control (NonDAC), rather than mandatory access control (MAC), to avoid any potential connotation of multi-level security.

principles are new. Several of them have appeared before in the literature. However, we believe that putting these principles together and illustrating them through the design of an actual system would useful for other researchers and developers designing and implementing usable access control systems.

2. We introduce the HIPP model, a simple, practical nonDAC model for host protection. It has several novel features compared with existing integrity protection models.

3. We report the experiences and evaluation results of our implementation of HIPP under Linux.

The rest of this paper is organized as follows. We discuss design principles in Section 2. The HIPP model is described in Section 3. Our implementation and its evaluation are described in Section 4. We then discuss related work in Section 5 and conclude in Section 6.

2 Design Principles for Usable Access Control Systems

Principle 1 *Provide "good enough" security with a high level of usability, rather than "better" security with a low level of usability.*

Our philosophy is that rather than providing a protection system that can theoretically provide very strong security guarantees but requires huge effort and expertise to configure correctly, we aim at providing a system that is easy to configure and can greatly increase the level of security by reducing the attack surfaces. Sandhu [21] made a case for good-enough security, observing that "*cumbersome technology will be deployed and operated incorrectly and insecurely, or perhaps not at all.*" Sandhu also identified three principles that guide information security, the second of which is "*Good enough always beat perfect*"[2]. He observed that the applicability of this principle to the computer security field is further amplified because there is no such thing as "perfect" in security, and restate the principle as "Good enough always beats 'better but imperfect'."

There may be situations that one would want stronger security guarantees, even though the cost of administration is much more expensive. However, to defend against threats such as botnets, what one needs to do is to protect the most vulnerable computers on the Internet, i.e., computers that are managed by users with little expertise in system security. One thus needs a protection system with a high level of usability.

While it is widely agreed that usability is very important for security technologies, how to design an access control system that has high usability has not been explored much in the literature. One immediate conclusion from this principle is that *sometimes one needs to tradeoff security for simplicity of the design*. Below we discuss five other principles, which further help achieve the goal of usable access control. The following five principles can be viewed as "minor" principles for achieving the overarching goal set by the first principle.

Principle 2 *Provide policy, not just mechanism.*

Raymond discussed in his book [19] the topic of "*what UNIX gets wrong*" in terms of philosophy, and wrote "*perhaps the most enduring objections to Unix are consequences of a feature of its philosophy first made explicit by the designers of the X windowing system. X strives to provide 'mechanism, not policy'. [...] But the cost of the mechanism-not-policy approach is that when the user can set policy, the user must set policy. Nontechnical end-users frequently find Unix's profusion of options and interface styles overwhelming.*"

The mechanism-not-policy approach is especially problematic for security, as security mechanisms may be used incorrectly. A security mechanism that is very flexible and can be extensively configured is not just overwhelming for end users, it is also highly error-prone. While there are right ways to configure the mechanism to enforce some desirable security policies, there are often many more incorrect ways to configure a system. And the complexity often overwhelms users so that the mechanism is simply not enabled.

[2]The first one is "*Good enough is good enough*" and the third one is "*The really hard part is determining what is good enough.*"

This mechanism-not-policy philosophy is implicitly used in designing non-discretionary access control mechanism. For example, systems such LIDS, systrace, and SELinux all aim at providing a mechanism that can be used to implement a wide range of policies. While a mechanism is absolutely necessary for implementing a protection system, having only a low-level mechanism is not sufficient.

Principle 3 *Have a well-defined security objective, and design the policy for it.*

The first step of designing a policy is to identify a security objective, because only then can one make meaningful tradeoffs between security and usability. To make tradeoffs, one must ask and answer the question: if the policy model is simplified in this way, can we still achieve the security objective? Often times, non-discretionary access control systems do not clearly identify the security objectives. A security objective should identify two things: one is what kind of adversaries the system is designed to protect against, i.e., what abilities does one assume the adversaries have, and the other is what security properties one wants to achieve even in the presence of such adversaries. For example, often times achieving multi-level security is identified together with defending against network attacks. They are very different kinds of security objectives. It is well known that designing usable multi-level secure systems is extremely hard, and it seems unlikely that one can build a usable access control system can achieve both objectives. See Appendix A for additional discussions about the pitfalls of designing security mechanism without clear policy objective, using Chen et al.'s excellent paper on setuid [6] as a case study.

Principle 4 *Design a default policy for the objective that leaves a few special cases to be specified.*

One key issue in making an access control system usable is to reduce the amount of configuration one needs to do. Given the complexity of modern operating systems, no simple policy model can capture all accesses that need to be allowed and forbid all dangerous accesses. One key to usability is to have a default policy that captures the majority of the cases and leaves a few special cases specified as exceptions.

Principle 5 *Rather than trying to achieve absolute least privilege, try to achieve "limited privilege".*

It is widely recognized that one problem with existing discretionary access control mechanisms is that it does not support the least privilege principle [20]. For example, in traditional UNIX access control, many operations can be performed only by the root user. If any program needs to perform some of these operations, it needs to be given the root privilege. As a result, an attacker that exploits vulnerabilities in these programs can abuse these privileges. Many propose to remedy the problem by supporting strict least privilege. For example, the guiding principles for designing policies for systems such as SELinux, systrace, and AppArmor is to identify all objects a program needs to access when it is not under attack and grants access only to those objects. This approach results in large number of policy rules. We take the approach of separating operations into sensitive ones and non-sensitive ones, and limits access only to sensitive operations. This principle can be viewed as a corollary of the previous principle. We state it as a separate principle because of the popularity of the least privilege principle.

Principle 6 *Use familiar abstractions in policy specification interface.*

Psychological acceptability is one of the eight principles for designing security mechanisms in the classical paper by Salzer and Schroeder [20]. They wrote "*It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.*" This entails that the policy specification interface should use concepts and abstractions that administrators are familiar with. This principle is violated by systems such as systrace and SELinux.

3 The Host Integrity Protection Policies

We now introduce the Host Integrity Protection Policy (HIPP) model, which was guided by the principles identified in Section 2. While the description of the HIPP model in this section is based on our design for Linux, we believe that the model can be applied to other UNIX variants with minor changes. While some (but not all) ideas would be applicable also to non-Unix operating systems such as the Microsoft Windows family, investigating the suitability of HIPP or a similar model for Microsoft Windows is beyond the scope of this paper.

Our focus is to provide a policy model that can be implemented using an existing mechanism (namely the Linux Security Modules framework). We now identify the security objective of our policy model. We aim at protecting the system integrity of the host system (hence the name HIPP) against network-based attacks. We assume that network server and client programs contain bugs and can be exploited if the attacker is able to feed input to them. We assume that users may make careless mistakes in their actions, e.g., downloading a malicious program from the Internet and running it. However, we assume that the attacker does not have physical access to the host to be protected. Our policy model aims at ensuring that under most attack channels, the attacker can get limited privileges and cannot compromise the system integrity. For example, if a host runs privileged network-facing programs that contain vulnerabilities, the host will not be completely taken over by an attacker as a bot. The attacker may be able to exploit bugs in these programs to run some code on the host. However, the attacker cannot install rootkits. Furthermore, if the host reboots, the attacker does not control the host anymore. Similarly, if a network client program is exploited, the damage is limited. We also aim at protecting against indirect attacks, where the attacker creates malicious programs to wait for users to execute them, or creates/changes files to exploit vulnerabilities in programs that later read these files.

The usability goals for HIPP is that configuring it should not be more difficult than installing, administering, and maintaining an operating system. Also, our goals are not to break existing application programs or common ways for using the system. Depending on the needs of a system, the administrator of the system should be able to configure the system in a less-secure, but easier-to-user manner.

3.1 Basic Design of The HIPP Model

An important design question for any access control system is: What is a principal? That is, when a process requests to perform certain operations, what information about the process should be used in deciding whether the request should be authorized. The traditional UNIX access control system treats a pair of (uid,gid) as a principal. The effective uid and gid together determine the privileges of a process. As many operations can be performed only when the effective uid is 0, many programs owned by the root user are designated setuid. One problem with this approach is that it does not consider the possibility that these programs may be buggy. If all privileged programs are written correctly, then this approach is fine. However, when privileged programs contain bugs, they can be exploited so that attackers can use the privileges to damage the system.

As having just uid and gid is too coarse-granulated, a natural extension is to treat a triple of uid, gid, and the current program that is running in the process as a principal. The thinking is that, if one can identify all possible operations a privileged program would do and only allows it to do those, then the damage of an attacker taking over the program is limited. This design is also insufficient, however. Consider a request to load a kernel module[3] that comes from a process running the program insmod with effective user-id 0. As loading a kernel

[3] A loadable kernel module is a piece of code that can be loaded into and unloaded from kernel upon demand. LKMs (Loadable Kernel Modules) are a feature of the Linux kernel, sometimes used to add support for new hardware or otherwise insert code into the kernel to support new features. LKMs run in kernel mode, and can augment or even replace existing kernel features, all without a system reboot. However, by using LKMs the attackers are able to inject malicious code into the running kernel, such as modifying the system call table. LKMs is one popular method for implementing kernel-mode RootKits on Linux. For example, some malicious loadable kernel modules change the way that various system calls are handled by kernel.

module is what insmod is supposed to do, such access must be allowed. However, this process might be started by an attacker who has compromised a daemon process running as root and obtained a root shell as the result of the exploits. If the request is authorized, then this may enable the installation of a kernel rootkit, and lead to complete system compromise. One may try to prevent this by preventing the daemon program from running certain programs (such as shell); however, certain daemons have legitimate need to run shells or other programs that can lead to running insmod. In this case, a daemon can legitimately run a shell, the shell can legitimately run insmod, and insmod can legitimately load kernel modules. If one looks at only the current program together with (uid,gid), then any individual access needs to be allowed; however, the combination of them clearly needs to be stopped.

The analysis above illustrates that, to determine what the current process should be allowed to do, one has to consider the parent process who created the current process, the process who created the parent process, and so on. We call this the *request channel*. For example, if insmod is started by a series of processes that have never communicated with the network, then this means that this request is from a user who logged in through a local terminal. Such a request should be authorized, because it is almost certainly not an attacker, unless an attacker gets physical access to the host, in which case not much security can be provided anyway. On the other hand, if insmod is started by a shell that is a descendant of the ftp daemon process, then this is almost certainly a result from an attack; the ftp daemon and its legitimate descendants have no need to load a kernel module.

The key challenge is how to capture the information in a request channel in a succinct way. The domain-type enforcement approach used in SELinux and DTE Unix can be viewed as summarizing the request channel in the form of a domain. Whenever a channel represents a different set of privileges from other channels, a new domain is needed. This requires a large number of domains to be introduced.

The approach we take is to use a few fields associated with a process to record necessary information about the request channel. The most important field is one bit to classify the request channel into high integrity or low integrity. If a request channel is likely to be exploited by an attacker, then the process has low integrity. If a request channel may be used legitimately for system administration, the the process needs to be high-integrity. Note that a request channel may be both legitimately used for system administration and potentially exploitable. In this case, administrators must explicitly set the policy to allow such channels for system administration. The model tries to minimize the attack surface exposed by such policy setting when possible.

When a process is marked as low-integrity, this means that it is potentially contaminated. We do not try to identify whether a process is actually attacked. The success of our approach depends on the observation that with such an apparently crude distinction of low-integrity and high-integrity processes, only a few low-integrity processes need to perform a small number of security critical operations, which can be specified using a few simple policies as exceptions.

> **Basic HIPP Model:** Each process has one bit that denotes its integrity level. When a process is created, it inherits the integrity level of the parent process. When a process performs an operation that makes it potentially contaminated, it drops its integrity. A low-integrity process by default cannot perform sensitive operations.

In the rest of this section, we clarify and refine the above basic model. There are two main aspects: (1) when does a process drops integrity (i.e., is considered been contaminated), and (2) what are the restrictions on a low-integrity process. Section 3.2 discusses contamination through network and interprocess communications. Section 3.3 discusses restrictions on low-integrity processes. Section 3.4 discusses contamination through files. Section 3.5 discusses protecting files owned by non-system accounts. A summary of the HIPP model is given in Section 3.6.

3.2 Dealing with Network and Inter-Process Process Communications

When a process receives remote network traffic (network traffic that is not from the localhost loopback), its integrity should drop, as the program may contain vulnerabilities and the traffic may be sent by an attacker to exploit such vulnerabilities. Under this default policy, system maintenance tasks (e.g., installing new softwares, updating system files, and changing configuration files) can be performed only through a local terminal. Users can log in remotely, but cannot perform these sensitive tasks. While this offers a high degree of security, it may be too restrictive in many systems, e.g., in a colocated server hosting scenario. The HIPP model provides a way to enable remote system administration.

Remote Administration Points A program may be identified as a remote administration point. If one wants to allow remote system administration through, e.g., Secure Shell Daemon, then one can identify /usr/sbin/sshd as a remote administration point, which will make the process running /usr/sbin/sshd maintain its integrity level even if it receives network traffic. This enables a system administrator to login remotely and perform system maintenance tasks such as upgrading the system, adding/removing users, etc. (Note that if a process descending from sshd receives network traffic, its integrity drops.)

The concept of remote administration points is the result of trade-off between security with usability in favor of usability. Allowing remote administration certainly makes the system less secure. If remote administration through sshd is allowed, and the attacker can successfully exploit bugs in sshd, then the attacker can take over the system, as this is specified as a legitimate remote administration channel. However, note that in this case the attack surface is greatly reduced from all daemon programs, to only sshd. Some daemon programs (such as httpd) are much more complicated than sshd and are likely to contain more bugs. Moreover, firewalls can be used to limit the network addresses from which one can connect to a machine via sshd; whereas one often has to open the httpd server to the world. Finally, techniques such as privilege separation [5, 18] can be used to further mitigate attacks against sshd. The HIPP model leaves the decision of whether to allow remote administration through channels such as sshd to the system administrators.

Dealing with inter-process communications We also need to consider what happens when a process receives network traffic from another process on the same host through local loopback, and when a process receives Inter-Process Communications (IPC) from another local process.

HIPP considers integrity contamination through those IPC channels that can be used to send free-formed data, because such data can be crafted to exploit bugs in the receiving process. Under Linux, such channels include UNIX Domain Sockets, Pipes, FIFOs, Message queues, shared memory, and shared files in the TMPFS filesystem. When a process reads from one of these IPC channels which have been written by a low-integrity process, then the integrity of the process drops. Note that even when a process is a remote administration point, its integrity still drops by IPC contaminations.

Dealing with loopback network communication, however, turns out to require a design different from IPC. Dropping integrity only when receiving traffic from low-integrity processes is insufficient. As many systems would need to enable remote administration through sshd, the sshd process would often be a remote administration point and have high integrity. On the other hand, ssh tunneling is often used for forwarding traffic to daemon programs, such as ftp daemon and email servers. In this case, sshd is communicating with a daemon program through local loopback, and sshd has high integrity, even though the traffic sshd forwards comes from a remote host and is potentially generated by an attacker. We want a daemon's integrity to drop when it receives such traffic that is apparently from a local process but is actually from a remote host.

One may consider another design in which a process drops its integrity whenever it receives any network traffic (whether it is from local loopback or not). One problem with this design is that during booting, some processes will communicate with each other through local loopback (e.g., the booting script may communicate with an X server and with the rhgb program for some Linux distributions). Under this design, both processes will drop integrity to low, even though none of them has communicated with outside, and after booting the

system stays in a low-integrity state.

The key issue here is that when a process communicates with a high-integrity process P, we want to know whether P is "actually" high integrity, i.e., whether P has never communicated remotely, or P is a remote administration point that has communicated remotely (or the descendant of such a process). To do this, we introduce another bit called "NET" to represent the state of a process.

The 'NET' bit and Administration Tunneling Points When a process receives remote network traffic, then the "NET" bit is set. If a process communicates through loopback with a process whose "NET" bit is set, then the process sets its "NET" bit and drops integrity. Another issue with this design arises if we want to allow remote administration through X. The X Window System is based on a client-server architecture. The server controls the display and input devices, the clients are the graphical applications that access those services. Clients connect to the server via Unix domain sockets (if local) or TCP/IP (if remote). Remote X is typically achieved through X forwarding via telnet or SSH, as illustrated in Figure 1.

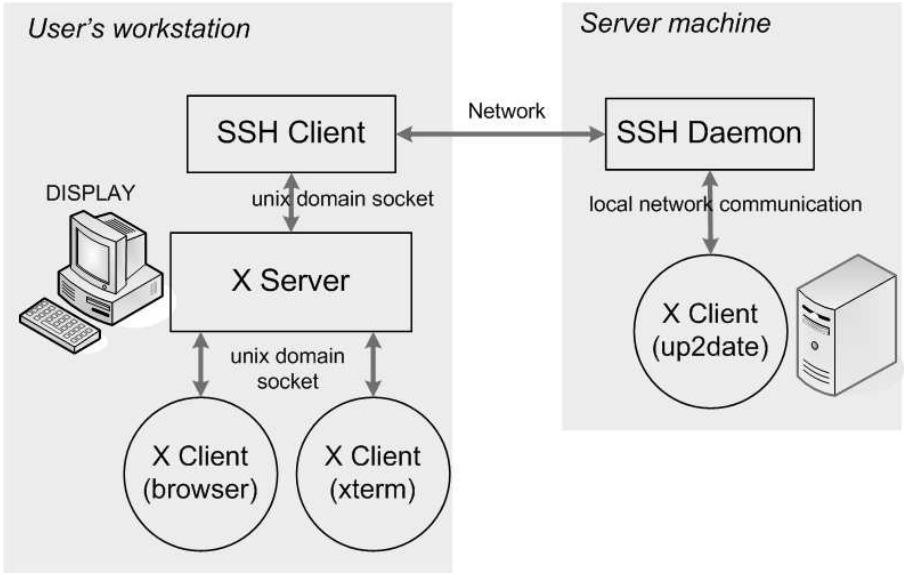

Figure 1: Remote Administration Using X Applications Through SSH Tunneling

Many administration tasks can be performed by X applications. Suppose that the user wants to administer the server machine remotely, then the X Client running on the server machine will connect to the SSH Daemon to receive instructions from the user, and the X Client needs to maintain high integrity to perform administration tasks. While remote administration through X applications may be considered a dangerous practice, we do not want to make the policy decision that this is *always* forbidden. Note that in this case, sshd is used to tunnel administrative traffic, and rather than connecting to the X application, it waits for the X application to connect to it. HIPP allows one to define a program as an Administration Tunneling Point (ATP). If a process connects to a high-integrity ATP through a TCP socket, then the process maintains its integrity, even if the ATP has performed network communication. Note that if a process accepts a connection from an ATP who has performed network communication, then the process drops its integrity. If one wants to allow remote administration through sshd, but not remote X administration, then one should identify sshd as a RAP, but not an ATP.

In summary, when dealing with local loopback network communications, we have made two design decisions that complicate the design, in order to provide better security and flexibility. These two designs are necessitated by the need to enable remote administration through the SSH daemon and by the multiple roles the SSH daemon can play. The first decision is to introduce the "NET" bit to handle ssh tunneling. The second decision is to introduce the concept of ATP to allow an administrator to decide whether to turn on or off remote

X administration through ssh tunneling.

3.3 Restrictions on low-integrity processes

Our approach requires the identification of security-critical operations that would affect system integrity so that our protection system can prevent low-integrity processes from accessing them. We classify security-critical operations into two categories, file operations and operations that are not associated with specific files.

Example non-file administrative operations include loading a kernel module, administration of IP firewall, modification of routing table, network interface configuration, rebooting the machine, ptrace other processes, mounting and unmounting file systems, and so on. These operations are essential for maintaining system integrity and availability, and are usually targeted by malicious code. In modern Linux, these operations are controlled by capabilities, which were introduced since version 2.1 of the Linux kernel. Capabilities break the privileges normally reserved for root down to smaller portions. As of Linux Kernel 2.6.11, Linux has 31 different capabilities. The default HIPP rule grants only two capabilities CAP_SETGID and CAP_SETUID to low-integrity processes; furthermore, low-integrity processes are restricted in that they can use setuid and setgid only in the following two ways: (1) swapping among effective, real, and saved uids and guids, and (2) going from the root account to another system account. (A system, with the exception of root, does not correspond to an actual human user and typically has an id below 500.) A low-integrity process running as root cannot set its uid to a new normal user. We will discuss why this offers useful protection in Section 3.5.

It is much more challenging to identify which files should be considered sensitive, as a large number of objects in an operating system are modeled as files. Different hosts may have different softwares installed, and have different sensitive files. The list of files that need to be protected is quite long, e.g., system programs and libraries, system configuration files, service program configuration files, system log files, kernel image files, and images of the memory (such as /dev/kmem and /dev/mem). We cannot ask the end users to label files as our goal is to have the system configurable by ordinary system administrators who are not security experts. Our novel approach here is to utilize the valuable information in existing Discretionary Access Control (DAC) mechanisms.

Using DAC info for NonDAC All commercial operating systems have built-in DAC mechanisms. For example, UNIX and UNIX variants use the permission bits to support DAC. While DAC by itself is insufficient for stopping network-based attacks, DAC access control information is nonetheless very important. For example, when one installs Linux from a distribution, files such as /etc/passwd and /etc/shadow would be owned by root and writable only by root. This indicates that writing to these files is security critical. And our policy prevents a low-integrity process from writing these files, even if the process has effective uid 0. Similarly, files such as /etc/shadow would be readable only by root, and our policy prevents a low-integrity process from reading these files. Such DAC information has been used by millions of users and examined for decades. Our approach utilizes this information, rather than asking the end users to label all files, which is a labor intensive and error-prone process. HIPP offers both read and write protection for files owned by system accounts. A low-integrity process is forbidden from reading a file that is owned by a system account and is not readable by world; such a file is said to be *read-protected*. A low-integrity process is also forbidden from writing to a file owned by a system account and is not writable by world. Such a file is said to be *write-protected*.

Exception policies: least privilege for sensitive operations Some network-facing daemons need to access resources that are sensitive. Because these processes receive network communications, they will be low-integrity, and the default policy will stop such access. We deal with this by allowing the specification of policy exceptions for system binaries. For example, one policy we use is that the binary "/usr/sbin/vsftpd" is allowed to use the capabilities CAP_NET_BIND_SERVICE, CAP_SYS_SETUID, CAP_SYS_SETGID, and CAP_SYS_CHROOT, to read the file /etc/shadow, to read all files under the directory /etc/vsftpd, and to read or write the file /var/log/xferlog. This daemon program needs to read /etc/shadow file to authenticate remote users.

If an attacker can exploit an vulnerability in vsftpd and inject code into the address space of vsftpd, this code can read /etc/shadow file. However, if the attacker injects shell code to obtain an shell by exploiting the vulnerabilities, then the exception policy for the shell process will be reset to NULL and the attacker loses the ability to read /etc/passwd. Furthermore, the attacker cannot write to any system binary or install rootkits. Under this policy, an administrator cannot directly upload files to replace system binaries. However, the administrator can upload files to another directory and login through a remote administration channel (e.g., through sshd) and then replace system binary files with the uploaded files.

When a high integrity process loads a program that has an exception policy, the process has special privileges as specified by the policy. Even when the process later receives network traffic and drops integrity, the special privileges remain for the process. However, when a low integrity process loads a program that has an exception policy, the process should be denied the special privileges in the policy. Some network administration tools (such as iptables) must perform network communications and will thus drop its integrity, so they need to be given capability exceptions for CAP_NET_ADMIN. However, we would not want a low-integrity process to invoke them and still have the special privileges. On the other hand, some programs need to invoke other programs to perform tasks. For example, sendmail needs to invoke procmail when its integrity is low, and procmail needs special privileges to do its job. We resolve this by defining loading relationships between programs. When two programs X and Y have a loading relationship, and a process running X loads Y, then even if the process is low integrity, the process will have special permissions associated with Y after loading.

3.4 Contamination propagation through files

As an attacker may be able to control contents in files that are not write-protected, a process's integrity level needs to drop after reading and executing files that are not write-protected. However, even if a file is write-protected, it may still be written by low-integrity processes, due to the existence of exception policies. We use one permission bit to track whether a file has been written by a low-integrity process. There are 12 permission bits for each file in a UNIX file system: 9 of them indicate read/write/execute permissions for user/group/world; the other three are setuid, setgid, and sticky bit. The sticky bit is no longer used for regular files (it is still useful for directories), and we use it to track contamination for files. When a low-integrity process writes to a file that is write-protected as allowed by an exception, the file's sticky bit is set. A file is considered to be low-integrity (potentially contaminated) when either it is not write-protected, or has the sticky bit set. When a process reads a low-integrity file, the process's integrity level drops. If a low-integrity process adds a file to a directory and a high-integrity process reads the directory, we do not consider the high-integrity process to be contaminated, as this process would read the directory through the file system, which should handle directory contents properly. When a file's permission is changed from world-writable to not world-writable, the sticky bit is set, as the file may have been contaminated while it was world-writable. A low-integrity process is forbidden from changing the sticky bit of a file. Only a high-integrity process can reset the sticky bit by running a special utility program provided by the protection system. The requirement of using a special utility program avoids the problem that other programs may accidentally reset the bit without the user intending to do it. This way, when a user clears the sticky bit, it is clear to the user that she is raising the integrity of the file. Similar to the concept to remote administration point, which allows a process to maintain integrity while receiving network traffic, we also need to introduce file processing programs (FPP). A process that is an FPP will maintain its integrity even after reading a low-integrity file. Programs that read a file's content and display the file on a terminal need to be declared to be FPP, e.g., vim, cat, etc.

The classes of files are given in Figure 2. As we can see, a file may be both write-protected and low-integrity. The preconditions and effects of a process accessing a file is given in the Figure 3.

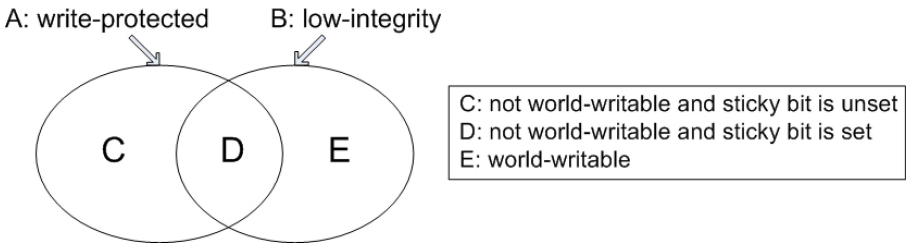

Figure 2: Classes of files in HIPP: $D = A \cap B$, $C = A \setminus B$, $E = B \setminus A$.

	Read(f)		Write(f)	
	Preconditions	Effects	Preconditions	Effects
High-integrity process P	None	if $f \in B$ then P drops to low	None	None
Low-integrity process P	f is world-readable OR P is given special privilege to read f	None	$f \notin A$, OR P is given special privilege to write f	if $f \in C$ then set f's sticky bit

Figure 3: The preconditions and effects of a process accessing a file. Refer to Figure 2 for the sets A, B, C.

3.5 Protecting files owned by non-system accounts

Not all sensitive files are owned by a system account. For example, consider a user account who has been given privileges to sudo (superuser do) all programs. The startup script files for the account are sensitive. We follow the approach of using DAC info in NonDAC. If a file is not writable by the world, then it is write-protected. HIPP allows exceptions to be specified for specific users. Different users may have different exception policies. An account's exception policy may specify global exceptions that apply to all programs running with that user's user-id. For example, a user may specify that a directory can be written by any low-integrity process and use the directory to store all files coming from the network.

If the system administrator does not want to enable integrity protection for a user, so that the user can use the system completely transparently (i.e., without knowing the existence of HIPP), then the policy can specify a global exception for the home directory of the user with recursion so that all low-integrity processes can access the user's files. We point out that even with such a global exception, HIPP still offers useful protection. First, the exceptions for a user will be activated only if the process's effective user id is that user. Recall that we disallow a low-integrity process from using setuid to change its user id to another user account. This way, if an attacker breaks in through one daemon program owned by account A, the attacker cannot write to files owned by account B, even if a global exception for B is in place. Second, if the user is attacked while using a network client program, and the users' files are contaminated. These files will be marked by the sticky bit, and other processes who happen to access them will drop integrity, so that the system integrity is still protected.

3.6 A summary of the HIPP Model

The HIPP model borrows concepts from classical work on integrity models such as Biba [4] and Clark and Wilson [7]. Compared with previous integrity protection models, HIPP has several novel features.

First, HIPP supports a number of partially trusted programs that can violate the default contamination rule. They are needed to ensure that existing applications and system administration practices can be used. They are summarized as follows.

- A process running a program that is identified as a RAP does not drop integrity when receiving traffic

from the network.

- A process running a program that is identified as an FFP does not drop integrity when reading a low-integrity file (i.e., a file whose DAC permission is world-writable or has the sticky bit set).
- A process connecting to a socket controlled by a high-integrity process P running a program that is identified as an ATP does not drop its integrity (even if P has 'NET' bit set).

Second, a file is protected as long as its DAC permission is such that it is not writable by the world. Even if a file has sticky bite set (i.e., considered to be low-integrity), a low-integrity process still cannot write to the file unless a policy exception exists. In other words, the set of write-protected files and and the set of low-integrity files intersect, as illustrated by Figure 2. Consider, for example, the system log files and the mail files. These files would have the sticky bit set because they have been written by processes who have communicated with the Internet. However, an attacker who broke into the system through, say, httpd, still cannot write to them, as the attacker's processes would be low integrity and do not have the policy exceptions. This is different from traditional integrity models such as Biba, where a low-integrity file can be written by any low-integrity subjects. Our design offers better protection.

Third, HIPP's integrity protection is compartmentized. Even if one user has an exception policy that allows all low-integrity processes to access certain files owned by the user, another user's low-integrity process is forbidden from such access, even if such access is allowed by DAC.

Fourth, HIPP allows low-integrity files to be upgraded to high-integrity. This means that low-integrity information (such as files downloaded through the Internet) can flow into high-integrity objects (such as system binaries); however, such upgrade must occur explicitly, i.e., by invoking a special program in a high-integrity channel to remove the sticky bit. Allowing such channels is necessary for, e.g., patching and system ungrade.

Fifth, HIPP uses DAC information to determine integrity and confidentiality labels for objects. We believe that this is one key to ease of deployment.

Finally, HIPP offers some confidentiality protection, in addition to integrity protection. For example, low-integrity processes are forbidden from reading files owned by a system account and not readable by the world.

4 An Implementation under Linux

We have implemented the HIPP model in a prototype protection system for Linux using the Linux Security Module (LSM) framework, and have been using evolving prototypes of the protection system within our group for a few months. We describe the implementation of the system in Section 4.1 and the evaluation in Section 4.2.

4.1 Implementation

The basic design of our protection system is as follows. Each process has a security label, which contains (among other fields) a field indicating whether the process is high integrity or low integrity. When a process issues a request, it is authorized only when both the Linux DAC system and our protection system authorize the request. A high-integrity process is not restricted by our protection system, and can perform any operation authorized by the existing DAC policies. A low-integrity process *by default* cannot perform any sensitive operation, even if it is running as root (i.e., with an effective uid 0). Any exception to the above default policy must be specified in a policy file, which is loaded when the module starts.

The Policy Specification The policy file includes a list of entries. Each entry contains four fields: (1) a path that points to the program that the entry is associated with, (2) the type of a program, which includes three bits indicating whether the program is a remote administration point (RAP), an administration tunneling point (ATP), and a file processing point (FPP) (3) a list of exceptions, and (4) a list of loading relationships, which

is a list of programs that can be loaded by the current program with the exception policies enabled, even if the process is low integrity. If a program is not listed, the default policy is that the program is not a RAP, an ATP, or a FPP, and the exception list and the loading relationship list are empty. An exception list consists of two parts, the capability exception list and the file exception list, corresponding to exceptions to the two categories of security critical operations. A file exception takes one of the following four forms.

Syntax		Meaning
(f, read)	f is a regular file or a directory	Allowed to read f
(f, full)	f is a regular file or a directory	Allowed to do anything to f
(d, read, R)	d is a directory	Allowed to read any file in d recursively.
(d, full, R)	d is a directory.	Allowed to do anything to any file in d recursively.

The authorization provided by file exceptions includes only two levels: read and full. We choose this design because of its simplicity. In this design, one cannot specify that a program can write a file, but not read. We believe this is acceptable because system-related files that are read-sensitive are also write-sensitive. In other words, if the attacker can write to a file, then he can pose at least comparable damage to the system as he can also read the file. A policy of the form "(d, read, R)" is used in the situation that a daemon or client program needs to read the configuration files in the directory d. A policy of the form "(d, full, R)" is used to define the working directories for programs. See Appendix B for more details about file protection.

Process State Changes See Figure 4 for a specification of what information the label in a process contains and how the label changes with events in the system.

4.2 Evaluation

We evaluate our design of the HIPP model and the implementation under Linux along the following dimensions: usability, security, and performance.

Usability One usability measure is transparency, which means not stopping legitimate accesses generated by normal system operations. Another measure is flexibility, which means that one can configure a system according to the security needs. A third usability measure is ease of configuration. Several features of HIPP contribute to a high level of usability: the use of existing DAC information, the existence of RAP, ATP, and FFP, and the use of familiar abstractions in the specification of policies. To experimentally evaluate the transparency and flexibility aspects, we established a server configured with Fedora Core 5 with kernel version 2.6.15, and enabled our protection system as a security module loaded during system boot. We installed some commonly used server applications (e.g., httpd, ftpd, samba, svn) and have been providing services to our research group over the last few months. The system works with a small and easily-understood policy specification (given in Appendix C. With this policy, we allow remote administration through SSH Daemon by declaring sshd as RAP, and remote administration through X Tunnelling over SSH by declaring sshd as ATP. If one also want to alow remote administration through VNC Tunnelling over SSH, then he can achieve that by declaring the VNC Server as ATP.

Security Most attack scenarios that exploit bugs in network-facing daemon programs or client programs can be readily prevented by our protection system. Successful exploitation of vulnerabilities in network-facing processes often results in a shell process spawned from the vulnerable process. After gaining access, the attacker typically try downloading and installing attacking tools and rootkits. As these processes are low-integrity, these access to sensitive operations is limited to those allowed by the exception. Furthermore, if the attacker loads a shell or any other program, the new process has no exception privileges. We conduct three types of attacks after the attacker successfully breaks in through network programs. They are installing RootKits, stealing shadow file and altering user's web page files. All attacks failed in a system protected by our module. See Appendix D for a detailed description of the test against attacks. Network client programs are under similar protections.

(a) File Policies. The policy file contains the following field for each entry:

Name	Values	Description
PATH	a file path	path of the binary file this entry is associated with
TYPE	[000..111]	three bits denoting whether this program is a RAP, a ATP, and/or a FPP
EL	a list of operations	a list of security operations that this program is authorized to perform even when IL (integrity level) is low
LR	a list of file paths	a list of programs that can be started with exception policies enabled by a low-integrity process running the current program

(b) Process States. The label of a process consists of the following fields:

Name	Values	Description
IL	'high' or 'low'	integrity level; when 'low', the process's access is restricted
NET	'yes' or 'no'	whether has network communication before; when 'yes', may drop other processes' integrity via loopback network
BIN	a file path	path of the binary currently executing
TYPE	[000..111]	three bits denoting whether this program is a RAP, a ATP, and/or a FPP
EL	a pointer to a special data structure	a list of security operations that can be performed even when IL is low
LR	a list of file paths	a list of programs that can be loaded with exception policies enabled by a low-integrity process running the current program

(c) Process state change rules

Operation	Effect
creation by fork	copy label from parent process

load file NBIN by exec	if (NBIN is low-integrity) IL:='low'; EL:=NULL; LR:=NULL;
	else if (IL='high' OR NBIN in BIN.LR) EL:=NBIN.EL; LR:=NBIN.LR;
	else EL:=NULL; LR:=NULL endif
	BIN:=NBIN; TYPE:=NBIN.TYPE;

receive any remote communication	net_taint(); net_taint() { NET:='yes'; if (nonRAP(TYPE)) IL:='low'; endif }

connect to a loopback socket listened by process X	if (X.INT='low') net_taint();
	else if (X.NET='yes') NET='yes'; if (nonATP(X.TYPE)) net_taint(); endif
	endif

conduct other loopback network comm with process X	if (X.NET='yes') net_taint(); endif
conduct IPC with process X	if (X.INT='low') INT:='low'; endif if (X.NET='yes') NET:='yes'; endif
read a regular file F	if (nonFFP(TYPE) AND (F is low-integrity)) INT='low'; endif

Figure 4: The HIPP Model implemented under Linux.

Finally, any program coming from the network will have low-integrity. Before the user explicitly upgrade its integrity, the damage caused by such programs are limited.

Performance We have conducted benchmarking tests to compare performance overhead incurred by our protection system. For most benchmark results, the percentage overhead is small ($\leq 5\%$). The performance of our module is significantly better than the data for SELinux as reported in [12]. See Appendix E for a detailed description of the experiment setting and evaluation results.

5 Related Work

Existing systems that are closely related to ours include SELinux [16], systrace [17], LIDS [11], securelevel [9], LOMAC [8], and AppArmor [1]. As we discussed in Section 1, SELinux, systrace, and LIDS, while flexible and powerful, require extensive expertise to configure. These systems focus on mechanisms, whereas our approach focuses on providing a policy model that achieve a high degree of protection without getting in the way of normal operations. Both systrace and LIDS require intimate familiarity with UNIX system calls and internals for configuration. SELinux adopts the approach that MAC information is independent from DAC. For example, the users in SELinux are unrelated with the users in DAC, each file needs to be given a label. This requires the file system to support additional labeling, and limits the applicability of the approach. Furthermore, labeling files is a labor-intensive and error-prone process. Each installation of a new software requires update to the policy to assign appropriate labels to the newly added files and possibly add new domains and types. SELinux policies are difficult to understand by human administrators because of the size of the policy and the many levels of indirection used, e.g., from programs to domains, then to types, and then to files. Our protection system, on the other hand, utilizes existing valuable DAC information, requires much less configuration, and has policies that are easy to understand.

AppArmor [1] is a Linux protection system that has similarities with our work. It confines applications by creating security profiles for programs. A security profile identifies all capabilities and files a program is allowed to access. It also uses file paths to identify programs and in the security profiles, which is similar to our approach. AppArmor uses the same approach as the Targeted Policy in Fedora Core Linux, i.e., if a program has no policy associated with it, then it is by default not confined, and if a program has a policy, then it can access only the objects specified in the policy. This approach violates the fail-safe defaults principle, as a program with no policy will by default run unconfined. By not confining high-integrity processes and allowing low-integrity processes to access unprotected files, HIPP can afford to follow the fail-safe default principle and only specify exceptions for programs. AppArmor does not maintain integrity levels for processes or files, and thus cannot differentiate whether a process or a file is contaminated or not. For example, without tracking contamination, one cannot specify a policy that system administration through X clients are allowed as long as as the X server and other X clients have not communicated with the network. Also, AppArmor cannot protect users from accidentally downloading and executing malicious programs.

Securelevel [9] is a security mechanism in *BSD kernels. When the securelevel is positive, the kernel restricts certain tasks; not even the superuser (i.e., root) is allowed to do them. Any superuser process can raise securelevel, but only init process can lower it. The weakness of securelevel is clearly explained in the FreeBSD FAQ [9]: "*One of its biggest problems is that in order for it to be at all effective, all files used in the boot process up until the securelevel is set must be protected. If an attacker can get the system to execute their code prior to the securelevel being set [...], its protections are invalidated. While this task of protecting all files used in the boot process is not technically impossible, if it is achieved, system maintenance will become a nightmare since one would have to take the system down, at least to single-user mode, to modify a configuration file.*" HIPP enables system administration through high-integrity channels, thereby avoiding the difficulty securelevel has. HIPP also has file contamination rules to ensure that all files read during booting are high integrity for the system to end up in a high-integrity state.

LOMAC [8] also aims at protecting system integrity and places emphasis on usability. HIPP has a number of differences from LOMAC. First, each LOMAC installation requires the specification of a mapping between existing files and integrity levels; whereas we use DAC information to determine this. Second, LOMAC does not protect confidentiality, whereas HIPP forbids low-integrity process to read read-protected files. Third, in the LOMAC model, once an object is created, its integrity level never changes. This, however, prevents system updates. Fourth, in the LOMAC model, remote administration can be performed only through a network interface that is identified as a remote management link. For a system with a single network interface, either one forbids remote administration, or one has to assign the network interface a high integrity level, which means that all network-facing daemons are high integrity. See Section 3.6 for additional features of HIPP.

6 Conclusions

We have identified six design principles for designing usable access control mechanisms. We have also introduced the HIPP model, a simple, practical nonDAC policy model for host protection, designed using these principles. The HIPP model defends against attacks targeting network server and client programs and protects users from careless mistakes. It supports existing applications and system administration practices, and has a simple policy configuration interface. HIPP also introduced several novel features in integrity protection protection. We have also reported the experiences and evaluation results of our implementation of HIPP under Linux. We plan to continue testing and improving the code and release it to the open-source community in near future. We also plan to develop tools that help system administrators analyze a HIPP configuration and identify channels through which an attacker may get a high-integrity process (e.g., by exploiting a remote administration point).

References

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

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *Proc. USENIX Security Symposium*, June 1995.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Proc. IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.

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

[5] D. Brumley and D. Song. PrivTrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, August 2004.

[6] H. Chen, D. Dean, and D. Wagner. Setuid demystified. In *Proc. USENIX Security Symposium*, pages 171–190, Aug. 2002.

[7] D. D. Clark and D. 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.

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

[9] *Frequently Asked Questions for FreeBSD 4.X, 5.X, and 6.X*. http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proc. USENIX Security Symposium*, pages 1–13, June 1996.

[11] LIDS: Linux intrusion detection system. *http://www.lids.org/*.

[12] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX track: USENIX Annual Technical Conference*, pages 29–42, June 2001.

[13] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example: Using Security Enhanced Linux*. Prentice Hall PTR, 2006.

[14] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., October 2004.

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

[16] Security-enhanced Linux. *http://www.nsa.gov/selinux*.

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

[18] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 2003 USENIX Security Symposium*, pages 231–242, August 2003.

[19] E. S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 2003.

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

[21] R. Sandhu. Good-enough security: Toward a pragmatic business-driven discipline. *IEEE Internet Computing*, 7(1):66–68, Jan. 2003.

[22] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proc. USENIX Security Symposium*, pages 17–31, 2002.

A The Pitfalls of Security Mechanism without Clear Policy Objective: A Case Study using setuid

One example security mechanism that has suffered from the lack of security objective is $setuid$. Chen et al. [6]'s "Setuid demystified" provided an excellent history and analysis of the problems associated with $setuid$. In early UNIX, a process has two user IDs, the *real uid* and the *effective uid*. Only one system call ($setuid$) manipulates them. This approach has the limitation that a process cannot drop its privilege temporarily and restore it later. To address this problem, System V added a new user ID called *saved uid* to each process, and a new system call $seteuid$. At the same time, BSD 4.2 continued to use read uid and effective uid, but changed the system call from $setuid$ to $setreuid$. As System V and BSD influence each other, they and other modern UNIX variants support three user ids and implemented all three system calls: $setuid$, $seteuid$, $setreuid$, although different variants often implement then with slightly different semantics. Some modern UNIX system introduced a new call $setresuid$. Chen et al. [6] discussed the operating system specific differences and found several bugs by

Abstracted file operations	When to allow when it is done by P with low-integrity	Effect [a]
read(f1)	f1 is not read-protected; OR P has a read exception or a full exception over f1.	
write(f1)	f1 is not write-protected; OR P has a full exception over f1.	set f1 as low-integrity
create(f1)	P is able to write(f1's parent directory); OR P has full exception over f1.	set f1 as low-integrity
unlink(f1) [b]	P is able to write(f1) and write(f1's parent directory)	
create_link(f1, f2) [c]	P is able to read(f1), write(f1) and create(f2)	set f1 as low-integrity
rename(f1, f2) [d]	P is able to create_link(f1, f2) and unlink(f1)	set f1 as low-integrity
set_permissions(f1) [e]	f1 is unprotected and P change it to be unprotected; OR P has a full exception over f1.	set f1 as low-integrity [f]

Figure 5: The file system protection implemented under Linux.

[a] Only when the object is a regular file and the subject process is low-integrity
[b] Remove a hard/soft link f1. Removing a hard link could remove the actual inode, if f1 is the one and only hard link to the inode.
[c] Create a hard/soft link f2 to f1
[d] Rename the file f1 to f2
[e] Change the owner, group and other permission bits of f1
[f] Also apply when a high-integrity process change a file from not write-protected to write-protected.

reading the source code and using finite state machines to analyze. Chen et al. [6] suggested an improved API, which includes three calls drop_priv_temp, drop_priv_perm, and restore_priv and found that all uid-setting system calls in OpenSSH 2.5.2 can be replaced by one of the three calls. These API calls return to the original security objectives that motivated the security mechanisms. System V and BSD chose two different mechanisms to implement the same policy objective. Then the mechanism evolved (and got more and more complicated) independently of the initial security objective. As a result, the mechanism is very difficult to use.

B Details about File System Protection Design

We treat a file [4] f as read-protected if f is owned by a system account and is not readable by the world. We treat a file f as write-protected if f is not writable by the world. We treat a file f as low-integrity if the sticky bit of f is set or if f is world-writable. We abstract six file operations that are considered as security critical. Figure 5 shows a specification of the conditions for a low-integrity process to perform those operations and the effect of the operations.

Figure 6 shows the mapping from security hooks in LSM to the abstracted file operations given in Figure 5.

C Sample Policies

Sample policies for our experimental machine is given in Figure 7.

D Testing against Attacks

Experiment Settings In our experiments, we use the NetCat tool to offer an interactive root shell to the attacker in the experiment. We execute NetCat in "listen" mode on target machine as root. When the attacker

[4] Here we use a file to denote a regular file, a directory, or a device file.

Security hooks for inode / file operations	Abstracted file operations
inode_create	create(f1)
inode_link	create_link(f1, f2)
inode_unline	unlink(f1)
inode_symlink	create_link(f1, f2)
inode_mkdir	create(f1)
inode_rmdir	unlink(f1)
inode_rename	rename(f1, f2)
inode_permission	read(f1) or write(f1)
inode_setattr	set_permissions(f1)
inode_delete	unlink(f1)
file_permission	read(f1) or write(f1)
file_mmap	read(f1) or write(f1)
file_mprotect	read(f1) or write(f1)
file_fcntl	write(f1)

Figure 6: Mapping from security hooks in LSM to the abstracted operation described in Section 4.1.

connects to the listening port, NetCat spawns a shell process, which takes input from the attacker and also directs output to him. From the root shell, we perform the following three attacks and analyze what happens in the case without our protection system and the case with our protection system.

Installing a RootKit RootKits can operate at two different levels. User-mode RootKits manipulate user-level operating system elements, altering existing binary executables or libraries. Kernel-mode RootKits manipulate the kernel of the operating system by loading a kernel module or manipulate the image of the running kernel's memory in the file system (/dev/kmem).

In terms of examining whether the system has been compromised after installing a RootKit, we apply two methods. The first one is just to try to use the RootKit and see whether it is successfully installed. The second one is to calculate the MD5 hash function for all the files (content, permission bits, last modified time) in the local file system before and after installing the RootKit. Each time of the calculation we reboot the machine by using an external operating system (e.g., from a CD) and mount the local file system. In this way we can ensure that the running kernel, the libraries and programs used in the calculation are clean. A comparison between the MD5 results can tell whether the system has been compromised. All changes are examined to see whether they are legislate or not. Such a check can ensure that the system is clean after a reboot.

We tried the following two typical RootKits.

1. Adore-ng. It is a kernel-mode RootKit that runs on Linux Kernel 2.2 / 2.4 / 2.6. It is installed by loading a malicious kernel module. The supported features include local root access, file hiding, process hiding, socket hiding, syslog filtering and so on. Adore-ng provides also has a feature to replace an existing kernel module that is loaded during boot with the trojaned module, so that adore-ng is activated during boot.

   When our protection is not enabled, we can successfully install Adore-ng in the remote root shell and activate it. The system is being controlled by the attacker. We can also successfully replace any existing kernel module with the trojaned module so that the RootKit module will be automatically loaded during boot.

   When our protection system is enabled, the request to load the kernel module of Adore-ng from the

Services and Path of the Binary	Type	File Exceptions	Capability Exceptions	Executing Relationships
SSH Daemon /usr/sbin/sshd	RAP ATP			
Automated Update: /usr/bin/yum	RAP			
/usr/bin/vim	FPP			
/usr/bin/cat	FPP			
FTP Server /usr/sbin/vsftpd	NONE	(/var/log/xferlog, full) (/etc/vsftpd, full, R) (/etc/shadow, read)	CAP_SYS_CHROOT CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE	
Web Server /usr/sbin/httpd	NONE	(/var/log/httpd, full, R) (/etc/pki/tls, read, R) (/var/run/httpd.pid, full)		
Samba Server /usr/sbin/smbd	NONE	(/var/cache/samba, full, R) (/etc/samba, full, R) (/var/log/samba, full, R) (/var/run/smbd.pid, full)	CAP_SYS_RESOURCE CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE CAP_DAC_OVERRIDE	
NetBIOS name server /usr/sbin/nmbd	NONE	(/var/log/samba, full, R) (/var/cache/samba, full, R)		
Version control server /usr/bin/svnserve	NONE	(/usr/local/svn, full, R)		
Name Server for NT /usr/sbin/winbindd	NONE	(/var/cache/samba, full, R) (/var/log/samba, full, R) (/etc/samba/secrets.tdb, full)		
SMTP Server /usr/sbin/sendmail	NONE	(/var/spool/mqueue, full, R) (/var/spool/clientmqueue, full, R) (/var/spool/mail, full, R) (/etc/mail, full, R) (/etc/aliases.db, read) (/var/log/mail, full, R) (/var/run/sendmail.pid, full)	CAP_NET_BIND_SERVICE	/usr/sbin/procmail
Mail Processor /usr/bin/procmail	NONE	(/var/spool/mail, full, R)		
NTP Daemon /usr/sbin/ntpd	NONE	(/var/lib/ntp, full, R) (/etc/ntp/keys, read)	CAP_SYS_TIME	
Printing Daemon /usr/sbin/cupsd	NONE	(/etc/cups/certs, full, R) (/var/log/cups, full, R) (/var/cache/cups, full, R) (/var/run/cups/certs, full R)	CAP_NET_BIND_SERVICE CAP_DAC_OVERRIDE	
System Log Daemon /usr/sbin/syslogd	NONE	(/var/log, full, R)		
NSF RPC Service /sbin/rpc.statd	NONE	(/var/lib/nfs/statd, full, R)		
IP Table /sbin/iptables	NONE		CAP_NET_ADMIN CAP_NET_RAW	

Figure 7: Sample policy

remote root shell will be denied, getting an "Operation not permitted" error. We get the same error when we try to replace the existing kernel module with the trojaned module. When we try to use the RootKit, we get a response saying "Adore-ng not installed". We check the system integrity using the methods described above. The result show that the system remains clean.

2. Linux RootKit Family (LRK). It is one of the most well-known user-mode RootKits for Linux. It will replace a variety of existing system programs and introduce some new programs, to build a backdoor, to hide the attacker, and to provide other attacking tools. In this experiment, we only test the backdoor feature in LRK5.

   When our protection is not enabled, we can successfully install the trojaned SSH daemon and replace the existing SSH daemon in the system. After that we can connect to the machine as root by using a predefined password.

   When our protection is enabled, we are failed to install the trojaned SSH daemon to replace the existing SSH daemon, getting "Operation not permitted" errors. The backdoor is not opened. The system remains clean when we apply the checking process.

Stealing the shadow File Without our protection system, the attacker can steal /etc/shadow file by send an email with the shadow file as an attachment, e.g., by the command "mutt -a /etc/shadow alice@haker.net < /dev/null". When our protection is enabled, the request to read the shadow file will be denied, getting an error saying "/etc/shadow: unable to attach file" .

Altering user's web page files Another attack that usually happens is to alter the user's web page files after getting into a web server. In our experiment, we put the user's web page files in a sub directory of the user's home directory "/home/Alice/www/". That directory and all the web page files under the directory are set as not writable by the world. When our protection is enabled, from the remote root shell, the attacker cannot modify any web page files in the directory "/home/Alice/www/". The attacker cannot create a new file in that directory. Our module successfully protect user's protected files from being changed by attackers who break in through network.

E Performance Evaluation

Our performance evaluation uses the Lmbench 3 benchmark and the Unixbench 4.1 benchmark suites. These microbenchmark tests were used to determine the performance overhead incurred by the protection system for various process, file, and socket low-level operations. The low level of these tests provides the transparency and precision required in order to make informed conclusions regarding performance.

We established a PC configured with RedHat Linux Fedora Core 5, running on Intel Pentium M processor with 1400Hz, and having 120 GB hard drive and 1GB memory. Each test was performed with two different kernel configurations. The base kernel configuration corresponds to an unmodified Linux 2.6.11 kernel. The enforcing configuration corresponds to a Linux 2.6.11 kernel patched with our protection system.

The test results are given in Figure 8 and Figure 9. We compared our performance result with SELinux. The performance data of SELinux is taken from [12].

Benchmark	Base	Enforcing	Overhead (%)	SELinux(%)
Dhrystone	335.8	334.2	0.5	
Double-Precision	211.9	211.6	0.1	
Execl Throughput	616.6	608.3	1	5
File Copy 1K	474.0	454.2	4	5
File Copy 256B	364.0	344.1	5	10
File Copy 4K	507.5	490.4	3	2
Pipe Throughput	272.6	269.6	1	16
Process Creation	816.9	801.2	2	2
Shell Scripts	648.3	631.2	0.7	4
System Call	217.9	217.4	0.2	
Overall	446.6	435.0	3	

Figure 8: The performance results of Unixbench 4.1 measurements.

Microbenchmark	Base	Enforcing	Overhead (%)	SELinux(%)
syscall	0.6492	0.6492	0	
read	0.8483	1.0017	18	
write	0.7726	0.8981	16	
stat	2.8257	2.8682	1.5	28
fstat	1.0139	1.0182	0.4	
open/close	3.7906	4.0608	7	27
select on 500 fd's	21.7686	21.8458	0.3	
select on 500 tcp fd's	37.8027	37.9795	0.5	
signal handler installation	1.2346	1.2346	0	
signal handler overhead	2.3954	2.4079	0.5	
protection fault	0.3994	0.3872	-3	
pipe latency	6.4345	6.2065	-3	12
pipe bandwidth	1310.19 MB/sec	1292.54 MB/sec	7	
AF_UNIX sock stream latency	8.2	8.9418	9	19
AF_UNIX sock stream bandwidth	1472.10 MB/sec	1457.57 MB/sec	9	
fork+exit	116.5581	120.3478	3	1
fork+execve	484.3333	500.1818	3	3
for+/bin/sh-c	1413.25	1444.25	2	10
file write bandwidth	16997 KB/sec	16854 KB/sec	0.8	
pagefault	1.3288	1.3502	2	
UDP latency	14.4036	14.6798	2	15
TCP latency	17.1356	18.3555	7	9
RPC/udp latency	24.6433	24.8790	1	18
RPC/tcp latency	29.7117	32.4626	9	9
TCP/IP connection cost	64.5465	64.8352	1	9

Figure 9: The performance results of lmbench 3 measurements (in microseconds).