# Usable Mandatory Integrity Protection for Operating Systems

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

## Abstract

*Existing mandatory access control systems for operating systems are difficult to use. We identify several principles for designing usable access control systems and introduce the Usable Mandatory Integrity Protection (UMIP) model that adds usable mandatory access control to operating systems. The UMIP model is designed to preserve system integrity in the face of network-based attacks. The usability goals for UMIP are twofold. First, configuring a UMIP system should not be more difficult than installing and configuring an operating system. Second, existing applications and common usage practices can still be used under UMIP. UMIP has several novel features to achieve these goals. For example, it introduces several concepts for expressing partial trust in programs. Furthermore, it leverages information in the existing discretionary access control mechanism to derive file labels for mandatory integrity protection. We also discuss our implementation of the UMIP model for Linux using the Linux Security Modules framework, and show that it is simple to configure, has low overhead, and effectively defends against a number of network-based attacks.*

## 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, and distributed denial of service. 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 address 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 defending against network-based attacks.

There are a lot of research efforts on making computer systems more secure by adding mandatory access control (MAC)[1] to operating systems, e.g., Janus [12], DTE Unix [3, 2], Linux Intrusion Detection System (LIDS) [13], LOMAC [10], systrace [17], AppArmor [8, 1], and Security Enhanced Linux (SELinux) [16]. Several of these systems are flexible and powerful. Through proper configuration, they could result in highly-secure systems. However, they are also complex and intimidating to configure. For 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 a more general setting is unclear.

In this paper, we tackle the problem of designing and implementing a usable MAC system to protect end hosts. We start by identifying several principles for designing usable access control mechanisms in general. We then introduce the Usable Mandatory Integrity Protection (UMIP) model, which was designed following these principles.

The security goal of the UMIP model is to preserve system integrity in the face of network-based attacks. We assume that programs contain bugs and can be exploited if the attacker is able to feed malicious inputs 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. UMIP aims at narrowing the channels through which a network-

[1]In this paper, we use MAC to refer to the approach where a system-wide security policy restricts the access rights of processes. This is a wider interpretation of MAC than that in the TCSEC [9], which focuses on multi-level security.

based attacker can take over a host. The usability goals for UMIP are twofold. First, configuring a UMIP should not be more difficult than installing and configuring an operating system. Second, existing application programs and common practices for using and administering the system can still be used under UMIP.

The basic UMIP policy is as follows: Each process has an integrity level, which is either high or low. 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. Such operations include communicating with the network, receiving data from a low-integrity process through an interprocess communication channel, and reading or executing a file that is potentially contaminated. A low-integrity process by default cannot perform sensitive operations.

One novel feature of UMIP is that, unlike previous MAC systems, UMIP uses existing DAC information to identify which files are to be protected. In UMIP, a file is write-protected if its DAC permission is not world-writable, and a file is read-protected if it is owned by a system account (e.g., root, bin, etc.) and is not world-readable. A low-integrity process (even if running as root) by default is forbidden from writing any write-protected file, reading any read-protected file, or changing the DAC permission of any (read- or write-) protected file.

While the basic UMIP policy achieves the security goal, many existing applications will not be able to run and many common practices for using and administering the system will become impossible. We thus need to extend the basic UMIP policy to balance the functional requirements, the security goal, and the simplicity of the design (for usability). UMIP introduces several concepts to model programs that are partially trusted; these programs can violate the default integrity policy in certain limited, well-defined ways. For example, a program can be declared to be a remote administration point, so that a process running the program does not drop integrity upon receiving network traffic. This enables remote system administration through, e.g., the ssh daemon. Such a program is only partially trusted in that a process running it still drops integrity when reading a low-integrity file or receiving data from another low-integrity process. For another example, exception policies can be specified for some programs so that even when they are running in low-integrity processes, they can access some protected resources.

We have implemented UMIP for Linux using the Linux Security Modules (LSM) framework [24], and have been using evolving prototypes of the UMIP system within our group for a few months. We have found that only a small number of exceptions and settings need to be specified for our environment.

The contributions of this paper are three-fold.

1. We identify several design principles for designing usable access control mechanisms. Not all of these 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 be useful for other researchers and developers designing and implementing usable access control systems.
2. We introduce the UMIP model, a simple, practical MAC model that preserves system integrity in the face of network-based attacks. It has several novel features compared with existing integrity protection models.
3. We report our design and implementation of UMIP under Linux, and show that it is simple to configure, has low overhead, and effectively defends against a number of network-based attacks.

The rest of this paper is organized as follows. We discuss design principles in Section 2. The UMIP model is described in Section 3. Our implementation of UMIP 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

While it is widely agreed that usability is very important for security technologies, how to design an access control system that has a high level of usability has not been explored much in the literature. In this section we present six principles for designing usable access control systems. Some of these principles challenge established common wisdom in the field, because we place an unusually high premium on usability. These principles will be illustrated by our design of UMIP in Section 3.

**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 that 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

[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.*"

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, one needs 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.

One corollary following 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. 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.

This mechanism-not-policy philosophy is implicitly used in the design of many MAC systems for operating systems. 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 enough.

**Principle 3** *Have a well-defined security objective.*

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? A security objective should identify two things: what kind of adversaries the system is designed to protect against, i.e., what abilities does one assume the adversaries have, and what security properties one wants to achieve even in the presence of such adversaries. Often times, MAC systems do not clearly identify the security objective. For example, achieving multi-level security is often identified together with defending against network attacks. They are very different kinds of security objectives. History has taught us that designing usable multi-level secure systems is extremely difficult, and it seems unlikely that one can build a usable access control system that can achieve both objectives.

**Principle 4** *Carefully design ways to support exceptions in the policy model.*

Given the complexity of modern operating systems and the diverse scenarios in which computers are used, no simple policy model can capture all accesses that need to be allowed, and, at the same time, forbid all illegal accesses. It is thus necessary to have ways to specify exceptions in the policy model. The challenges lie in designing the policy model and the exception mechanisms so that the number of exceptions is small, the exceptions are easy and intuitive to specify, the exceptions provide the desired flexibility, and the attack surface exposed by the exceptions is limited. Little research has focused on studying how to support exceptions in an MAC model. As we will see, much effort in designing UMIP goes to designing mechanisms to support exceptions.

**Principle 5** *Rather than trying to achieve "strict least privilege", aim for "good-enough least privilege".*

It is widely recognized that one problem with existing DAC 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 a program needs to perform any of these operations, it needs to be given the root privilege. As a result, an attacker can exploit vulnerabilities in the program and abuse these privileges. Many propose to remedy the problem by using very-fine-grained access control and to achieve 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 a large number of policy rules. We believe that it is sufficient to restrict privileges just enough to achieve the security objective; and this enables one to design more usable access control systems. This principle can be viewed as a corollary of Principle 1. 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 identified 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 UMIP Model

We now introduce the Usable Mandatory Integrity Protection (UMIP) model, which was guided by the principles identified in Section 2. While the description of the UMIP 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 UMIP or a similar model for Microsoft Windows is beyond the scope of this paper.

We now identify the security objective of our policy model. We aim at protecting the system integrity 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 only 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 UMIP are twofold: First, configuring a UMIP system should not be more difficult than installing and configuring an operating system. Second, existing applications and common usage practices can still be used under UMIP. 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.

One constraint that we have for UMIP is that it can be implemented using an existing mechanism (namely the Linux Security Modules framework).

### 3.1 An overview of the UMIP model

An important design question for any operating system 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 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,

[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. Using LKMs is one popular method for implementing kernel-mode rootkits on Linux.

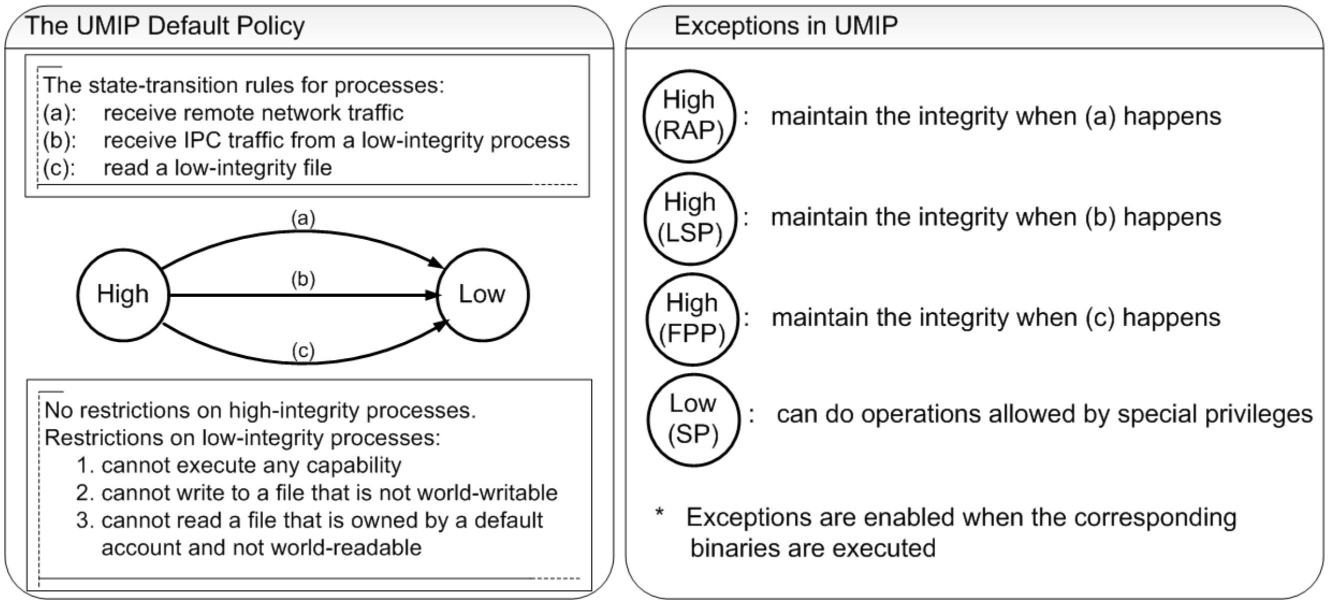
The UMIP Default Policy
The state-transition rules for processes:
(a): receive remote network traffic
(b): receive IPC traffic from a low-integrity process
(c): read a low-integrity file
High
Low
(a)
(b)
(c)
No restrictions on high-integrity processes.
Restrictions on low-integrity processes:
1. cannot execute any capability
2. cannot write to a file that is not world-writable
3. cannot read a file that is owned by a default account and not world-readable
Exceptions in UMIP
High (RAP) : maintain the integrity when (a) happens
High (LSP) : maintain the integrity when (b) happens
High (FPP) : maintain the integrity when (c) happens
Low (SP) : can do operations allowed by special privileges
* Exceptions are enabled when the corresponding binaries are executed

**Figure 1. The summary of the UMIP model**

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, then 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 UMIP 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.

The basic UMIP model is then extended with exceptions to support existing softwares and system usage practices. Figure 1 gives an overview of UMIP. A high-integrity process may drop its integrity to low in one of three ways. There are two classes of exceptions that can be specified for programs. The first class allows a program binary to be identified as one or more of: RAP (Remote Administration Point), LSP (Local Service Point), and FPP (File Processing Program). Such exceptions allow a process running the binary to maintain its integrity level when certain events that normally would drop the process's integrity occur. In the second class, a program binary can be given special privileges (e.g., using some capabilities, reading/writing certain protected files) so that a process running the program can have these privileges even in low integrity.

In the rest of this section, we describe the UMIP model in detail. 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. In Section 3.6 we discuss how the design principles in Section 2 are applied in designing UMIP. Comparison of UMIP with closely related integrity models is given in Section 3.7.

### 3.2 Dealing with communications

When a process receives remote network traffic (network traffic that is not from the localhost loopback), its integrity level 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 collocated server hosting scenario.

In the UMIP model, a program may be identified as a *remote administration point (RAP)*. The effect is that a process running the program maintains its integrity level when receiving network traffic. If one wants to allow remote system administration through, e.g., the secure shell daemon, then one can identify /usr/sbin/sshd as a remote administration point. (Note that if a process descending from sshd runs a program other than sshd and receives network traffic, its integrity level drops.) Introducing RAP is the result of trading off security 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 UMIP model leaves the decision of whether to allow remote administration through channels such as sshd to the system administrators.

We also need to consider what happens when a process receives Inter-Process Communications (IPC) from another local process. UMIP 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 socket, pipe, fifo, message queue, shared memory, and shared file in the tmpfs filesystem. In addition, UMIP treats local loopback network communication as a form of IPC. When a process reads from one of these IPC channels which have been written by a low-integrity process, then the integrity level of the process drops, even when the process is a RAP.

Similar to the concept of RAP, a program may be identified as a Local Service Point (LSP), which enables a process running the program to maintain its integrity level after receiving IPC communications from low-integrity processes. For example, if one wants to enable system administration and networking activities (such as web browsing) to happen in one X Window environment, the X server and the desktop manager can be declared as LSPs. When some X clients communicate with network and drop to low-integrity, the X server, the desktop manager and other X clients can still maintain high integrity.

### 3.3 Restricting 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 carrying them out. We classify security-critical operations into two categories, file operations and operations that are not associated with specific files.

Examples of 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 often used 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 pieces. As of Linux Kernel 2.6.11, Linux has 31 different capabilities. The default UMIP 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 gids, and (2) going from the root account to another system account. (A system account, with the exception of root, does not correspond to an actual human user.) We allow low-integrity processes to use setuid and setgid this way because many daemon programs do them and they do not compromise our security objective. Note that by this design, a low-integrity process running as root cannot set its uid to a new normal user.

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 MAC** 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 writable only by root. This indicates that writing to these files is security critical. Similarly, files such as /etc/shadow would be readable only by root, indicating that reading them is security critical. 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. UMIP offers both read and write protection for files owned by system accounts. A low-integrity process (even if having effective uid 0) 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*. Finally, a low-integrity process is forbidden from changing the DAC permission of any (read- or write-) protected file.

**Exception policies: least privilege for sensitive operations** Some network-facing daemons need to access resources that are protected. 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 to authenticate remote users. If an attacker can exploit a 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 is denied the special privileges in the policy. The rationale is as follows. 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 when its integrity is low, and the invoked program needs special privilege. For example, sendmail needs to invoke procmail when its integrity is low, and procmail needs to write to the spool directory which is write-protected. We resolve this by defining executing relationships between programs. If there is an executing relationship between the program $X$ to the program $Y$, then when a process running $X$ executes $Y$, even if the process is in the state of low-integrity, the process will have the special permissions associated with $Y$ after executing. In the example, we define an executing relationship from sendmail to procmail and give procmail the special permission to write to the spool directory.

## 3.4 Contamination 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 the 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. We do not consider reading a directory that was changed by a low-integrity process as contamination, as the directory is maintained by 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 potentially raising the integrity of the file. The special utility program cannot be changed by low-integrity processes, so that its integrity level is always high.

Similar to the concept of RAP, we introduce file processing programs (FPP). A process running an FPP maintains its integrity level even after reading a low-integrity file. Programs that read a file's content and display the file on a terminal (e.g., vi, cat, etc.) need to be declared to be FPP.

We observe that our approach for handling file integrity is different from existing integrity models (such as Biba [4]), in which an object has one integrity level.

The integrity level of an object can be used to indicate two things: (1) the importance level of the object as a container (i.e., whether the object is used in some critical ways), and (2) the quality (i.e., trustworthiness, or, alternatively, contamination level) of information currently in the object. These two may not always be the same. When only one integrity level is used, one can keep track of only one of the two, which is problematic. Consider, for example, the system log files and the mail files. They are considered to be contaminated because they are written by processes who have communicated with the network. However, it is incorrect not to protect them, as an attacker who broke into the system through, say, httpd, would be able to change the log.

UMIP handles this by using a file's DAC permission to determine the importance level of the file, and using the sticky bit to track the contamination level. Even if a file has the sticky bit set (i.e., considered contaminated), as long as the file's DAC permission is not writable by the world, 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 the set of contaminated files intersect. This way, files such as system logs and mails are protected. This is different from other integrity models such as Biba, where once an object is contaminated, every subject can write to it. UMIP's design reduces the attack surface.

## 3.5 Files owned by non-system accounts

Not all sensitive files are owned by a system account. For example, consider a user account that 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 MAC. If a file is not writable by the world, then it is write-protected. UMIP 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 processes with that user's user-id. For example, a user may specify that a directory can be written by any low-integrity process and uses the directory to store all files 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 transparently (i.e., without knowing the existence of UMIP), then the policy can specify a global exception for the home directory of the user with recursion so that all low-integrity processes with the user's user-id can access the user's files. We point out that even with such a global exception, UMIP still offers useful protection. First, the exception 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 any process that later accesses them will drop its integrity level; the overall system integrity is still protected.

## 3.6 Design principles in UMIP

We now briefly examine how the design of UMIP illustrates the principles identified in Section 2. We follow principle 1 and aim at providing good enough security with a high level of usability. Following principles 2 and 3, we use an existing mechanism (namely, LSM) and focus on designing a policy model to achieve the security objective laid out in the beginning of Section 3. A major part of the work in developing the UMIP model is in designing the exception mechanisms (principle 4). Regarding principle 5, our approach differs from strict least privilege in two important ways. First, no limitation is placed on high-integrity processes, so they may operate with more privileges than strictly necessary. Second, non-sensitive files are not protected. Both design choices were made because they do not compromise our security objective and they increase the simplicity (and hence usability) of our model. Finally, following principle 6, UMIP uses files and capabilities in policy specifications, rather than exposing kernel data structures in the policy specification interface.

We believe that using DAC information is one key to the usability of UMIP. This makes deployment and installation of new software easy, as no labeling process is needed. This also uses concepts that users are already familiar with.

## 3.7 Other integrity models

The UMIP model borrows concepts from classical work on integrity models such as Biba [4] and LOMAC [10]. Here we discuss UMIP's novel features.

The Biba [4] model has five mandatory integrity policies: (1) the strict integrity policy, in which subject and object integrity labels never change; (2) the subject low-water mark policy, in which a subject's integrity level drops after reading a low-integrity object; (3) the object low-water mark policy, in which an object's integrity level drops after being written by a low-integrity subject; (4) the low-water mark integrity audit policy, which combines the previous two and allow the integrity levels of both subjects and objects to drop; (5) the ring policy, which allows subjects to read low-integrity objects while maintaining its integrity level. LOMAC [10] is an implementation of the subject low-water mark policy for operating systems. Each object is assigned an integrity level. Once assigned, an object's level never changes. It aims at protecting system integrity and places emphasis on usability. Compared with Biba and LOMAC, UMIP has the following novel features.

First, UMIP supports a number of ways to specify some programs as partially trusted to allow them to violate the default contamination rule or the default restrictions on low-integrity processes in some limited way. This enables one to use existing applications and administration practices, while limiting the attack surfaces exposed by such trust.

Second, in UMIP a file essentially has two integrity level values: whether it is protected and whether it is contaminated. The former is determined by the DAC permission, and does not change unless the file's permission changes. The latter is tracked using the sticky bit for protected files, and may change dynamically. The advantages of our approach is explained in Section 3.4.

Third, UMIP's integrity protection is compartmentalized by users. 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.

Fourth, UMIP allows low-integrity files to be upgraded to high-integrity. (This feature also exists in LOMAC.) This means that low-integrity information (such as files downloaded from 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 patching and system ungrade.

Fifth, UMIP 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.

Finally, UMIP uses DAC information to determine integrity and confidentiality labels for objects, whereas in LOMAC each installation requires manual specification of a mapping between existing files and integrity levels.

## 4 An Implementation under Linux

We have implemented the UMIP model in a prototype protection system for Linux, using the Linux Security Module (LSM) framework. We have been using evolving prototypes of the system within our group for a few months.

### 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's integrity level is high or low. When a process issues a request, it is authorized only when both the Linux DAC system and our protection system authorize it. A high-integrity process is not restricted by our protection system. A low-integrity process *by default* cannot perform any sensitive operation. 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), a local service point (LSP), and a file processing point (FPP); (3) a list of exceptions; and (4) a list of executing relationships, which is a list of programs that can be executed by the current program with the exception policies

| 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. |

**Figure 2. The four forms of file exceptions in UMIP.**

enabled, even if the process is low integrity. If a program does not have a corresponding entry, the default policy is that the program is not an RAP, a LSP or an FPP, and the exception list and the executing 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 four forms shown in the Figure 2.

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 that 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 a 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.

## 4.2 Evaluation

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

**Usability** One usability measure is transparency, which means not blocking 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 UMIP contribute to a high level of usability: the use of existing DAC information, the existence of RAP, LAP, and FPP, 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 simple policy specification as given in Figure 3. With this policy, we allow remote administration through the SSH daemon by declaring sshd as RAP. In this setting, one can also do remote administration through X over SSH tunneling and VNC over SSH tunneling. If one wants to allow remote administration through VNC without SSH tunneling, then he can declare the VNC Server as a RAP.

**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 shell access, the attacker typically tries downloading and installing attacking tools and rootkits. As these processes are low-integrity, the 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.

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 the test machine as root. When the attacker 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 compare what happens without our protection system with what happens when our protection system is enabled.

*1. 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 manipulating the image of the running kernel's memory in the file system (/dev/kmem).

We use two methods to determine whether a system has been compromised after installing a rootkit. The first method is to try to use the rootkit and see whether it is successfully installed. The second method is to calculate the hash values for all the files (content, permission bits, last modified time) in the local file system before and after installing the rootkit. For the calculation we reboot the machine using an external operating system (e.g., from a CD) and mount the local file system. This ensures that the running kernel and the programs used in the calcula-

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

**Figure 3. Sample policy**

tion are clean. A comparison between the hash results can tell whether the system has been compromised.

We tried two well-known rootkits. The first one is Adore-ng, 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 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 was not enabled, we were able to successfully install Adore-ng in the remote root shell and activat it. We were also able to replace any existing kernel module with the trojaned module so that the rootkit module would be automatically loaded during booting. When our protection system was enabled, the request to load the kernel module of Adore-ng from the remote root shell was denied, getting an "Operation not permitted" error. We got the same error when trying to replace the existing kernel module with the trojaned module. When trying to use the rootkit, we received a response saying "Adore-ng not installed". We checked the system integrity using the methods described above. The result showed that the system remained clean.

The second is Linux Rootkit Family (LRK). It is a well-known user-mode rootkit and replaces 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. When our protection was not enabled, we were able to install the trojaned SSH daemon and replace the existing SSH daemon in the system. After that we successfully connected to the machine as root using a predefined password. When our protection was enabled, installation of the trojaned SSH daemon failed, getting the "Operation not permitted" error. The system remained clean.

2. *Stealing the shadow File*: Without our protection system, we were able to steal /etc/shadow by send an email with the file as an attachment, using the command "mutt -a /etc/shadow alice@haker.net < /dev/null". When our protection was enabled, the request to read the shadow file was denied, getting an error saying "/etc/shadow: unable to attach file" .

3. *Altering user's web page files*: Another common attack is to alter web files after getting into a web server. In our experiment, we put the user's web files in a sub directory of the user's home directory "/home/Alice/www/". That directory and all the files under the directory were set as not writable by the world. When our protection was enabled, from the remote root shell, we could not modify any web files in the directory "/home/Alice/www/". We could not create a new file in that directory. Our module successfully prevented user's protected files from being changed by low-integrity processes.

**Performance** We have conducted benchmarking tests to compare performance overhead incurred by our protection system. 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.

We set up 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 with our protection system loaded as a kernel module.

The test results are given in Figure 4 and Figure 5. We compare our performance result with SELinux. The performance data of SELinux is taken from [14]. For most benchmark results, the percentage overhead is small ($\leq 5\%$). The performance of our module is significantly better than the data for SELinux.

## 5 Related Work

In Section 3.7 we have compared UMIP with Biba [4] and LOMAC [10]. Another well-known integrity model is the Clark-Wilson model [7], which divides data items into constrained data items (CDI's) and unconstraint data items (UDI's). CDI's are considered to have high integrity, and can be changed only by transformation procedures (TP's) that are certified to change the CDI's in ways that preserve their integrity. The Clark-Wilson model requires that, for each TP, the system lists which CDI's the TP is certified to access. PACL [23] also uses the idea of limiting the programs that can access certain objects. It uses an access control list for each file to store the list of programs that are allowed to access the file. Later approaches store such information with programs. As we have discussed in Section 3.1, determining access based just on the user id and the program that is running (without considering the history) is limited. While the policy exception part has its root in Clark-Wilson, UMIP is fundamentally different in that it maintains dynamic integrity levels for subjects and objects.

Our work differs from previous work that add MAC into UNIX, such as Trusted Solaris and IX [15], in that our goal is not multi-level security, but rather to preserve system integrity when the softwares running on the system are buggy and there are network-based attackers. Other systems that are closely related to ours include SELinux [16], systrace [17], LIDS [13], securelevel [11], and AppArmor [8, 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 pol-

| 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 4. 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 5. The performance results of lmbench 3 measurements (in microseconds).**

icy model that achieves a high degree of protection without getting in the way of normal operations. Both systrace and LIDS require intimate familiarity with UNIX 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 [8, 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. Similar to our approach, AppArmor also uses file paths to identify programs and files in the security profiles. Regarding policy design, 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 [20], 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, UMIP 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 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 [11] 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 the init process can lower it. The weakness of securelevel is clearly explained in the FreeBSD FAQ [11]: "*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.*" UMIP enables system administration through high-integrity channels, thereby avoiding the difficulty securelevel has. UMIP also tracks file contamination to ensure that all files read during booting are high integrity for the system to end up in a high-integrity state.

In UMIP, a program may be partially trusted in the sense that it is allowed to violate the default contamination rule and/or the limitations on low-integrity processes. Such trust is necessary for ensuring that existing applications and administration practices can be used. However, in UMIP programs are viewed as blackboxes, and the trust on them is not justified. The CW-Lite work [22] addresses this issue of trust by explicitly analyzing source code of programs. One identifies inputs into programs and annotates the source code with indications where filtering occurs. One then checks whether low-integrity inputs are properly filtered before they flow into high-integrity objects. This process enables one to discover bugs in the policy configuration or in the program source code. The CW-Lite work is thus complementary to UMIP.

## 6 Conclusions

We have identified six design principles for designing usable access control mechanisms. We have also introduced the UMIP model, a simple, practical MAC model for host integrity protection, designed using these principles. The UMIP 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. To achieve these, we introduced in UMIP several novel features in integrity protection. We have also reported the experiences and evaluation results of our implementation of UMIP 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 UMIP configuration and identify channels through which an attacker may get a high-integrity process (e.g., by exploiting a remote administration point).

**Acknowledgement** This work is supported by NSF CNS-0448204 and by sponsors of CERIAS. We thank Xuxian Jiang for helpful discussions and suggestions on approaches to evaluate the UMIP implementation under Linux. We also thank the anonymous reviewers and shepherd of our paper

Trent Jaeger for valuable comments that have greatly improved the paper.

## 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] 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.

[9] DOD. *Trusted Computer System Evaluation Criteria*. Department of Defense 5200.28-STD, Dec. 1985.

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

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

[12] 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.

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

[14] 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.

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

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

[23] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Crossley, P. Kerchen, K. N. Levitt, and R. Lo. Pacl's: An access control list approach to anti-viral security. In *Proceedings of the 13th National Computer Security Conference*, pages 340–349, Oct. 1990.

[24] 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.