LOMAC: Low Water-Mark Integrity Protection for COTS Environments

Timothy Fraser
tfraser@nai.com
NAI Labs
3060 Washington Road
Glenwood, MD 21738, USA

Abstract

We hypothesize that a form of kernel-resident access-control-based integrity protection can gain widespread acceptance in Commercial Off-The-Shelf (COTS) environments provided that it couples some useful protection with a high degree of compatibility with existing software, configurations, and practices. To test this hypothesis, we have developed a highly-compatible free open-source prototype called LOMAC, and released it on the Internet. LOMAC is a dynamically loadable extension for COTS Linux kernels that provides integrity protection based on Low Water-Mark access control. We present a classification of existing access control models with regard to compatibility, concluding that models similar to Low Water-Mark are especially well-suited to high-compatibility solutions. We also describe our practical strategies for dealing with the pathological cases in the Low Water-Mark model's behavior, which include a small extension of the model, and an unusual application of its concepts.

1 Introduction

In recent years, many commercial organizations providing Commercial Off-The-Shelf (COTS) software have rushed to integrate Internet-related functionality into their products, knowing that the new functionality will distinguish them from their competition, helping them to gain commercial advantage. This integration has increased the scope of the global security threat by exposing large numbers of previously isolated single-user workstations to attacks by malicious code and users via the Internet. In the race to be first with new application functionality, there is little time for security concerns. Consequently, public forums such as BugTraq are rife with reports of security problems in deployed COTS software that leave the integrity of processes and data vulnerable to attack. To counter this increased threat, there is a need to apply integrity protection technologies to existing COTS systems. However, such application is not easy. Integrity protection technologies which require users to replace their investment in currently deployed COTS software, or interfere with application functionality are unlikely to be widely adopted, regardless of the protection they provide. Applicable technologies must enhance, rather than replace, existing COTS software.

Fortunately, several such technologies exist, and have met with success as commercial products. These technologies include firewalls, virus scanners, and intrusion detection systems. We hypothesize that integrity protection technology based on kernel-resident access control can also be applied as an enhancement to existing COTS systems, provided that, like the successful technologies above, it provides at least some useful protection at a near-zero cost in compatibility. To test this hypothesis, we have implemented a form of Low Water-Mark access control [5] in a free open-source prototype for COTS Linux systems. We call this prototype LOMAC, an acronym derived from "Low Water-Mark" and "Access Control". We are providing LOMAC to potential adopters via the Internet and measuring its rate of acceptance. LOMAC is designed to meet the following specific Compatibility Goals to the greatest extent possible:

1. LOMAC should be compatible with the existing deployed COTS operating system kernels and applications. This goal implies that LOMAC should not require the replacement or source-code modification of any existing software in order to operate.

2. LOMAC should not require any changes to pre-existing kernel or application configurations.

3. LOMAC should not cause failures in previously working applications.

4. LOMAC's existence should be largely invisible to the user except at the moments when specific integrity threats occur. This goal implies that the user should not be required to learn any new behaviors in order to work in a LOMAC-enhanced environment.

5. LOMAC should provide some useful protection in its default configuration. This goal implies that LOMAC

must offer a useful default “one size fits all” integrity policy for those adopters who do not wish to expend the effort required to learn about and configure LOMAC. We refer to this policy as LOMAC’s Default Policy.

We define Total Compatibility Cost as the cost incurred by failing to meet these Compatibility Goals. Although a zero Total Compatibility Cost is probably not achievable, based on the commercial success of the integrity protection technologies listed above, we believe that a solution bearing a small but near-zero Total Compatibility Cost, coupled with some useful protection benefit, can still be widely adopted in COTS environments. We can determine the extent to which LOMAC meets most of these goals by examining the structure and behavior of the prototype itself. Unfortunately, we cannot effectively measure compliance with the third goal, because doing so would require us to examine all of the application programs existing in a given LOMAC-enhanced environment, searching for potential failures. In the general case, the number of application programs is far too great to make such an analysis practical. However, we can characterize the cost associated with LOMAC using the the notion of Partial Compatibility Cost, which we define as the same metric as Total Compatibility Cost excluding the troublesome third goal. We supplement this characterization with experimental evidence concerning LOMAC’s avoidance of application failures, and argue that a proper analysis is possible in the special case of LOMAC’s Default Policy.

Our experiment is ongoing. However, this paper presents the two primary results derived from its early stages. The first result concerns our analysis of potentially applicable access control models, which suggests that some models possess a specific property (defined below) that makes them easier to apply to a COTS environments with low Partial Compatibility Cost than others. Surprisingly, the class of models with this property includes examples which have seldom been implemented, such as Low Water-Mark and Chinese Wall, and excludes examples that have received greater attention, such as Ring Integrity, Strict Integrity, Clark-Wilson, Type Enforcement, and Domain and Type Enforcement (DTE). The second result concerns the well-known pathological case inherent in the Low Water-Mark model’s behavior which, if left unaddressed, could cause application failures and unacceptably increase LOMAC’s Total Compatibility Cost. Although we cannot entirely remove the pathological case without also removing the protective properties of the model, we have developed techniques to avoid it in COTS UNIX environments. These techniques include a small extension of the formal model, and an unusual mapping of its concepts to the UNIX operating system abstractions. Our experience and our preliminary usability analysis indicate that it is possible to accomplish meaningful work on LOMAC-enhanced COTS Linux systems, with near-zero cost according to our Compatibility Goals.

In section 2, we begin by presenting our analysis of access control models that are applicable to the COTS environment, and argue that some classes of models enable low Partial Compatibility Cost solutions more easily than others. This argument justifies our choice of the Low Water-Mark model as the basis for LOMAC. In section 3, we describe the LOMAC architecture and its tradeoffs between low Partial Compatibility Cost and quality of integrity protection. We conclude that, despite the extreme nature of the tradeoffs, a low Partial Compatibility Cost architecture can still provide useful integrity protection. In section 4, we describe how LOMAC applies the Low Water-Mark model’s concepts to the Linux kernel’s abstractions, and how this mapping results in useful integrity protection. We explain how the Low Water-Mark model’s behavior makes it unnecessary for LOMAC to be aware of many installation-specific details, such as the assignment of duties among users, or the purposes of individual daemons. This freedom allows LOMAC to provide a “one-size-fits-all” Default Policy that allows it to enforce useful integrity protection without being configured. Section 5 discusses how the Low Water-Mark model can increase Total Compatibility Cost by causing application failures, and presents our unusual application of model concepts and our model extension for overcoming this problem in the COTS UNIX environment. In section 6, we compare the LOMAC experiment to related work, emphasizing in particular the difference between LOMAC’s goals and the goals of trusted operating systems. We conclude with an analysis of the quality of LOMAC’s integrity protection in section 7, and discuss a strategy to measure Total Compatibility Cost in the special case of LOMAC’s Default Policy.

2 Models and Partial Compatibility Cost

There are many access control models capable of providing integrity protection in COTS environments. However, some models are more suitable than others because they possess specific properties which help to minimize Partial Compatibility Cost. In this section, we define these properties and use them to classify the population of models according to their effect on Partial Compatibility Cost. Based on this analysis and our experience with the LOMAC prototype, we conclude that the models most suitable for the COTS environment are those which are capable of revising the privileges they award to a given subject based on the objects the subject observes during its runtime. This class includes the Chinese Wall [7] and Low Water-Mark models [5]. Out of the universe of models capable of providing integrity protection, we focus our analysis on the

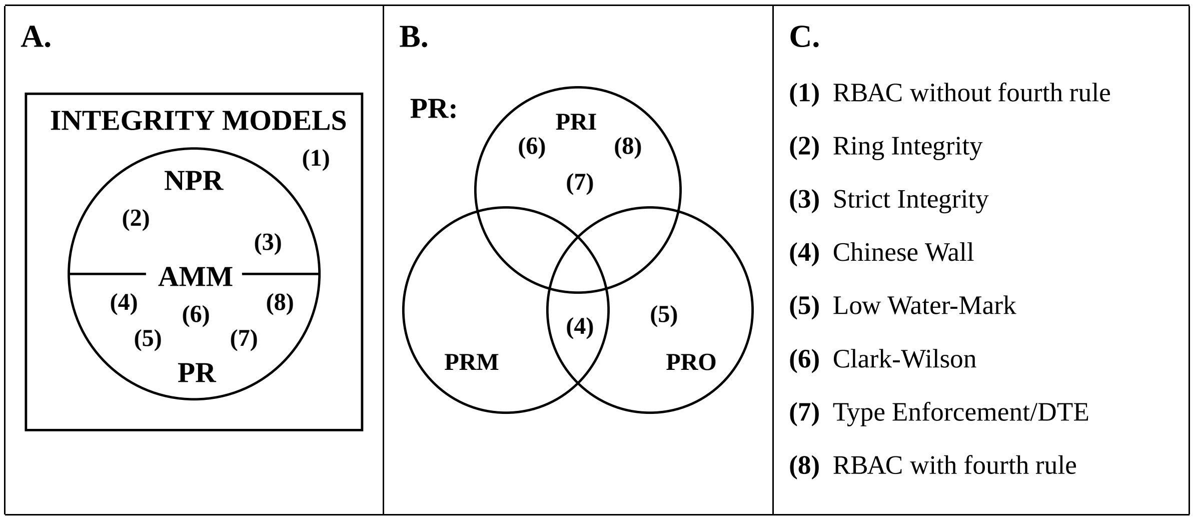

Figure 1. Venn diagrams describing the relationships between models with numbered examples.

set of models which include subjects, objects, and Access Matrices among their fundamental concepts [14, 15]. Figure 1A contains a Venn diagram representing this set as a circle marked AMM for Access Matrix Models. We chose this set because, as shown by the index in figure 1C, it contains many (though not all) well-known examples of models which provide useful integrity protection. These examples include the Ring Integrity, Strict Integrity, Low Water-Mark [5], Chinese Wall [7], Clark-Wilson [8], Type Enforcement [6], and DTE [3] models. It also includes the Role-Based Access Control model in the case where the model's fourth rule is applied to define an Access Matrix [9].

All of the Access Matrix Models assign a particular set of privileges to each subject - each subject's "privilege set". The diagram partitions the AMM set into two halves. One half is marked NPR for "No Privilege Revision". This half contains all models in which a subject's privilege set does not change once assigned. The other half is marked PR for "Privilege Revision". This half contains the rest of the AMM models, in which a subject may be assigned a new privilege set when it performs certain operations. The members of the PR set can be further subdivided according to which of the three fundamental operations of the Access Matrix Models: invoke, observe, or modify, prompts them to revise a subject's privilege set assignment. Figure 1B contains a Venn diagram showing this further classification of the PR models. The PR models which may revise a subject's privilege set assignment when that subject performs an Invoke operation (a program execution or procedure call) are contained in the set labelled PRI, for "Privilege Revision on Invoke." The PR models which reassign privilege sets in response to Observe (read) and Modify (write) operations are contained in the sets labelled PRO and PRM, for "Privilege Revision on Observe" and "on Modify," respectively.

When applied to a COTS environment, the most discriminating characteristic of an AMM model with regard to Partial Compatibility Cost is the method it uses to choose which privilege set to assign to a given subject. Other issues which affect Partial Compatibility Cost, such as the method used to divide existing objects into integrity classes, are equally difficult among all the AMM models. Note that we define the PRI, PRO and PRM properties in terms of the operations which prompt a PR model to revise a subject's privilege set assignment at a given point in time. Our definition does not limit the criteria on which a model can base its privilege set choice once it is prompted to choose. Although all the example PRI models shown in figure 1B choose based (in part) on the identity of invoked procedures or programs, and all the example PRO models choose based on the integrity level of observed objects, our definition does not require this limited behavior. For example, we admit the possibility of a PRM model which, when prompted to choose a new privilege set by a subject's modify operation, can consider the subject's observation or invocation history before choosing.

The observe operation is the only one of the fundamental three which allows a subject to decrease its own level of integrity by importing data from a lower-integrity object into itself. The Access Matrix Models provide integrity protection by preventing subjects from transferring data from low-integrity objects (via observe operations) to high-integrity objects (via modify operations). They accomplish this provision by assigning those subjects which must observe low-integrity objects (due to the functionality needs of their application) privilege sets which do not permit them to modify higher-integrity objects. They may assign privilege sets permitting the modification of higher-level objects to those subjects which refrain from observing low-integrity objects. The proper choice of privilege set for a given subject is, consequently, dependent on the integrity level of the

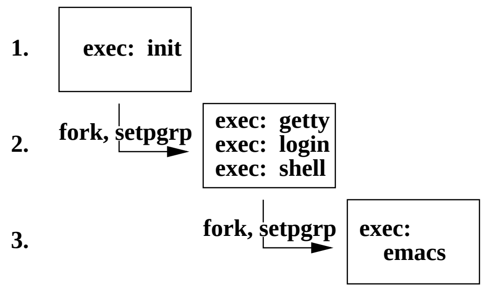

Figure 2. The UNIX login process tree

objects the subject must observe.

The PRO models use a simple approach to choose the appropriate privilege set to assign to a given subject. When a subject is created, a PRO model will assign it an initial privilege set. As described in section 4, the Low Water-Mark PRO model initially assigns the most permissive privilege set to the first subject, and subsequently lets every new subject inherit its creator's privilege set as its own initial privilege set. When a given subject decreases its level of integrity by observing an object, a PRO model can assign it a new, more restrictive privilege set reflecting the subject's loss of integrity. The PRO models require no knowledge of the algorithms of applications or of the duties of users in order to choose which privilege set to assign to a subject. They simply react to each subject's observations, as they occur.

The PRI models use a more complicated approach to choosing appropriate privilege sets for subjects. PRI models may also assign an initial privilege set to each subject, as described above. However, rather than revising their choice when a subject performs an observe operation, they may revise their choice when a subject performs an invoke operation, such as executing a program or calling a procedure. Reacting to invoke rather than observe operations complicates the task of choosing a privilege set, since the invoke operation is not the fundamental operation which actually allows a subject to decrease its own integrity. A PRI model must use the identity of the program a subject chooses to execute to predict which objects the subject will observe in the future, and choose which privilege set to assign to the subject based on this prediction. The proper choice may not be clear when a given program can be used to access objects of varying integrity level. Some PRI models supplement program identity with additional criteria to avoid such ambiguous situations. The more numerous the criteria, the fewer ambiguous situations will occur. The Clark-Wilson PRI model, for example, makes it choice based on both the identity of the program and of the user controlling the subject. The DTE PRI model chooses based on the identity of the program the subject wishes to execute, and also the identities of the programs the subject and its ancestors have executed in the past.

In general, regardless of the number of additional criteria, ambiguous situations can never be entirely prevented. The COTS UNIX login process tree provides a pertinent example. Figure 2 contains a diagram of this process tree, divided into three sequential steps. Each square in the diagram represents a subject, equivalent to a job containing one process in terms of UNIX abstractions. The first step in the diagram shows a UNIX kernel's initial subject, which executes the **init** program. The initial subject creates one new subject for every hardware terminal line. The diagram's second step shows one of these new subjects, which begins by executing the **getty** program. When a user attempts to log in via its hardware terminal line, the subject executes the **login** program to handle authentication, and then executes the user's interactive shell if the authentication is successful. At the user's command, the interactive shell creates a new subject to execute the **emacs** editor application in step three.

Ambiguity may occur when the **root** (administrator) user logs in. The **root** user may command the subject executing **emacs** to observe a low-integrity object. In this case, the proper privilege set for the **emacs** subject is one that prohibits modification of high-integrity objects. On the other hand, the **root** user may avoid low-integrity objects, commanding the **emacs** subject to observe and modify only high-integrity objects. In this case, the privilege set which prohibits modification of high-integrity objects would be inappropriate. In this ambiguous situation, there are two potentially appropriate privilege sets for one user, entering the system from one hardware terminal line, with one execution history, executing one program. The appropriate choice is not apparent until after the subject executes the **emacs** program and observes objects, at which point it is too late for a PRI model is to reassign its privilege set.

There are two possible methods of resolving these ambiguous situations: manual, and automatic. Both methods incur Partial Compatibility Cost. A manual solution might force a user to choose the appropriate privilege set for a given subject via a menu. Adding this feature to a COTS environment would require the modification or replacement of existing software. An automatic solution might make assumptions about the configuration of the system or the behavior of its users. For example, it might assume that all subjects operating on behalf of a user with a particular identity, at a particular terminal, or from a given network address deserve a particular privilege set. These automatic solutions may also involve some Partial Compatibility Cost if existing user behaviors or configurations must adapt to avoid contradicting their assumptions. The PRI model's need to predict a subject's future observation behavior at the time it performs an invoke operation can lead to ambiguity in certain situations. In these situations, all of the available solutions may involve some Partial Compatibility Cost. In contrast to the PRI models, the PRO models react to a subject's observe operations directly; they do not suffer from the problems of prediction and ambiguity. Consequently, the PRO models are capable of providing low Partial Compatibility Cost solutions in more cases than the PRI models.

The NPR models, which cannot revise their privilege set assignments at all, must predict a subject's future observation behavior at the time the subject is created. The criteria on which the NPR models may base their predictions are fewer than those available to the PRI models. The NPR models must choose the appropriate privilege set for a given subject before it has indicated which program it intends to invoke. Consequently, program identity is not an available criterion. With fewer criteria available, ambiguity, and its potential for increased Partial Compatibility Cost, will occur in more cases with the NPR models than with the PRI models. Consequently, the NPR models are capable of providing low Partial Compatibility Cost solutions in fewer cases than the PRO and PRI models.

The only class of models not addressed above is PRM. In some cases, a pure PRM model can be equivalent to a PRO model in Partial Compatibility Cost. First, it must consider a subject's observation history when it revises the subject's privilege set assignment in response to a modify operation. Second, it must treat invoke operations as it does modify operations. This second condition prevents a subject from making dangerous invocations after it has made a corrupting observation, but before it prompts the model to revise its privilege set assignment by performing a modify operation. However, the effort required to maintain an observation history for each subject may exceed the effort required to implement PRO functionality in addition to PRM functionality. The Chinese Wall model, for example, avoids the maintenance of observe history by responding to both modify and observe operations, making it a member of both the PRO and PRM sets. Consequently, we consider the PRO models to be an easier route to a low Partial Compatibility Cost solution than a purely PRM model.

The observation-sensitive property of the PRO models allows them to provide low Partial Compatibility Cost solutions in more situations than the PRI and NPR models. Consequently, the PRO models are the most applicable Access Matrix Models to the COTS environment in terms of Partial Compatibility Cost. We chose the Low Water-Mark PRO model for LOMAC over the Chinese Wall model because Low Water-Mark never prevents a subject from reading an object. We perceived this behavior to be useful in a COTS UNIX environment where many objects, such as program binaries, are presented to low-integrity subjects in a read-only fashion.

3 Prototype architecture

In order to be suitable for our experiment, the LOMAC prototype is designed to minimize Partial Compatibility Cost first, and to provide useful integrity protection second. This ordering reverses the priorities of several past efforts to apply kernel-resident access control [12, 20, 26], and consequently, LOMAC cannot exploit some of the more expensive techniques demonstrated by these past efforts to improve the quality of its protection. However, LOMAC's architecture embodies the proper tradeoffs to test our hypothesis concerning compatibility and acceptance. The LOMAC architecture uses a combination of two familiar techniques: the use of a Loadable Kernel Module (LKM) to extend an existing COTS operating system, and the use of Interposition at the system call interface to modify the operating system's behavior [10, 11, 22]. Both of these techniques bring benefits that help to minimize Partial Compatibility Cost, and drawbacks that decrease the quality of protection. We discuss the overall impact of each technique separately.

Most COTS operating systems support LKM or LKM-like functionality, including Linux, Solaris, Windows NT, and the majority of the BSD-derived operating systems. The use of an LKM allows us to avoid the unacceptable Partial Compatibility Cost of replacing existing deployed COTS operating systems by retaining and enhancing them, instead. As an LKM, LOMAC is compiled into a relocatable object file, and then dynamically loaded into the kernel's address space during bootstrap. Unfortunately, LKM functionality is not sufficient to implement a proper Reference Monitor [1]. Although LKM functionality allows us to load LOMAC into the kernel's address space, where it is protected from tampering by user-space programs, it is not protected from tampering by other LKMs or by the kernel itself. Research into safer means of kernel extension is ongoing [4, 23, 28]; however, we must make do with the LKM functionality available in our COTS environment. Consequently, LOMAC offers a lower quality of protection than would be provided by a tamper-proof reference monitor.

Once loaded into the kernel's address space, LOMAC uses Interposition to intercept calls to the security-relevant portion of the kernel's system call vector. LOMAC must intercept these calls in order to make and enforce access control decisions. Like the use of the LKM, the use of Interposition allows us to minimize Partial Compatibility Cost by extending existing deployed COTS operating systems instead of replacing them. However, Interposition also has its drawbacks [11, 29]. To support its decision-making, LOMAC must associate security attributes with many existing kernel abstractions. Since LOMAC interfaces with the kernel by Interposition, avoiding modifications to the kernel's source code, the kernel's existing data structures do not provide storage for these attributes. Consequently, LOMAC is forced to implement its own additional data structures to manage these attributes. This additional implementation makes LOMAC larger and more complex than an equivalent Reference Monitor that interfaces with the kernel through direct modifications of the kernel's source code. LOMAC's correctness is therefore more difficult to verify with formal methods, again resulting in a lower quality of protection. However, the conditions of our experiment demand compatibility first, and quality of protection second. Consequently, LOMAC's use of an LKM and Interposition is justified in this case.

4 Practical integrity protection

The LOMAC prototype implements a form of access control based on a slightly extended version of the Low Water-Mark model [5]. We describe the basic operation of the model informally here, and leave the more formal discussion of the model and our extension to section 5. The model defines the concepts of subject, object, and level. Subjects are active entities that execute programs. LOMAC applies the model's subject concept to the Linux job abstraction. Objects are passive entities that contain data. LOMAC applies the model's object concept to the Linux file, named pipe, and socket abstractions. Levels are labels indicating a relative level of integrity. LOMAC represents levels with positive integers; level 2 indicates a greater amount of integrity than level 1. LOMAC assigns a level to each existing object, effectively partitioning all objects into classes based on their level of integrity. Once assigned, an object's level never changes. LOMAC provides integrity protection by preventing the movement of potentially corrupted data from lower-level to higher-level objects by restricting the behavior of subjects.

At configuration time, each LOMAC installation can define its own policy. A policy specifies the number of levels in use, and the mapping between existing objects and levels. For those installations which cannot afford to spend effort on configuration, LOMAC implements a simple Default Policy that provides a basic level of integrity protection appropriate for all environments. This Default Policy contains only two levels: level 1 for low integrity objects, such as downloaded Internet content, and level 2 for high integrity objects, such as the system binaries installed from the Linux CD-ROM distribution. The Default Policy assigns levels to existing objects by following a series of simple rules. Our current prototype uses the following three rules: First, all objects existing immediately after the operating system is installed receive level 2. Second, the user objects created subsequently receive level 1. Third, all hardware devices allowing access to the system, such as the console and serial lines for terminals, receive level 2, except for Network Interface Cards (NICs), which receive level 1. From that point, all new objects created during system run-time inherit the level of the subject which created them.

Potentially corrupted data can move upward in level when a subject observes (reads from) a low-levelled object and subsequently modifies (writes to) a higher-levelled object. In order to prevent this kind of upward movement, LOMAC assigns an initial level to each subject upon creation. Unlike object levels, subject levels can decrease over time. Whenever a subject observes an object with a level lower than its own, LOMAC "demotes" the subject, reducing its level to match the object's. A subject is never "promoted" - its level can never increase. A subject's level determines its privileges - while LOMAC allows any subject to observe any object, it prevents subjects from modifying objects whose levels are higher than their own. The act of observing a lower-levelled object renders a subject incapable of spreading corruption by modifying a higher-levelled object.

LOMAC's demotion behavior provides useful integrity protection against viruses, Trojan horses, and users intent upon misuse. For example, the Default Policy causes LOMAC to assign level 1 to objects downloaded from the Internet. If a particular downloaded object contained interpretable content implementing a Trojan Horse, it might use a stack-smashing attack to take control of a subject which observed (read and executed) it. Without LOMAC, if the interpreting subject had **root** (administrator) privileges, the Trojan Horse would would be capable of inserting backdoors into the system by modifying system binary objects. However, with LOMAC and the Default Policy, the act of observing the downloaded object demotes the interpreting subject to the object's level, 1. Despite its **root** privileges, the level 1 interpreting subject would be incapable of modifying system binary objects, which exist at level 2.

As stated above, a LOMAC subject is equivalent to a UNIX job. A job contains one or more cooperating processes. The process is the abstraction which actually executes programs in many COTS UNIX systems, and is arguably a more intuitive equivalent for the subject concept than is the job. However, LOMAC's subject-job equivalence brings specific Compatibility Cost benefits, as described in section 5, and consequently is more appropriate. In LOMAC-enhanced Linux systems, existing subjects create new subjects via the **setpgrp** system call. The Linux kernel creates the first subject to execute bootstrap programs. Subsequently, this subject creates others to execute other system programs and daemons (servers). These subjects, in turn, create a third generation of subjects to execute user programs. LOMAC uses a simple scheme to determine the which initial level to assign to each subject. LOMAC assigns the first subject the highest level defined by the policy. Subsequently, LOMAC allows each new subject to inherit the level of its creator. As the system runs, LOMAC's de-

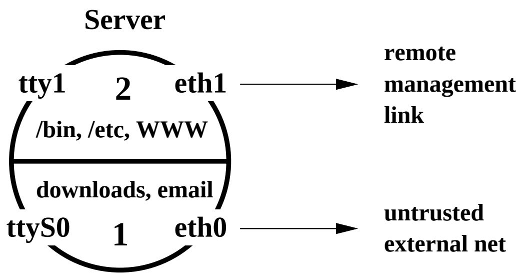

Figure 3. LOMAC's Default Policy

motion behavior ensures that subjects that deal with potentially dangerous low-integrity objects execute with appropriately restricted privileges.

We can exploit this subject level assignment behavior to provide subjects executing daemons which serve remote clients with privileges proportional to the local system's faith in the remote clients' good intentions. We can also exploit this behavior to automatically assign appropriately restricted privileges to subjects operating on behalf of local users based on the duties the individual users must perform. As shown in figure 3, we accomplish this privilege assignment through the familiar technique of associating levels with the hardware devices through which local users and remote clients access the system. The figure contains a diagram of a small Linux server used in early LOMAC testing. The server is protected by LOMAC's Default Policy. The diagram represents the system as a circle split into a halves, one for level 2 and the other for level 1. Each half has two breaks in the circle's perimeter representing hardware devices providing entry into the system. In the level 2 half, there is a console device (**tty1**) and a NIC acting as one endpoint of a point-to-point link for remote management (**eth1**). Physical access to the console and the remote management system is restricted to administrative users only. In the level 1 half, there is a serial line connected to a dumb terminal (**ttys0**), and a second NIC connected to the Internet (**eth0**). The terminal and the Internet are not subject to physical access restrictions. LOMAC depends solely upon the restrictions on physical access to these devices to authenticate users and remote clients. This minimal dependence shields LOMAC from weaknesses in a COTS system's existing authentication mechanisms, such as a reliance on weak passwords or the continuity of hijack-able network sessions. LOMAC does not treat these hardware devices strictly as objects: it does not restrict the ability of subjects to modify (write to) them. However, when a subject observes (reads from) one of these devices, LOMAC assumes the device has the level indicated by the diagram for demotion purposes.

As stated above, under the Default Policy LOMAC assigns level 2 to the initial subject, and all of the new subjects it creates inherit this level. LOMAC demotes level-2 subjects which read from level 1 devices such as **ttyS0** or **eth0** to level 1. When a subject reads its first login from **ttyS0** or its first remote client request from **eth0**, LOMAC demotes it to level 1 - an appropriate level for subjects operating on behalf of potentially malicious users. By associating level 2 with **tty1** and **eth1**, LOMAC allows subjects operating on behalf of administrative users to remain at the highest level of privilege, unless their duties specifically require them to observe level-1 objects. By exploiting the Low Water-Mark model's demotion behavior in this manner, LOMAC ensures that all users and daemons operate with appropriately restricted privileges without foreknowledge of the duties of each user, or the purpose of each daemon. As described in section 2, this potential for automatic appropriate level assignment makes it easy to apply the Low Water-Mark model to COTS UNIX systems at low Partial Compatibility Cost.

In certain situations, a given process can move itself or another process from one subject-job to another. If left unaddressed, this movement between subject-jobs could could allow processes to carry potentially corrupted data from lower to higher levels. LOMAC addresses this problem by enforcing three additional safeguards concerning the behavior of processes. First, LOMAC restricts the use of certain security-critical system calls (most notably the system call to trigger reboot) to processes residing in subject-jobs at the policy's highest level. Second, LOMAC prevents processes from sending signals (UNIX software interrupts) to, or changing the process groups of, other processes residing in subject-jobs at levels higher than their own. Third, LOMAC prohibits a process from moving from one subject-job to another subject-job at a higher level under any circumstances. The first two safeguards help to prevent processes residing in lower-level subject-jobs from interfering with the execution of processes residing in higher-level ones. The third prevents the transfer of corrupted data from lower to higher levels. These safeguards are not part of the Low Water-Mark model; the model is unaware of processes. These safeguards are constraints derived from our particular application of the Low Water-Mark model to COTS Linux systems, similar to the constraints "subject are jobs" and "objects are files, named pipes, and sockets" described above.

Unfortunately, along with the protection benefits described above, the use of LOMAC brings an inherent Compatibility Cost. Like many other access control schemes (including those discussed in section 2), Low Water-Mark is pessimistic. It prevents data movements that represent *potential* integrity threats - some prevented movements may, in reality, have been harmless. In a LOMAC-protected system, users who habitually move data in dangerous ways (perhaps by inserting data arriving in low-integrity email attachments into high-integrity local documents) may find themselves forced to change their behavior. Such behavior changes increase Partial Compatibility Cost. In section 7, we argue that using LOMAC with the Default Policy requires minimal behavior changes.

5 Applying the Low Water-Mark model

The Low Water-Mark model has seen relatively little application in the past. This fact is probably due largely to the existence of a pathological case inherent in its demotion behavior [5, 6]. This pathological case, which we call the Self-Revocation Problem, can cause failures in application functionality. Consequently, if left unaddressed, it can increase the Total Compatibility Cost of LOMAC. The Self-Revocation Problem describes the situation where the Low Water-Mark model's demotion behavior unexpectedly revokes a subject's right to modify an object. For example, this situation can occur when a subject at a high level creates an object, and subsequently observes another object, which is at a low level. According to the Low Water-Mark model, this observation demotes the subject to the same low level. Consequently, the subject, now at a low level, cannot modify the object it created, which remains at a high level.

It is important to note that this is the proper behavior from the standpoint of protection. By reading the low-levelled object, the subject may have infected itself with malicious code - it must be prevented from spreading the infection to higher levels by writing to the object it created. However, from the standpoint of application functionality, this behavior is troublesome. Due to the capability-based nature of UNIX file descriptors, most UNIX application programs expect to run in an environment where mediation is done once, when they acquire the capability to access an object [21]. They do not, in general, expect the access to be subsequently revoked. However, this revocation is exactly what occurs in the Self-Revocation Problem, and it is reasonable to expect most UNIX application programs to respond to this unexpected event by misbehaving or failing entirely [25].

We must address the Self-Revocation Problem in order to reduce LOMAC's Total Compatibility Cost. As described in section 1, measuring the application failure aspect of Total Compatibility Cost in the general case is difficult, at best. Our experience with the LOMAC prototype indicates that most instances of the Self-Revocation Problem occur during Inter-Process Communication (IPC) involving unnamed pipes and shared memory abstractions. Consequently, we have focused our efforts on avoiding the Self-Revocation Problem in these two cases. Although the impracticality of measuring Total Compatibility Cost prevents us from claiming that our solutions provide a sufficiently problem-free environment in the general case, section 7 describes a method through which we could eventually measure the effectiveness of our approach in the special case of LOMAC's Default Policy.

Unfortunately, since the Self-Revocation Problem is an inherent aspect of the Low Water-Mark model's proper protection behavior, it cannot be entirely removed from the formal model without removing the protection behavior, as well. In lieu of removing it entirely, we must endeavor to reduce the number of situations in which it will cause application failures in practice. Research in Security Agility has demonstrated the effectiveness of adding functionality to user-space applications that enables them to adapt to privilege revocation and to continue operating in some partial capacity rather than simply failing altogether in some cases [25]. We have chosen to take an alternate approach, in order to avoid increasing Partial Compatibility Cost by modifying or replacing existing applications. As described in section 4, we have chosen to employ an unusual mapping between the Low Water-Mark model's concepts and the actual operating system abstractions in order to reduce occurrences of the Self-Revocation Problem in shell pipelines. We have also extended the formal model slightly to avoid the Self-Revocation Problem during IPC based on shared memory abstractions.

Early versions of the LOMAC prototype were plagued with application functionality failures due to the Self-Revocation Problem during the execution of shell pipelines. The early prototypes mapped the concepts of the standard Low Water-Mark model to the operating system abstractions provided by the Linux kernel in a naive fashion. This naive mapping followed a traditional pattern successfully demonstrated by previous applications of several access control models without Self-Revocation Problems to operating systems [2, 3]. The naive mapping considered each process a subject, and each inode (the operating system abstraction for files, sockets, unnamed pipes, and FIFOS) an object. The vulnerability of shell pipelines to the Self-Revocation Problem under this naive mapping is illustrated by the example diagrammed in figure 4.

Step 1 of the diagram shows the initial state of a typical shell pipeline. It contains two subjects: the Linux **ps** and **grep** utility applications, connected by one object: an unnamed pipe. Both subjects and the objects are initially at level 2. The Linux **ps** utility application reads information from the **/proc** filesystem. As shown in step 2 of the diagram, some **/proc** filesystem entries are at level 1. Step 3 shows the consequences of the **ps** utility application's read: it is demoted to level 1. When it attempts to pass its output to the **grep** utility application by writing to the unnamed pipe, its write is denied. As shown in step 4 of the diagram, the unnamed pipe remains at level 2, and cannot be written to by the demoted **ps** utility application. In practice, the **ps** utility application fails to pass its output to **grep**, which is forced to present the user with a meaningless lack of output. This misbehavior is an unacceptable failure of application

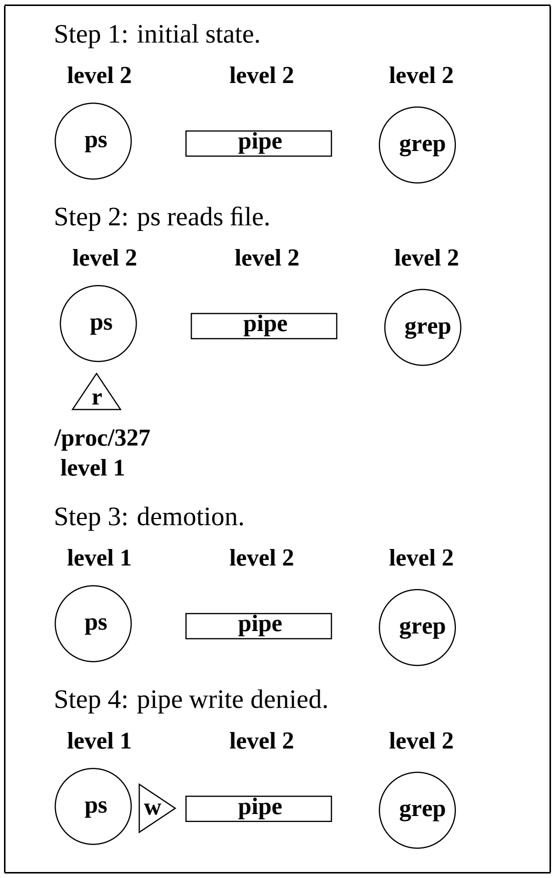

Figure 4. The Self-Revocation Problem.

functionality in terms of our Compatibility Goals.

Although this self-revocation behavior is consistent with the formal Low Water-Mark model, it is inconsistent with our expectations as a user. As a user, we consider the combination of the **ps** and **grep** to be a unit, or "job", operating to complete a single task. A superior application of the Low Water-Mark model's concepts might treat the entire job as a single subject, demoting it as a unit upon reading from the **/proc** filesystem, and avoiding a break in the pipeline. While this redefinition of the meaning of subject is easily done, the question remains: what to do with unnamed pipe objects? In the example of figure 4, little would be gained by demoting both **ps** and **grep** as a single subject at step 3 if the unnamed pipe object that connects them remains unwritable at level 2.

One answer to this question might be to modify the formal model, causing it to adjust the level of unnamed pipe objects to follow subject demotion. However, this option is unattractive, since it violates the basic tenet of the Low Water-Mark model that object levels never change. Rather than complicate the formal model, a better solution might be to modify our application of the model's object concept to the actual operating system abstractions. The most simple solution of this kind would be not to consider unnamed pipes as objects at all, and to implement a rule in the prototype to guarantee that pipes may link only processes that are part of the same subject. We refer to this option as the "Unnamed Pipe Possession Rule". This option is attractive, since it allows the proper operation of unnamed pipes by exempting them from LOMAC's access control, while simultaneously maintaining integrity protection by preventing IPC between subjects using unnamed pipes. Unfortunately, the usage of unnamed pipes by critical UNIX applications, particularly the C shell [16], prevents the use of such a simple rule, as shown in figure 5.

Figure 5A contains a simplified diagram of the algorithm used by the C shell to create a typical job consisting of two fictitious application programs, named **source** and **sink**, connected by an unnamed pipe. At each step in the algorithm, the diagram lists the number of subjects and objects present according to the naive application of Low Water-Mark concepts described above. The diagram also introduces the process group operating system abstraction; processes possessing the same process group identifier are members of the same job. Step 1 shows an initial state where the shell has created a pipe. Step 2 and 3 show how the shell subsequently creates (via the **fork** system call) the **sink** and **source** processes, and gives them a new process group identifier unique to their job (via the **setpgrp** system call). Step 2 is critical to the unnamed pipe-handling issue. At this stage in the algorithm, two processes in different jobs possess the same unnamed pipe - a critical item of C shell functionality that would be prohibited by the Unnamed Pipe Possession Rule.

Because we wish to avoid causing failures in existing applications, we must reject the Unnamed Pipe Possession Rule in favor of a slightly less restrictive version - the Unnamed Pipe Usage Rule. This rule states that the subject-job containing the first process to read from or write to a particular unnamed pipe possesses it for all time, and subsequent reads and writes to this unnamed pipe are allowed only for that subject-job. This rule has the advantage that it allows the capability to read and/or write an unnamed pipe to be passed between jobs as required by the C shell, which does not read from or write to the unnamed pipes it creates for other jobs. It also prevents subjects from bypassing the Low Water-Mark model's integrity protection by passing data between subject-jobs via unnamed pipe IPC.

Although it allows LOMAC to avoid the Self-Revocation Problem in shell pipelines, the Unnamed Pipe Usage Rule also has the potential to cause applications which attempt unnamed pipe IPC across job boundaries to fail. Although the job concept is meant to encapsulate groups of processes that cooperate via unnamed pipe IPC, the LOMAC prototype will incur Total Compatibility Cost in unusual environments which do not use the job abstraction in this way. A

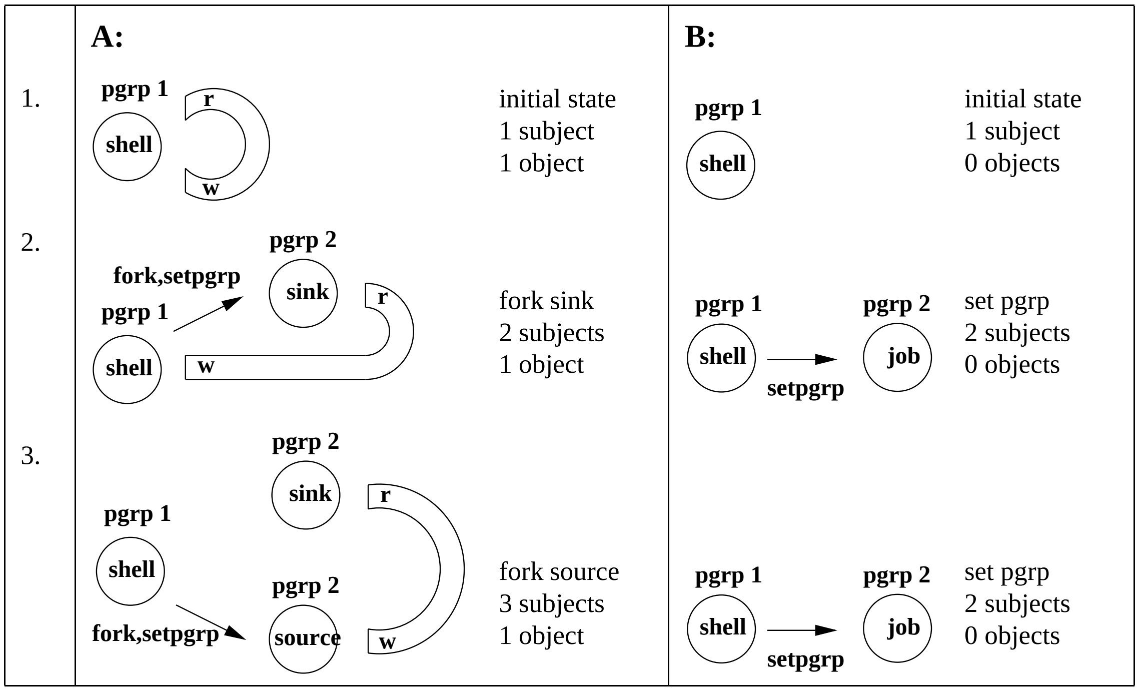

Figure 5. Simplified Representation of Shell Pipeline Behavior.

more liberal rule expressed in terms of the pipe users' levels instead of job boundaries could avoid this problem. However, its enforcement would require far more additional data structures than the Unnamed Pipe Usage Rule, which takes advantage of the existing job encapsulation. Based on the quality of protection concerns discussed in section 3, we have decided to employ the Unnamed Pipe Usage Rule in LOMAC, at least until we encounter a discouraging job-related failure. Regardless of the rule we use to handle unnamed pipes, the mapping of the subject concept to the job abstraction is essential to the avoidance of the Self-Revocation Problem in shell pipelines, since it ensures that all processes in a given job are always at the same level. The diagram in figure 5B shows the same C shell job-creation algorithm as figure 5A, except that it pictures only those aspects of the algorithm that are pertinent when using the improved application of the Low Water-Mark model's concepts with the Unnamed Pipe Usage Rule. The unnamed pipe is not shown in the diagram, since the improved application of model concepts does not consider it to be an object. Only two subjects exist at the end of the algorithm: the shell and the new job it has created. The shell pipeline Self-Revocation Problem diagrammed in figure 4 cannot occur in the improved prototype, since the entire job is considered a single subject, and the use of unnamed pipes is invisible to the Low Water-Mark mechanism.

Although shell pipelines are the most common situation in which the Self-Revocation Problem can lead to failures in application functionality, it is not the only one. IPC done via shared memory abstractions such as semaphores and writable shared memory segments are vulnerable as well. A solution to the Self-Revocation Problem similar to the Unnamed Pipe Usage Rule is not practical in the case of shared memory abstraction IPC. Unlike shell pipeline IPC, shared memory abstraction IPC cannot be neatly contained inside an operating system abstraction, such as the job. A rule that constrained shared memory abstraction IPC to members of the same job would be compatible with those applications which use shared memory within a single job, perhaps to simulate multi-threading. However, it would be incompatible with those applications which use shared memory abstractions to support IPC between client and server components executing in separate jobs. This potential for incompatibility would be unacceptable according to our Compatibility Goals.

Consequently, the improved version of the LOMAC prototype uses a different strategy to support shared memory abstraction IPC while maintaining integrity protection: it extends the Low Water-Mark model by introducing the notion of a "group". The group notion allows the extended Low Water-Mark model to treat all those subjects that share a particular shared memory abstraction as a unit, without consideration for the read and write IPC operations occurring among them. Just as the encapsulation provided by the operating system's job abstraction allows the improved LOMAC prototype to avoid occurrences of the Self-Revocation Problem in shell pipelines, the encapsulation provided by the group notion allows the LOMAC prototype

A:

Original Definitions:

S :	the universal set of subjects;
O :	the universal set of objects;
I :	the universal set of integrity levels;
$\underline{il}$:	a function $S \cup O \;\rightarrow\; I$ defining the integrity level associated with each subject and object;
$\underline{leq}$:	a reflexive, symmetric, and transitive relation (a subset of $I \;\times\; I$) that defines the "less-than-or-equal" relationship on I;
$\underline{min}$:	a function $POWERSET(I) \;\rightarrow\; I$ returning the greatest lower bound of the specified subset of I; more formally $(i_1 = \underline{min}(x) \;\Rightarrow\; (i_1 \in x \;\wedge\; \forall i_2 \in x,\; i_1 \;\underline{leq}\; i_2))$;
$\underline{o}$:	a relation (a subset of $S \;\times\; O$) that defines the fact that a subject, $s \in S$, has observed an object, $o \in O$, $s \;\underline{o}\; o$;
$\underline{m}$:	a relation (a subset of $S \;\times\; O$) that defines the fact that a subject $s \in S$, has modified an object, $o \in O$, $s \;\underline{m}\; o$;
$\underline{i}$:	a relation (a subset of $S \times S$) that defines the fact that a subject $s_1 \in S$, has invoked another subject, $s_2 \in S$, $s_1 \;\underline{i}\; s_2$. LOMAC interprets invocation as signal and setpgrp operations.

Original Axioms:

(A1)	$\forall s \in S,\; o \in O,\; s \;\underline{o}\; o \;\Rightarrow\; \underline{il}'(s) \;=\; \underline{min}\{\; \underline{il}(s),\; \underline{il}(o) \;\}$	(originally A3.1, revised)
(A2)	$\forall s \in S,\; o \in O,\; s \;\underline{m}\; o \Rightarrow \underline{il}(o) \;\underline{leq}\; \underline{il}(s)$	(originally A3.2, retained)
(A3)	$\forall s_1, s_2 \in S,\; s_1 \;\underline{i}\; s_2 \Rightarrow \underline{il}(s_2) \;\underline{leq}\; \underline{il}(s_1)$	(originally A3.3, retained)

B:

Additional Definitions:

$\underline{a}$:	a reflexive and antisymmetric relation (a subset of $S \;\times\; S$) that defines the fact that a subject, $s_1 \in S$, has attached to a shared memory abstraction to which another subject, $s_2 \in S$ is already attached, $s_1 \;\underline{a}\; s_2$;
$\underline{s}$:	a reflexive, symmetric, and transitive relation (a subset of $S \;\times\; S$) that defines the fact that one subject, $s_1 \in S$, shares at least one shared memory abstraction with another subject, directly or indirectly, $s_2 \in S$, $s_1 \;\underline{s}\; s_2$;
$\underline{group}$:	a function $S \;\rightarrow\; POWERSET(S)$ returning the set containing the specified subject, along with all those other subjects that share, directly or indirectly, a shared memory abstraction with the specified subject. More formally: $\underline{group}(s_1) \;=\; \{s_2 : s_2 \;\underline{s}\; s_1\}$

Table 1. Axioms and Definitions for the Original (A) and Extended (B) Low Water-Mark Models

to avoid occurrences of the Self-Revocation Problem during shared memory abstraction IPC.

The group notion also provides a secondary, practical benefit. As described in section 3, LOMAC uses Interposition at the operating system's system call interface to detect operations corresponding to the Low Water-Mark model's notion of observe and modify. Unfortunately, in the case of shared memory abstractions, these operations are not visible at the system call interface, and consequently cannot be detected by LOMAC. Since the group notion makes observe and modify operations on shared memory abstractions irrelevant to the extended Low Water-Mark model, it obviates LOMAC's need to detect them.

We will now present the group extension in formal terms, and argue that the extended Low Water-Mark model provides integrity protection at least as strong as the original model. We use a notation similar to that used in the original model's description [5]. Table 1A contains the definitions and axioms for the original Low Water-Mark model. Our extended model retains all of these, except for axiom (A1), which we will revise to accommodate our new group concept. We use $\rightarrow$ to indicate function mapping, and $\Rightarrow$ to indicate implication. We use $\underline{il}$ to indicate a level before an event, and $\underline{il}'$ to indicate a level after an event. For example, in Axiom (A1), $\underline{il}(s)$ represents the level of s before it observed o, and $\underline{il}'(s)$ is its level after it observed o.

Table 1B contains the three definitions required to extend the Low Water-Mark model. The first is the $\underline{a}$ relation, which can be applied to the operating system services which allow a subject to attach (acquire access) to a shared

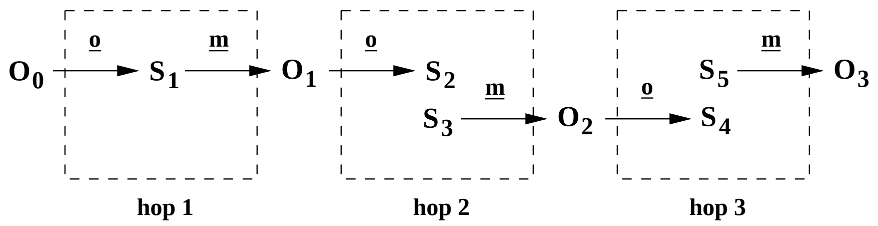

Figure 6. Information Transfer Path

memory abstraction. In order to avoid the Self-Revocation Problem, we wish to avoid representing the actual shared memory abstractions as objects. Consequently, we describe $\underline{a}$ as a relation that links one subject to another in a sharing relationship, rather than a relation that links a subject to some kind of object. The use of the $\underline{a}$ relation is governed by Axiom (A4), which prevents subjects at lower levels from corrupting subjects at higher levels by attaching to them. More precisely:

Axiom (A4) $\forall s_1, s_2 \in S,\ s_1\ \underline{a}\ s_2\ \Rightarrow\ \underline{il}(s_2)\ \underline{leq}\ \underline{il}(s_1).$

Our use of the $\underline{a}$ operation is pessimistic: it does not distinguish between harmless read-only attachments and dangerous read-write attachments. As described in table 1B, the $\underline{a}$ relation is antisymmetric. We require this property to distinguish between the situations $s_1\ \underline{a}\ s_2$ and $s_2\ \underline{a}\ s_1$. In the case where s_1 and s_2 have different levels, only one of these two situations is allowable in a correct system according to Axiom (A4). However, in our theorems, we wish to use the $\underline{group}$ function to describe the set of all subjects that share memory with a given subject, regardless of which subjects were attachers and which were attachees. In order to support a succinct definition of $\underline{group}$, table 1B defines a second relation, $\underline{s}$, which has the symmetric and transitive properties. The use of the $\underline{s}$ relation is governed by Axiom (A5), which states that once one subject is attached to another, it becomes part of a larger group that shares memory through a graph of direct and indirect bidirectional attachments. More precisely:

Axiom (A5) $\forall s_1, s_2 \in S,\ s_1\ \underline{a}\ s_2\ \Rightarrow\ s_1\ \underline{s}\ s_2.$

Table 1B's third definition is for the $\underline{group}$ relation, which describes groups of subjects that share memory, defined in terms of the $\underline{s}$ relation.

The extended Low Water-Mark model provides integrity protection by ensuring that all subjects in a particular group remain at the same level. Axiom (A6) ensures that higher-levelled subjects that join a lower-levelled group are demoted to bring them into compliance. More precisely:

Axiom (A6) $\forall s_1, s_2 \in S,\ s_1\ \underline{a}\ s_2\ \Rightarrow$
$\underline{il}'(s_1)\ = \underline{min}\{\ \underline{il}(s_1),\ \underline{il}(s_2)\ \}.$

Similarly, the extended Low Water-Mark model ensures that all subjects in a particular group are demoted as a unit whenever one member of the group observes an object beneath the group's level. More precisely:

Axiom (A7) $\forall s_2 \in \underline{group}(s_1), o \in O,\ s_1\ \underline{o}\ o\ \Rightarrow$
$\underline{il}'(s_2)\ =\ \underline{min}\{\ \underline{il}(s_1),\ \underline{il}(o)\ \}.$

Axiom (A7) revises and replaces the original Axiom (A1) in the extended Low Water-Mark Model. The operation of Axiom (A7) is effectively identical to that of Axiom (A1), except that Axiom (A7) demotes a group of subjects as a unit where Axiom (A1) demotes only a single subject. Due to this similarity, the original Low Water-Mark Model remains as a special case of the extended Low Water-Mark model where no subject shares memory with another. This special case can be described more precisely as:

Special Case Proposition (P1) $\forall s \in S,$
$\underline{group}(s)\ =\ \{\ s\ \}.$

Corollary (C1) The survival of the original Low Water-Mark model in the special case where Proposition (P1) is true can be demonstrated by deriving the original Axiom (A1) from its replacement (A7). In the special case described by Proposition (P1), all groups contain only one subject. Consequently, given Proposition (P1), the two subjects described in Axiom (A7) as $s_2 \in \underline{group}(s_1)$ are actually the same subject, which we may name s. By replacing references to s_1 and s_2 in Axiom (A7) with the single reference s, we derive Axiom (A1). Therefore, the demotion behavior of the extended Low Water-Mark model is identical to the demotion behavior of the original Low Water-Mark model in the special case described by Proposition (P1).

We now argue that the extended Low Water-Mark provides at least as much integrity protection as the original Low Water-Mark model by applying the same test applied to the extended model as was applied to the original in [5]. We define the notion of information transfer path, and show that information cannot flow from lower to higher levels via such a path.

The diagram in figure 6 presents the structure of a path in terms of the extended Low Water-Mark model concepts. It shows a series of subjects, numbered s_1 through s_5, transferring data serially through a series of objects, numbered

A:

Additional Definitions:

$\underline{hop}$:	a relation (a subset of $O \times O$) defining individual “hops” in an information transfer path, where information is transfered from one object, $o_1 \in O$, to another $o_2 \in O$, $o_1\ \underline{hop}\ o_2$. More precisely: $\forall o_1, o_2 \in O,\ o_1\ \underline{hop}\ o_2 \ \Rightarrow\ \exists s_1, s_2 \in S \mid (\ s_1\ \underline{o}\ o_1\ \wedge\ s_2\ \underline{m}\ o_2\ \wedge\ s_2 \in \underline{group}(s_1)\ )$.
$\underline{path}$:	a relation (a subset of $O \times O$) defining a sequence of $n+1$ objects $< o_0, ..., o_i, ..., o_n > \in O$ connected in a series of n hops $o_i\ \underline{hop}\ o_{i+1}$, where $0 \leq i < n$, called an information transfer path. More precisely: $o_0\ \underline{path}\ o_n \ \Rightarrow\ \exists < o_0, ..., o_i, ..., o_n > \in O \mid \forall o_{0 \leq i < n}\ :\ o_i\ \underline{hop}\ o_{i+1}$

B:

Lemma (L1) A hop does not allow the transfer of information from a lower-levelled object to a higher-levelled one. More precisely: $\forall o_1, o_2 \in O,\ o_1\ \underline{hop}\ o_2 \ \Rightarrow\ \underline{il}(o_2)\ \underline{leq}\ \underline{il}(o_1)$.

Proof:

1.	$(\ s_1\ \underline{o}\ o_1\ )\ \wedge\ (\ s_2\ \underline{m}\ o_2\ )\ \wedge\ (\ s_2 \in \underline{group}(s_1)\ )$	given
2.	$\underline{il}'(s_2)\ =\ \underline{min}\{\ \underline{il}(s_1),\ \underline{il}(o_1)\ \}$	1, (A7)
3.	$\underline{il}'(s_2)\ \underline{leq}\ \underline{il}(o_1)$	2, definition of $\underline{min}$
4.	$\underline{il}(o_2)\ \underline{leq}\ \underline{il}'(s_2)$	1, (A2)
5.	$\underline{il}(o_2)\ \underline{leq}\ \underline{il}(o_1)$	3,4, transitive property of $\underline{leq}$

Step 5 is the lemma; the lemma is correct.

C:

Theorem (T1) The extended Low Water-Mark model prevents data from flowing from low-levelled objects to higher-levelled objects through an information transfer path, just as the original model does. More precisely: $\forall o_0, o_n \in O,\ o_0\ \underline{path}\ o_n \ \Rightarrow\ \underline{il}(o_n)\ \underline{leq}\ \underline{il}(o_0)$.

Proof:

1. First, consider the base case of an information transfer path containing only 1 hop ($n = 1$):

	$o_0\ \underline{hop}\ o_1$	base case assumption, definition of $\underline{path}$
2.	$\underline{il}(o_1)\ \underline{leq}\ \underline{il}(o_0)$	1, (L1)

3. Assuming theorem (T1) is correct for the case of an information transfer path with j hops ($n = j$), consider the inductive case of an information transfer path with $j + 1$ hops ($n = j + 1$):

	$\underline{il}(o_j)\ \underline{leq}\ \underline{il}(o_0)$	inductive case assumption
4.	$o_j\ \underline{hop}\ o_{j+1}$	3, definition of $\underline{path}$
5.	$\underline{il}(o_{j+1})\ \underline{leq}\ \underline{il}(o_j)$	4, (L1)
6.	$\underline{il}(o_{j+1})\ \underline{leq}\ \underline{il}(o_0)$	3, 5, transitive property of $\underline{leq}$

Step 6 indicates that, given that the theorem is correct for a path with j hops, it is also correct for a path with $j + 1$ hops. According to the first principle of mathematical induction, this fact, coupled with the result of step 2, indicates that the theorem is correct for all information transfer paths with a finite length of 1 or more hops.

Table 2. Definitions for Information Transfer Paths (A), Lemma (L1) (B), and Theorem (T1) (C).

o_0 through o_3. The subjects accomplish the transfer in a series of three hops, shown enclosed in dashed boxes. In the first hop, subject s_1 observes the object to its left, acquiring information, and subsequently modifies the object to its right, transferring the information from one to the other. This pattern is repeated in the second and third hops, except that in these cases, one subject does the observing, and another subject in its group does the modifying. In the second and third cases, we assume that the data is passed from the observing subject to the modifying subject via shared memory abstraction IPC. In these cases, the flow within the group via read/write memory operations is invisible to the model. Furthermore, as in the original description of information transfer path, we assume that the observe and modify operations are atomic, and occur serially, in order from left to right across all three hops in the diagram.

We demonstrate the integrity protection provided by the extended Low Water-Mark model by showing that the object at the beginning of the path is always at least as high in level as the object at its end. That is, that the model prevents the spread of data corruption by preventing information from flowing "upstream" from objects at low levels to objects at a higher levels. Table 2A contains precise definitions of our notions of hop and path. Using these definitions, we first show with Lemma (L1) in table 2B that the extended Low Water-Mark prevents upstream flows in individual hops. Next, with Theorem (T1) in table 2C, we use Lemma (L1) to show inductively that upstream flows are not possible in paths made up of one or more hops.

Corollary (C1) shows that the original Low Water-Mark model's demotion behavior is a special case of the extended model's behavior. Theorem (T1) demonstrates that the extended Low Water-Mark model prevents the upstream spread of corruption through information transfer paths in the same manner as the original Low Water-Mark model. Based on this evidence, we assert that the extended Low Water-Mark model provides at least as much integrity protection as the original.

6 Related work

LOMAC is not the only experiment that tests a hypothesis concerning the applicability of integrity protection extensions to deployed COTS systems. The Janus [13] and TCP Wrappers [30] projects are both examples of Interposition-based access control extensions for COTS UNIX environments which have a relatively low Partial Compatibility Cost. TCP Wrappers focuses on access made by remote clients to local services (daemons). Janus, on the other hand, focuses on mediating system calls made by local processes, just as LOMAC does. Janus is capable of enforcing policies based on a variety of AMM models, while LOMAC enforces only Low Water-Mark policies. However, based on the performance results of other kernel-resident mechanisms that Interpose their control at the system call interface, such as SLIC [11] and Generic Software Wrappers [10], we expect the LOMAC to degrade application performance less than Janus, which operates entirely in user-space. SLIC and Generic Software Wrappers provide generalized LKM and Interposition functionality that could be used to implement a prototype like LOMAC. Their use might increase Partial Compatibility Cost, however, since an adopter would first have to configure the SLIC or Generic Software Wrappers system before taking advantage of LOMAC's configuration-free Default Policy.

Not every attempt to apply kernel-resident access control is an attempt to explore our hypothesis concerning its applicability to existing COTS systems. Some projects, such as DTE [3], KSOS [20], RSBAC [24], and UCLA Secure UNIX [26] have applied kernel-resident access control to the COTS UNIX environment by modifying the operating system at the source code level, to the extent of introducing a new architecture in some cases. Unlike LOMAC, these projects emphasize the provision of high-quality protection before the minimization of Total Compatibility Cost. Their approach requires the expensive replacement of deployed COTS operating systems in order to provide protection in production environments. The cost is reduced somewhat by the similarity between the original and modified operating systems from the perspective of user-space applications. Also, DTE reduces this cost further by retaining the original file system format, allowing the use of the unmodified operating system after a reboot. However, the cost is still greater than LOMAC's. In exchange for this increased cost, these projects gain a higher quality of protection and assurance than LOMAC. The LOMAC experiment's emphasis on compatibility before quality of protection and assurance is not a suggestion that assurance is not required in todays COTS environments. To the contrary, the circumstances we described to justify the need for LOMAC in section 1 can also be taken as arguments for the need for greater assurance [19]. LOMAC is a means of testing our hypothesis. Regardless of the outcome of our experiment, it will not provide evidence against the argument that assurance and high-quality protection can overcome significant Total Compatibility Cost and gain acceptance in the COTS environment given sufficient time.

7 Conclusions

The LOMAC project is an ongoing experiment to test the hypothesis that a kernel-resident access control mechanism can be widely accepted as an extension to deployed COTS systems provided that it provides some useful integrity protection at a sufficiently low Total Compatibility Cost. As of this time, the prototype itself contains only enough functionality to prove the viability of our application of the Low Water-Mark model's concepts to the COTS Linux environment. It is not yet mature enough to deploy in production environments. Although implementation continues, we discuss the quality of protection and Total Compatibility Cost of LOMAC as it exists today. As described in section 3, LOMAC's architecture trades quality of protection for decreased Partial Compatibility Cost through its use of an LKM and Interposition. The choice of the the Low Water-Mark model over the other Access Matrix Models described in section 2 also involves such a tradeoff. The Low Water-Mark model has two significant disadvantages in terms of protection, when compared to models such as Type Enforcement and DTE.

First, the Low Water-Mark model's ability to confine subjects according to the Principle of Least Privilege is limited [27, 31]. The scenario in section 4 described how LOMAC's Default Policy protects level 2 objects from a compromised level 1 subject. It is important to note, however, that a compromised subject is all-powerful at its own level and below. The Default Policy provides no protection for other subjects and objects at level 1. This deficiency could be addressed by the addition of Integrity Categories to complement the model's existing level concept [18]. However, the assignment of Integrity Categories to existing objects at configuration time could require too much knowledge of the behavior of individual users and daemons to be accomplished automatically in the Default Policy. The potential of this solution in the context of LOMAC remains to be explored.

Second, the Low Water-Mark model cannot implement Assured Pipelines, and must rely on the execution of formally verified programs to provide integrity protection in some situations [6]. The Default Policy avoids this reliance, since formally verified programs are scarce in COTS Linux environments. Consequently, it cannot protect the integrity of the system log object. The log object must reside at level 1 so that all subjects can write their log messages to it. However, as described above, level 1 objects are vulnerable to all compromised subjects. This deficiency can be addressed to some extent through the use of Partially Trusted Subjects [17]. These are subjects which, while executing a formally verified program, receive a limited set of additional privileges that enable them to perform some exceptional task. By limiting the set of additional privileges, rather than allowing the subject to operate without any restrictions, this approach limits the number of program properties that must be formally verified. We have applied this technique to the system log object problem with some success. However, the Low Water-Mark model's limited ability to enforce the Principle of Least Privilege makes it difficult to define limited sets of additional privileges, making it proportionally difficult to reduce the required amount of formal verification work in a meaningful way.

Despite these shortcomings, LOMAC manages to provide some useful integrity protection at a low Partial Compatibility Cost, as described in sections 2 and 4. Section 5 describes our efforts to avoid runtime situations in which LOMAC causes application failures by mapping the model's subject concept to the Linux job abstraction, and a grouping of subjects based on shared memory abstractions. Our experience to date with the LOMAC prototype and its Default Policy suggests that LOMAC avoids such failures well enough to allow productive work and a low Total Compatibility Cost. Although the impracticality of formally verifying the failure-free behavior of all application programs will probably prevent us from ever claiming that low Total Compatibility Cost can be achieved in the general case, such a claim may be practically verifiable in the special case of the Default Policy.

As described in section 5, the Self-Revocation Problem can cause an application to fail only when LOMAC demotes the subject executing the application. Under the Default Policy, the vast majority of applications execute at the lowest level, 1, and are consequently not demotable and immune to the Self-Revocation Problem. Applications which execute at level 2 are far fewer, and are mainly administrative in nature. An analysis of the set of administrative applications, or at least a minimal essential set of these applications, may be tractable. COTS UNIX distributions designed to fit on a single 1.44MB floppy disk, such as PicoBSD, contain such minimal essential sets. This analysis may allow us to characterize LOMAC's Total Compatibility Cost while enforcing the Default Policy.

8 Notes

The author would like to thank Lee Badger, Hongyu Lu, and Wayne Morrison for many helpful discussions. This work benefitted greatly from their review, insight, and advice.

PicoBSD is a version of FreeBSD, a Registered Trade Mark of FreeBSD, Inc. and Walnut Creek CDROM. Solaris is a Registered Trade Mark of Sun Microsystems, Inc. UNIX is a Registered Trade Mark of the X/Open Company, Ltd. Windows NT is a Registered Trade Mark of the Microsoft Corporation.

This is NAI Labs report #0775.

References

[1] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[2] J. Arnold, W. Havener, and J. Singh. Final Evaluation Report Trusted Information Systems, Inc. Trusted XENIX version 3.0. Technical Report CSC-EPL-92/001, National Computer Security Center, Fort George G. Meade, Maryland, April 1992.

[3] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A Domain and Type Enforcement UNIX Prototype. *USENIX Computing Systems*, 9(1):47–83, Winter 1996.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[5] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.

[7] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, California, May 1989.

[8] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California, April 1987.

[9] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.

[10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Oakland, California, May 1999.

[11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[12] B. D. Gold, R. R. Linde, R. J. Peeler, M. Schaefer, J. F. Scheid, and P. D. Ward. A security retrofit of VM/370. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 335–344, Montvale, New Jersey, 1979.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, California, July 1996.

[14] G. S. Graham and P. J. Denning. Protection - Principles and Practice. In *AFIPS Conference Proceedings Volume 40 Spring Joint Computer Conference*, Montvale, New Jersey, 1972.

[15] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8), August 1976.

[16] W. Joy. An Introduction to the C shell. In *4.4BSD User's Supplementary Documents*, chapter 4. O'Reilly & Associates, Inc., 1994. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[17] T. M. P. Lee. Using Mandatory Integrity to Enforce "Commercial" Security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 140–146, Oakland, California, April 1988.

[18] S. B. Lipner. Non-Discretionary Controls for Commercial Applications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 2–10, Oakland, California, April 1982.

[19] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrel. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Arlington, VA, October 1998.

[20] E. J. McCauley and P. J. Drongowski. KSOS – The Design of a Secure Operating System. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 345–353, Montvale, New Jersey, 1979.

[21] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley Longman, Inc., 1996.

[22] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the 13th Annual Computer Security Applications Conference*, San Diego, California, December 1997.

[23] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, pages 229–243, Seattle, Washington, October 1996.

[24] A. Ott. Regel-basierte Zugriffskontrolle nach dem Generalized Framework for Access Control-Ansatz am Beispiel Linux. Master's thesis, Universitat Hamburg, Fachbereich Informatik, 1997.

[25] M. Petkac, L. Badger, and W. Morrison. Security Agility for Dynamic Execution Environments. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 377–390, Hilton Head, SC, January 2000.

[26] G. J. Popek, M. Kampe, C. S. Kline, A. Stoughton, M. Urban, and E. J. Walton. UCLA Secure UNIX. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 355–364, Montvale, New Jersey, 1979.

[27] J. H. Saltzer and M. D. Schroder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE Vol. 63(9)*, pages 1278–1308, September 1975.

[28] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, DC, August 1999.

[30] W. Venema. TCP Wrapper: Network Monitoring, Access Control, and Booby Traps. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 85–92, Baltimore, MD, September 1992.

[31] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, pages 21–36, San Jose, California, July 1996.