A Privilege Mechanism for UNIX System V Release 4 Operating Systems

Charles Salemi, Suryakanta Shah, Eric Lund – UNIX System Laboratories, Inc.

ABSTRACT

Any multi–user, multi–tasking operating system, such as the UNIX SVR4 Operating System, must provide protection mechanisms that prohibit one user from interfering with another user, or limit the execution of certain system operations that affect critical system resources. These protection mechanisms must also provide the ability to override these restrictions, commonly referred to as *privilege*. For over twenty years, UNIX–based operating systems have had one such privilege, called "root" or "super–user" which is signified by a process whose effective user ID is 0. The "super–user" has the ability to override the restrictions imposed by these protection mechanisms. In the UNIX System V Release 4 Enhanced Security product this single, omnipotent, privilege is divided into a set of discrete privileges designed to assure that sensitive system services execute with the minimum amount of privilege required to perform the desired task.

This paper describes the *privilege control mechanism* implemented as part of the UNIX System V Release 4.1 Enhanced Security (SVR4.1ES) product. The SVR4.1ES privilege control mechanism separates the privilege mechanism from the access control mechanism, it provides for fine grained control over sensitive operation access by users, and it controls the propagation of privilege from one process to another. Our goals also include accommodating multiple privilege control mechanisms within the UNIX System V kernel. These privilege mechanisms can be "plugged" into the kernel through well defined interfaces, much the same way as UNIX file systems are currently added to the kernel.

Introduction

The granularity of the "root" privilege is too coarse for a system that dictates fine granularity in the assignment of privilege and the ability to control the assertion of privilege throughout the execution of a process. The SVR4.1ES privilege control mechanism is designed to meet the the following goals:

- to make the privilege control mechanism a separate, loadable module,
- to make minimal changes to existing kernel code,
- to separate the privilege control mechanism and access mechanism,
- to make the file–based privileges file system independent,
- to preserve UNIX System V compatibility.

Our approach was to use the concept of privilege sets that are assigned to both processes and executable files. The concept of privilege sets is not new. We are aware of two papers discussing the implementation of privilege control mechanisms that also made use of privilege sets [1][2].

The feature that makes the implementation of our privilege control mechanism unique is our fourth goal: file system independence. This feature makes it easy for file system developers to make use of our privilege mechanism. In addition, existing file system types function properly without any modifications. This paper describes the kernel interface routines we defined that are required to support the concept of separate privilege modules. It also describes the major differences between the two privilege modules supported under SVR4.1ES.

Problem Definition

Before we begin our discussion, it is necessary to provide a definition for the term privilege. Simply stated, "privilege" is the ability to override restrictions imposed by protection mechanisms. For example, a process[1] requires privilege to change the system date, mount or unmount a file system, or modify file attributes (if not the owner of the file) because these operations are restricted by the protection mechanisms.

Our mission from the start was explicit: add the capability of supporting discrete privileges assigned to each sensitive system operation in the kernel. This was necessary to provide a privilege policy that based propagation of privilege on attributes other than the effective user ID. However, we also had the requirement to remain compatible with previous versions of the UNIX Operating System by maintaining the concept of a "super–user."

[1] The term *process* is defined in [3] as an instance of an *executable file* in execution. An executable file is defined to be a *program*.

The privilege control mechanism we chose for UNIX SVR4.1ES removes the omnipotence of effective user ID 0 by defining a set of discrete privileges, each one used to override individual restrictions imposed by *sensitive* system operations. It also defines an inheritance policy, by associating privileges with executable files, to control propagation of privileges. In addition, it gives flexibility to designers of privilege modules by providing well–defined interface routines that are general enough to support a privilege policy based on any process attribute.

Defining the Kernel Privilege Interface

Why a Loadable Privilege Control Mechanism?

Our first goal was to make the privilege control mechanism "plug compatible" so administrators could choose the privilege policy they preferred at system configuration time.

There were quite a few "religious" debates surrounding this particular goal. One school of thought argued very strongly that the operating system should be configured with a privilege policy that was in effect for the entire life of the system. It was felt that any deviation from the privilege policy only meant that the potential existed for more things to go wrong.[2] Another school of thought disagreed contending that the system went through one or more phases before the entire security policy was in place. They felt that while the system was transitioning to the final phase, the privilege policy should be based on the effective user ID and once the transition to a secure system was complete, change the privilege policy to one that was file–based.

We decided to go with the first interpretation for several reasons:

- The privilege mechanism has always been a "point of attack" with respect to security break–ins. Adding complexity for determining which privilege policy was in effect appeared to weaken the privilege mechanism.
- Using the modular approach and defining separate privilege policies does not preclude the use of a privilege control module that behaves in the manner described by the second interpretation.
- It is flexible enough to allow a designer of a privilege module the option of basing the privilege policy on other attributes of the process such as the effective or real group ID, the sensitivity label of the process, etc. with minor modification to existing kernel source code. In this case, only the privilege module source code would require modification. The *privilege request* checks in the kernel would remain undisturbed.

[2] We are all acutely aware of *Murphy's Law.*

Therefore, to accomplish this goal, a general privilege interface was required that would allow for the flexibility of specifying the privilege policy desired for a particular privilege module.

General Privilege Interface

Each concept below has a corresponding routine in a specific privilege control mechanism. The behavior for each routine is relative to the privilege policy enforced by that privilege mechanism.

Initialization

To begin with, there must be a way to initialize the system privilege mechanism at system initialization time. Functions that might be performed by this procedure include: allocation of data structures, installation of "bootstrap" data, or initialization of local static variables.

Privilege Requests

A procedure is needed to determine if a privilege requested by a process is contained in the set of privileges currently in effect for that process. This procedure is the *policy maker* of any privilege module because it makes the decision to grant or deny privilege when a request is made.

Propagation

Another procedure is required to calculate the privileges for new processes. This procedure provides the privilege propagation model for the module.

Process Privilege Manipulation

A procedure is required to allow a process to count, set, clear, and retrieve its privilege sets.

Process Privilege Recalculation

A procedure is required that will recalculate process privileges whenever the effective user ID is modified.[3]

File Privilege Manipulation

Another procedure is required to assign and retrieve privilege associated with a file. This allows software external to the policy module to identify files that are included in the system. The system is made up of files that have been analyzed and determined to securely use certain privileges.

When retrieving privileges this procedure must retrieve them from the same data structure where they are stored for use by the propagation procedure.

File Privilege Removal

A routine is required to remove the privilege information associated with system files whenever media containing "trusted" files is removed from the system. This prevents the introduction of "Trojan Horses" when a new file system is mounted on the same mount point.

[3] This procedure is required to maintain compatibility with ID–based privilege modules.

Privilege Mechanism Information

A procedure is required to allow a process the ability to retrieve specific information regarding the privilege module. This information might be in the form of how many privilege sets are supported by the privilege mechanism, what the names of those sets are, the number of privileges contained in each set, which privilege mechanism is in effect, etc. Currently, two privilege modules exist that make use of the privilege interface routines. They are:

SUM module provides a privilege policy that supports a set of discrete privileges and the "super–user" concept. This module is considered to be ID–based since the propagation of privilege is based on the effective user ID of the process.

LPM module provides a privilege policy that supports a set of discrete privileges and an inheritance policy.[4] This module is considered to be *file–based* since the propagation of privilege is based on the inheritance policy that uses the maximum privilege set of the current process and the set (or sets) of discrete privileges assigned to the executable program file being `exec`'ed.

Definitions

The following is a list of the privilege sets required and their definitions. Also indicated is which privilege sets are used by the two privilege modules supported in SVR4.1ES.

Process Privilege Sets

A process can have several privilege sets.[5] These are used to control the assignment of privilege throughout the life of the process. Currently, both privilege modules support the following process privilege sets:

- **Maximum privileges**: the set of privileges held in trust for a process. This is the set of privileges required by a process to complete all of the program tasks.
- **Working privileges**: the set of privileges necessary to complete a particular program task in a process.

The following rules apply to the maximum and working process privilege sets:

- The working set is always a subset of the maximum set.
- The working set may be altered at any time. The new working set must be a subset of the maximum set.
- The maximum set may be altered at any time. The new maximum set is a subset of the previous maximum set.

  When the maximum set is altered, privileges in the working set that are not in the new maximum set are removed from the working set.
- Process privilege sets are unchanged during a `fork`(2) operation. The privilege sets of the child process are identical to those of the parent.

File Privilege Sets:

A file can have several privilege sets.[6] File privilege sets are used to establish the privileges of the *new*[7] process when the file is executed. The following file privilege set is currently supported for both privilege modules:

- **Fixed privileges**: a set of privileges always given to the new process independent of the process privileges of the invoking process.

The following file privilege set is supported only in the *file–based* privilege module:

- **Inheritable privileges**: a set of privileges given to the new process only if the privilege was in the maximum set of the invoking process.

The following rules apply to file privilege sets:

- File privilege sets can only be assigned to ordinary, executable file types.
- All privileges associated with a file are removed when the file is modified.

Isolation of the Privilege Mechanism Code in the Kernel

The second and third goals involve the isolation of the privilege kernel mechanism in the kernel code. The privilege mechanism was *tightly–coupled* with the access mechanism because a process with effective user ID 0 had unlimited access. This association had to be severed. When we did this, however, we wanted to keep the modification of existing kernel source code to a minimum.

Minimizing Kernel Changes

To achieve the second goal, minimizing kernel changes, the kernel source code was analyzed. Two mechanisms for determining privilege in the kernel were identified:

1. calling the internal kernel routine `suser()` to check if the effective user ID was 0, and
2. explicitly checking whether or not the effective user ID was 0.

Existing routines in the kernel that either called the `suser()` routine or checked the effective user ID

[4] Defined in the *privilege propagation* interface routine.
[5] The current implementation limit is 255.

[6] File privilege sets are also limited to 255.
[7] The term *new* is used here rather than child because a process may be *exec*'ed directly over the calling process without invoking the `fork`(2) system call. The privilege control mechanism will work properly because the privilege setting for the new process is done before the new process is executed.

had to be modified to call the *privilege request* routine defined in the privilege module.[8]

Separation of Privilege and Access Mechanisms

Our third goal, separation of the privilege mechanism in the kernel, does not affect the current *setuid* and *setgid* mechanism. A process using the *setuid/setgid* mechanism works as it currently does by only allowing a user access to files that they might not normally have based on the file access bits. Removing the "omnipotence" of user ID 0 in the kernel means that the file access permission works exactly the same for user ID 0 as any other user ID.

Associating Privilege with a File

Privilege needs to be associated with files that are part of the system. One method of associating privilege with a file is to store the privilege in the *inode* of the file. This method provides a protection from users because of the well-defined system interface for accessing the *inode*. However, this implementation is limited because file system types already in use can not take advantage of a modular privilege implementation.

Because of this limitation another method was needed to achieve the fourth goal of file system independence. This method must provide similar protection from users that the *inode* scheme provides, i.e., it should be accessed only through a well-defined interface. Also, to achieve the first goal, it has to fit well into the modular privilege interface.

To meet these goals a memory-based kernel table is used to define the privileged commands in the operating system. Using a memory based table provides the following benefits:

- File system independence. The privilege control mechanism can work across file system types.
- The number of privileged commands can be reduced, or expanded, independent of the privilege policy in use.
- Privilege can never be *imported* from other media since the privilege information is not part of the file attributes.

ID-Based Mechanism Feature

Using the *setuid-on-exec* mechanism[9] and setting the owner of an executable file to *root* is still supported in the ID-based privilege mechanism because that privilege policy supports the concept of "super-user." However, this mechanism also supports *fixed* privileges on executable files. Therefore, it is possible to assign to an executable file only those privileges *required* to complete all its tasks instead of giving it *all* privileges via the *setuid-on-exec* mechanism.

Compatibility

Our fifth and final goal was to preserve compatibility with previous releases of the UNIX Operating System. As mentioned previously, there are two privilege modules supported in SVR4.1ES.

Most of the routines defined for each privilege module are the same because of the similarities between the two privilege policies. The major differences occur in the *propagation* routine and the *process privilege recalculation* routine.

Privilege Propagation

The *propagation* routine determines how privilege is propagated for the particular privilege policy in use.

File-Based Propagation Model

The following model is in effect for the *file-based* privilege module:

> A privilege can be acquired only if the privilege exists in the *maximum set* of the *calling* process or is in the *fixed set* of a file. A process with an empty *maximum set* can **never** pass a privilege to another process. The computation of the new *maximum* and *working* sets is done in the `exec()` kernel code:

Step [A] The *maximum* set of the calling process is **intersected** with the *inheritable* set associated with the program file being executed.

Step [B] The result of Step [A] is **union**ed with the *fixed* set associated with the program file being executed to form the new *maximum* and *working* sets.

> There is an important principle implemented by this feature: A process gains only those privileges that were in either its fixed or inheritable privileges. A starting process cannot **force** a privilege onto or **through** a new process. If a process mistakenly executes the wrong new process (i.e., a "Trojan Horse"), it cannot pass any privileges that the new process was not designed to enforce.

Figure 1 illustrates the file-based propagation model.

ID-Based Propagation Model

The following model is in effect for the ID-based privilege module:

> A privilege can be acquired only if the privilege exists in the *maximum set* of the *calling* process or is in the *fixed set* of a file. The computation of the *maximum* and *working* sets for the resulting process is done in the *exec*() kernel code:

[8]The check for privileges in device drivers had already been addressed in SVR4 by providing the DDI/DKI interface routine, `drv_priv` [4].

[9]U. S. Patent # 4,135,240.

First, the maximum privilege set of the calling process is stored in a temporary privilege set when the *propagation* routine is entered. Then, the following conditions are evaluated to determine the maximum and working privilege sets for the resulting process:

Step [A] Does the executable file being **exec**'ed have the *setuid–on–exec* bit asserted? If yes, go to Step [B]. If no, set the temporary privilege set to 0 and go to Step [D].

Step [B] Is the owner of the executable file being **exec**'ed "root"? If yes, turn on all privileges in the temporary privilege set and go to Step [D]. Otherwise, go to Step [C].

Step [C] Is the effective user ID of the calling process "root"? If no, turn off all privileges in the temporary privilege set and go to Step [D]. Otherwise, set an indicator for use later and go to Step [D].

Step [D] Does the executable file being **exec**'ed have any *fixed* privileges? If so, add these privileges to the temporary privilege set.

If the privilege sets for the calling process differ from the temporary privilege set generated above, do the following:

a. set the maximum privilege set for the resulting process to the temporary privilege set.
b. If the indicator was set in Step [C], set the working privilege set for the resulting process to the fixed set of privileges found on the executable file.[10] Otherwise,

[10] This was done to maintain compatibility with older versions of the UNIX operating system.

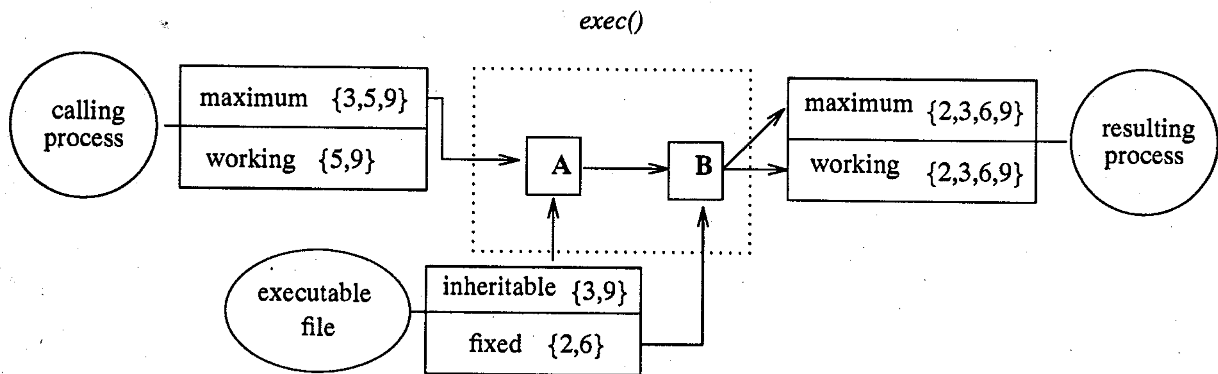

Figure 1: The File–Based Propagation Model

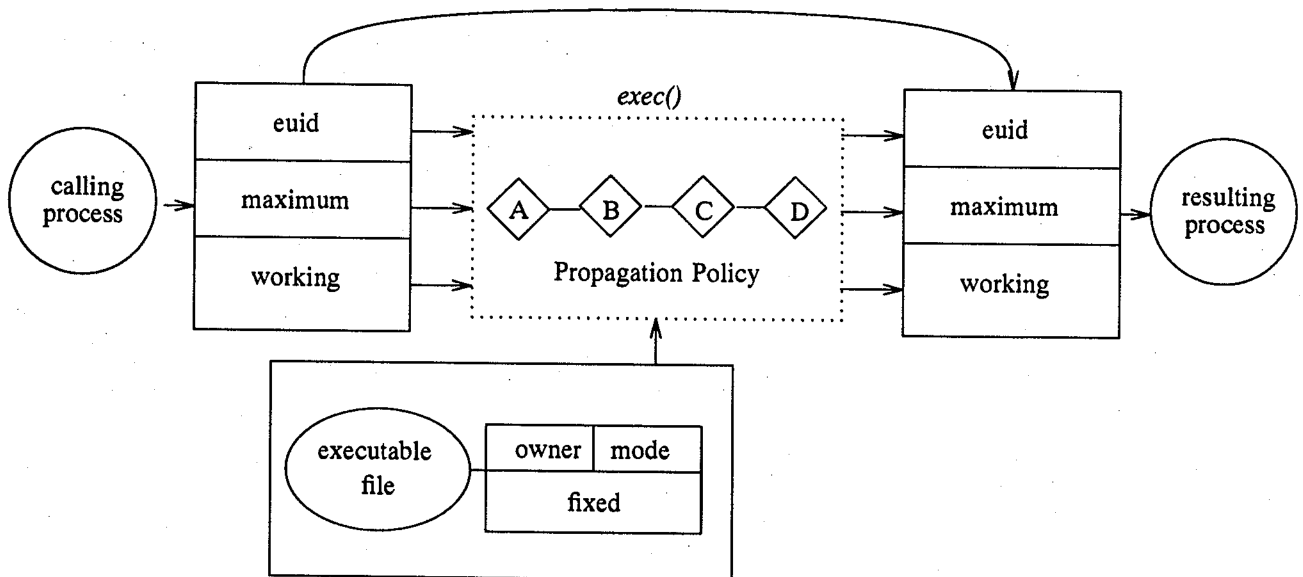

Figure 2: The ID–Based Propagation Model

set the working privilege set for the resulting process to 0.

Figure 2 illustrates the ID–based propagation model.

Privilege Recalculation

The *privilege recalculation* routine adjusts the maximum and working privilege sets of a process according to the privilege module in use.

File–Based Recalculation Model

This routine has a null effect in the file–based privilege module. This is because we intentionally separated the access mechanism from the privilege mechanism (see Section 3.2).

ID–Based Recalculation Model

This routine has extreme significance in the ID–based privilege module because of the tight-coupling between the access and privilege mechanism. This routine is called by the `access()`, `setuid()`, and `seteuid()` system calls.

The following model is in effect for the ID–based privilege mechanism:

The maximum and working privilege sets for the current process are adjusted whenever the effective user ID is modified based on the following conditions:

Condition [A] Clear the maximum and working privilege sets for the current process if none of the process UIDs (effective UID, saved UID, or real UID) equal the *privileged*[11] ID.

Condition [B] Otherwise, set the working privilege set to the maximum privilege set if the effective UID is equal to the *privileged* ID.

Otherwise, only clear the working privilege set if neither Condition [A] nor Condition [B] are true.

Figure i illustrates the ID–based recalculation model.

[11] The *privileged* ID has traditionally been user ID 0. However, this is now a tunable variable allowing for any user ID to be considered ''all–powerful'' in an ID–based privilege mechanism. User ID 0 is the default value for this variable.

Conclusion

Moving the privilege mechanism from the kernel proper to a separate, loadable module was extremely simple. We were fortunate that the checks for privilege in the kernel were somewhat well–defined. We were also fortunate that our system was based on UNIX System V Release 4 since a lot of the ground work to make the separation easier was done in that release.

In addition, we maintained compatibility with SVR4 despite extensive modifications required in the kernel. This means that any operating system configured with the SUM module behaves in the exact same manner as an SVR4 Operating System with hard–coded privilege checks for effective UID 0.

References

[1] Knowles, F. and Bunch, S., *A Least Privilege Mechanism for UNIX*. Proceedings of the 10th National Computer Security Conference. (Sep 1987) Baltimore, MD.

[2] Hecht, M. S., et al. *UNIX Without the Superuser*. Summer USENIX Technical Conferences and Exhibition. (Jun 1987) Phoenix, AZ.

[3] Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice-Hall, Inc., 1986

[4] UNIX Press Title, *UNIX System V Release 4 Device Driver Interface/Driver–Kernel Interface Reference Manual,*

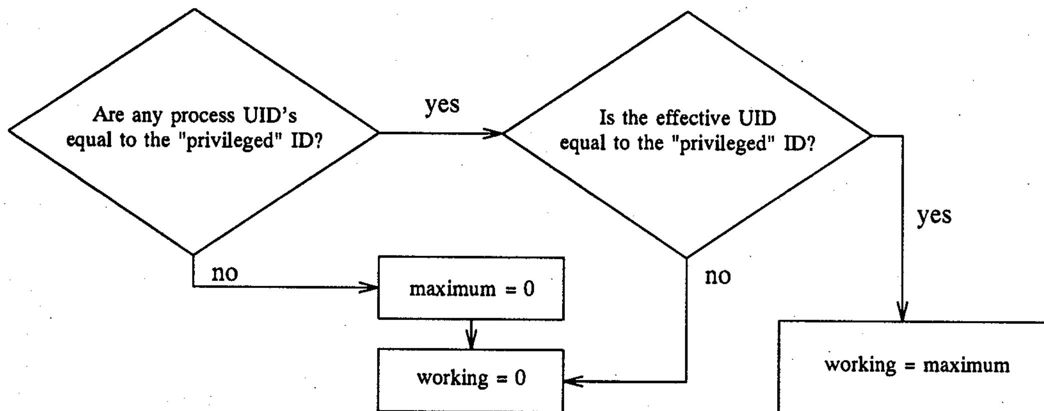

Figure 3: The ID–Based Recalculation Model

Author Information

Charles Salemi is currently a Member of Staff at UNIX System Laboratories, Inc. He has been involved with the UNIX Operating System since he received his degree in Computer Science from the City University of New York in 1977. He has been a member of the Security Development Team since 1987. Within this project his principal interests have been in the area of the privilege mechanism and the Identification and Authentication facility. Along with his other accomplishments, he was one of several people responsible for the development of the Source Code Control System (SCCS).

Suryakanta Shah is a Senior Member of Staff at UNIX System Laboratories. She received a MS in Computer Science from Queens College in 1983. During the past 10 years at UNIX System Laboratories, she has worked on numerous features of System V including B2 security, process management, virtual memory, and the RFS distributed file system. Currently, Kanta is adding multiprocessing capabilities to the B2 security features.

Eric Lund received a BA in English from Tufts University in 1984. Since Spring of 1991 he has been a Senior Programmer / Analyst developing on Multi-Level Security features for Cray Research Inc. in Eagan, Minnesota. Prior to his work at Cray, Eric worked at USL where he helped develop the Privilege, Trusted Path, and Audit mechanisms, developed the Trusted Facility Management mechanism, and performed Covert Channel Analysis on System V Release 4.1ES. Eric was also one of the principal developers of the System V Verification Suite. In his spare time, Eric enjoys camping, rock climbing, juggling, woodworking, and beer brewing/tasting.