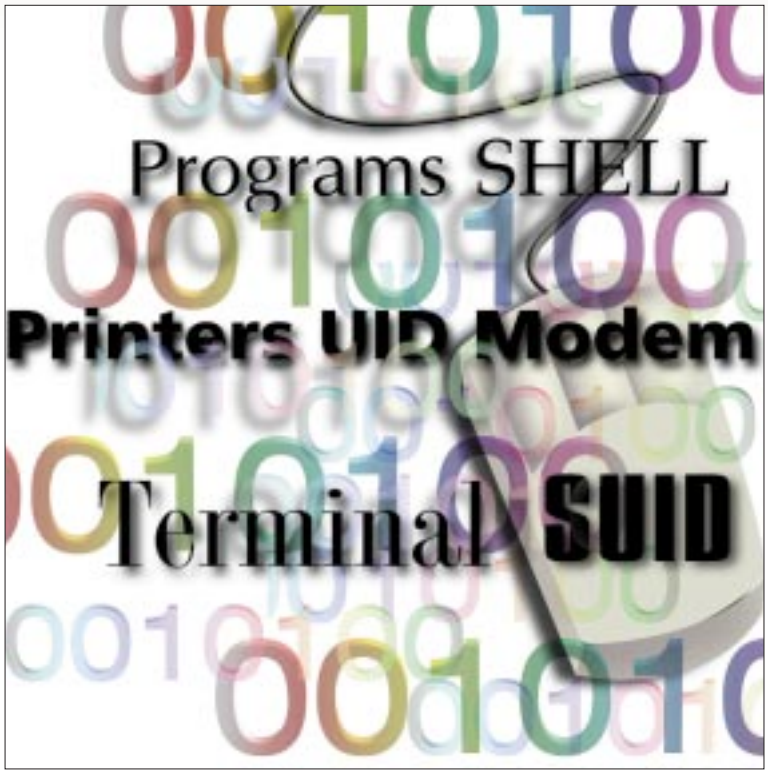
by Peter Collinson, **Hillside Systems**



WENDY GROSSMAN

Resource Access

Every operating system is designed to provide protection for some of its hardware or software. The protection can be slight. MS-DOS, for example, doesn't stop a program from writing to the part of the memory that's occupied by the operating system nor does it prevent a program from writing random data to sensitive parts of its disks. This lack of protection allows people to subvert the operating system, creating the phenomenon of computer viruses.

We want to prevent people from doing things on the system that will cause it to stop functioning correctly. We also want to prevent programming errors caused by human mistakes. On MS-DOS, it's possible for a program to unintentionally write data to some memory that is occupied by code owned by another application. The randomly written data will probably cause problems later.

Incidentally, don't think that MS-DOS provides no protection at all. It does have file permission bits and will not allow the user to delete or change a file that is marked "read-only." Having read-only files can help prevent users from accidentally deleting important files.

Of course, there are occasions where users need to delete those files. If you've installed an application, then it's reasonable to set its files to be read-only and doing so will provide protection against accidental deletion or overwriting. All is well until you want to remove the application from the system or install a new version that will replace some of the files. Then you need to provide an installation program that reverts each of the files to a read/write state, allowing their replacement or removal.

This example of read-only files demonstrates the fundamental problem with protection mechanisms–we need to be able to work around them in some circumstances. A software package has the requirement of being "installed," "reinstalled," "updated" and "deleted." These actions will happen rarely and, most of the time, we just "use" the package. For any specific object on the computer system, there is a set of actions that can be applied. In general, we want to be able to control when certain actions are performed; the degree of control depends on the circumstances.

Another Example

Let's consider another example: a line printer spooling system on a multiuser system. First, we want to ensure that normal users cannot send data directly to the printer device, bypassing the spooling system. We humans are taught to wait in line and take our turn. We also know that given half a chance someone will always leap to the front of the queue, and so we want to establish a system that will prevent this.

A good spooling system allows users to queue files and then processes the files according to a predefined policy. The policy could be first-come, first-served, but doesn't have to be. For example, I worked at a university in the late '70s and we used spoolers that were implemented to discriminate against large files, causing

small files to be printed first. Small files were generally student output, and we wanted to give them priority. Other policies or combinations of policies are possible.

We want users to be able to inspect the queue so they can find out if their print job is done, discover where they are in the print queue or estimate how much longer they will need to wait if other work is in the queue ahead of them. Users will also expect to be able to remove items from the queue and we want to ensure that they can only delete their own print jobs.

So, the set of normal user actions for a spooler is: queuing a file for printing; inspecting the queue; and deleting your own job from the queue. The queue will be a set of files somewhere on the system and we need to protect these files against the normal commands that users use to do their work. We cannot have people copying data willy-nilly into the spooling area or deleting someone else's file. Our spooler control programs require special privilege to gain access to the queue. We want to give users that privilege for the time it takes them to operate the spooler: to queue the file or delete it.

Back when computers were ensconced in machine rooms, there was another set of actions that we might wish to consider for our spooling system. The folks who managed the printers were often given tools to control the queues, perhaps diverting a queued file from one printer to another, advancing the position of some work in a queue or deleting a job that was unnecessary or bad in some way.

A set of spooler control commands for computer operators produces some more requirements that need privilege, but notice that the operators require a different level of privilege from the normal user commands. We don't want all users to be able to perform these control operations because that would give them the ability to circumvent the spooler policies. Users must fit in with the extant policies that are used to run the system.

The need to supply different levels of privilege has been well understood for around 30 years. Early computers were expensive and there were never enough resources to go around, whether those resources were CPU cycles or space on the printer queue. Much work was put into sharing out the resources fairly, or at least allowing them to be shared according to some policy. Sharing resources means protecting them from unauthorized use.

When I was studying Computer Science in the late '60s, the key idea used in operating systems was that of "rings" of protection, which you can visualize by thinking of it in terms of an onion. The kernel of the operating system is the center of the onion; it has the most privilege. User programs run in the outside layer (the skin) and have the least privilege. Moving from the skin of the onion to the center takes you past several layers, each of which has increasing privilege.

When the user program running with least privilege wishes to take some action, it makes a system call that runs code in the next inner ring. Making a system call means the program is now executing some code that's running in a higher priviledged environment. Using a system call allows the operating system to intervene and apply a protection policy.

So, our user-level spooler queuing programs would run on the outermost ring, passing data and information requests through to system code running at a slightly higher privilege. The operator commands would be provided in one of the inner rings. Both the user commands and the operator commands interact with the spooler that is running on a still higher privileged inner ring. The spooler needs privilege to access the user's files and talk to the code sending information to the printer that is supported by the kernel running at the center.

This type of ring protection system was employed on MULTICS, which was the system on which Ken Thompson and Dennis Richie had been working prior to the design of UNIX. The name UNIX is itself derived from MULTICS–it was supposed to be a single-user MULTICS, a UNI-ICS or UNICS, hence UNIX.

UNIX retains some of the ideas of rings of protection but is much simplified, having only two layers. The outer layer runs programs as processes, but when the processes wish to take some action, they issue a system call and switch into the kernel. The process runs code in the kernel and that code will check on the privilege of the process before taking any action. Privilege on UNIX depends on ownership and the particular action a process wishes to take.

When a system call is made, the process is suddenly running in the kernel in a completely different environment from that of the user processes. For example, the kernel can access physical devices or the full address space of the machine.

The kernel uses the memory management hardware on the machine to control access to its own memory space and the memory spaces of all the processes it is controlling. When a user process is running, the memory management system is set up so that the process can only access its own address space. Therefore, a UNIX process cannot trample randomly on the memory space of other processes or the kernel. When the process switches into the kernel, the memory management setup is altered so that the kernel code can access its own memory. If we think about a process running along as a linear path through some code, parts of the process will be running in user space able to access the address space of the process, while other parts will be running in kernel space, where a set of tried-and-true primitive operations permit the copying of data to and from the user process.

Ownership

On a multiuser system, we are concerned with ownership. I wish to prevent users from reading my email, but I am happy for them to look at some of my data files. However, I am uneasy about allowing others to change any files. I'd like users to be able to give me files from time to time. For my files, I want to adopt a policy depending on the nature of the file.

To permit the concept of ownership, we need some way of identifying users and the files they own. The ability of a UNIX process to access a certain file is determined by a pair of numbers that are set up when the user logs in. The first of these, the user ID, or UID, is unique to that system. The second, the group ID, or GID, identifies a collection of users to which the user belongs. Actually, these days, most UNIX systems allow a user to be "in" several groups at the same time. This was origin-

ally implemented to make groups more usable. I am going to ignore this ability for the purposes of this column.

A user will log in with a name and password and the `login` program will translate that information into a (UID, GID) pair that is loaded into the process. Every process the user starts will contain these two numbers and their values are checked against the numbers stored with the files in the file system.

Of course, everything on a UNIX system (or nearly everything) is represented by a file in the file system, so by providing permission checking on files we are also enabling or preventing access to system resources. UNIX extended the need to support file ownership into a mechanism that supplies privilege. There is a set of operations, or system calls, that can be used with files. In general, to gain access to a file, a process must use the `open` system call. When the kernel code that implements the `open` call is executed for a file, it checks the (UID, GID) pair owned by the process against the pair stored with the file.

Each file has three attributes: read, write and execute. Three sets of these attributes are stored with the file and a specific set is checked, depending on the match that can be made between the (UID, GID) pair of the process and the duple stored with the file. If the UIDs match, the leftmost three bits are checked, if the GIDs match the middle three bits are checked, if neither ID values match, then the last three bits are checked. Note that if the user is in the same group as the file, and group permissions denies them access, then the "other" permissions are not checked.

Once the set of three bits has been selected, a test is made to see if the action the process wishes to make is supported by the permissions. Actually, the execute attribute is used by the `exec` system call to see if the user is allowed to load data from the file as a new program. It's also used to allow access to directories, because it doesn't make sense to want to execute a directory.

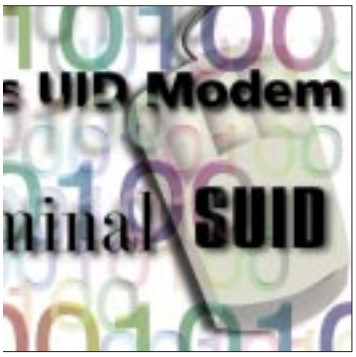
This mechanism is simple, both to understand and to implement. However, it doesn't provide for any special privilege. There are occasions when it's necessary to bypass the normal protection mechanisms to take some special action. Ownership is augmented by the notion of the superuser, whose UID is zero. If the code doing the testing discovers that the UID of the process is zero, then access is (nearly) always granted. Therefore, UNIX has two classes of user: the "superuser" and the "rest." In *Animal Farm* terms: one user is more equal than the others.

Incidentally, the superuser is also vested with the power to execute some system calls that mere mortals cannot. For example, we don't generally want users to have the ability to become other users by setting the (UID, GID) pair in their shell, so only the superuser can use the `setuid` and `setgid` system calls to establish the initial identity of a user when a user logs in.

setuid on Files

Given that many UNIX system management and interrogation programs are actually user processes, and these programs often need to have superuser rights to access certain key files, how do we provide these programs with the necessary superuser privilege? Well, the answer is the notion of the `setuid` bit in the file permissions, and this was worth a patent that was granted to Dennis Ritchie.

To start a new command, a process will first *fork*, creating a clone of itself. The new process (the child) will then generally use the `exec` system call to load a new process and, by doing so, will completely replace its memory and data with information from the file. Part of each process is a collection of *process state* that survives the `exec` system call. For example, any files that are open in the old process will remain open after the `exec` system call. The (UID, GID) pair also survives the `exec` system call and this is the way ownership is normally passed from process to process.



However, the (UID, GID) pair is not preserved if the file permission contains an enabled `setuid` or `setgid` bit. In this case, when the new program is `exec`'d, its ownership values are set from the values stored with the file being loaded. To confer special privilege on a command, I create a file whose owner is `root` (a UID of zero) and whose group is `bin` (a GID of two on my machine). Now, I turn the `setuid` bit on and the program will run with a UID of zero, owned by `root`. If I turn on `setgid`, the program will be placed in the group with a GID of five.

We now have a mechanism where we can set the (UID, GID) pair of a file containing the command and ensure that when the process runs it will have file access that matches the owner of the file. If you look around your system, you'll find several processes that take advantage of the ability to temporarily switch to another user. Many of these programs require special access to devices. For example, `df` and `ps` both require special access to physical devices to do their job. For normal use, these devices are protected against access by being owned by `root` and bearing appropriate file permissions. When the `setuid` programs are run, they are run *as* `root`, conferring the right of access to the devices.

The mechanism is slightly more complicated than I have described above, the (UID, GID) pair is not actually replaced because we'd like to provide the process with the ability to revert to the original (UID, GID) pair. The process state actually contains an effective (UID, GID) pair that is used to check file permissions, and it is this effective pair that is set by the `exec` system call when appropriate `setuid` or `setgid` bits are turned on.

Security

The notion of having a superuser and allowing users to become "temporary" superusers creates a single target for hackers who want to subvert the system. If a hacker can become superuser, then they have the keys to the kingdom. The problem is that the integrity of the system relies on the skill of all the people who have written `setuid` programs.

For example, I want to provide a mechanism whereby a user can edit a protected file owned by superuser. So, I create a program that is `setuid` to `root`. As `root`, the program opens the file and places the user into an editor to change it. Sound OK? Well no. What I've forgotten is that most editors contain shell escapes, allowing the user to start an interactive shell. This subshell will inherit the rights of its parent and will be owned by superuser. Thus, I've managed to give all users a way to become superuser.

I need to code my program so that it creates a file that is owned by the actual user, and reverts to the (UID, GID) pair of the user before starting the editor. I load the original ownership value in the child before I use `exec` to run the editor, while preserving the superuser capabilities in the parent. When the editor exits, my privileged program can inspect the changes the user has made and apply them appropriately.

The notion of superuser and the idea of `setuid` programs are used to provide extra privilege to specific programs, enabling them to have controlled access to resources that would otherwise be unavailable. The ideas have also widened the ability of the hacker to break into the system and, in recent years, many UNIX software suppliers have gone from one panic to the next, plugging holes in various `setuid` programs as new subversion methods came to light.

One thing to note is that it is easy to become `root` on your system if you provide users with a shell script that is `setuid` to `root`. Many people do this to "make it easy on their users." The problem is that it's too easy to subvert shell scripts without changing them. For example, if you create a script that contains a command `fred`, I can create a command called `fred` and make your shell script run my command by setting the `PATH`. If I can get your shell script to run my command when your shell script is running as `root`, then I too can become `root`. There were several other holes that now appear to have been plugged on my Solaris 2.6 system. However, don't do it. Don't create `setuid` scripts that change to be the superuser.

Further Reading

One of the sources for this article is the excellent *Advanced Programming in the UNIX Environment* by W. Richard Stevens (part of the Addison-Wesley Publishing Co. Professional Computing Series, 1992, ISBN 0-201-56317-7). This book is a classic. The other is *Operating Systems, Design and Implementation* by Andrew S. Tanenbaum, Prentice-Hall Inc., 1986, ISBN 0-13-637406-9. Well, 1986 doesn't seem all that new, but this book defined the MINIX operating system, which (I understand) is deemed to be the parent of Linux.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.