

Identification, Permissions, and Security

Processes are the active elements in a computer system: they are the agents that perform various operations on system elements. For example, a process can create or delete a file, map a region of memory, and signal another process.

In most cases, a process performs its work on behalf of a human user. In effect, the process represents the user in the system. And in many cases, it is desirable to control what different users are allowed to do to certain system objects. This chapter deals with the issues of identifying a user with a process, and of granting or denying the rights to manipulate objects. In particular, it also covers security — denying certain rights despite the users best efforts to obtain them.

7.1 System Security

The operating system is omnipotent

The operating system can do anything it desires. This is due in part to the fact that the operating system runs in kernel mode, so all the computer's instructions are available. For example, it can access all the physical memory, bypassing the address mapping that prevents user processes from seeing the memory of other processes. Likewise, the operating system can instruct a disk controller to read any data block off the disk, regardless of who the data belongs to.

The problem is therefore to prevent the system from performing such actions on behalf of the wrong user. Each user, represented by his processes, should be able to access only his private data (or data that is flagged as publicly available). The operating system, when acting on behalf of this user (e.g. during the execution of a system call), must restrain itself from using its power to access the data of others.

In particular, restricting access to file data is a service that the operating system

provides, as part of the file abstraction. The system keeps track of who may access a file and who may not, and only performs an operation if it is permitted. If a process requests an action that is not permitted by the restrictions imposed by the system, the system will fail the request rather than perform the action — even though it is technically capable of performing this action.

7.1.1 Levels of Security

Security may be all or nothing

It should be noted that when the operating system's security is breached, it typically means that *all* its objects are vulnerable. This is the holy grail of system hacking — once a hacker “breaks into” a system, he gets full control. For example, the hacker can impersonate another user by setting the user ID of his process to be that of the other user. Messages sent by that process will then appear to all as if they were sent by the other user, possibly causing that user some embarrassment.

Example: the Unix superuser

In unix, security can be breached by managing to impersonate a special user known as the superuser, or “root”. This is just a user account that is meant to be used by the system administrators. By logging in as root, system administrators can manipulate system files (e.g. to set up other user accounts), change system settings (e.g. set the clock) or clean up after users who had experienced various problems (e.g. when they create files and specify permissions that prevent them from later deleting them). The implementation is very simple: all security checks are skipped if the requesting process is being run by root.

Windows was originally designed as a single-user system, so the user had full privileges. The distinction between system administrators and other users was only introduced recently.

Alternatively, rings of security can be defined

An alternative is to design the system so that it has several security levels. Such a design was employed in the Multics system, where it was envisioned as a set of concentric rings. The innermost ring represents the highest security level, and contains the core structures of the system. Successive rings are less secure, and deal with objects that are not as important to the system's integrity. A process that has access to a certain ring can also access objects belonging to larger rings, but needs to pass an additional security check before it can access the objects of smaller rings.

7.1.2 Mechanisms for Restricting Access

Access can be denied by hiding things

A basic tool used for access control is hiding. Whatever a process (or user) can't see, it can't access. For example, kernel data structures are not visible in the address space of any user process. In fact, neither is the data of other processes. This is achieved using the hardware support for address mapping, and only mapping the desired pages into the address space of the process.

A slightly less obvious example is that files can be hidden by not being listed in any directory. For example, this may be useful for the file used to store users' passwords. It need not be listed in any directory, because it is not desirable that processes will access it directly by name. All that is required is that the system know where to find it. As the system created this file in the first place, it can most probably find it again. In fact, its location can even be hardcoded into the system.

Exercise 145 One problem with the original versions of Unix was that passwords were stored together with other data about users in a file that was readable to all (/etc/passwd). The password data was, however, encrypted. Why was this still a security breach?

Conversely, permissions can be granted using opaque handles

Given that system security is an all or nothing issue, how can limited access be granted? In other words, how can the system create a situation in which a process gains access to a certain object, but cannot use this to get at additional objects? One solution is to represent objects using opaque handles. This means that processes can store them and return them to the system, where they make sense in a certain context. However, processes don't understand them, and therefore can't manipulate or forge them.

A simple example is the file descriptor used to access open files. The operating system, as part of the `open` system call, creates a file descriptor and returns it to the process. The process stores this descriptor, and uses it in subsequent `read` and `write` calls to identify the file in question. The file descriptor's value makes sense to the system in the context of identifying open files; in fact, it is an index into the process's file descriptors table. It is meaningless for the process itself. By knowing how the system uses it, a process can actually forge a file descriptor (the indexes are small integers). However, this is useless: the context in which file descriptors are used causes such forged file descriptors to point to other files that the process has opened, or to be illegal (if they point to an unused entry in the table).

To create un-forgeable handles, it is possible to embed a random bit pattern in the handle. This random pattern is also stored by the system. Whenever the handle is presented by a process claiming access rights, the pattern in the handle is compared

against the pattern stored by the system. Thus forging a handle requires the process to guess the random patterns that is stored in some other legitimate handle.

Exercise 146 *Does using the system's default random number stream for this purpose compromise security?*

7.2 User Identification

As noted above, users are represented in the system by processes. But how does the system know which user is running a particular process?

Users are identified based on a login sequence

Humans recognize each other by their features: their face, hair style, voice, etc. While user recognition based on the measurement of such features is now beginning to be used by computer systems, it is still not common.

The alternative, which is simpler for computers to handle, is the use of passwords. A password is simply a secret shared by the user and the system. Any user that correctly enters the secret password is assumed by the system to be the owner of this password. In order to work, two implicit assumptions must be acknowledged:

1. The password should be physically secured by the user. If the user writes it on his terminal, anyone with access to the terminal can fool the system.
2. The password should be hard to guess. Most security compromises due to attacks on passwords are based on the fact that many users choose passwords that are far from being random strings.

To read more: Stallings [1, sect. 15.3] includes a nice description of attacks against user passwords.

Processes may also transfer their identity

It is sometimes beneficial to allow other users to assume your identity. For example, the teacher of a course may maintain a file in which the grades of the students are listed. Naturally, the file should be accessible only by the teacher, so that students will not be able to modify their grades or observe the grades of their colleagues. But if users could run a program that just reads their grades, all would be well.

Unix has a mechanism for just such situations, called "set user ID on execution". This allows the teacher to write a program that reads the file and extracts the desired data. The program belongs to the teacher, but is executed by the students. Due to the set user ID feature, the process running the program assumes the teacher's user ID when it is executed, rather than running under the student's user ID. Thus it has access to the grades file. However, this does not give the student unrestricted access

to the file (including the option of improving his grades), because the process is still running the teacher's program, not a program written by the student.

Exercise 147 *So is this feature really safe?*

A more general mechanism for impersonation is obtained by passing opaque handles from one process to another. A process with the appropriate permissions can obtain the handle from the system, and send it to another process. The receiving process can use the handle to perform various operations on the object to which the handle pertains. The system assumes it has the permission to do so because it has the handle. The fact that it obtained the handle indirectly is immaterial.

Exercise 148 *Does sending a Unix file descriptor from one process to another provide the receiving process with access to the file?*

7.3 Controlling Access to System Objects

System objects include practically all the things you can think of, e.g. files, memory, and processes. Each has various operations that can be performed on it:

<i>Object</i>	<i>Operations</i>
File	read, write, rename, delete
Memory	read, write, map, unmap
Process	kill, suspend, resume

In the following we use files for concreteness, but the same ideas can be applied to all other objects as well.

Access to files is restricted in order to protect data

Files may contain sensitive information, such as your old love letters or new patent ideas. Unless restricted, this information can be accessed by whoever knows the name of the file; the name in turn can be found by reading the directory. The operating system must therefore provide some form of protection, where users control access to their data.

There are very many possible access patterns

It is convenient to portray the desired restrictions by means of an *access matrix*, where the rows are users (or more generally, domains) and the columns are objects (e.g. files or processes). The i, j entry denotes what user i is allowed to do to object j . For example, in the following matrix file3 can be read by everyone, and moti has read/write permission on dir1, dir2, file2, and the floppy disk.

users	objects							
	dir1	dir2	file1	file2	file3	file4	floppy	tape
yossi	r		r		r			
moti	rw	rw		rw	r		rw	
yael		r			r			r
rafi					r			rw
tal	r		rw		r	rw	rw	
...								

Most systems, however, do not store the access matrix in this form. Instead they either use its rows or its columns.

You can focus on access rights to objects...

Using columns means that each object is tagged with an *access control list* (ACL), which lists users and what they are allowed to do to this object. Anything that is not specified as allowed is not allowed. Alternatively, it is possible to also have a list cataloging users and specific operations that are to be prevented.

Example: ACLs in Windows NT

Windows NT uses ACLs to control access to all system objects, including files and processes. An ACL contains multiple *access control entries* (ACEs). Each ACE either allows or denies a certain type of access to a user or group of users.

Given a specific user that wants to perform a certain operation on an object, the rules for figuring out whether it is allowed are as follows.

- If the object does not have an ACL at all, it means that no access control is desired, and everything is allowed.
- If the object has a null ACL (that is, the ACL is empty and does not contain any ACEs), it means that control is desired and nothing is allowed.
- If the object has an ACL with ACEs, all those that pertain to this user are inspected. The operation is then allowed if there is a specific ACE which allows it, provided there is no ACE which forbids it. In other words, ACEs that deny access take precedence over those which allow access.

Exercise 149 *Create an ACL that allows read access to user yosi, while denying write access to a group of which yosi is a member. And how about an ACL that denies read access to all the group's members except yosi?*

... or on the capabilities of users

Using rows means that each user (or rather, each process running on behalf of a user) has a list of *capabilities*, which lists what he can do to different objects. Operations that are not specifically listed are not allowed.

A nice option is sending capabilities to another process, as is possible in the Mach system. This is done by representing the capabilities with opaque handles. For example, a process that gains access to a file gets an unforgeable handle, which the system will identify as granting access to this file. The process can then send this handle to another process, both of them treating it as an un-interpreted bit pattern. The receiving process can then present the handle to the system and gain access to the file, without having opened it by itself, and in fact, without even knowing which file it is!

Grouping can reduce complexity

The problem with both ACLs and capabilities is that they may grow to be very large, if there are many users and objects in the system. The solution is to group users or objects together, and specify the permissions for all group members at once. For example, in Unix each file is protected by a simplified ACL, which considers the world as divided into three: the file's owner, his group, and all the rest. Moreover, only 3 operations are supported: read, write, and execute. Thus only 9 bits are needed to specify the file access permissions. The same permission bits are also used for directories, with some creative interpretation: read means listing the directory contents, write means adding or deleting files, and execute means being able to access files and subdirectories.

Exercise 150 *what can you do with a directory that allows you read permission but no execute permission? what about the other way round? Is this useful? Hint: can you use this to set up a directory you share with your partner, but is effectively inaccessible to others?*

7.4 Summary

Abstractions

Security involves two main abstractions. One is that of a user, identified by a user ID, and represented in the system by the processes he runs. The other is that of an object, and the understanding that practically all entities in the system are such objects: table entries, memory pages, files, and even processes. This leads to the creation of general and uniform mechanisms that control all types of access to (and operations upon) system objects.

Resource management

Gaining access to an object is a prerequisite to doing anything, but does not involve resource management in itself. However, handles used to access objects can in some cases be resources that need to be managed by the system.

Workload issues

Because security is not involved with resource management, it is also not much affected by workload issues. But one can take a broader view of “workload”, and consider not only the statistics of what operations are performed, but also what patterns are desired. This leads to the question of how rich the interface presented by the system should be. For example, the access permissions specification capabilities of Unix are rather poor, but in many cases they suffice. Windows NT is very rich, which is good because it is very expressive, but also bad because it can lead to overhead for management and checking and maybe also to conflicts and errors.

Hardware support

Security is largely performed at the operating system level, and not in hardware. However, it does sometimes use hardware support, as in address mapping that hides parts of the memory that should not be accessible.

Bibliography

- [1] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd ed., 1998.