Operating Systems

Protection & Security

By Paul Krzyzanowski
December 1, 2010 [updated April 20, 2011]

> *He that has eyes to see and ears to hear may convince himself that no mortal can keep a secret. If his lips are silent, he chatters with his fingertips; betrayal oozes out of him at every pore. — Sigmund Freud*

Introduction

The earliest computer systems (and most personal computers, until relatively recently) were rather simple compared with systems of today. With only one process in the system's memory at any time and a very limited number of I/O devices, a process could do no harm onto others. Security was still an issue, however. Some of the earliest machines (ENIAC, EDVAC) were given the task of computing ballistic firing tables. The data was sensitive and access to computing facilities was controlled by walls, locks, and guns and only trusted people were allowed near the computer (or to write programs for it). Even today, this is the most effective method of ensuring physical and information security. For instance, to get the Department of Defense's highest security level, A1, uncleared users cannot have access to the system.

As time-sharing systems emerged, system security took on a new meaning. Even in a trusted community, one has to consider cases where individual programs or commands may go awry (and scribble in memory regions, devices, and/or files that are used by others). Also, in a community in which users may not always be trusted, one now has to consider that users may wish to keep certain data (and possibly devices) restricted from others.

The next big change big era in computer security was the emergence of computer networking. With packet switched networks, one machine on a network may potentially snoop on other packets that are flowing in the network. Machines legitimately receiving packets now may wonder whether the packets are genuine or forgeries. In addition to that, authenticating (identifying) a user needs greater security, since it is no longer the case that people in physical proximity or directly connected to the computer will be the only ones trying to access it.

Finally, we find ourselves at a stage where not only do we want our machines connected to a wide area network, but we welcome the teeming multitudes and grant them restricted access to our machines and resources (e.g. the World-Wide Web, gopher, anonymous ftp). If that isn't enough, we also connect to random machines in the world while surfing the web and allow them to run code on our machines.

Protection versus security

We often throw the terms *security* and *protection* around interchangeably. Let's make a distinction (perhaps not a clear-cut one). By **security**, we refer to a set of policies that prevent unauthorized access to a system. This encompasses malicious as well as accidental access. Overwriting the return value on a stack or reading a file that is not yours may be innocent mistakes or malevolent transgressions. The *access* operations we talk about include any scope of access, such as a user login, physical access to the machine, or a process trying to access devices or data that it should not be allowed to. Such accesses may include reading, alteration, creation, or destruction of data.

By **protection**, we refer to the *mechanism* that provides and enforces controlled access of resources to to users and processes. A protection mechanism *enforces* security policies.

Principle of least privilege

A key approach to security policies is the **principle of least privilege**. This states that *at each abstraction layer, every element should be able to access only the resources necessary to perform its task*. By *element*, we generally refer to a process but it can also encompass everything that a user is allowed to do or perhaps a specific actions allowable to a function.

The rationale for this principle is that, even if an element is compromised, the scope of damage can be limited. Let's consider a few examples of how this principle can be applied or violated:

- *A compromised print spooler allows the attacker to add new user accounts.*
  An attacker exploited a bug in a print spool daemon and is able to use it to execute arbitrary commands. Since the process is running with administrative privileges, the attacker has it execute a command to create a new user account. This is a violation of the principle since a print spooler has no legitimate reason to ever create user accounts. It should only be able to deal with the files in the print queue and interface with the printer.
- *A game reads files in a user's home directory.*
  A game may need to read a file containing saved state or high scores but accessing any other file violates the principle of least privilege. By granting this access, we make it possible for the game to steal files, remove or destroy files, or even accidentally create files in places it shouldn't.
- *Java programs don't allow a function in one object to access private variables of another object.*
  This is an example of the principle of least privilege in operation within a programming language. The programmer made policy decisions on which variables are will be private and which will not be. There are severe limitations imposed by the language in structuring the policy. We cannot, for instance, identify which classes have *write* access, which have *read* access, and which will be allowed no access.
- *New user accounts get administrative privileges.*
  This is (was) a common policy for personal computers and is a violation of the principle. Users of personal machines tend to be administrators but giving a user administrative access means that any programs they run will have administrative access as well. For example, many viruses and rootkits (we'll discuss these later) are installed onto system files or system directories — places that should be off-limits to user processes.

This principle of least privilege is often difficult to define, implement, and enforce. More often than not, our operating systems (and programming languages) do not give us enough granularity to fully define these policies. For example, you usually cannot specify that a certain program cannot use a certain set of system calls or can only access a specific set of files in the file system.

Privilege separation

Privilege separation is the process of dividing a program into multiple communicating parts, where each part has only the privileges it needs to operate. In a way, the interaction between a process and the operating system is an example of privilege separation. A user's process is restricted from certain operations (e.g., accessing other user's files, accessing devices directly, changing memory mappings). When it needs to do these actions, it sends a request to the the operating system via a system call. With user programs, privilege separation involves creating multiple processes with different access rights and well-defined communication interfaces between them. As with the principle of least privilege, the hope is to minimize the amount of harm that a process can do.

On POSIX systems, every process has associated with it both a **real** and **effective user ID**. In most cases, they are the same. When a program is run, the process assumes the user ID (*uid*) and effective user ID (*euid*) of the user that ran it. Access privileges are evaluated

A file can be set to run under the ID of a user *bob* regardless of who runs it by changing its owner to bob:

based on the effective user ID (*euid*) of the process. An executable file may have a **setuid bit** set as one of its attributes. If this bit is set then, when the command is run, the effective user ID is set to the file owner's ID, not the user's. The user ID is still set to the user's ID.

```
chown bob filename
```
and then setting the setuid bit and execute bit with:
```
chmomd +sx filename
```

An example of how we may separate a program into two parts with different privileges is to set the owner of the program to a user that has access to whatever restricted resources the program needs and set the permissions of the file to *setuid*. When the program is run, a single process is created. The first thing the program will do is to create an interprocess communication link to itself (via the *pipe (http://www.manpages.info/freebsd/pipe.2.html)* system call, shared memory (http://www.manpages.info/sunos/shmat.2.html), sockets, or any other available mechanism). It then forks into two processes. One process will call a function that will process requests from the other process. It will run under the effective UID of the program's owner. The other process will execute the system calls:

```
seteuid(getuid);
```

This will set the effective user ID to the real user ID, giving this process only the privileges that the user has.

Goals

The overall goals of security include the following:

- **Authentication**: Ensure that users, machines, programs, and any other system resources are properly identified. It makes no sense to enforce security policies if we don't have confidence in which policy we have to enforce.
- **Confidentiality**: We want to ensure that processes cannot access any unauthorized data.
- **Integrity**: We need to ensure that data has not been compromised and be able to detect if it has.
- **Availability**: The system should be accessible to do what it is supposed to do. If we cannot create new processes cannot or access network services on a machine then we may have a security violation as well. For instance, a denial of service attack attempts to take machines out of service, or at least make them appear unresponsive on the network.

The operating system steps in

The operating system and system hardware provides user programs with access to resources. For example:

process scheduler	Provides controlled access to the CPU.
memory manager	Provides controlled access to system memory and gives the process the illusion that it is the only one accessing it.
device drivers and the buffer cache	Provides controlled access system peripherals.
file system	Provides controlled access to logical regions of persistent data.
sockets	Provides controlled access to communication networks.

All resource access requests go through the operating system. The operating system decides whether access should be granted and what sort of access is permissible. In many cases, all resource access itself goes through the operating system as well (e.g. read/write files). For memory, we've seen how the operating system may set hardware permissions in the memory management unit to enable applications to access memory in a protected manner directly through hardware.

Domains of protection

In formulating a security policy, we first look at operating system protection in the abstract. We can view a computer system as a collection of processes and objects, where the objects are both the system hardware (CPU, memory, devices) and software (files, programs, semaphores, sockets). Each object has some unique name or identifier and can be accessed through well-defined operations. Thus, the objects essentially are abstract data types where the allowable operations depend on the object (a CPU object can only be used for execution, memory segments can be read/written/read-for-execute, files may be opened/closed /created/read/written/...).

A process should be allowed to access only the resources that it has been authorized to access. Moreover, a process should be able to access only the resources it needs to complete the job and no more.

Each process operates within a **protection domain**. This domain specifies resources that processes may access and how they may access them. The domain itself is a set of objects and the types of operations that can be performed upon them. An **access right** is the ability to execute an operation on an object. A domain can then be viewed as a collection of access rights, where each access right can be represented by the ordered pair: *<object-name, right-set>*. For example, if the right *<file F, {read, write}>* exists in domain *D* it means that a process that is executing in domain *D* may perform *read* and *write* operations on file *F*. Domains in general need not be disjoint; they may share access rights. The association between processes and domains may be either static (a process is always operating within one domain) or dynamic (a process may switch between domains).

A domain may be realized at one of several levels. Every user may be a domain. In this case, object access depends on the user ID. Every process may be a domain. In this case, access depends on the process ID. Finally, each procedure may be treated as a domain. In such an environment, the objects correspond to local variables.

Modeling the domain

One generic way of representing domains of protection is with an **access matrix**. This is a two-dimensional structure where the rows correspond to *domains* and columns correspond to *objects*. Each entry contains the set of access rights of a domain on an object. For example, Figure 1 shows a very simple access matrix:

domains of protection	objects		
	F_0	F_1	Printer
D_0	read	read-write	print
D_1	read-write-execute	read	
D_2	read-execute		
D_3		read	print
D_4			print

Figure 1. Access matrix

A process running in domain D_1 will be allowed *read*, *write*, and *execute* access to file F_0. The operating system is

domains of protection	objects							
	F_0	F_1	Printer	D_0	D_1	D_2	D_3	D_4
D_0	read	read-write	print	–	switch	swtich		
D_1	read-write-execute	read			–			
D_2	read-execute				swtich	–		
D_3		read	print					
D_4			print					

Figure 2. Access matrix with domain switching

responsible for ensuring that a process executing in domain D_i can access only objects in row i as allowed by the matrix. An access control list can also model other properties in a domain. For example, **domain switching** can be supported, which is the permission to switch from one domain to another. It may be represented as a *switch* operation upon a domain. We can extend the access matrix to show this by adding domains to the columns (Figure 2).

This allows processes to switch domains only if they have the right to do so. Another bit of functionality that is needed is the ability to change permissions in this matrix. We can define a **copy right** as the permission to copy an access right on an object in one domain onto the same object in another domain. To allow rights to be created and removed, we can define an **owner right**. If a domain is granted an *owner* right for an object, it is allowed to create or remove any rights for that object in other domains (staying in the same column). Finally, to allow access to objects within a domain to be controlled, we can define a **control right**. If an access in the matrix gives a process the *control* right, it can remove any access right from that entire row.

Implementing an access matrix

One way of implementing an access matrix could be for the operating system to maintain the access matrix asa table or as a list of triples containing <*domain*, *object*, *right-set*>. Such a table will generally be very large and not practical to store in memory. Hence, extra I/O will be involved in accessing it. In addition, it becomes difficult to take advantage of special groupings, such as a file that can be read by everyone). Adding a new object (e.g., creating a file) means adding entries for every domain in the system. Adding a new domain (e.g., creating a new user) means iterating through every object in the system and creating a new access right for the domain. Given that objects come and go frequently, this clearly is not practical.

Access control lists

A more directly implementable approach to managing an access matrix is to associate a column of the table with each object. Each column in an access matrix can be stored as an **access control list** (**ACL**) for that one object. To save space, empty entries can be discarded. This list contains a set of ordered pairs, <*domain, right-set*> for that object. The list can also be extended to define a default set of access rights. If the domain is not found within the list, default access permissions for that object will apply.

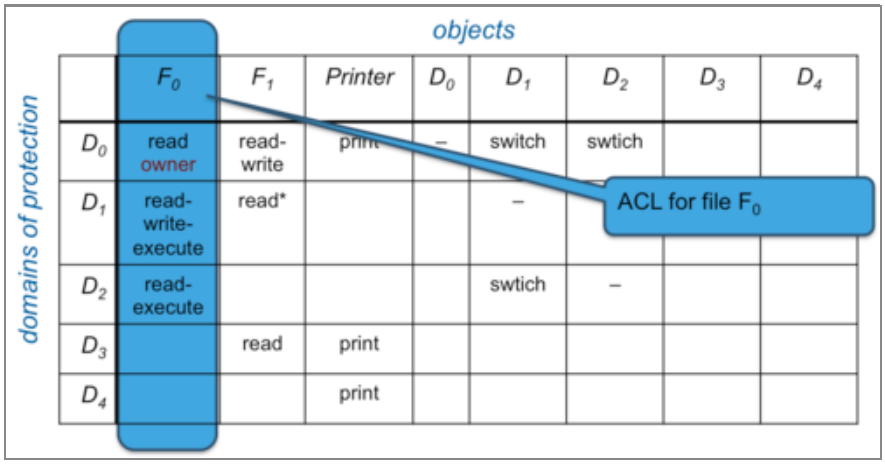

Figure 2. Access control list

Capability lists

Access control lists associate columns in an access matrix. A **capability list** associates each row with its domain. It is a list containing objects along with the operations allowed on them.

The object is often represented by a name or address, called a **capability**. To execute operation *M* on object O_j, a process requests operation *M* specifying the capability (pointer) for object O_j as a parameter. Possession of the capability means that access is allowed. The capability list is associated with a domain but is never directly accessible to a process in that domain. It is a protected object maintained by the operating system. Capability-based protection is based on the fact that capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified).

Unix approach: limited ACLs

The UNIX time-sharing system associates a domain with a user. Cross-domain switching corresponds to switching the user ID temporarily. This occurs by executing a *setuid* program or by having *super user* privileges and calling the *setuid* system call to set a specific user explicitly. Since devices are accessed through the file system, all access permissions for devices as well as files are stored within the file's inode.

A true access control list was deemed inefficient and expensive to implement. It could not be contained within a fixed-sized inode entry. Hence, auxiliary disk I/O would be required to access an access control list. Instead, a simplified version of the concept was implemented. Access permissions are divided into three parts: *user* (the owner of the resource), *group* (a user may belong to a group, which is defined in the `/etc/groups` file), and *other* (everyone else). For each of the groupings, the allowable permissions are *read*, *write*, and *execute*. For directories, execute corresponds to search permission. Two additional permissions are *setuid*, in which the process' user ID will be changed to the owner of a file when it executes that file and *setgid*, in which the process' group ID will be changed to the group owner of the file when that file is executed.

The UNIX administrator, the *super user*, or *root* (user ID 0) has supreme powers and access permissions do not apply for this user.

For certain applications, UNIX's granularity of access protection was deemed inadequate (for example, when trying to obtain B-2 security certification from the Department of Defense). As a result, most versions of UNIX were extended to support full Access Control Lists (ACLs), In which the basic permissions serve as a default and auxiliary data allows an owner to specify a full access list. Hence, the use of ACLs is entirely optional. With the ext3 file system, ACLs are stored as *extended attributes*. Each inode has a field called `i_file_acl` that, if it's not zero, contains the block number of the block that stores a list of extended attributes.

Windows approach (NT platform releases: Windows Servers, Vista, Windows 7, etc.)

All the system resources that need protection processes, threads, events, timers, semaphores, files, ...) are treated as objects and all access to them goes through the *object manager* (names of objects appear as file system names since the object manager is responsible for presenting its namespace. Since the object manager is a common point of access to resources, it is responsible for protecting objects. This ensures that all objects are protected in a uniform manner.

The NT security subsystem is responsible for authenticating a user. Once a user has been identified and authenticated, the security subsystem creates an object called an *access token* which is attached to the user's process. This token serves to identify the user whenever the process requests a system resource. It includes a *security ID* (typically the user's login name, a list of group IDs to which the user belongs, privileges, a default owner, primary group, and a default access control list. The access control list is an initial list of permissions that will apply to objects that the process creates.

When a process requests access to an object (via an *open* operation), the object manager contacts the security reference monitor. This monitor gets the process' token and determines whether it may access the object. If access is permitted, the monitor returns a set of granted access rights for that requesting process. The object manager stores these rights in the object

handle that is created for that object (that is used for future references to the object). This enables the security system to perform rapid checks when the process accesses the object.

Mandatory vs. Discretionary access control models

Most operating systems we use provide a **discretionary access control** model (**DAC**). Ths is one where the subject (the domain) can pass information onto any other subject. In some cases, access rights to the object itself may be transferred onto a new domain. For example, a process that has the right to open a file that might be restricted from others can open it, read it, and create a less-protected copy that everyone can see. Alternatively, a user that owns a file can change its permission mode so that others can read it.

Mandatory access control (**MAC**) refers to a *centrally-controlled* policy in which the operating system restricts actions of domains (subjects) on objects. Users do not have the ability to modify this policy.

Mandatory Access Control is often associated with the **Multi-Level Secure** (**MLS**) access model. The **Bell-LaPadula model** is the most popular example of this model. It identifies the ability of a process (domain) to access and communicate data. All objects are classified into a hierarchy of sensitivity levels (unclassified, confidential, secret, top secret). Users (domains) are assigned a clearance level as well. The overall policy is *no read up; no write down*. This means that a user cannot read objects that belong to a higher clearance level than the user is assigned. The user also cannot write or modify objects that belong to a lower clearance level. The MLS approach fits well with the government's use of classified information but generally does not translate well to civilian life.

References

- Solaris Operating System Data Sheet, Solaris 10 Operating System and Security - Advanced Features Enable Secure Systems (http://www.sun.com/software/solaris/ds/security.jsp)
- Multi-level Security (MLS) (http://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-mls-ov.html), Red Hat Enterprise Linux Deployment Guide
- Computer Security (http://en.wikipedia.org/wiki/Computer_security)
- Separation of protection and security (http://en.wikipedia.org/wiki/Distinction_of_protection_and_security)
- Bell-LaPadula model (http://en.wikipedia.org/wiki/Bell-LaPadula_model)
- Cryptography As An Operating System Service: A Case Study (http://www.openbsd.org/papers/crypt-service.pdf), Angelos D. Keromytis, Jason L. Wright, Theo DeRaadt, Matthew Burnside.
- Crash course on cryptography: Digital Certificates (http://www.iusmentis.com/technology/encryption/crashcourse/certificates/), Ius mentis.
