Operating Systems 2230

Computer Science & Software Engineering

Lecture 11: OS Protection and Security

Computer protection and security mechanisms provided by an operating system must address the following requirements:

Confidentiality: (or *privacy*) the requirement that information maintained by a computer system be accessible only by authorised parties (users and the processes that run as/represent those users).

Interception occurs when an unauthorised party gains access to a resource; examples include illicit file copying and the invocation of programs.

Integrity: the requirement that a computer system's resources can be modified only by authorised parties.

Modification occurs when an unauthorised party not only gains access to but changes a resource such as data or the execution of a running process.

Availability: the requirement that a computer system be accessible at required times by authorised parties.

Interruption occurs when an unauthorised party reduces the availability of or to a resource.

Authenticity: the requirement that a computer system can verify the identity of a user.

Fabrication occurs when an unauthorised party inserts counterfeit data amongst valid data.

Assets and their Vulnerabilities

Hardware is mainly vulnerable to interruption, either by theft or by vandalism. Physical security measures are used to prevent these attacks.

Software is also vulnerable to interruption, as it is very easy to delete. Backups are used to limit the damage caused by deletion. Modification or fabrication through alteration (e.g. by viruses) is a major problem, as it can be hard to spot quickly. Software is also vulnerable to interception through unauthorised copying: this problem is still largely unsolved.

Data is vulnerable in many ways. Interruption can occur through the simple destruction of data files. Interception can occur through unauthorised reading of data files, or more perniciously through unauthorised analysis and aggregation of data. Modification and fabrication are also obvious problems with potentially huge consequences.

Communications are vulnerable to all types of threats.

Passive attacks take the form of eavesdropping, and fall into two categories: reading the contents of a message, or more subtly, analysing patterns of traffic to infer the nature of even secure messages.

Passive attacks are hard to detect, so the emphasis is usually on prevention.

Active attacks involve modification of a data stream, or creation of a false data stream. One entity may *masquerade* as another (presumably one with more or different privileges), maybe by capturing and replaying an authentication sequence. *Replay* is a similar attack, usually on data. Message contents may also be *modified*, often to induce incorrect behaviour in other users. *Denial of service* attacks aim to inhibit the normal use of communication facilities.

Active attacks are hard to prevent (entirely), so the emphasis is usually on detection and damage control.

Protection

Multiprogramming involves the sharing of many resources, including processor, memory, I/O devices, programs, and data. Protection of such resources runs along the following spectrum:

No protection may be adequate e.g. if sensitive procedures are run at separate times.

Isolation implies that entities operate separately from each other in the physical sense.

Share all or nothing implies that an object is either totally private or totally public.

Share via access limitation implies that different entities enjoy different levels of access to an object, at the gift of the owner. The OS acts as a guard between entities and objects to enforce correct access.

Share via dynamic capabilities extends the former to allow rights to be varied dynamically.

Limit use of an object implies that not only is access to the object controlled, the use to which it may be put also varies across entities.

The above spectrum is listed roughly in order of increasing fineness of control for owners, and also increasing difficulty of implementation.

Intruders

Intruders and viruses are the two most publicised security threats. We identify three classes of intruders:

A masquerador is an unauthorised individual (an outsider) who penetrates a system to exploit legitimate users' accounts.

A misfeasor is a legitimate user (an insider) who accesses resources to which they are not privileged, or who abuses such privilege.

A clandestine user is an individual (an insider or an outsider) who seizes control of a system to evade auditing controls, or to suppress audit collection.

Intruders are usually trying to gain access to a system, or to increased privileges to which they are not entitled, often by obtaining the password for a legitimate account. Many methods of obtaining passwords have been tried:

- trying default passwords;
- exhaustively testing short passwords;
- trying words from a dictionary, or from a list of common passwords;
- collecting personal information about users;
- using a Trojan horse;
- eavesdropping on communication lines.

The usual methods for protecting passwords are through one-way encryption, or by limiting access to password files. However, passwords are inherently vulnerable.

Malicious Software

The most sophisticated threats to computer systems are through malicious software, sometimes called *malware*. Malware attempts to cause damage to, or consume the resources of, a target system.

Malware can be divided into programs that can operate independently, and those that need a host program; and also into programs that can replicate themselves, and those that cannot.

A trap door is a secret entry point into a program, often left by the program's developers, or sometimes delivered via a software update.

A logic bomb is code embedded in a program that "explodes" when certain conditions are met, e.g. a certain date or the presence of certain files or users. Logic bombs also often originate with the developers of the software.

A Trojan horse is a useful (or apparently useful) program that contains hidden code to perform some unwanted or harmful function.

A virus is a program that can "infect" other programs by modification, as well as causing local damage. Such modification includes a copy of the virus, which can then spread further to other programs.

A worm is an independent program that spreads via network connections, typically using either email, remote execution, or remote login to deliver or execute a copy of itself to or on another system, as well as causing local damage.

A zombie is an independent program that secretly takes over a system and uses that system to launch attacks on other systems, thus concealing the original instigator. Such attacks often involve further replication of the zombie itself. Zombies are often used in denial-of-service attacks.

The last three of these involve replication. In all cases, prevention is **much** easier than detection and recovery.

Trusted Systems

So far we have discussed protecting a given resource from attack by a given user. Another requirement is to protect a resource on the basis of levels of security, e.g. the military-style system, where users are granted clearance to view certain categories of data.

This is known as *multi-level security*. The basic principle is that a subject at a higher level may not convey information to a subject at a lower level against the wishes of the authorised user. This principle has two facets:

No read-up implies that a subject can only read objects of less or equal security level.

No write-down implies that a subject can only write objects of greater or equal security level.

These requirements are implemented by a *reference monitor*, which has three roles:

Complete mediation implies that rules are imposed on every access.

Isolation implies that the monitor and database are protected from unauthorised modification.

Verifiability implies that the monitor is provably correct.

Such a system is known as a *trusted system*. These requirements are very difficult to meet, both in terms of assuring correctness and in terms of delivering adequate performance.

Protection and Security Design Principles

Saltzer and Schroeder (1975) identified a core set of principles to operating system security design:

Least privilege: Every object (users and their processes) should work within a minimal set of privileges; access rights should be obtained by explicit request, and the default level of access should be "none".

Economy of mechanisms: security mechanisms should be as small and simple as possible, aiding in their verification. This implies that they should be integral to an operating system's design, and not an afterthought.

Acceptability: security mechanisms must at the same time be robust yet non-intrusive. An intrusive mechanism is likely to be counter-productive and avoided by users, if possible.

Complete: Mechanisms must be pervasive and access control checked during all operations — including the tasks of backup and maintenance.

Open design: An operating system's security should not remain secret, nor be provided by stealth. Open mechanisms are subject to scrutiny, review, and continued refinement.

The Unix/Linux Security Model

Unix, in comparison to more modern operating systems such as Windows-NT, provides a relatively simple model of security.

System calls are the only mechanism by which processes may interact with the operating system and the resources it is protecting and managing.

Each user and each process executed on behalf of that user, is identified by (minimally) two non-negative 16-bit integers:

The user-identifier is established when logging into a Unix system. A correct combination of user-name and password when logging in, or the validation of a network-based connection, set the user-identifier (uid) in the process control block of the user's *login shell*, or *command interpreter*.

Unless modified, this user-identifier is inherited by all processes invoked from the initial login shell. Under certain conditions, the user-identifier may be changed and determined with the system calls *setuid()* and *getuid()*.

The effective user-identifier is, by default, the same as the user-identifier, but may be temporarily changed to a different value to offer temporary privileges.

The successful invocation of *set-user-id* programs, such as *passwd* and *login* will, typically, set the effective user-identifier for the lifetime of that process. Under certain conditions, the effective user-identifier may be changed and determined with the system calls *seteuid()* and *geteuid()*.

Properties of the Unix Superuser

Unix uses the special userid value of 0 to represent its only special user, the *superuser* (or *root*).

Processes acting on behalf of the superuser can often access additional resources (often incorrectly stated as "everything") because most system calls responsible for checking user permissions bypass their checks if invoked with $userid = 0$.

The result is that there appear to be no files, etc, that cannot be accessed from the superuser using the standard application programs which report and manipulate such resources.

Instead, attackers must attempt to hide additional or modified files using other techniques which typically exploit social engineering issues — playing on human nature to overlook or insufficiently check for problems.

Although the superuser has greater access to otherwise protected resources, the Unix kernel will not permit the superuser to undermine the integrity of the operating system itself.

For example, although the superuser can create a new file in any directory through a call to the *open()* or *creat()* system calls, and details of this new file are written to the directory by the kernel itself, the superuser cannot open and explicitly write to the directory.

Unix is frequently criticised for both having a concept such as a superuser, or for encouraging security practices which now rely on it.

It is thus the single greatest target of attack on a Unix system.

The Unix Security Model — Groups

Each process is also identified by a *primary group identifier* and a list of up to 32 (see *<asm/limits.h>*) *secondary group identifiers*.

Under Linux, each group identifier is a non-negative 16-bit integer. The unlimited membership of each group consists of *user identifiers*.

As with user identification, the login procedures establish the primary and secondary groups of the user's *login shell*, and these group identifiers are inherited by all processes invoked from the initial login shell.

Each process also has a single *effective group identifier*, which may be set by privileged programs or by invoking *set-group-id* programs.

The system calls of interest here are *setgid()*, *getgid()*, *seteuid()*, and *geteuid()*.

Information about the array of 32 secondary groups is set/read with *setgroups()* and *getgroups()*.

Protection For Unix Files and Directories

The use of user identifiers and group identifiers under Linux is most visible with regard to file system access.

All files and directories have certain access permissions which constrain access to only those users having the correct user and group permissions.

Let's consider a typical example, running */bin/ls -l*:

```
-rw-r--r--  1 chris  staff  8362 Oct 17 12:40 /tmp/textfile
```

The first character of the access permissions indicates what type of file is being displayed.

```
-    plain files (such as Java and C source code)
d    directories
c    character special files (such as terminals)
b    block special files (such as disk drives)
=    named pipes (FIFOs)
```

Each of the following three triples describes, from left to right, the *read*, *write*, and *execute* permission for (respectively) the owner, the file's group, and the "rest-of-the-world".

Each file and directory is owned by a single user, and considered to be "in" a single group (the owner does not have to be in that group). The UID and GID may be obtained via the *stat* system call.

The Meaning of Permissions

What do these permissions mean?

- Read permission means that the file may be displayed on the screen, copied to another file or printed on the printer — any operation which requires reading the contents of the file. Having read permission on a directory means that its contents may be listed — *ls* may read the file's names (and attributes).
- Write permission means that the file or directory may be modified, changed or overwritten. Most importantly, write permission means that a file may be deleted. Write permission on a directory gives permission to delete a file from within that directory, if the permission also exists for the file.
- Execute permission means that the file may be executed. Execute permission for a directory means that the user may change into that directory.
- Shellscripts must have both read and execute permission – *zsh* must both be able to read the shellscript and know to execute it.

Annoyingly, on different variants of Unix/Linux the permission mode bits, in combination, have some obscure meanings:

- having execute access, but not read access, to a directory still permits someone to "guess" filenames therein,
- having the sticky bit set on a directory permits only the owner of a file therein to remove or modify the file,
- having the setgid bit set on a directory means that files created in the directory receive the groupid of the directory, and not of their creator (owner).

Changing File and Directory Permissions

The permissions on files and directories may be changed with *chmod*, standing for "change mode". *chmod* is both the name of a command (Section 1 of the online manual), and the name of a system call (Section 2):

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

- Only the three permission triplets may be changed — a directory cannot be changed into a plain file nor vice-versa.
- Permissions given to the *chmod* command can be either absolute (described here) or relative.
- Each triplet is the sum of the octal digits 4, 2 and 1, read from left to right. For example *rwx* is represented by 7, *rw-* by 6 and *r–* by 4, and so on.

Octal value	**Protection**
400	Read by owner
200	Write (delete) by owner
100	Execute (search in directory) by owner
040	Read by group
020	Write (delete) by group
010	Execute (search) by group
004	Read by others
002	Write (delete) by others
001	Execute (search) by others

- The complete permission on any file or directory is the "sum" of the appropriate values.
- Home directories are typically 700 which provides you (the owner) read, write and execute (search) permission but denies all access by others.

```
chmod  700  /home/year2/charles-p
```

- If you wish others to read some of your files, set their mode to 644:

```
chmod  644  funny.jokes
```

Alternatively, we can set a file's permission bits using a symbolic notation:

```
chmod  u+rwx  /home/year2/charles-p

chmod  u+rw,g+r,o+r  funny.jokes

chmod  u+w,a+r  funny.jokes
```

Programatically, we can set a file's protection mode when it is initially created, and change it thereafter:

```
fd = open ("myfile", O_RDWR | O_CREAT, 0600);
...
(void) chmod ("myfile", 0644);
```

The Windows-NT Security Model

While the Unix security model provides system-wide and consistent support of user and group identification, it constrains their manipulation to the system administrator (root).

In contrast, the newer Windows-NT security model enables each authorised user (and process) to both examine and manipulate access to a variety of objects.

Again, the access controls provided by Windows-NT are best seen by examining the file system — but we must be using a partition supporting the Windows-NT File System (NTFS) and not simply a FAT-based (ala. Windows'98) partition.

Access controls in Windows-NT can be very specific – for example, considering a file on an NTFS volume, one can:

- let no one but the owner access it,
- let any single user access it,
- let several individual users access it,
- name an NT group and let any group members access the file,
- name an NT group and let any group members access the file, while also denying access to individual members of that group,
- name multiple groups that can access it, or
- let anyone, possibly excluding certain individuals, access it.

[See Stallings Pages 715–9]

A variety of objects under Windows-NT can have also locks applied to them.

Under Windows-NT, these locks are termed *security descriptors*. They may be set when an object is created, or they may be examined or set (with permission) once the object exists.

A security descriptor has four attributes.

- An owner identifier, indicating the current owner of the object (it can be given away),
- A primary group identifier,
- A *system access control list* (SACL) containing auditing information, and
- A *discretionary access control list* (DACL), that determines which users can and cannot access the object.

Security descriptors may be ascribed to files (under NTFS), directories (under NTFS), registry keys, processes, threads, mutexes and semaphores (synchronization objects), events, named pipes, anonymous pipes, mailslots, console screen buffers, file mappings, network services, and private objects(!).

Access control lists (ACLs)

An alternative to directories is for objects to manage their own access. An *access control list* (ACL) contains a list of which subjects (and their processes) may access the object, and in which ways.

There is one ACL for each object. Each list is traditionally maintained, and access checked, by the kernel, although it could be feasible for objects to maintain and constrain their own access (but remember that most objects are passive). ACLs

- will typically list valid access modes by individual users. The owner of an object may or may not have permission to modify the ACL of the object itself.
- may list valid access modes by whole named groups of users,
- similarly deny access by users even if they are members of permitted groups, and
- permit access using wildcards naming users and groups. Wildcards usually appear, or are evaluated, last so that access may be first be denied.

Searching an access control list is undertaken until a match of the requesting user and access mode is located. Default conditions (considered unwise) may be supported using "open" wildcards, or specific default access elements at the tail of each ACL.

In contrast to the use of directories, it is now more difficult for a subject to list all objects to which they have access, but easier for an object to determine which subjects may access it.

(ACLs for Linux filesystems are constantly discussed but never resolved; see *http://acl.bestbits.at*)

Access Tokens and User Rights

When a user logs into a Windows-NT system they are given an *access token*. Each process control block entry (i.e. each process) contains its default access token.

The access token identifies the user, the primary groups, such as *Power User*, *Backup Operator*, etc, and any custom groups, such as *year2*.

The access token also contains the user rights or privileges. These may be ascribed on a per-user or per-group basis, and include the abilities to:

- modify the token itself,
- create audit logs,
- perform backups,
- debug processes,
- change a process priority,
- change quotas,
- lock physical pages into memory,
- shutdown the system,
- take ownership of an object, and
- view security logs.

The Discretionary Access Control List

The Discretionary Access Control List (DACL) is the heart of Windows-NT security, determining who can and cannot access an object.

It is a list (actually stored as an array) of *access control entries* (ACEs) each of which indicates abilities of users and groups on that object.

For example, if the user *charles-p* may read a file, an *access allowed ACE* will permit this, while a separate *access denied ACE* may deny access to *diana-l*.

The System Access Control List

The System Access Control List (SACL) also contains ACEs, but these ACEs determine who (which users and groups) will be *audited*. An ACE in a SACL is termed an *audit access ACE*.

For example, an audit access ACE could indicate that every time *charles-p* reads an object (such as a file object), that information should be logged to a file.

Similarly, each time *diana-l* attempts to read that object, that attempt will be logged.