Access Control

Petr Ročkai

December 9, 2019

Lecture Overview

1. Multi-User Systems
2. File Systems
3. Sub-user Granularity

Part 1: Multi-User Systems

Users

- originally a proxy for people
- currently a more general abstraction
- user is the unit of ownership
- many permissions are user-centered

Computer Sharing

- computer is a (often costly) resource
- efficiency of use is a concern
    - a single user rarely exploits a computer fully
- data sharing makes access control a necessity

Ownership

- various objects in an OS can be owned
  - primarily files and processes
- the owner is typically whoever created the object
  - ownership can be transferred
  - usually at the impetus of the original owner

Process Ownership

- each process belongs to some user
- the process acts on behalf of the user
  - the process gets the same privilege as its owner
  - this both constrains and empowers the process
- processes are active participants

File Ownership

- each file also belongs to some user
- this gives rights to the user (or rather their processes)
  - they can read and write the file
  - they can change permissions or ownership
- files are passive participants

Access Control Models

- owners usually decide who can access their objects
  - this is known as discretionary access control
- in high-security environments, this is not allowed
  - known as mandatory access control
  - a central authority decides the policy

(Virtual) System Users

- users are an useful ownership abstraction
- various system services get their own “fake” users
- this allows them to own files and processes
- and also limit their access to the rest of the OS

Principle of Least Privilege

- entities should have minimum privilege required
  - applies to software components
  - but also to human users of the system
- this limits the scope of mistakes
  - and also of security compromises

Privilege Separation

- different parts of a system need different privilege
- least privilege dictates splitting the system
  - components are isolated from each other
  - they are given only the rights they need
- components communicate using simple IPC

Process Separation

- recall that each process runs in its own address space
  - but shared memory can be requested
- each user has a view of the filesystem
  - a lot more is shared by default in the filesystem
  - especially the namespace (directory hierarchy)

Access Control Policy

- there are 3 pieces of information
  - the subject (user)
  - the verb (what is to be done)
  - the object (the file or other resource)
- there are many ways to encode this information

Access Rights Subjects

- in a typical OS those are (possibly virtual) users
  - sub-user units are possible (e.g. programs)
  - roles and groups could also be subjects
- the subject must be named (names, identifiers)
  - easy on a single system, hard in a network

Access Rights Verbs

- the available “verbs” (actions) depend on object type
- a typical object would be a file
  - files can be read, written, executed
  - directories can be searched or listed or changed
- network connections can be established &c.

Access Rights Objects

- anything that can be manipulated by programs
  - although not everything is subject to access control
- could be files, directories, sockets, shared memory, …
- object names depend on their type
  - file paths, i-node numbers, IP addresses, …

Subjects in POSIX

- there are 2 types of subjects: users and groups
- each user can belong to multiple groups
- users are split into normal users and root
    - root is also known as the super-user

User Management

- the system needs a database of users
- in a network, user identities often need to be shared
- could be as simple as a text file
  - `/etc/passwd` and `/etc/group` on UNIX systems
- or as complex as a distributed database

User and Group Identifiers

- users and groups are represented as numbers
  - this improves efficiency of many operations
  - the numbers are called `uid` and `gid`
- those numbers are valid on a single computer
  - or at most, a local network

Changing Identities

- each process belongs to a particular user
- ownership is inherited across `fork()`
- super-user processes can use `setuid()`
- `exec()` can sometimes change a process owner

Login

- a super-user process manages user logins
- the user types their name and provides credentials
  - upon successful authentication, `login` calls `fork()`
  - the child calls `setuid()` to the user
  - and uses `exec()` to start a shell for the user

User Authentication

- the user needs to authenticate themselves
- passwords are the most commonly used method
  - the system needs to know the right password
  - user should be able to change their password
- biometric methods are also quite popular

Storing Passwords

- passwords are often stored as hashes
- along with salt, to counter rainbow tables
- on UNIX: `/etc/shadow` (only `root` can read)
- also: key derivation functions (`bcrypt`, `argon2`)

Remote Login

- authentication over network is more complicated
- passwords are easiest, but not easy
    - encryption is needed to safely transmit passwords
    - along with computer authentication
- 2-factor authentication is a popular improvement

Computer Authentication

- how to ensure we send the password to the right party?
  - an attacker could impersonate our remote computer
- usually via asymmetric cryptography
  - a private key can be used to sign messages
  - the server will sign a message establishing its identity

2-factor Authentication

- 2 different types of authentication
  - harder to spoof both at the same time
- there are a few factors to pick from
  - something the user knows (password)
  - something the user has (keys)
  - what the user is (biometric)

Enforcement: Hardware

- all enforcement begins with the hardware
  - the CPU provides a privileged mode for the kernel
  - DMA memory and IO instructions are protected
- the MMU allows the kernel to isolate processes
  - and protect its own integrity

Enforcement: Kernel

- kernel uses hardware facilities to implement security
  - it stands between resources and processes
  - access is mediated through system calls
- file systems are part of the kernel
- user and group abstractions are part of the kernel

Enforcement: System Calls

- the kernel acts as an arbitrator
- a process is trapped in its own address space
- processes use system calls to access resources
  - kernel can decide what to allow
  - based on its access control model and policy

Enforcement: Service APIs

- userland processes can enforce access control
  - usually system services which provide IPC API
- e.g. via the `getpeereid()` system call
  - tells the caller which user is connected to a socket
  - user-level access control is rooted in kernel facilities

Part 2: File Systems

File Access Rights

- file systems are a case study in access control
- all modern file systems maintain permissions
  - the only extant exception is FAT (USB sticks)
- different systems adopt different representation

Representation

- file systems are usually object-centric
  - permissions are attached to individual objects
  - easily answers “who can access this file”?
- there is a fixed set of verbs
  - those may be different for files and directories
  - different systems allow different verbs

The UNIX Model

- each file and directory has a single owner
- plus a single owning group
  - not limited to those the owner belongs to
- ownership and permissions are attached to i-nodes

Access vs Ownership

- POSIX ties ownership and access rights
- only 3 subjects can be named on a file
  - the owner (user)
  - the owning group
  - anyone else

Access Verbs in POSIX File Systems

- read: read a file, list a directory
- write: write a file, link/unlink i-nodes to a directory
- execute: `exec` a program, enter the directory
- execute as owner (group): `setuid`/`setgid`

Permission Bits

- basic UNIX permissions can be encoded in 9 bits
- 3 bits per 3 subject designations
  - first comes the owner, then group, then others
  - written as e.g. `rwxr-x---` or `0750`
- plus two numbers for the owner/group identifiers

Changing File Ownership

- the owner and `root` can change file owners
- `chown` and `chgrp` system utilities
- or via the C API
  - `chown()`, `fchown()`, `fchownat()`, `lchown()`
  - same set for `chgrp`

Changing File Permissions

- again available to the owner and to `root`
- `chmod` is the user space utility
  - either numeric argument: `chmod 644 file.txt`
  - or symbolic: `chmod +x script.sh`
- and the corresponding system call (numeric-only)

`setuid` and `setgid`

- special permissions on executable files
- they allow `exec` to also change the process owner
- often used for granting extra privileges
  - e.g. the `mount` command runs as the super-user

Sticky Directories

- file creation and deletion is a directory permission
  - this is problematic for shared directories
  - in particular the system `/tmp` directory
- in a sticky directory, different rules apply
  - new files can be created as usual
  - only the owner can unlink a file from the directory

Access Control Lists

- ACL is a list of ACE's (access control elements)
  - each ACE is a subject + verb pair
  - it can name an arbitrary user
- ACL is attached to an object (file, directory)
- more flexible than the traditional UNIX system

ACLs and POSIX

- part of POSIX.1e (security extensions)
- most POSIX systems implement ACLs
  - this does not supersede UNIX permission bits
  - instead, they are interpreted as part of the ACL
- file system support is not universal (but widespread)

Device Files

- UNIX represents devices as special i-nodes
  - this makes them subject to normal access control
- the particular device is described in the i-node
  - only a super-user can create device nodes
  - users could otherwise gain access to any device

Sockets and Pipes

- named sockets and pipes are just i-nodes
  - also subject to standard file permissions
- especially useful with sockets
  - a service sets up a named socket in the file system
  - file permissions decide who can talk to the service

Special Attributes

- flags that allow additional restrictions on file use
  - e.g. immutable files (cannot be changed by anyone)
  - append-only files (for logfile integrity protection)
  - compression, copy-on-write controls
- non-standard (Linux `chattr`, BSD `chflags`)

Network File System

- NFS 3.0 simply transmits numeric uid and gid
  - the numbering needs to be synchronised
  - can be done via a central user database
- NFS 4.0 uses per-user authentication
  - the user authenticates to the server directly
  - filesystem uid and gid values are mapped

File System Quotas

- storage space is limited, shared by users
  - files take up storage space
  - file ownership is also a liability
- quotas set up limits space use by users
  - exhausted quota can lead to denial of access

Removable Media

- access control at file system level makes no sense
  - other computers may choose to ignore permissions
  - user names or id's would not make sense anyway
- option 1: encryption (for denying reads)
- option 2: hardware-level controls
  - usually read-only vs read-write on the entire medium

The chroot System Call

- each process in UNIX has its own root directory
  - for most, this coincides with the system root
- the root directory can be changed using `chroot()`
- can be useful to limit file system access
  - e.g. in privilege separation scenarios

Uses of chroot

- chroot alone is not a security mechanism
    - a super-user process can get out easily
    - but not easy for a normal user process
- also useful for diagnostic purposes
- and as lightweight alternative to virtualisation

Part 3: Sub-User Granularity

Users are Not Enough

- users are not always the right abstraction
  - creating users is relatively expensive
  - only a super-user can create new users
- you may want to include programs as subjects
  - or rather, the combination user + program

Naming Programs

- users have user names, but how about programs?
- option 1: cryptographic signatures
  - portable across computers but complex
  - establishes identity based on the program itself
- option 2: i-node of the executable
  - simple, local, identity based on location

Program as a Subject

- program: passive (file) vs active (processes)
  - only a process can be a subject
  - but program identity is attached to the file
- rights of a process depend on its program
  - `exec()` will change privileges

Mandatory Access Control

- delegates permission control to a central authority
- often coupled with security labels
  - classifies subjects (users, processes)
  - and also objects (files, sockets, programs)
- the owner cannot change object permissions

The Bell-LaPadula Model

1. simple security property
   - you can't read what is beyond your clearance
2. the star property
   - also called no write down
   - you cannot write to 'more public' files

Capabilities

- not all verbs (actions) need to take objects
- e.g. shutting down the computer (there is only one)
- mounting file systems (they can't be always named)
- listening on ports with number less than 1024

Dismantling the root User

- the traditional root user is all-powerful
  - “all or nothing” is often unsatisfactory
  - violates the principle of least privilege
- many special properties of root are capabilities
  - root then becomes the user with all capabilities
  - other users can get selective privileges

Security and Execution

- security hinges on what is allowed to execute
- arbitrary code execution are the worst exploits
  - this allows unauthorized execution of code
  - same effect as impersonating the user
  - almost as bad as stolen credentials

Untrusted Input

- programs often process data from dubious sources
  - think image viewers, audio & video players
  - archive extraction, font rendering, …
- bugs in programs can be exploited
  - the program can be tricked into executing data

Process as a Subject

- some privileges can be tied to a particular process
  - those only apply during the lifetime of the process
  - often restrictions rather than privileges
  - this is how privilege dropping is done
- processes are identified using their numeric `pid`
  - restrictions are inherited across `fork()`

Sandboxing

- tries to limit damage from code execution exploits
- the program drops all privileges it can
  - this is done before it touches any of the input
  - the attacker is stuck with the reduced privileges
  - this can often prevent a successful attack

Untrusted Code

- traditionally, you would only execute trusted code
  - usually based on reputation or other external factors
  - this does not scale to a large number of vendors
- it is common to execute untrusted, even dubious code
  - this can be okay with sufficient sandboxing

API-Level Access Control

- capability system for user-level resources
  - things like contact lists, calendars, bookmarks
  - objects not provided directly by the kernel
- enforcement e.g. via a virtual machine
  - not applicable to execution of native code
  - alternative: an IPC-based API

Android/iOS Permissions

- applications from a store are semi-trusted
- typically single-user computers/devices
- permissions are attached to apps instead of users
- partially virtual users, partially API-level