File Permissions

- Besides user authentication, the most visible aspect of OS security
- Read protection — provide confidentiality
- Write protection — provide integrity protection
- Other permissions as well

What Do We Protect?

- Most obvious — files
- That can be done in non-hierarchical file systems
- In hierarchical file systems, must protect directories, too
- Often, other things protected via similar mechanisms, such as shared memory segments

Classical Unix File Permissions

- All files have "owners"
- All files belong to a "group"
- Users, when logged in, have one userid and several groupids.
- 3 sets of 3 bits: read, write, execute, for user, group, other
- (512 possible settings. Do they all make sense?)
- Written `rwxrwxrwx`
- 111 101 001 (751 octal): User has read/write/exec; group has read/exec; other has exec-only
- Some counter-intuitive settings are very useful

3 / 41

Permission-Checking Algorithm

```
if curr_user.uid == file.uid
        check_owner_permissions();
else if curr_user.gid == file.gid
        check_group_permissions();
else
        check_other_permissions();
fi
```

Note the `else` clauses — if you own a file, "group" and "other" permissions aren't checked

4 / 41

Execute Permission

- Why is it separate from “read”?
- To permit *only* execution
- Cannot copy the file
- Readable only by the OS, for specific purposes

5 / 41

Directory Permissions

- “write”: create a file in the directory
- “read”: list the directory
- “execute”: trace a path through a directory

6 / 41

Example: Owner Permissions

```
$ id
uid=54047(smb) gid=54047(smb) groups=0(wheel),3(sys),54047(smb)
$ ls -l not_me
----r--r--  1 smb  wheel  29 Sep 12 01:35 not_me
$ cat not_me
cat: not_me: Permission denied
```

I own the file but don't have read permission on it

Example: Directory Permissions

```
$ ls -ld oddball
dr--r--r--  2 smb  wheel  512 Sep 12 01:36 oddball
$ ls oddball
cannot_get_at
$ ls -l oddball
ls: cannot_get_at: Permission denied
$ cat oddball/cannot_get_at
cat: oddball/cannot_get_at: Permission denied
```

I can read the directory, but not trace a path through it to oddball/cannot_get_at

Deleting Files

- What permissions are needed to delete files?
- On Unix, you need write permission on the parent directory
- You can delete files that you can't write. You can also write to files that you can neither create nor delete
- Other systems make this choice differently

9 / 41

When Are Permissions Checked?

- Most of the time, permissions are checked only at file open time
- Changing permissions on an open file usually does not block further access
- Better for efficiency — no need to check each time
- But for some file systems, such as NFS, file permission changes do take effect immediately

10 / 41

Access Control Lists

- 9-bit model not always flexible enough
- Many systems (Multics, Windows XP, Solaris, some Linux) have more general *Access Control Lists*
- ACLs are explicit lists of permissions for different parties
- Wildcards are often used

Sample ACL

`smb.*`	rwx
`4118-ta.*`	rwx
`*.faculty`	rx
`*.*`	x

Users "smb" and '4118-ta" have read/write/execute permission. Anyone in group "faculty" can read or execute the file. Others can only execute it.

Order is Significant

With this ACL:

```
*.faculty  rx
smb.*      rwx
4118-ta.*  rwx
*.*        x
```

I would not have write access to the file

13 / 41

Some Other Possible Permissions

Append: Append to a file, but not overwrite it
Delete: Delete file from directory
Own: Own the file; can change its permissions

14 / 41

Setting File Permissions

- Where do initial file permssions come from?
- Who can change file permissions?

15 / 41

Unix Initial File Permissions

- Unix uses "umask" — a set of bits to *turn off* when a program creates a file
- Example: if umask is 022 and a program tries to create a file with permissions 0666 (rw for user, group, and other), the actual permissions will be 0644.
- Default system umask setting has a great effect on system file security
- Set your own value in startup script; value inherited by child processes

16 / 41

Multics Initial File Permissions

- Directories contain "initial access control list" — values set by default for new files
- Common setting:

```
smb.faculty  rw
*.sysdaemon  r
*.*          -
```

- If group "sysdaemon" doesn't have read permission, the file can't be backed up!

17 / 41

Other Access Controls

18 / 41

MAC versus DAC

- Who has the right to set file permissions?
- Discretionary Access Control — the file owner can set permissions
- Mandatory Access Control — only the security officer can set permissions
- *Enforce* site security rules
- Note: viruses and other malware change change DAC permissions, but *not* MAC permissions

18 / 41

Privileged Users

- `Root` or `Administrator` can override file permissions
- This is a serious security risk — there is no protection if a privileged account has been compromised
- There is also no protection against a rogue superuser. . .
- Secure operating systems do not have the concept of superusers

19 / 41

Complex Access Control

- Simple user/group/other or simple ACLs don't always suffice
- Some situations need more complex mechanisms

20 / 41

Temporal Access Control

- Permit access only at certain times
- Model: time-locks on bank vaults

21 / 41

Implementing Temporal Access Control

- Obvious way: add extra fields to ACL
- Work-around: timer-based automatic job that changes ACLs dynamically

22 / 41

Access Control Matrices

Access Control Matrix

- List all proceses and files in a matrix
- Each row is a process ("subject")
- Each column is a file ("object")
- Each matrix entry is the access rights that subject has for that object

23 / 41

Sample Access Control Matrix

Subjects p and q
Objects f, g, p, q
Access rights r, w, x, o

	f	g	p	q
p	rwo	r	rwx	w
q	-	r	r	rwxo

24 / 41

Access Control Matrix Operations

- System can transition from one ACM state to another
- Primitive operations: create subject, create object; destroy subject, destroy object; add access right; delete access right
- Transitions are, of course, conditional

Conditional ACM Changes

Process p wishes to give process q read access to a file f owned by p.

command *grant_read_file(p, f, q)*
  if o **in** $a[p, f]$
  then
    enter r **into** $a[q, f]$
  fi
end

Safety versus Security

- *Safety* is a property of the abstract system
- *Security* is a property of the implementation
- To be secure, a system must be safe *and* not have any access control bugs

27 / 41

Undecidable Question

- Query: given an ACM and a set of transition rules, will some access right ever end up in some cell of the matrix?
- Model ACM and transition rules as Turing machine
- Machine will halt if that access right shows up in that cell
- Will it ever halt?
- Clearly undecidable
- Conclusion: We can never tell if an access control system is safe (Harrison-Ruzzo-Ullman (HRU) result)

28 / 41

Virtual File System

- Linux supports very many different file system
- Examples: ext2 and ext3 (primary native file systems), FAT and NTFS (Windows), CD and DVD, many more
- Also support special file systems such as `/proc`

A Common Model

- Clearly, each file system type needs some special code — a Unix directory looks nothing like a FAT
- Just as clearly, we do not want everything to be different
- Solution: the *Virtual File System* (VFS)
- A common abstraction layer for all Linux file systems
- All higher-layer functions call the file system-specific implementations of the various VFS functions

VFS Objects

Superblock	Information about the file system itself
i-node	Information about specific files
file	The data itself
dentry	Directory entry

32 / 41

VFS Operations

- Most file-related system calls go through the VFS layer
- Some map directly to underlying file system; some must be emulated
- Example: FAT file systems don't have `..`, but `..` still has to work in paths
- Similarly, non-Unix file systems don't have Unix-style permissions, but `ls -l` has to say something

33 / 41

Creating Ownership and Permissions

- Where do file owner/group and permissions come from on, say, a FAT file system?
- Linux synthesizes them.
- User and group come from mount options (NetBSD uses the user and group of the mount point)
- Permissions are synthesized from things like read-only status and the `umask` specified at mount time

34 / 41

Sample Operation: Lookup

- Converts a path name to an i-node
- Must check permissions as it goes
- Must honor common directory entry (dentry) cache

35 / 41

Dentry Cache

- Directory lookups are very common
- Results are cached
- Cache validity has to be checked, in case the file was deleted, renamed, changed, etc

Lookup

- Many levels of preliminary subroutines
- The real work is in `fs/namei.c:__link_path_walk()`
- At each level, it checks the directory's execute permission
- Note: this is faked by lower layers for non-Unix file systems
- This routine handles `.` and `..`
- As needed, it (indirectly) calls the VFS lookup routine

Permissions

- Primary routine: `fs/namei.c:permission()`
- Looks for permission routine for this i-node (originally set via VFS)
- If not there, calls `generic_permission()` to check user/group/other bits
- Then checks ACLs

38 / 41

Extended Attributes

- Actually, Linux doesn't have ACLs per se
- It has *extended attributes* for files
- Extended attributes are name:value pairs
- Names are qualified by namespaces, such as `system.posix_acl_access`

39 / 41

Special File Systems

- Linux uses a variety of special file systems for various things
- Example: `/proc` and `sysfs` provide access to system data
- The debugger can use `/proc` to connect to a given process

40 / 41

Implementing Special File Systems

- At higher layers, just like real file systems
- But — fs-specific routines consult other data structures, rather than a real disk
- Can use Unix permissions to restrict access to some "files"
- Example: `/proc/$$/mem` is the current shell's memory; it's typically readable only the the owner

41 / 41