Authorization

• Protecting *what* from *whom*
  – protecting *objects* from *subjects*
    - subjects
      • processes
      • threads
    - objects
      • files
      • web sites
      • processes
      • threads
Access Matrix

<table>
<thead>
<tr>
<th></th>
<th>/a/b/c</th>
<th>/x/y/z</th>
<th>Process 112</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grace</td>
<td>rw</td>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>Anita</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ada</td>
<td></td>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>Barbara</td>
<td>r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grace’s protection domain

/a/b/c’s ACL
### Subjects Labeling Rows

<table>
<thead>
<tr>
<th>Subject</th>
<th>/a/b/c</th>
<th>/x/y/z</th>
<th>Process 112</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 112</td>
<td>rw</td>
<td></td>
<td>rw</td>
</tr>
<tr>
<td>Process 13452</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 23293</td>
<td></td>
<td></td>
<td>rw</td>
</tr>
<tr>
<td>Process 26421</td>
<td>r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process 112’s C-list:
- /a/b/c: rw
- Process 112: rw
Principle of least privilege

make the protection domain as small as possible

the capability list contains only what’s absolutely necessary
Caveat

• Meaning of “privilege” has changed in the past 30 years
  – 30 years ago
    - anything a process could do (such as file access) was labeled a privilege
  – now
    - a privilege is the ability to do something that affects the system as a whole
      • superuser privilege in Unix
      • administrator privilege in Windows
      • set-system-clock privilege
      • backup-files privilege
Modern OSes …

• Principle of least privilege
  – run code in smallest protection domain
    - Windows: many users run with “administrator” privilege
    - Unix and Windows: no smaller protection domain than that of a user

• Better use of hardware protection
  – data, such as stacks, should not be executable
Access Control

• Two approaches
  – who you are
    - subjects’ identity attributes determine access to objects
  – what you have
    - capabilities possessed by subjects determine access to objects
Who-You-Are-Based Access Control

• Discretionary access control (DAC)
  – objects have owners
  – owners determine who may access objects and how they may access them

• Mandatory access control (MAC)
  – system-wide policy on who may access what and how
  – object owners have no say
Access Control in Traditional Systems

• Unix and Windows
  – primarily DAC
  – file descriptors and file handles provide capabilities
  – MAC becoming more popular
    - SELinux
    - Windows
Unix

• Process’s security context
  – user ID
  – set of group IDs
  – more discussed later

• Object’s authorization information
  – owner user ID
  – group owner ID
  – permission vector
Is there a means in Unix to specify that members of two different groups have read access to a file, without resorting to features we haven’t yet discussed?

a) No, it can’t be done
b) Yes, but it’s complicated
c) Yes, it’s just as easy as specifying that the file is readable by just one group
Initializing Authorization Info

- permission_vector = mode & ~umask
  - mode is from open system call
- Owner user ID
  - effective user ID of creating process
- Group owner ID
  - “set either to the effective group ID of the process or to the group ID of the parent directory (depending on file system type and mount options, and the mode of the parent directory, see the mount options bsdgroups and sysvgroups described in mount(8))”
    - Linux man page for open(2)
Security Identifier (SID)

- Identify principals (users, groups, etc.)
- S-V-Auth-SubAuth_1-SubAuth_2-…-SubAuth_n-RID
  - S: they all start with “S”
  - V: version number (1)
  - Auth: 48-bit identifier of agent who created SID
    - local system
    - other system
  - SubAuth: 32-bit identifier of subauthority
    - subsystem, etc.
  - RID: relative identifier
    - makes it unique
    - user number, group number, etc.

• S-1-5-123423890-907809-43
Security Descriptor

• Owner’s SID
• DACL
  – discretionary access-control list
• SAACL
  – system access-control list
    - controls auditing
    - more later
• Flags
DACLS

• Sequence of ACEs — access-control entries
• Each indicates
  – who it applies to
    - SID of user, group, etc.
  – what sort of access
    - bit vector
  – action
    - permit or deny
Initializing DACLs

• Individual ACEs in directories may be marked inheritable

• When an object is created, DACL is initialized
  – explicitly provided ACEs appear first
  – then any ACEs inherited from parent
  – then any ACEs inherited from grandparent
  – etc.
Decision Algorithm

accesses_permitted = null
walk through the ACEs in order
   if access token’s user SID or group SID match ACE’s SID
      if ACE is of type access-deny
         if a requested access type is denied
            Stop — access is denied
      if ACE is of type access-allow
         if a requested access type is permitted
            add access type to accesses_permitted
            if all requested accesses are permitted
               Stop — access is allowed
      if not all requested access types permitted
         Stop — access is denied
Order Matters …

![Diagram showing order matters in access control]

- **Left Side:**
  - allow
  - inGroup
  - read, write
  - deny
  - Mary
  - read, write

- **Right Side:**
  - deny
  - Mary
  - read, write
  - allow
  - inGroup
  - read, write
Preferred Order

- *Access-denied* entries first
- *Access-allowed* entries second
- However …
  - not enforced
  - system GUIs don’t show order
  - only way to find out is to ask for “effective permissions”
There’s More

• ACE inheritance
  – designated ACEs propagate down tree
  – an object’s ACL can be flagged “protected”
    - no inheritance
  – an object may have an “inherit-only” ACL
    - applies to descendants, not to itself
  – revised preferred order
    - first explicit ACEs
    - then ACEs inherited from parent
    - then ACEs inherited from grandparent
    - etc.
    - within group, first access-denied, then access permitted
Unix ACLs

• POSIX 1003.1e
  – deliberated for 10 years
    - what to do about backwards compatibility?
  – gave up …
  – but implemented, nevertheless
    - setfacl/getfacl commands in Linux
Unix ACLs

• ACEs
  – user_obj: applies to file’s owner
  – group_obj: applies to file’s group
  – user: applies to named user
  – group: applies to named group
  – other: applies to everyone else
  – mask: maximum permissions granted to user, group_obj, and group entries
Unix ACLs

- Access checking
  - if effective user ID of process matches file’s owner
    - *user_obj entry* determines access
  - if effective user ID matches any *user ACE*
    - *user entry* ANDed with *mask* determines access
  - if effective group ID or supplemental group matches file’s group or any *group ACE*
    - access is intersection of *mask* and the union of all matching group entries
  - otherwise, *other ACE* determines access
Example

% mkdir dir
% ls -ld dir
drwxr-x--- 2 twd fac 8192 Mar 30 12:11 dir
% setfacl -m u:nlindsay:rwx dir
% ls -ld dir
drwxr-x++++ 2 twd fac 8192 Mar 30 12:16 dir
% getfacl dir
# file: dir
# owner: twd
# group: cs-fac
user::rwx
user:nlindsay:rwx
group::r-x
mask::r-x
other::---
Example (continued)

% setfacl -dm u::rwx,g::rx,u:nlindsay:rwx dir
% getfacl dir
# file: dir
# owner: twd
# group: cs-fac
user::rwx
user:nlindsay:rwx
group::r-x
mask::rwx
other::---
default:user::rwx
default:user:nlindsay:rwx
default:group::r-x
default:mask::rwx
default:other::---
Example (continued)

% cd dir
% cp /dev/null file # creates file with mode = 0666
% ls -l
total 0
-rw-rw----+ 1 twd fac 0 Mar 30 12:16 file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:nlindsay:rwx          #effective:rw-
group::r-x                #effective:r--
mask::rw-
other::---
Example (continued)

% cd dir
% new file 0466 # creates file with mode = 0466
% ls -l
total 0
-rw-rw----+ 1 twd fac 0 Mar 30 12:16 file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:nlindsay:rwx  #effective:rw-
group::r-x  #effective:r--
mask::rw-
other::---
Example (and still continued)

% setfacl -m o:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:nlindsay:rwx
group::r-x
mask::rwx
other::rw-
Example (and still continued)

% setfacl -m g:cs1670ta:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:nlindsay:rwx
group::r-x
group:cs-1670ta:rw-
mask::rwx
other::rw-
Example (end)

% setfacl -m m:r file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:nlindsay:rwx #effective:r--
group::r-x #effective:r--
group:cs-1670ta:rw- #effective:r--
mask::r--
other::rw-
Quiz 2

Unlike Windows ACLs, UNIX ACLs have no deny entries. Is it possible to set up an ACL that specifies that everyone in a particular group has rw access, except that a certain group member has no access at all?

a) No, it can’t be done
b) Yes, but it’s complicated
c) Yes, it’s easy
NFSv4 ACLs

- NFSv4 designers wanted ACLs
  - on the one hand, NFS is used by Unix systems
  - on the other hand, they’d like it to be used on Windows systems
  - solution:
    - adapt Windows ACLs for Unix
    - NFSv4 servers handle both Unix and Windows clients
    - essentially Windows ACLs plus Unix notions of file owner and file group
ACLs at Brown CS

• Linux systems support POSIX ACLs
• Windows systems support Windows ACLs
• Servers run GPFS file system and handle NFSv3 and CIFS clients
  – GPFS support NFSv4 ACLs
  – translated to POSIX ACLs and Unix bit vectors for NFSv3 clients
  – translated to Windows ACLs for CIFS clients
Extending the Basic Models

• Provide a file that others may write to, but only if using code provided by owner

• Print server
  – pass it file names
  – print server may access print files if and only if client may

• Password-changing program
Superuser (Unix)

• User ID == 0
  – bypasses all access checks
  – can send signals to any process
Attaining Super (or Lesser) Powers

• Setuid protection bit
  – the exec’ing process’s UID is set to owner of file
User and Group IDs

• Real user and group IDs — usually used to identify who created the process

• Effective user and group IDs — used to determine access rights to files

• Saved user and group IDs — hold the initial effective user and group IDs established at the time of the exec, allowing one to revert back to them
Exec

• Normally the real and effective IDs are the same
  – they are copied to the child from the parent during a *fork*

• *execs* done on files marked *setuid* or *setgid* change this
  – if the file is marked *setuid*, then the effective and saved user IDs become the ID of the owner of the file
  – if the file is marked *setgid*, then the effective and saved group IDs become the ID of the group of the file
Exercise of Powers

- Permission to access a file depends on a process’s effective IDs
  - the access system call checks permissions with respect to a process’s real IDs
    - this allows setuid/setgid programs to determine the privileges of their invokers

- The kill system call makes use of both forms of user ID; for process A to send a signal to process B, one of the following must be true:
  - A’s real user ID is the same as B’s real or saved user ID
  - A’s effective user ID is the same as B’s real or saved user ID
  - A’s effective user ID is 0
Race Conditions

// a setuid-root program:

if (access("/tmp/mytemp",
          W_OK) == 0) {
  // ... fail
}

fd = open("/tmp/mytemp",
          O_WRITE|O_APPEND);

len = read(0, buf,
           sizeof(buf));

write(fd, buf, len);

// another program:

unlink("/tmp/mytemp");
symlink("/etc/passwd",
        "/tmp/mytemp");

• TOCTTOU vulnerability
  • time of check to time of use ...
Changing Identity (1)

- The `setuid` and `setgid` system calls give a process a limited ability to change its IDs

```c
int setuid(uid_t uid)
```

```c
int setgid(gid_t gid)
```

- if the caller is super user, then these calls set the real, effective, and saved IDs
- otherwise, these calls set only the effective IDs and do so only if the caller’s real, saved, or effective ID is equal to the argument
Changing Identity (2)

• The `seteuid` and `setegid` system calls are the same except that they change only the effective IDs.

• The system calls `getuid`, `getgid`, `geteuid`, and `getegid` respectively return the real user ID, the real group ID, the effective user ID, and the effective group ID of the caller.
Avoiding the Race Condition

```c
uid_t caller_id = getuid();
uid_t my_id = geteuid();
seteuid(caller_id);
fd = open("/tmp/mytemp", O_WRITE|O_APPEND);
if (fd == -1) {
    // fail ...
}
seteuid(my_id);
len = read(0, buf, sizeof(buf));
write(fd, buf, len);
```
Unix Security Context

• Security context of a process
  – real user and group IDs
  – effective user and group IDs
  – saved user and group IDs
  – more?
More …

– supplementary groups
– alternate root
– file-descriptor table
– privileges
  - super user at finer granularity
  - called capabilities in Linux
Same But Different

/* handin: a setuid-twd program */

if (access(argv[1], R_OK) == 0) {
    // ... fail
}

fd = open(argv[1], O_RDONLY);
/* copy argv[1] to course directory */

% handin my_asgn

... % handin my_asgn

// another program:

unlink("my_asgn");
symlink("/u/twd/solution", "my_asgn");
How to Solve?

• Could use previous solution
  or
• Rewrite handin to read file from stdin
  – caller must open it
Same But Even More Different

You (Client Process) → Handin (Server Process)

file-descriptor table

my_asgn

file-descriptor table
File Descriptor as **Capability**

You (Client Process)

Handin (Server Process)

file-descriptor table

my_asgn

file-descriptor table