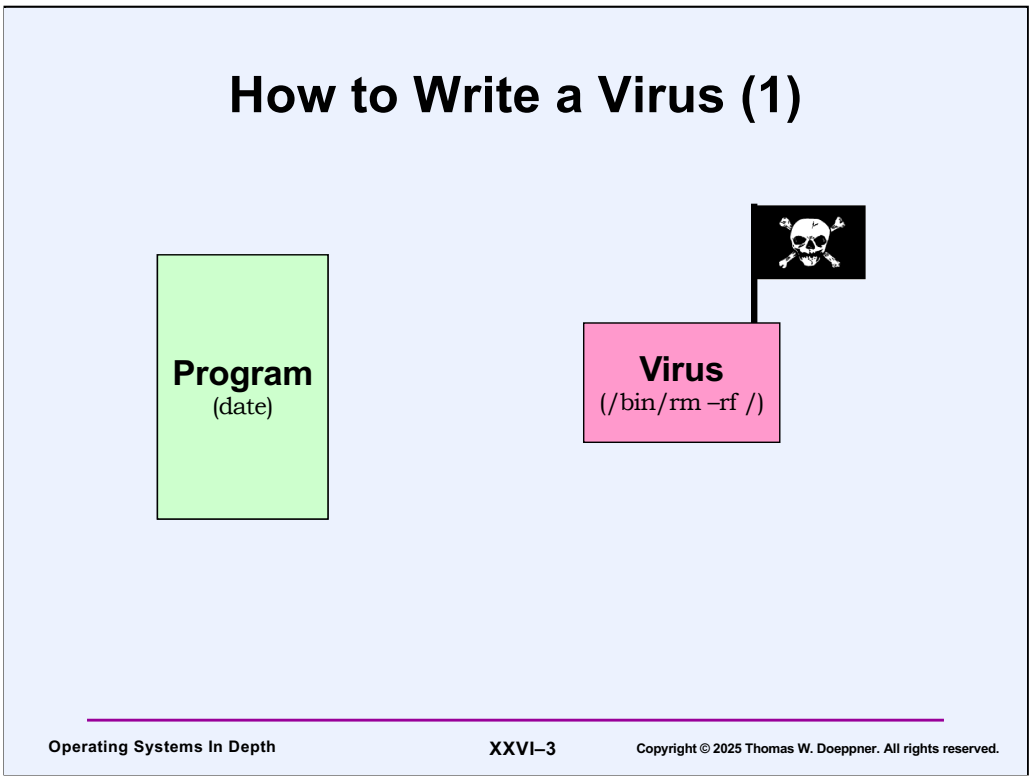
Security Part 2

Live Anonymous Q&A:
https://tinyurl.com/cs1670feedback

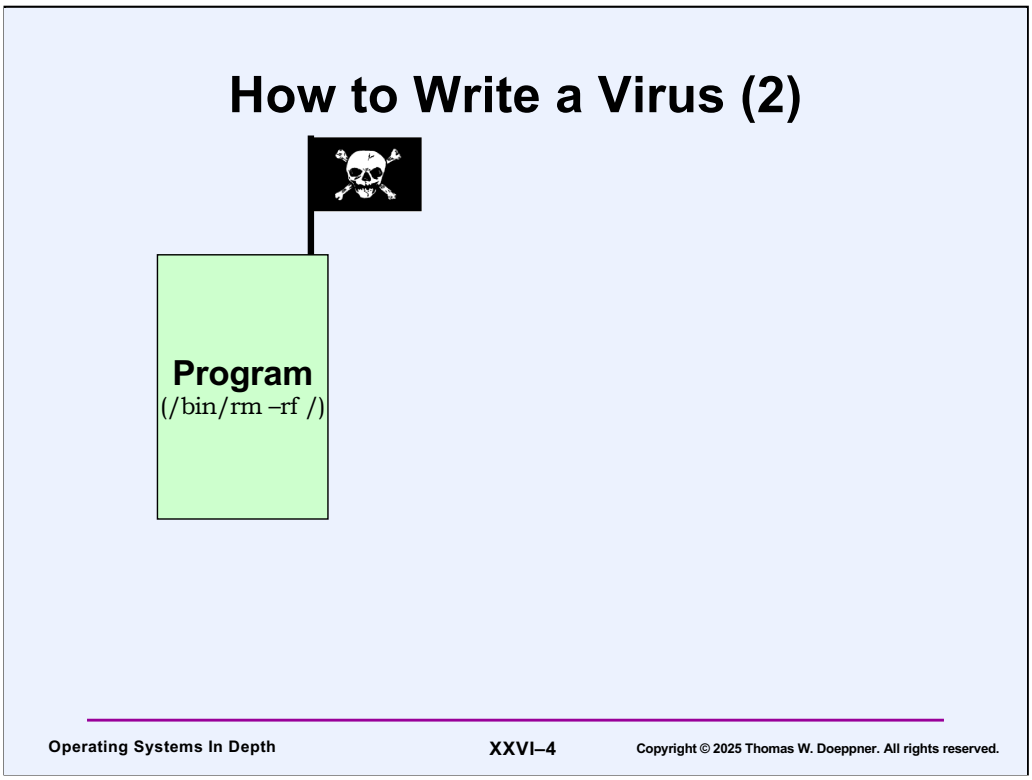
Viruses and Worms

- **Virus: an “infection” of a program that replicates itself**
- **Worm: a standalone program that actively replicates itself**

Our discussion of viruses and worms is at a very high level and is meant to define the concepts and is not a “how to”!

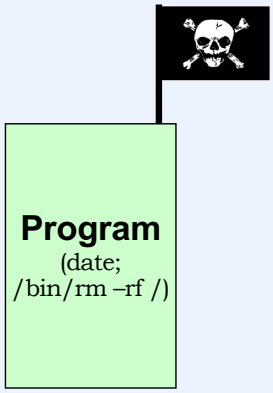


We'd like to modify the Unix "date" command so that it deletes all files on the file system.



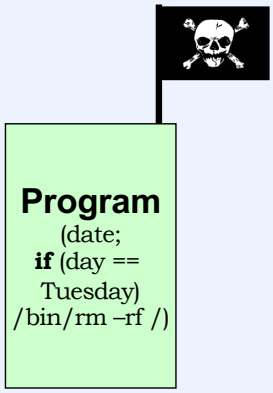
We've done that, but if someone runs it, it's pretty obvious as to which commend caused the problem.

How to Write a Virus (3)



So we make sure that the command at least does what it's supposed to do (in addition to delete all the files.

How to Write a Virus (4)



But to make it even less obvious that this command caused the problem, let's have it wreak havoc only occasionally, such as just on Tuesdays.

How to Write a Virus (5)

Program
(date;
if (day ==
Tuesday)
/bin/rm –rf /;
infect
others)

But for it to be an effective virus, it should spread, i.e., others should be infected.

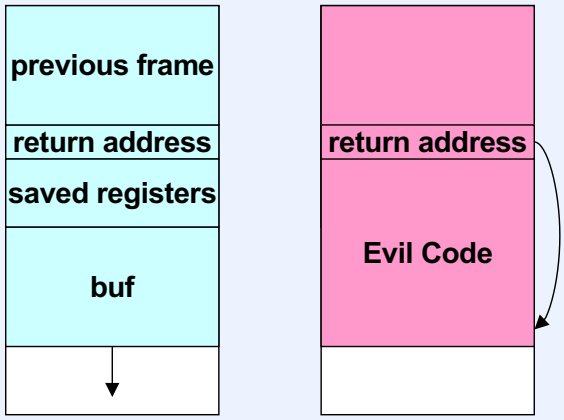
Further Issues

- Make program appear unchanged
  - don't change creation date
  - don't change size
- How to infect others
  - email
  - web
  - direct attack
  - etc.

Exploiting and Protecting Processes

Buffer Overflow

```
void fingerd( ) {
    char buf[80];
    …
    gets(buf);
    …
}
```



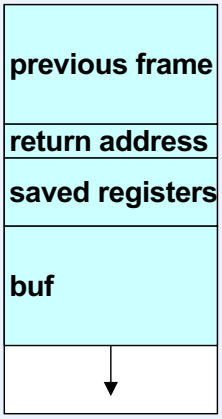
Programs susceptible to buffer-overflow attacks are amazingly common and thus such attacks are probably the most common of the bug-exploitation techniques. Even drivers for network interface devices have such problems, making machines vulnerable to attacks by maliciously created packets.

The function shown here, **fingerd**, was one of the primary culprits in the infamous Internet worm attack in 1988, in which pretty much all computers connected to the internet were successfully attacked (there were many fewer such computers then than now). Fingerd is the server for the **finger** command, used to issue queries about users of a system. It was relatively recently renamed to **elbow** on Brown CS systems because the original name is offensive to many.

Defense

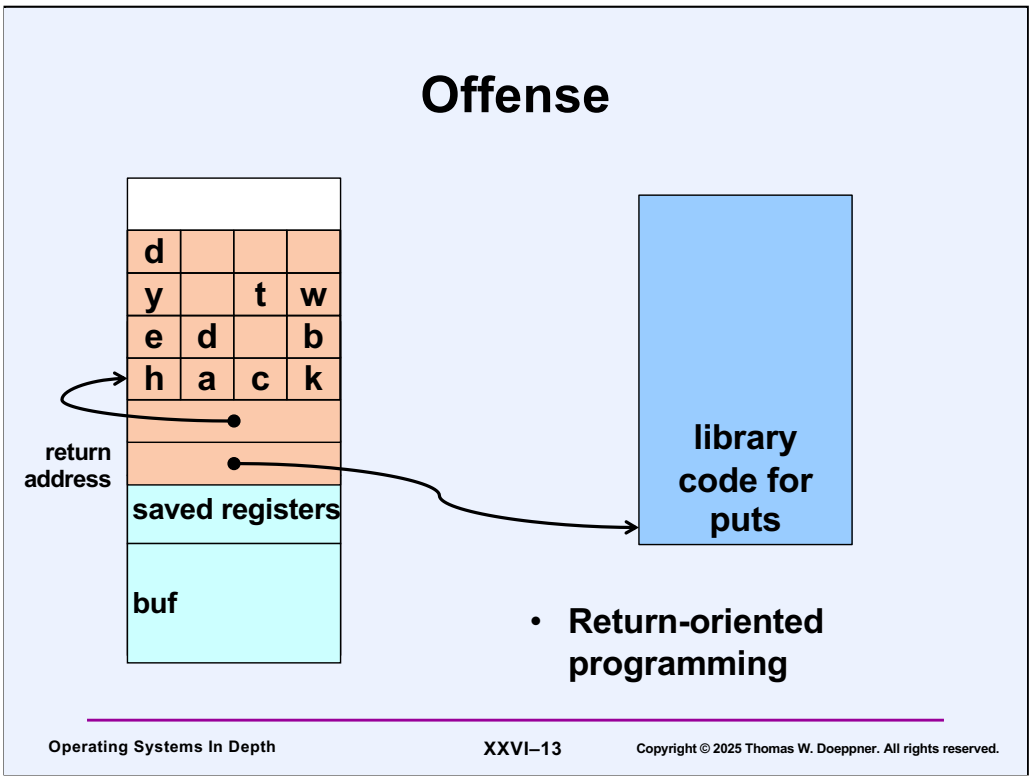
```
void proc( ) {
    char buf[80];
    ...
    fgets(buf, 80, stdin);
    ...
}
```

Passes buffer size: overflow no longer possible!



Better Defense

- Why should the stack contain executable code?
  - no reason whatsoever
- So, don't allow it
  - mark stack *non-executable*
    - (how come no one thought of this earlier?)
    - Intel didn't support it until ~2000
    - x86-64: NX bit in page tables
- Data execution prevention (DEP)
  - adopted by Windows and Linux in 2004
  - by Apple in 2006

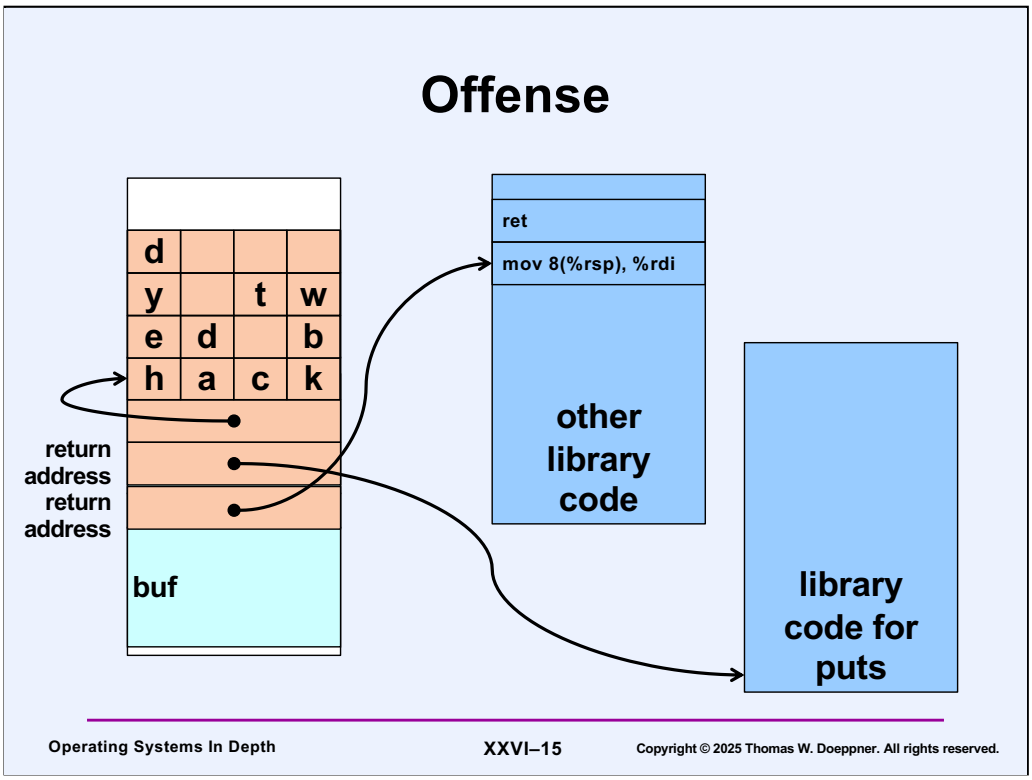


Since we can't have executable code on the stack, rather than call puts, we modify the return address so that the program "returns" to the beginning of the function puts.

This particular form of return-oriented programming is known as "return to libc", since the code we're using is in the C library.

Defense

- Example assumes parameters passed on stack
  - 32-bit x86 convention
- Switch to x86-64
  - parameters passed in registers
  - example breaks
- Offense foiled?

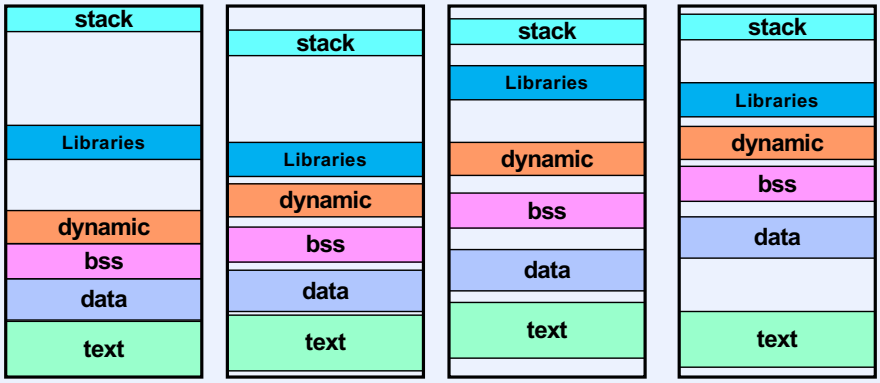


We need to move the argument into %rdi (so puts will find it where it expects it to be). So we examine all of libc (which contains a lot of code) and find the appropriate **mov** instruction followed by a **ret** instruction (recall that, in our memory diagrams, higher addresses are at the top; thus first the **mov** is executed, then the **ret** is executed).

See "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," by Hovav Shacham, http://cseweb.ucsd.edu/~hovav/papers/s07.html.

Defense

- Address space layout randomization (ASLR)
  - start sections at unpredictable locations



Offense

- One possibility
  - guess the start address
    - perhaps $1/2^{16}$ chance of getting it right on x86-32
    - repeat attack a 100,000 times
      - won't be noticed on busy web server
      - very likely it will (eventually) work

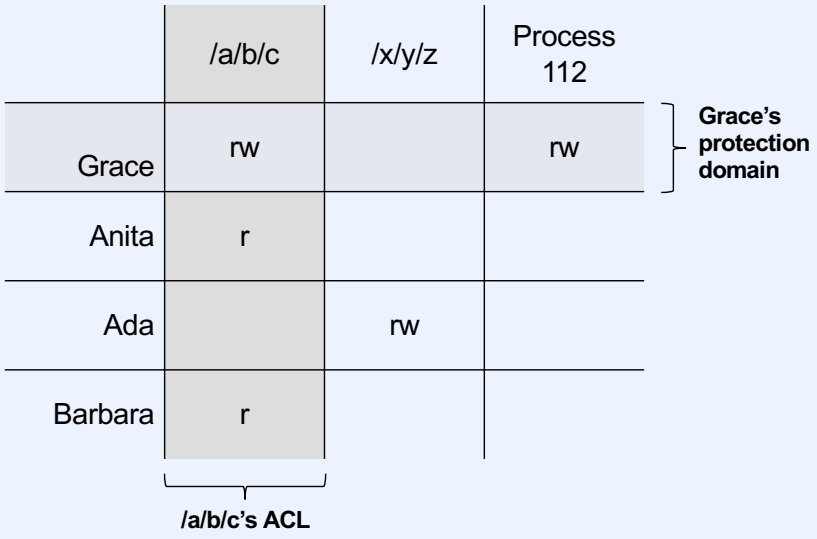
See "Launching Return-Oriented Programming Attacks against Randomized Relocatable Executables," by Liu, Han, Gao, Jing, and Zha, flyer.sis.smu.edu.sg/trustcom11.pdf. The paper was published in 2011 using 32-bit x86 on Linux. The results might well be different in a 64-bit address space (where there's much more room to hide things).

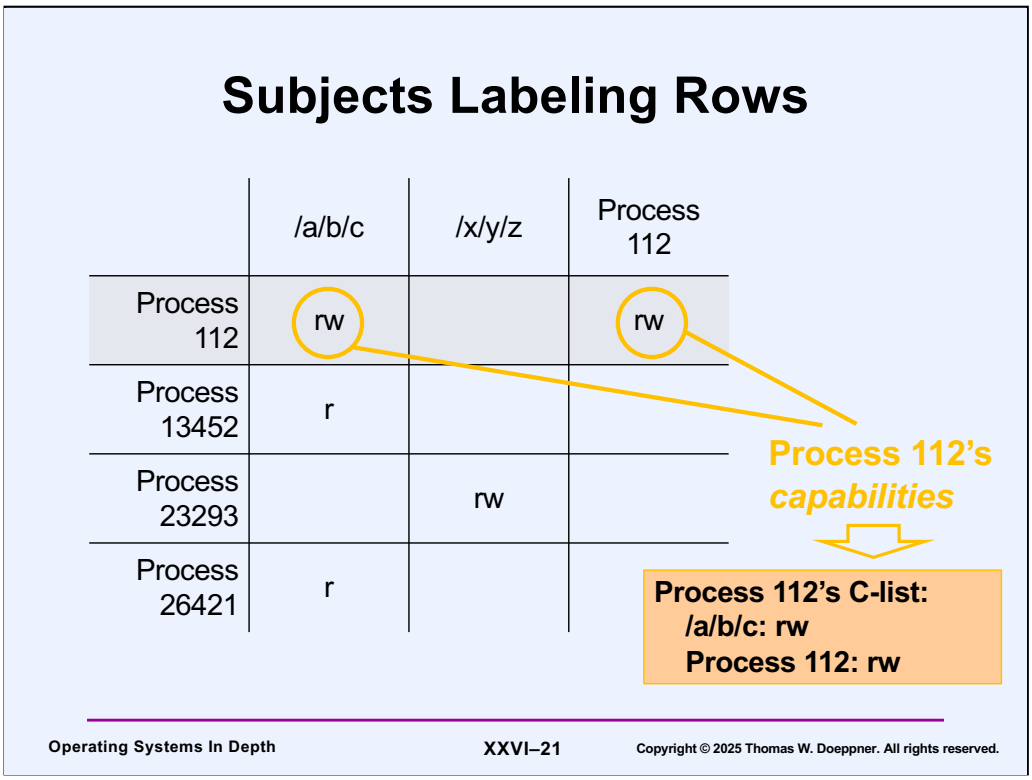
Access Control

Authorization

- Protecting *what* from *whom*
  - protecting *objects* from *subjects*
    - subjects
      - users
      - processes
      - threads
    - objects
      - files
      - web sites
      - processes
      - threads

Access Matrix





C-list = *capability* list. We speak of process 112 having a read-write capability for file /a/b/c.

Principle of least privilege

make the protection domain as small as possible

the capability list (C-List) contains only what's absolutely necessary

Caveat

- Colloquial 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 possible protection domain
    - but …
      - Windows: many users run with "administrator" privilege
      - Unix and Windows: no smaller protection domain than that of a user for resource access (e.g., to files)
- Better use of hardware protection
  - data, such as stacks, are not 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

next

revisit later

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

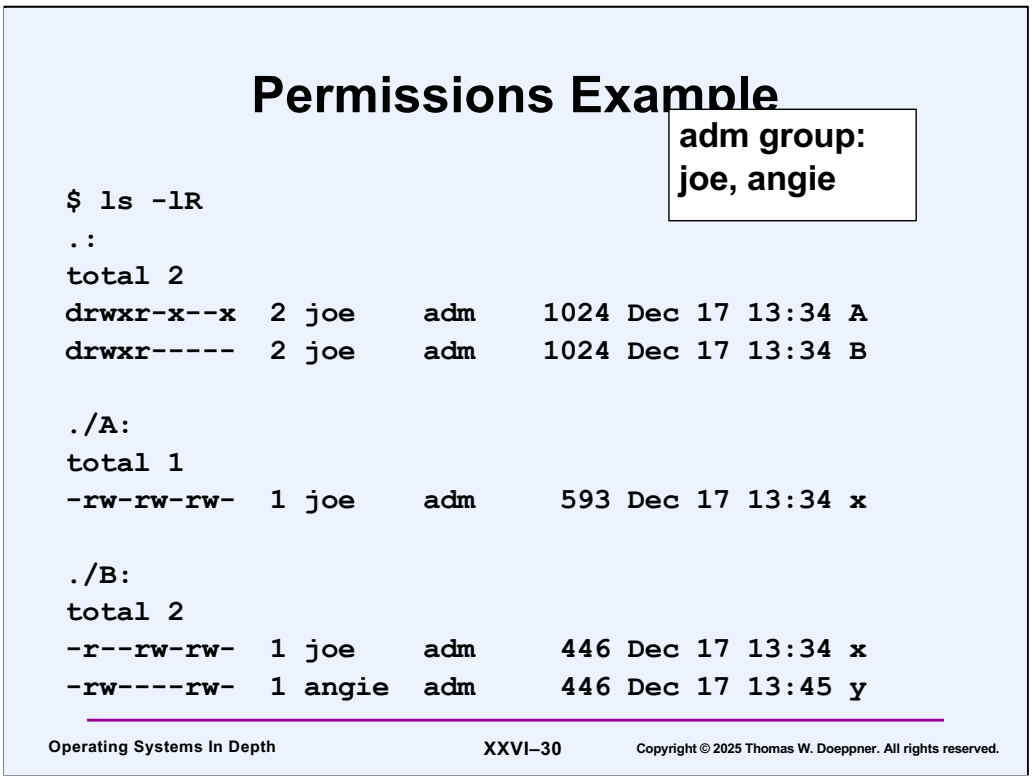
- Unix and Windows
  - primarily DAC
  - file descriptors and file handles provide capabilities
  - MAC becoming more popular
    - SELinux
    - Windows

will discuss in later lecture

Case Study 1: Unix Permissions

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



The `ls -lR` command lists the contents of the current directory, its subdirectories, their subdirectories, etc. in long format (the `l` causes the latter, the `R` the former).

In the current directory are two subdirectories, **A** and **B**, with access permissions as shown in the slide. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group's members, and the last three are the permissions for the rest of the world.

Quiz: the users **joe** and **angie** are members of the **adm** group; **leo** is not.

May **leo** list the contents of directory *A*?

May **leo** read *A/x*?

May **angie** list the contents of directory *B*?

May **angie** modify *B/y*?

May **joe** modify *B/x*?

May **joe** read *B/y*?

Quiz 1

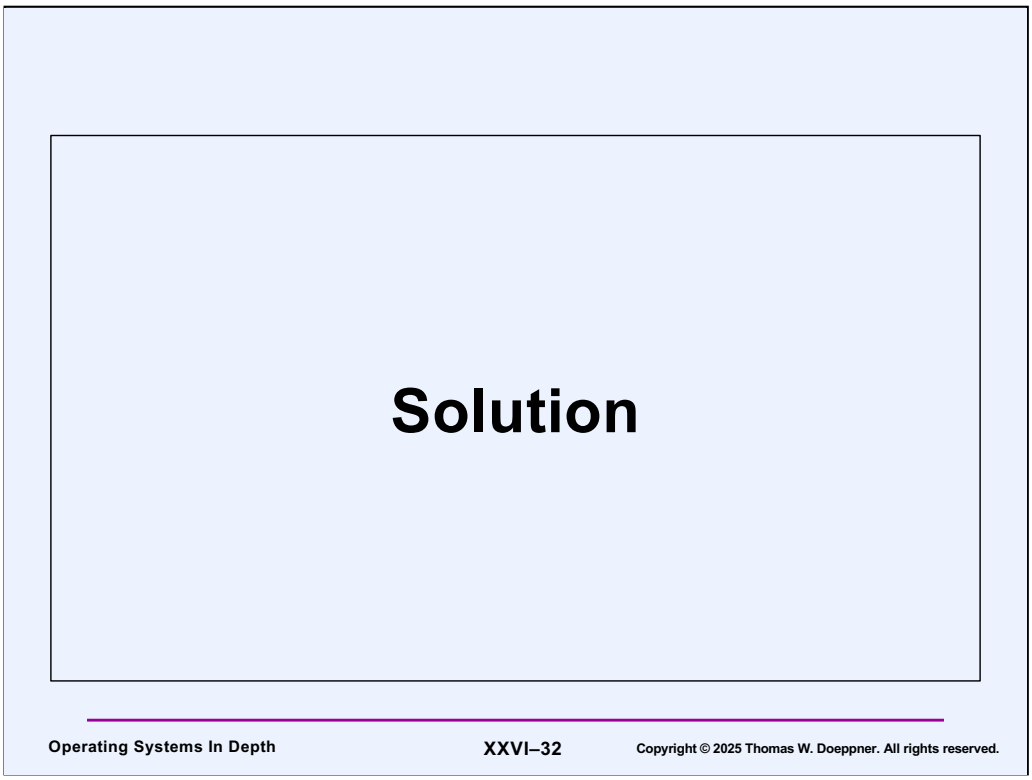
Recall that in Unix, each file/directory has an owner and a group (e.g., owner “joe”, group “adm”).

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 can be done in a single command just like setting the file readable by just one group.



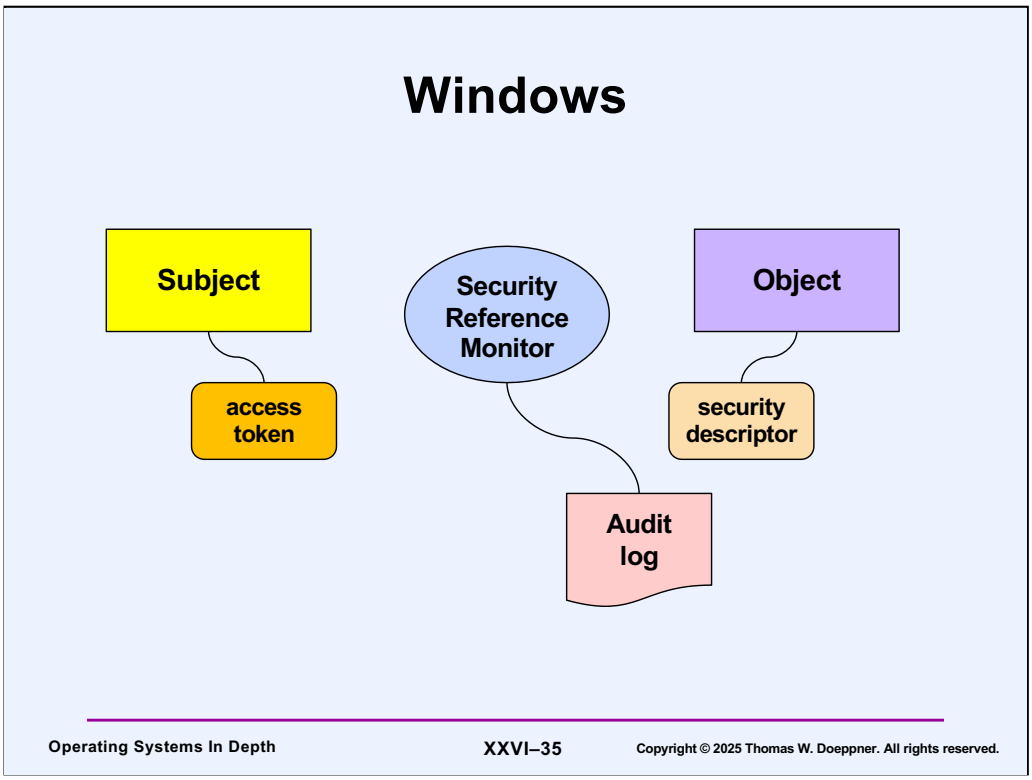
The “none” group is a group with no members.

Initializing Authorization Info

- permission_vector = mode & ~umask
  - mode is from `open`/`creat` system call
  - umask is process-wide, set via `umask` syscall
- 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)

We explain "effective user ID" in the next lecture. For now, simply think of it as "user ID".

Case Study 2: Windows Security Architecture



A subject (a process) is attempting to access an object (perhaps a file). The **access token** represents the identity of the subject. The **security descriptor** describes who may access the object in what ways. The reference monitor, using the **access token** and **security descriptor** as input, determines whether the desired access should be permitted. These decisions may be recorded in an **audit log**.

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.
- E.g., S-1-5-123423890-907809-43

Security **principals** are the users and groups of users for whom access is to be mediated. They're uniquely identified by **Security Identifiers**, as shown in the slide.

Security Descriptor

- Owner's SID
- DACL
  - discretionary access-control list
- SACL
  - system access-control list
    - controls auditing
- Flags

A security descriptor is provided for each object. It indicates who owns the object, who is allowed to access it, and what sorts of accesses should be audited.

DACLs

- Sequence of Access-Control Entries (ACEs)
- Each indicates
  - who it applies to
    - SID of user, group, etc.
  - what sort of access
    - bit vector
  - action
    - permit or deny

A DACL (discretionary access control list) describe which principals may access an object and how they may access it.

See http://msdn.microsoft.com/en-us/library/aa374876(VS.85).aspx for more details about ACLs.

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.

ACE = access control entry.

An ACL is an ordered list of ACEs.

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 …

allow
inGroup
read, write
deny
Mary
read, write

deny
Mary
read, write
allow
inGroup
read, write

Mary is a member of inGroup. Thus she is permitted read/write access in the leftmost ACL, but is denied read/write access in the rightmost ACL.

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

When the ACL of a directory is modified, its inheritable ACEs are propagated to its descendants.

Case Study 3:
POSIX Advanced ACLs

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**

This is slightly simplified. See the acl man page for details.

Example

```
% mkdir dir
% ls -ld dir
drwxr-x--- 2 twd fac 8192 Mar 30 12:11 dir
% setfacl -m u:floria: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:floria:rwx
group::r-x
mask::rwx
other::---
```


Here we create a directory, then add an ACL giving floria read, write, and execute permission. (Note that the "-m" flag of setfacl stands for "modify" -- we're adding a user entry.) The **ls** command, via the "+", indicates that the permissions on the directory are more complicated than it can show. However, the **getfacl** command shows the complete permissions. Note that a mask is automatically created allowing floria to have the full permissions requested. The user and group entries that don't explicitly refer to a particular user or group give the access permissions of the user_obj (file owner) and group_obj.

Note that this example assumes that all files (and directories) are on local file systems, not on remote file systems. Thus it cannot be replicated in, for example, your home directory on CS department machines, since home directories are on remote file systems (and ACLs aren't necessarily implemented on remote file systems).

Example (continued)

```
% setfacl -dm u::rwx,g::rx,u:floria:rwx dir
% getfacl dir
# file: dir
# owner: twd
# group: cs-fac
user::rwx
user:floria:rwx
group::r-x
mask::rwx
other::---
default:user::rwx
default:user:floria:rwx
default:group::r-x
default:mask::rwx
default:other::---
```


Now we give the directory a default ACL, which is used to initialize the ACLs for files and directories created within the directory.

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:floria:rwx                 #effective:rw-
group::r-x                      #effective:r--
mask::rw-
other::---
```


Here we create a file within the directory. The cp command supplies a **mode** argument (requested permissions) of 0666 to the **creat** system call – this is because it's copying a file whose permissions are 0666. (The umask is 007). The **mask ACE** was set automatically to rw (because of the default entry we put in the directory in the previous slide and **mode** having been specified as 0666), thus ensuring that floria's effective permissions are rw.

Example (continued)

```
% 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:floria:rwx                 #effective:rw-
group::r-x                      #effective:r--
mask::rw-
other::---
```


We now create the file again within the directory, but through the use of a program that we wrote, called **new**, with which we can control the mode bits in the **creat** system call. In this example, the file is created with requested permissions of 0466 (read-only for the user, read and write for the group and others), but adjusted by the umask. Strangely, both the owner of the file and floria are given rw permissions. (This is, again, because of the default ACL entry we put in the directory.)

Example (and still continued)

```
% setfacl –m o:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx
group::r-x
mask::rwx
other::rw-
```


We further modify the ACL by setting the permissions for others to be rw. This appears to have the side effect of changing the mask to rwx, which gives both floria and the file's group execute access to the file. It's not at all clear why this happens and is likely a bug.

Example (and still continued)

```
% setfacl –m g:cs1670ta:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx
group::r-x
group:cs1670ta:rw-
mask::rwx
other::rw-
```


We further modify the ACL by adding a group entry for cs1670ta and setting it to rw.

Example (end)

```
% setfacl –m m:r file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx                 #effective:r--
group::r-x                      #effective:r--
group:cs1670ta:rw-              #effective:r--
mask::r--
other::rw-
```


Finally, we change the mask to read-only. This affects the explicit user entry and the group entries, but does not affect the owner of the file or others.