Security Part 5

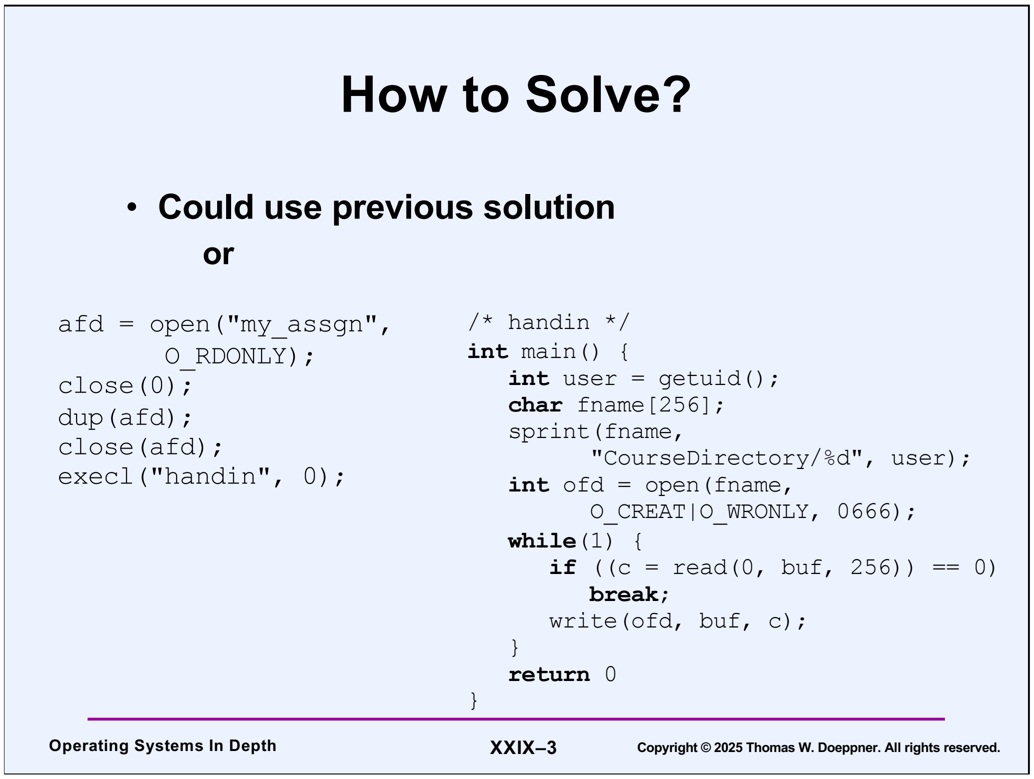
Live Anonymous Q&A:
https://tinyurl.com/cs1670feedback

Recap: TOCTTOU vulnerability

```
/* 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_assgn
…
```

Hidden Code



How to Solve?

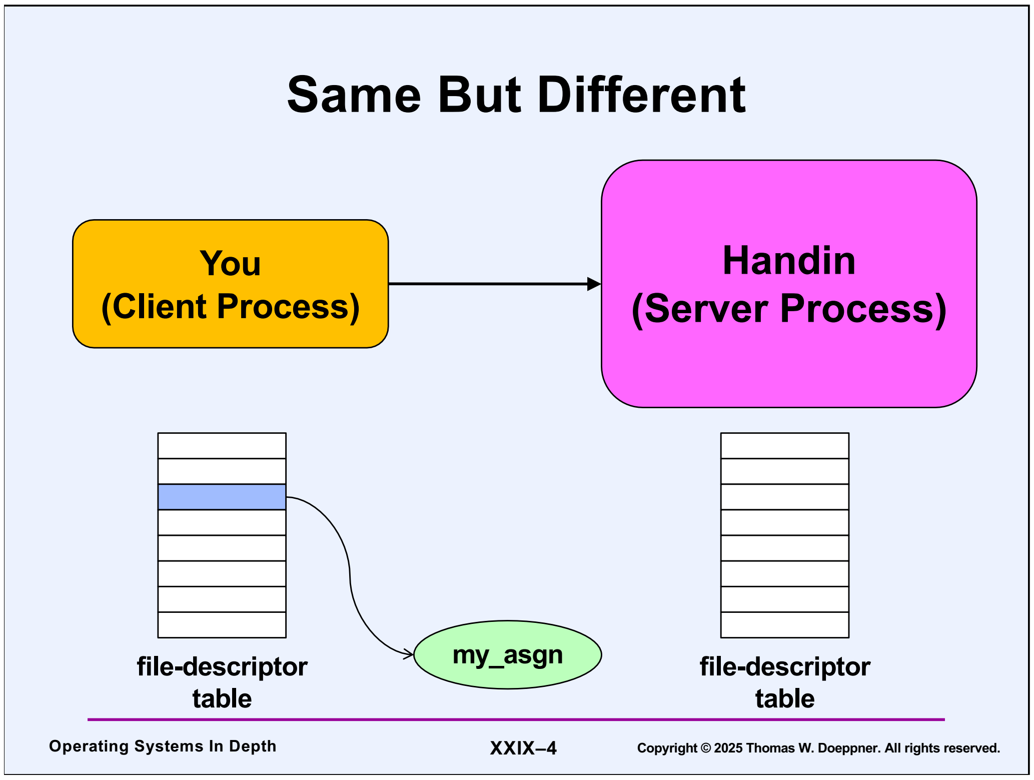
- **Could use previous solution**

  or

```
afd = open("my_assgn",
        O_RDONLY);
close(0);
dup(afd);
close(afd);
execl("handin", 0);
```

```
/* handin */
int main() {
   int user = getuid();
   char fname[256];
   sprint(fname,
         "CourseDirectory/%d", user);
   int ofd = open(fname,
         O_CREAT|O_WRONLY, 0666);
   while(1) {
      if ((c = read(0, buf, 256)) == 0)
         break;
      write(ofd, buf, c);
   }
   return 0
}
```

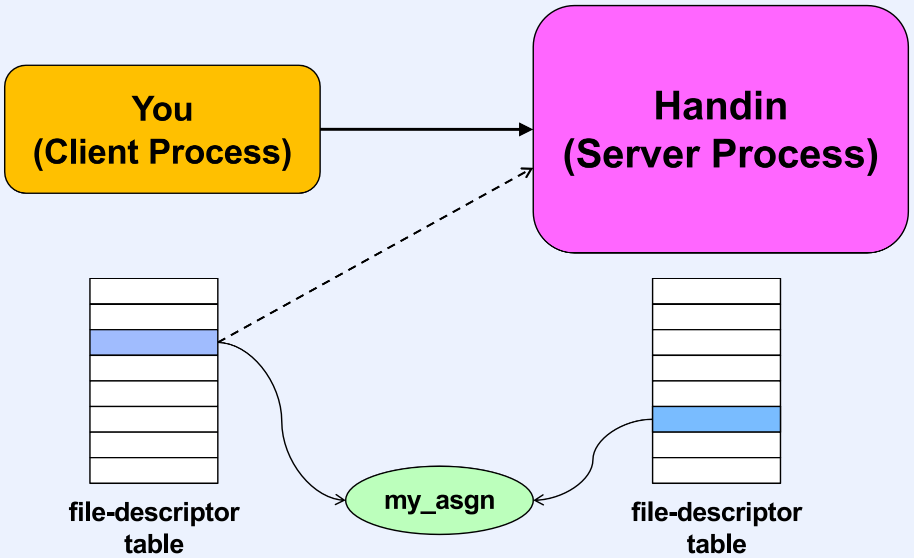
handin is a setuid-twd program. The calling code (on the left) runs in a student's account. Thus, my_assgn is opened using the student's privileges. After the call to **dup**, file descriptor 0 will refer to the file, and **handin** may assume that its invoker has access rights to it (since the invoker had to open it).



Same But Different

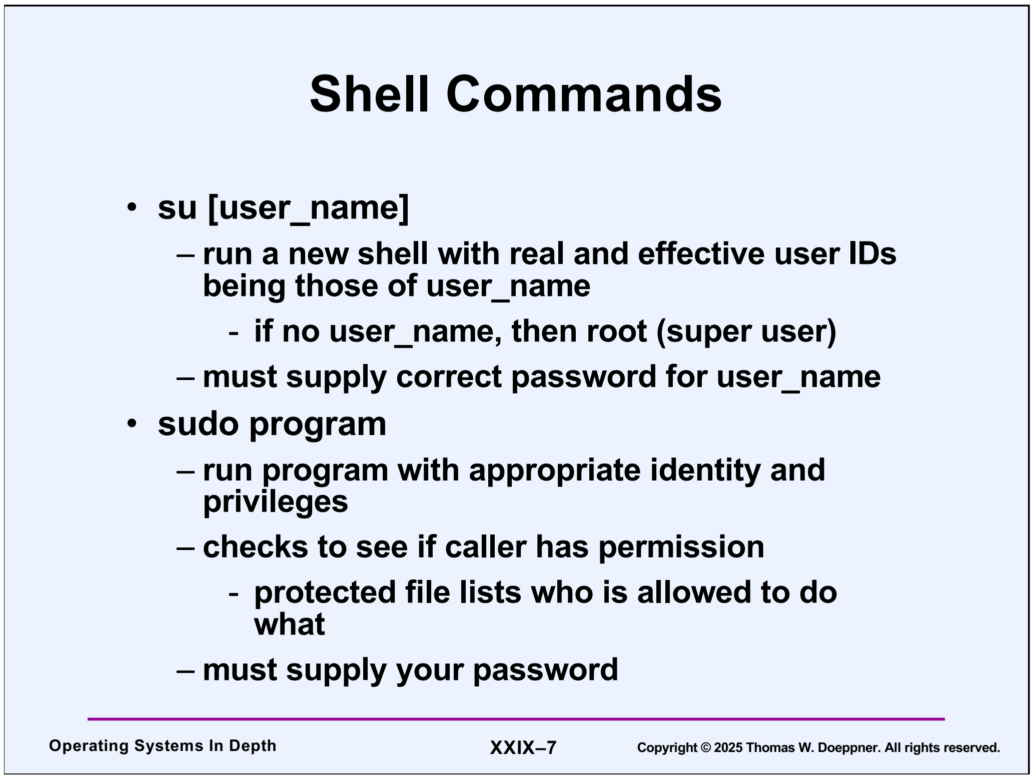
In this example you are to send the name of the file containing your homework to a server process running the handin program. But the handin program, running as twd, might not have suitable permission to access your file.

File Descriptor as *Capability*



Unix provides a means for transmitting file descriptors from one process to another via a Unix-domain socket. If the handin program receives this file descriptor, it can then use it to access your file, despite the fact that it would not be able to open it on its own. Thus a file descriptor acts as a **capability** for an object, a concept we explore in more detail soon.

Changing Security Context



Shell Commands

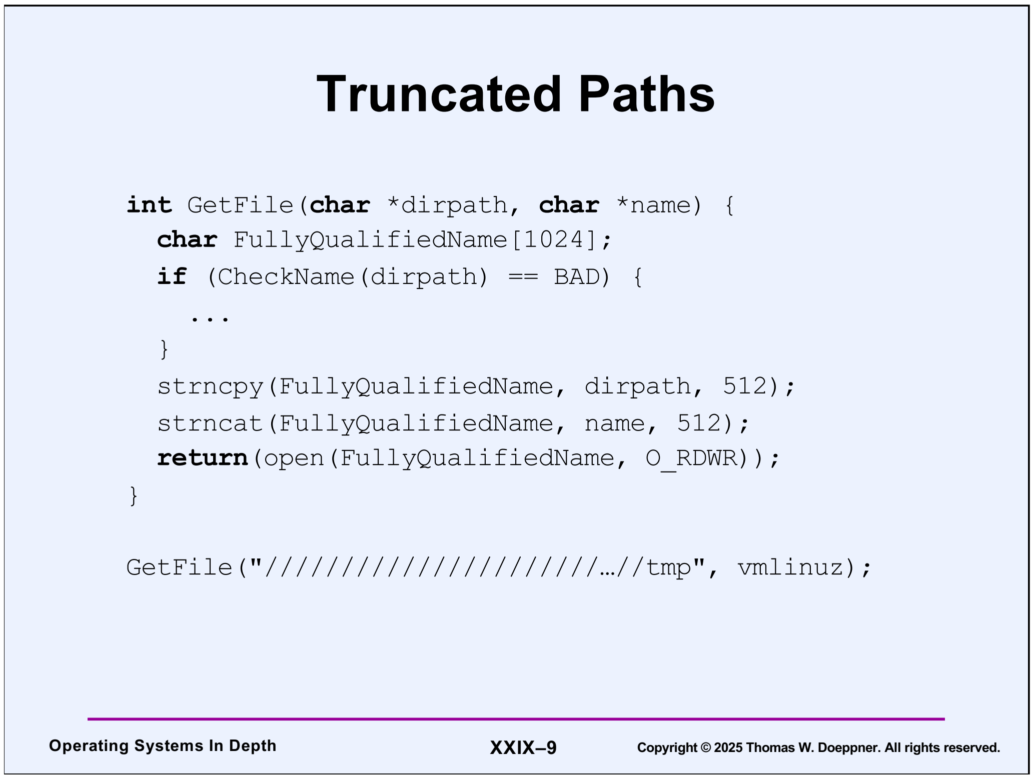
- su [user_name]
  - run a new shell with real and effective user IDs being those of user_name
    - if no user_name, then root (super user)
  - must supply correct password for user_name
- sudo program
  - run program with appropriate identity and privileges
  - checks to see if caller has permission
    - protected file lists who is allowed to do what
  - must supply your password

These shell commands are used for very different purposes. The **su** command might be used, for example, by an administrator, who uses a normal account, but occasionally needs super-user powers. So they su to root only when necessary.

The **sudo** command extends the usefulness of setuid programs by allowing the specification of who is allowed to run the command. Thus such chores don't have to be done in the command itself, which can assume the user is authorized. The name sudo stands for "superuser do", though, in its current form, it's not restricted to running commands as superuser.

Programming Securely

- It’s hard!
- Some examples …



Truncated Paths

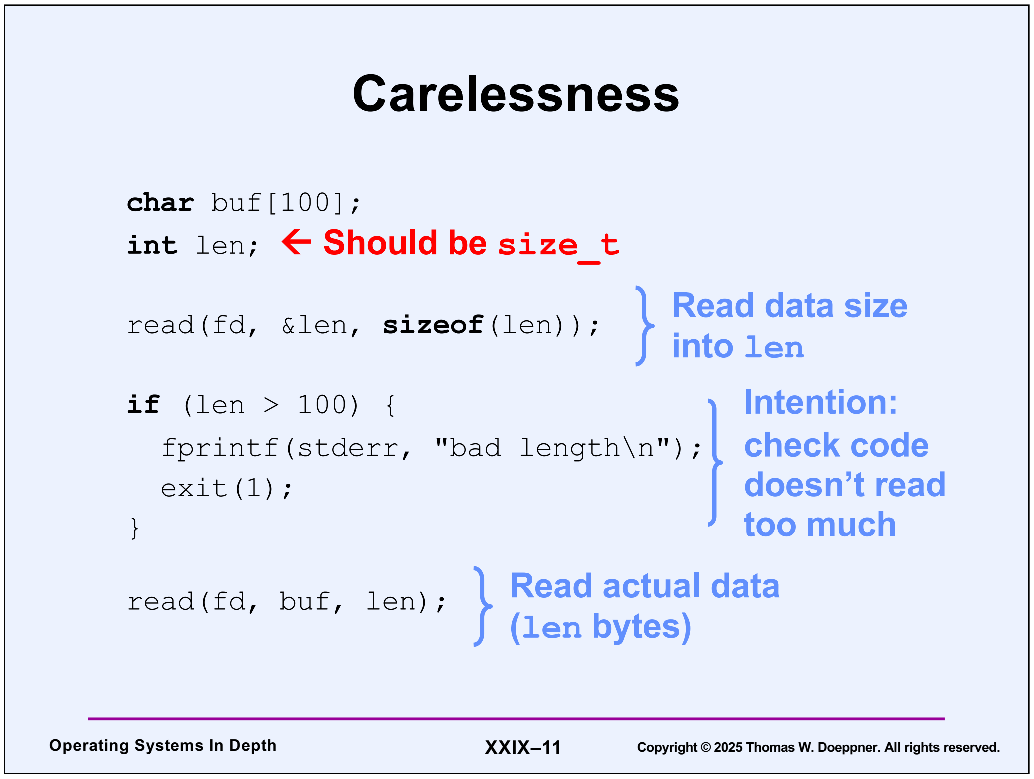
```
int GetFile(char *dirpath, char *name) {
  char FullyQualifiedName[1024];
  if (CheckName(dirpath) == BAD) {
    ...
  }
  strncpy(FullyQualifiedName, dirpath, 512);
  strncat(FullyQualifiedName, name, 512);
  return(open(FullyQualifiedName, O_RDWR));
}

GetFile("//////////////////////…//tmp", vmlinuz);
```

Getfile is a program that takes two arguments. The first is supposed to be a path leading to a directory, and the second is a file in that directory. It checks to make sure the directory is one that it's ok for the caller to be using, then opens the file read-write and returns the file descriptor. Thus the intent is that it opens the file only if it's safe to do so.

But here, **GetFile** is called with a first argument that consists of 512 slashes followed by "tmp". However, **strncpy** is called with the assumption that the directory pathname is no more than 512 characters long. However, since Unix pathname semantics are such that a sequence of one or more slashes are equivalent to a single slash, the file that's opened is /vmlinuz (the executable that's executed when the system is booted). However, the access check that's done in CheckName looks at the actual directory that's passed in, whose name is equivalent to /tmp — items in this directory might well be legitimately accessible by the caller.

Defense

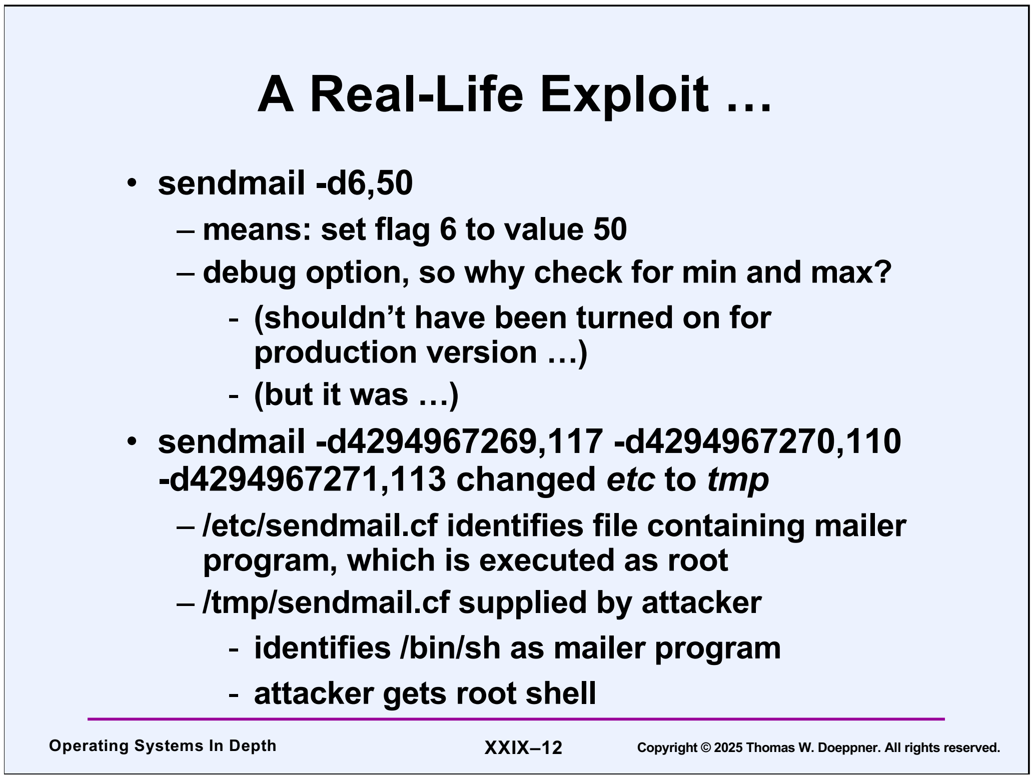
- It’s not enough to avoid buffer overflow …
- Check for truncation!



Carelessness

```
char buf[100];
int len;  ← Should be size_t

read(fd, &len, sizeof(len));        } Read data size into len

if (len > 100) {
  fprintf(stderr, "bad length\n");  } Intention: check code doesn't read too much
  exit(1);
}

read(fd, buf, len);                 } Read actual data (len bytes)
```

Though this code seems to check the length correctly, there's a problem if a negative value is passed as len. Since the third parameter to read is a **size_t**, which is unsigned, this negative value will be treated as a large positive value.



A Real-Life Exploit …

- sendmail -d6,50
  - means: set flag 6 to value 50
  - debug option, so why check for min and max?
    - (shouldn't have been turned on for production version …)
    - (but it was …)
- sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113 changed *etc* to *tmp*
  - /etc/sendmail.cf identifies file containing mailer program, which is executed as root
  - /tmp/sendmail.cf supplied by attacker
    - identifies /bin/sh as mailer program
    - attacker gets root shell

This example is adapted from http://www.dwheeler.com/secure-programs/secure-programming-handouts.pdf. It was used in the infamous "internet worm" attack in 1988 by Robert Morris and is perhaps the first example of a widespread successful exploit.

What You Don’t Know …

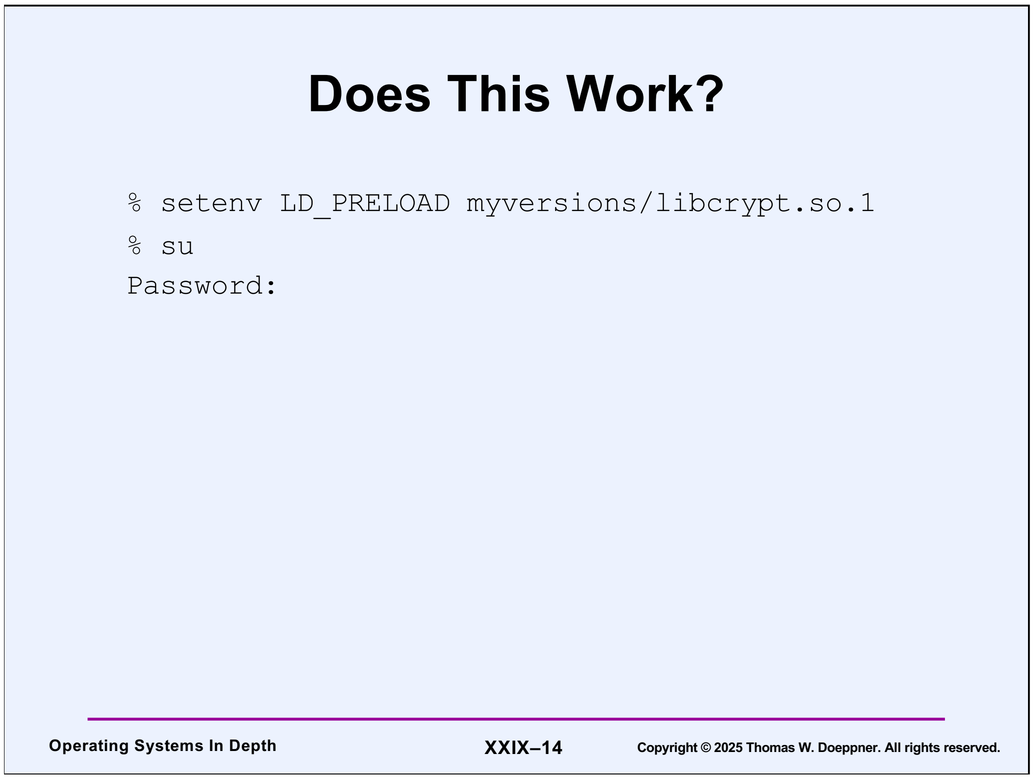
```
int TrustedServer(int argc, char *argv[]) {
  ...
  printf(argv[1]);
  ...
}
```

```
% TrustedServer "wxyz%n"
```

from the printf man page:

%n	The number of characters written so far is stored into the integer indicated by the int * (or variant) pointer argument. No argument is converted.

→ `printf` changes arbitrary (?) memory location



Does This Work?

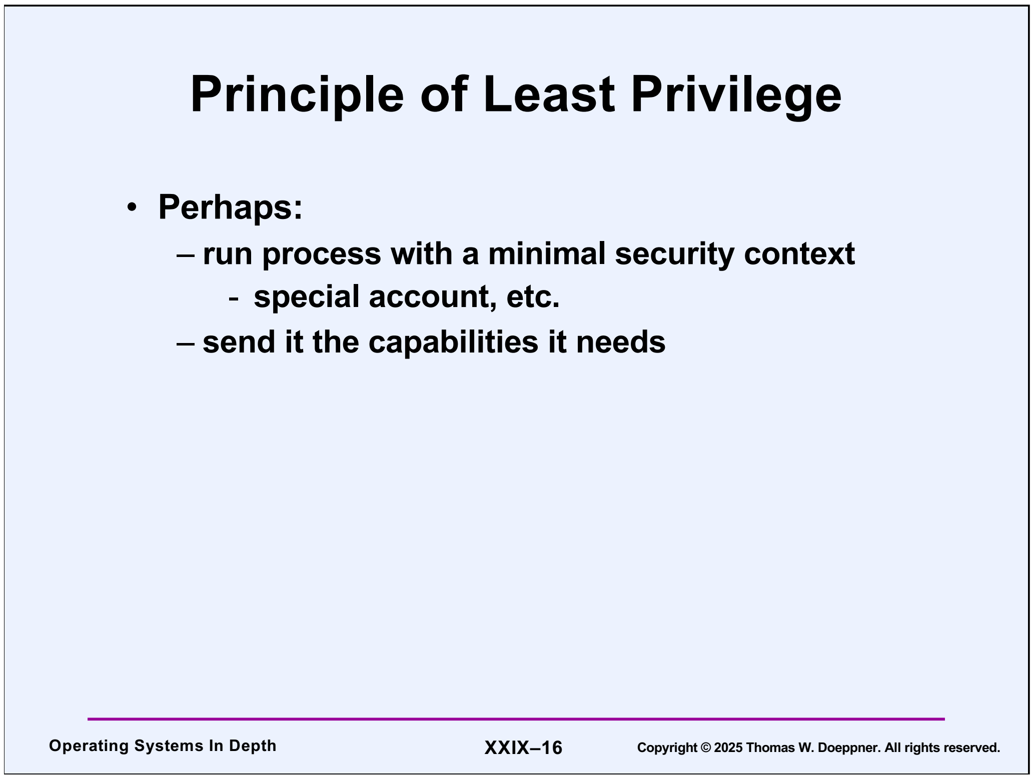
```
% setenv LD_PRELOAD myversions/libcrypt.so.1
% su
Password:
```

Recall that the environment variable LD_PRELOAD is used by the runtime loader (ld-linux.so) to indicate which libraries should be used to supply functions referenced by a program. Here we have put together our own version of libcrypt, which contains the functions necessary to determine if a supplied password is correct. However, since it's our version and not the system's version, we can arrange so that it allows any password to gain access to superuser privileges.

So, could we use the code shown in the slide to get superuser privileges?

Fortunately, at least on Linux, this doesn't work. su is a setuid-to-root program. ld-linux.so, when invoked by such a program, does not allow arbitrary settings for LD_PRELOAD, but allows only libraries that are in the standard system directories.

Isolating Security Contexts



Principle of Least Privilege

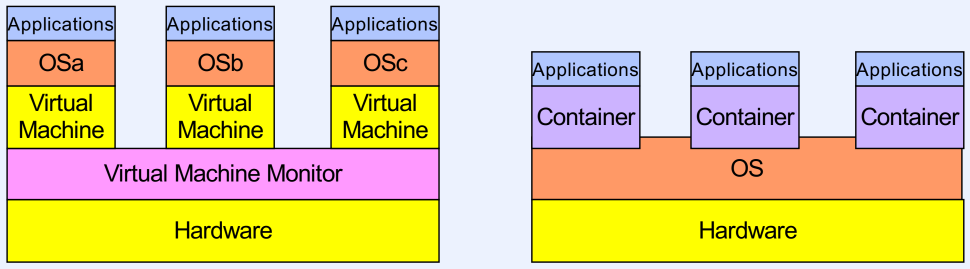
- Perhaps:
  - run process with a minimal security context
    - special account, etc.
  - send it the capabilities it needs

This is suggestive, but not really practical.

Complete Isolation

- Would like to run multiple applications in complete isolation from one another
  - run them on separate computers with no common file system
  - run them on separate virtual machines
  - run them in separate *containers* on one OS instance

VMs versus Containers

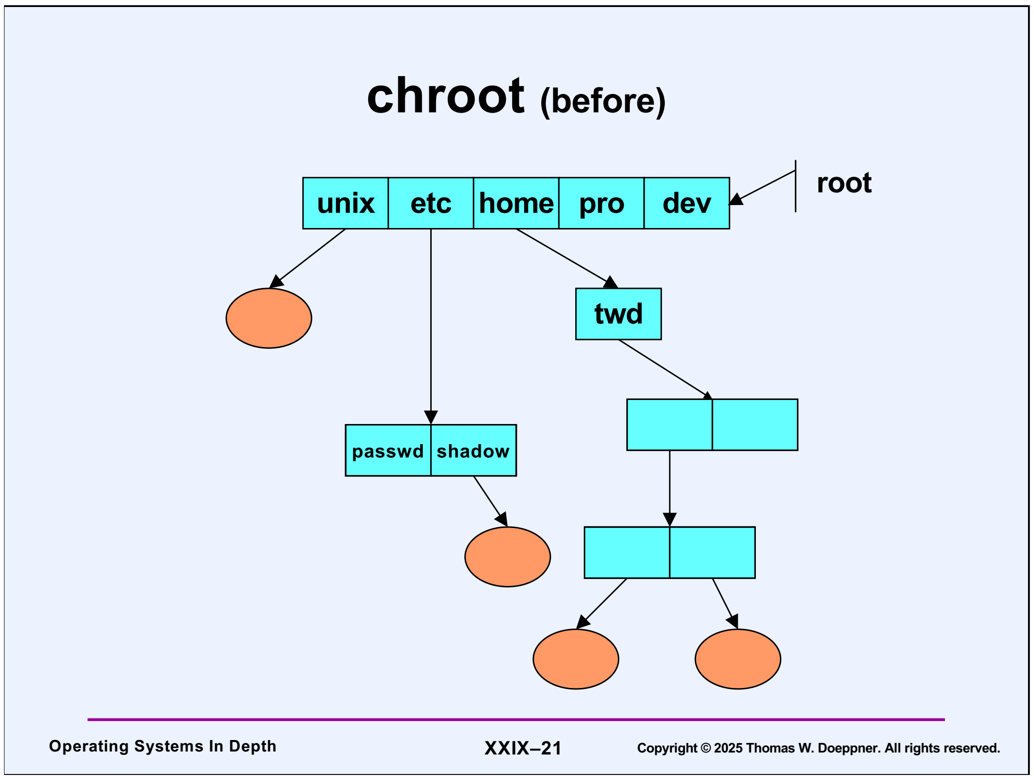


Containers

- Isolated
  - processes in a container can’t access what’s not in the container
  - processes in a container shouldn’t even be aware of what’s not in the container

Container Building Blocks

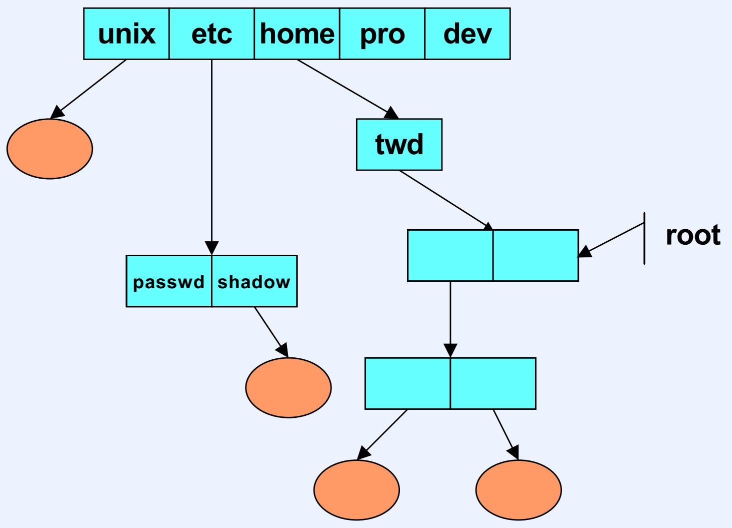
Part 1: File system (chroot)



chroot (before)

The **chroot** system call allows a process to change the root of its file-system hierarchy to a directory lower down in the tree.

chroot (after)



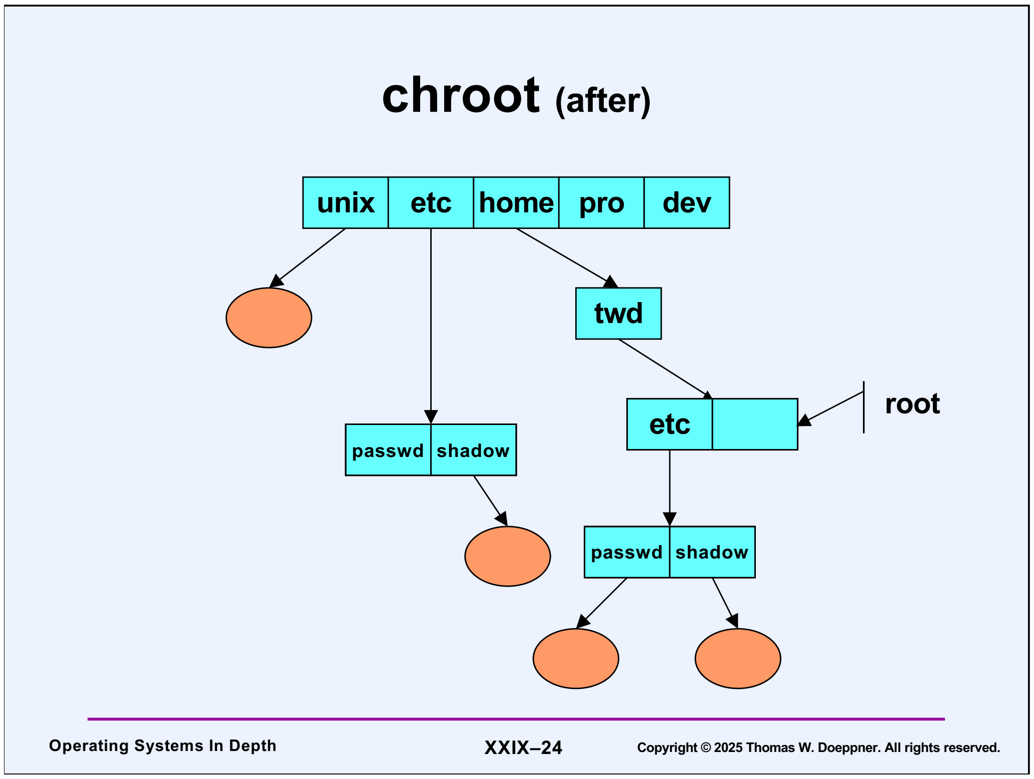
Not a Quiz

Restricting a process to a particular subtree

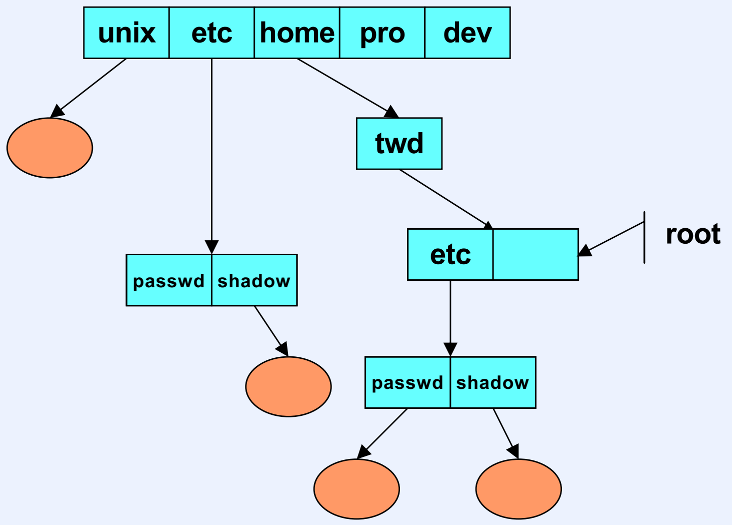
a) improves security by effectively running the process in a smaller protection domain

b) has little effect on security

c) potentially makes security worse



chroot (after)



One must be careful with its use — it could be used to trick a setuid program into treating files you provide as if they were standard system files. For example, the su program, which reads from the /etc/passwd and /etc/shadow files to verify a password, might be convinced that you know everyone's passwords.

Relevant System Calls

- `chroot(path_name)`

- `chdir(path_name)`
- `fchdir(file_descriptor)`

Not a Quiz

After executing *chroot*, “/” refers to the process’s new root directory. Thus “..” is the same as “.” at the process’s root, and the process cannot cd directly to the “parent” of its root. Also, recall that hard links may not refer to directories.

a) *chroot* does effectively limit a process to a subtree

b) *chroot* does not effectively limit a process to a subtree

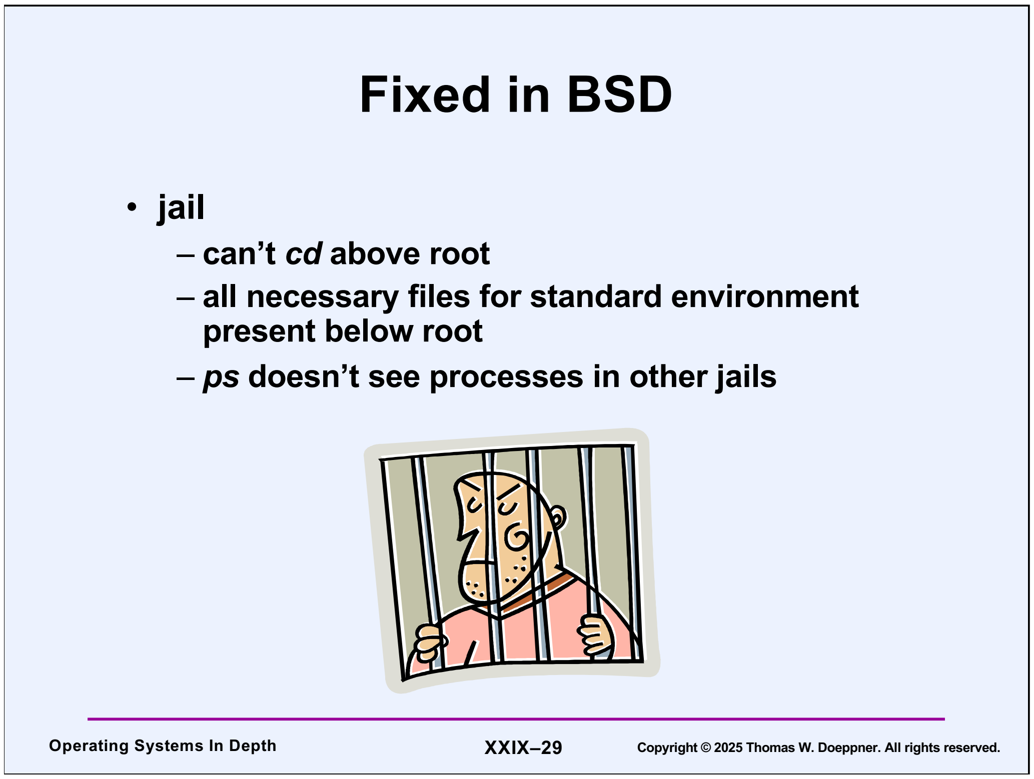
Escape!

```
chdir("/");
pfd = open(".", O_RDONLY);
mkdir("Houdini", 0700);
chroot("Houdini");
fchdir(pfd);
for (i=0; i<100; i++)
  chdir("..");
chroot(".");
```

The *fchdir* system call changes the current directory to be the directory referred to by the file descriptor passed as an argument.

Namespace Isolation

- Isolate process by restricting it to a subtree
  - chroot isn't foolproof
- Fix chroot
  - make it superuser only
  - make sure processes don't have file descriptors referring to directories above their roots



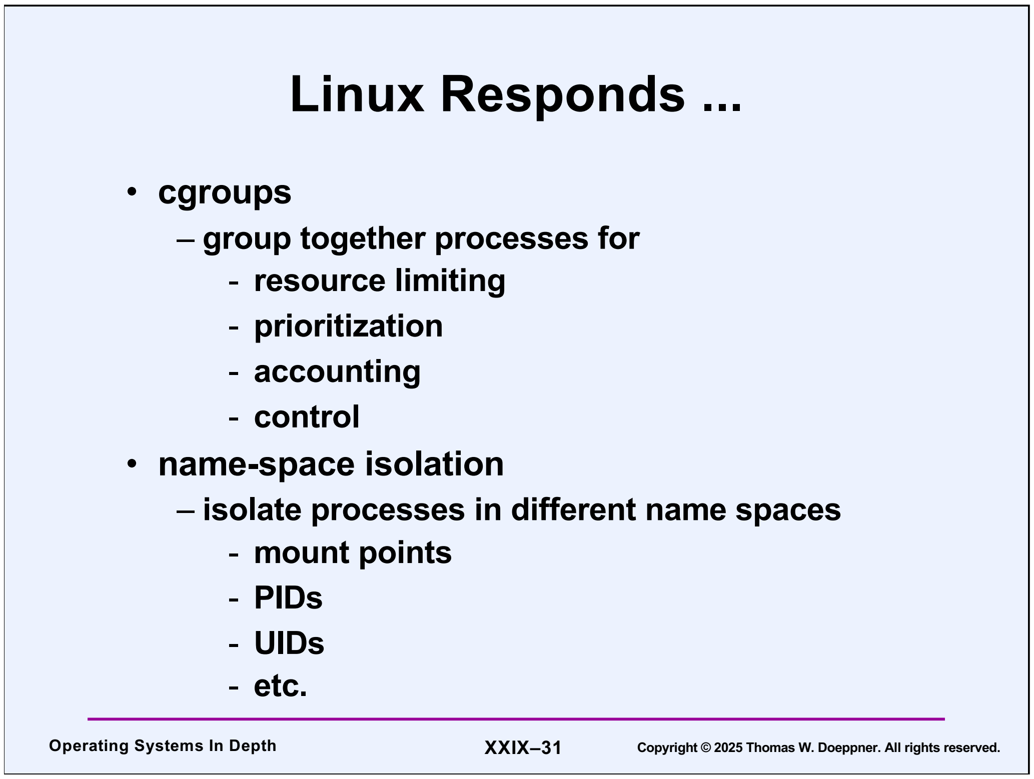
Fixed in BSD

- jail
  - can't *cd* above root
  - all necessary files for standard environment present below root
  - *ps* doesn't see processes in other jails

Solaris has a similar, but more powerful feature called *zones*.

Container Building Blocks

Part 2: Resources & Namespaces



Linux Responds ...

- cgroups
  - group together processes for
    - resource limiting
    - prioritization
    - accounting
    - control
- name-space isolation
  - isolate processes in different name spaces
    - mount points
    - PIDs
    - UIDs
    - etc.

This is from https://en.wikipedia.org/wiki/Cgroups.

Linux Containers

- Reside in isolated subtrees
  - (fixed) chroot restricts processes in a container to the subtree
  - file systems are mounted in container namespaces, so that other containers can't see them
- Separate UID and PID spaces
  - PIDs start at 1 for each container
  - container UIDs mapped to OS UIDs
    - UID 0 has privileges in container, but not outside of container
- Limits placed on CPU, I/O and other usages

Docker

- Runs in Linux containers (also runs on Windows)
  - container contains all software and files needed for execution
  - provides standard API for applications
    - even if on Windows
- On macOS, actually runs a hypervisor and runs containers inside (Linux) VM

Windows Security

Back to Windows

- Security history
  - DOS and early Windows
    - no concept of logging in
    - no authorization
    - all programs could do everything
  - later Windows
    - good authentication
    - good authorization with ACLs
    - default ACLs are important
      - few understand how ACLs work …
    - many users ran with admin privileges
      - all programs can do everything …

Privileges in Windows

- Properties of accounts
  - administrator ≈ superuser
  - finer breakdown for service applications
- User account control (starting with Vista)
  - accounts with administrator privileges have two access tokens
    - one for normal usage
    - another with elevated rights

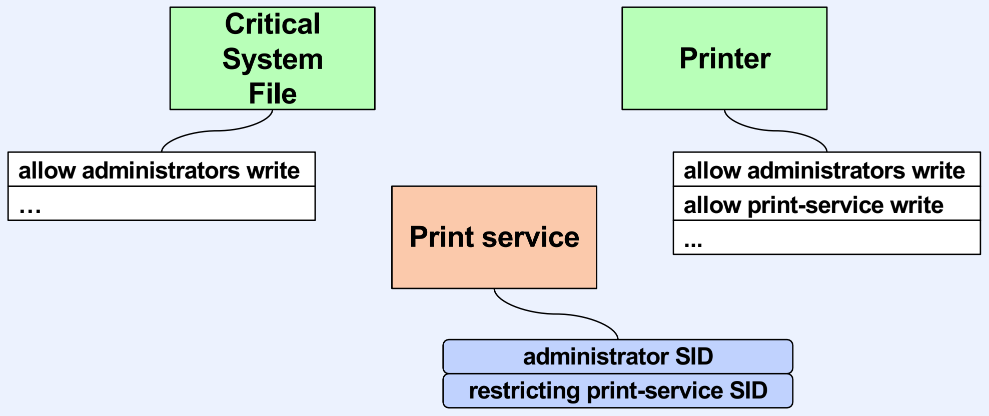
Least Privilege

- Easy answer
  - disable privileges
  - works only if the process has any …
- Another answer
  - restricting SIDs
    - limit what a server can do
    - two passes over ACL for access check
      - first: as previously specified
      - second: using only restricting SIDs

Least Privilege for Servers

- Pre-Vista:
  - services ran in local system account
    - all possible privileges
    - successful attackers “owned” system
    - too complicated to give special account to each service
- Vista and beyond
  - services still run in system account
  - per-service SIDs created
    - used in DACLs to indicate just what service needs
    - marked *restricting* in service token

Example


Critical System File
allow administrators write
…
Printer
allow administrators write
allow print-service write
...
Print service
administrator SID
restricting print-service SID

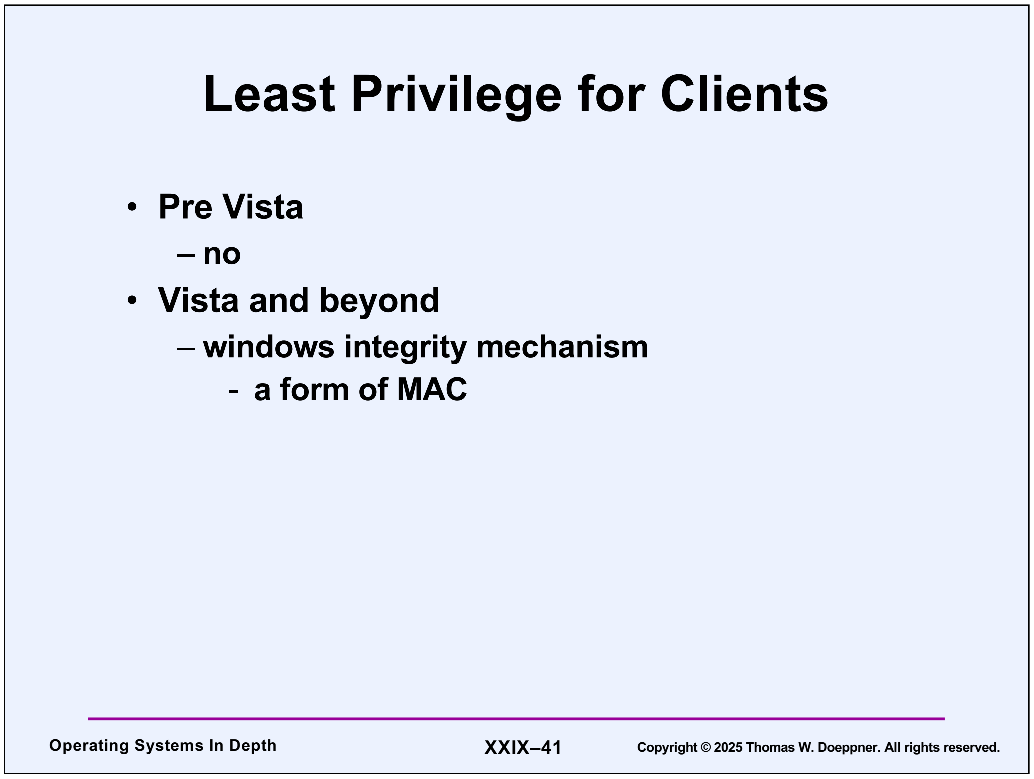
Here the print service runs as the system administrator and hence would normally be able to access all files. In particular, it has write access to some critical system file, as well as to the printer. By adding a print-service SID to its access token as a restricting SID and adding an ACL entry to the printer (but not to the critical system file), we allow the print service to access the printer, but not any files that lack ACL entries allowing the print service access.

The print service must be granted access both as an administrator and as the print service.

Not a Quiz

- Why are there two passes made over the ACL?

- Answer: a restricting SID is not an additional access right, but it diminishes what can be done with existing rights
  - one must first show that one has an access right, then check if it has been diminished



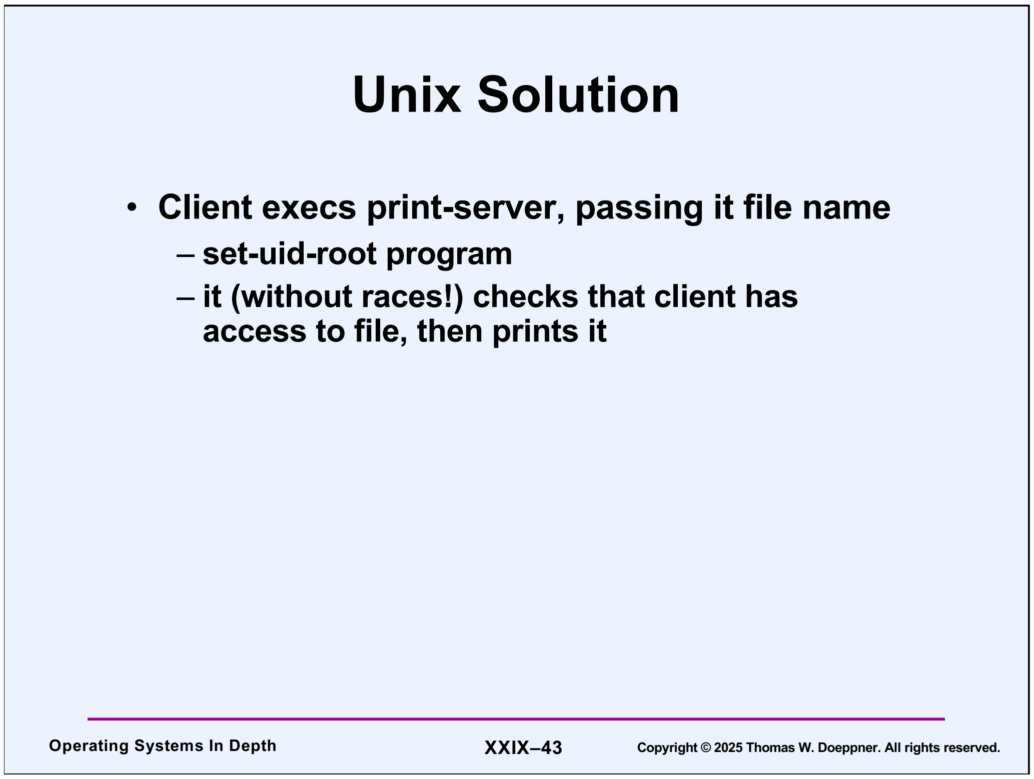
Least Privilege for Clients

- Pre Vista
  - no
- Vista and beyond
  - windows integrity mechanism
    - a form of MAC

We cover this later in this lecture, when we discuss MAC.

Print Server

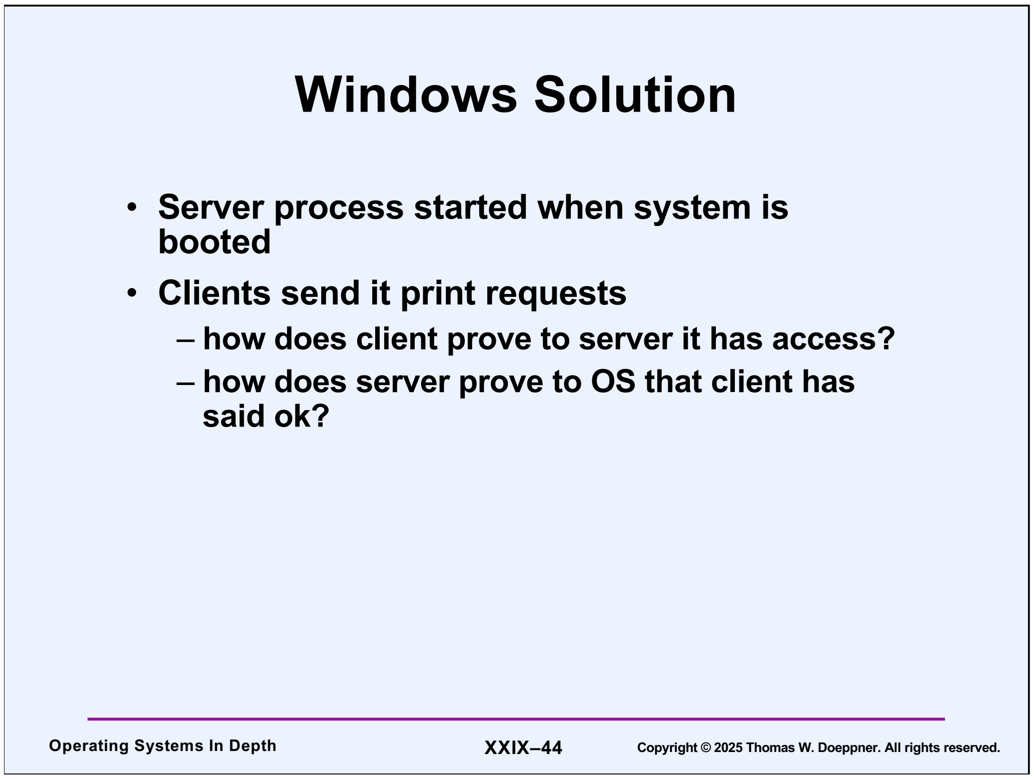
- Client sends request to server
  - print contents of file X
- Server acts on request
  - does client have read permission?
    - server may have (on its own) read access, but client does not
    - server might not have read access, but client does



Unix Solution

- Client execs print-server, passing it file name
  - set-uid-root program
  - it (without races!) checks that client has access to file, then prints it

Note that this is not *the* Unix solution, but *a* Unix solution.



Windows Solution

- Server process started when system is booted
- Clients send it print requests
  - how does client prove to server it has access?
  - how does server prove to OS that client has said ok?

Again, this is not *the* Windows solution, but *a* Windows solution.

Impersonation

- Client sends server *impersonation token*
  - subset of its access token
- Server temporarily uses it in place of its own access token

Quiz 1

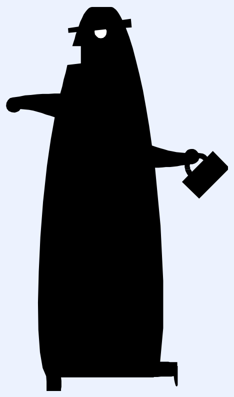
I've written a print server. You would like to use it to print a file. However, you don't trust me — you're concerned that my print server software might read some of your files that you don't want me to read. My print server uses either the Unix approach (setuid-to-twd) or the windows approach (you send it an impersonation token) to deal with access control.

a) You have nothing to worry about
b) You have nothing to worry about if it uses the Unix approach
c) You have nothing to worry about if it uses the Windows approach
d) You have a lot to worry about with both

Security Models

Serious Security

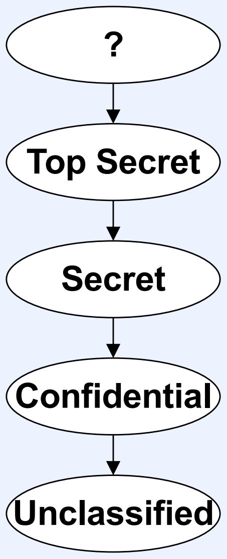
- National defense
- Proprietary information
- Personal privacy

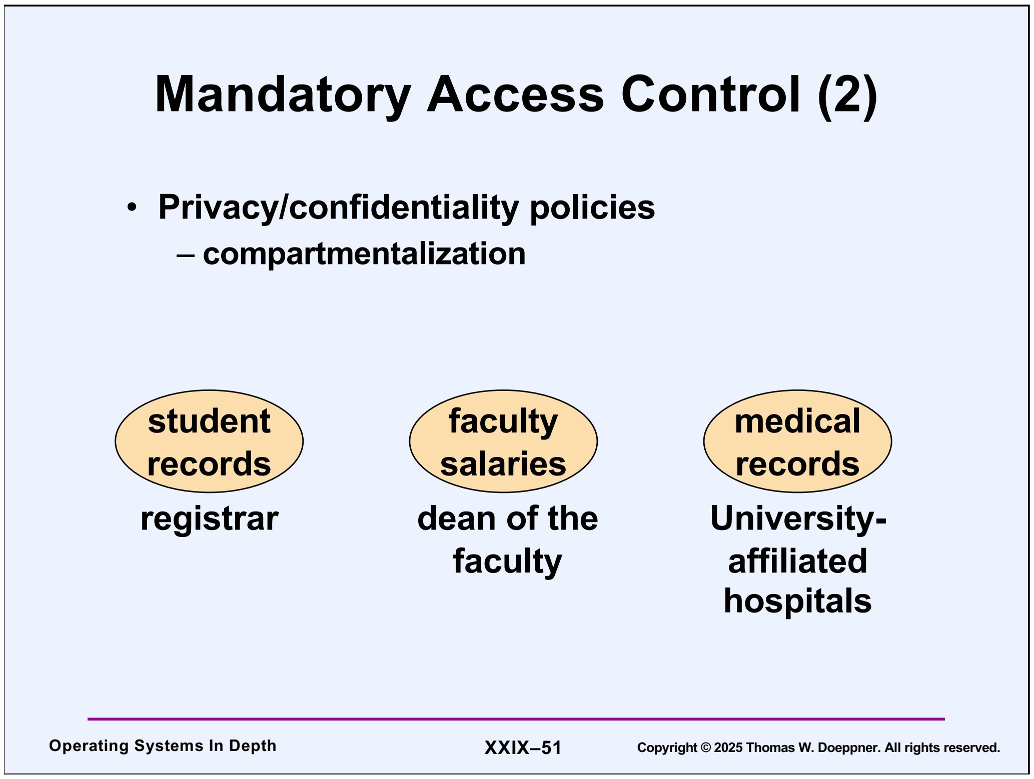


Mandatory vs. Discretionary Access Control

- Discretionary
  - ACLs, capabilities, etc.
    - access is at the discretion of the owner
- Mandatory
  - government/corporate security, etc.
    - access is governed by strict policies

Mandatory Access Control (1)





Mandatory Access Control (2)

- Privacy/confidentiality policies
  - compartmentalization

student records
registrar
faculty salaries
dean of the faculty
medical records
University-affiliated hospitals

Another use of MAC is to enforce compartmentalization. For example, it might be Brown's policy that, for example, people working in the registrar's office have access to student records, but do not have access to faculty salaries. People working in the dean of the faculty's office do have access to faculty salaries, but do not have access to student records. This should continue to be the case even if someone switches jobs (but not computer IDs), moving from the registrar's office to the dean of the faculty's office. Note that this requires a notion of "role": one's role might change from being a registrar person to being a dean-of-the-faculty person.

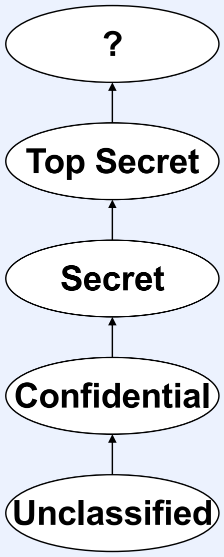
Mandatory Access Control (3)

- Local computer policy
  - web-server
    - may access only designated web-server data
  - administrators
    - may execute only administrative programs
    - (may not execute code supplied by ordinary users)

Bell-LaPadula Model

1) Simple security property **no-read-up**
   - no subject may read from an object whose classification is higher than the subject’s clearance
2) *-property **no-write-down**
   - no subject may write to an object whose classification is lower than the subject’s clearance

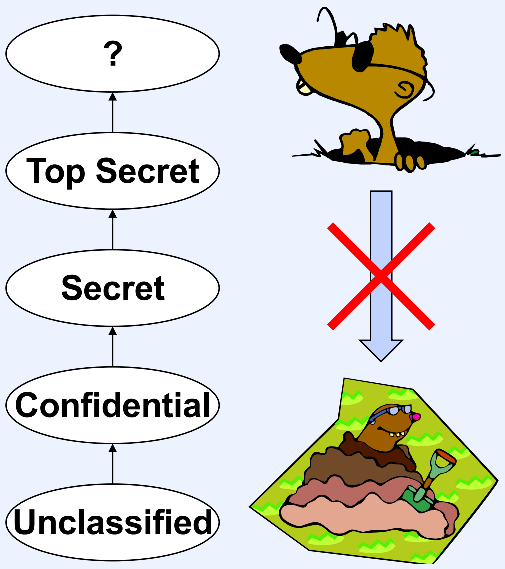
Information Black Hole



Managing Confidentiality

- Black-hole avoidance
  - – trusted vs. untrusted subjects
  - – trusted subjects may write down

Espionage

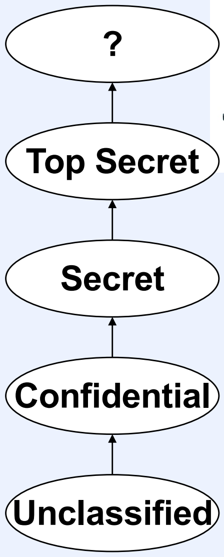


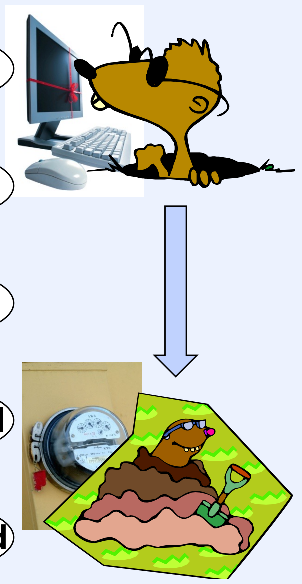
agent X learns of invasion plans

communication not possible

agent Y can send email to spymaster (but doesn't know what to send)

Covert Channels





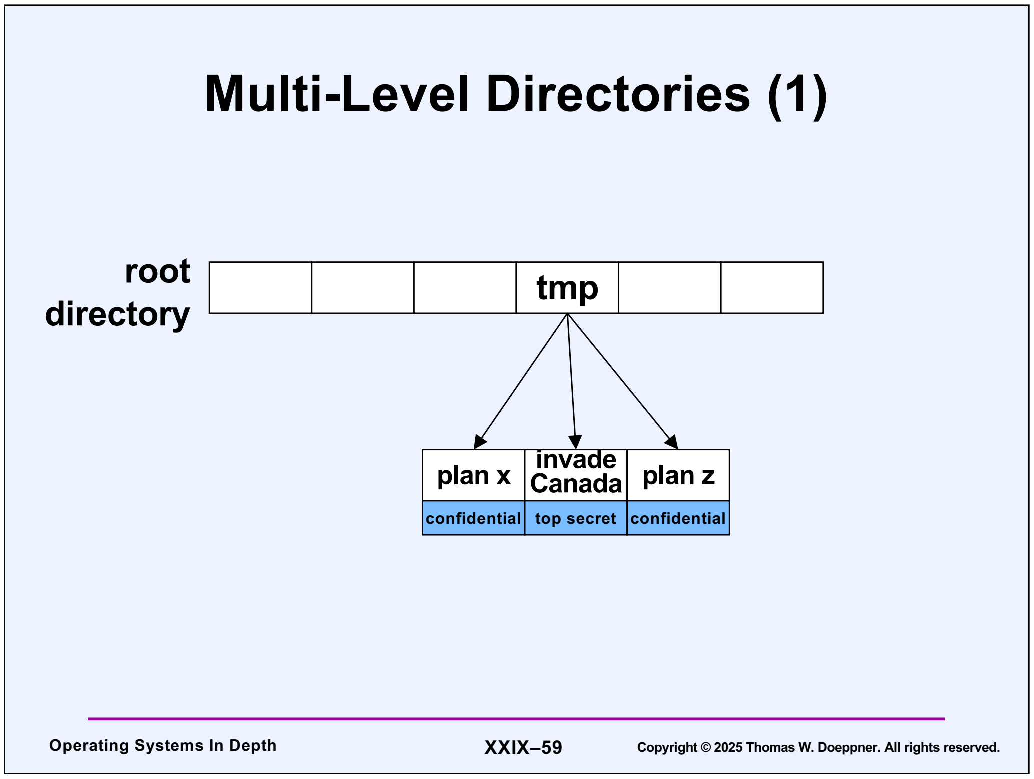
agent X runs resource-intensive program

sneaky communication possible

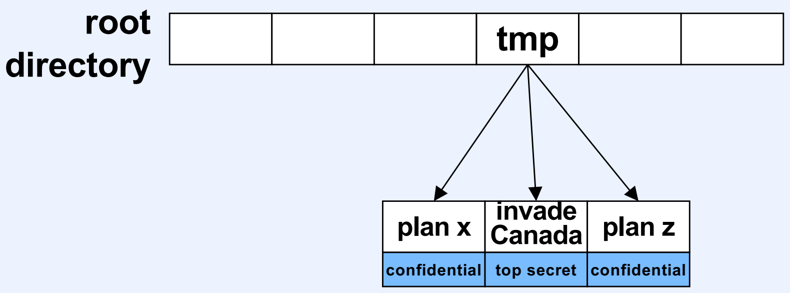
agent Y monitors load sends email to spymaster

Defense

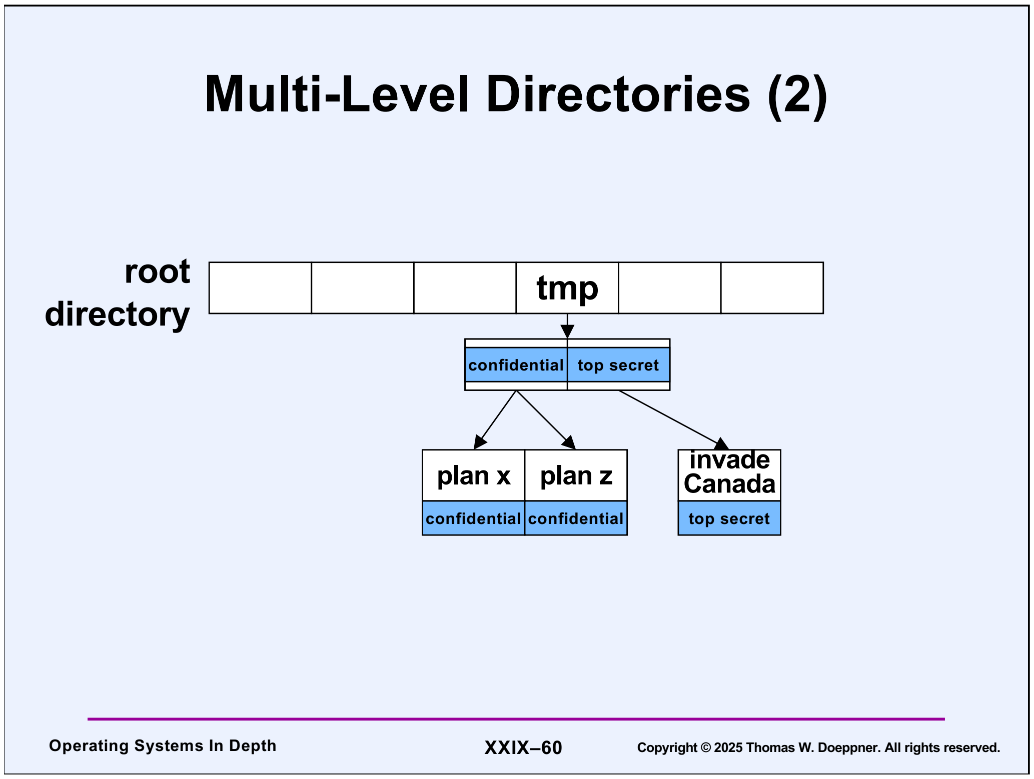
- Identify all covert channels
  - (good luck …)
- Eliminate them
  - find a suitable scheduler
    - eliminates just one channel



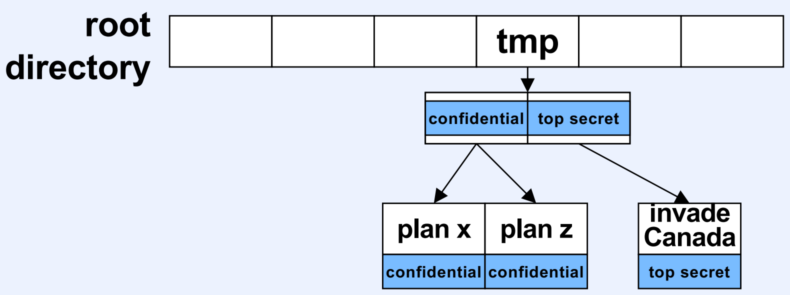
Multi-Level Directories (1)



That there is a file named “invade y” might be considered to be information that shouldn’t be made available to just anyone. However, it’s in a directory that accessible to just anyone. We might come up with an access-permission type that prohibits those without the necessary clearance from seeing the name of a directory entry, but what if someone cleared only for confidential tries to create the file “/tmp/invade y”?



Multi-Level Directories (2)



The solution is to create an implicit subdirectory of /tmp with an entry for each classification. Thus if one is in the **top secret** domain, references to /tmp are actually to /tmp/top secret.