Security Part 4

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).

Quiz 1

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, it can be done in two commands

c) Yes, but it's complicated and requires more than two commands

Real-world Problem: Cross-OS ACL Interoperability

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

For details on NFSv4 ACLs, see RFC 3530 (http://www.ietf.org/rfc/rfc3530.txt), section 5.11.

We discuss NFS in later lectures.

ACLs at Brown CS (Up Till Fall 2019)

- Linux systems support POSIX ACLs
- Windows systems support Windows ACLs
- Servers run GPFS file system and handle NFSv3 and SMB clients
  - GPFS supports NFSv4 ACLs
  - translated to POSIX ACLs and Unix bit vectors for NFSv3 clients
  - translated to Windows ACLs for SMB clients

GPFS stands for general parallel file system. It was developed by IBM for high-performance distributed file services.

SMB stands for server message block and is the means for handling distributed file systems on Windows (and others that have adopted it).

We will discuss both GPFS and SMB in later lectures.

ACLs at Brown CS
(What was Planned for Fall 2019)

- Switch to Isilon servers managed by CIS
  - support NFSv4 and SMB clients
- Linux and Mac clients use NFSv4
  - switch to NFSv4 ACLs
- Windows clients use SMB

This was the plan ...

OSX (Macs)

- Native support for NFSv4 ACLs
  - no setfacl/getfacl commands, but built into chmod
- No NFSv4 client support
  - third-party implementations exist, but they don't work

ACLs at Brown CS (Fall 2019 – Present)

- Isilon servers managed by OIT
  - support NFSv4 and SMB clients
- Linux clients use NFSv4
  - switch to NFSv4 ACLs
- Windows clients use SMB
- OSX clients use SMB
  - no groups – just the authenticated user
  - all remote files seen as allowing 0700 access
  - clients can't observe or modify access protection
    - (though still enforced on server)

Brown's OIT (Office of Information Technology) is the new name for what was called CIS (Computer and Information Services).

Advanced Access Control

setuid and friends

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 (`passwd`)

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

The setuid bit was patented in 1973 by Dennis Ritchie.

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*
- *exec*s 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 …**

The intent of this program is to copy data read from stdin and append it to the end of /tmp/mytemp. But with careful timing and some luck, one can replace /tmp/mytemp with a symbolic link that refers to /etc/passwd, and thus one can add data to the end of the system's password file.

Note that superuser privileges aren't necessary in the code of the slide – assume that this code is an excerpt from code that requires superuser privileges.

Changing Identity (1)

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

```
int setuid(uid_t uid)

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

```
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);
```


This program is exec'd as **setuid** to root. The caller's ID might be twd, the effective ID is root. Thus the program initially has an effective user ID of root. The first call to **seteuid** changes the effective user ID to **caller_id** (twd). Thus the **open** succeeds only if twd may access the file. The program then switches back to an effective user ID of root, and can safely write to the file with the assurance that twd had access to it.

What happens if the owner of the file remove's twd's access rights to it just after open is called? We take this issue up in a later lecture, but, in standard Unix (and Linux), access rights are checked only when a file is opened.

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

Quiz 2

```
/* 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
```

This adds my_assign to the course directory.

a) **It works every time**
b) **It might fail occasionally (nothing gets added)**
c) **It might fail occasionally (something else gets added)**
d) **It never works**

```
/* 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?

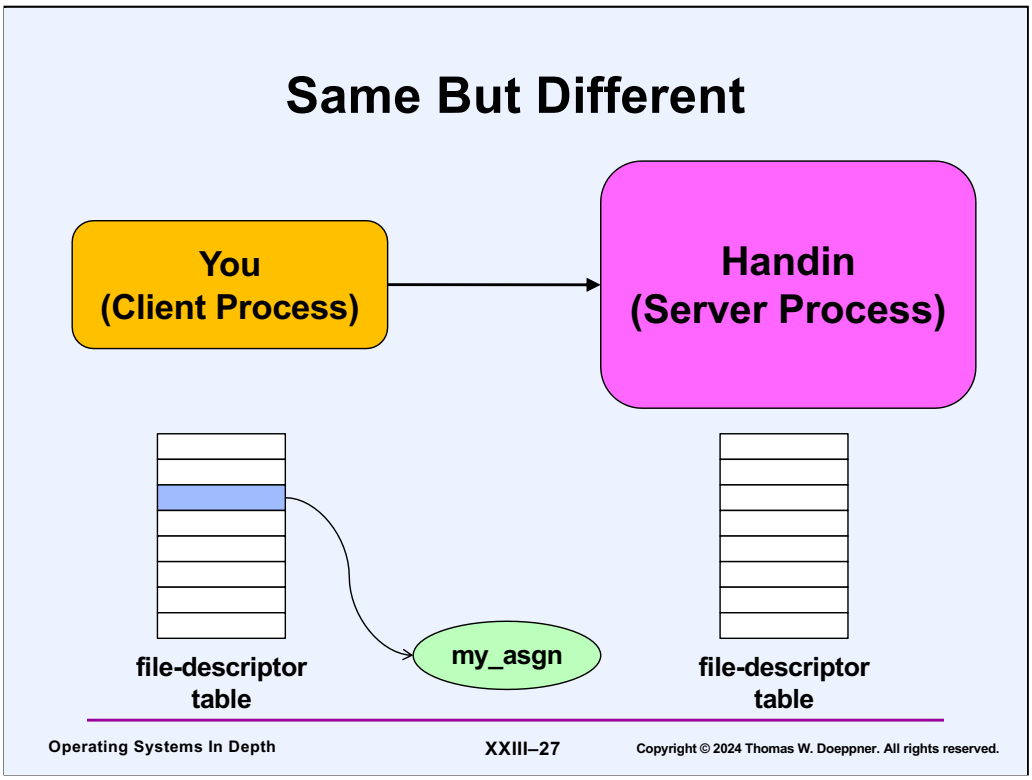
- **Could use previous solution**

or

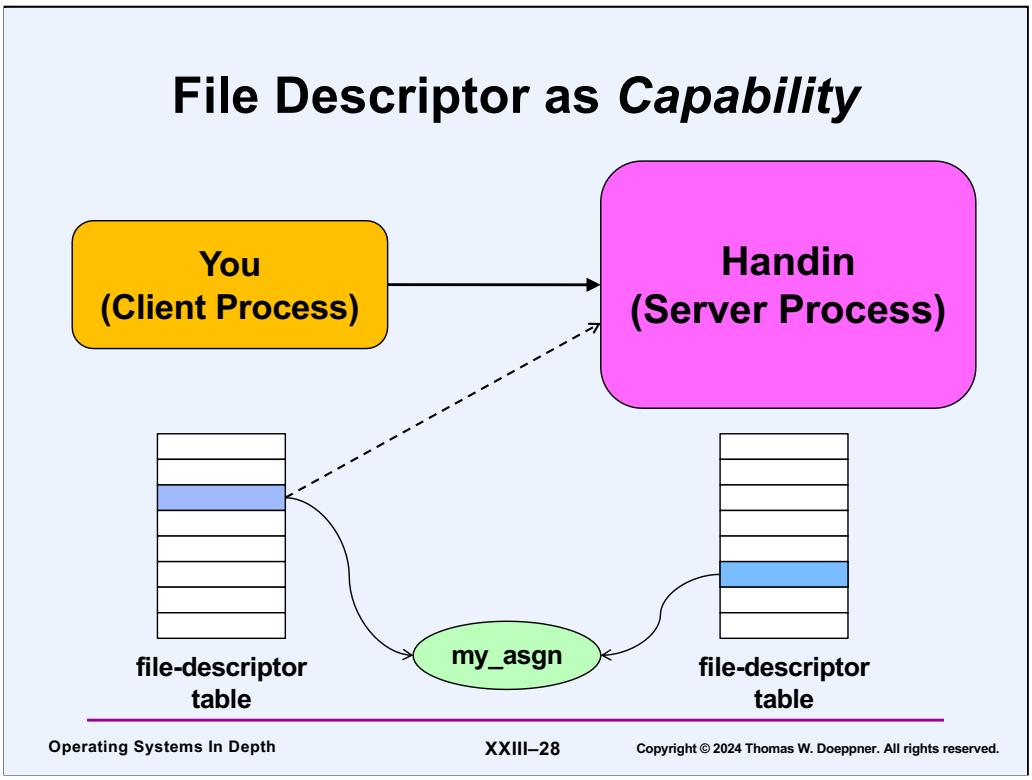
```
afd = open("my_assgn",
       O_RDONLY);
close(0);
dup(afd);
close(afd);
execl("handin", 0);
```

```
/* 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).



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.



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;

read(fd, &len, sizeof(len));

if (len > 100) {
  fprintf(stderr, "bad length\n");
  exit(1);
}

read(fd, buf, len);
```


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.

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

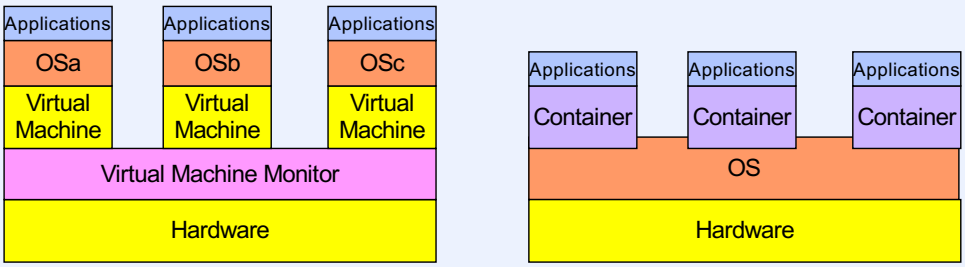
- Perhaps:
  - run process with a minimal security context
    - special account, etc.
  - send it the capabilities it needs

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

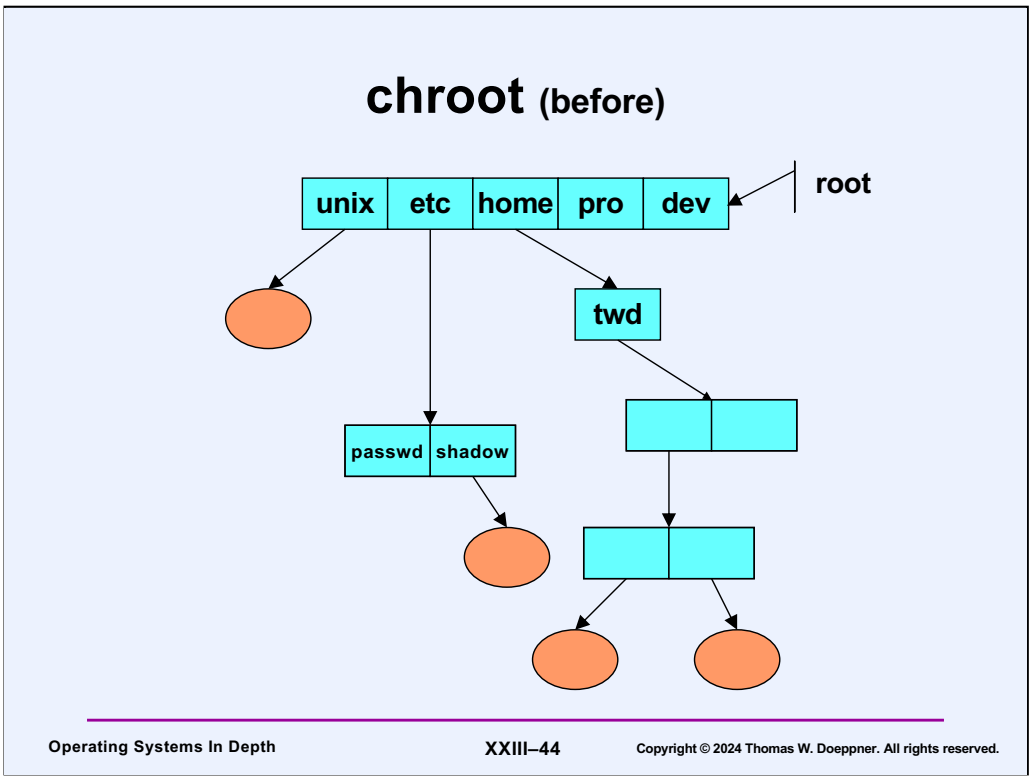


Containers

- 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

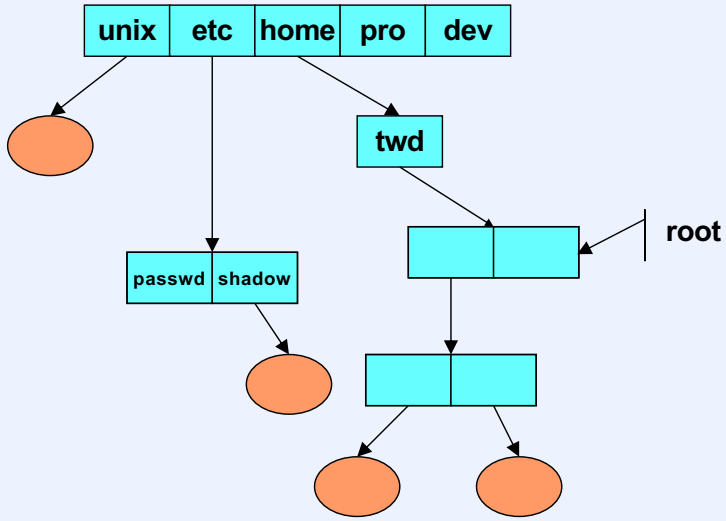
Container Building Blocks

Part 1: File system (chroot)



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

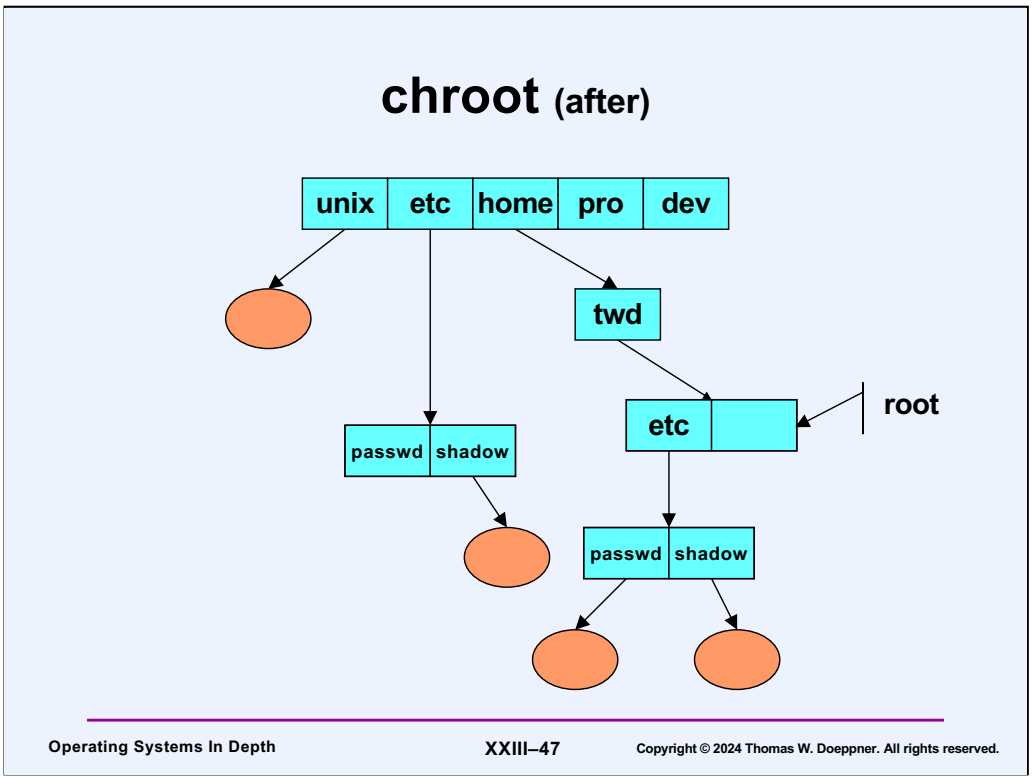
chroot (after)



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



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