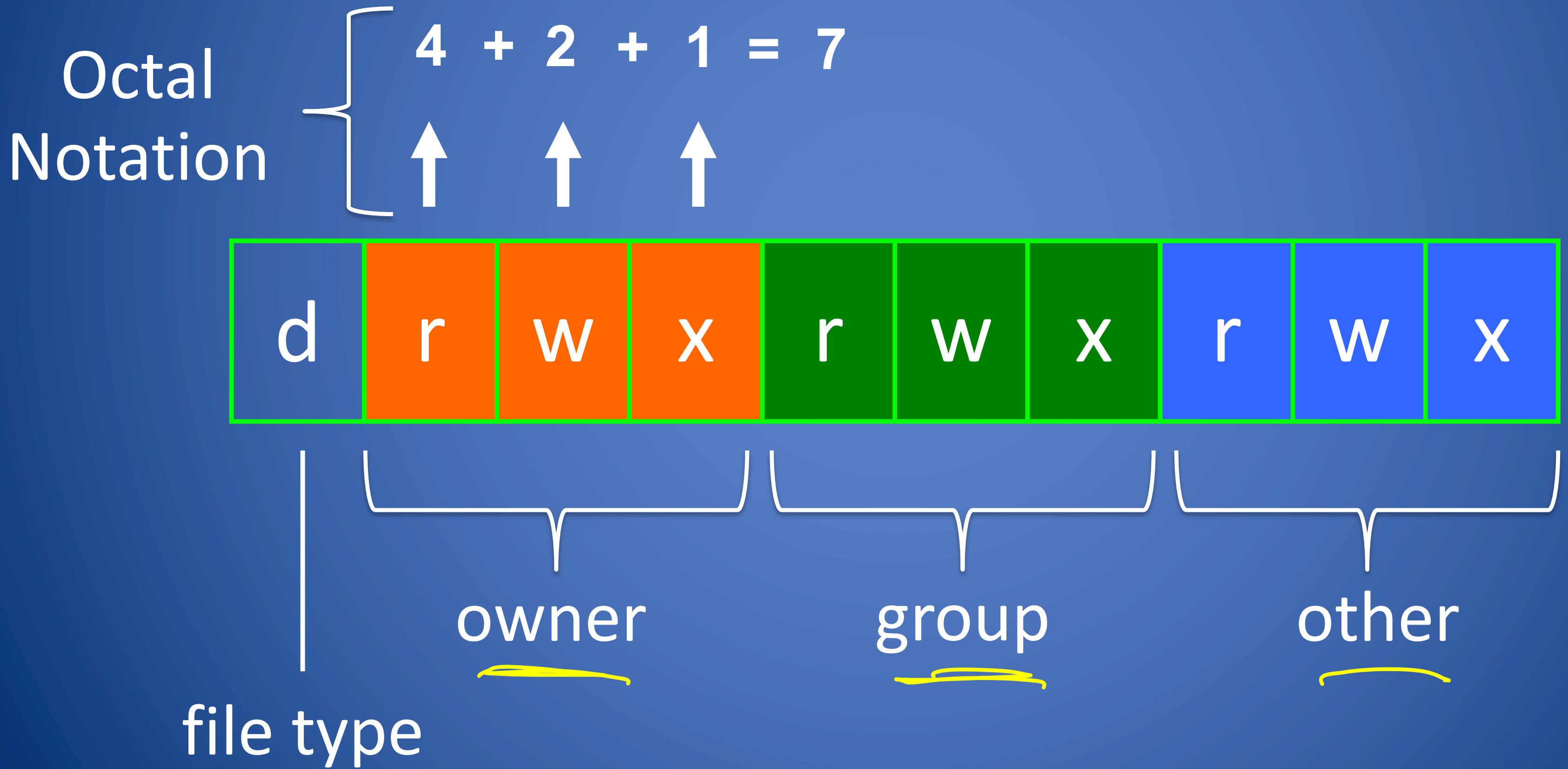
Operating Systems Security II

Privileges and setuid/setgid

CS 1660: Introduction to Computer Systems Security

RECAP

Unix File Types RWX and octal notation



When a user runs a process, the OS keeps track of:
- UID: user running the process
- GID: group ID for that user

- EUID: "effective UID => UID as used for permissions checks, etc.
- EGID: "effective" GID

UID 1000 => open("/home/alice/file.txt", ...)
    => Considers effective UID/GID to decide if you have access

=> Normally UID == EUID, GID == EGID, except with setuid or setgid

setuid/setgid

Special permissions bits:

- setuid (Set User ID): executable runs with privileges of owner, regardless of who runs it
- ~~setuid~~ setgid (Set Group ID): executable runs with privileges of group, regardless of who runs it

setuid/setgid

Special permissions bits:

- setuid (Set User ID): executable runs with privileges of owner, regardless of who runs it
- setuid (Set Group ID): executable runs with privileges of group, regardless of who runs it

Unprivileged user can run program with higher privileges!
=> Powerful, but very dangerous

Disclaimer

setuid/setgid is dangerous. Using it incorrectly can cause serious problems.

Just as you should never implement your own crypto,
you should not write your own setuid/setgid programs.

You are about to see why.

Background: environment variables

System variables that control how processes execute

Set up when a user logs in, as part of shell

```
# Get variables
cs1660-user@6010f6e96b02:~$ echo $TERM
xterm
cs1660-user@6010f6e96b02:~$ echo $PWD
/home/cs1660-user

# Set a variable
cs1660-user@6010f6e96b02:~$ export SOMETHING=hello
cs1660-user@6010f6e96b02:~$ echo $SOMETHING
Hello

# Show the environment
cs1660-user@6010f6e96b02:~$ env
. . .
```

Scope is per-shell: log out/open new term => different vars

Background: $PATH

Where the shell looks when you run programs

=> List separated by ":", traversed in order

```
# Get variables
cs1660-user@6010f6e96b02:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/go/bin


# which:  $PATH lookup
cs1660-user@6010f6e96b02:~$ which ls
/usr/bin/ls

cs1660-user@6010f6e96b02:~$ which go
/usr/local/go/bin/go
```

Problems

Input from user pollutes execution environment

=> Another form of code injection!

Not every command can be overridden…

Aside: Some common commands, like "echo" are so common they're part of the bash shell itself => these are called builtins

Older shells execute almost everything as a comment, but more modern shells like bash optimize this with some builtin commands

=> Search "bash builtins" for more info

Background: symbolic links

Indirection in the filesystem: path of one file can point to another

```
# Create a symlink
registrar@ceres:~$ ln –sv scripts/reg-v01.sh reg.sh
reg.sh -> scripts/reg-v01.sh


# How it looks
registrar@ceres:~$ ls –la reg.sh
lrwxrwxrwx 1 reg reg 9 Mar 12 16:40 reg.sh -> scripts/reg-v01.sh


# Use it just like a normal file
registrar@ceres:~$ ./reg.sh
```

Problem: anyone can create a symlink to anything!
=> Permissions checked on access, not at **creation**

What can we do about vulnerabilities like this?

Problem: if we compare a path like this:

```
if compare /source/path /dest/path
    // DO thing
```

The open() and read() of the file is happening in a privileged context

Options:
- Drop privileges: do check as effective user
- Could get unprivileged program to provide input in a different way
  => Pass code on stdin, that way alice is the one that needs to open() and read() the file

Principle of least privilege => avoid doing operations with more privileges than necessary

What ELSE *can go wrong?*

TOCTOU: Time of check/time of use

```
# Check for access
if ! __effective_user_can_access $code_from_user; then
        echo "You don't have permission to view this file"
        exit 1
fi


# Do the access
if cmp --silent $code_expected $code_from_user; then
    echo "Override code approved!"
    add_to_course $course $user
else
    echo "Please use a valid override code"
fi
```

A race condition!

So why is setuid/gid bad?

Up to the developer to decide what parts of the program can run with elevated privileges

=> Particularly dangerous for shell scripts

Break!

So setuid/setgid is dangerous...

setuid/setgid is dangerous...

In modern times: only for programs that really need it

- System programs that changing passwords/users, legacy programs
  - Don't do this yourself!
- Very very bad idea for shell scripts

What else can we do?

When do we need this?

In the shell: su, sudo

- Run as another user (if you have permissions)

```
user@shell:~$ su -c "command" other user
```

- Run commands as root (or another user) based on system config file (/etc/sudoers)
  - Can restrict to specific commands, environment, ...

```
user@shell:~$ sudo whoami
root
```

```
/etc/sudoers:
%wheel ALL=(ALL) NOPASSWD: ALL

. . .
```

From man page on /etc/sudoers: (aka sudoers(5))

```
ALL                CDROM = NOPASSWD: /sbin/umount /CDROM,\
                   /sbin/mount -o nosuid\,nodev /dev/cd0a /CDROM


Any user may mount or unmount a CD-ROM on the machines in the CDROM
Host_Alias (orion, perseus, hercules) without entering a password.
```

sudo has a LOT of features, see
`man sudoers` for details!

Principle of Least Privilege

An operation should only be able to perform the operations necessary for its intended purpose

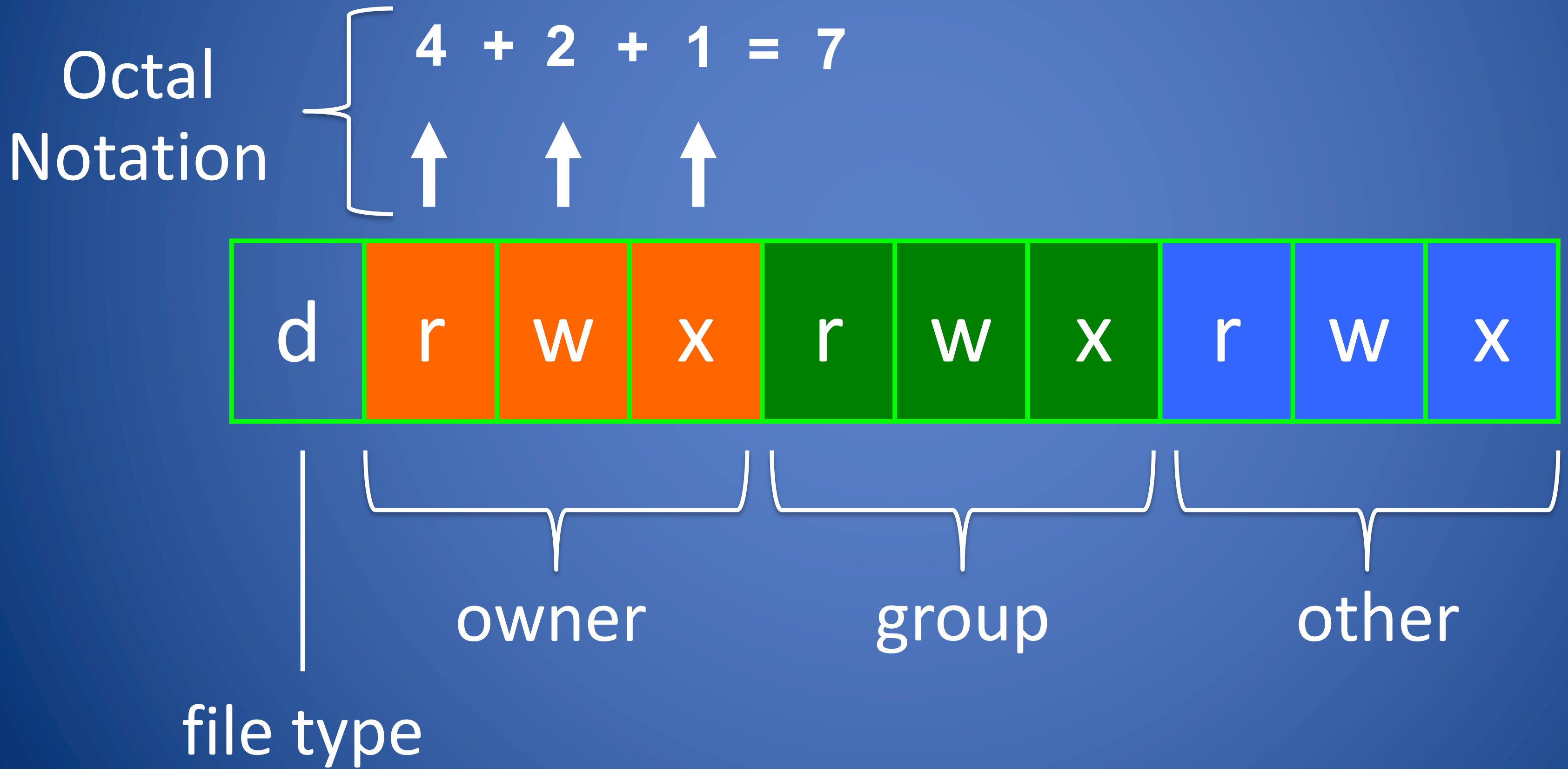
What ELSE could we do?

Separation of processes

- System service runs as privileged user
- Client program run by unprivileged users
- Some API for how these programs communicate
  - Local network connection
  - Unix socket
  - dbus or other IPC mechanism
  - …

More content for reference

Unix File Types RWX and octal notation



Octal Notation (recap)

FOR PERMISSIONS

Another way to specify permissions

- Digits from left (most significant) to right(least significant):

  [special bits][user bits][group bits][other bits]

- Special bit digit =

  (4 if setuid) + (2 if setgid) + (1 if sticky)

- All other digits =

  (4 if readable) + (2 if writable) + (1 if executable)

Permissions Examples (Regular Files)

-rw-r—r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden to others
-rwx------	read/write/execute for owner, forbidden to everyone else
-r--r--r--	read-only to everyone, including owner
-rwxrwxrwx	read/write/execute to everyone

Permissions for Directories

- Permissions bits interpreted differently for directories
- *Read* bit allows listing names of files in directory, but not their properties like size and permissions
- *Write* bit allows creating and deleting files within the directory
- *Execute* bit allows entering the directory and getting properties of files in the directory
- Lines for directories in `ls -l` output begin with d, as below:

```
jk@sphere:~/test$ ls -l
Total 4
drwxr-xr-x  2 jk ugrad 4096 2005-10-13 07:37 dir1
-rw-r--r--  1 jk ugrad    0 2005-10-13 07:18 file1
```

Permissions Examples (Directories)

drwxr-xr-x	all can enter and list the directory, only owner can add/delete files
drwxrwx---	full access to owner and group, forbidden to others
drwx--x---	full access to owner, group can access known filenames in directory, forbidden to others
-rwxrwxrwx	full access to everyone

The /tmp Directory

- In Unix systems, directory /tmp is
  - Read/write for any user
  - Wiped on reboot (or lives entirely in memory)

Convenience
  - Place for temporary files used by applications
  - Files in /tmp are not subject to the user’s space quota

What could go wrong?

setuid bit: Set-user-ID

- On executable files, causes the program to run as file owner regardless of who runs it

- How to view: shown as s instead of x

  -rwsr-xr-x: setuid, executable by all

  -rwxr-xr-x: executable by all, but not setuid

Setuid Programs

- Unix processes have two user IDs:
  - real user ID (UID): user launching the process
  - effective user ID (EUID): user whose privileges are granted to the process
- If a user A executes a setuid file owned by B, then the effective user ID of the process is B and not A

Setuid Programs

- System call setuid(uid) allows a process to change its effective user ID to uid
- Some programs that access system resources are owned by root and have the setuid bit set (setuid programs)
  - e.g., passwd and su
- Setuid generally ignored on shell scripts—why?

setgid bit: Set-group-ID (recap)

- On executable files: causes the program to run with the file's group, regardless of whether the user who runs it is in that group
- On directories, causes files created within the directory to have the same group as the directory

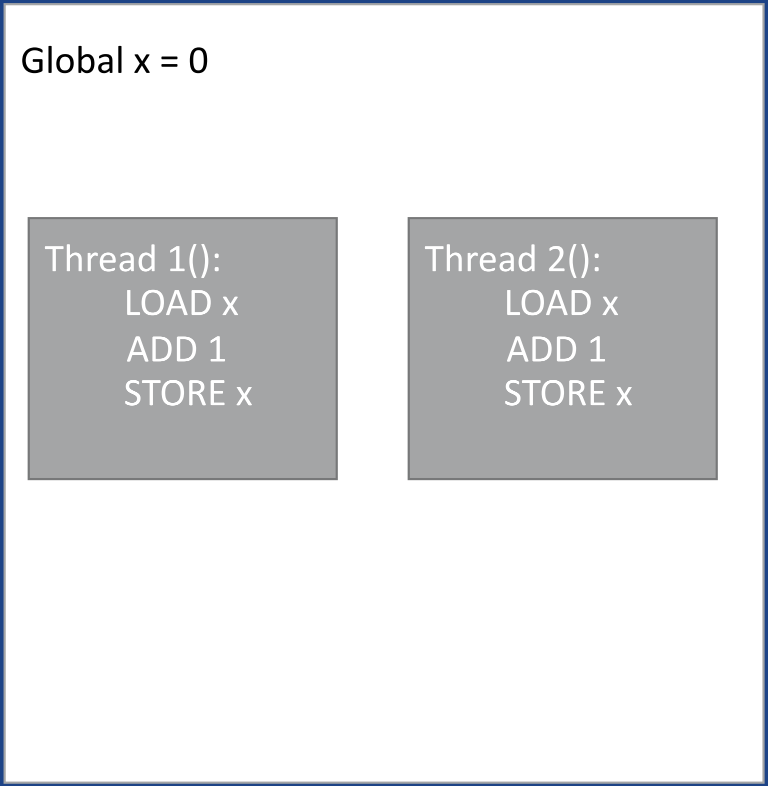
```
Examples
      -rwxr-sr-x: setgid file, executable by all
      drwxrwsr-x: setgid directory; files within will have group of directory
```

Time of Check /Time of Use (TOCTOU) eg. Race Condition

Race Condition

- A race condition occurs when two threads want to access the same memory

- Run Thread 1() and Thread 2()
  - Outcome is 1 or 2



Race Condition

```
1.   if (!access("/tmp/X", W_OK)) {
         /* the real user ID has access right */
2.       f = open("/tmp/X", O_WRITE);
3.       write_to_file(f);
         }
     else {
         /* the real user ID does not have access
         right */
4.       fprintf(stderr, "Permission denied\n");
         }
```

Source: Kevin Du, Race Condition Vulnerability, Lecture Notes

- Fragment of setuid program that writes into file /tmp/X on behalf of a user who created it
- access verifies permission of real user ID
  - Transparently follows symlinks
- open verifies permission of effective user ID
  - Transparently follows symlinks
- What can go wrong?

TOCTOU Vulnerability

```
1.  if (!access("/tmp/X", W_OK)) {
        /* the real user ID has access right */
2.      f = open("/tmp/X", O_WRITE);
3.      write_to_file(f);
        }
    else {
        /* the real user ID does not have
        access right */
4.      fprintf(stderr, "Permission denied\n");
        }
```

- What can go wrong?
  - In between (1) and (2), user could replace /tmp/X with symlink to /etc/passwd
  - Not easy to accomplish (timing)
- Example of time of check to time of use (TOCTOU) vulnerability

Attempt to Fix the Race Condition

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.    int f = open("/tmp/X", O_RDWR);
4.    fstat(f, &statAfter);
5.    if (statAfter.st_ino == statBefore.st_ino) {
        /* the I-node is still the same */
6.      write_to_file(f);
      }
7.    else perror("Race Condition Attacks!");
      }
8. else fprintf(stderr, "Permission denied\n");
   }
```

Source: Kevin Du, Race Condition Vulnerability, Lecture Notes

- lstat and fstat access file descriptor for a path, which includes unique file ID (st_ino)
  - lstat does not traverse symlink
  - fstat accesses descriptor of open file, after symlink traversed by open
- Step (5) compares IDs of
  - file checked in (1) and
  - file opened in (3)
- Check-use-check_again approach
  - Defeats swapping in symlink between access and open
- Fails also if /tmp/X is a symlink when (2) is executed

Does the Fix Work?

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.    int f = open("/tmp/X", O_RDWR);
4.    fstat(f, &statAfter);
5.    if (statAfter.st_ino == statBefore.st_ino) {
        /* the I-node is still the same */
6.      write_to_file(f);
       }
7.    else perror("Race Condition Attacks!");
       }
8. else fprintf(stderr, "Permission denied\n");
    }
```

- New attack
  - Before (1) /tmp/X is a hard link to /etc/passwd
  - Between (1) and (2) swap in hard link to user-owned file
  - Between (2) and (3) swap in again hard link to /etc/passwd
- This passes the ID check in (5) and allows the user to write to /etc/passwd

Negative Result

- Assumptions
  - Setuid program
  - Path-based permission check for real user ID via syscall access(path, permission) that returns 0 or -1
  - No atomic check-and-open file syscall
- Theorem
  - Program is vulnerable to TOCTOU race condition
- Proof
  - Attacker can always swap good file before access and bad file after access
  - lstat/fstat do not help since they are path-based as well
- Reference
  - Drew Dean, Alan J. Hu: Fixing Races for Fun and Profit: How to Use access (2). USENIX Security Symposium, 2004.

Mitigating and Eliminating Race Conditions

- Hardness amplification
  - Force the adversary to win a large number of races instead of just one or two in order to exploit the vulnerability
  - Reduces the probability of success
  - Complex to accomplish correctly
  - Reference
    - Dan Tsafrir, Tomer Hertz, David Wagner, Dilma Da Silva: Portably Solving File TOCTTOU Races with Hardness Amplification. USENIX File and Storage Technologies, 2008

- Temporary privilege downgrade
  - Within same process
    - Drop to real user ID privileges via setuid(real_userid)
    - Open file
    - Restore root privileges
  - With child process
    - Fork child process with real user ID privileges to open file
  - Approach not portable across Unix variants

https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf

Historical setuid Unix Vulnerabilities: lpr

- Command lpr
  - running as root setuid
  - copied file to print, or symbolic link to it, to spool file named with 3-digit job number (e.g., print954.spool) in /tmp
  - Did not check if file already existed
  - Random sequence was predictable and repeated after 1,000 times
- How can we exploit this?

- Attack
  - A dangerous combination: setuid, /tmp, symlinks, …
  - Create new password file newpasswd
  - Print a very large file
  - lpr –s /etc/passwd
  - Print a small file 999 times
  - lpr newpasswd
  - The password file is overwritten with newpasswd

https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Race_Condition.pdf

What We Have Learned

- Code as Data
- Setuid programs
- Dangers of symlinks, setuid, and shared directories
- Race conditions and time-of-check-to-time-of-use for access/open syscalls
- Examples of Attacks