CS202 (003): Operating Systems
Unix Security

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish’s CS202

Last time

Protections and security in Unix

U(ser)ID and G(roup)ID

Files and directories are access-controlled:
system stores with each file who owns it (in inode)

Root (UID 0)

Has all the permissions: read any file, do anything, …

Some legitimate actions require more privileges than UID

How should users change their passwords (root-owned)?

Each process has a real and effective UID/GID

Real is user who launched the program, effective is owner/group executables, used in access checks

Setuid

a program that is run in with **raised privilege level**

SetUID/SetGID

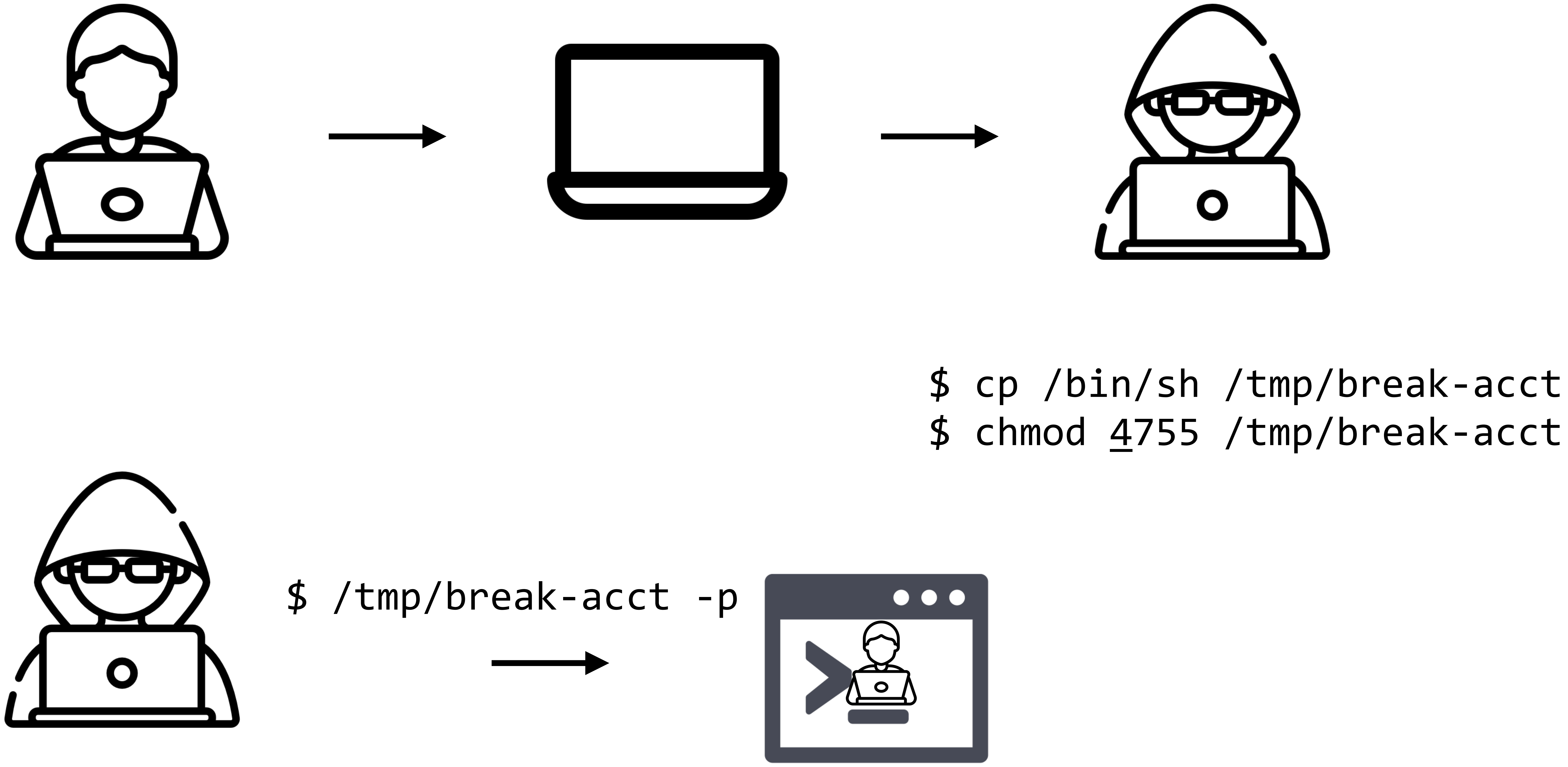
A way for the root (or another user) to delegate its ability to do something

```
cs202-user@4b5e43aed385:~/cs202-labs$ ls -l `which passwd`
-rwsr-xr-x 1 root root 72056 May 30  2024 /usr/bin/passwd
```

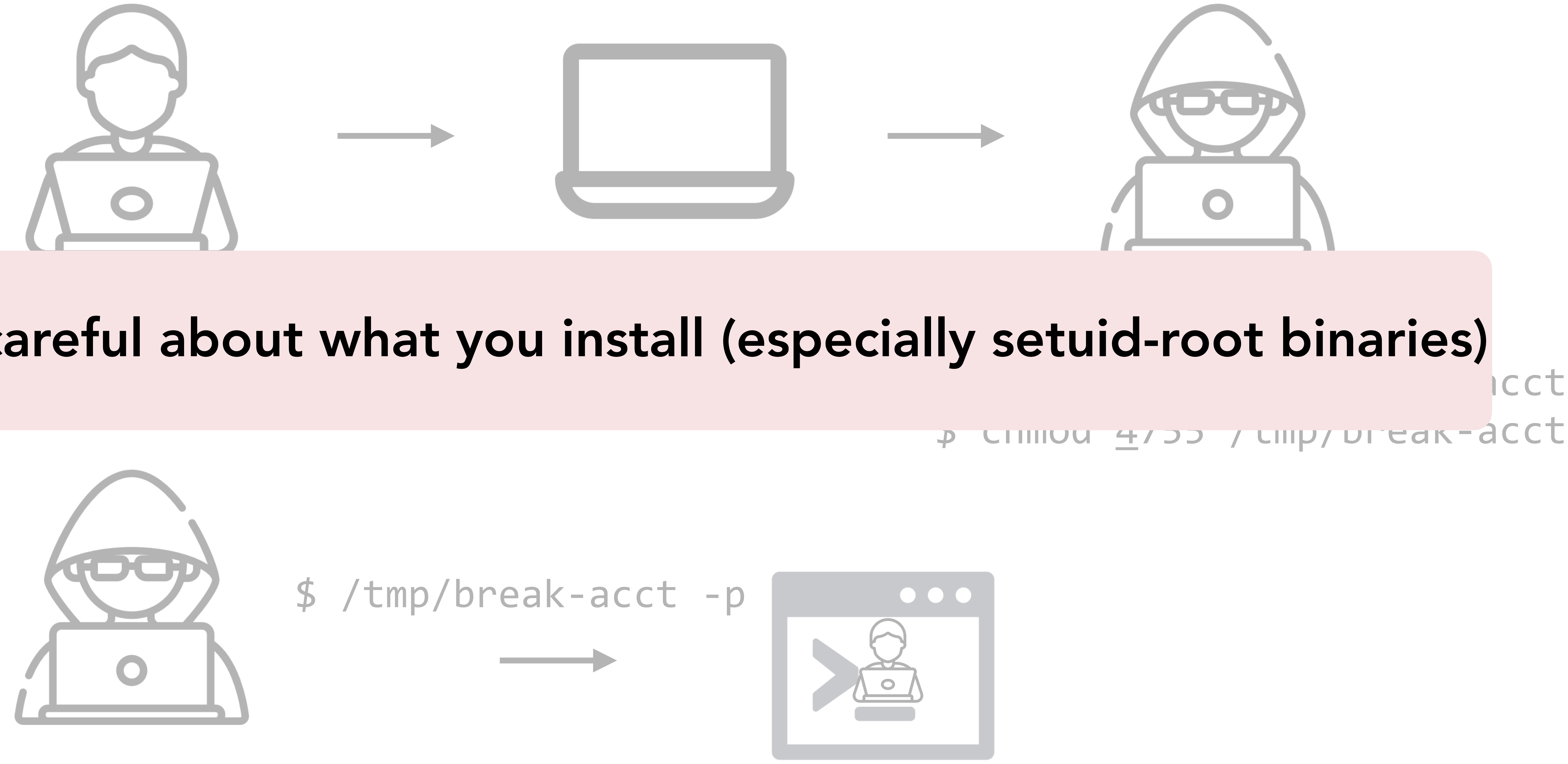
special bit in the permissions

Need to own the file to set the setuid bit
(also need to be in group to set setgid bit)

Consider the following scenario



Consider the following scenario



Be careful about what you install (especially setuid-root binaries)

Example attacks

```
Attacker Setup:
close(2);                // Attacker closes stderr (file descriptor 2)
exec("/usr/bin/passwd") // Then launches the passwd program
```

```
passwd:
main() {
    fd = open("/etc/passwd", ...);  // Opens the password file
    ……
    fprintf(stderr, "Err msg\n");   // Tries to print an error message to stderr
}
```

Example attacks

<u>Attacker Setup:</u>

```
ulimit -f 0               // Sets the max file size limit to zero
exec("/usr/bin/passwd") // Then launches the passwd program
```

<u>passwd:</u>

```
... // verify the user’s current password
... // prompt for and validate the new password
... // try to update the password information
```

<u>What might happen:</u>

password update write fail

need to handle the error

-> if not handle well, you might correct the password database, or etc.

Example attacks

IFS (Internal File Separator)
a special shell environment variable in Unix and Unix-like systems that defines the characters the shell uses to split words and process command lines

Example attacks

The Starting Point:

- There was a program called "preserve" that was installed with setuid **root** permissions
- This program was used by old text editors like vi to create backup files in root-accessible directories
- When preserve runs, it uses the system() call to execute "/bin/mail" to notify users about backups

The Vulnerability Chain:

1. The attacker first manipulates the IFS (Internal Field Separator) environment variable, setting it to "/"
2. When running vi, which triggers preserve:
   - vi executes preserve with setuid root privileges
   - preserve then calls system("/bin/mail")
   - Due to the modified IFS, the shell parses "/bin/mail" as two separate words: "bin" and "mail"

The Exploit:

- The attacker creates a malicious executable named "bin" in their directory
- When the system() call runs, instead of executing /bin/mail, it finds and executes the attacker's "bin" program
- The malicious "bin" program, now running with root privileges, can:
  1. Reset IFS to normal (spaces, tabs, newlines)
  2. Create a copy of /bin/sh
  3. Change the ownership to root (chown root)
  4. Set the setuid bit (chmod 4755)

http://web.deu.edu.tr/doc/oreily/networking/puis/ch05_05.htm

Example attacks

The Starting Point:

- There was a program called "preserve" that was installed with setuid **root** permissions
- This program was used by old text editors like vi to create backup files in root-accessible directories
- When preserve runs, it uses the system() call to execute "/bin/mail" to notify users about backups

The Vulnerability Chain:

1. The attacker first manipulates the IFS (Internal Field Separator) environment variable, setting it to "/"
2. When running vi, which triggers preserve:
   - vi executes preserve with setuid root privileges
   - preserve then calls system("/bin/mail")
   - Due to the modified IFS, the shell parses "/bin/mail" as two separate words: "bin" and "mail"

The Exploit:

- The attacker creates a malicious executable named "bin" in their directory
- program
-

How can we fix this?

(shell has to ignore IFS if the shell is running as root or if EUID != UID)

3. Change the ownership to root (chown root)
4. Set the setuid bit (chmod 4755)

Example attacks

ptrace

Provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers.

It is primarily used to implement breakpoint debugging and system call tracing.

https://man7.org/linux/man-pages/man2/ptrace.2.html

Example attacks

Attack 1 - Direct Privilege Escalation:

- The fundamental issue is an unprivileged process attempting to ptrace a privileged (setuid) program
- This would allow the attacker to manipulate the memory of a root process, effectively gaining root privileges
- The solution implemented was to prevent processes from ptracing more privileged processes or processes owned by other users
- The security check requires the tracing process to have matching real and effective UIDs as the target

Attack 2 - Privilege Escalation via exec():

- More subtle attack where an unprivileged process A traces another unprivileged process B
- Initially this is fine since they have the same privileges
- The vulnerability occurs when B executes a setuid program (like 'su')
- This would result in A having debug control over a now-privileged process
- The fix was to disable the setuid bit when a traced process calls exec()
- An exception is made for root, which can still ptrace any process

Attack 3 - Complex Privilege Escalation Chain:

- This is a sophisticated attack that bypasses the previous two fixes
- Process A traces Process B (both unprivileged)
- A executes "su attacker" (becoming temporarily root during su execution)
- During this window, B executes "su root"
- Because A is temporarily root, B's exec() maintains the setuid bit (bypassing Attack 2's fix)
- The attacker can then manipulate B's memory during the password check
- This results in A being connected to a root shell

Example attacks

TOCTTOU attacks (time-of-check-to-time-of-use)
Exploit the time gap between when a program checks a resource's properties and when it actually uses that resource. This race condition can lead to serious security vulnerabilities.

Example attacks

Problem:

```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #1:

```
if (access(logfile, W_OK) < 0)
    return ERROR;
fd = open(logfile, ...);
```

Does this solve the problem?

Attack Sequence:

The attacker runs the setuid program with "/tmp/X" as the logfile parameter

```
    The program                               Attacker
                                             create("/tmp/X");
  check access("/tmp/X") --> OK
                                             unlink("/tmp/X");
                                            symlink("/etc/passwd", "/tmp/X")
    open("/tmp/X")
```

The program then opens what it thinks is "/tmp/X" but is actually "/etc/passwd"

Issue: check (access()) and use (open()) operations are not atomic

Result: The privileged program inadvertently writes to the password file

Example attacks

Problem:
```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #2:
Use file descriptor-based operations that are relative to an already opened directory:

```
  openat(), renameat(), unlinkat(), symlinkat(), faccessat()
  fchown(), fchownat(), fchmod(), fchmodat(), fstat(), fstatat()
```

```
// CHECK: Does /home/user/file exist?
if (access("/home/user/file", W_OK) < 0)
    return ERROR;

// USE: Open /home/user/file
// BUT what if the path changed
   between check and use?
fd = open("/home/user/file", O_WRONLY);
```

```
// Open the directory first
int dir_fd = open("/home/user", O_DIRECTORY);

// All subsequent operations are relative to this diectory
// Even if attacker changes symlinks/paths,
    we're still operating relative to our original directory
if (faccessat(dir_fd, "file", W_OK, 0) < 0)
    return ERROR;

fd = openat(dir_fd, "file", O_WRONLY);
```

Example attacks

Problem:

```
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

a setuid program that is readable/writable by everyone

Fix #3:

Path Traversal Verification:

- Manually traverse the path
- Verify each directory component
- Check for unexpected symbolic links
- Verify path hasn't been modified during operations

Fix #4:

Transactional Approaches:

- Use operating system-level transactions where available