Set-UID Privileged Programs

WPA2 Krack

Oh no! They found a vulnerability in WPA2 - we can't trust any WiFi any more!
We need to change every router, starting with ours!
Calm down guys, we'll just need to update the firmware and everything'll be fine...
Are you sure?
Yes! We just need to wait for the manufacturers to release a patch for all their router models, and then for all the sysadmins to flash the new firmware...
CommitStrip.com

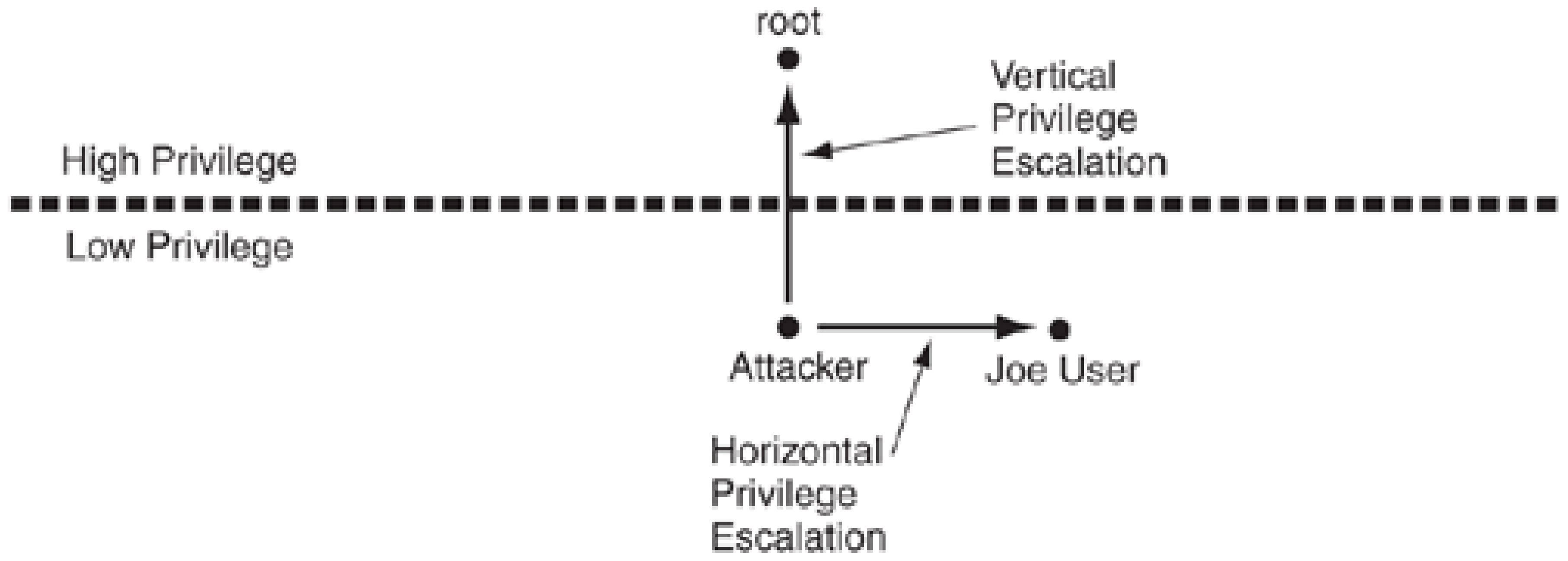
Nearly all men can stand adversity, but if you want to test a man’s character, give him power.

–ABRAHAM LINCOLN

Introduction

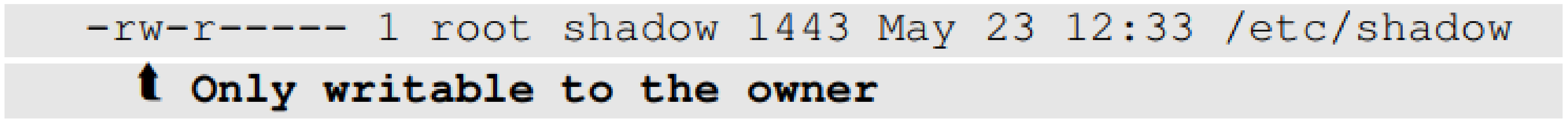
- Privileged programs grant regular users a limited amount of access to some shared resource,
  - physical memory,
  - a hardware device, or
  - special files
  - binding to low-numbered (<1024) network ports
  - altering global OS configuration (registry and/or files)
- Written improperly
  - A vertical privilege escalation attack allows uncontrolled access to the elevated privileges
  - A horizontal privilege escalation, attacker circumvents an application’s access control mechanisms to access resources belonging to another user

Privilege Escalation Types


root
Vertical Privilege Escalation
High Privilege
Low Privilege
Attacker
Joe User
Horizontal Privilege Escalation

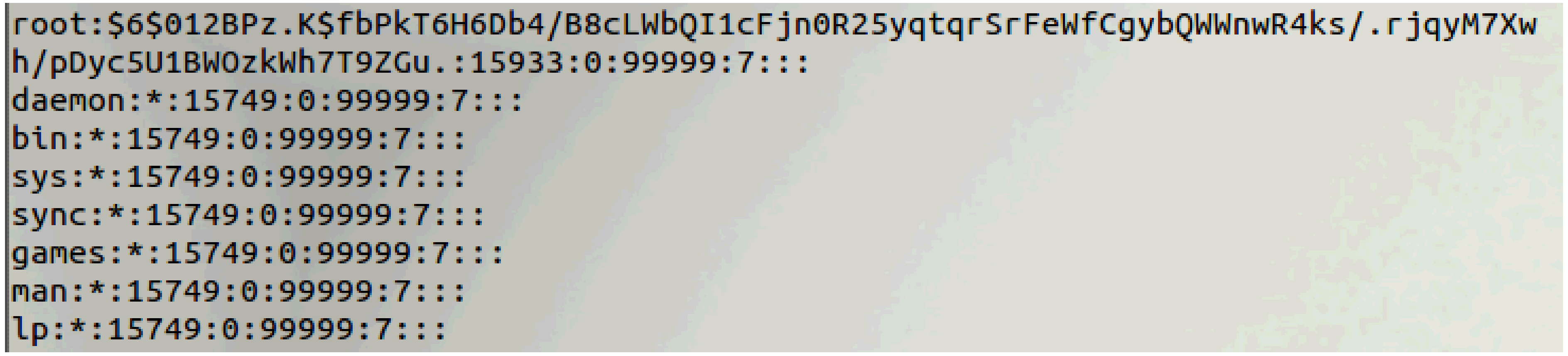
Need for Privileged Programs

- Password Dilemma
  - Permissions of /etc/shadow File:



```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow
  ↑ Only writable to the owner
```

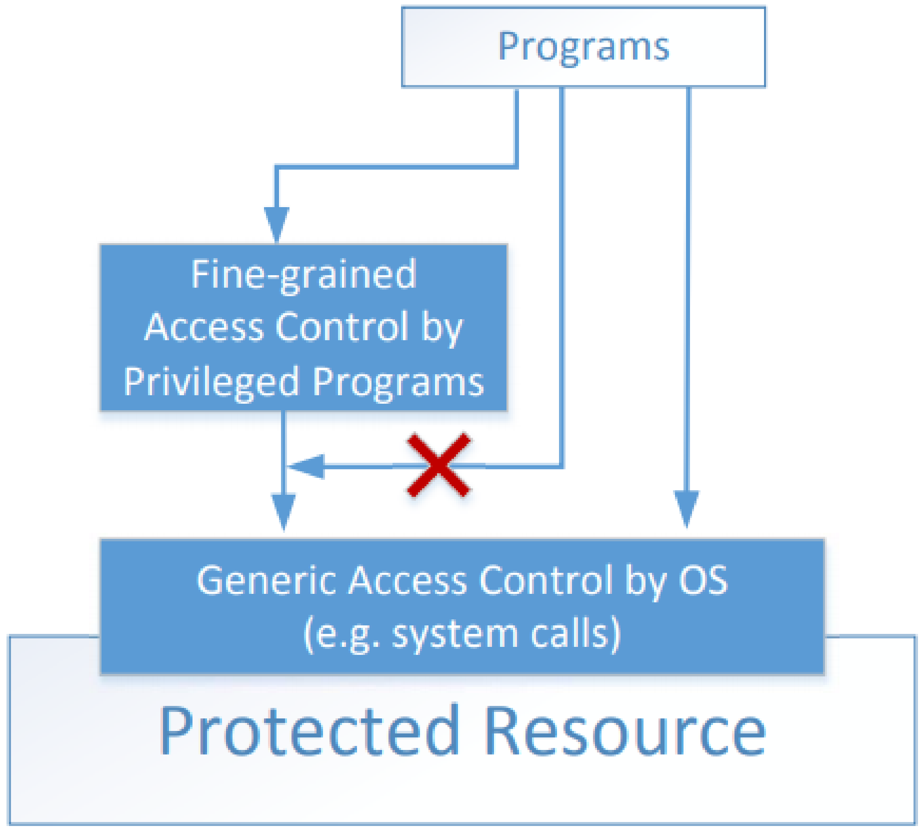
  - How would normal users change their password?



```
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0R25yqtqrSrFeWfCgybQWWnwR4ks/.rjqyM7Xw
h/pDyc5U1BWOzkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::
man:*:15749:0:99999:7:::
lp:*:15749:0:99999:7:::
```

Two-Tier Approach

- Implementing fine-grained access control in operating systems make OS over complicated.
- OS relies on extension to enforce fine grained access control
- Privileged programs are such extensions


Programs
Fine-grained Access Control by Privileged Programs
Generic Access Control by OS (e.g. system calls)
Protected Resource

Types of Privileged Programs

- Daemons
  - Computer program that runs in the background
  - Needs to run as root or other privileged users
- Set-UID Programs
  - Widely used in UNIX systems
  - Program marked with a special bit

Superman Story

- Power Suit
  - Superpeople: Directly give them the power
  - Issues: bad superpeople
- Power Suit 2.0
  - Computer chip
  - Specific task
  - No way to deviate from pre-programmed task
- Set-UID mechanism: A Power Suit mechanism implemented in Linux OS

Set-UID Concept

- **Allow user to run a program with the program owner's privilege.**
- Allow users to run programs with temporary elevated privileges
- Example: the `passwd` program

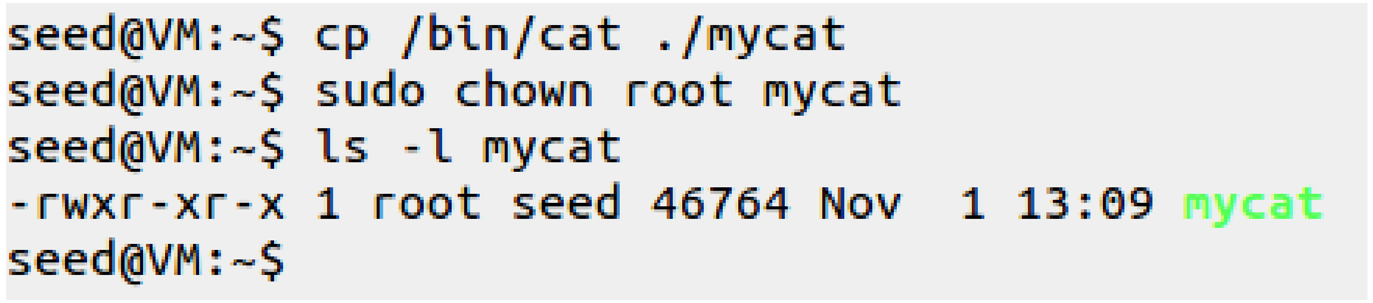
```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Sep 12  2012 /usr/bin/passwd
```

Set-UID Concept

- Every process has three User IDs.
  - **Real UID (RUID)**: Identifies real owner of process
  - **Effective UID (EUID)**: Identifies privilege of a process
    - Access control is based on EUID
  - **Saved UID (SUID)**: Holds an inactive user ID that is recoverable
- Every process also has three Group IDs, identical to those described above but apply to groups instead of users.
- When a normal program is executed, RUID = EUID, they both equal to the ID of the user who runs the program
- When a Set-UID program is executed, RUID ≠ EUID. RUID still equal to the user's ID, but EUID equals to the program **owner**'s ID.
  - If the program is owned by root, the program runs with the root privilege.

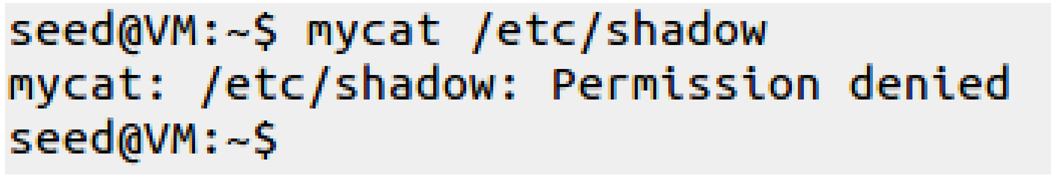
Turn a Program into Set-UID

- Change the owner of a file to root :



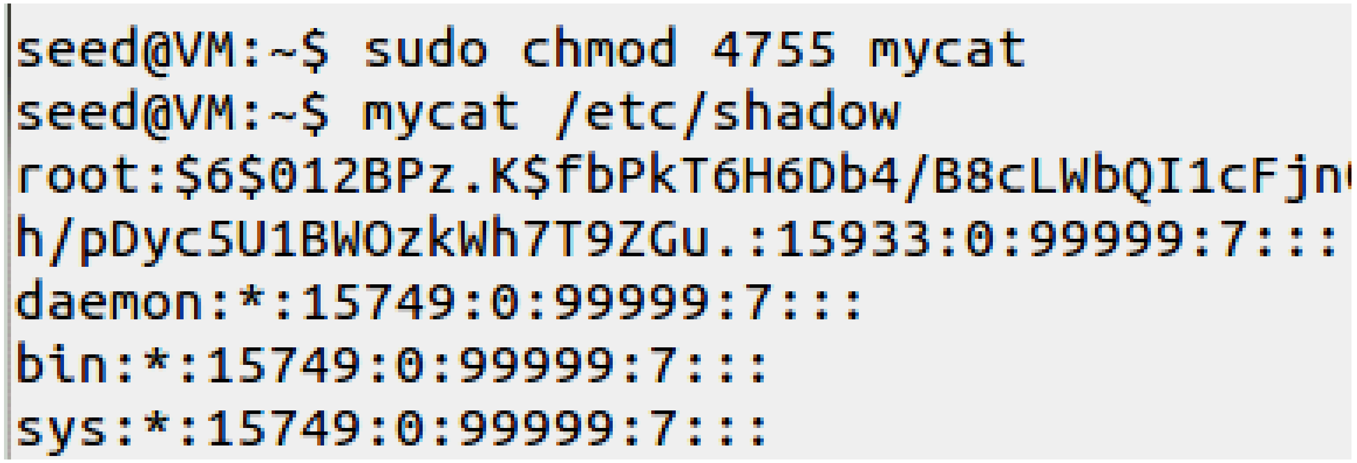
```
seed@VM:~$ cp /bin/cat ./mycat
seed@VM:~$ sudo chown root mycat
seed@VM:~$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Nov  1 13:09 mycat
seed@VM:~$
```

- Before Enabling Set-UID bit:



```
seed@VM:~$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
seed@VM:~$
```

- After Enabling the Set-UID bit :



```
seed@VM:~$ sudo chmod 4755 mycat
seed@VM:~$ mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn
h/pDyc5U1BWOzkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
```

How it Works

A Set-UID program is just like any other program, except that it has a special marking, which a single bit called Set-UID bit

```
$ cp /bin/id ./myid
$ sudo chown root myid
$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

```
$ sudo chmod 4755 myid
$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Example of Set UID

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

Not a privileged program

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

Become a privileged program

```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

It is still a privileged program, but not the root privilege

How is Set-UID Secure?

- Allows normal users to escalate privileges
  - This is different from directly giving the privilege (sudo command)
  - Restricted behavior – similar to superman designed computer chips

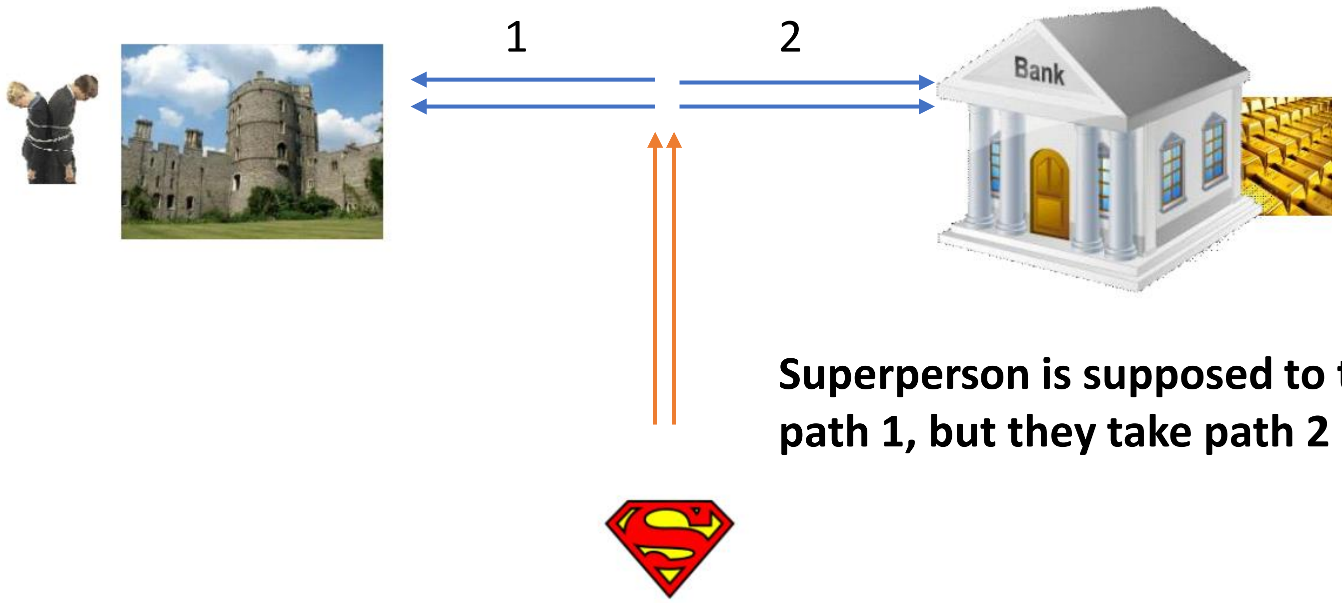
- Unsafe to turn all programs into Set-UID
  - Example: /bin/sh
  - Example: vi

Attack on Superman

- Cannot assume that user can only do whatever is coded
  - Coding flaws by developers

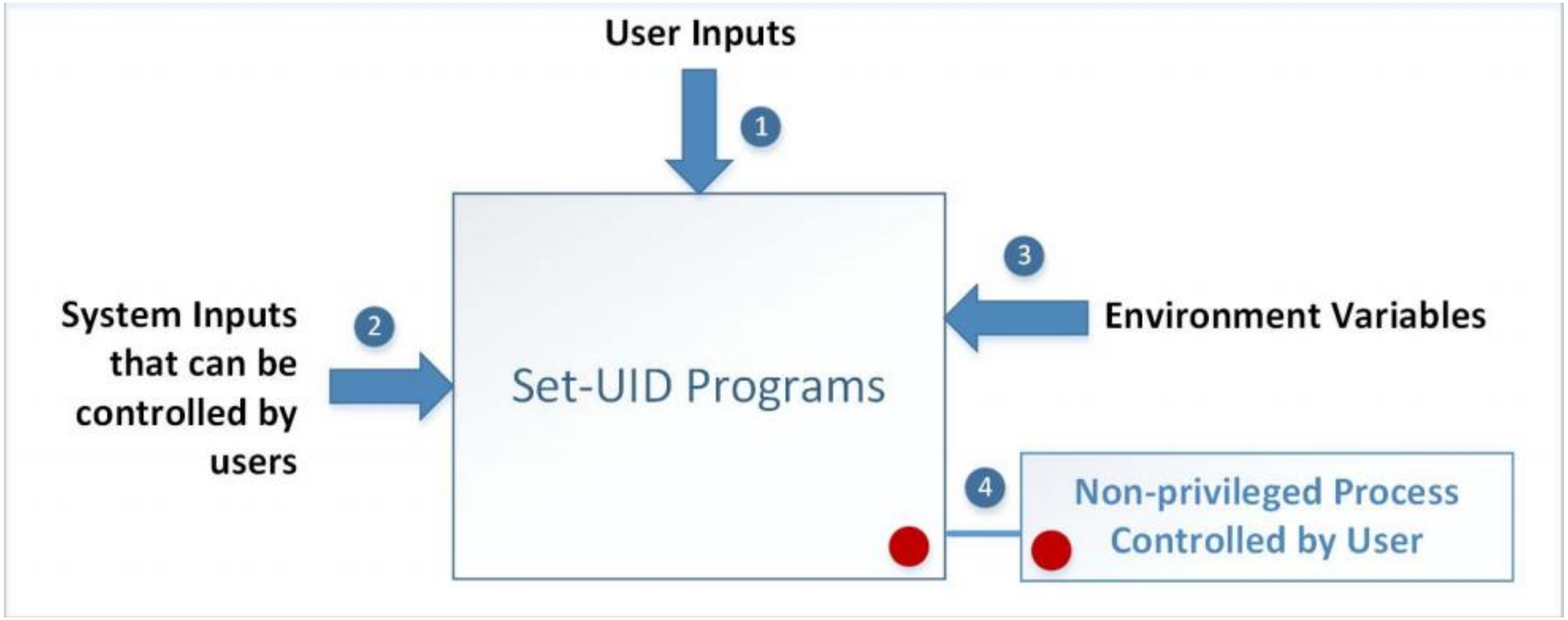
- Superperson Mallory
  - Fly north then turn left
  - How to exploit this code?

- Superperson Malorie
  - Fly North and turn West
  - How to exploit this code?



Superperson is supposed to take path 1, but they take path 2

Attack Surfaces of Set-UID Programs


User Inputs
1
System Inputs that can be controlled by users
2
Set-UID Programs
3
Environment Variables
4
Non-privileged Process Controlled by User

Attacks via User Inputs

User Inputs: Explicit Inputs

- Buffer Overflow – More information in Chapter 4
  - Overflowing a buffer to run malicious code

- Format String Vulnerability – More information in Chapter 6
  - Changing program behavior using user inputs as format strings

Attacks via User Inputs

CHSH – Change Shell

- Set-UID program with ability to change default shell programs
- Shell programs are stored in /etc/passwd file

Issues

- Failing to sanitize user inputs
- Attackers could create a new root account

Attack

```
bob:$6$jUODEFsfwfi3:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

Attacks via System Inputs

System Inputs

- Race Condition – More information in Chapter 7
  - Symbolic link to privileged file from a unprivileged file
  - Influence programs
  - Writing inside world writable folder

Attacks via Environment Variables

- Behavior can be influenced by inputs that are not visible inside a program.
- Environment Variables : These can be set by a user before running a program.
- Detailed discussions on environment variables will be in Chapter 2.

Attacks via Environment Variables

- `PATH` Environment Variable
  - Used by shell programs to locate a command if the user does not provide the full path for the command
  - system(): call /bin/sh first
  - system(“ls”)
    - /bin/sh uses the PATH environment variable to locate “ls”
    - Attacker can manipulate the PATH variable and control how the “ls” command is found
- More examples on this type of attacks can be found in Chapter 2

Capability Leaking

- In some cases, Privileged programs downgrade themselves during execution
- Example: The `su` program
    - This is a privileged Set-UID program
    - Allows one user to switch to another user (say user1 to user2)
    - Program starts with EUID as root and RUID as user1
    - After password verification, both EUID and RUID become user2’s (via privilege downgrading)
- Such programs may lead to capability leaking
    - Programs may not clean up privileged capabilities before downgrading

Attacks via Capability Leaking: An Example

The /etc/zzz file is only writable by root

File descriptor is created (the program is a root-owned Set-UID program)

The privilege is downgraded

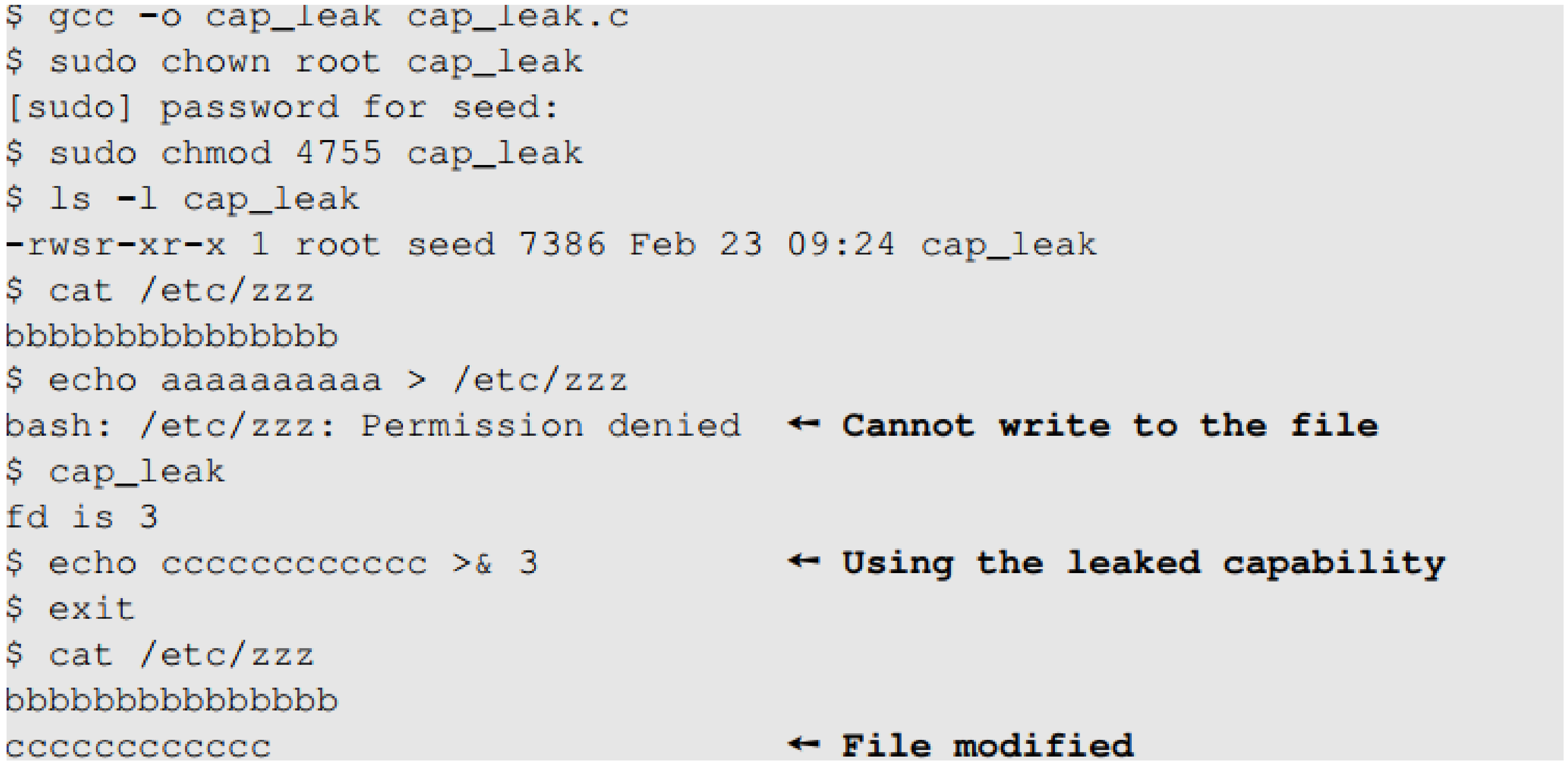
Invoke a shell program, so the behavior restriction on the program is lifted

```
fd = open("/etc/zzz", O_RDWR | O_APPEND);
if (fd == -1) {
   printf("Cannot open /etc/zzz\n");
   exit(0);
}

// Print out the file descriptor value
printf("fd is %d\n", fd);

// Permanently disable the privilege by making the
// effective uid the same as the real uid
setuid(getuid());

// Execute /bin/sh
v[0] = "/bin/sh"; v[1] = 0;
execve(v[0], v, 0);
```

Attacks via Capability Leaking (Continued)

The program forgets to close the file, so the file descriptor is still valid.

Capability Leak



```
$ gcc -o cap_leak cap_leak.c
$ sudo chown root cap_leak
[sudo] password for seed:
$ sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root seed 7386 Feb 23 09:24 cap_leak
$ cat /etc/zzz
bbbbbbbbbbbbbbb
$ echo aaaaaaaaaa > /etc/zzz
bash: /etc/zzz: Permission denied   ← Cannot write to the file
$ cap_leak
fd is 3
$ echo cccccccccccc >& 3            ← Using the leaked capability
$ exit
$ cat /etc/zzz
bbbbbbbbbbbbbbb
cccccccccccc                        ← File modified
```

How to fix the program?

Destroy the file descriptor before downgrading the privilege (close the file)

Capability Leaking in OS X – Case Study

- OS X Yosemite found vulnerable to privilege escalation attack related to capability leaking in July 2015 (OS X 10.10)
- Added features to dynamic linker `dyld`
  - DYLD_PRINT_TO_FILE environment variable
- The dynamic linker can open any file, so for root-owned Set-UID programs, it runs with root privileges. The dynamic linker `dyld`, does not close the file. There is a capability leaking.
- **Scenario 1 (safe)**: Set-UID finished its job and the process dies. Everything is cleaned up and is safe.
- **Scenario 2 (unsafe):** Similar to the "`su`" program, the privileged program downgrades its privilege, and lifts the restriction.

Invoking Programs

- Invoking external commands from inside a program
- External command is chosen by the Set-UID program
  - Users are not supposed to provide the command (or it is not secure)
- Attack:
  - Users are often asked to provide input data to the command.
  - If the command is not invoked properly, user’s input data may be turned into command name. This is dangerous.

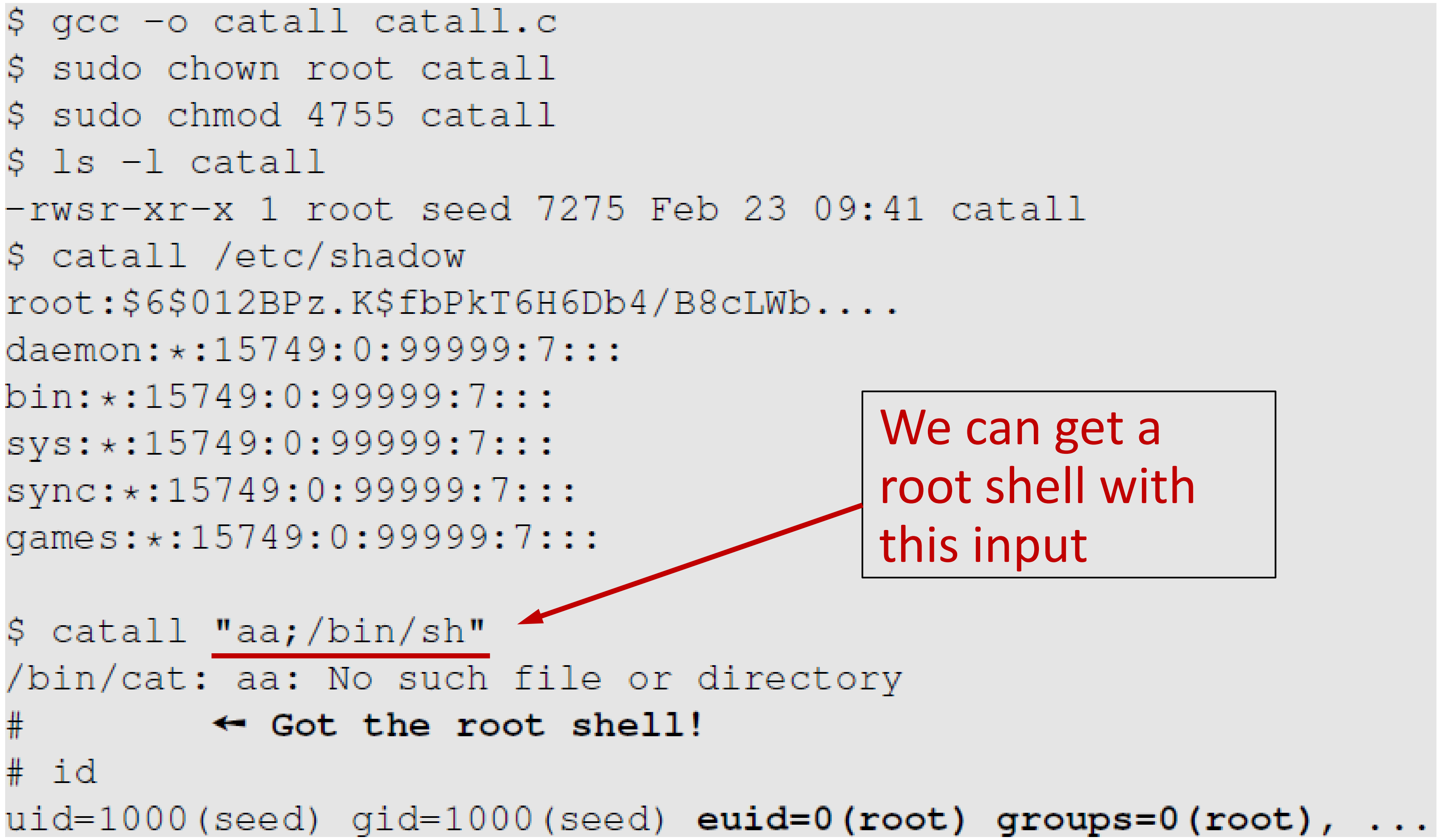
Invoking Programs : Unsafe Approach

```
int main(int argc, char *argv[])
{
  char *cat="/bin/cat";

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
  sprintf(command, "%s %s", cat, argv[1]);
  system(command);
  return 0 ;
}
```

- The easiest way to invoke an external command is the system() function.
- This program is supposed to run the `/bin/cat` program.
- It is a root-owned Set-UID program, so the program can view all files, but it can't write to any file.

Question: Can you use this program to run other commands, with the root privilege?

Invoking Programs : Unsafe Approach (Continued)



```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#         ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

We can get a root shell with this input

Problem: Some part of the data becomes code (command name)

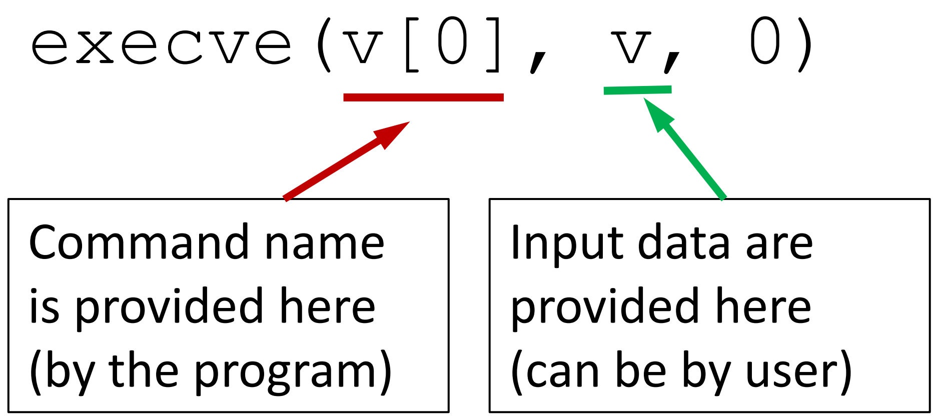
A Note

- In Ubuntu 16.04, /bin/sh points to /bin/dash, which has a countermeasure
  - It drops privilege when it is executed inside a set-uid process
- Therefore, we will only get a normal shell in the attack on the previous slide
- Do the following to remove the countermeasure

```
Before experiment: link /bin/sh to /bin/zsh
$ sudo ln -sf /bin/zsh /bin/sh

After experiment: remember to change it back
$ sudo ln -sf /bin/dash /bin/sh
```

Invoking Programs Safely: using **execve()**

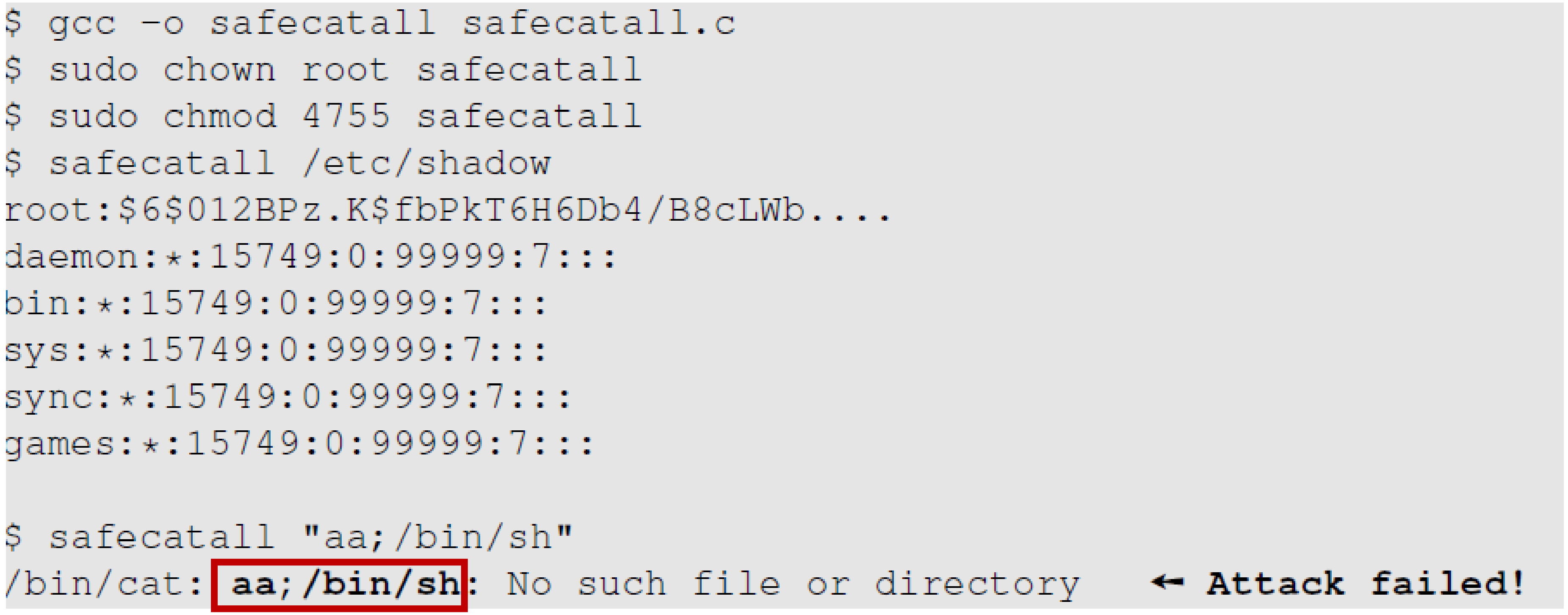
```
int main(int argc, char *argv[])
{
  char *v[3];

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
  execve(v[0], v, 0);

  return 0 ;
}
```


execve(v[0], v, 0)
Command name is provided here (by the program)
Input data are provided here (can be by user)

Why is it safe?

Code (command name) and data are clearly separated; there is no way for the user data to become code

Invoking Programs Safely (Continued)



```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ safecatall "aa;/bin/sh"
/bin/cat: aa;/bin/sh: No such file or directory   ← Attack failed!
```

The data are still treated as data, not code

Additional Consideration

- Some functions in the exec() family behave similarly to execve(), but may not be safe
  - execlp(), execvp() and execvpe() duplicate the actions of the shell. These functions can be attacked using the PATH Environment Variable

Invoking External Commands in Other Languages

- Risk of invoking external commands is not limited to C programs
- We should avoid problems similar to those caused by the system() functions
- Examples:
  - Perl: open() function can run commands, but it does so through a shell
  - PHP: system() function

```
<?php
  print("Please specify the path of the directory");
  print("<p>");
  $dir=$_GET['dir'];
  print("Directory path: " . $dir . "<p>");
  system("/bin/ls $dir");
?>
```

  - Attack:
    - `http://localhost/list.php?dir=.;date`
    - Command executed on server : “`/bin/ls .;date`”

Principle of Isolation

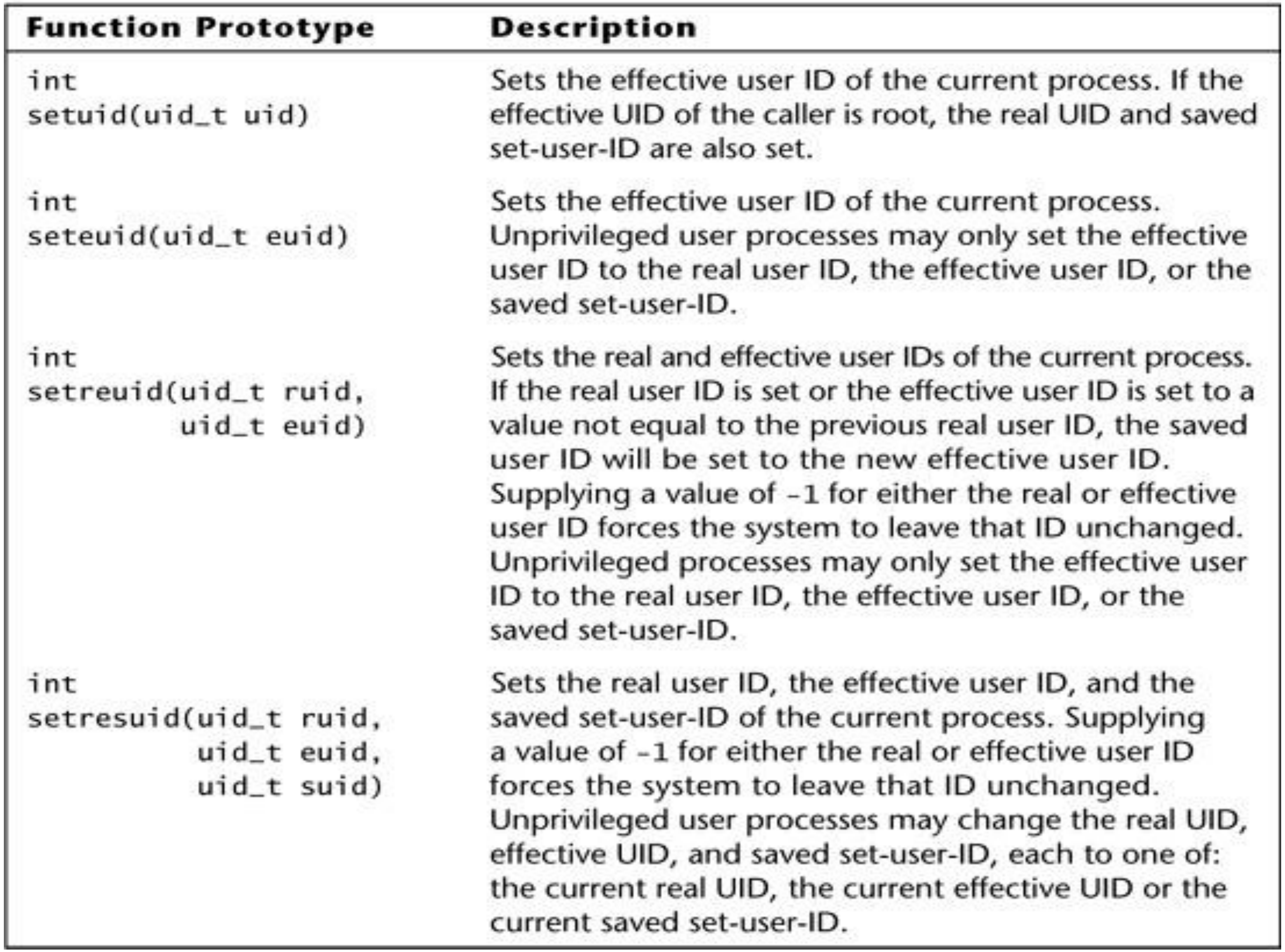
Principle: Don’t mix code and data.

Attacks due to violation of this principle :

- system() code execution (Command Injection)
- Cross Site Scripting – More Information in Chapter 10
- SQL injection - More Information in Chapter 11
- Buffer Overflow attacks - More Information in Chapter 4

Principle of Least Privilege

- A privileged program should be given the power which is required to perform its tasks.
- More privileges → the greater the potential damage
  - Programs should not require their users to have extraordinary privileges to perform ordinary tasks.
  - Privileged programs should minimize the amount of damage they can cause when something goes wrong.
- Disable the privileges (temporarily or permanently) when a privileged program doesn’t need those.
- In Linux, seteuid() and setuid() can be used to disable/discard privileges.
- Different OSes have different ways to do that.



Function Prototype	Description
`int setuid(uid_t uid)`	Sets the effective user ID of the current process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set.
`int seteuid(uid_t euid)`	Sets the effective user ID of the current process. Unprivileged user processes may only set the effective user ID to the real user ID, the effective user ID, or the saved set-user-ID.
`int setreuid(uid_t ruid, uid_t euid)`	Sets the real and effective user IDs of the current process. If the real user ID is set or the effective user ID is set to a value not equal to the previous real user ID, the saved user ID will be set to the new effective user ID. Supplying a value of `-1` for either the real or effective user ID forces the system to leave that ID unchanged. Unprivileged processes may only set the effective user ID to the real user ID, the effective user ID, or the saved set-user-ID.
`int setresuid(uid_t ruid, uid_t euid, uid_t suid)`	Sets the real user ID, the effective user ID, and the saved set-user-ID of the current process. Supplying a value of `-1` for either the real or effective user ID forces the system to leave that ID unchanged. Unprivileged user processes may change the real UID, effective UID, and saved set-user-ID, each to one of: the current real UID, the current effective UID or the current saved set-user-ID.

Drop/Restore Privilege Example

```
int main(int argc, *char[] argv) {
  uid_t caller_uid = getuid();
  uid_t owner_uid = geteuid();

  /* Drop privileges right up front, but we'll need them back
     in a little bit, so use effective id */
  if (setresuid(-1, caller_uid, owner_uid) != 0) {
    exit(-1);
  }

  /* Privileges not necessary or desirable at this point */
  processCommandLine(argc, argv);

  /* Regain privileges */
  if (setresuid(-1, owner_uid, caller_uid) != 0) {
    exit(-1);
  }
  openSocket(88); /* requires root */

  /* Drop privileges for good */
  if (setresuid(caller_uid, caller_uid, caller_uid) != 0) {
    exit(-1);
  }

  doWork();
}
```

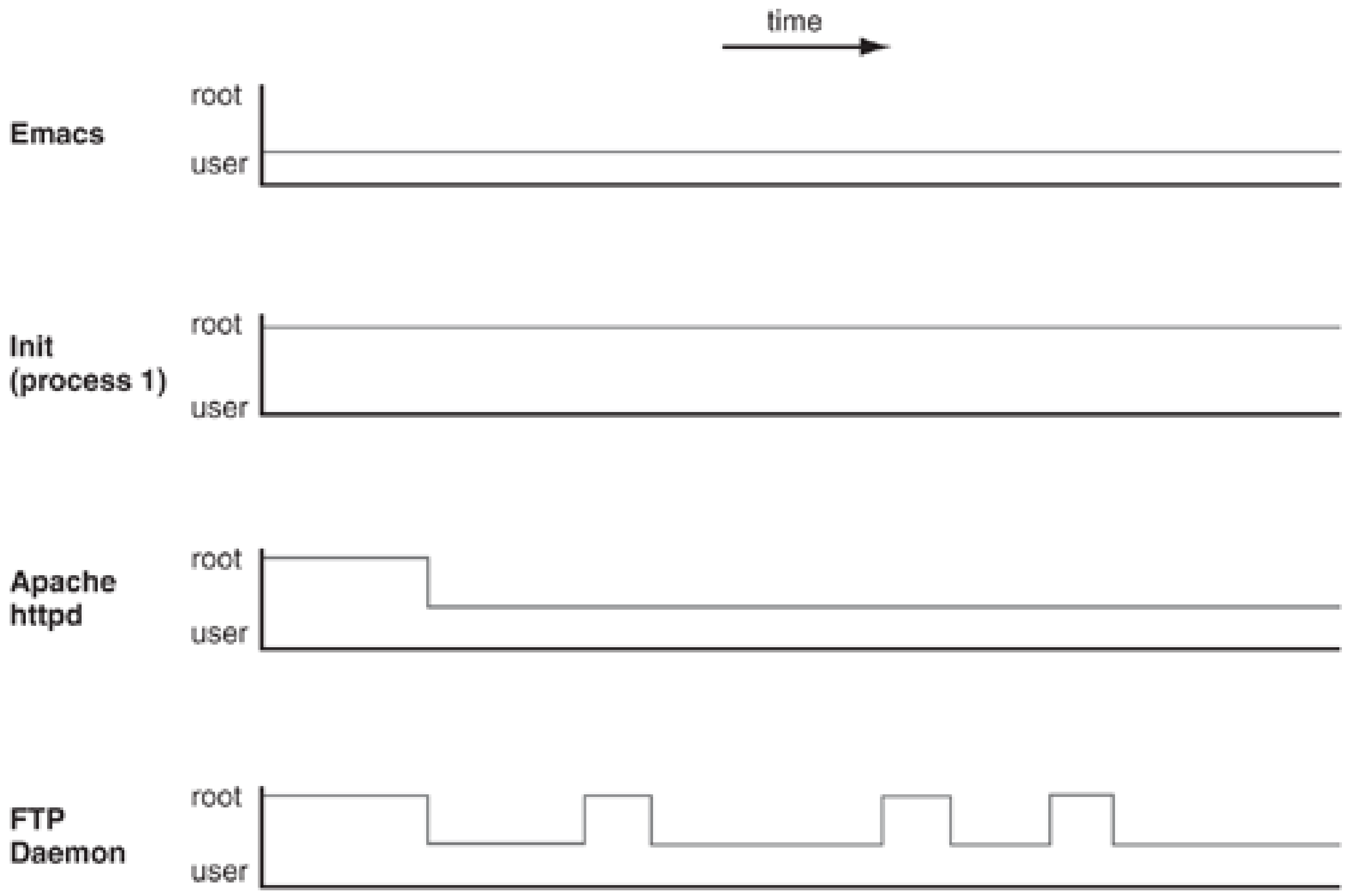
Privilege Management Considerations

- seteuid(), setuid(), and setreuid()
  - loosely defined by the POSIX standard
  - left up to individual implementations
  - leads to common errors and misuses
- Be careful of cross platform inconsistencies
- Check for Every Error Condition
  - If a function returns an error code or any other evidence of its success or failure, always check for error conditions, even if there is no obvious way for the error to occur.
  - Always pay attention to the return value of a privilege management call
  - If a privilege management function fails unexpectedly, ensure that the program's behavior remains secure
- Disable Signals Before Acquiring Privileges
  - Disable signals before elevating privileges to avoid having signal handling code run with privileges.
  - Re-enable signals after dropping back to standard user privileges.
  - Signal handlers and spawned processes run with the privileges of the owning process
- Value Security over Robustness: Die on Errors
  - Don't attempt to recover from unexpected or poorly understood errors.

Privilege Transitions

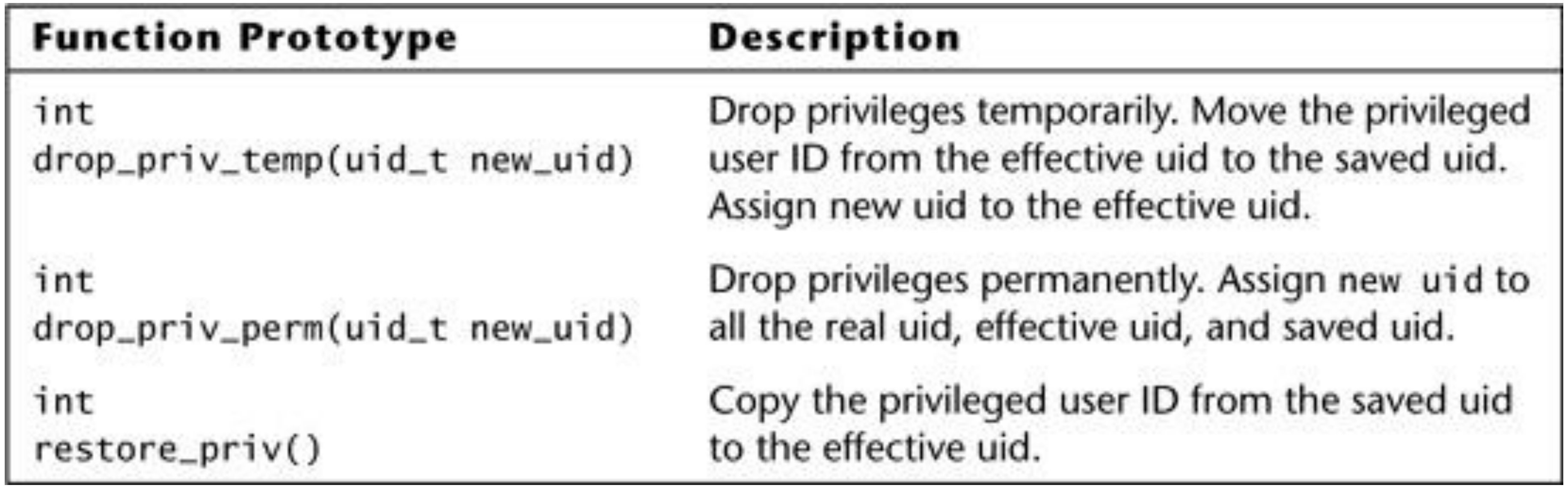
- Transitions between privileged and unprivileged states define a program's *privilege profile*.
    - Normal programs that run with the same privileges as their users.
        - Example: Emacs.
    - System programs that run with root privileges for the duration of their execution.
        - Example: Init (process 1).
    - Programs that need root privileges to use a fixed set of system resources when they are first executed.
        - Example: Apache httpd, which needs root access to bind to low-numbered ports.
    - Programs that require root privileges intermittently throughout their execution.
        - Example: An FTP daemon, which binds to low-numbered ports intermittently throughout execution

Least Privilege Requires Context



Using Wrappers

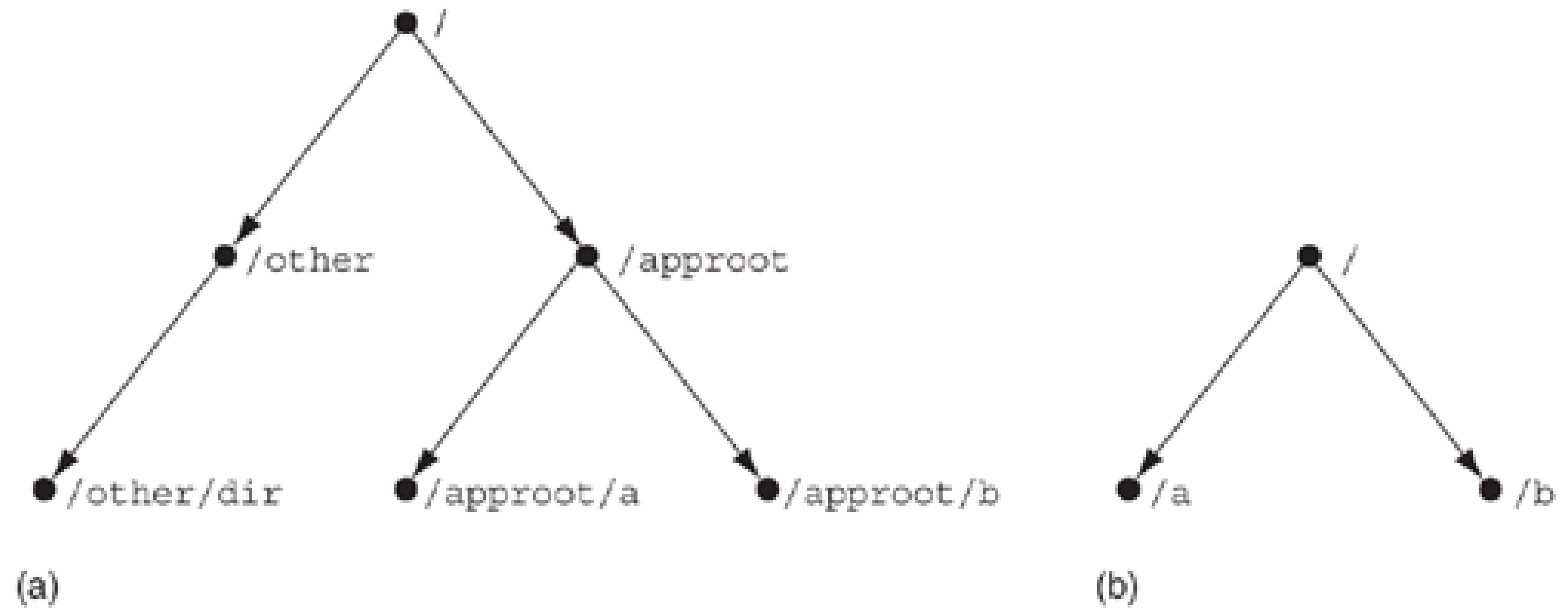
- A reference implementation of this interface can be found in Setuid Demystified [Chen, 2002].
  - Drop privileges with the intention of regaining it
  - Drop privileges permanently
  - Regain previously stored privileges



Function Prototype	Description
`int drop_priv_temp(uid_t new_uid)`	Drop privileges temporarily. Move the privileged user ID from the effective uid to the saved uid. Assign new uid to the effective uid.
`int drop_priv_perm(uid_t new_uid)`	Drop privileges permanently. Assign new `uid` to all the real uid, effective uid, and saved uid.
`int restore_priv()`	Copy the privileged user ID from the saved uid to the effective uid.

Restrict Privilege on the Filesystem

- Restrict your program's privileges by limiting its view of the filesystem using the chroot()
  - invoking chroot(), a process cannot access any files outside the specified directory tree - called a chroot jail



Four Common Mistakes with chroot()

1. Calling chroot() does not affect any file descriptors that are currently open
   - close every open file handle before calling chroot().
2. The chroot() function call does not change the process’s current working directory,
   - Relative paths such as ../../../../../etc/passwd can still refer to filesystem resources outside the chroot jail
   - Always follow a call to chroot() with the call chdir("/")
   - Verify that any error handling code between the chroot() call and the chdir() call does not open any files and does not return control to any part of the program other than shutdown routines.

Four Common Mistakes with chroot()

3. A call to chroot() can fail. Check the return value from chroot() to make sure the call succeeded.
4. To call chroot(), the program must be running with root privileges. As soon as the privileged operation has completed, the program should drop root privileges and return to the privileges of the invoking user.

Privileges on Microsoft Windows (1/2)

- Historically, applications were poorly written and wouldn’t behave correctly without full administrator privileges.
- Microsoft now defaults users (as of Windows Vista) to a general user (non-elevated) privilege level.
- User Account Control (UAC), added in Windows Vista, helps allow users to run applications with reduced privileges and elevate only when necessary.
  - When an application needs elevated privileges, the UAC dialog would pop up, prompting users to allow the application to perform the requested action.
  - Application developers have been shifting to writing their applications to run correctly with reduced privileges.

Privileges on Microsoft Windows (2/2)

- Windows operating system uses access tokens to control privileges on executing processes.
- API functions are used to change privileges assigned to an access token (AdjustTokenPrivileges).
  - Must have the appropriate privileges to update an access token.
- Windows server processes will use impersonation tokens on behalf of the client in access control decisions.
- Resources
  - https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights
  - https://docs.microsoft.com/en-us/windows/win32/secbp/running-with-special-privileges
  - https://docs.microsoft.com/en-us/windows/win32/secauthz/enabling-and-disabling-privileges-in-c--
  - https://docs.microsoft.com/en-us/windows/win32/secauthz/client-impersonation
  - https://docs.microsoft.com/en-us/windows/win32/secbp/running-with-administrator-privileges?redirectedfrom=MSDN

Summary

- The need for privileged programs
- How the Set-UID mechanism works
- Security flaws in privileged Set-UID programs
- Attack surface
- How to improve the security of privileged programs