CS 366
Intro to Cybersecurity

Dr. Stella Sun

EECS

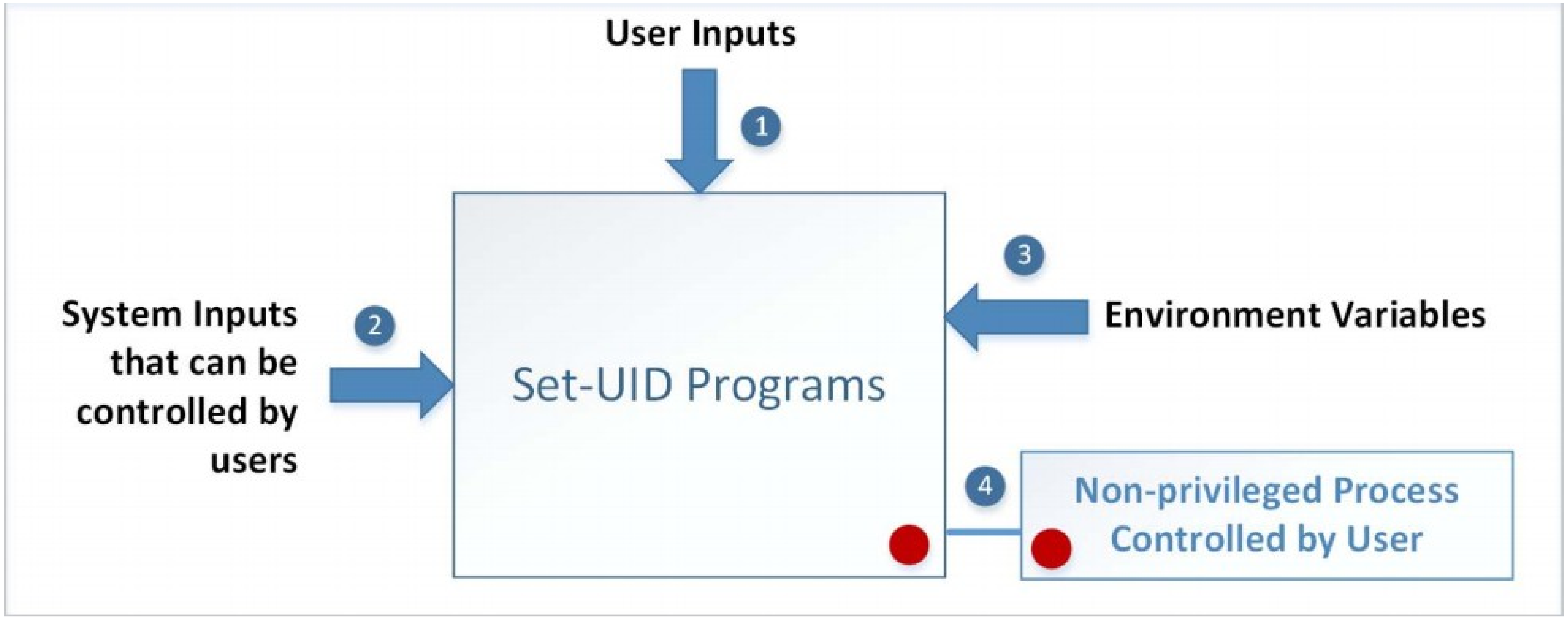
University of Tennessee

Fall 2022

Today’s Class

- ➢ Operating system security
- ➢ SetUID attack surface and countermeasures

Attack Surface 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

- Stack smashing - overflowing a buffer to run malicious code
- Format string vulnerability - 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
    - Symbolic link to privileged file from a unprivileged file
    - Influence programs
    - Writing inside world writable folder

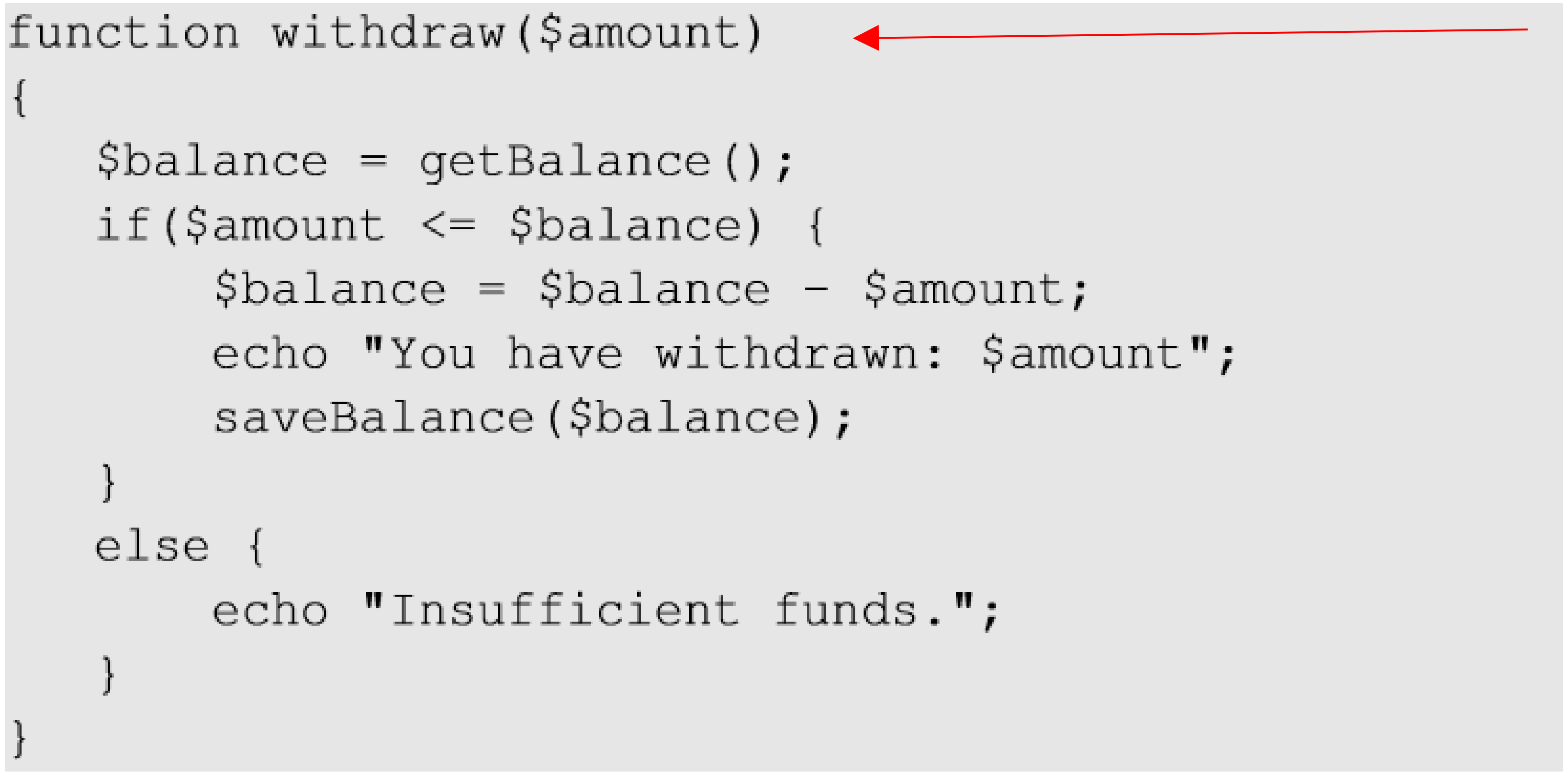
Race Condition

- Happens when:
  - Multiple processes access and manipulate the same data concurrently.
  - The outcome of execution depends on a particular order.

- If a privileged program has a race condition, the attackers may be able to affect the output of the privileged program by putting influences on the uncontrollable events.

Race Condition Problem

When two concurrent threads of execution access a shared resource in a way that unintentionally produces different results depending on the timing of the threads or processes.



```
function withdraw($amount)
{
   $balance = getBalance();
   if($amount <= $balance) {
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
       saveBalance($balance);
   }
   else {
       echo "Insufficient funds.";
   }
}
```

Race Condition can occur here if there are two simultaneous withdraw requests.

A Special Type of Race Condition

- Time-Of-Check To Time-Of-Use (TOCTTOU)
- Occurs when checking for a condition before using a resource

Race Condition Vulnerability

```
if (!access("/tmp/X", W_OK)) {
    /* the real user has the write permission*/
    f = open("/tmp/X", O_WRITE);
    write_to_file(f);
}
else {
   /* the real user does not have the write permission */
   fprintf(stderr, "Permission denied\n");
}
```

- Root-owned Set-UID program.
- Effective UID : root
- Real UID : seed

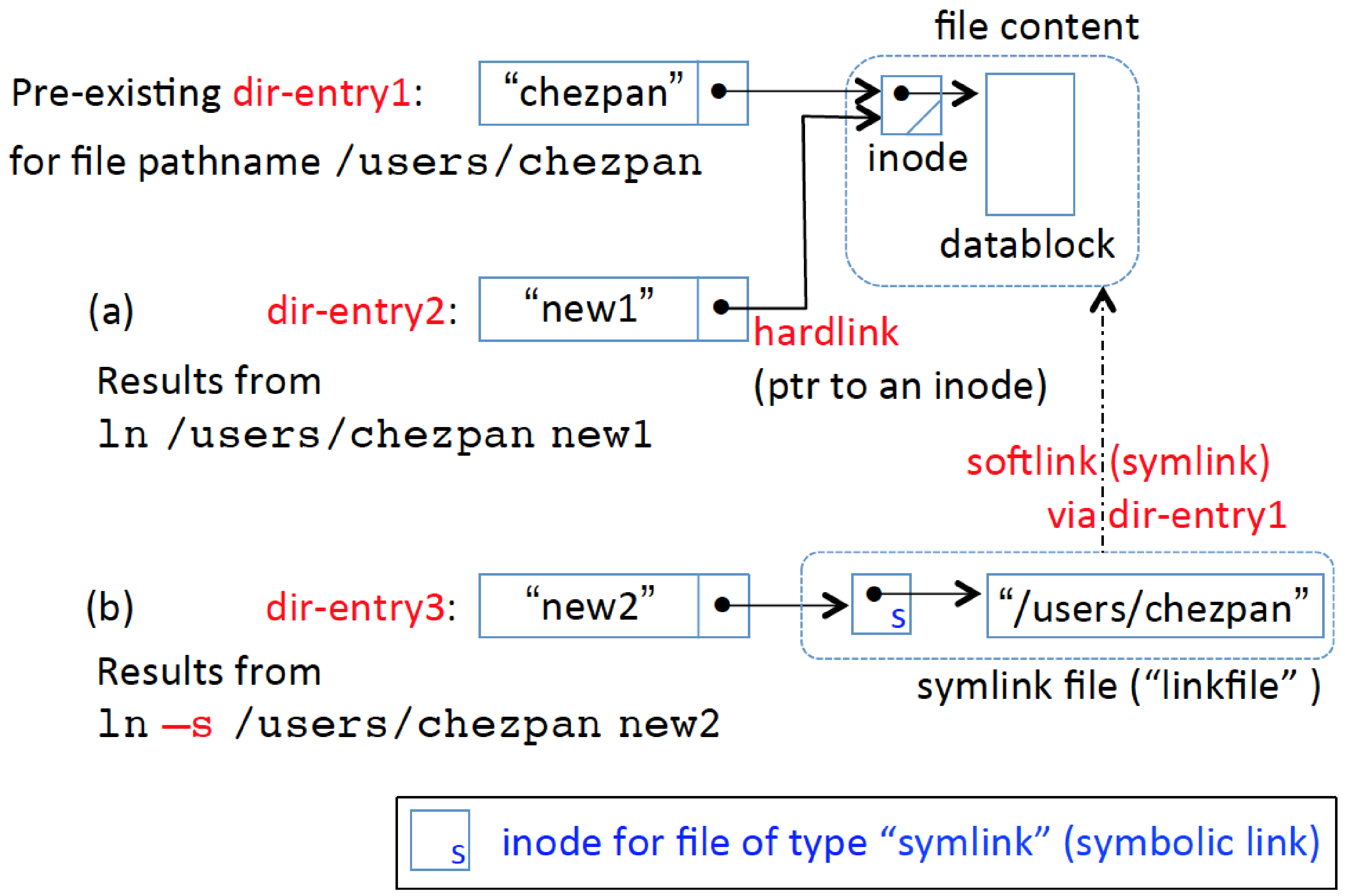
- The above program writes to a file in the `/tmp` directory (world-writable)
- As the root can write to any file, the program ensures that the real user has permissions to write to the target file using `access()`.
- `access()` system call checks if the Real User ID has write access to /tmp/X.
- After the check, the file is opened for writing.
- `open()` checks the effective user id which is 0 and hence file will always be opened.

Race Condition Vulnerability

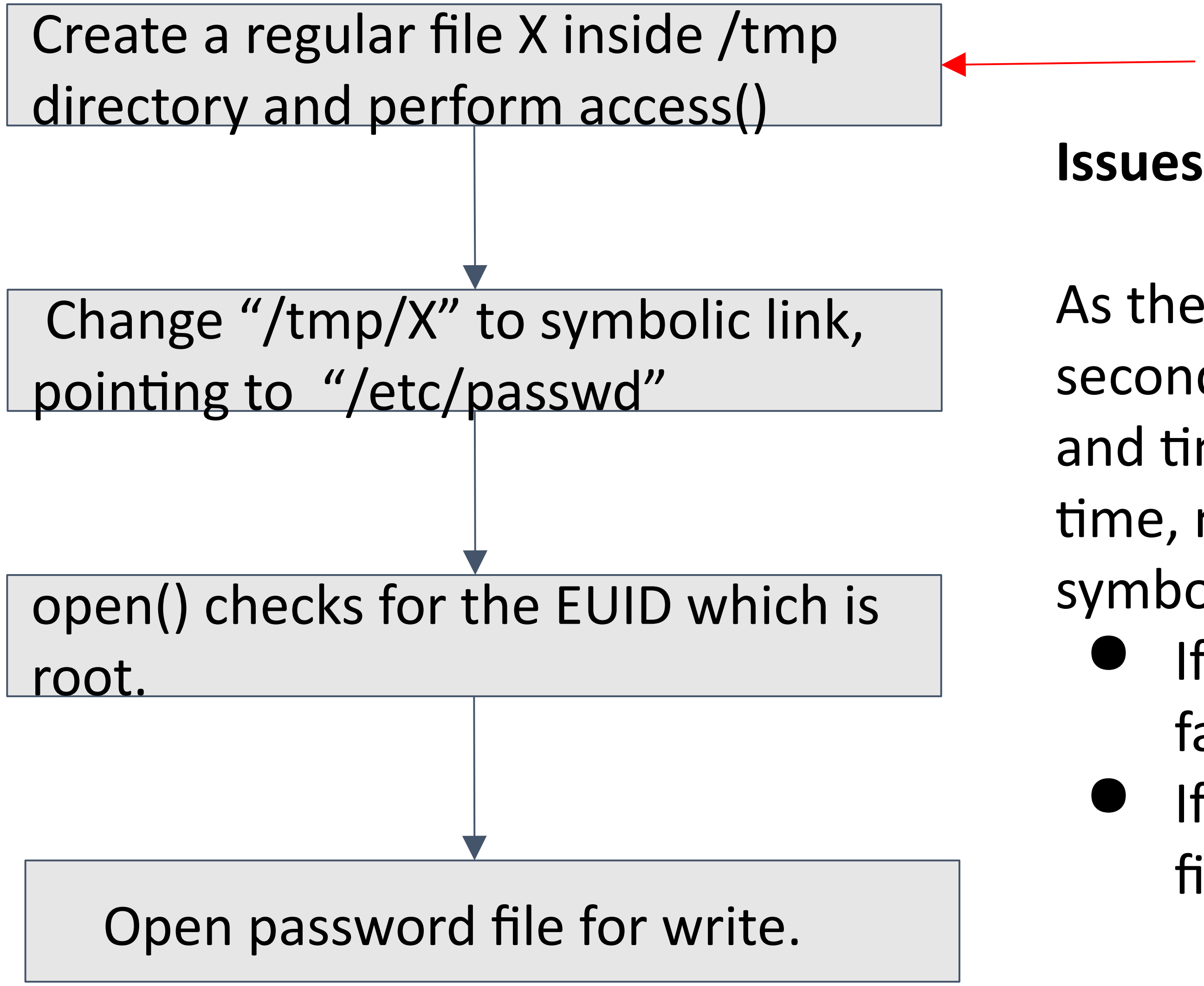
Goal : To write to a protected file like `/etc/passwd`.
To achieve this goal we need to make `/etc/passwd` as our target file without changing the file name in the program.

- Symbolic link (soft link) helps us to achieve it.
- It is a special kind of file that points to another file.

Symbolic Link



Race Condition Vulnerability



Create a regular file X inside /tmp directory and perform access()

Change "/tmp/X" to symbolic link, pointing to "/etc/passwd"

open() checks for the EUID which is root.

Open password file for write.

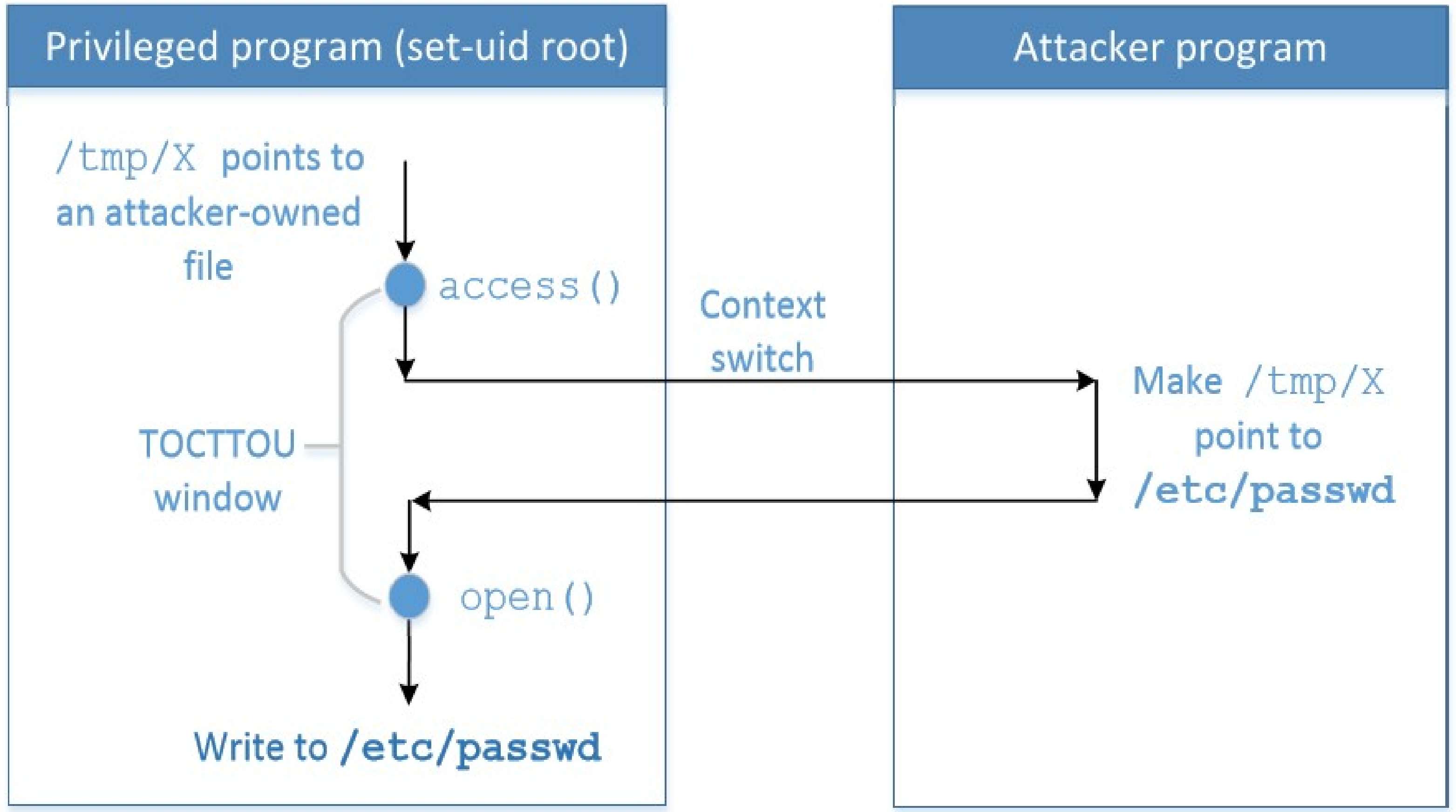
Pass the access() check

Issues :

As the program runs billions of instructions per second, the window between the time to check and time to use lasts for a very short period of time, making it impossible to change to a symbolic link

- If the change is too early, `access()` will fail.
- If the change is little late, the program will finish using the file.

Race Condition Vulnerability


Privileged program (set-uid root)
Attacker program
/tmp/X points to an attacker-owned file
access()
Context switch
Make /tmp/X point to /etc/passwd
TOCTTOU window
open()
Write to /etc/passwd

To win the race condition (TOCTTOU window), we need two processes :

- Run vulnerable program in a loop
- Run the attack program in a loop

Understanding the attack

Let’s consider steps for two programs :
A1 : Make “/tmp/X” point to a file owned by us
A2 : Make “/tmp/X” point to /etc/passwd
V1 : Check user’s permission on “/tmp/X”
V2 : Open the file

Attack program runs: A1,A2,A1,A2…….

Vulnerable program runs : V1,V2,V1,V2…..

As the programs are running simultaneously on a multi-core machine, the instructions will be interleaved (mixture of two sequences)

A1, V1 , A2, V2 : vulnerable prog opens /etc/passwd for editing.

Another Race Condition Example

```
file = "/tmp/X";
fileExist = check_file_existence(file);

if (fileExist == FALSE){
  // The file does not exist, create it.
  f = open(file, O_CREAT);

  // write to file
```

Set-UID program that runs with root privilege.

1. Checks if the file “/tmp/X” exists.
2. If not, open() system call is invoked. If the file doesn’t exist, new file is created with the provided name.
3. There is a window between the check and use (opening the file).
4. If the file already exists, the open() system call will not fail. It will open the file for writing.
5. So, we can use this window between the check and use and point the file to an existing file “/etc/passwd” and eventually write into it.

Countermeasures

- Atomic Operations: To eliminate the window between check and use
- Repeating Check and Use: To make it difficult to win the “race”.
- Sticky Symlink Protection: To prevent doing anything after race is won.
- Principles of Least Privilege: To prevent damages even if attacker can do something after race is won.

Atomic Operations

f = open(file, O_CREAT | O_EXCL)

- These two options combined together will not open the specified file if the file already exists.
- Guarantees the atomicity of the check and the use.

f = open(file ,O_WRITE | **O_REAL_USER_ID**

- This is just an idea, not implemented in the real system.
- With this option, open() will only check the real User ID
- Therefore, open() achieves check and use on it’s own and the operations are atomic.

Repeating Check and Use

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
   struct stat stat1, stat2, stat3;
   int fd1, fd2, fd3;

   if (access("tmp/XYZ", O_RDWR)) {
      fprintf(stderr, "Permission denied\n");     1
      return -1;
   }                                           ← Window 1
```

- Check-and-use is done three times.
- Check if the inodes are same.
- For a successful attack, “/tmp/XYZ” needs to be changed 5 times.
- The chance of winning the race 5 times is much lower than a code with one race condition.

```
else fd1 = open("/tmp/XYZ", O_RDWR);
                                      ← Window 2
if (access("tmp/XYZ", O_RDWR)) {
   fprintf(stderr, "Permission denied\n");       2
   return -1;
}                                     ← Window 3
else fd2 = open("/tmp/XYZ", O_RDWR);
                                      ← Window 4
if (access("tmp/XYZ", O_RDWR)) {
   fprintf(stderr, "Permission denied\n");       3
   return -1;
}                                     ← Window 5
else fd3 = open("/tmp/XYZ", O_RDWR);

// Check whether fd1, fd2, and fd3 has the same inode.
fstat(fd1, &stat1);
fstat(fd2, &stat2);
fstat(fd3, &stat3);

if(stat1.st_ino == stat2.st_ino && stat2.st_ino == stat3.st_ino) {
   // All 3 inodes are the same.
   write_to_file(fd1);
}
else {
```

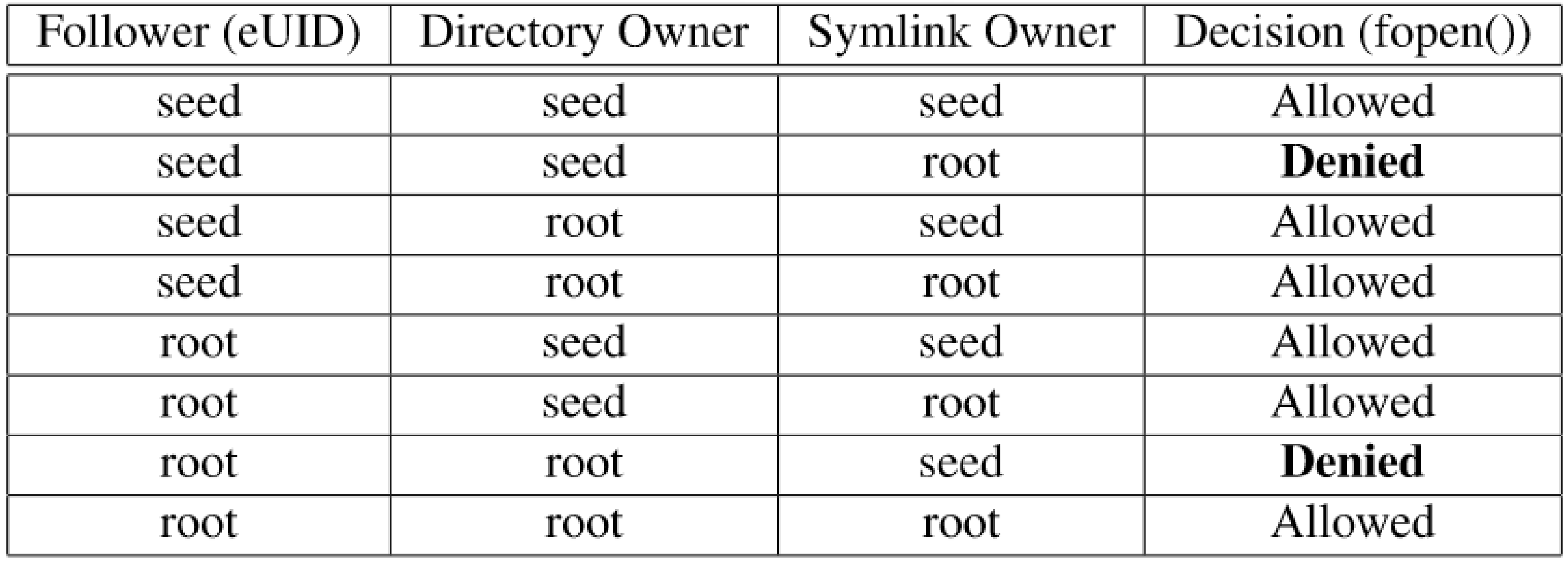
Sticky Symlink Protection

To enable the sticky symlink protection for world-writable sticky directories:

```
// On Ubuntu 12.04, use the following:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1

// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=1
```

- When the sticky symlink protection is enabled, symbolic links inside a sticky world-writable can only be followed when the owner of the symlink matches either the follower or the directory owner.

Sticky Symlink Protection



Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	**Denied**
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	**Denied**
root	root	root	Allowed

- Symlink protection allows fopen() when the owner of the symlink match either the follower (EUID of the process) or the directory owner.

- In our vulnerable program (EUID is root), /tmp directory is also owned by the root, the program will not allowed to follow the symbolic link unless the link is created by the root.

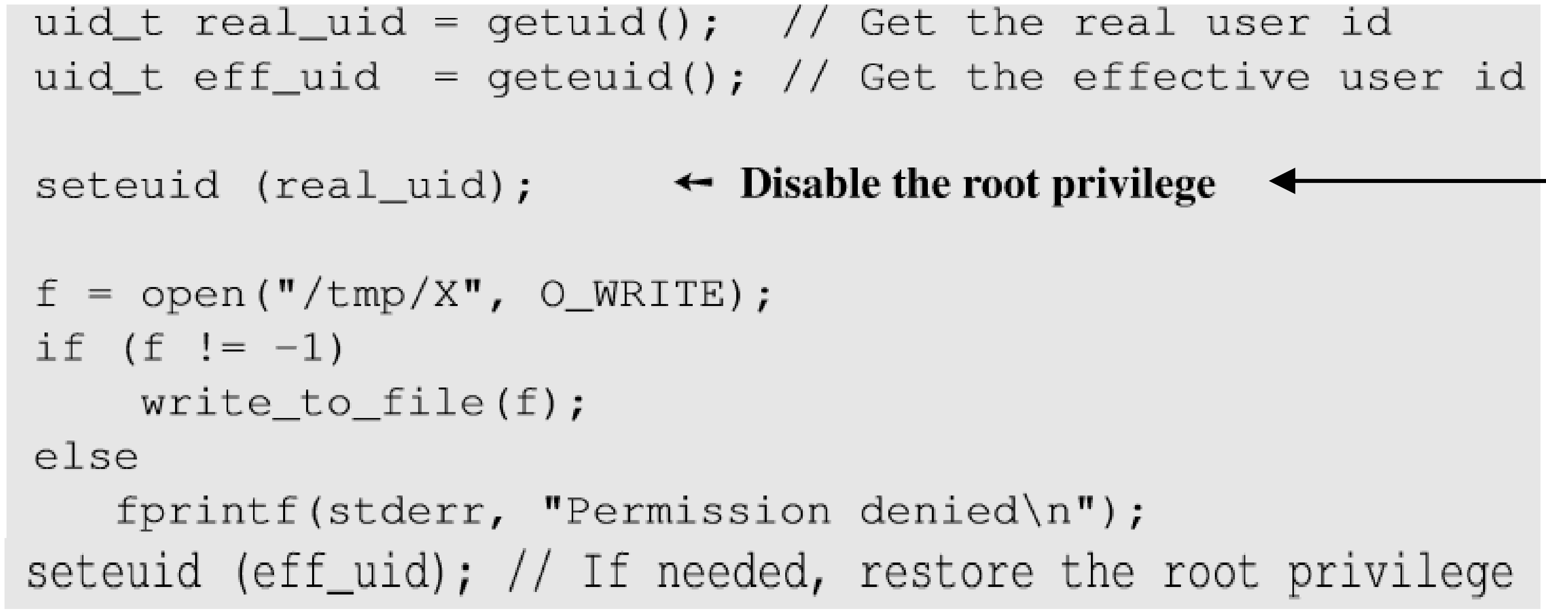
Principle of Least Privilege

Principle of Least Privilege:

A program should not use more privilege than what is needed by the task.

- Our vulnerable program has more privileges than required while opening the file.
- seteuid() and setuid() can be used to discard or temporarily disable privileges.

Principle of Least Privilege



```
uid_t real_uid = getuid();  // Get the real user id
uid_t eff_uid  = geteuid(); // Get the effective user id

seteuid (real_uid);       ← Disable the root privilege

f = open("/tmp/X", O_WRITE);
if (f != -1)
    write_to_file(f);
else
   fprintf(stderr, "Permission denied\n");
seteuid (eff_uid); // If needed, restore the root privilege
```

Right before opening the file, the program should drop its privilege by setting EUID = RUID

After writing, privileges are restored by setting EUID = root

Principle of Least Privilege

- A privileged program should be given the power which is required to perform it’s tasks.
- Disable the privileges (temporarily or permanently) when a privileged program doesn’t need those.
- In Unix, seteuid() and setuid() can be used to disable/discard privileges.
- Different OSes have different ways to do that

  - Linux POSIX

  - Android

Attacks via Environment Variables

Environment variables: hidden input

- 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.

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

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

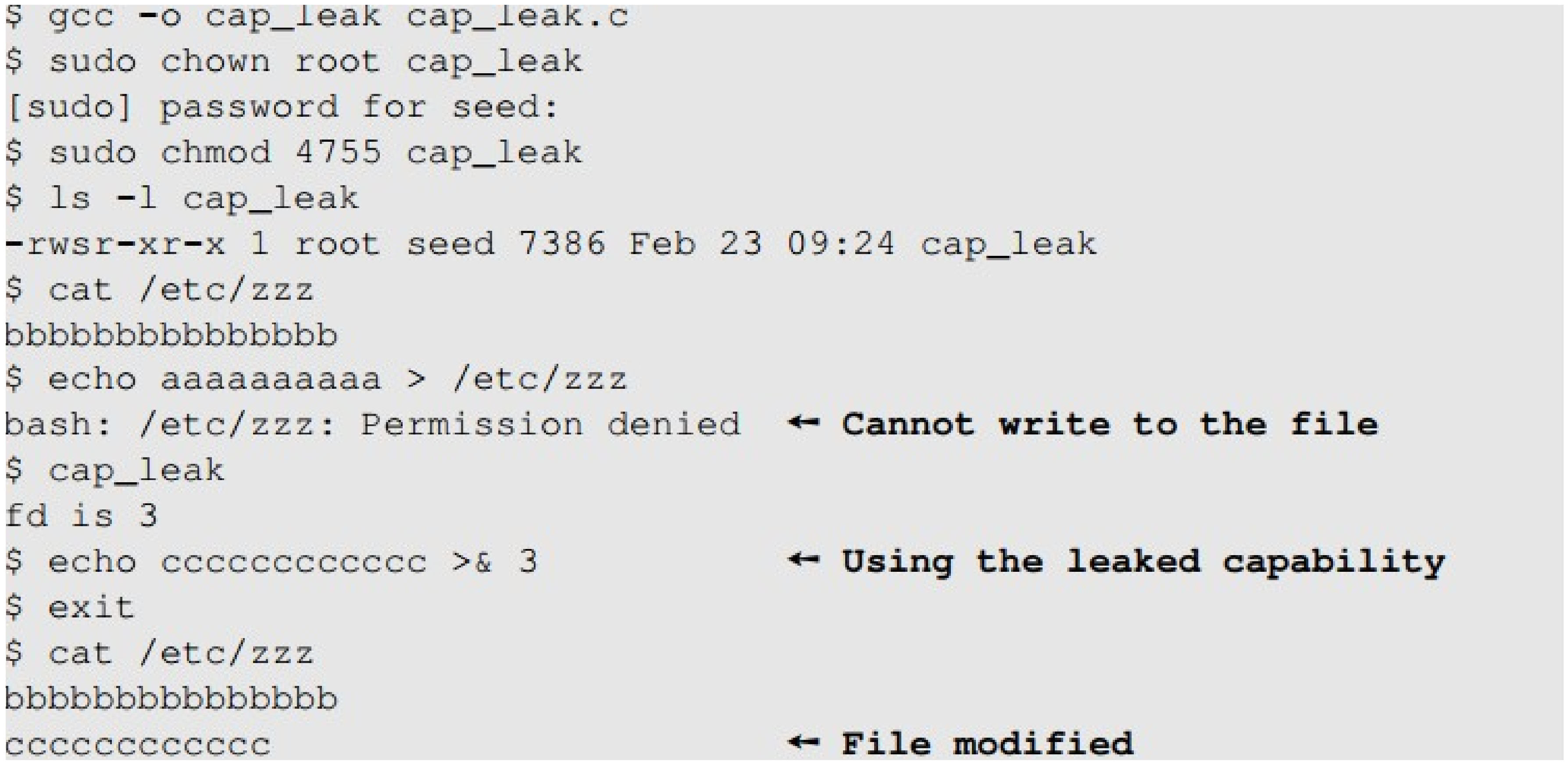
Invoke a shell program, so the behavior restriction on the program is lifted

```
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
bbbbbbbbbbbbbb
$ 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
bbbbbbbbbbbbbb
cccccccccccc                        ← File modified
```

How to fix the program?

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

Principle of Isolation- 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 command, with the root privilege?

Invoking Programs : Unsafe Approach (Continued)

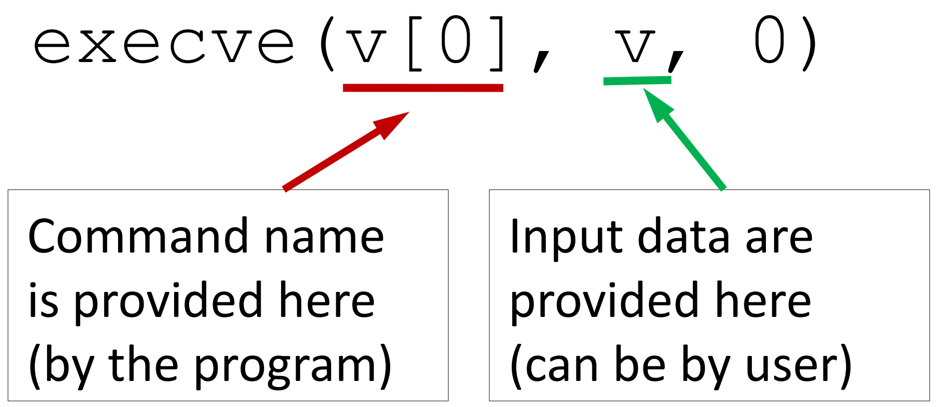
```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::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

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

Principle of Isolation

Principle: Don’t mix code and data.

Attacks due to violation of this principle :

- system() code execution
- Buffer overflow
- Cross site scripting
- SQL injection

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