Secure System Design: Threats and Countermeasures

CS-392

Spring 2019

Secure System Design: Threats and Countermeasure

- Instructor:
  - Samrat Mondal
  - samrat@iitp.ac.in
  - Office Location: R-405, Block-3
  - Office Hours: Mon from 4 pm to 5 pm or appointment through email
- TA :
  - Suryakanta (suryakanta.pcs15@iitp.ac.in)
  - Mainak (mainak.mtcs17@iitp.ac.in)
- Course Materials:
  - Will be available in http://172.16.1.3/~samrat/

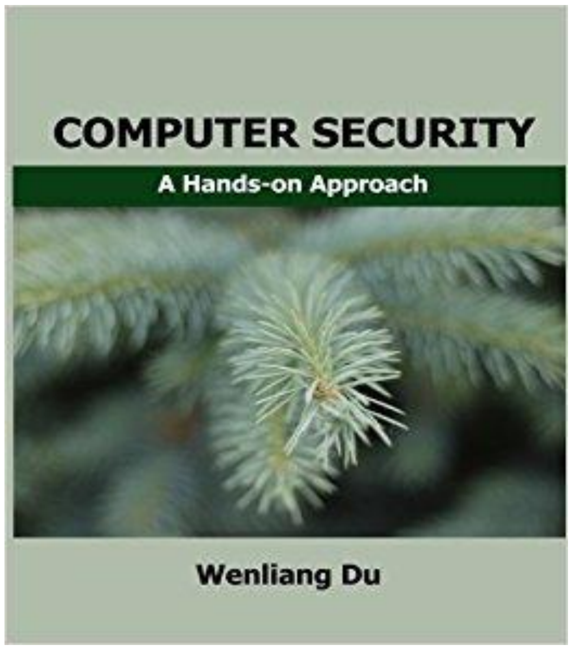
Class Timings/Venue:
Monday: 12 noon to 12.55 pm/R302
Wednesday: 11 am to 11:55 am/R307
Thursday: 10 am to 10:55 pm/R301

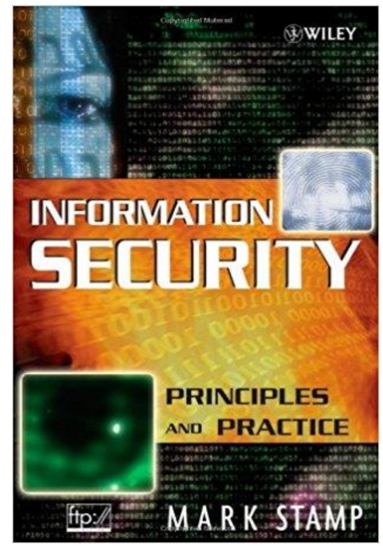
Tentative Plans

- Pre-midsem
- Overview of Unix Security basics and some classical attacks like buffer overflow, format string, race condition;
- Shell functions, Shellshock vulnerability, Shellshock attack on Set-UID program, Shellshock attack on CGI Programs;
- Return to libc attack;
- Dirty Cow Attack;
- Password file compromise Attack, Countermeasures;

- Post-midsem
- Code Analysis using Software Reverse Engineering;
- Access Control in Android Smartphone, Attack on Android Smart phone;
- Interaction with the database in Web Application, SQL-Injection Attack, Countermeasures;
- ClickJacking attack;
- Cross-Site Requests and Its Problems, Cross-Site Request Forgery Attack, Scripting Attack;
- Side Channel Attack; Attack against CPU

Books


COMPUTER SECURITY
A Hands-on Approach
Wenliang Du


WILEY
INFORMATION
SECURITY
PRINCIPLES
AND PRACTICE
ftp://
MARK STAMP

Evaluation Policy

- Assignments: 30%
- Quizzes: 20%
- MidTerm: 20%
- Final: 30%

Students who are caught cheating will receive zero for their assignment, and the final grade will only be BC or lower, regardless how good the overall score is.

75% attendance is mandatory to appear for the final exam

Objectives of this Course

- To get familiar with the important security concerns that a software developer or manager or a stakeholder must be aware of
- To understand the various classical flaws in systems that can lead to security problems.
- Also, some possible countermeasures will also be discussed

- For programming assignments and practice, you can use virtual box and install 32 bit Pre-built ubuntu image
- The links will be provided in the course page

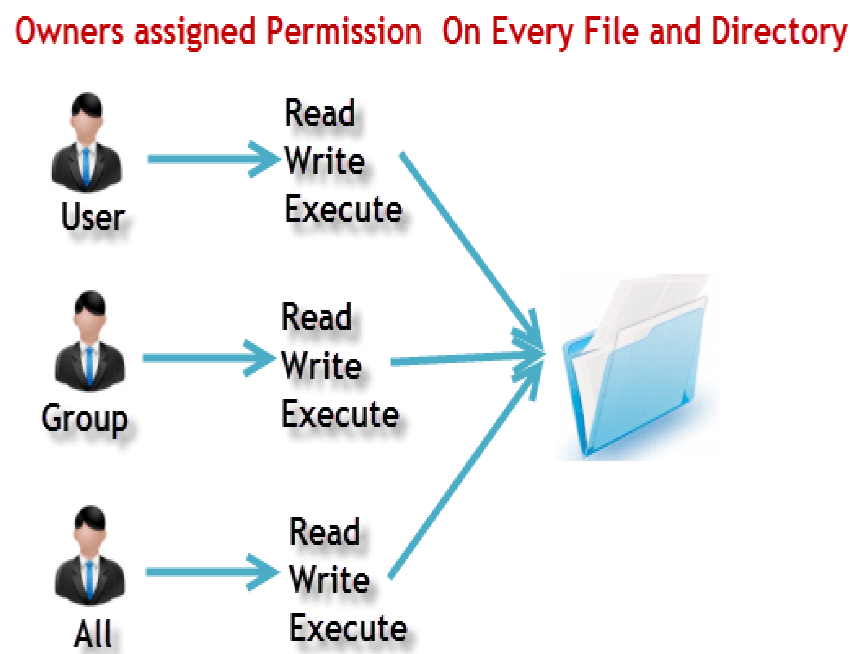
Let’s Begin

Ownership of Linux Files

- Every file and directory on Unix/Linux system is assigned 3 types of owner
  - User: A user is the owner of the file. By default, the person who created a file becomes its owner. Hence, a user is also sometimes called an owner.
  - Group: All users belonging to a group will have the same access permissions to the file.
  - Other: Any other user who has access to a file.

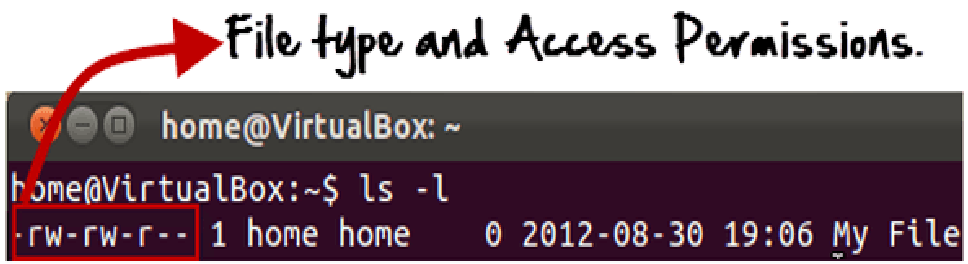
Permissions

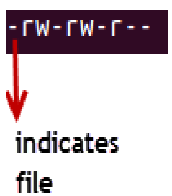
- Every file and directory in UNIX/Linux system has following 3 permissions defined for all the 3 owners.
  - Read
  - Write
  - Execute

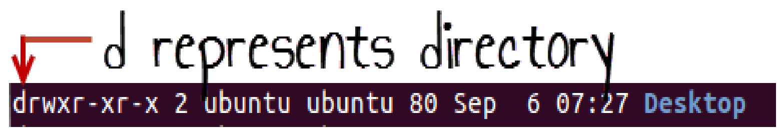


ls command to check permission

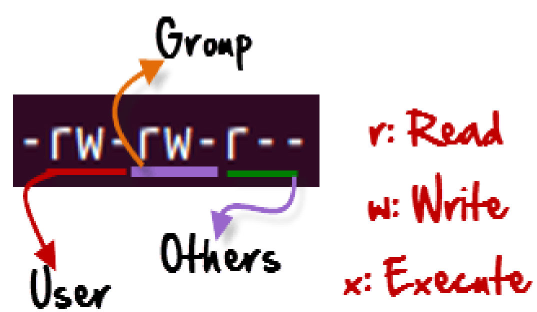
```
ls - l
```







r = read permission
w = write permission
x = execute permission
- = no permission



chmod command

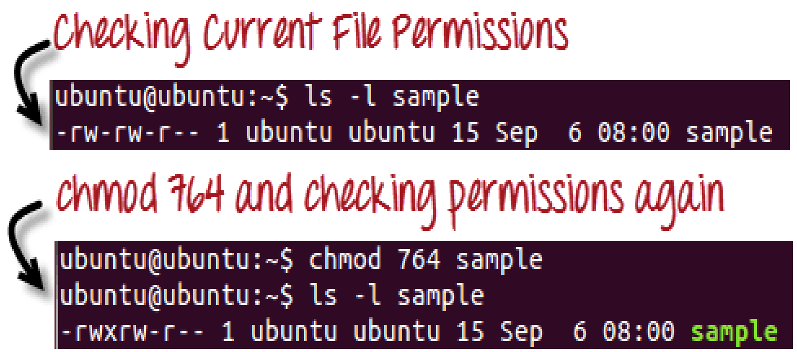
- The '**chmod'** command stands for 'change mode'. Using the command, we can set permissions (read, write, execute) on a file/directory for the owner, group and the world.
- Syntax: *chmod permission filename*
- Two ways-
  - Absolute mode
  - Symbolic mode

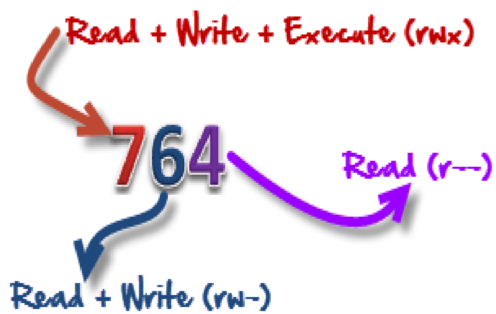
Absolute Mode

- In this mode, file **permissions are not represented as characters but a three-digit octal number**.

Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rwx

chmod in Absolute mode





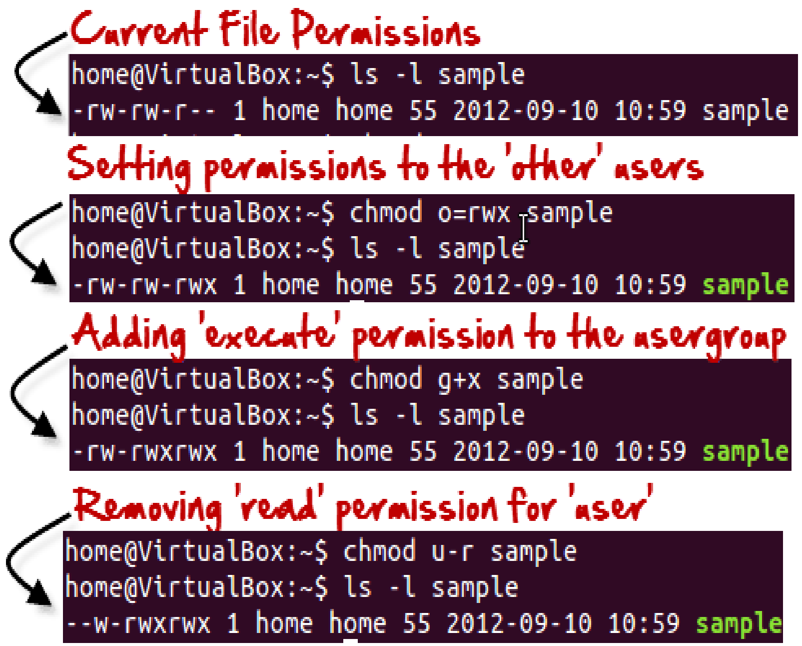
Symbolic mode

- Useful to modify permissions of a specific owner. It makes use of mathematical symbols to modify the file

Operator	Description
+	Adds a permission to a file or directory
-	Removes the permission
=	Sets the permission and overrides the permissions set earlier.

User Denotations	
u	user/owner
g	group
o	other
a	all

chmod in symbolic mode



Changing ownership

- For changing the ownership of a file/directory, you can use the following command:
  - Syntax: chown user
- To change the user as well as group for a file or directory use the command
  - Syntax: chown user: group filename

chown command

Check the current file ownership using ls -l

```
-rw-rw-r--  1 root n10    18 2012-09-16 18:17 sample.txt
```

Change the file owner to n100 . You will need sudo

```
n10@N100:~$ sudo chown n100 sample.txt
```

Ownership changed to n100

```
-rw-rw-r--  1 n100 n10    18 2012-09-16 18:17 sample.txt
```

Changing user and group to root 'chown user:group file'

```
n10@N100:~$ sudo chown root:root sample.txt
```

User and Group ownership changed to root

```
-rw-rw-r--  1 root root    18 2012-09-16 18:17 sample.txt
```

Linux Password file

- Traditional Linux systems keep user account information, including one-way encrypted passwords, in a text file called “/etc/passwd”
- As this file is used by many tools (such as ``ls'') to display file ownerships, etc. by matching user id #'s with the user's names, the file needs to be world-readable.

/etc/passwd file

- ``/etc/passwd'' file contains account information, and loo smithj:x:561:561:Joe Smith:/home/smithj:/bin/bash

Each field in a passwd entry is separated with ":" colon characters, and are as follows:

- Username, up to 8 characters. Case-sensitive, usually all lowercase
- An "x" in the password field. Passwords are stored in the ``/etc/shadow'' file.
- Numeric user id. This is assigned by the ``adduser'' script. Unix uses this field, plus the following group field, to identify which files belong to the user.
- Numeric group id. Red Hat uses group id's in a fairly unique manner for enhanced file security. Usually the group id will match the user id.
- Full name of user.
- User's home directory. Usually /home/username (eg. /home/smithj). All user's personal files, web pages, mail forwarding, etc. will be stored here.
- User's "shell account". Often set to ``/bin/bash'' to provide access to the bash

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

```
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:::
```

d?

/etc/shadow file

1: Username: login name

2: Password: It is in encrypted form. Algorithms such as MD5, Blowfish, SHA-256, SHA-512 are used to store the password

3: Last Password changed: Days since 1st Jan 1970

4: Minimum: The minimum number of days required between password change

5: Maximum: The maximum number of days the password is valid. After that the user is forced to change his/her password

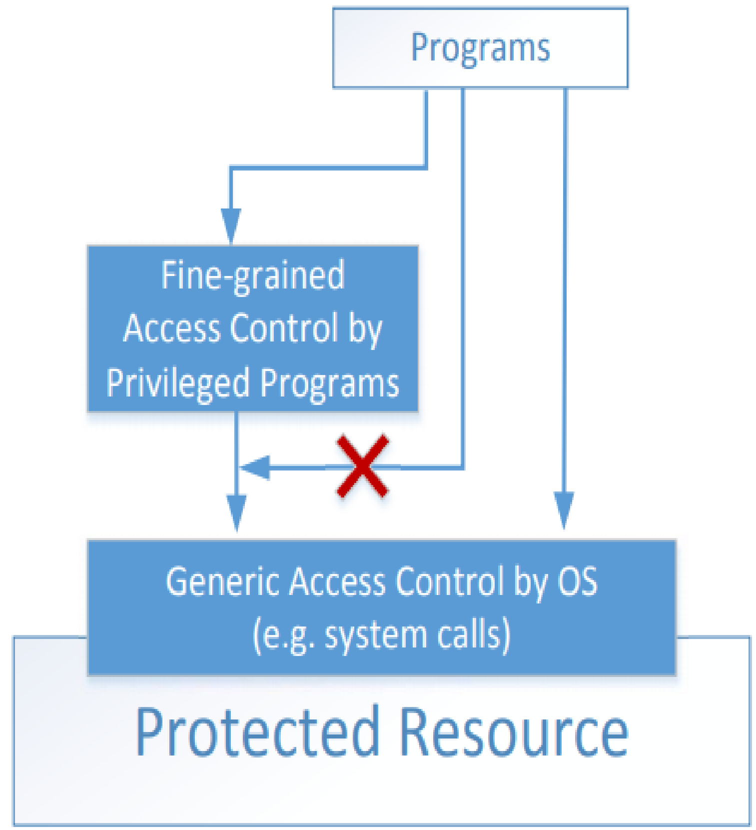
6: Warn: The number of days before the password is to expire that the user is warned that his/her password must be changed

7: Inactive: The number of days after password expires the account is disabled

8: Expire: An absolute date specifying when the login may no longer be used

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



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 two User IDs.
- **Real UID (RUID)**: Identifies real owner of process
- **Effective UID (EUID)**: Identifies privilege of a process
  - Access control is based on EUID
- 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 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

Set UID

- When an executable file's setuid permission is set, users may execute that program with a level of access that matches the user who owns the file.
- When viewing a file's permissions with the **ls - l** command, the setuid permission is displayed as an "**s**" in the "user execute" bit position.

```
ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root 54192 Nov 20 17:03 /usr/bin/passwd
```

- To set the set-uid bit

```
chmod u+s myfile
```

- Non-executable files can be marked as set-uid, but it has no effect;

```
ls -l myfile
```

```
-rw-r--r-- 1 user 0 Mar 6 10:45 myfile
```

```
chmod u+s myfile
```

```
ls -l myfile
```

```
-rwSr--r-- 1 user 0 Mar 6 10:45 myfile
```

Uppercase letter

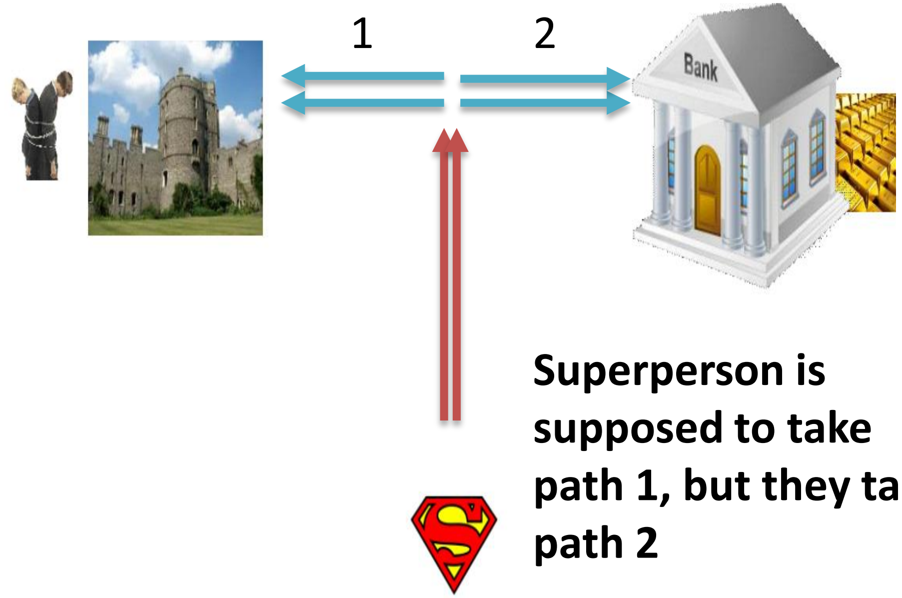
If we change the permission to **u+x**, then the set-uid permission comes into effect.

Attack on Superman

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

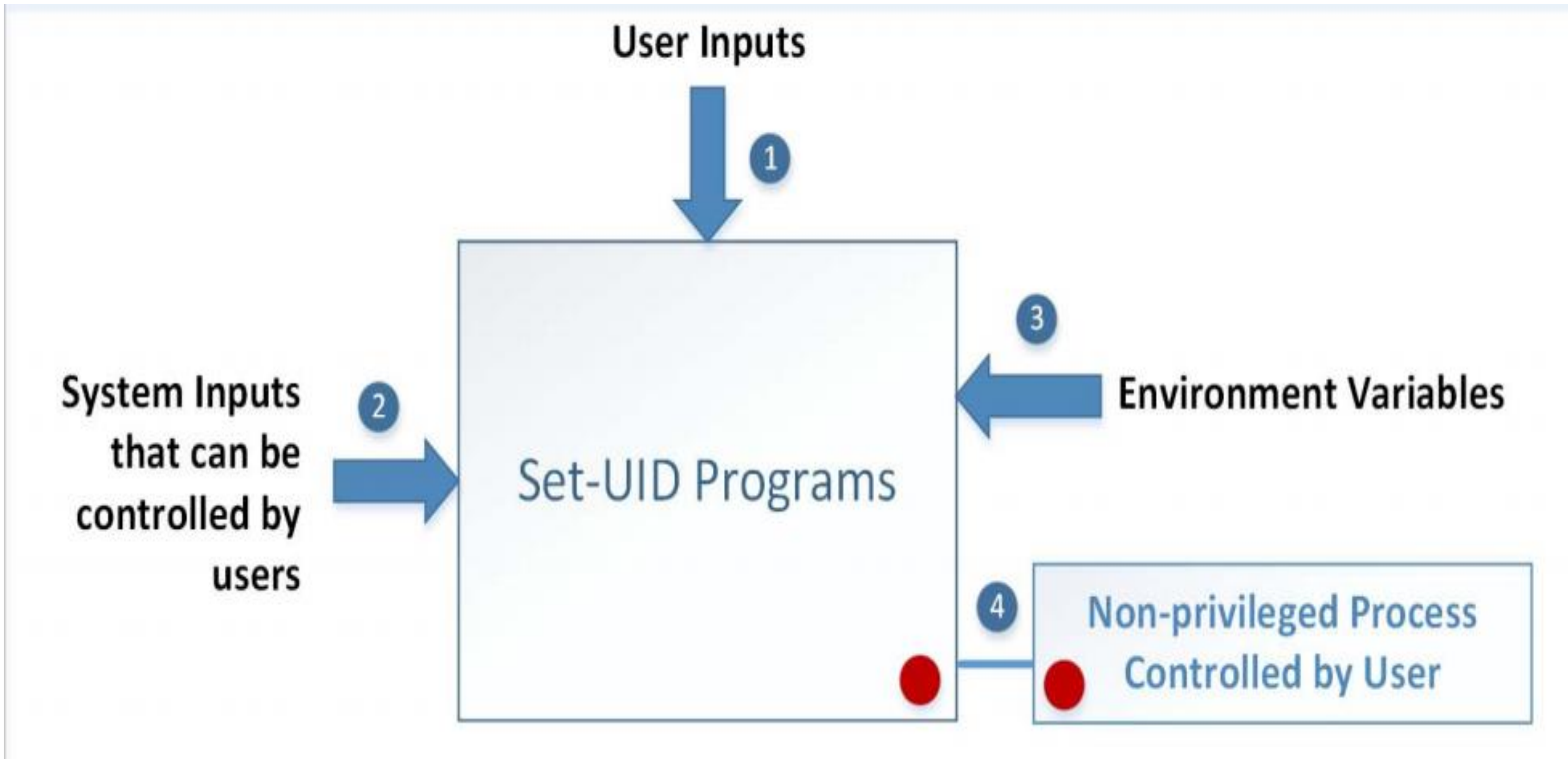
- Superperson Mallory
  – Fly north then turn left
  – How to exploit this code?

- 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



Attacks via User Inputs

User Inputs: Explicit Inputs

- Buffer Overflow
  - 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

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

Attack

Attacks via System Inputs

System Inputs

- Race Condition
  - 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 done later.

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 will be presented later

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 it is safe.
- **Scenario 2 (unsafe):** Similar to the "`su`" program, the privileged program downgrade its privilege, and lift 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 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

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
- Cross Site Scripting
- SQL injection
- Buffer Overflow attacks

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 Linux, seteuid() and setuid() can be used to disable/discard privileges.
- Different OSes have different ways to do that.

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