(Advanced) Computer Security!

Privileged Programs & Program Inputs:

The Set-UID Mechanism & Environment Variables

(part I)

Prof. Travis Peters

Montana State University

CS 476/594 – Computer Security

Spring 2021

https://www.traviswpeters.com/cs476

Today

Reminder!
Please update your Slack, GitHub, Zoom
(first/last name, professional photo/background)

- Announcements
  - Lab 00 recap (look @ a few exemplary README.md files)
  - Lab 01 released

- Learning Objectives
  - Review fundamental ideas in Linux security + related Linux commands
  - Understand the need for privileged programs
  - Understand how the Set-UID mechanism works
  - Examine attack surface of (Set-UID) programs
  - Examine the role of environment variables in shells and Set-UID programs

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

How would you protect your computer & its resources? What sorts of things should we consider?

Think. Pair (Break Out Rooms). Share.

Encrypt important data - at rest (secret)

VPNs - data in transit

Back-ups (integrity) loss of data → ransomware

Checksums (crypto hashes)

antivirus sw (active scanning)

passwords / password hygene

plugins / sw block ads (adware) (other things)

Cookies

2-factor auth (multi-factor)

(Public key Crypto)

Access Control

Access Control:

How would you protect your computer & its resources?

who can do **what** to **whom**

users/groups

Need notions of "identity"

objects

Usually *things* on a *filesystem*

permissions (read/write/execute)

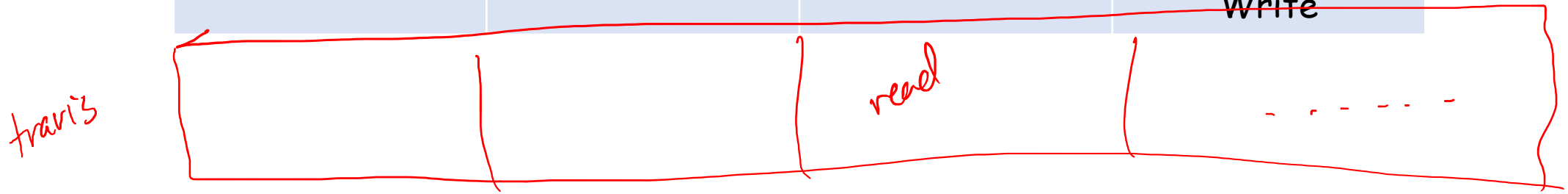
OK, I know the who/whom—what are **you permitted to do?**

Access Control:

Access Control Matrix

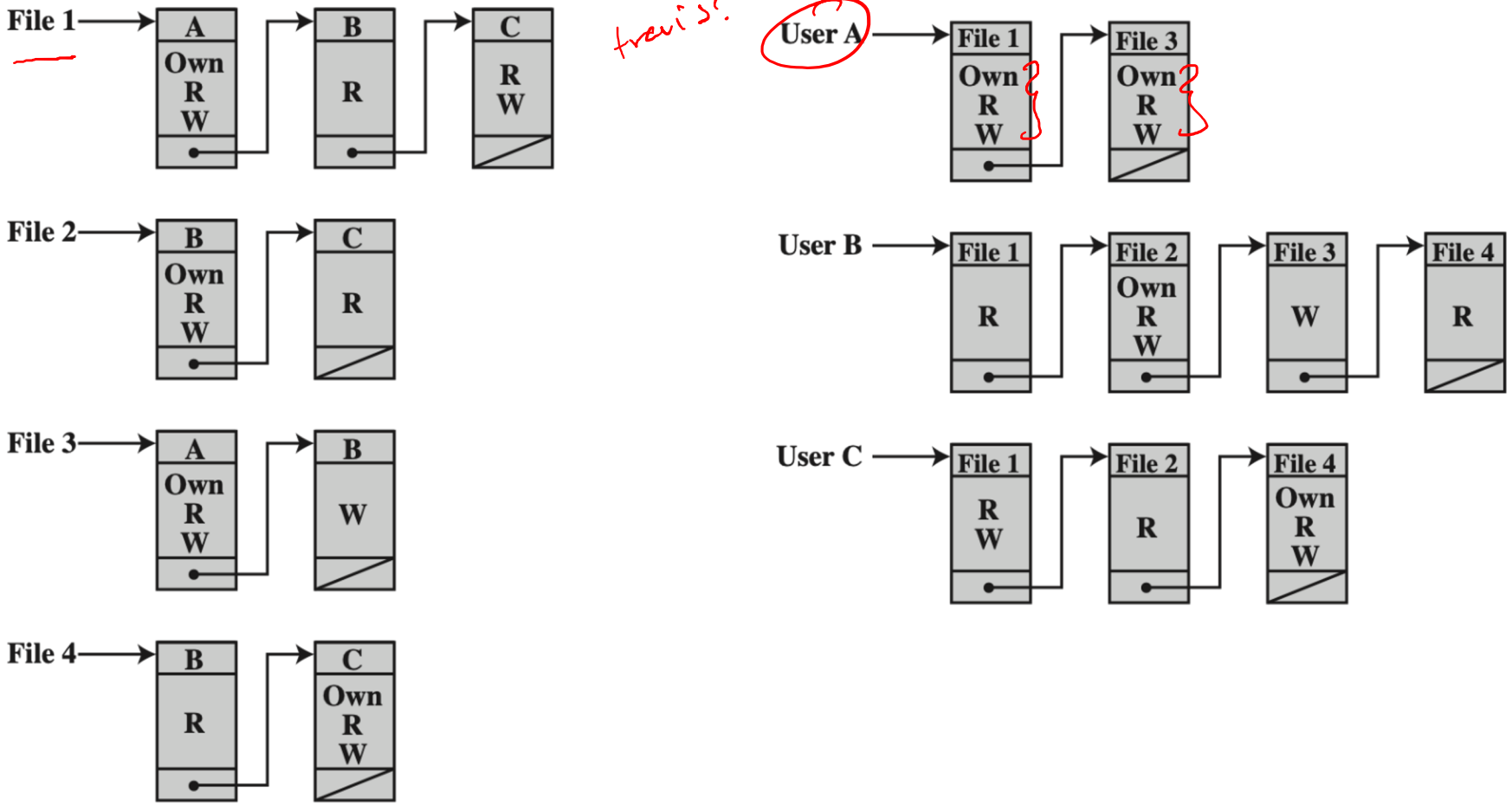
Objects

Subjects	File 1	File 2	File 3	File 4
User A	Own Read Write		Own Read Write	
User B	Read	Own Read Write	Write	Read
User C	Read Write	Read		Own Read Write



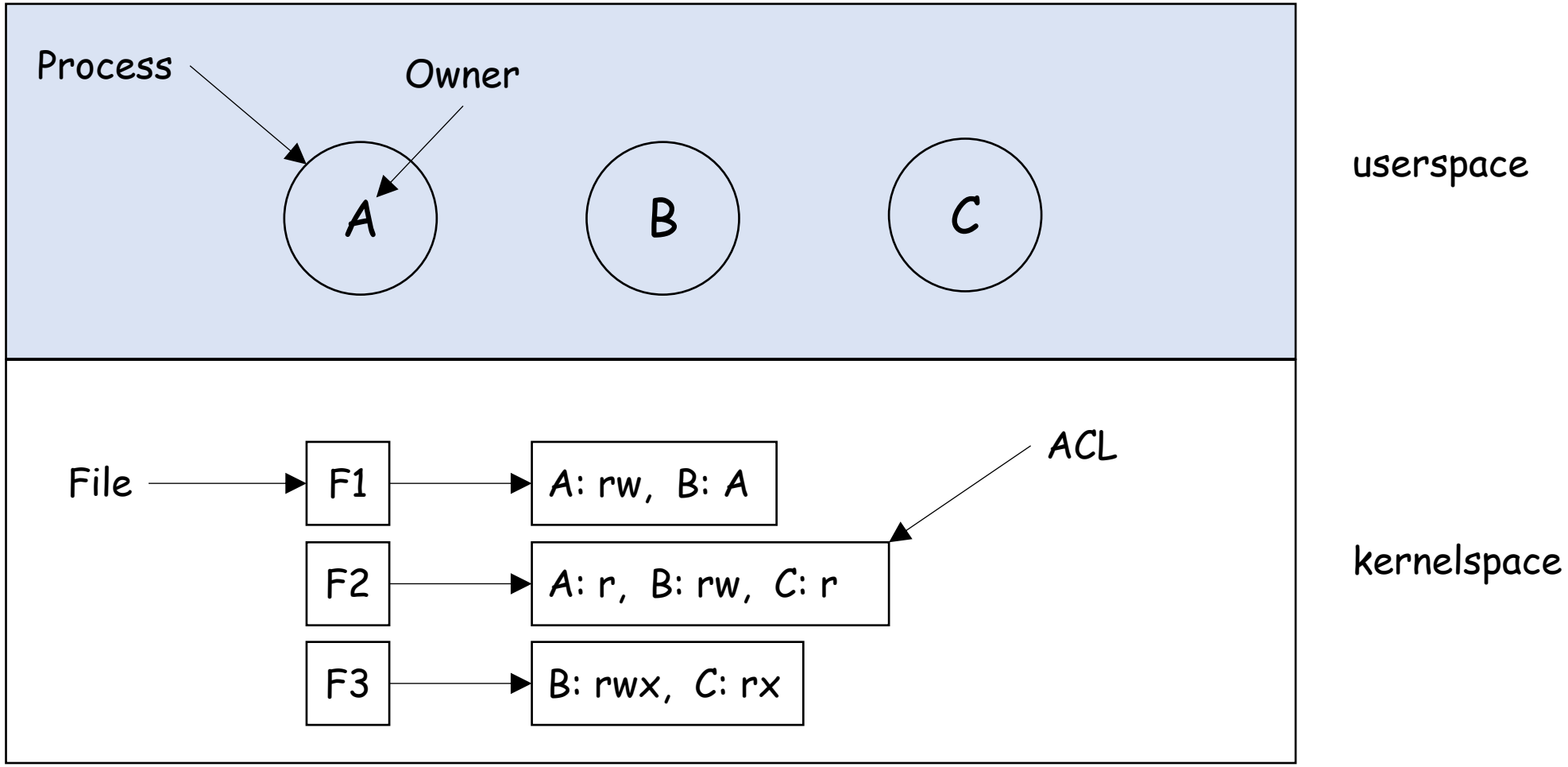
Access Control:

Access Control List (ACL)



Access Control:

A Unix ACL Example



Access Control:

Unix File Modes & Permissions

- Every Unix file has a set of **permissions** that determine whether someone* can read, write, or run the file.
  - Try: `ls -l ~`
  - Try: `ls -l /dev`

Access Control:

Unix File Modes & Permissions

- Every Unix file has a set of **permissions** that determine whether someone* can read, write, or run the file.
  - Try: ls -l ~
  - Try: ls -l /dev

Every file has...

```
$ ls -l file
-rw-r--r-- owner group date/time file
```

- File mode (4 parts)
  -> [file type][user][group][other]
  - file type
  - ugo -> rwx

- How do we change these settings?
  - Ex: allow other members of my **group** to **write** "file"?

A typical who can do what to whom flow

If **user A** asks to perform **operation O** on a **file object F**, the OS checks:
e.g., A tries to read F, where:

```
$ ls -l F
-rw-rw-r--  B  G  ...  F
```

- Is **A** the owner of **F**?
  - use **owner permissions** to decide whether A can do operation O.
  - e.g., A is not F's owner
- Is **A** a member of **F's** group?
  - use **group permissions** to decide…
  - e.g., A is not F's owner or a member of F's group, G
- Otherwise…
  - use **other permissions** (i.e., "everyone else") to decide…
  - e.g., A can (r)ead the file

You Try!

Think. Pair (Break Out Rooms). Share.

Suppose **user C** asks to **execute** a **file object F2**. Will they be able to do so?

```
$ ls -l F
-rwxrwxrwx   B H   ...   F1
-rwxr-xr--   D G   ...   F2
-rw-r-----   D H   ...   F3
-rw-rw-rw-   B G   ...   F4
```

Note:

- Group = G = {A, C, K, M, Q, Z}
- Group = H = {A, B, C, Q}

Demos/Activities:
Basic File Ops, Users, and Groups

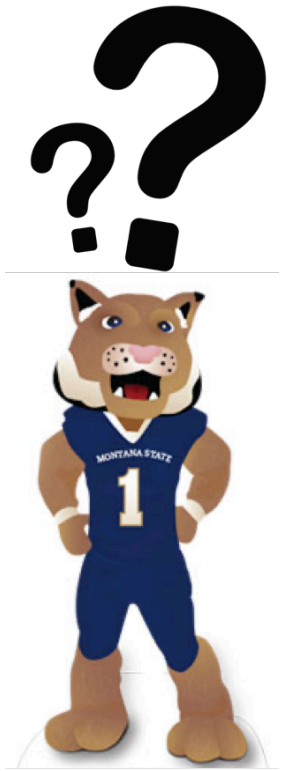
Examples and commands to try can be found here:

https://github.com/traviswpeters/cs476-code/tree/master/00_intro (see README.md)

Limitations of File-Based Access Control

Question: How can a non-privileged user 'champ' change their own password?

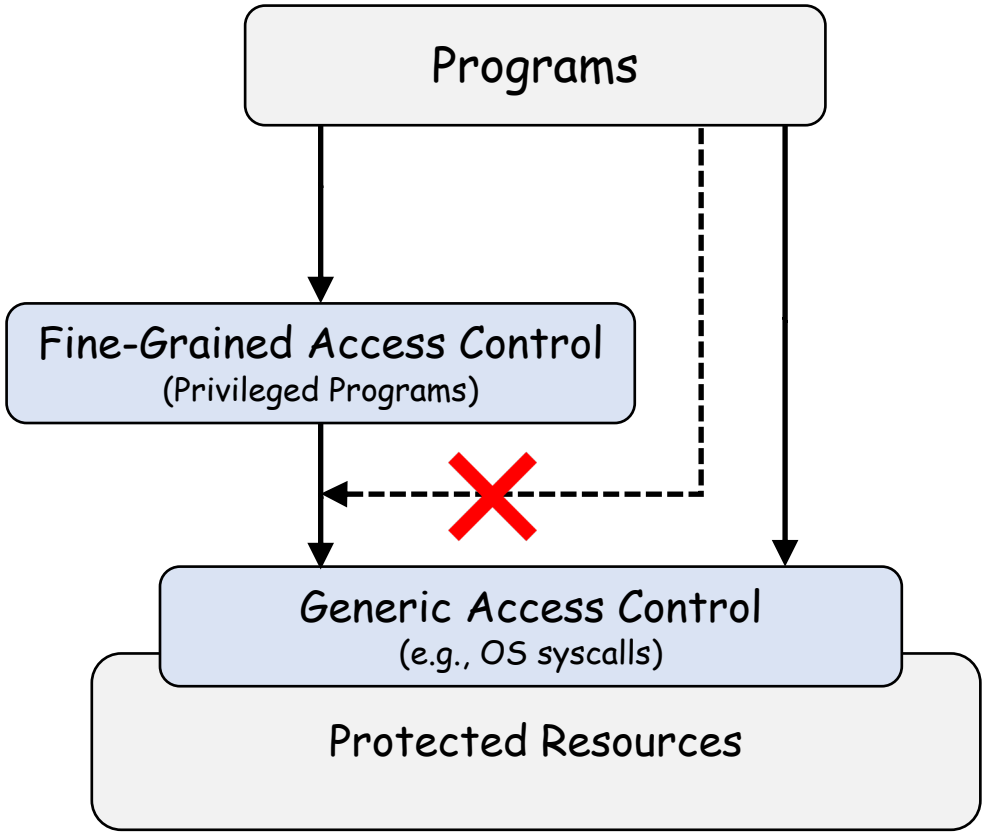
```
[seed@VM][~]$ ls -al /etc/passwd
-rw-r--r-- 1 root root 2886 Nov 24 09:12 /etc/passwd

[seed@VM][~]$ ls -al /etc/shadow
-rw-r----- 1 root shadow 1514 Nov 24 09:12 /etc/shadow
```



Ideas?

Two-Tier Approach to Access Control

- Implementing fine-grained access control in the OS makes OS code really complicated… (we generally try to avoid this…)
- OS relies on other extensions/features to enforce fine-grained access control
  -> "Privileged programs"



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
  - A normal program… but marked with a special bit

Superman's Past (The Stories You Never Heard…)

- **Superman's 1st Attempt—The Power Suit**
  - **Superman** got tired of saving *everyone*
  - **Super*people:*** give normal people superman's power!
  - **Problem:** not all superpeople are good……

Superman's Past (The Stories You Never Heard…)

- **Superman's 2nd Attempt—Power Suit 2.0**
  - Power suit w/ a sweet computer in it
  - Power suit can only perform a specific task
  - No way to deviate from the pre-programmed task……

The Set-UID mechanism is a lot like superman's power suit 2.0
(but implemented in UNIX systems…)

Set-UID In A Nutshell

There is also a Set-GID (Set Group ID), which works in basically the same way, but applies to group privileges

- Allow user to run a program with the program ***owner's*** privilege
  - i.e., a UNIX mechanism for changing user/group identity
  - Allows users to run programs w/ temporarily elevated privileges

- Created to deal with inflexibilities of UNIX access control
  - *Why might this be useful?*
  - *Why might this be a bad idea?*

- Example: the **passwd** program

```
[seed@VM][~]$ ls -al /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 May 28  2020 /usr/bin/passwd
```

Set-UID In A Nutshell (cont.)

There is also a Set-GID (Set Group ID), which works in basically the same way, but applies to group privileges

- Every process has two User IDs
  - **Real UID (RUID)** — identifies the **owner** of the process
  - **Effective UID (EUID)** — identifies **current privilege** of the process
  - Access control decisions are based on EUID!

- When a normal program is executed,
  - RUID == EUID
    EUID == RUID == user who ***runs*** the program

- When a Set-UID program is executed,
  - RUID != EUID
    EUID == ID of program's owner

If program owner == root, the program runs with root privileges

Demos / Activities

If we have time… else look at some slides

So Uh… How Do You Set… The Set-UID Bit Thingy

Change the owner of a file to root

```
[seed@VM][~]$ cp /bin/cat ./mycat
[seed@VM][~]$ sudo chown root mycat
[seed@VM][~]$ ls -al mycat
-rwxr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
```

Before enabling the Set-UID bit

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

After enabling the Set-UID bit

```
[seed@VM][~]$ sudo chmod 4755 mycat
[seed@VM][~]$ ls -al mycat
-rwsr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
[seed@VM][~]$ mycat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
...
```

How It Works

Try It!

```
[seed@VM][~]$ cp /usr/bin/id ./myid
[seed@VM][~]$ sudo chown root myid
[seed@VM][~]$ ./myid
???
```

```
[seed@VM][~]$ sudo chmod 4755 myid
[seed@VM][~]$ ./myid
???
```

How It Works

A Set-UID program is just like any other program,
except that it has a special bit set
(the Set-UID bit)

```
[seed@VM][~]$ cp /usr/bin/id ./myid
[seed@VM][~]$ sudo chown root myid
[seed@VM][~]$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),...
```

...if set-uid bit enabled, set EUID according to file owner

```
[seed@VM][~]$ sudo chmod 4755 myid
[seed@VM][~]$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom)
```

...access control decisions made based on EUID!

So… Is Set-UID Secure?

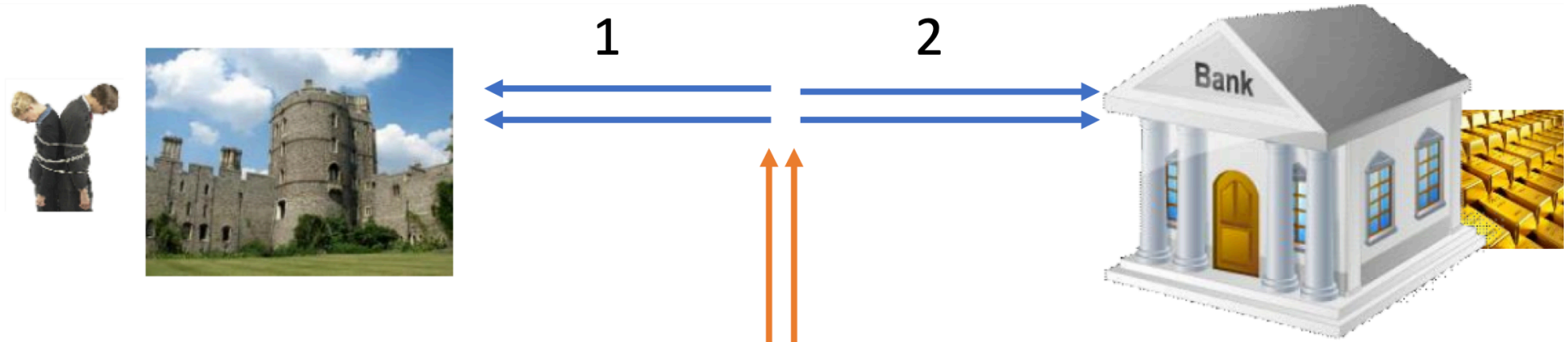
- Allows normal users to escalate privileges
  - This is different from directly giving escalated privileges (e.g., **sudo**)
  - Restricted behavior — similar to superman's power suit

- Unsafe to run all programs as Set-UID programs…….. Examples?
  - /bin/sh — *why?*
  - vi(m) — *why?*

Software (and Superman's Power Suit…) Is Only As Good As We Make It…

Shouldn't assume that the user can only do whatever is coded…

What sorts of attacks are possible on Superman's PS2.0?

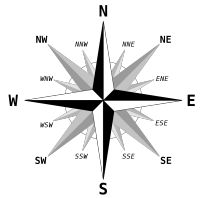


Mallory (v1)

Fly north, turn left,
knock down wall, capture bad guy
Where could this go wrong?

Mallory (v2)

Fly north, turn west,
knock down wall, capture bad guy
Where could this go wrong?



(Advanced) Computer Security!

Privileged Programs & Program Inputs:

The Set-UID Mechanism & Environment Variables

(part II)

Prof. Travis Peters

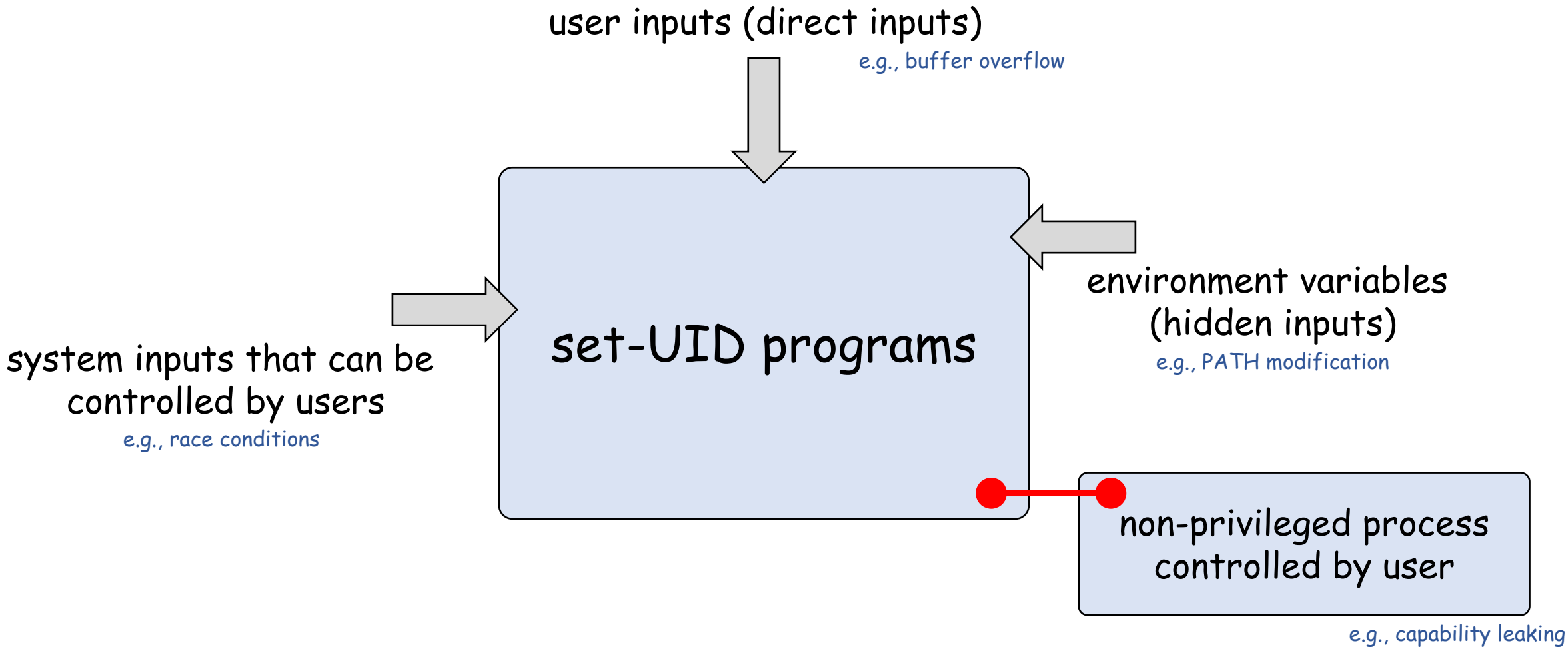
Montana State University

CS 476/594 – Computer Security

Spring 2021

https://www.traviswpeters.com/cs476

Attack Surface of (Set-UID) Programs



Invoking Programs… From Within Programs…

system() vs. exec*()

Invoking Programs

- We can invoke external commands/programs from INSIDE another program
  - system()
  - exec()-family

- External commands should not be arbitrary
  - Developer of Set-UID programs should define these calls
  - Users are certainly not supposed to control/influence these calls
  - But input from the users is often needed……

Be careful not to mix code and data!

Invoking Programs (The Unsafe Way)

- **Idea:** Use **system()** – why not?
  "It's super easy... I know what I'm doing..."

<Let's Check out Task 8 – Lab 01>

Why is system() considered to be unsafe? How does system() work?

Can you use this program to run
some other command w/ root privileges?

Invoking Programs (The **Unsafe** Way)

- Idea: Use system() – why not?
  "It's super easy... I know what I'm doing..."

  -> system() invokes a shell and passes string to shell as command to run!

<Let's Check out Task 8 – Lab 01>

system("v[0] v[1]")

no clear separation between **command** and **data**

Invoking Programs (The Unsafe Way)

Change shell to circumvent dash countermeasure

```
[seed@VM][~]$ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh (no countermeasure)
[seed@VM][~]$ sudo ln -sf /bin/dash /bin/sh  # set shell to dash (has countermeasure)
```

Make audit program (compile, make root-owned set-uid program) --> compile version with **system()**

```
[seed@VM][~]$ gcc -o audit catall.c
[seed@VM][~]$ sudo chown root audit
[seed@VM][~]$ sudo chmod 4755 audit
[seed@VM][~]$ ls -al audit
...
[seed@VM][~]$ ./audit /etc/shadow
...
[seed@VM][~]$ ./audit "aa;/bin/sh"
...
# <--------- root shell
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=.........
```

Invoking Programs (The Safe(r) Way)

- **Idea:** Use **execve()**

<Let's Check out Task 8 – Lab 01>

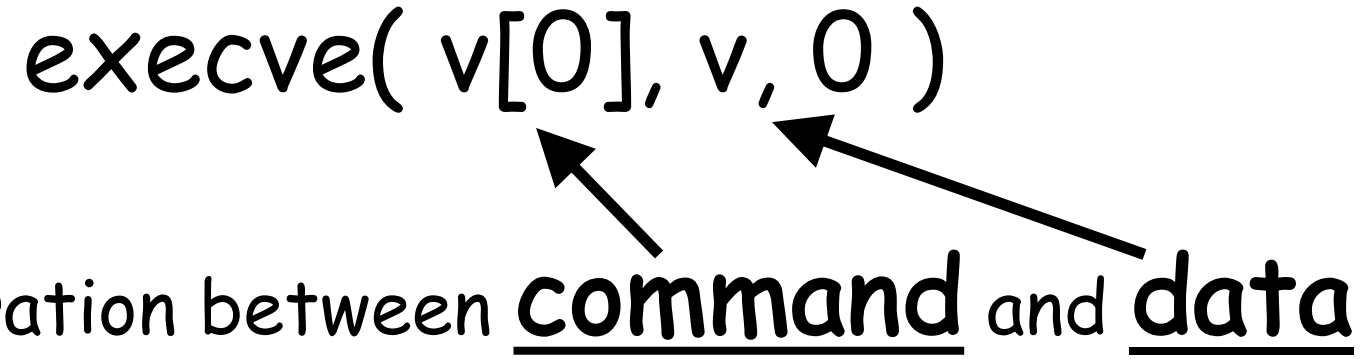
Why is execve() considered to be safe(r)?

Invoking Programs (The Safe(r) Way)

- Idea: Use execve()
  -> execve() runs the command directly

<Let's Check out Task 8 – Lab 01>

Why is execve() considered to be safe(r)?



BONUS: If absolute path to command is specified, execve() doesn't need to search PATH either!
(see later slides on environment variables)

Invoking Programs (The Safe(r) Way)

Change shell to circumvent dash countermeasure

```
[seed@VM][~]$ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh (no countermeasure)
[seed@VM][~]$ sudo ln -sf /bin/dash /bin/sh  # set shell to dash (has countermeasure)
```

Make audit program (compile, make root-owned set-uid program) --> compile version with **execve()**

```
[seed@VM][~]$ gcc -o audit catall.c
[seed@VM][~]$ sudo chown root audit
[seed@VM][~]$ sudo chmod 4755 audit
[seed@VM][~]$ ls -al audit
...
[seed@VM][~]$ ./audit /etc/shadow
...
[seed@VM][~]$ ./audit "aa;/bin/sh"
...
Attack Failed! No root shell! Input data is treated as data, not as code!
```

Invoking Programs (The Safe(r) Way)

- I say safe(r) because not all functions in the exec() family behave similarly to execve()…

- For example,
  - execlp(),
  - execvp(), and
  - execvpe()

All duplicate actions of the shell, and can be attacked via environment variables (e.g., PATH)

Invoking Programs in Other Languages

- The ability (and risks) of invoking external commands is not limited to C
- We should avoid problems similar to those caused by system() functions…..
- Examples
  - Python has a system() function as well……
  - Perl — open() can run commands, but does so through a shell
  - PHP — system()
  - …

  (OK we get it…)

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

  // go to:
  //   http://localhost/list.php?dir=.;date
  // and the resulting command executed on the server:
  //   $ /bin/ls .; date
?>
```

Environment Variables

Env. Variables - What Are They?!

- A set of dynamic named values
  - part of the environment in which a process runs.
  - Set of key=value pairs
  - Affect the way that running process will behave
  - **commands:** env, printenv, set, unset, export, echo…
  - **env. variables:** USER, HOME, EDITOR, SHELL, PATH, LANG…

- Example: The **PATH** variable
  - How does the shell know where to find commands?!
  - For instance, you typically say "ls" not "/bin/ls"
  - If full command is not provided,
    the shell process will use the PATH env. variable to search for it

Env. Variables – How Do I Access Them?!

There are 2 ways to access env. variables from C code...

```c
// Print environment variables using envp.
//
// Compile:
//  $ gcc myenv_envp.c -o myenv_envp

#include <stdio.h>

int main(int argc, char *argv[], char* envp[]) {
    int i = 0;
    while (envp[i] != NULL) {
        printf("%s\n", envp[i++]);
    }
    return 0;
}
```

envp - argument passed to main()

```c
// Print environment variables using environ.
//
// Compile:
//  $ gcc myenv_environ.c -o myenv_environ

#include <stdio.h>

extern char **environ;

int main(int argc, char *argv[], char* envp[]) {
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
    return 0;
}
```

environ – global variable

preferred way -> POSIX, global access, updates on changes, etc.

Env. Variables – How Do Processes Get Them?!

- Processes can get environment variables in one of two ways:
  - **fork()** -> the child process inherits its parent process's environment variables.
  - **exec*()** -> the memory space is overwritten, and all old environment variables are lost. However, execve() can explicitly pass environment variables from one process to another.

- Passing environment variables when invoking execve():

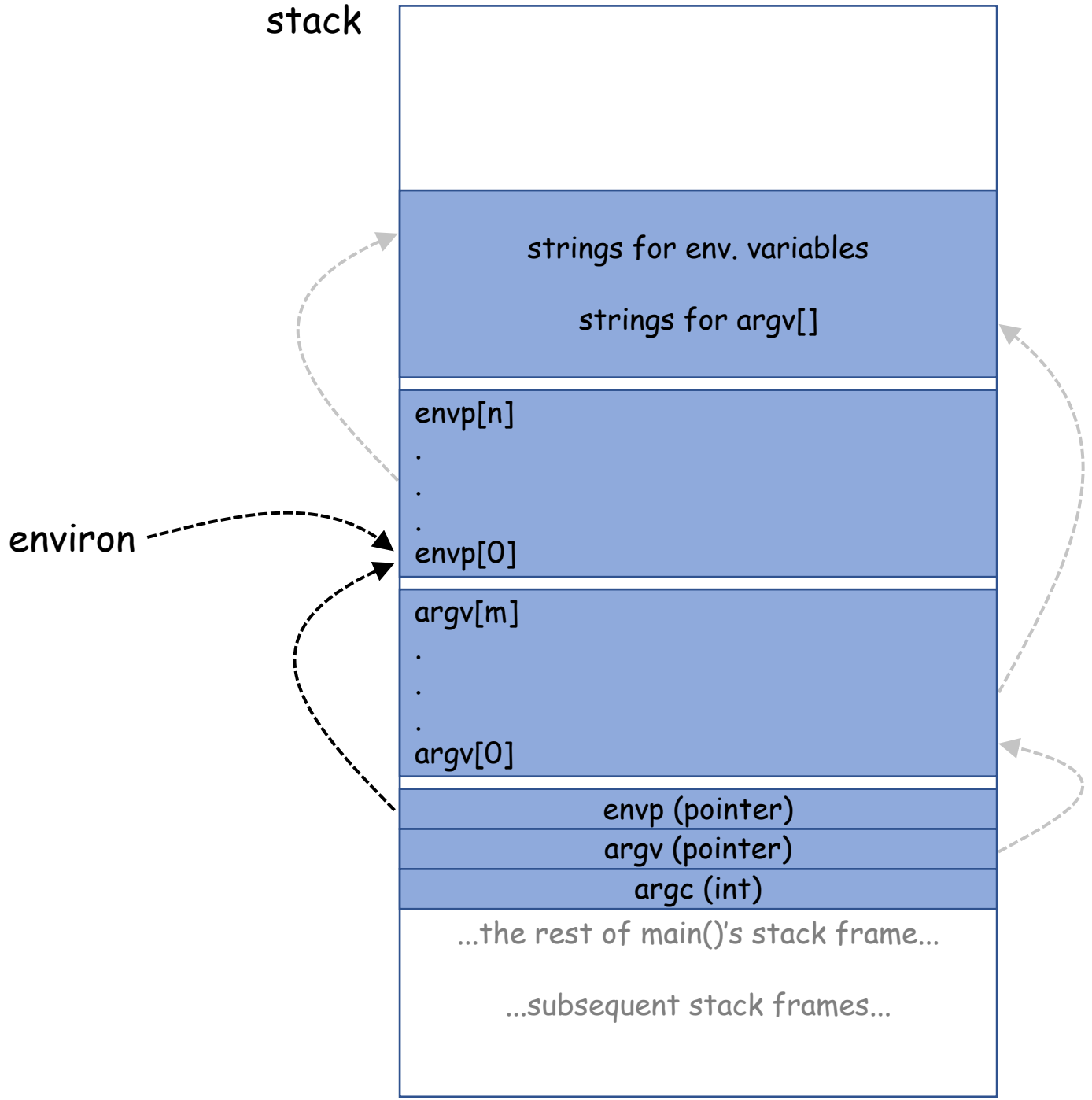
```
EXECVE(2)                                                EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const argv[],
                  char *const envp[]);
```

Env. Variables & execve() - Example

- passenv.c
- Execute /usr/bin/env, which prints out the environment variables of the current process.
- Try variations of environment as the 3rd argument passed to execve().
- Compile and run with options 1/2/3

```c
#include <stdio.h>
#include <unistd.h>

extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0; char* v[2]; char* newenv[3];
    if (argc < 2) return;

    // Construct the argument array
    v[0] = "/usr/bin/env";   v[1] = NULL;

    // Construct the environment variable array
    newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

    switch(argv[1][0]) {
    case '1': // Passing no environment variable.
        execve(v[0], v, NULL);
    case '2': // Passing a new set of environment variables.
        execve(v[0], v, newenv);
    case '3': // Passing all the environment variables.
        execve(v[0], v, environ);
    default:
        execve(v[0], v, NULL);
    }
}
```

Env. Variables - Where Are They In Memory?



Shell Vars. vs. Environment Vars.

Try it! Investigate shell and env. variables

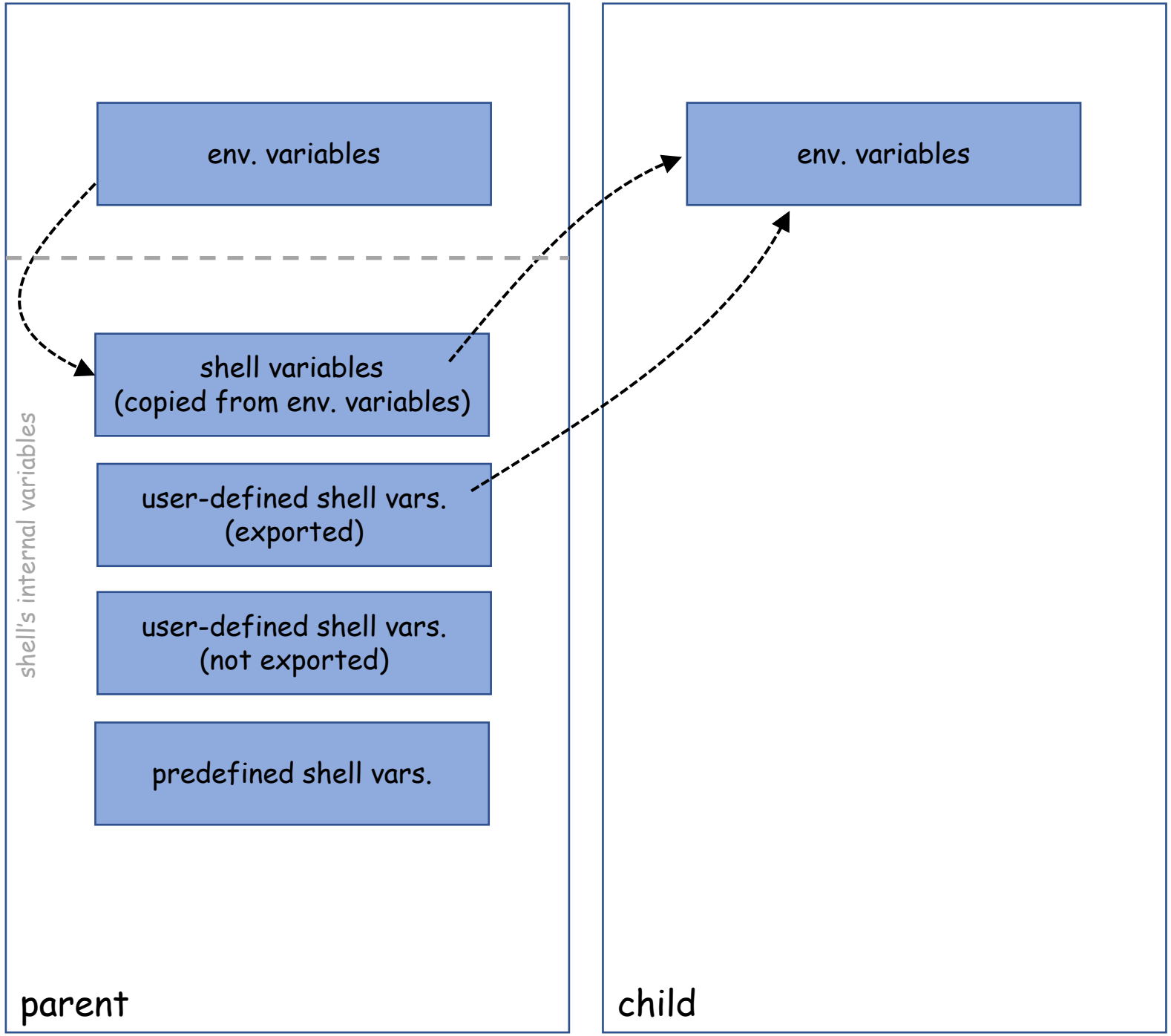
```
[seed@VM][~]$ strings /proc/$$/environ | grep LOGNAME
[seed@VM][~]$ echo $LOGNAME
[seed@VM][~]$ LOGNAME=bob
[seed@VM][~]$ echo $LOGNAME
[seed@VM][~]$ strings /proc/$$/environ | grep LOGNAME
[seed@VM][~]$ unset LOGNAME
[seed@VM][~]$ echo $LOGNAME
[seed@VM][~]$ strings /proc/$$/environ | grep LOGNAME
```

```
[seed@VM][~]$ strings /proc/$$/environ | grep LOGNAME
[seed@VM][~]$ LOGNAME2=alice
[seed@VM][~]$ export LOGNAME3=bob
[seed@VM][~]$ env | grep LOGNAME
[seed@VM][~]$ unset LOGNAME
[seed@VM][~]$ env | grep LOGNAME
```

NOTE:
When a new shell starts, it copies the environment variables into its shell variables.
Changes made to the shell variables later will not be reflected on the environment variables.

NOTE:
When we type env in the shell, the shell will create a new child process

Passing Shell Vars. vs. Env. Vars



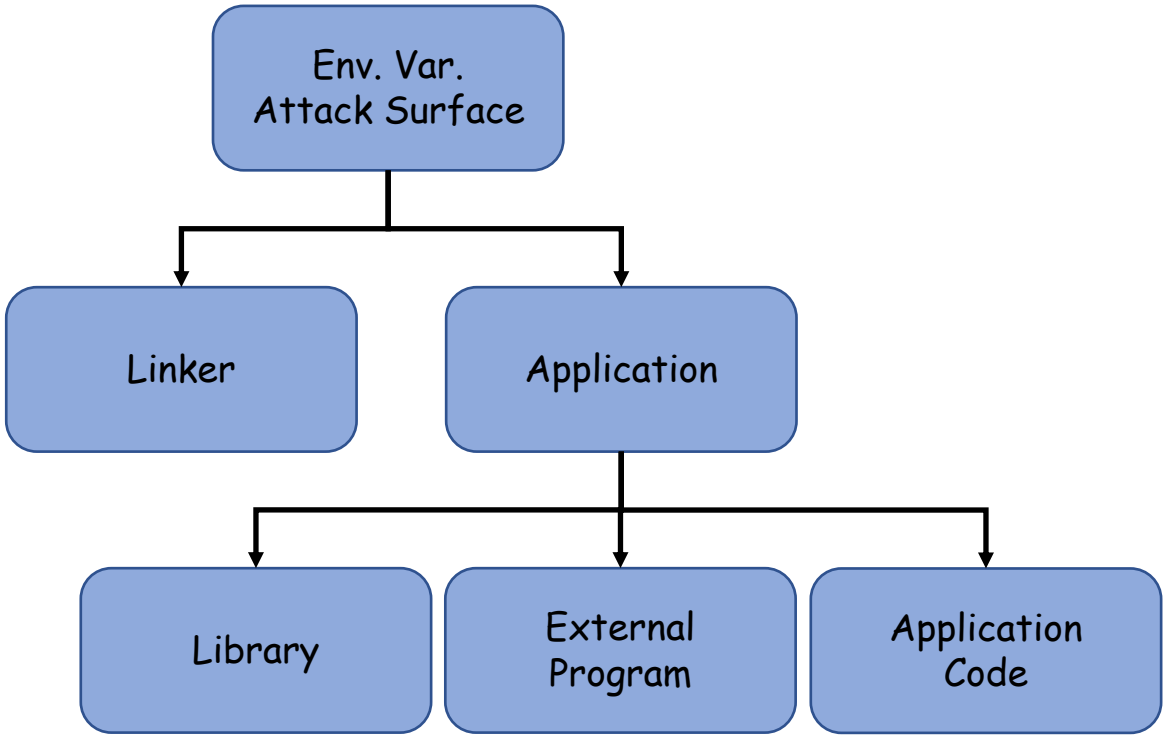
Case Studies

Case Study:

Attacks via Env. Variables

Mapping the Attack Surface of Programs That Depend on Environment Variables

- Hidden usage of environment variables is part of what makes them so dangerous…
- Also, users can modify environment variables…
- If Set-UID programs make use of environment variables, they become part of the attack surface of Set-UID programs!



Env. Attacks – Locating External Programs

- A program may invoke an external program (e.g., via system()) to do some work
- PATH contains a list of directories to search for executable programs
- If a program is invoked without using the absolute path to the program, e.g., `system("ls")`, the PATH env. variable is used to search for the program.
- PATH can be set by users, which provides an opportunity for users to influence which programs are found and executed!
- If the vuln. program were a Set-UID program…
  …and I could get it to run MY PROGRAM instead…

Lab 01: Task 6

Env. Attacks – Locating External Programs (cont.)

Change shell to circumvent dash countermeasure

```
[seed@VM][~]$ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh (no countermeasure)
[seed@VM][~]$ sudo ln -sf /bin/dash /bin/sh  # set shell to dash (has countermeasure)
```

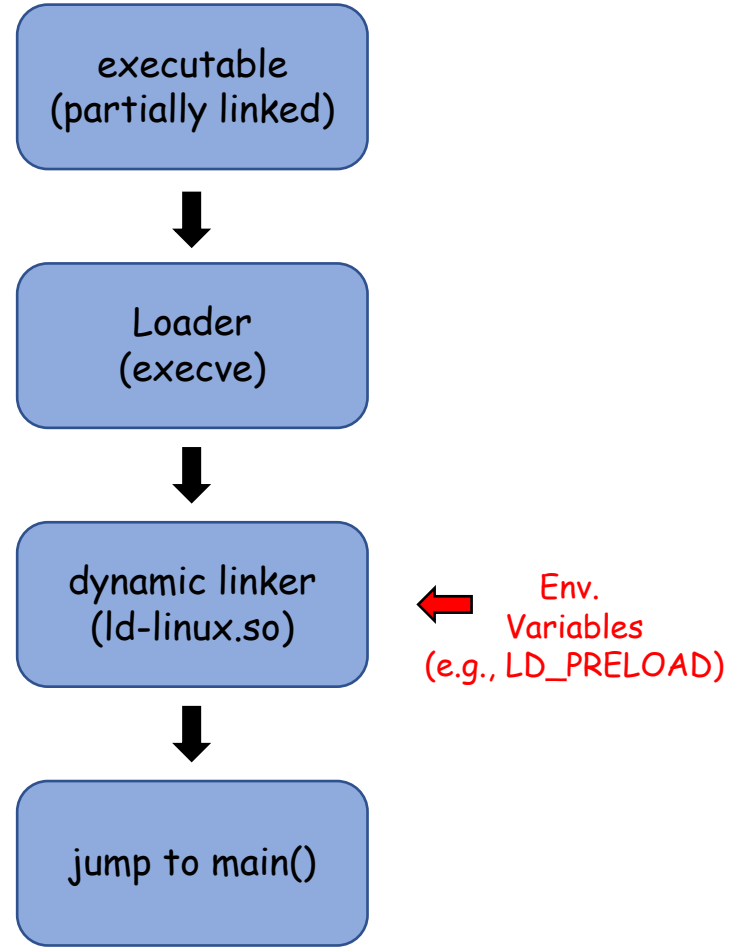
Make vulnerable program (compile, make root-owned set-uid program)

```
[seed@VM][~]$ gcc -o ls_vul ls_vul.c
[seed@VM][~]$ sudo chown root ls_vul
[seed@VM][~]$ sudo chmod 4755 ls_vul
[seed@VM][~]$ ls -al ls_vul
...

Q: How to get ls_vul to run attacker code for ls instead of /bin/ls program?!?!

[seed@VM][~]$ export PATH=/home/seed:$PATH # set PATH to look in seed's home dir first...
[seed@VM][~]$ echo $PATH
...

...and now...
```

Env. Attacks - Dynamic Linking

- Linking finds the external library code referenced in a program
- **Static Linking** — linker combines program code/external code into final executable
- **Dynamic Linking** — linker uses env. variables to locate external dependencies (increase the attack surface)

  -> how/where does the linker find libraries?!



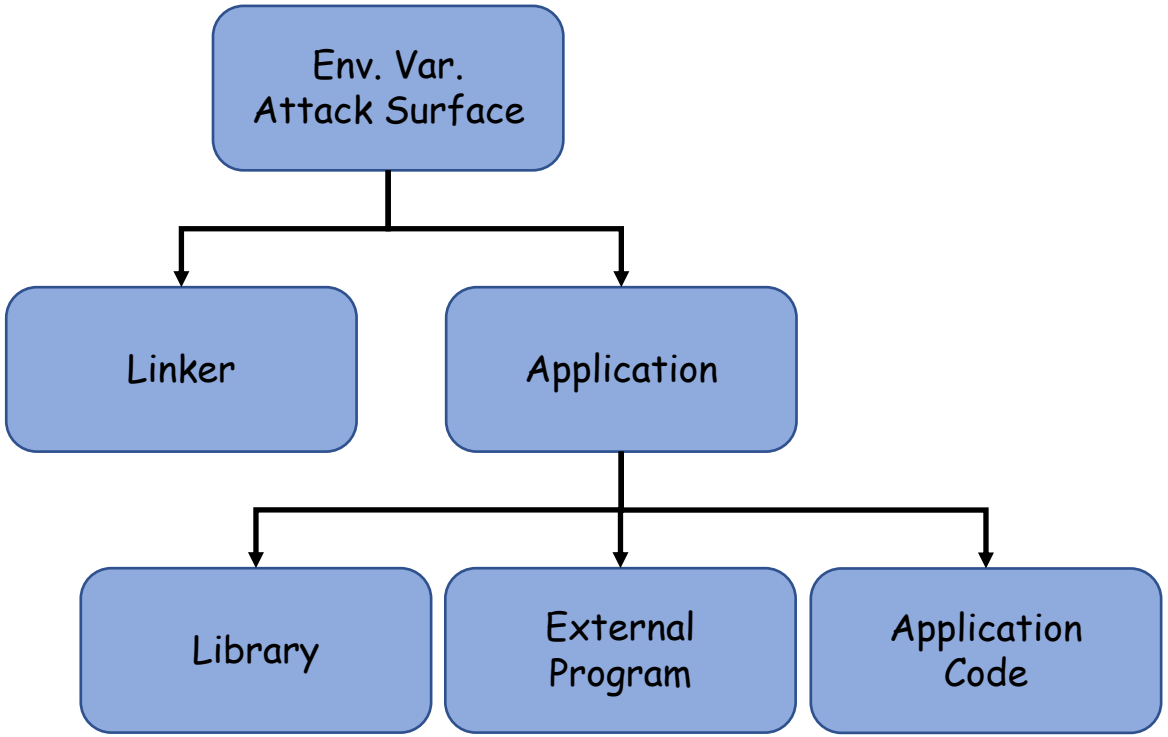
Env. Attacks - Dynamic Linking (cont.)

- LD_PRELOAD contains a list of shared libraries to search first
- If not all unresolved functions are found, the linker will search elsewhere, including the locations specified in LD_LIBRARY_PATH
- Both variables can be set by users, which provides an opportunity for users to influence linking!
- If the program were a Set-UID program…

Lab 01: Task 7

Going Further

- Can you devise similar attacks via environment variables?
  - Attack a program that calls an external program without specifying its absolute path (e.g., ls, cal)
  - Attack a program that depends on a function from a library (e.g., sleep()/printf()/etc. from libc)
  - Attack an app that directly uses env. variables



Case Study:

Capability Leaking

Capability Leaking

- In some cases, privileged programs **de-escalate** ("drop" or "downgrade") themselves during execution

- Example: The **su** program
  - The **su** program is a privileged Set-UID program
  - Allows one user to switch to another user (e.g., user*A* to userB)
  - Programs starts with EUID==root and RUID==user*A*
  - After password for userB is verified, EUID==RUID==userB (via privilege de-escalation)

- Programs that de-escalate privileges need to take care not to leak privileges i.e., programs must clean up privileged capabilities *before* dropping privileges

Capability Leaking (cont.)

- /etc/zzz file only writable by root
- The FD is created before dropping privileges
- Drop privileges
- Child does its work (w/ less privilege, but the write capability leaked!)

```
16      fd = open("/etc/zzz", O_RDWR | O_APPEND);
17      if (fd == -1) {
18          printf("Cannot open /etc/zzz\n");
19          exit(0);
20      }
21
22      // Simulate the tasks conducted by the program
23      sleep(1);
24
25      /*
26       * After the task, elevated privileges are no longer needed;
27       * it is time to relinquish these privileges!
28       * NOTE: getuid() returns the real UID (RUID)
29       */
30      setuid(getuid());
31
32      if (fork()) {  /* parent process */
33          close (fd);
34          exit(0);
35      } else {  /* child process */
36
37          /*
38           * Now, assume that the child process is compromised, and that
39           * malicious attackers have injected the following statements into this process
40           */
41          write (fd, "Malicious Data\n", 15);
42          close (fd);
43      }
44  }
```

How would you fix this program?

Summary

Important Security Principles

- **Principle of Isolation**
  - Don't mix **code** and **data**
  - Avoid using system() and unsafe variants of exec()
  - Sanitize inputs

- **Principle of Least Privilege**
  - Privileged programs should be given only the capabilities it requires
  - Disable privileges (temporarily/permanently) when they aren't needed
  - In Linux, use **seteuid()** and **setuid()** to disable/discard privileges

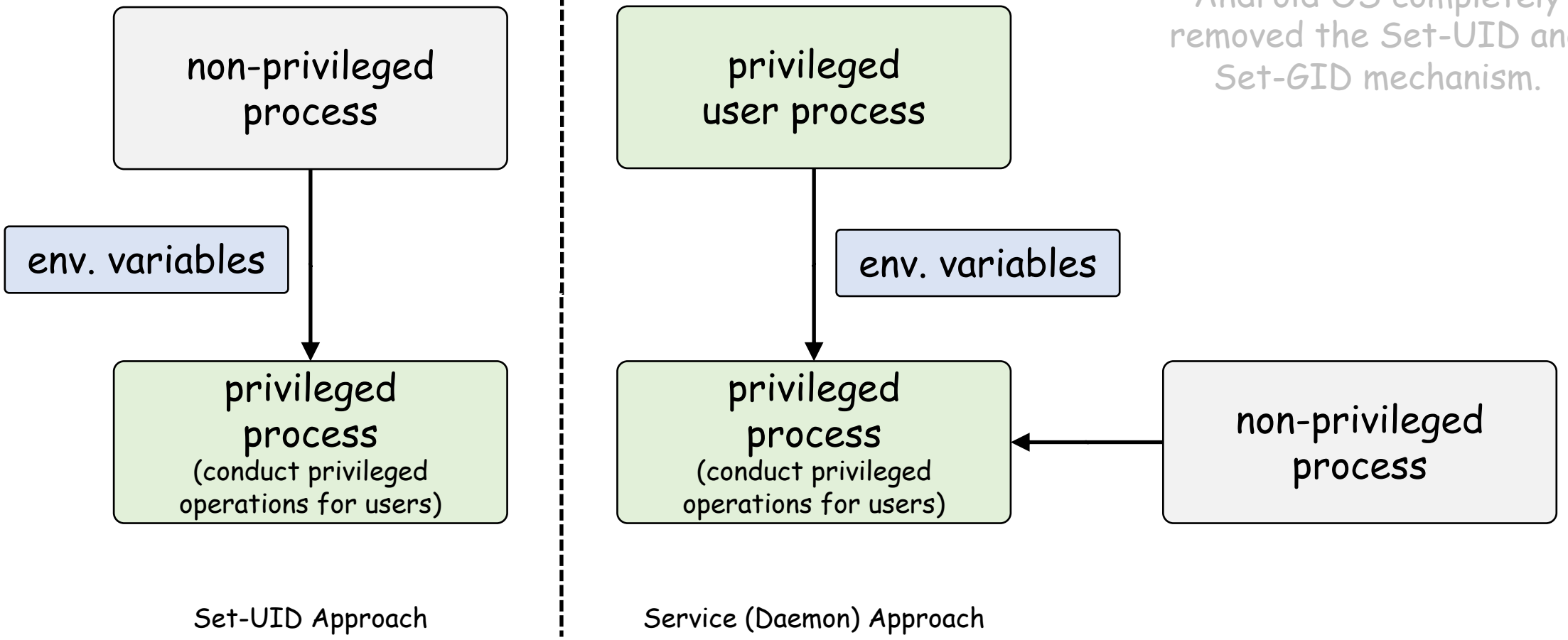
Practical Countermeasures

- Attack surface of execve() smaller than system()
  - execve() doesn't invoke a full shell!
  - when invoking external programs from within privileged programs, use execve()!

- When environment variables musts be used by privileged Set-UID programs, they ***must*** be sanitized properly.

- Developers may choose to use a secure versions of functions. E.g., replace **getenv()** with **secure_getenv()**.
  - getenv() -> search the env. variable list and return a pointer to the string found
  - secure_getenv() -> similar, but return NULL when "secure execution" is required (secure execution is defined by conditions, e.g., process's user/group EUID != RUID)

Set-UID Approach vs. Service (Daemon) Approach



Due to Set-UID issues, the Android OS completely removed the Set-UID and Set-GID mechanism.

Extra Practice!

1. Alice runs a Set-UID program that is owned by Bob. The program tries to read from **/tmp/x**, which is readable to Alice, but not to anybody else. Can this program successfully read from the file?

2. A process tries to open a file for read. The process's effective user ID is 1000, and real user ID is 2000. The file is readable to user ID 2000, but not to user ID 1000. Can this process successfully open the file?

3. The **chown** command automatically disables the **Set-UID** bit, when it changes the owner of a **Set-UID** program. Please explain why it does that.

4. Both **system()** and **execve()** can be used to execute external programs. Why is **system()** unsafe while **execve()** is safe?

5. A privileged **Set-UID** program needs to find out which directory it is currently in. There are two typical approaches. One is to use the **PWD** environment variable, which contains the full path of the current directory. Another approach is to use the **getcwd()** function (you can find its manual online). Please describe which approach you would take and why.

6. There are two typical approaches for letting normal users do privileged tasks, one is to write a **root-owned Set-UID program**, and let the user run; another approach is to use a **dedicated root daemon** to do those privileged tasks for users. Please compare the attack surfaces of these two approaches, and describe which one is more secure.