Software Security

Privileged Programs & Program Inputs: The Set-UID Mechanism & Environment Variables (Part I)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

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!

Today

Announcements

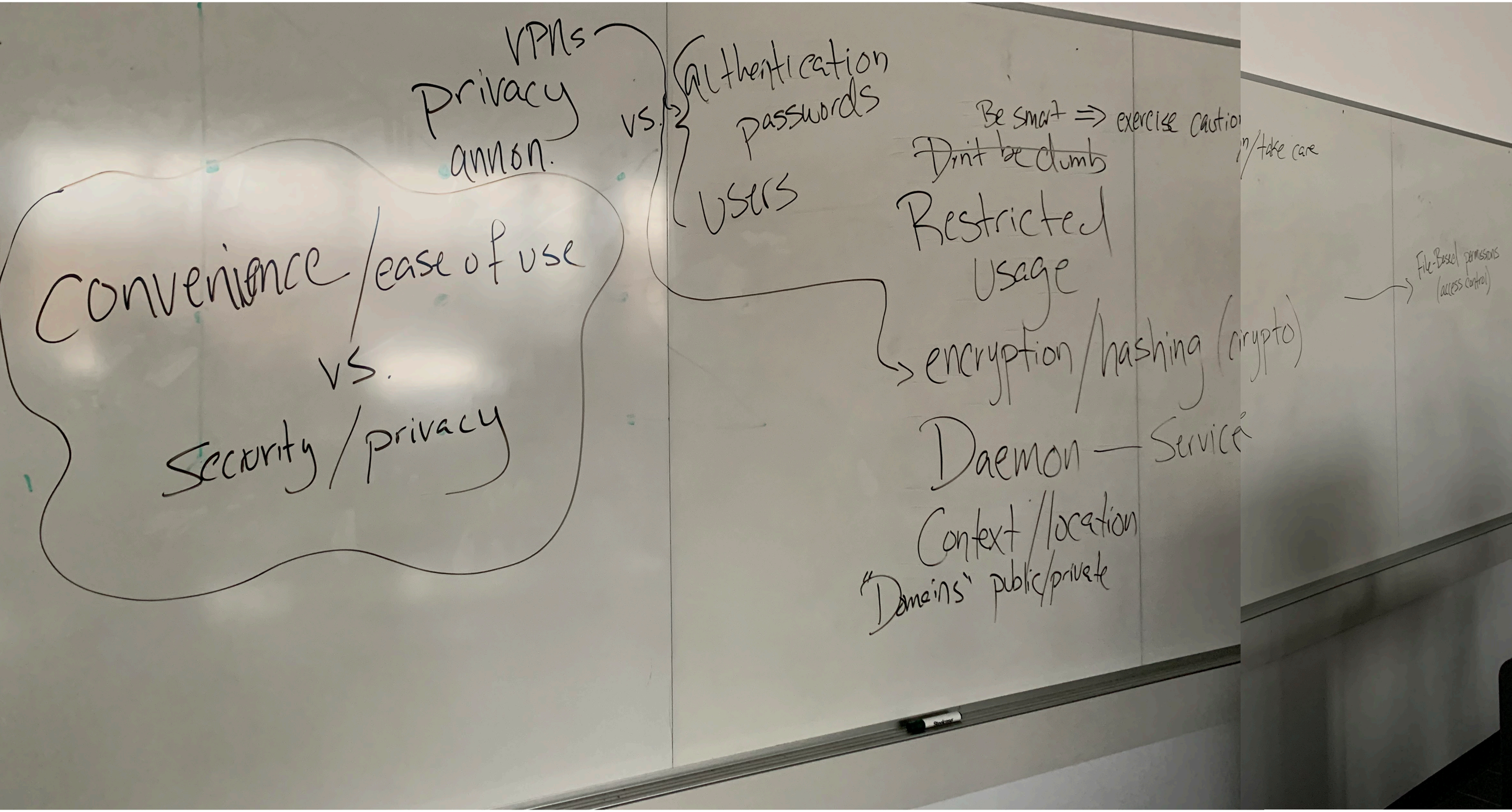
- We need a note taker for the class! ➡ ***Contact ODS if interested***
- Lab 00 ➡ ***It's up!***
- Lab 01 ➡ ***Will be posted Thursday***
- ***REMINDER:*** Bring your laptops (especially on Thursdays!)
- ***REMINDER:*** Sign-up for Slack ASAP!

Goals & Learning Objectives

- Wrap up review of basic Linux commands + basic ideas in Linux security
- Understand the need for privileged programs
- Understand how the Set-UID mechanism works + attack surface of (Set-UID) programs

How would you protect your computer & its resources?

Ideas?!


VPNs
Privacy
anon.
vs.
Authentication
passwords
users
Convenience / ease of use
vs.
Security / privacy
Be smart ⇒ exercise caution
~~Don't be dumb~~
Restricted
usage
encryption / hashing (crypto)
Daemon — Service
Context / location
"Domains" public/private
in/take care
File-Based permissions
(access control)

How would you protect your computer & its resources?

Modeling and managing system security the UNIX-y way: file permissions & access control

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

users/groups

Need notions of "identity"

A human?

Groups of humans?

A service?

An administrator? (e.g., "root")

objects

Usually *things* on a *filesystem*

(e.g., programs, data)

permissions (read/write/execute)

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

Read the file?

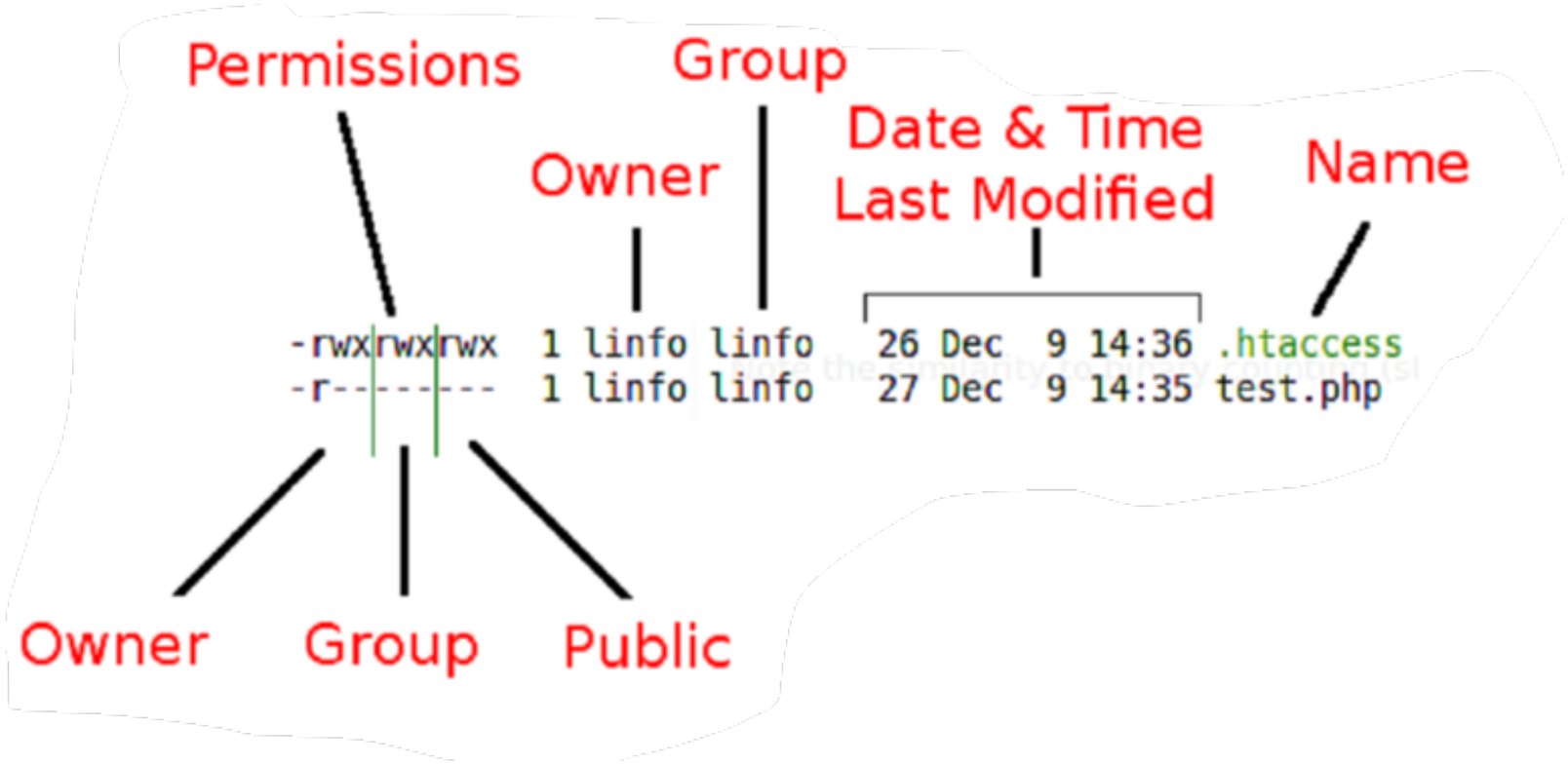
Write to it/modify it?

Run it?

How would you protect your computer & its resources?

Modeling and managing system security the UNIX-y way: file permissions & access control

Every file has…



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:

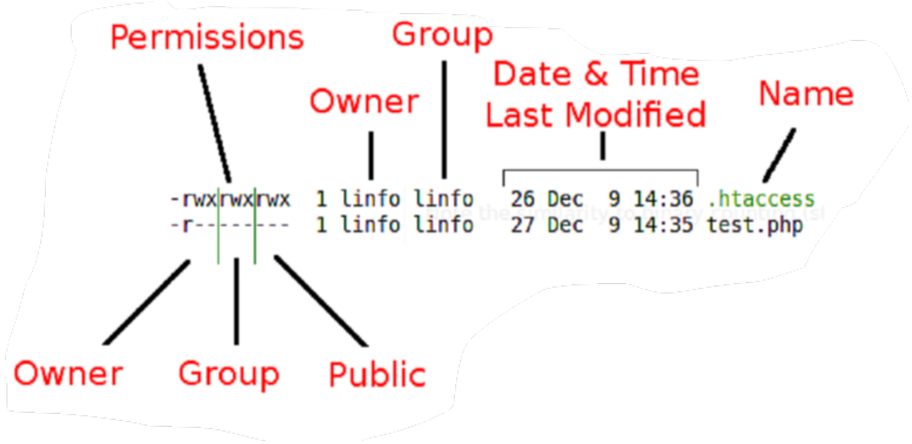
1. Is **A** the owner of **F**? >>> use **owner permissions** to decide whether A can do operation O.

   A is not F's owner

2. Is **A** a member of **F's** group? >>> use **group permissions** to decide…

   A is not F's owner or a member of F's group

3. >>> use the **"everyone else" / "others" permissions** to decide…



Some in class exploration — let's take a quick look at these ideas in a VM.

The Limitations of File-Based Access Control

Question

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

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

Only writable to the owner

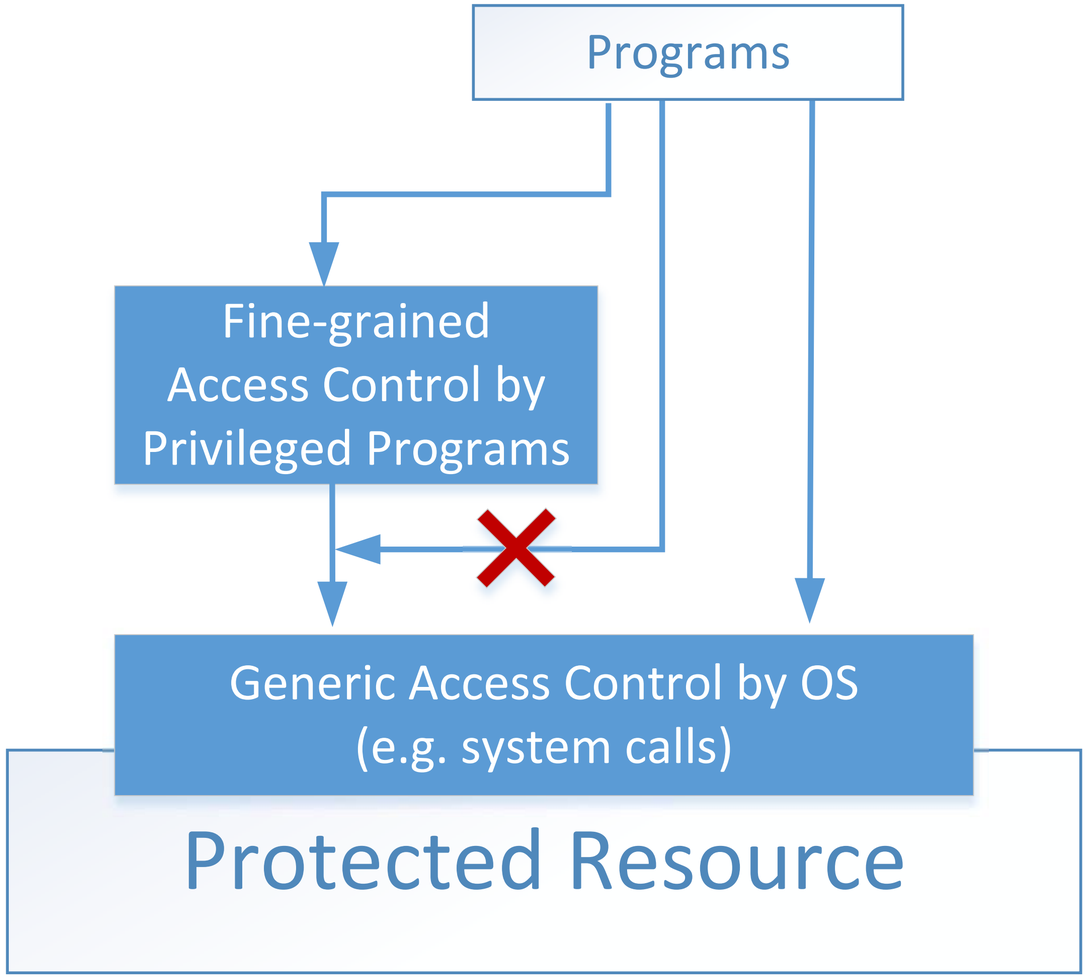
- **Idea:** grant access to user 'champ' to edit /etc/shadow
  Good idea or bad idea?

- **Better Idea:** grant access to user 'champ' to edit ONLY its own entry in /etc/shadow
  Good idea or bad idea?

- **BETTER Idea:** let user 'champ' run a special program that has one purpose:
  to edit the /etc/shadow file with inputs to the program
  Good idea or bad idea?

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" 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
  - 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 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

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Jan 21  2020 /usr/bin/passwd
```

Set-UID In A Nutshell *(cont.)*

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

- When a normal program is executed,

  RUID == EUID (== 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! *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:~/csci_sandbox$ cp /bin/cat ./mycat
seed@VM:~/csci_sandbox$ sudo chown root mycat
seed@VM:~/csci_sandbox$ ls -al mycat
-rwxr-xr-x 1 root seed 51036 Jan 21 02:02 mycat
```

- **Before** enabling the Set-UID bit

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

- **After** enabling the Set-UID bit

```
seed@VM:~/csci_sandbox$ sudo chmod 4755 mycat
seed@VM:~/csci_sandbox$ mycat /etc/shadow
root:$6$NrF4601p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEE
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
```

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

```
seed@VM:~/csci_sandbox$ sudo chmod 4755 myid
seed@VM:~/csci_sandbox$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
```

Is Set-UID Secure?

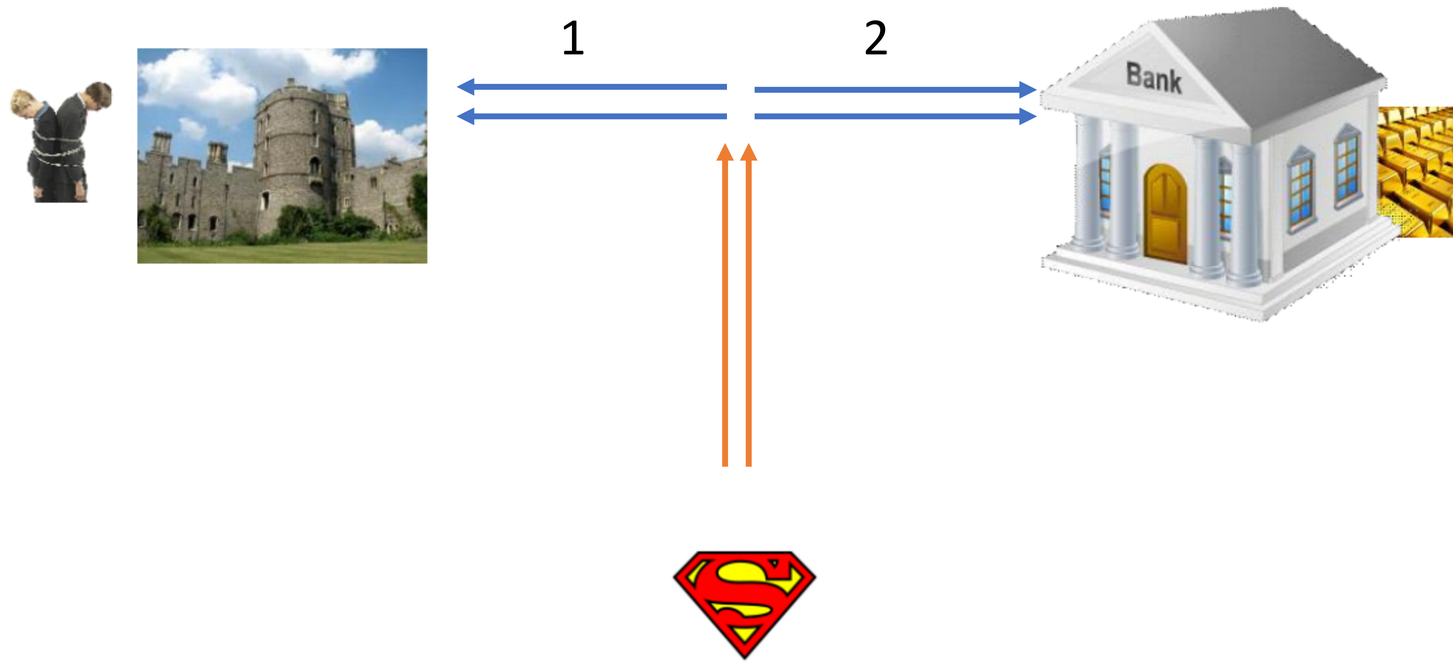
- Allows normal users to escalate privileges
  - This is different from directly giving escalated privileges (e.g., **`sudo`**)
  - Restricted behavior — similar to superman's power suit

- Unsafe to run all programs as Set-UID programs……

  Examples?
  - `/bin/sh` — *why?*
  - `vi(m)` — *why?*

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

- Shouldn't assume that user can only do whatever is coded…
- ***Recall Superman: 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?



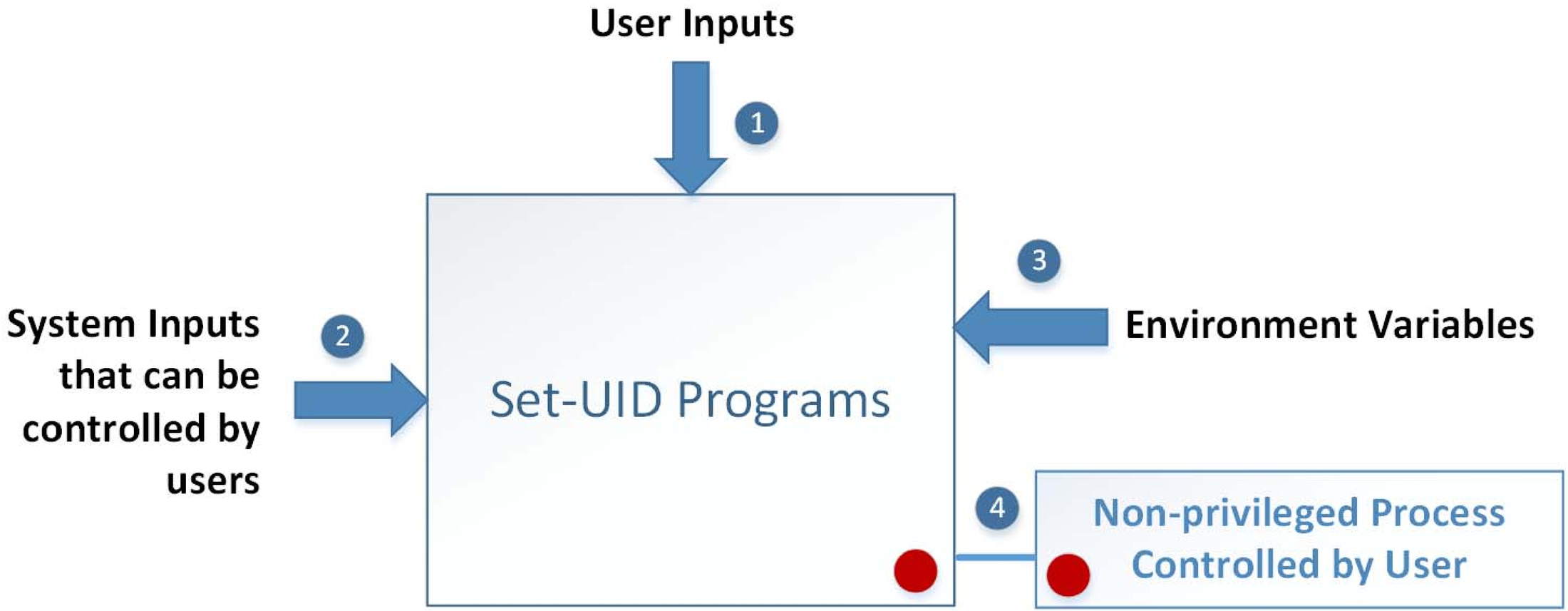
Software Security

Privileged Programs & Program Inputs: The Set-UID Mechanism & Environment Variables (Part II)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

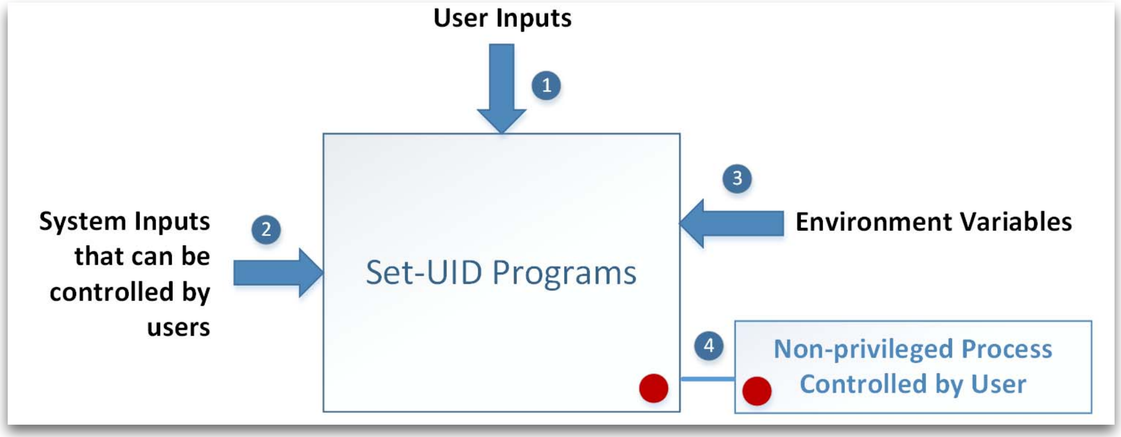
Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

Attack Surface of (Set-UID) Programs



Attacks via Program Inputs

- User Inputs (Explicit Inputs)
  - Buffer Overflow (Chapter 4)
    - Overflowing a buffer can lead to malicious code execution
  - Format String Vulnerabilities (Chapter 6)
    - Format strings can be exploited to change program behavior



- System Inputs
  - Race Condition Vulnerabilities (Chapter 7)
    - Symbolic links to privileged files from an unprivileged file

- Environment Variables (Hidden Inputs; Chapter 2)
  - Program behavior can be influenced by inputs that are not visible inside a program
  - Environment variables can be set by a user prior to program execution (e.g., **PATH**)

- Capability Leaking ***(next slides…)***

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., userA to userB)
  - Programs starts with `EUID==root` and `RUID==userA`
  - 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

Example: Attacks via Capability Leaking

- The /etc/zzz file is only writable by root
- The FD is created before dropping privileges
- Drop privileges
- Invoke a shell program with restricted privileges

```
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);
```

https://github.com/traviswpeters/csci476-code/blob/master/01_setuid/cap_leak.c

Example: Attacks via Capability Leaking *(cont.)*

What's the problem?

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

```
$ 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 ccccccccccc >& 3            ← Using the leaked capability
$ exit
$ cat /etc/zzz
bbbbbbbbbbbbbb
ccccccccccc                        ← File modified
```

How would you fix this program?

Close the FD before dropping privileges and turning over control :-)

Invoking Programs… From Within Programs…

—http://meetingsnorthwest.com/wp-content/uploads/2016/03/mind-blown_shortcuts.png

Invoking Programs

- It is possible to invoke external commands from inside a program
  - `system()`
  - `exec()`-family

- The external command is *(should be…)* chosen by the Set-UID program…
  - …users are not supposed to provide the command
  - …***but*** users are often asked to provide input data to commands
  - If the command is not invoked properly,
    the user's input ***data*** may be interpreted as ***code***

Invoking Programs the Unsafe Way — `system()`

```
$ 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
```

```
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 ;
}
```

↳ i.e., **`system("/bin/catall FILENAME");`**

https://github.com/traviswpeters/csci476-code/blob/master/01_setuid/catall.c

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

Question:

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

Invoking Programs the Unsafe Way — `system()` *(cont.)*

RECALL: **CHANGE DEFAULT SHELL ~~>**

```
seed@VM:~/csci_sandbox$ sudo ln -sf /bin/zsh /bin/sh
```

PROBLEM

part of the ***data***

is interpreted as ***code***

NOTE: *Page 18 describes a workaround for a countermeasure in the Ubuntu16.04 VM. Commands shown here to link/reset the shell used by* `system()`.

```
$ 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

RECALL: **RESET SHELL ~~>**

```
seed@VM:~/csci_sandbox$ sudo ln -sf /bin/dash /bin/sh
```

Invoking Programs the Safe(r) Way — execve( )

```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
```

```
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 ;
}
```

https://github.com/traviswpeters/csci476-code/blob/master/01_setuid/safecatall.c

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

Command name is provided here (by the program)

Input data are provided here (can be by user)

Question: Why is this considered safe?

Code (the command name) and data (inputs to the command) are clearly separated

Invoking Programs the Safe(r) Way — `execve()` *(cont.)*

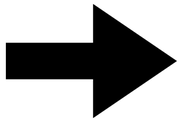
```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

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

Input data is treated as data, not as code ✓

Invoking Programs the Safe(r) Way — `execve()` *(cont.)*

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

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



All duplicate the actions of the shell, and can be attacked using the **PATH** environment variable (later…)

Invoking External Commands in Other Languages

- The ability (and risks) of invoking external commands is not limited to C programs
- 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
  print("Please specify the path of the directory");
  print("<p>");
  $dir=$_GET['dir'];
  print("Directory path: " . $dir . "<p>");
  system("/bin/ls $dir");
?>
```

Possible Attack ➡

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

Take-Aways

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

- **Principle of Least Privilege**
  - A privileged program should be give only the power it requires to perform its task(s)
  - Disable privileges (temporarily/permanently) when they aren't needed
  - In Linux, use `seteuid()` and `setuid()` to disable/discard privileges

Environment Variables

Environment Variables — *What Are They?!*

- A set of dynamic named values (key=value pairs); part of the environment in which a process runs.
  - Useful commands: `env, printenv, set, unset, export, echo`…
  - Useful env. variables: `USER, HOME, EDITOR, SHELL, PATH, LANG`…
- Affect the way that running process will behave
- Introduced in UNIX and also adopted by Microsoft Windows
- Example: The **`PATH`** variable
  - How does the shell know where to find commands?!
  - For instance, you typically say “`ls`” not “`/bin/ls`”
  - If the full command is not provided, the shell process will use the `PATH` env. variable to search for it

Environment Variables — *How Do I Access Them!*

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (envp[i] !=NULL) {
      printf("%s\n", envp[i++]);
   }
}
```

From the **`main()`** function

More reliable way:
Using the global **`environ`** variable

```
#include <stdio.h>

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

Environment Variables — *How Does A Process Get Them?*

- Processes can get environment variables in one of two ways:

    1. If a new process is created using the **`fork()`** syscall,
       the child process inherits its parent process's environment variables.

    2. If a process runs a new program in itself, typically by way of the **`execve()`** syscall,
       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()`:

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

Example: Environment Variables & `execve()`

- This program executes another program `/usr/bin/env`, which prints out the environment variables of the current process.
- We construct a new variable `newenv`, and use it as the 3rd argument passed to `execve()`.

```
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);
  }
}
```

https://github.com/traviswpeters/csci476-code/blob/master/02_envvars/passenv.c

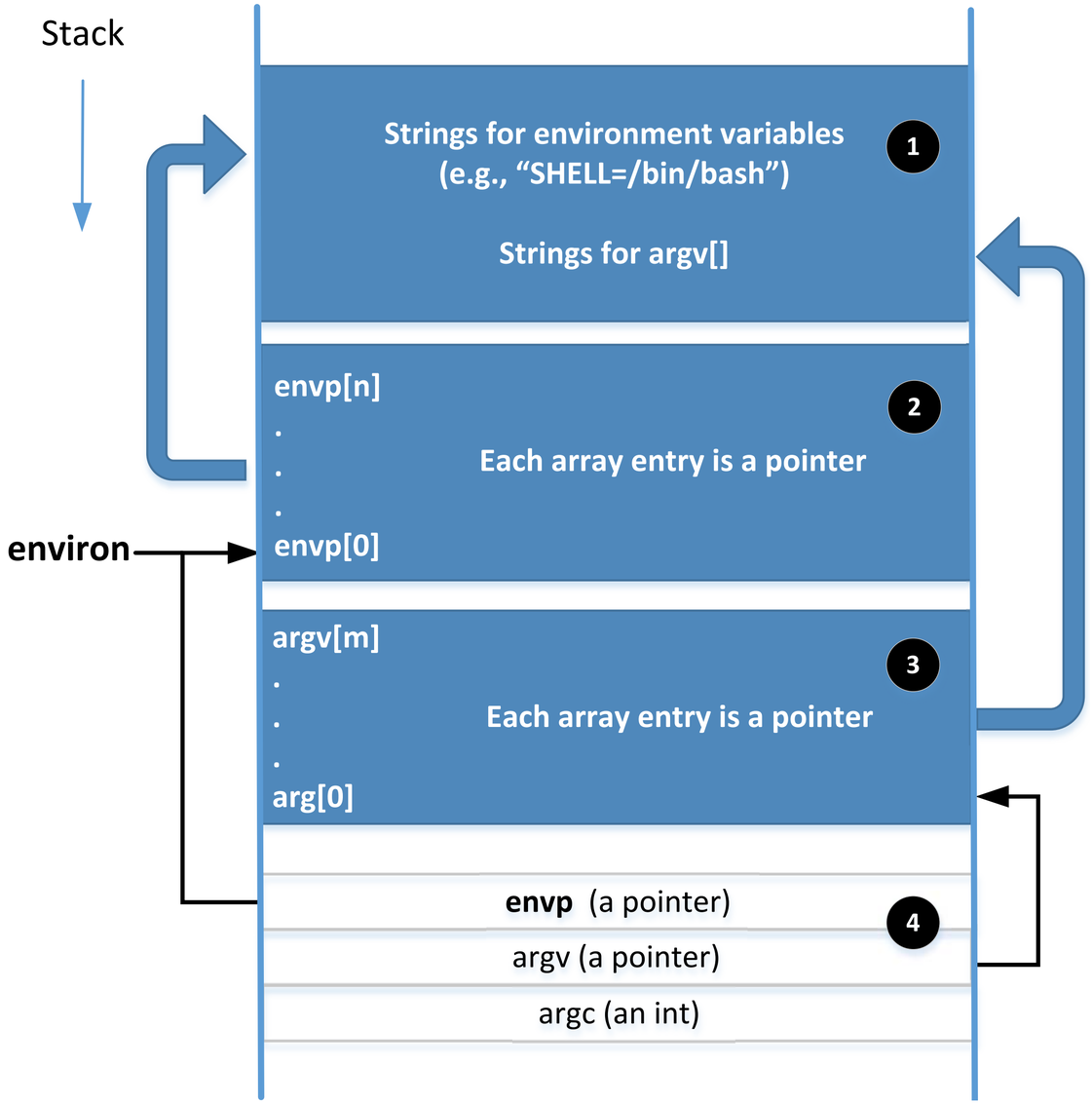
Example: Environment Variables & `execve()` *(cont.)*

Obtained from the parent process

```
$ a.out 1     ← Passing NULL
$ a.out 2     ← Passing newenv[]
AAA=aaa
BBB=bbb
$ a.out 3     ← Passing environ
SSH_AGENT_PID=2428
GPG_AGENT_INFO=/tmp/keyring-l2UoOe/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=6da3e071019f...
WINDOWID=39845893
OLDPWD=/home/seed/Book/Env_Variables
...
```

Environment Variables — *Where Are They In Memory?*

- `envp` and `environ` point to the same location in memory (initially)
- `envp` is only accessible inside `main()`, while `environ` is a global variable.
- When changes are made to environment variables (e.g., add new var.), the location for storing environment variables may be moved to the heap — `environ` will change (`envp` does not change)



Aside: The `/proc` File System

- /proc is a virtual file system in Linux
- Contains a directory for each process — named by the Process ID (PID)
- Each directory has a virtual file called environ, which contains the environment of the process
  - e.g., /proc/932/environ contains environment variables for process 932
  - e.g., strings /proc/$$/environ prints out the env. variables of the current process
    - $$ == shorthand for current Process ID
- When env program is invoked in a (bash) shell, it runs in a child process.
  Thus, it prints out the environment variables of the shell's child process, not its own.

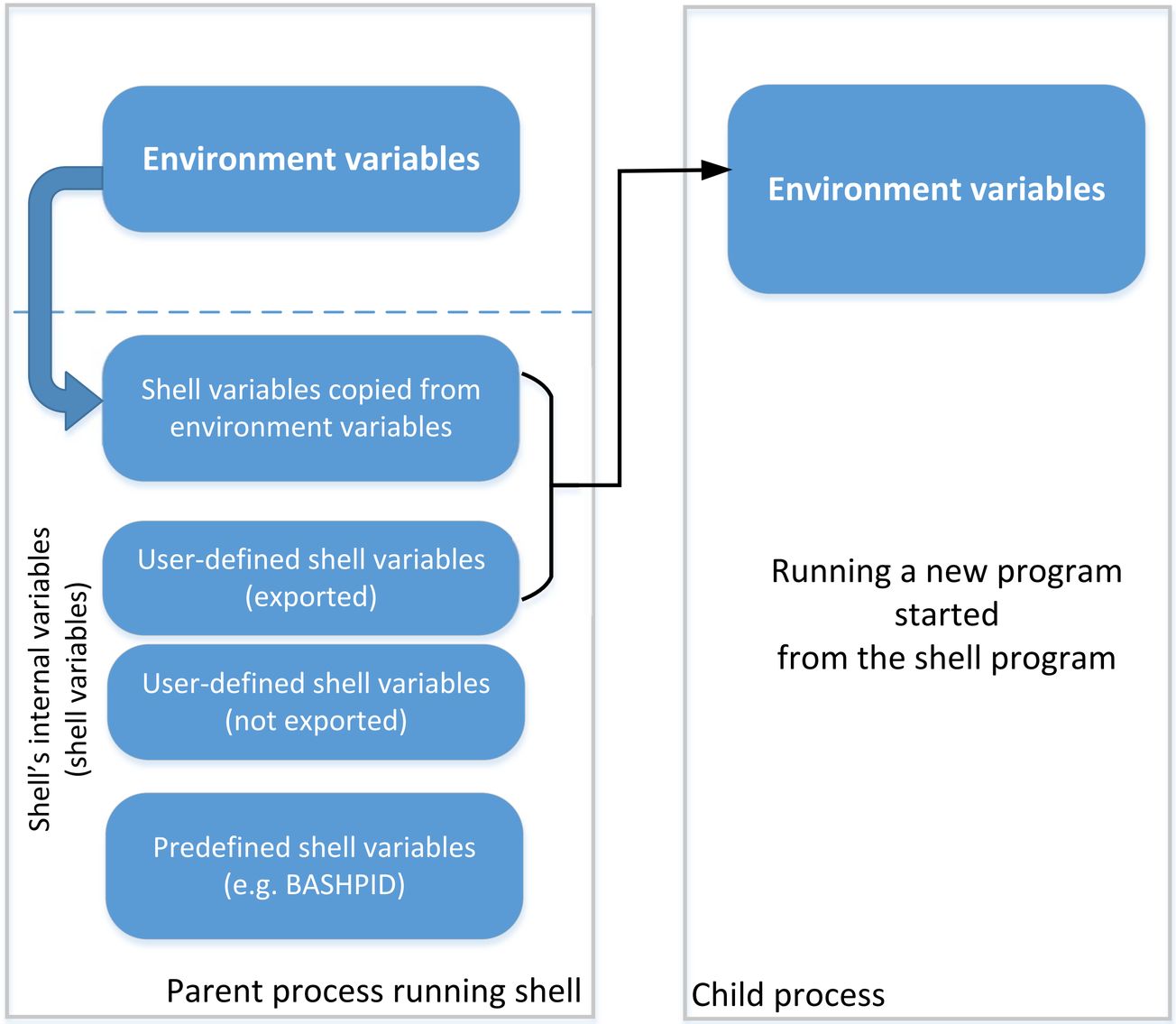
Shell Variables vs. ***Environment*** Variables

Environment variable ➡
Shell variable ➡
Shell variable is changed ➡
Environment variable is the same ➡
Shell variable is gone ➡
Environment variable is still here ➡

```
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ echo $LOGNAME
seed
seed@ubuntu:~/test$ LOGNAME=bob
seed@ubuntu:~/test$ echo $LOGNAME
bob
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ unset LOGNAME
seed@ubuntu:~/test$ echo $LOGNAME

seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
```

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.*

Shell Variables vs. ***Environment*** Variables *(cont.)*



Shell Variables vs. ***Environment*** Variables *(cont.)*

When we type **env** in the shell, the shell will create a child process

Print out environment variable

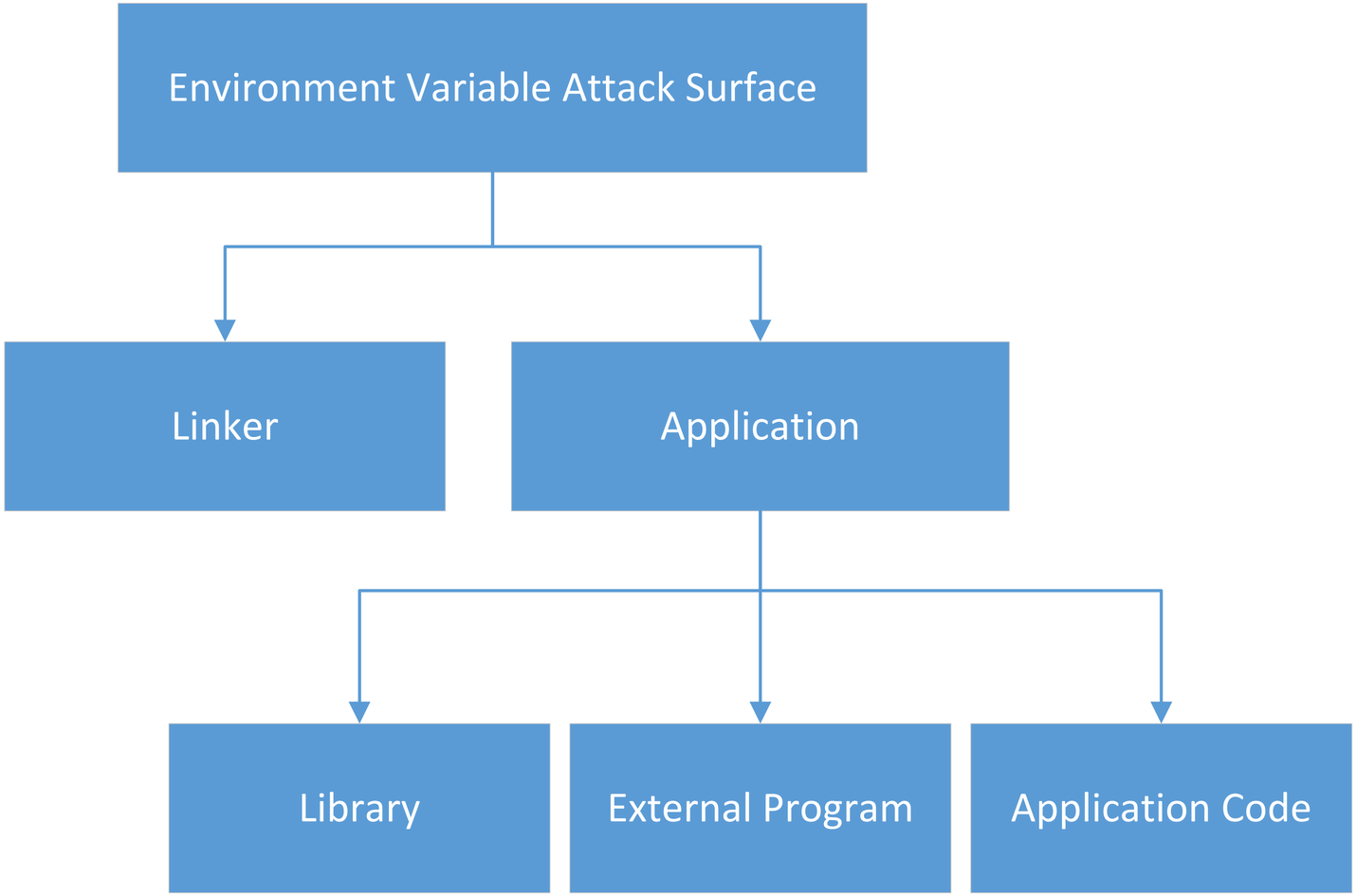
Only LOGNAME and LOGNAME3 get into the child process, but not LOGNAME2. Why?

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

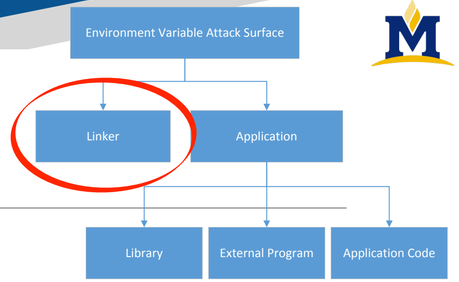
Environment Variables & Attacks

Attack Surface of 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!



Attacks via Linker



- Linking finds the external library code referenced in a program
- Linking can be done during runtime or at compile time
  - 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)

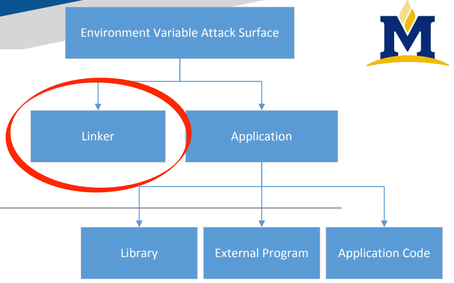
```
#include <stdio.h>
int main(void)
{
   printf("hello world\n");
   return 0;
}
```

https://github.com/traviswpeters/csci476-code/blob/master/02_envvars/hello.c

```
seed@VM:$ gcc hello.c -o hello_dynamic
seed@VM:$ gcc hello.c -o hello_static -static
seed@VM:$ ls -l hello_*
-rwxr-xr-x 1 seed seed   7344 Jan 23  2020 hello_dynamic
-rwxr-xr-x 1 seed seed 728292 Jan 23  2020 hello_static
```

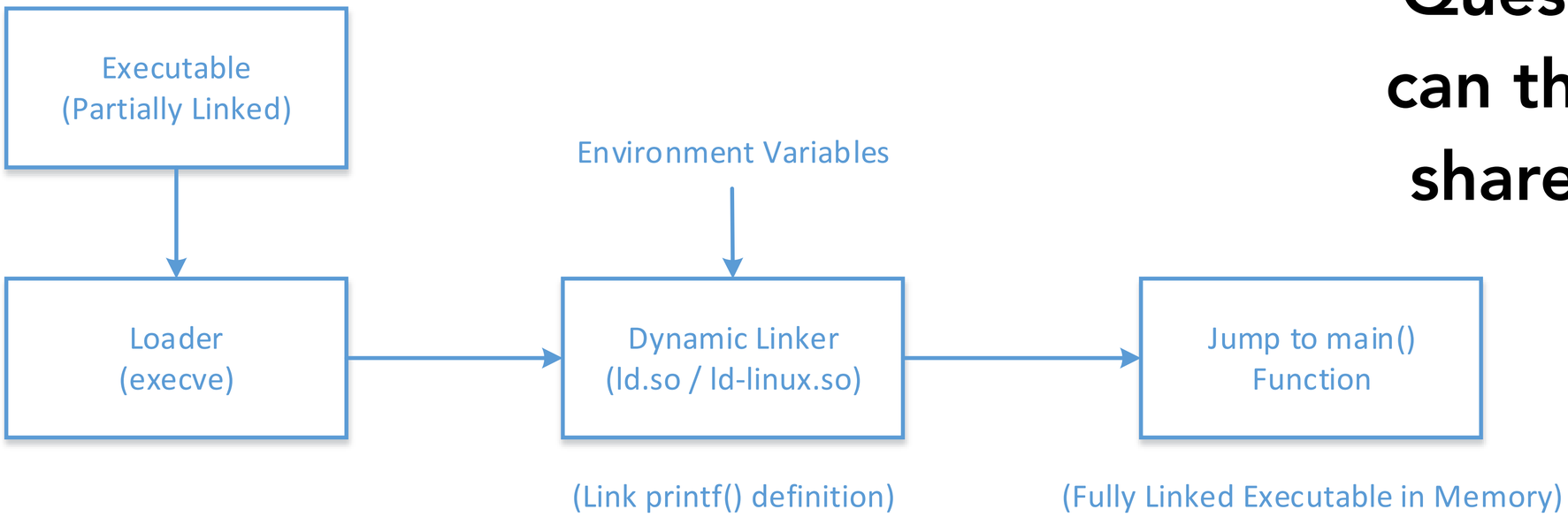
What are advantages/disadvantages of each of these approaches?

Attacks via Dynamic Linker

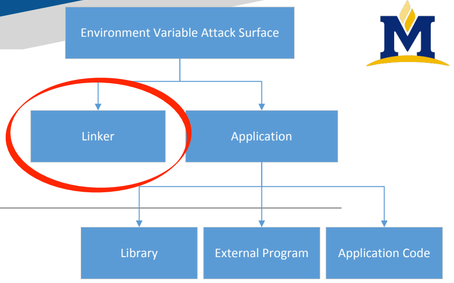


- In dynamic linking, **linking** is done @ runtime
  (Shared Libraries in Linux, DLLs in Windows)
- Before a program compiled w/ dynamic linking is run, its executable is **loaded** into memory
- There are missing references to code though…
- The **linker** finds the implementation of code in **shared libraries** (e.g., `printf()`)
- Once linking is done, control is given to `main()`

Question: Where can the linker find shared libraries?



Attacks via Dynamic Linker



We can use the `ldd` command to see what shared libraries a program depends on:

```
$ ldd hello_static
   not a dynamic executable
$ ldd hello_dynamic
   linux-gate.so.1 =>  (0xb774b000)
   libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb758e000)
   /lib/ld-linux.so.2 (0xb774c000)
```

for system calls

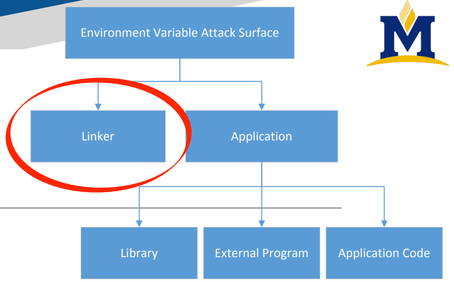
The dynamic linker itself is in a shared library. It is invoked before the main function gets invoked.

The libc library (contains functions like printf() and sleep())

Take-Aways

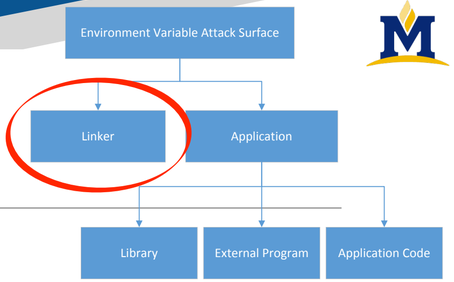
- Dynamic linking saves memory…
  …but this means that a part of the program's code is undecided during compile time
- If the user can influence the missing code, they can compromise the integrity of the program!

Attacks via Dynamic Linker: Case Study



- **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…

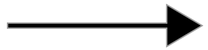
Attacks via Dynamic Linker: Case Study *(cont.)*



<Normal Program>

- Program calls the `sleep()` function, which is dynamically linked:

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

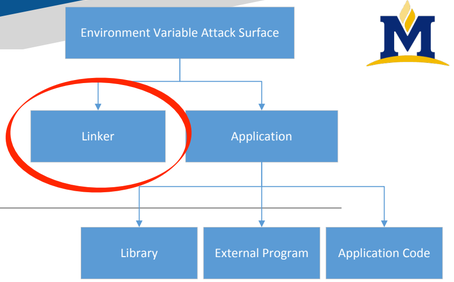


```
seed@ubuntu:$ gcc mytest.c -o mytest
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

- 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: Case Study *(cont.)*

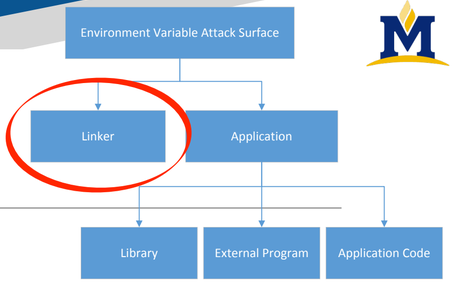


<Normal Program>

- *To get our sleep() function to be called, we need to…*
  - compile our 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: Case Study *(cont.)*



<Set-UID Program>

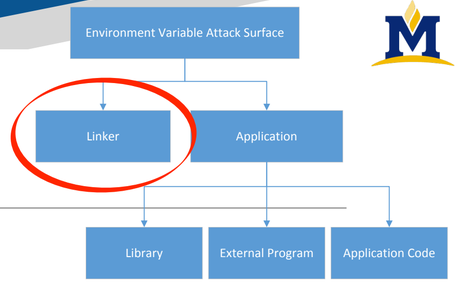
- If the technique in example 1 works on Set-UID programs, that would be *very* dangerous! Let's see…

```
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:$
```

- *Hmm… our sleep() function was not invoked…*

- This is due to a countermeasure implemented by the dynamic linker. If the linker sees that the EUID and RUID differ, it ignores the LD_PRELOAD and LD_LIBRARY_PATH env. variables.

Attacks via Dynamic Linker: Case Study *(cont.)*



<Set-UID Program>

- For fun, let's verify the effectiveness of the countermeasure.
  Make a copy of the `env` program and make it a Set-UID program.

```
seed@ubuntu:$ cp /usr/bin/env ./myenv
seed@ubuntu:$ sudo chown root myenv
seed@ubuntu:$ sudo chmod 4755 myenv
seed@ubuntu:$ ls -l myenv
-rwsr-xr-x 1 root seed 22060 Dec 27 09:30 myenv
```

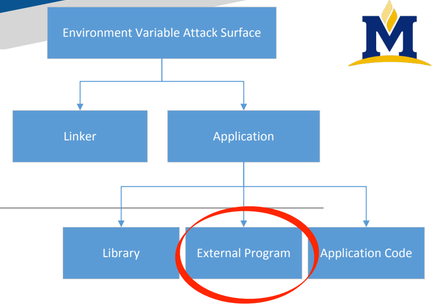
- Now, export `LD_PRELOAD` and `LD_LIBRARY_PATH` and run both programs.

Run the original `env` program

Run our `env` program

```
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ export LD_LIBRARY_PATH=.
seed@ubuntu:$ export LD_MYOWN="my own value"
seed@ubuntu:$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
LD_MYOWN=my own value
seed@ubuntu:$ myenv | grep LD_
LD_MYOWN=my own value
```

Attacks via External Program



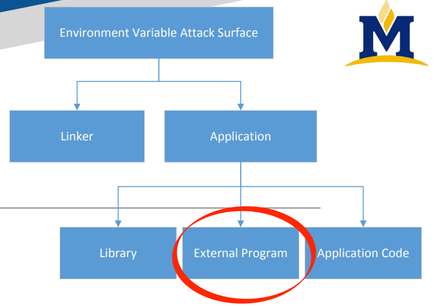
- An application may invoke an external program

- The application itself may not use environment variables, ***but the invoked external command might!***

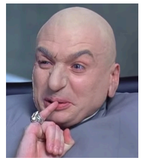
- Recall the typical ways of invoking external programs:
  - `exec()`-family, which calls `execve()` to run the program directly
  - `system()`, which calls `execl()`, which calls `execve()` to run `/bin/sh` to run the program

- The attack surfaces differ for these approaches, but here we look at how environment variables can be manipulated to exploit (privileged programs)

Attacks via External Program: Case Study



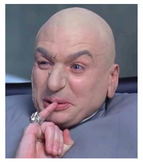
- The behavior of the shell program is affected by many environment variables; the most common of which is the **PATH** variable.
- When a shell program runs a command where the absolute path is not provided, it uses the **PATH** variable to locate the command.



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

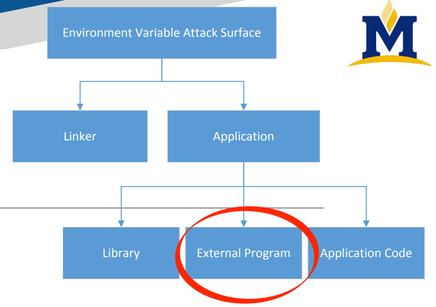
Consider a program that uses system() to invoke the cal program. Here, the full path is not provided….

I think I'll go ahead and get "vul" to give me a shell…



```
/* our malicious "cal" program */
#include <stdlib.h>
int main()
{
    system("/bin/bash -p");
}
```

Attacks via External Program: Case Study




```
seed@VM:$ gcc -o vul vul.c
seed@VM:$ sudo chown root vul
seed@VM:$ sudo chmod 4755 vul
seed@VM:$ vul
    January 2020
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@VM:$ gcc -o cal cal.c
seed@VM:$ export PATH=.:$PATH
seed@VM:$ echo $PATH
.:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
seed@VM:$ vul
# id
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

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

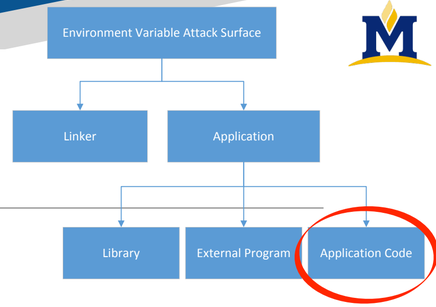
② We now change the PATH environment variable

Got a root shell!

Take-Aways

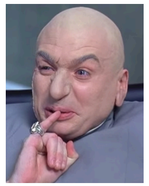
- Compared to `system()`, `execve()`'s attack surface is smaller
  - It doesn't invoke a full shell, so it isn't affected by env. variables
- When invoking external programs from within privileged programs… **use `execve()`**!

Attacks via Application Code



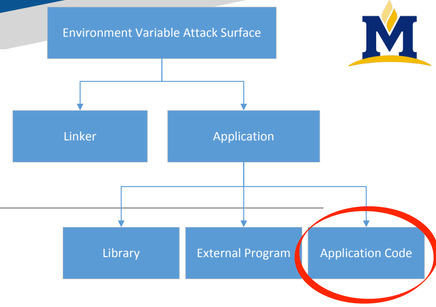
Some programs may ***directly*** use environment variables.

Similar to other case studies, if these programs are privileged programs, this provides an opportunity for users to inject untrusted inputs…



```c
/* 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;
}
```

Attacks via Application Code



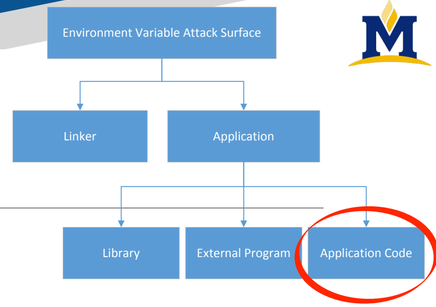
- The program uses `getenv()` to know its current directory from the `PWD` environment variable
- The program then copies this into an array `arr`, but forgets to check the length of the input. This results in a potential buffer overflow.
- Value of `PWD` comes from the shell program, so every time we 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 & Take-Aways

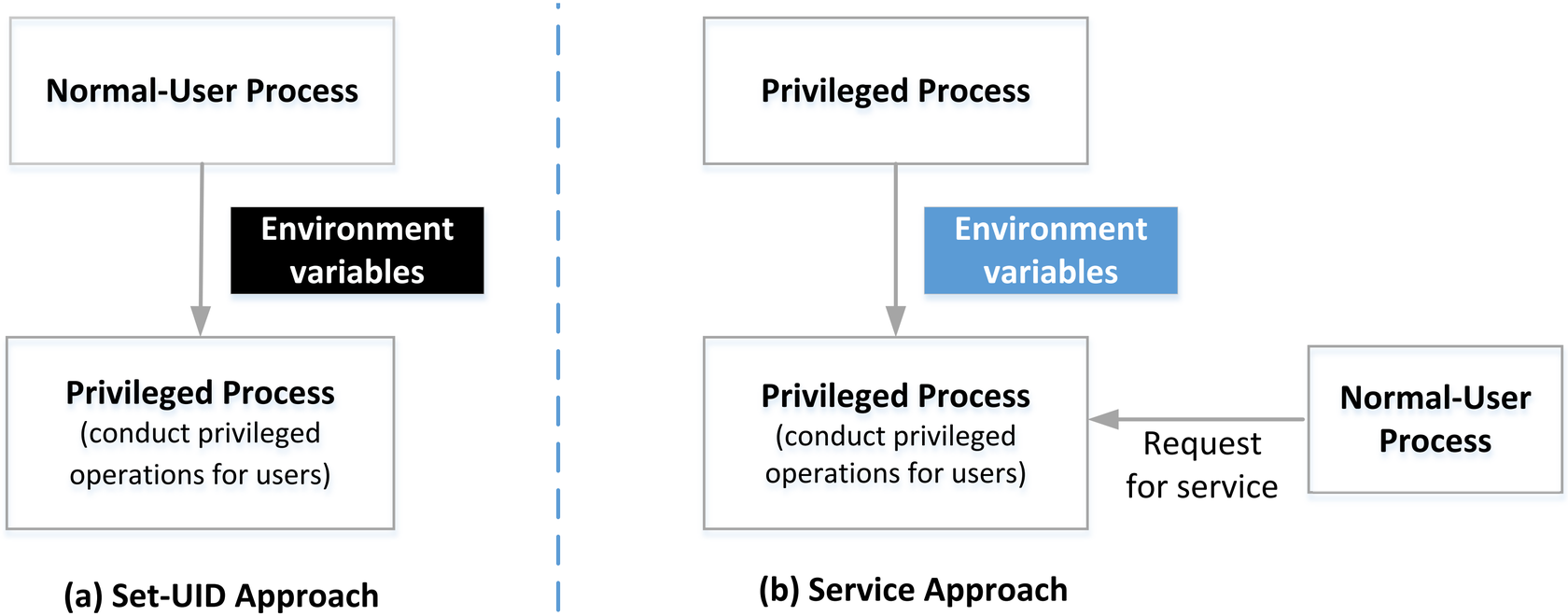
- When environment variables are used by privileged Set-UID programs, they ***must*** be sanitized properly.

- 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
  - `secure_getenv()` works similarly, except it returns NULL when "secure execution" is required
  - Secure execution is defined by conditions like when the process's user/group EUID != RUID

Wrapping Up: Set-UID Approach vs. Service (Daemon) Approach

- Most OSs offer two approaches to allow normal users to perform privileged operations:
  - **Set-UID Approach:** normal users invoke a special program to gain elevated privileges temporarily.
  - **Service Approach:** normal users make a request to a privileged service to perform actions for them.
- Set-UID has a larger attack surface, largely due to environment variables
  - Env. variables **should not be trusted** under the Set-UID approach
  - Env. variables **can be trusted** under the Service approach

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



You Try!

Exam-like problems that you can use for practice!

- 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?

- 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?

- 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.

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

- 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.

- 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.