# Environment Variables
# &
# Attacks

# 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.

# 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);
  }
}
```

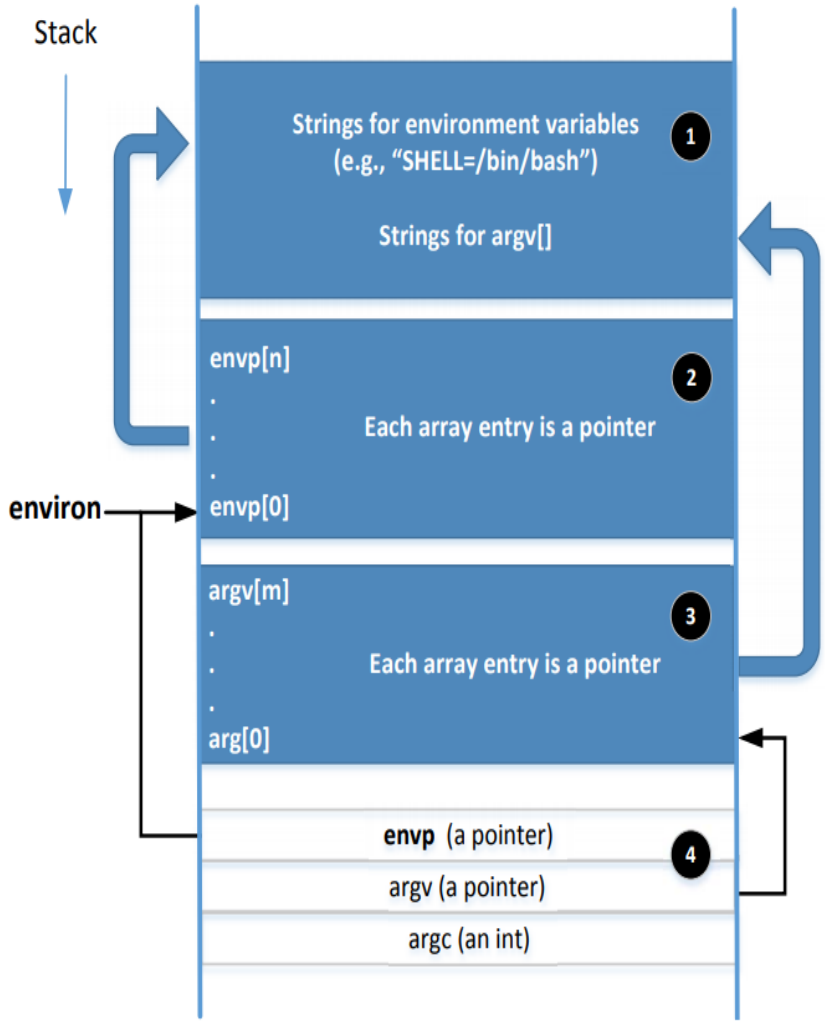
# execve() and Environment variables

```
$ 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
...
```

Obtained from the parent process

# 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'll 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

Shell variable

Shell variable is changed

Environment variable is the same

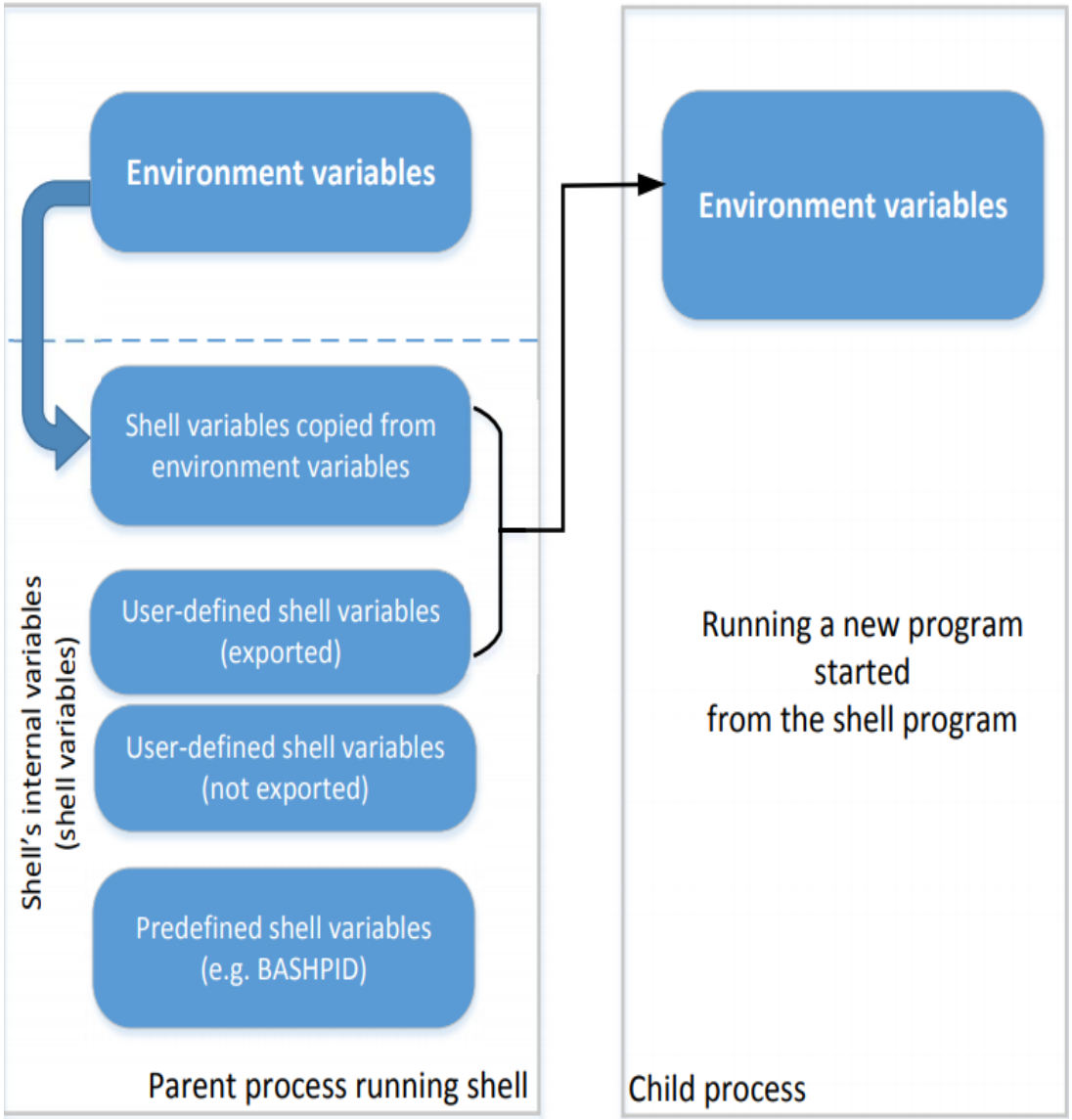
Shell variable is gone

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
```

# 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 becomes 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

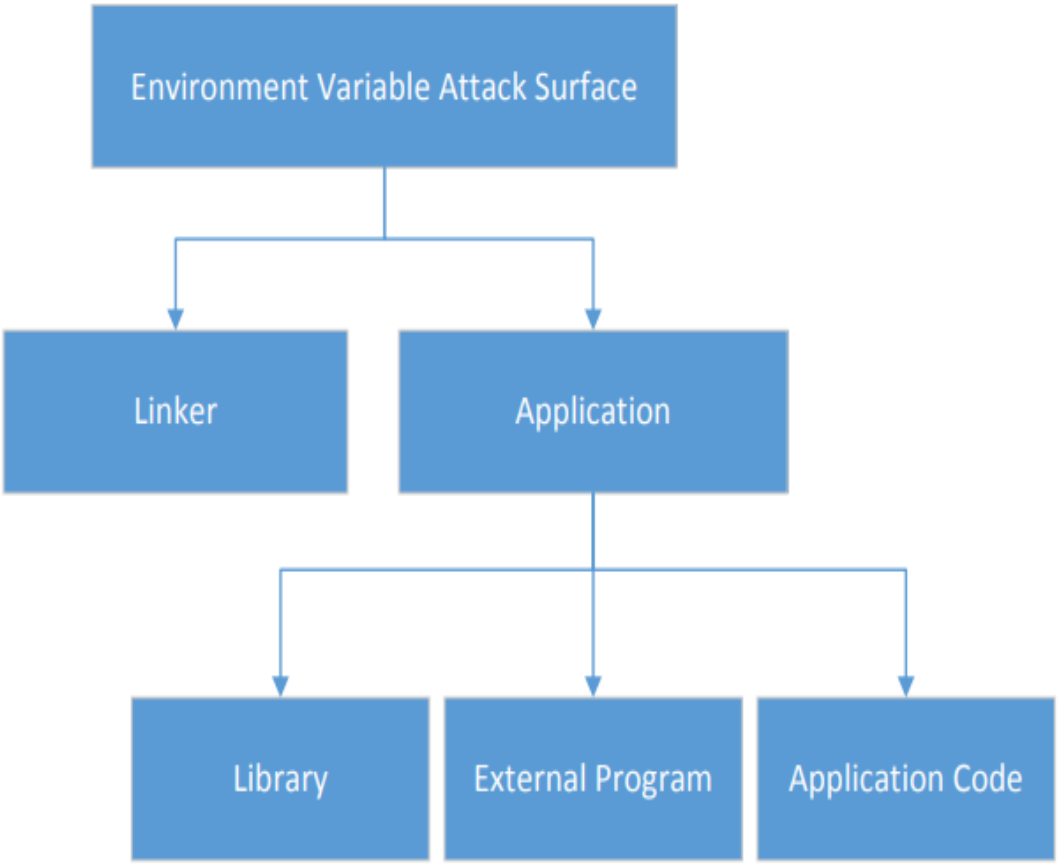
Print out environment variable

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
```

# 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**

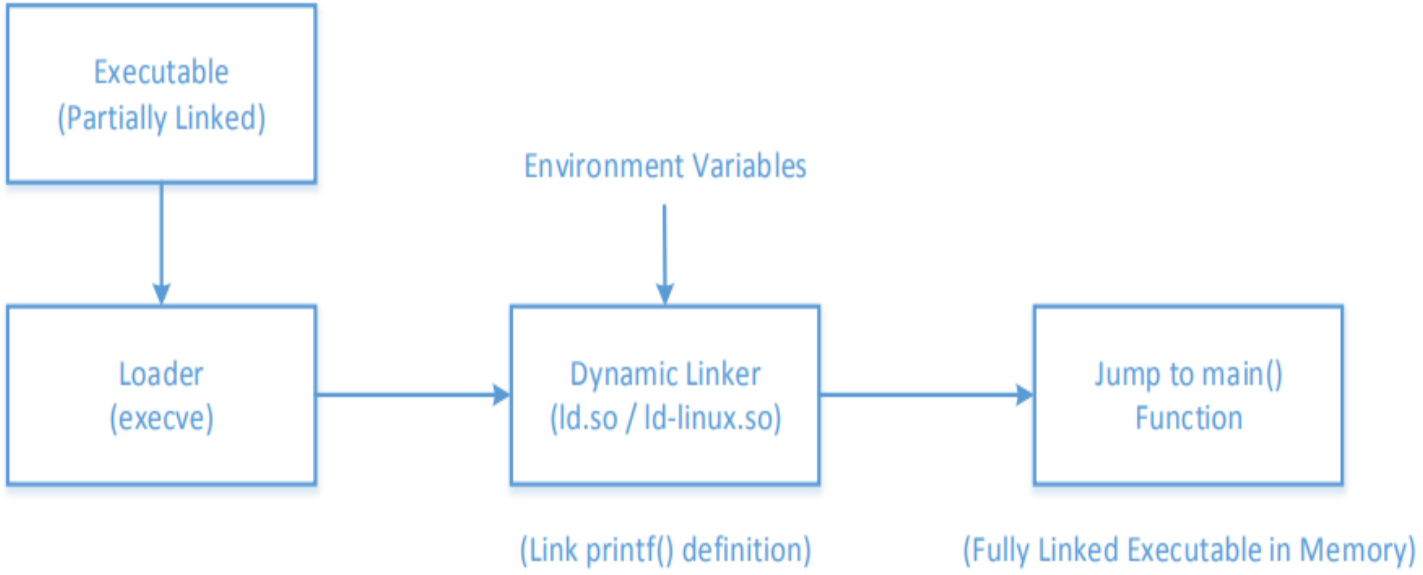
- The linker combines the program's code and the library code containing the printf() function
- We can notice that the size of a static compiled program is 100 times larger than a dynamic program

```
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

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
```