# Environment Variable and `Set-UID` Program Lab


The development of this document is/was funded by the following grants from the US National Science Foundation: No. 1303306 and 1318814. This lab was imported into the Labtainer framework by the Naval Postgraduate School, Center for Cybersecurity and Cyber Operations under National Science Foundation Award No. 1438893. 

## 1 Overview

The learning objective of this lab is for students to understand how environment variables affect program and system behavior. Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are used by most operating systems, since they were introduced to Unix in 1979. Although environment variables affect program behavior, how they achieve that is not well understood by many programmers. As results, if a program uses environment variables, but the programmer do not know that they are used, the program may have vulnerabilities. In this lab, students will understand how environment variables work, how they are propogated from parent process to child, and how they affect system/program bahivors. We are particularly interested in how environment variables affect the behavior of `Set-UID` programs, which are usually privileged programs.

## 2 Lab Environment

This lab runs in the Labtainer framework, available at http://my.nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

From your labtainer-student directory start the lab using:

```
labtainer setuid-env
```

Links to this lab manual and to an empty lab report will be displayed. If you create your lab report on a separate system, be sure to copy it back to the specified location on your Linux system.

## 3 Lab Tasks

### 3.1 Task 1: Manipulating environment variables

In this task, we study the commands that can be used to set and unset environment variables. We are using the Bash in this lab. The default shell that a user uses is set in the `/etc/passwd` file (the last field of each entry). You can change this to another shell program using the command `chsh` (please do not do it for this lab). Please do the following tasks:

- Use `printenv` or `env` command to print out the environment variables. If you are interested in some particular environment variables, such as `PWD`, you can use "`printenv PWD`" or "`env | grep PWD`".

- Use `export` and `unset` to set or unset environment variables. It should be noted that these two commands are not seperate programs; they are two of the Bash's internal commands (you will not be able to find them outside of Bash).

### 3.2 Task 2: Inheriting environment variables from parents

In this task, we study how environment variables are inherited by child processes from their parents. In Unix, `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent; however, several things are not inherited by the child (please see the manual of `fork()` by typing the following command: `man fork`). In this task, we would like to know whether the parent's environment variables are inherited by the child process or not.

**Step 1.** Please compile and run the `printenv.c` program (the listing is below), and describe your observation. Because the output contains many strings, you should save the output into a file, such as using `a.out > child` (assuming that `a.out` is your executable file name).

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

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);
  }
}
```

**Step 2.** Now comment out the `printenv()` statement in the child process case, and uncomment the `printenv()` statement in the parent process case. Compile and run the code, and describe your observation. Save the output in another file.

**Step 3.** Compare the difference of these two files using the `diff` command. Please draw your conclusion.

### 3.3 Task 3: Environment variables and `execve()`

In this task, we study how environment variables are affected when a new program is executed via `execve()`. The function `execve()` calls a system call to load a new command and execute it; this function never returns. No new process is created; instead, the calling process's text, data, bss, and stack are overwritten by that of the program loaded. Essentially, `execve()` runs the new program inside the calling process. We are interested in what happens to the environment variables; are they automatically inherited by the new program?

**Step 1.** Please compile and run the `execve.c` program, (the listing is below), and describe your observation. This program simply executes a program called `/usr/bin/env`, which prints out the environment variables of the current process.

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

extern char **environ;

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

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

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

  return 0 ;
}
```

**Step 2.** Now, change the invocation of `execve()` to the following, and describe your observation.

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

**Step 3.** Please draw your conclusion regarding how the new program gets its environment variables.

### 3.4 Task 4: Environment variables and `system()`

In this task, we study how environment variables are affected when a new program is executed via the `system()` function. This function is used to execute a command, but unlike `execve()`, which directly executes a command, `system()` actually executes `"/bin/sh -c command"`, i.e., it executes `/bin/sh`, and asks the shell to execute the command.

If you look at the implementation of the `system()` function, you will see that it uses `execl()` to execute `/bin/sh`; `excel()` calls `execve()`, passing to it the environment variables array. Therefore, using `system()`, the environment variables of the calling process are passed to the new program `/bin/sh`. Please compile and run the following `system.c` program, (listing is below) to verify this.

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

int main()
{
  system("/usr/bin/env");

  return 0 ;
}
```

### 3.5 Task 5: Environment variable and `Set-UID` Programs

`Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program runs, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky. Although the behavior of a `Set-UID` programs is decided by its program logic, not by users, users can indeed affect the behavior via environment variables. To understand how `Set-UID` programs are affected, let us first figure out whether environment variables are inherited by the `Set-UID` program's process from the user's process.

**Step 1.** We will use the `printall.c` program, (listing below) that prints out all the environment variables in the current process.

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

extern char **environ;

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

**Step 2.** Compile the `printall.c` program, change its ownership to `root`, and make it a `Set-UID` program.

```
sudo chown root:root a.out
sudo chmod a+s a.out
```

**Step 3.** In your Bash shell use the `export` command to set the following environment variables (they may have already exist):

- `PATH`
- `LD_LIBRARY_PATH`
- `ANY_NAME` (this is an environment variable defined by you, so pick whatever name you want).

These environment variables are set in the user's shell process. Now, run the `Set-UID` program from Step 2 in your shell. After you type the name of the program in your shell, the shell forks a child process, and uses the child process to run the program. Please check whether all the environment variables you set in the shell process (parent) get into the `Set-UID` child process. Describe your observation. If there are surprises to you, describe them.

### 3.6 Task 6: The system() function and `Set-UID` Programs

One of many changes made to the setuid feature over the years relates to its interaction with the system() function. Since the results of system() are dependent on environment variables, it was a source of increased risk. For example, use of the `PATH` environment variable will determine which of potentially several instances of a given program will be invoked. The following command:

```
export PATH=~/:$PATH
```

would cause the shell to first look in your home directory for a program of a given name.

The setuid `path-suid.c` program list below is supposed to execute the `/bin/ls` command; however, the programmer only uses the relative path for the `ls` command, rather than the absolute path:

```
# path-suid.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
   uid_t euid = geteuid();
   printf("euid %d\n", euid);
   system("ls");
   return 0;
}
```

Please compile the `path-suid.c` program, and change its owner to `root`, and make it a `Set-UID` program.

Can you cause the path-suid.c `Set-UID` program run the ls.c program instead of `/bin/ls`? If you can, is your code running with the root privilege? Describe and explain your observations.

```
# ls.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int main(){
    uid_t euid = geteuid();
    printf("my ls prog, euid is %d\n", euid);
    return 0;
}
```

### 3.7 Task 7: The LD_PRELOAD environment variable and Set-UID Programs

In this task, we study how Set-UID programs deal with some of the environment variables. Several environment variables, including LD_PRELOAD, LD_LIBRARY_PATH, and other LD_* influence the behavior of dynamic loader/linker. A dynamic loader/linker is the part of an operating system (OS) that loads (from persistent storage to RAM) and links the shared libraries needed by an executable at run time.

In Linux, ld.so or ld-linux.so, are the dynamic loader/linker (each for different types of binary). Among the environment variables that affect their behavior, LD_LIBRARY_PATH and LD_PRELOAD are the two that we are concerned in this lab. In Linux, LD_LIBRARY_PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories. LD_PRELOAD specifies a list of additional, user-specified, shared libraries to be loaded before all others. In this task, we will only study LD_PRELOAD.

**Step 1.** First, we will see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Please follow these steps:

1. Let us build a dynamic link library. The following program listing, is in mylib.c. It basically overrides the sleep() function in libc:

   ```
   #include <stdio.h>
   void sleep (int s)
   {
     /* If this is invoked by a privileged program,
        you can do damages here!  */
     printf("I am not sleeping!\n");
   }
   ```

2. We can compile the mylib.c program using the following commands (in the -lc argment, the second character is $\ell$):

   ```
   % gcc -fPIC -g -c mylib.c
   % gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
   ```

3. Now, set the LD_PRELOAD environment variable:

   ```
   % export LD_PRELOAD=./libmylib.so.1.0.1
   ```

4. Finally, compile the myprog.c program in the same directory as the above dynamic link library libmylib.so.1.0.1:

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

**Step 2.** After you have done the above, please run myprog under the following conditions, and observe what happens.

- Make myprog a regular program, and run it as a normal user.
- Make myprog a Set-UID root program, and run it as a normal user.
- Become root with sudo su, export the LD_PRELOAD environment variable again and run the myprog program again.
- Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD_PRELOAD environment variable again as the user1 user and run it.

**Step 3.** You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Please design an experiment to figure out the main causes, and explain why the behavior in Step 2 is different. (Hint: the child process may not inherit the LD_* environment variables).

### 3.8 Task 8: Capability Leaking

To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to hand over its control to the user; in this case, root privileges must be revoked. The setuid() system call can be used to revoke the privileges. According to the manual, "setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set". Therefore, if a Set-UID program with effective UID 0 calls setuid(n), the process will become a normal process, with all its UIDs being set to n.

When revoking the privilege, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privilege is dropped, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities, e.g., access to a protected file.

Compile the leak.c program, change its owner to root, and make it a Set-UID program. Run the program as a normal user, and describe what you have observed. Will the file /etc/zzz be modified? Please explain your observation.

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

```
void main()
{ int fd;

  /* Assume that /etc/zzz is an important system file,
   * and it is owned by root with permission 0644.
  fd = open("/etc/zzz", O_RDWR | O_APPEND);
  if (fd == -1) {
     printf("Cannot open /etc/zzz\n");
     exit(0);
  }

  /* Simulate the tasks conducted by the program */
  sleep(1);

  /* After the task, the root privileges are no longer needed,
     it's time to relinquish the root privileges permanently. */
  setuid(getuid());  /* getuid() returns the real uid */

  if (fork()) { /* In the parent process */
    close (fd);
    exit(0);
  } else { /* in the child process */
    /* Now, assume that the child process is compromised, malicious
       attackers have injected the following statements
       into this process */

    write (fd, "Malicious Data\n", 15);
    close (fd);
  }
}
```

# 4 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation, beyond what is required by the lab description. Your can earn bonus points for extra efforts (at the discretion of your instructor).

If you edited your lab report on a separate system, copy it back to the Linux system at the location identified when you started the lab, and do this before running the stoplab command. After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab setuid-env
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.