Lab 01: Environment Variables & Set-UID Programs Lab

Environment Variables & Set-UID Programs Lab

Due Sunday February 12th @ 11:59 PM

Adapted from SEED Labs: A Hands-on Lab for Security Education.

The learning objective of this lab is for students to understand how environment variables affect program and system behaviors. 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 into Unix in 1979. Although environment variables affect program behaviors, how they achieve that is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities.

In this lab, students will understand how environment variables work, how they are propagated from parent process to child, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs. This lab covers the following topics:

- Environment variables
- Set-UID programs
- How to securely invoke external programs
- The dynamic linker/loader

Resources

- Code related to this lab can be found in our class's GitHub repository. Specifically, see `01_envvars_setuid/`.
- Chapters 1 & 2 in the SEED Textbook.
- A related video lecture (Udemy course) recorded by Kevin Du.
- Checklist for Security of Setuid Programs
- Setuid Demystified, Chen et al.
- How to write a Setuid program, Matt Bishop

Lab Tasks

This lab has been tested on the pre-built SEED VM (Ubuntu 20.04 VM).

Task 1: Manipulating Environment Variables

Task 1.1

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

Task 1.2

Use `export` and `unset` to set or unset environment variables. Please show that you can set your own environment variables using `export`

Task 2: Passing Environment Variables (Parent -> Child)

In this task, we study how a child process gets its environment variables from its parent. 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.

Task 2.1

Please compile and run the following program, and describe your observations. You should save the output of this program to a new file (`./myprintenv > out1`)(NOTE: This file is also in out GitHub repo that you cloned, `myprintenv.c`

```c
// Compile:
//  $ gcc myprintenv.c -o myprintenv
//
// Run the program and redirect output to a text file:
//  $ ./myprintenv > myenv1

#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++;
    }
}

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

myprintenv.c delivered with ❤ by emgithub view raw

Task 2.2

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 again, and describe your observation. Save the output in another file (e.g., `./myprintenv > out2`.

Task 2.3

Compare the difference of these two files using the `diff` command. To compare the two outputs with `diff`, run the command `diff out1 out2` Please draw your conclusions. HINT: If you see no ouput running the diff command , that is a valid answer (what does that mean?)

Task 3: Environment Variables 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, when anyone runs this program, the program gains root's privileges during its execution. Set-UID allows us to do many interesting things, but since it escalates the user's privilege, it is quite risky. Although the behavior of Set-UID programs is decided by their program logic, not by users, **users can indeed affect the behavior via environment variables**. To understand how Set-UID programs can be affected, let us first figure out whether environment variables are inherited by a Set-UID program's process from the user's process.

Task 3.1

Use the following program that can print out all the environment variables in the current process. Verify that your implementation correctly prints the environment variables.

```c
// Print environment variables using environ.
//
// Compile:
//  $ gcc myenv_environ.c -o myenv_environ

#include <stdio.h>

extern char **environ;

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

myenv_environ.c delivered with ❤ by emgithub view raw

Task 3.2

Compile the above program, change its ownership to `root`, and make it a Set-UID program.

```
$ sudo chown root myenv_environ   # chown = (ch)ange (own)er
$ sudo chmod 4755 myenv_environ   # chmod = (ch)ange file (mod)e bits
```

Task 3.3

In your shell (you need to be in a normal user account, not the `root` account), use the `export` command to set the following environment variables *(**NOTE:** they may exist already!)*:

- `PATH` — *prepend the current directory symbol to `PATH`*
- `LD_LIBRARY_PATH` — *prepend the current directory symbol to `LD_LIBRARY_PATH`*
- `TASK5` — *this is a non-standard variable; define this however you want*

To be clear, these environment variables should be set in the user's shell process.

After you have exported the above environment variables into the user's shell environment, run the Set-UID program from Task 3.2 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) are inherited in the Set-UID child process. Describe your observations. If there are any surprises to you, describe them.

> **Before you proceed... Shell Countermeasures**
>
> An important change is needed to circumvent shell countermeasures
>
> The `system(cmd)` function executes the `/bin/sh` program first, and then asks this shell program to run the `cmd` command. In Ubuntu 20.04 (and several versions before), `/bin/sh` is actually a symbolic link pointing to `/bin/dash`. This shell program has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if `dash` detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping its privileges.
>
> Since our victim program is a Set-UID program, the countermeasure in `/bin/dash` can prevent our attack. To see how our attack works without such a countermeasure, we will link `/bin/sh` to another shell that does not have such a countermeasure. There is another shell program called `zsh` in our Ubuntu 20.04 VM. Use the following commands to link `/bin/sh` to `/bin/zsh`, and to reset `/bin/sh` to `/bin/dash`, respectively:
>
> ```
> $ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh - zsh has no set-uid countermeasure
> $ sudo ln -sf /bin/dash /bin/sh  # set shell to dash - dash has a set-uid countermeasure
> ```

Task 4: Exploiting a SET-UID Program with the `system()`function

Although `system()` and `execve()` can both be used to run new programs, `system()` is quite dangerous if used in a privileged program, such as Set-UID programs. We have seen how the PATH environment variable affect the behavior of system(), because the variable affects how the shell works. `execve()` does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario. Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run `/bin/cat` to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file

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

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

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

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);

    /*
     * Use only one of the following (comment out the other):
     */

    system(command);
    //execve(v[0], v, 0);

    return 0;
}
```

catall.c hosted with ❤ by GitHub view raw

Task 4.1

Compile the above program, make it a root-owned Set-UID program. The program will use `system()` to invoke the command. If you were Bob, can you compromise the integrity of the system? For example, can you remove a file that is not writable to you? (Hint: Look at lecture from February 1st)

Task 4.2

Comment out the `system(command)` statement, and uncomment the `execve()` statement; the program will use `execve()` to invoke the command. Compile the program, and make it a root-owned Set-UID. Do your attacks in Step 1 still work? Please describe and explain your observations.

Task 5: `PATH` and Set-UID Programs

In this task, we study how Set-UID programs deal with environment variables. Specifically, we examine the `PATH` environment variable and its potential impact on Set-UID programs. Because `system()` runs commands by invoking a shell, calling `system()` within a Set-UID program is quite dangerous. One concern is that the actual behavior of the shell program can be affected by environment variables, such as `PATH`; these environment variables are provided by the user, who may be malicious. By changing these variables, malicious users can potentially control the behavior of the Set-UID program.

In `Bash`, you can change the `PATH` environment variable in the following way (this example adds the directory `/home/seed` to the beginning of the `PATH` environment variable):

```
$ export PATH=/home/seed:$PATH
```

The Set-UID program 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:

```c
#include <stdlib.h>

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

ls_vuln.c delivered with ❤ by emgithub view raw

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

Can you make this Set-UID program run your code (e.g., code that launches a new shell) instead of `/bin/ls`?

If you can, is your code running with root privileges? Describe and explain your observations.

Task 6: `LD_PRELOAD` and Set-UID Programs

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

In Linux, `ld.so` or `ld-linux.so`, are the dynamic linker/loader (`ld-linux.so` supports ELF, which is a common file format for executables today). Among the environment variables that affect the behavior of the dynamic linker/loader, `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 focus on `LD_PRELOAD`.

Task 6.1

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

1. Let us build a dynamically linked library. Create the following program, and name it `mylib.c`. This program basically overrides the `sleep()` function in `libc`:

   ```c
   #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 above program using the following commands (in the `-lc` argument, the second character is the lowercase letter "L"):

   ```
   $ 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 following program `myprog` within the same directory as the dynamically linked library you created above (`libmylib.so.1.0.1`):

   ```c
   /* myprog.c */
   #include <unistd.h>
   int main()
   {
     sleep(1);
     return 0;
   }
   ```

Task 6.2

After you have done Task 6.1, 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.
- Make `myprog` a Set-UID root program, export the `LD_PRELOAD` environment variable again in the root account and run it.

  > *You can run `sudo su` to login as root. Make sure to `exit` when you are done.*

Task 6.3

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 behaviors you observed in the previous part are different. *(**HINT:** the child process may not inherit the `LD_` environment variables!).*

Submission Instructions

The lab report is to help me see that you did the lab and followed the instructions. For each task, you should include a screenshot to show you completed the task. If the task asks you to write down observations, you should also include those in your lab report. For the tasks that requires you to do some thinking and find ways to exploit a program, you should write a brief description about your approach and the steps you took to get your output. This is a lab report taken from a previous offering of this course. This is a good example of how you should format your lab report:
https://www.cs.montana.edu/pearsall/classes/spring2023/476/labs/SampleLabReport.pdf
Once you are ready, submit your lab report **AS A PDF** to the appropriate D2L submission box.