Environment Variable and SET-UID Lab

Contents

1 Aims and Objectives

The learning objective of this lab is for students to understand how environment variables affect program and system behaviours. 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 behaviours, 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 behaviours. We are particularly interested in how environment variables affect the behaviour of Set-UID programs, which are usually privileged programs.

This lab covers the following topics:

- Environment variables
- Set-UID programs
- Securely invoke external programs
- Capability leaking
- Dynamic loader/linker

1.1 Related Text

Detailed coverage of the Set-UID mechanism, environment variables, and their related security problems can be found in the following:

- **Required Reading:** SetUID Privileged Programs Powerpoint Presentation on Blackboard.

- Chapters 1 and 2 of the SEED Book, Computer & Internet Security: A Hands-on Approach, 2nd Edition, by Wenliang Du. See details at https://www.handsonsecurity.net.

2 Lab Tasks

2.1 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 shell in the 'uwe' user account. 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).

Here is the output from the tail of the default VM /etc/passwd file:

```
$ tail /etc/passwd
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
uwe:x:1000:1000:uwe,,,:/home/uwe:/bin/bash
```

Question: What is the difference between a user shell of `/usr/sbin/nologin` and `/bin/false`? Explain the difference.

Please do the following tasks:

- Use the `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 separate programs; they are two of the Bash's internal commands (you will not be able to find them outside of Bash).

Proof: Demonstrate proof of the above commands.

2.2 Task 2: Passing Environment Variables from Parent Process to Child Process

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.

Step 1. Please compile and run the following program, 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();                     ①
            exit(0);
      default: /* parent process */
            //printenv();                   ②
            exit(0);
      }
}
```

Step 2. Now comment out the `printenv()` statement in the child process case (Line ①), and uncomment the `printenv()` statement in the parent process case (Line ②). Compile and run the code again, saving the output in another file.

Question: Describe your observations between the two programs and suggest why this may be.

2.3 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 following program, 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);          ①

      return 0;
}
```

Question: Describe your observations of the program and explain what you think is happening.

Step 2. Change the invocation of `execve()` in Line ① to the following;

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

Question: Describe your observations of the changed program.

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

2.4 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; execl()` calls `execve()`, passing to it the environment variables array. Therefore, using `system()`, the environment variables of the calling process is passed to the new program `/bin/sh`.

Please compile and run the following program to verify this.

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

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

2.5 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 behaviours of `Set-UID` programs are decided by their program logic, not by users, users can indeed affect the behaviours 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. Write the following program that can print 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 above program, change its ownership to `root`, and make it a `Set-UID` program.

```
// Assuming the program's name is foo
$ sudo chown root foo
$ sudo chmod 4755 foo
```

Step 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 (they may already exist):

- PATH
- LD LIBRARY PATH
- <YOUR_SURNAME> (this is an environment variable defined by you, so pick your Surname).

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.

Question: Describe your observation. If there are surprises to you, describe them.

2.6 Task 6: The PATH Environment Variable and Set-UID Programs

Because the shell program is invoked, calling `system()` within a `Set-UID` program is quite dangerous. This is because the actual behaviour 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 control the behaviour of the `Set-UID` program. In Bash, you can change the `PATH` environment variable in the following way (this example adds the directory `/home/uwe` to the beginning of the PATH environment variable):

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

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

Please compile the above program, and change its owner to `root`, and make it a `Set-UID` program. Can you let this `Set-UID` program run your code instead of `/bin/ls`? If you can, is your code running with the root privilege?

Note: The `system(cmd)` command executes the `/bin/sh` program first, and then asks the new shell program to run the `(cmd)` command. In Ubuntu 16.04, the /bin/sh program is actually a symbolic link that points to the /bin/dash shell.

```
$ ls -ltr   /bin/sh
lrwxrwxrwx 1 root root 9 Sep  9 13:00 /bin/sh -> /bin/dash
```

The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. 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 the privilege. The `dash` program in Ubuntu 12.04 does not have this behaviour. 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. We have installed a shell program called `zsh` in our Ubuntu 16.04 VM. Use the following commands to link `/bin/sh` to `zsh`.

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Question: Describe and explain your observations.

2.7 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 behaviour 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 behaviours, 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 behaviour of dynamic loader/linker when running a normal program. Please follow these steps:

1. Let us build a dynamic link library. Create the following program, and name it 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 damage 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 a lower case ‘L’ not a ‘1’/’one’):

```
      $ 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, and 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.
- Make `myprog` a `Set-UID` root program, export the `LD PRELOAD` environment variable again in the root account and run it.
- 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 in a different user's account (not-root user) and run it.

Question: You should be able to observe different behaviours in the scenarios described above, even though you are running the same program. Note the different behaviours of the 4 programs here.

Step 3. Explain why the behaviours in the four different programs in Step 2 are different.

2.8 Task 8: Invoking External Programs Using `system()` versus `execve()`

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 behaviour 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.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.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 following
       * commands in each test.
       */
      system(command);
      // execve(v[0], v, NULL);

      return 0;
}
```

Step 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?

Step 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?

Question: Please describe and explain your observations for Step 1 and Step 2.

2.9 Task 9: 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 downgraded, 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.

Compile the following 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. Before running this program, you should create the file `/etc/zzz` first.

Question: 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.
       * 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);
      }

      /* 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);
      }
}
```

2.10 Lab Clean-up

Restore the symlink for /bin/sh to point to /bin/dash which we modified in section “0

Task 6: The PATH Environment Variable and Set-UID Programs”.

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

3 Further research and a real-world case study

Produce an 800–1000-word report detailing the following:

1. Investigate and explain how the `dash` shell countermeasures work with regard to dash being executed from within a Set-UID process. (**approximately 400 words**)

2. A real-world case study involving security issues with privileged SetUID binaries. For example, CVE-2021-26936 was published on 10th February 2021, which demonstrates that these basic issues security principles can still be lacking today.

   Find and research a real-world case study involving SET-UID programs. Explain what the security incident was, how the incident arose, and any potential mitigations that could have been taken to avoid the issue. (**approximately 600 words**)

4 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. The format of the lab report is up to you. You can copy the questions from this worksheet into a new document and answer them in the separate report. The report should be of a professional standard.

You need to provide explanation to the observations that are interesting or surprising. Please also list any important code snippets you have written followed by explanation. Simply attaching code or screenshots without any explanation will not receive credits. The report must demonstrate your understanding of the subject and material and not just be a log of your actions.

All screenshots in the report must have your student number and date stamp in the user prompt. Failure to include these details in the screenshots will invalidate the report and receive a mark of zero.

5 Marking Criteria

	0-29%	30-39%	40-49%	50-59%	60-69%	70-84%	85-100%
Completion and evidence of all specified tasks (30%)	Little or no effort made to complete the tasks detailed	Some tasks complete with major omission	Most tasks complete but with minor omissions	All tasks complete in full. Evidence incomplete or unclear in places	All tasks complete in full. Evidence of a good standard to detail tasks.	All tasks complete in full. Excellent use of evidence to detail tasks.	All tasks complete in full. Highly reflective use of evidence to develop argument
Depth of understanding (30%)	Serious gaps or errors in understanding the topic	Some evidence of understanding the topic with major errors or gaps	Evidence of understanding the topic but with minor errors or gaps	Adequate understanding of topic	Clear understanding of topic	Thorough and comprehensive understanding of topic	Impressive and original depth of understanding of topic
Analysis & explanation of `dash` countermeasures (10%)	Little or no understanding of dash countermeasures provided	Some evidence of dash countermeasures provided	Key details of dash countermeasures articulated	Adequate explanation of dash countermeasures articulated	Clear understanding of dash countermeasures articulated	Thorough and comprehensive understanding of dash countermeasures articulated	Impressive and original depth of understanding of dash countermeasures
Description and analysis of real-world security incident (20%)	Little or no evidence of research related to a real-world security incident	Some evidence of research related to a real- world security incident	A real-world security incident has been identified with key details being discussed.	A real-world security incident has been identified, with some discussion on why the incident occurred.	A well-detailed real-world security incident has been identified, with some discussion on why the incident occurred and how this could have been mitigated.	A well-detailed real-world security incident has been identified, with good discussion on why the incident occurred and how this could have been mitigated	A well-detailed real-world security incident has been identified, with excellent discussion on why the incident occurred and justification of how this could have been mitigated.
Report Presentation (10%)	Very poor presentation	Weak presentation	Has not followed required conventions; poor proof-reading	Usually follows required practices; some issues to be addressed e.g., typos, punctuation	Follows required presentational practices; a few typos/errors in punctuation or grammar	Excellent presentation: typos/errors in punctuation etc. are rare	Excellent presentation

6 Document Revision History

Version	Date	Changes
1.0	12th February 2021	Initial Release