MH4920
Supervised Independent Study I

Environment Variable & Set-UID

Brandon Goh Wen Heng

Academic Year 2017/18

Contents

1 Introduction

Environment variables are dynamic-named variables that affects running programs on a particular system. Common environment variables include `PATH`, where it is used to locate files for execution and `TMP`, used to describe the location or folder to store temporary files.

2 Overview

This lab will explore the use of environment variables, the process of propagation from parent to child processes and how environment variables affect running processes in the system. In particular, we pay special attention to the use of environment variables with respect to Set-UID programs.

3 Lab

3.1 Manipulating Environment Variables

We look at the basics of environment variables, which is to firstly set, list and remove the variables.

1. To set environment variables, we use the `export` command.
2. To list the environment variables, the commands `env | grep <env name>` or `printenv`.
3. To remove environment variables, `unset` command is nice.

Looking in detail at the execution of this commands, the figure below lists environment variables that exists in the system by default.

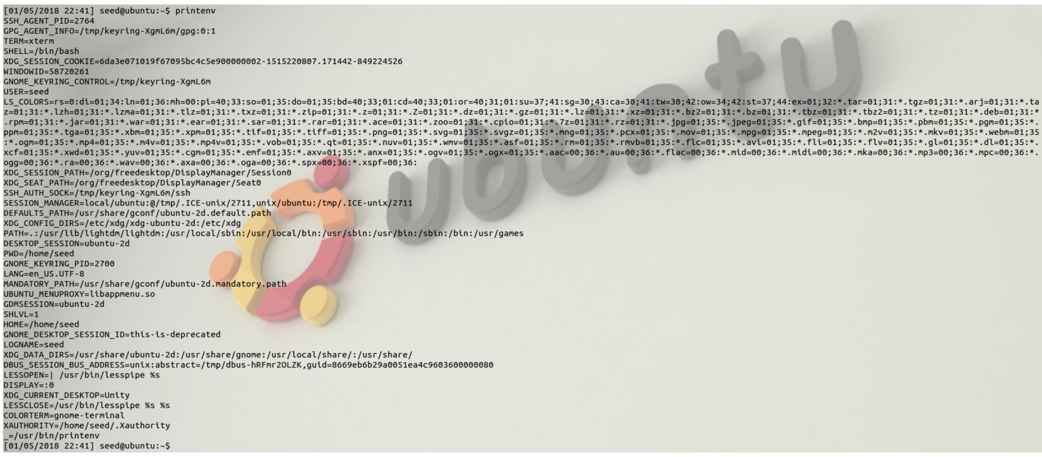

Figure 1: Environment Variable List

Environment variables can be inserted into the system using `export` where needed. In this instance, the environment variable `SOMEVAR` is set with the value `defined`. Figure 3 shows the existence of the newly defined environment variable and can now be located when we query the list of environment variables. The user-defined variable is highlighted in red.

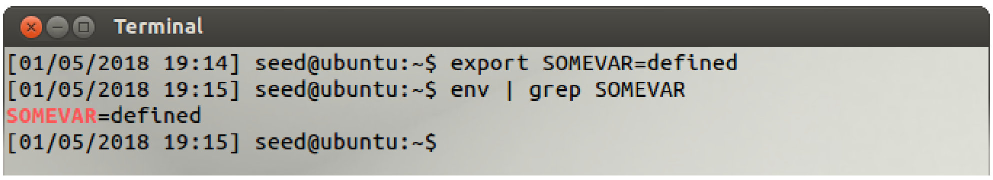

Figure 2: Defined Environment Variable

Finally, removing the user-defined environment variable requires the use of the `unset` command and it will not be displayed when the `env | grep SOMEVAR` command is executed.

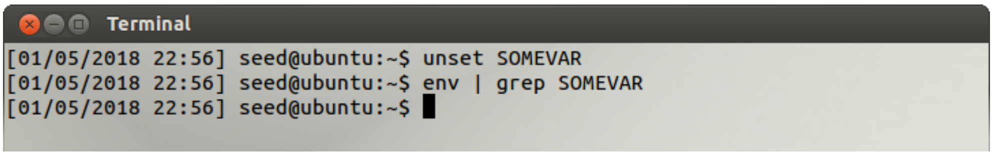

Figure 3: Removal of Environment Variable

3.2 Process Inheritance

In the current subsection, we fork the parent process to study how the environment variables affect both the parent and the child process. We also look at the inheritance properties of these processes. The `C` code that helps us to print the environment variables for the child and the parent process has been attached to the Appendix.

The code below compiles the `C` code for the parent and child process separately (after toggling the respective `printenv()` lines), executes the two programs and makes use of the `diff` command to show the difference in the outputs of the environment variables of the parent and the child process.

```
$ gcc -o childproc childprocess.c
$ gcc -o parproc parprocess.c
$ parproc > parproc.txt && childproc > childproc.txt
$ diff parproc.txt childproc.txt > diff.txt
```

The output from the `diff` command only lies in the last line where it denotes the name of the file being executed. Figure 4 shows the graphical output from the terminal. This indicates that the same set of environment variables residing on the system affects both the parent and the child processes.

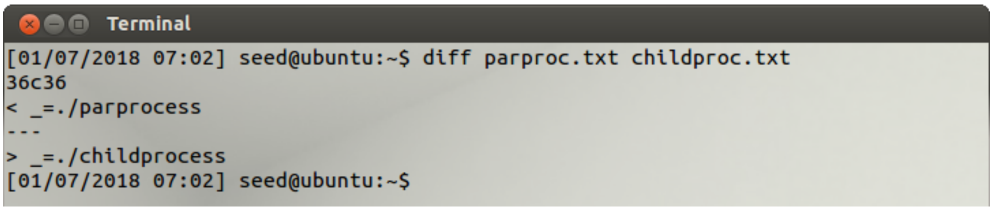

Figure 4: No Difference in Environment Variables

3.3 Execution of `execve()`

The `execve()` command is analysed in this subsection, to determine whether the execution of the program is affected by the environment variables. The `C` code that will be used has been attached to the Appendix.

We need to first look at the function header of `execve()` and understand its function before continuing.

```
execve(const char *filename, char *const argv[],
char *const envp[]);
```

The first parameter is the file to be executed or the command name, while the second parameter will include the parameters for the executed file. The last parameter will include the environment variables that may be used by the program during execution with the form *Name=Value*. If there are no environment variables to be included in the execution of the program, "`NULL`" is used.

When executing the `C` program with the following line,

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

there is no output from the program. This is due to the existence of `NULL` in the third parameter of the function. When `NULL` is used, no environment variables are passed when calling the `env` function and therefore there are no environment variables for output.

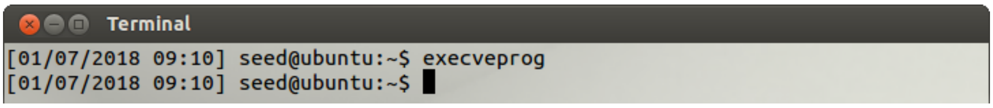

Figure 5: `NULL` in `execve`

The third parameter is now changed from `NULL` to `environ`, i.e.

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

Execution of this compiled program now shows all the environment variables. `environ` is used to list all the environment variables in the user environment. As this is passed to the `env` function, execution will now output all the environment variables in the user environment.

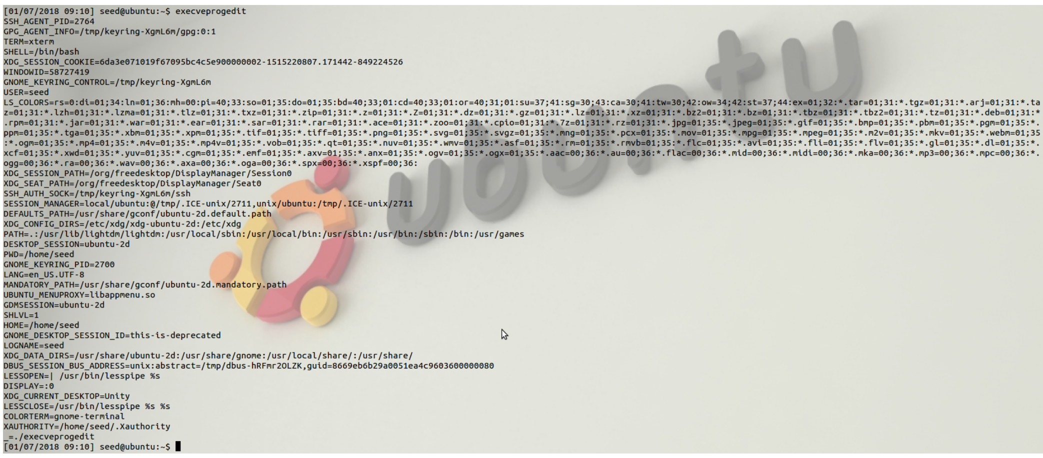

Figure 6: `environ` in `execve`

3.4 Execution of `system()`

In this subsection, we focus on calling the `system` function and observe how environment variables are passed. When `system` is called, `execl` is executed and the command is passed as one of the parameters. Since `execl` does not have the parameter where `const char* envp[]` can be explicitly defined, the variable `environ` will be used instead. These will be used as the parameters when executing `execve` and therefore the execution of the code will show all environment variables. The `C` code used to output the environment variables using `system` is attached in the Appendix.

Figure 7: `system` execution

3.5 Environment Variables & `Set-UID` Programs

`Set-UID` is an important security feature in Unix systems. In this subsection, it is important to understand how `Set-UID` programs are affected by environment variables and the user's process.

To begin, a `C` program is created to print out all the environment variables in the current process. The `C` code has been attached in the Appendix for reference. Assuming that the program name is *setuid*, we use the following commands to make root the owner of the `Set-UID` program.

```
$ su
# gcc -o setuid setuid.c
# chmod 4755 setuid
$ exit
```

To examine if the `Set-UID` program is affected by the user's process, we add three environment variables using the user account (not root).

```
$ export PATH=/home/seed/lab:$PATH
$ export LD_LIBRARY_PATH=/home/seed/lib
$ export NEWENV=var
```

As PATH already exists in the system, :$PATH is added to the back to ensure that the existing PATH value is concatenated to the back of the newly added value.

Execution of the Set-UID programs shows that the edited PATH and NEWENV are displayed in the output. The LD_LIBRARY_PATH is however not in the list of environment variables when the Set-UID program is being run. As LD_LIBRARY_PATH can be used to run malicious libraries, it is ignored by default if it is a Set-UID program. Figure 8 and 9 shows the difference in the user environments between a Set-UID and a non Set-UID program.

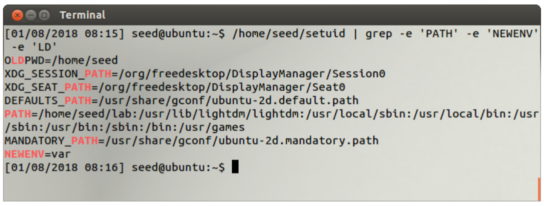

Figure 8: Set-UID program

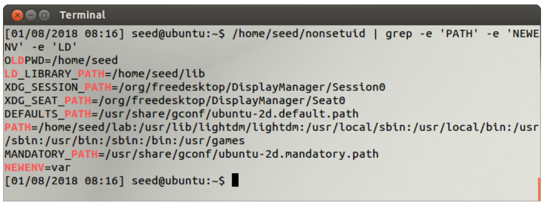

Figure 9: Non Set-UID program

3.6 PATH Environment Variable & Set-UID

Using system as a Set-UID program is dangerous as the PATH environment variable can be exploited to run malicious code. In this subsection, we will explore the use of the PATH environment variable and the interaction with the Set-UID program.

A Set-UID program written in C is defined such that it uses the system command to execute ls. The program is compiled with the name *setuidpath* for readability purposes and the code can be referenced in the Appendix. The PATH environment variable is now edited to point to another directory and placed at the front. Placing the directory at the front ensures that the program will always look into our added directory first before moving on to the following directories in the list to find the respective program to be executed. In this instance, we run the following code to update PATH,

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

We create a file that calls sh using system. The code has also been attached in the Appendix. The following line of code is run to ensure that the code is being compiled into a program with the name ls in the directory /home/seed that was just added.

```
$ gcc -o ls callsh.c
```

When *setuidpath* is run, a shell is obtained as the process that calls the shell is privileged. Further scripting reveals that we have obtained root access to the system using a Set-UID program.
The results of this subsection proves that it is extremely dangerous when a

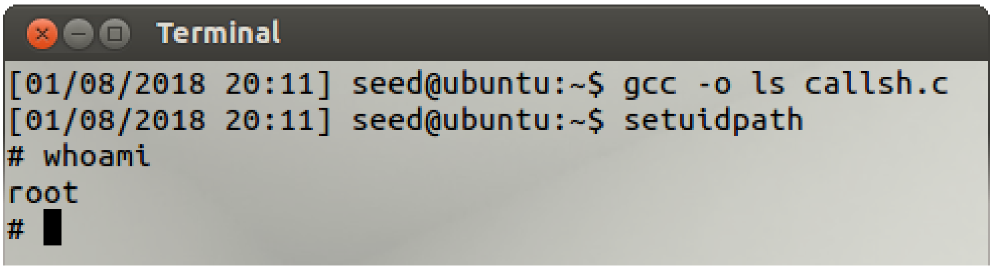

Figure 10: Exploit using `PATH`

`Set-UID` program uses the `system` command. It has also been shown that the `PATH` environment variable can be easily edited by a malicious user even without root privileges. Furthermore, using a relative path further increases the risk of the system being compromised.

3.7 `LD_PRELOAD` Environment Variable & `Set-UID`

The current subsection will focus on how `Set-UID` programs interact with `LD_*` environment variables, in particular `LD_PRELOAD`. The `LD_*` environment variables affect the behaviour of the dynamic loaders in Linux and `LD_PRELOAD` specifies additional user-specified shared library directories to be loaded before using the default set of directories.

A dynamic link library is created to replace the `sleep()` function in `libc`. The code for this library is attached in the Appendix. This is compiled and `LD_PRELOAD` is now edited to include the newly compiled library.

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

The `-fPIC` argument is used to ensure that the code generated is independent of the virtual address. Using `PIC` instead of `pic` ensures that the code generated is platform independent.

A new program *myprog* is created to execute the `sleep` function. The code for this simple program can be referenced from the Appendix. When *myprog* is run under the following circumstances, the results obtained were different.

1. Running *myprog* as a regular program and executing as a normal user, the string “I am not sleeping!” is displayed, which is expected as the sleep function in our user-defined library is used due to reading of the `LD_PRELOAD` environment variable.

2. Running *myprog* as a `Set-UID` program and running it with a normal user will result in the program going to sleep for 5 seconds. (To observe the results clearly, `sleep(1)` was amended to `sleep(5)` instead.)

3. Running *myprog* as a `Set-UID` program and exporting the `LD_PRELOAD` environment variable under the root account results in the string being displayed. Due to the security loophole of `LD_PRELOAD`, the user-defined library is only loaded if the environment variable was exported with root and the program run with root.

4. Setting *myprog* as a user1 `Set-UID` program with `LD_PRELOAD` environment variable set under user2 and executing it results in the program going to sleep for 5 seconds.

We analyse the results obtained from the four different conditions and notice that the `LD_PRELOAD` is ignored if the owner and the user executing the program is different, then there will be no execution of the user-defined library.

3.8 Invoking External Programs using `system()` Versus `execve()`

This section looks at using `system()` and `execve()` at executing external programs that are not intended. The `C` code that will be used limits the user to using the `cat` function. Due to this, the owner assumes that the user will cannot execute any write function on the system and will not be able to edit any file. The `C` code for this has been attached in the Appendix.

The code is compiled as a `Set-UID` program with the name *audit* and root as the owner. In particular, we note that the "`;`" operator can be used to initiate the next command. As such, running the program using the following code will allows us to obtain a shell.

```
$ audit "sometext.txt;/bin/sh"
```

In this instance, a dummy file with the file name and extension `sometext.txt` will just act as the argument needed to execute the `cat` function before the shell can be obtained. After the shell has been obtained, system operations such as rm -rf can be performed even without being given root privileges.

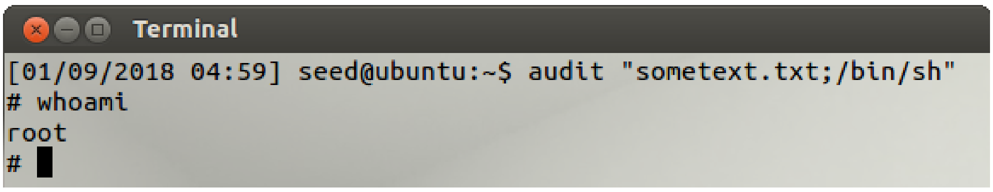

Figure 11: Shell obtained using `system()`

When we use `execve()` instead and compile again, we note that a shell cannot be obtained this time. This is due to `execve` interpreting the entire string as an argument and hence will not be able to find the file.

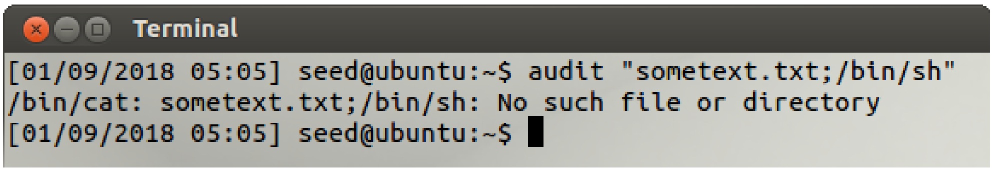

Figure 12: Cannot obtain Shell using `execve()`

3.9 Capability Leaking (Exploit)

In this section, we take a look at capability leaking, where privileges are downgraded after execution but may still be accessible by non-privileged processes. Using `setuid()` sets the effective user ID of the calling process and removes root privileges if the user executing the program is not root.

In the `C` code provided for this section, a vulnerability can be found in the program where root access is revoked but the file that was previously opened under root privileges is still able to have its file edited.

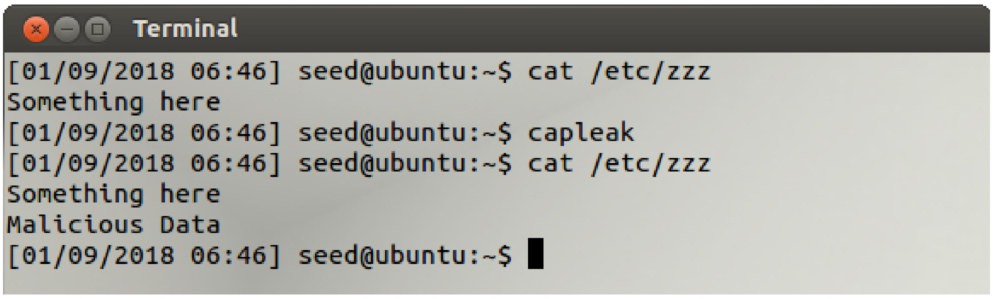

Figure 13: Capability Leaking

In Figure 13, it can be seen that the file was successfully written as the handle that opens the file still retains root privileges and as a result is successful in writing the contents into the file even after `setuid(getuid())` has been executed.

4 Appendix

4.1 Process Inheritance C Code

```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:
            printenv(); /*Child process*/
            exit(0);
        default:
            //printenv(); /*Parent process*/
            exit(0);
    }
}
```

4.2 Execution of execve()

```
#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;
}
```

4.3 Execution of system()

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

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

    return 0;
}
```

4.4 Envrionment Variables & Set-UID Programs

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

extern char **environ;

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

4.5 PATH Environment Variable & Set-UID

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

4.6 Call Shell

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

4.7 sleep() Library

```c
#include <stdio.h>

void sleep (int s)
{
    printf("I am not sleeping!\n");
}
```

4.8 Execute sleep() Function

```c
int main()
{
    sleep(1);
    return 0;
}
```

4.9 Invoking External Programs using `system()` Versus `execve()`

```
#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]);

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

    return 0;
}
```

4.10 Capability Leaking

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

void main()
{
    int fd;
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd==-1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    sleep(1);

    setuid(getuid());

    if (fork()){
        close (fd);
        exit(0);
    } else {
    write(fd, "Malicious Data\n", 15);
    close (fd);
    }
}
```