Lab 3: Environment variables

Sai Sukheshwar Boganadula
sabg22@student.bth.se

Bala Subramanyam Pavan Kumar Kasturi
baka22@student.bth.se

Task 1: Manipulating Environment Variables:

Objective

Learn how to set and unset environment variables using Bash commands.

Steps

1. Print Environment Variables:
    - Use the printenv or env command to print out all environment variables.
    - To print a specific environment variable, use:

        printenv PWD or env | grep PWD

```
[root@3163 ~]# printenv PWD

/root
```

```
[root@sgpllab3163 ~]# printenv

SHELL=/bin/bash

HISTCONTROL=ignoredups

HISTSIZE=1000

HOSTNAME=sgpllab3163

PWD=/root

LOGNAME=root

XDG_SESSION_TYPE=tty

MOTD_SHOWN=pam

HOME=/root

LANG=en_US.UTF-8
```

2. Set and Unset Environment Variables:
    - Use the export command to set an environment variable:

        export VAR_NAME=value

```
[root@3163 ~]# export VAR_NAME=value

[root@3163 ~]# printenv VAR_NAME

value
```

    - Use the unset command to remove an environment variable:

        Unset=VAR_NAME

```
[root@sgpllab3163 ~]# unset VAR_NAME

[root@sgpllab3163 ~]# printenv VAR_NAME
```

Notes

- export and unset are internal commands of Bash and are not separate programs.

  Here is the reference:

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

Objective

Understand how environment variables are inherited by a child process from its parent process in Unix.

Steps

Step 1: Compile and Run the Program

1. Save the following code in a file named child_env.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();
         break;
   }
}
```

```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();
         break;
   }
}
```

2. Compile the program:
   gcc -o child_env child_env.c
3. Run the program and save the output to a file:
   ./child_env > child_output.txt
4. Observation:

The child process will print the environment variables it inherited from the parent process. Save the output in child_output.txt.

```
SSH_CONNECTION=10.212.135.11 50385 192.168.3.163 22

XDG_SESSION_CLASS=user

SELINUX_ROLE_REQUESTED=

TERM=xterm

LESSOPEN=||/usr/bin/lesspipe.sh %s

USER=root

SELINUX_USE_CURRENT_RANGE=

SHLVL=1
```

```
XDG_SESSION_ID=10

XDG_RUNTIME_DIR=/run/user/0

SSH_CLIENT=10.212.135.11 50385 22

which_declare=declare -f

XDG_DATA_DIRS=/root/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share

PATH=/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin

SELINUX_LEVEL_REQUESTED=

DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/0/bus

MAIL=/var/spool/mail/root

SSH_TTY=/dev/pts/0

BASH_FUNC_which%%=() {  ( alias;

 eval ${which_declare} ) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --show-dot $@

}

_=./child_env

OLDPWD=/root
```

Step 2: Modify and Re-run the Program

1. Comment out the printenv() statement in the child process case, and uncomment the printenv() statement in the parent process case:

```
#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();
         break;
   }
}
```

```
#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();
         break;
   }
}
```

2. Compile the modified program:
   gcc -o child_env2 child_env2.c
3. Run the modified program and save the output to another file:
   ./child_env2 > parent_output.txt
4. Observation:

The parent process will print its environment variables. Save the output in parent_output.txt.

```
SSH_CONNECTION=10.212.135.11 50385 192.168.3.163 22

XDG_SESSION_CLASS=user

SELINUX_ROLE_REQUESTED=

TERM=xterm

LESSOPEN=||/usr/bin/lesspipe.sh %s

USER=root

SELINUX_USE_CURRENT_RANGE=
```

```
SHLVL=1

XDG_SESSION_ID=10

XDG_RUNTIME_DIR=/run/user/0

SSH_CLIENT=10.212.135.11 50385 22

which_declare=declare -f

XDG_DATA_DIRS=/root/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share

PATH=/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin

SELINUX_LEVEL_REQUESTED=

DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/0/bus

MAIL=/var/spool/mail/root

SSH_TTY=/dev/pts/0

BASH_FUNC_which%%=() {  ( alias;

 eval ${which_declare} ) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --show-dot $@

}

_=./child_env2

OLDPWD=/root
```

Step 3: Compare the Outputs

1. Use the diff -u command to compare the two output files:

```
diff -u child_output.txt parent_output.txt
```

2. Conclusion:
    - The comparison should show that the environment variables printed by the child process are identical to those printed by the parent process.

```
[root@sgpllab3163 test]# diff -u child_output.txt parent_output.txt

--- child_output.txt    2024-08-20 00:28:14.191068391 +0530

+++ parent_output.txt   2024-08-20 00:35:15.551505465 +0530

@@ -30,5 +30,5 @@

 BASH_FUNC_which%%=() {  ( alias;

  eval ${which_declare} ) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --show-
dot $@
 }
-_=./child_env

+_=./child_env2

 OLDPWD=/root
```

Task 3: Environment Variables and execve()

Objective

Understand how environment variables are affected when a new program is executed via execve().

Steps

Step 1: Compile and Run the Program

1. Save the following code in a file named list_env.c:

```
#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;
}
```

```
#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;

}
```

NOTE: For execve, you need to include the <unistd.h> header.

2. Compile the program:

   gcc -o list_env list_env.c

3. Run the program and observe the output:

   ./list_env

4. Observation:
   - The program executes /usr/bin/env, which prints out the environment variables of the current process. Since execve() is called with NULL for the environment, the new program does not inherit any environment variables from the calling process. The output should be empty or minimal, showing only the default environment variables set by the system.

```
[root@sgpllab3163 test]# gcc -o  list_env2 list_env.c

[root@sgpllab3163 test]# ./list_env2
```

Step 2: Modify and Re-run the Program

1. Change the invocation of execve() to pass the environment variables:

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

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

```c
#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;
}
```

NOTE: For execve, you need to include the <unistd.h> header.

2. Compile the modified program:

   gcc -o list_env_modified list_env.c

3. Run the modified program and observe the output:

   ./list_env_modified

4. Observation:
   - This time, the program executes /usr/bin/env with the environment variables passed from the calling process. The output should display all the environment variables inherited from the parent process.

```
SSH_CONNECTION=10.212.135.12 55203 192.168.3.163 22
XDG_SESSION_CLASS=user

SELINUX_ROLE_REQUESTED=
TERM=xterm

LESSOPEN=||/usr/bin/lesspipe.sh %s

USER=root


SELINUX_USE_CURRENT_RANGE=
SHLVL=1

XDG_SESSION_ID=15

XDG_RUNTIME_DIR=/run/user/0

SSH_CLIENT=10.212.135.12 55203 22

which_declare=declare –f

XDG_DATA_DIRS=/root/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share

PATH=/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin

SELINUX_LEVEL_REQUESTED=
```

```
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/0/bus

MAIL=/var/spool/mail/root

SSH_TTY=/dev/pts/0

BASH_FUNC_which%%=() {  ( alias;
eval ${which_declare} ) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --show-
dot $@
}
_=./list_env

OLDPWD=/root
```

Step 3: Output Comparison and Conclusion

1. Compare the outputs of the two runs:
    - The first run (with NULL environment) shows minimal or no environment variables.
    - The second run (with environ environment) shows all the environment variables inherited from the parent process.
2. Conclusion:
    - When execve() is called with NULL for the environment parameter, the new program does not inherit any environment variables from the calling process.
    - When execve() is called with the environ array, the new program inherits the environment variables from the calling process.

Task 4: Environment Variables and system()

Objective

Understand how environment variables are affected when a new program is executed via the system() function.

Steps

Step 1: Compile and Run the Program

1. Save the following code in a file named system_env.c:

```c
#include <stdio.h>

#include <stdlib.h>



int main() {

   system("/usr/bin/env");

   return 0;
}
```

```c
#include <stdio.h>

#include <stdlib.h>



int main() {

   system("/usr/bin/env");

   return 0;

}
```

2. Compile the program:

```
gcc -o system_env system_env.c
```

3. Run the program and observe the output:

```
./system_env
```

4. Observation:
    - The program executes /usr/bin/env using the system() function, which in turn uses /bin/sh -c to execute the command. The environment variables of the calling process are passed to the new program /bin/sh, and subsequently to /usr/bin/env. The output should display all the environment variables inherited from the parent process.

SSH_CONNECTION=10.212.135.12 55203 192.168.3.163 22

XDG_SESSION_CLASS=user

SELINUX_ROLE_REQUESTED=

TERM=xterm

LESSOPEN=||/usr/bin/lesspipe.sh %s

USER=root

SELINUX_USE_CURRENT_RANGE=

SHLVL=1

XDG_SESSION_ID=15

XDG_RUNTIME_DIR=/run/user/0

SSH_CLIENT=10.212.135.12 55203 22

which_declare=declare –f

XDG_DATA_DIRS=/root/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share

```
PATH=/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin

SELINUX_LEVEL_REQUESTED=

DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/0/bus

MAIL=/var/spool/mail/root

SSH_TTY=/dev/pts/0

OLDPWD=/root

BASH_FUNC_which%%=() {  ( alias;
 eval ${which_declare} ) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --show-dot $@
}
```

Explanation

- The system() function executes a command by invoking /bin/sh -c command. This means it first executes the shell (/bin/sh), which then executes the specified command.
- The system() function internally uses execl() to execute /bin/sh, which in turn uses execve() to execute the command. The environment variables of the calling process are passed to the new program /bin/sh.

Conclusion

- When using the system() function, the environment variables of the calling process are inherited by the new program executed by /bin/sh. This behavior is consistent with the way execve() works when environment variables are passed explicitly.

Task 5: Environment Variables and Set-UID

Programs

Objective

Understand how environment variables are inherited by Set-UID programs and how they can affect the behavior of these programs.

Steps

Step 1: Write the Program

1. **Save the following code in a file named** setuid_env.c:

```
#include <stdio.h>

#include <stdlib.h>



extern char **environ;



void main() {

   int i = 0;

   while (environ[i] != NULL) {

      printf("%s\n", environ[i]);

      i++;
```

```
    }
}
```

```
#include <stdio.h>

#include <stdlib.h>



extern char **environ;



void main() {

    int i = 0;

    while (environ[i] != NULL) {

        printf("%s\n", environ[i]);

        i++;

    }

}
```

Step 2: Compile and Set Up the Program

1. Compile the program:

```
gcc -o setuid_env setuid_env.c
```

2. Change the ownership to root and make it a Set-UID program:

   sudo chown root setuid_env

   sudo chmod 4755 setuid_env

Step 3: Set Environment Variables and Run the Program

1. In your shell (as a normal user, not root), set the following environment variables:

   export PATH=/some/path

   export LD_LIBRARY_PATH=/some/library/path

   export MY_VAR=my_value

2. Run the Set-UID program:

   ./setuid_env

3. Observation:

   Expected Behavior: When you run the Set-UID program, you would expect it to print all environment variables that were set in your shell, including PATH, LD_LIBRARY_PATH, and MYVAR.

   Actual Behavior:

   PATH and MYVAR: These variables will likely be displayed as you set them in your shell. This is because most environment variables are inherited by the child process.

   LD_LIBRARY_PATH: This variable may not appear as you set it or may be ignored. Modern Unix-like operating systems, including Linux, sanitize certain environment variables (like LD_LIBRARY_PATH) when executing a Set-UID program to prevent security risks. This behavior is a security measure to prevent the user from manipulating the program's behavior by injecting malicious libraries.

```
XDG_SESSION_ID=2940
HOSTNAME=sgpllab2152
SELINUX_ROLE_REQUESTED=
TERM=xterm
```

SHELL=/bin/bash

HISTSIZE=1000

SSH_CLIENT=10.212.135.11 56060 22

SELINUX_USE_CURRENT_RANGE=

QTDIR=/usr/lib64/qt-3.3

QTINC=/usr/lib64/qt-3.3/include

SSH_TTY=/dev/pts/0

QT_GRAPHICSSYSTEM_CHECKED=1

USER=james

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=01;05;37;41: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:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=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:*.jpg=01;35:*.jpeg=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:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=01;36:*.au=01;36:*.flac=01;36:*.mid=01;36:*.midi=01;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;36:*.oga=01;36:*.spx=01;36:*.xspf=01;36:

MY_VAR=my_value

MAIL=/var/spool/mail/ james

PATH=/some/path

PWD=/home/ james

LANG=en_US.UTF-8

QT_GRAPHICSSYSTEM=native

KDEDIRS=/usr

SELINUX_LEVEL_REQUESTED=

HISTCONTROL=ignoredups

KRB5CCNAME=KEYRING:persistent:635403156

SHLVL=1

HOME=/home/ james

LOGNAME= james

QTLIB=/usr/lib64/qt-3.3/lib

XDG_DATA_DIRS=/home/ james/.local/share/flatpak/exports/share:/var/lib/flatpak/exports/share:/usr/local/share:/usr/share

SSH_CONNECTION=10.212.135.11 56060 192.168.2.152 22

LESSOPEN=||/usr/bin/lesspipe.sh %s

XDG_RUNTIME_DIR=/run/user/635403156

QT_PLUGIN_PATH=/usr/lib64/kde4/plugins:/usr/lib/kde4/plugins

_=./setuid_env

Conclusion:

The main surprise you might encounter is that while most environment variables are inherited by the Set-UID program, certain critical ones like LD_LIBRARY_PATH are sanitized or ignored for security reasons. This behavior is intended to prevent potential privilege escalation attacks by controlling the dynamic linker/loader behavior.

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

Objective:

The goal of this task is to understand the security risks associated with Set-UID programs, specifically how environment variables like PATH can be exploited by a malicious user to alter the behavior of such programs. This task will demonstrate how a user can hijack a Set-UID program to execute their own code with elevated privileges.

Step 1: Write and Compile the Set-UID Program

Write the Set-UID Program:

Here's the simple Set-UIStep 1: Write and Compile the Set-UID Program

1. **Write the Set-UID Program**:

   Here's the simple Set-UID program:

```
#include <stdlib.h>

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

2. **Compile the Program**:

   Compile the program into an executable:

   gcc -o setuid_ls setuid_ls.c

3. **Change Ownership to Root and Set the Set-UID Bit**:

   sudo chown root setuid_ls
   sudo chmod 4755 setuid_ls

   This makes setuid_ls a Set-UID program owned by root.

Step 2: Modify the PATH Environment Variable

1. **Create a Malicious Script**:

   Create a custom script that will replace the ls command. This script could perform any action you want, but for simplicity, let's have it print a message and check the effective user ID to confirm it’s running as root:

```
echo -e '#!/bin/bash\n echo "Malicious ls command executed"; id' > `~/ls
chmod +x ~/ls
```

2. **Modify the PATH Environment Variable**:

   Prepend your custom script directory to the PATH:

   export PATH=~:$PATH

   This modification causes the shell to look in ~/ls for the ls command before it checks the usual system directories like /bin.

Step 3: Run the Set-UID Program

1. **Execute the Set-UID Program**:

   Run the Set-UID program:

   ./setuid_ls

Step 4: Observation and Explanation

Observation:

- When you run the setuid_ls program after modifying the PATH, it should execute your custom ls script from ~/ instead of the expected /bin/ls.
- The output from your script will be displayed, and the id command should show that the script is running with root privileges (e.g., uid=0(root)).

```
[james@2152 ~]$ ls

Malicious ls command executed

uid=635403156( james) gid=635400513(domain users)
groups=635400513(domain users),635401262(sudoers),635401342(vpn
```

```
users),635401798(sgpl-users),635403242(docker-users)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

[james@2152 ~]$
```

-

Explanation:

- **Why the Attack Works**:
  - The system("ls") function in the Set-UID program does not specify the absolute path to the ls command, which means it will use the PATH environment variable to locate the command. By placing your malicious ls script at the start of PATH, you can trick the Set-UID program into executing your script instead of the legitimate /bin/ls.
  - Since the Set-UID program is running with root privileges, your script is also executed with root privileges.
- **Privilege Escalation**:
  - Your custom script runs with root privileges because the Set-UID program itself is running with root privileges and does not drop them before calling system(). This demonstrates how dangerous it is to use system() in a Set-UID program without specifying absolute paths or sanitizing environment variables.

Step 5: Countermeasures

To prevent this kind of attack, developers should:

1. **Avoid Using system() with Untrusted Input**: Use functions like execve() that allow specifying the absolute path of the command and provide more control over the environment.
2. **Sanitize the Environment**: Clear or carefully set the environment variables before executing external commands.
3. **Use Absolute Paths**: Always use absolute paths for commands executed in a Set-UID program.
4. **Check the Effective UID**: Some shells, like dash in Ubuntu 20.04, mitigate this by dropping privileges if they detect they are running in a Set-UID process. As the task mentions, zsh does not have this countermeasure by default, which is why linking /bin/sh to zsh allows the exploit to work.

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

Objective:

The aim of this task is to understand how the `LD_PRELOAD` environment variable can be used to manipulate the behavior of dynamically linked executables. Specifically, we will explore how `LD_PRELOAD` interacts with Set-UID programs, which run with the privileges of the file owner rather than the executing user.

Background:

`LD_PRELOAD` is an environment variable that specifies a shared library (or libraries) that the dynamic loader/linker should load before any others. This allows users to override functions in standard libraries, potentially altering the behavior of an executable. However, this can be a security risk, particularly in Set-UID programs, where a malicious user might exploit `LD_PRELOAD` to execute arbitrary code with elevated privileges.

Step 1: Create the Necessary Files

Create puts.c:

This code overrides the puts() function in the standard C library (libc) to print "Cruel world!" instead of the standard output.

```
#include <stdio.h>

int puts(const char *s) {

   FILE* fout = stdout;

   fprintf(fout, "Cruel world!\n");

return 0;

}
```

```
/* puts.c */

#include <stdio.h>

int puts(const char *s) {

   FILE* fout = stdout;

   fprintf(fout, "Cruel world!\n");

   return 0;

}
```

2.Compile puts.c into a Shared Library:

The shared library will be used to override the puts() function in other programs.

```
[james@2152 ~]$ gcc -fPIC -g -c puts.c

[james@2152 ~]$ gcc -shared -o libputs.so.1.0.1 puts.o -lc
```

3.Create myprog.c:

This simple program calls the puts() function.
/* myprog.c */

```
#include <stdio.h>



int main(int argc, char *argv[]) {
```

```
    puts("Hello world");

    return 0;

}
```

4.Compile myprog.c:

[james@2152 ~]$ gcc -o myprog myprog.c

Step 2: Experiment with Different Scenarios

Now that you have the necessary files, you can test how LD_PRELOAD affects the behavior of myprog under different conditions.

1. Run myprog as a Regular User:
export LD_PRELOAD=./libputs.so.1.0.1

[james@2152 ~]$ export LD_PRELOAD=./libputs.so.1.0.1

Expected Result: The output should be "Cruel world!" instead of "Hello world," as the puts() function is overridden by your custom implementation.

2. Make myprog a Set-UID Root Program and Run It:

Change the ownership and set the Set-UID bit:

sudo chown root:root myprog

sudo chmod 4755 myprog
./myprog

[james@2152 ~]$ sudo chown root:root myprog

```
[james@2152 ~]$  sudo chmod 4755 myprog

. [james@2152 ~]$  /home/ james/myprog
```

Expected Result: Despite the LD_PRELOAD environment variable being set, the output should be "Hello world" rather than "Cruel world!" This is because, for security reasons, most modern Linux systems ignore LD_* environment variables when running Set-UID programs.

2. **Make myprog a Set-UID Root Program and Run It as Root:**

Run as root after setting LD_PRELOAD:
sudo -i

export LD_PRELOAD=./libputs.so.1.0.1

./myprog

```
[root@2152 ~]# /home/james/myprog
Hello world
```

Expected Result: Similar to the previous case, the output should be "Hello world." Even though you're running as root, LD_PRELOAD is ignored to prevent privilege escalation attacks.

4. Make myprog a Set-UID Alice Program and Run It as a Different User:

Create another user (if not already present) and change ownership:

sudo useradd alice

sudo chown alice:alice myprog

sudo chmod 4755 myprog

```
[james@2152 ~]$  sudo useradd alice

[james@2152 ~]$  sudo chown alice:alice myprog

[james@2152 ~]$  sudo chmod 4755 myprog
```

Run as a different user (e.g., bob):

su - bob

export LD_PRELOAD=./libputs.so.1.0.1

./myprog

```
[james@2152 ~]$  su - bob

[bob@2152 ~]$  export LD_PRELOAD=./libputs.so.1.0.1

[bob@2152 ~]$  sudo /home/james/myprog

Hello world
```

Expected Result: Again, the output should be "Hello world," as LD_PRELOAD is ignored for Set-UID programs.

Step 3: Explain the Behavior

Security Mechanisms:When a Set-UID program is run, especially one owned by root, environment variables such as LD_PRELOAD are often ignored to prevent malicious users from injecting their own code into a privileged program.

The system does this by ensuring that the LD_* variables are cleared or ignored when the program starts if it has elevated privileges.

Environment Variables:LD_PRELOAD works by preloading specified shared libraries before others. For regular programs, this allows for function overriding, but for Set-UID programs, this is a security risk, hence the environment variables are not inherited.

Child Processes:

When a Set-UID program spawns a child process, the child's environment may not inherit LD_* variables to maintain security, especially when the parent process is running with elevated privileges.

Summary of Observations:

Regular Programs: LD_PRELOAD is effective and the custom shared library is loaded, altering program behavior.

Set-UID Programs: LD_PRELOAD is ignored, preserving the security of the system by preventing unauthorized code from being executed with elevated privileges.

This experiment demonstrates how the operating system handles environment variables in the context of Set-UID programs to mitigate potential security risks.