Permissions and System Calls

1 Beginning strace

- Add another user to your system.

  When logged in as this new user, attempt to read a file owned by your primary user. Note that, depending on the file's permissions, it may fail.

  As your primary user, create a file and give it permissions such that the new user will not be able to read it.

  As the new user, use strace to figure out how failure is being communicated to the program attempting to read the file for which you do not have permission.

  Finally, figure out how to do all of the above using sudo.

I created a user called test_user using sudo:

```
$ sudo useradd -m -G wheel test_user

```

Usually, we should set a pssword to the new user:

```
$ sudo passwd test_user
```

but we don't need to do that for the purpose of this exercises. My original user should not be able to open the test_user root directory since it is not root user. To test this, all we have to do is open a shell and travel to the root directory of users, then try to cd into test_user:

```
[@majd ~]$ cd ..
[@majd home]$ ls
majd  test_user
[@majd home]$ cd test_user
bash: cd: test_user: Permission denied
```

Instead of creating a new file in test_user, we can use the roor directory of test_user. We can't strace the "cd" command, but we can for ls:

```
[@majd home]$ strace cd test_user
strace: Cant stat "cd": No such file or directory
[@majd home]$ ls test_user
ls: cannot open directory 'test_user': Permission denied
[@majd home]$ strace ls test_user
execve("/usr/bin/ls", ["ls", "test_user"], 0x7ffdcd018a28 /* 34 vars */) =
    0
brk(NULL)                               = 0x563624e5d000
...
```

After looking through the strace output, we find that the "read" command is failing due to permissions:

```
...
openat(AT_FDCWD, "test_user", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) =
    -1 EACCES (Permission denied)
...
```

This represent how the the failure is being communicated to the ls program. The "openat" system called returns a -1 and set errno to EACCES. If we try to do ls the same file but from a shell that is logged into the test_user, we get:

```
[@majd home]$ su - test_user
Password:
[@test_user ~]$ cd ..
[@test_user home]$ ls
majd  test_user
[@test_user home]$ strace ls test_user1
execve("/usr/bin/ls", ["ls", "test_user"], 0x7ffc57d035e8 /* 11 vars */) =
    0
brk(NULL)                               = 0x562829d1c000
....
openat(AT_FDCWD, "test_user", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) =
    3
....
```

The "su" command stand for switch user. It changed the shell user to another user. We can see that openat() was successful and returned 3. When we look at the man page for open(), we see that 3 is the file descriptor assigned by the system to
test_user:

```
...
The  return  value of open() is a file descriptor, a small, nonnegative
   integer that is an index to an entry in the  processs table  of  open
   file  descriptors.   The  file  descriptor is used in subsequent system
    calls (read(2), write(2), lseek(2), fcntl(2), etc.)  to  refer  to the
    open file.   The file descriptor returned by a successful call will be
     the lowest-numbered file descriptor not currently open for the
   process.
...
```

2 Manually invoke a system call

- Here's an assembly program that invokes the exit system call: exit.asm.

```
BITS 64
GLOBAL _start
SECTION .text
_start:
    mov rax, 231
    mov rdi, 42
    syscall
```

Before you can run it, you'll need to assemble and link it:

```
$ nasm -f elf64 -o exit.bin exit.asm
$ ld -o exit exit.bin

```

Run this program and verify that it calls the exit system call.

Modify this program to invoke a different system call. (The system call table is in the file arch/x86/entry/syscalls/syscall_64.tbl). Verify that it invokes this system call.

Modify the program again to print a message to the standard output. The most straightforward way to accomplish this is by using the write system call. The file descriptor for standard out (ie, the output stream that displays to the terminal) is 1. You'll also need to specify data to output, which you can do like so in your assembly program:

```
SECTION .data
foo: db "hello there",0xa,0
```

Then you can refer to foo in assembly instructions within the .text section.

Modify this program to print a message to the standard output and then exit with an exit code of your choosing.

We can verify that exit.asm calls the exit system call using strace:

```
[@majd 2]$ nasm -f elf64 -o exit.bin  exit.asm
[@majd 2]$ ld -o exit  exit.bin
[@majd 2]$ strace ./exit
execve("./exit", ["./exit"], 0x7fff75e5b6b0 /* 34 vars */) = 0
exit_group(42)                          = ?
+++ exited with 42 +++
```

we see the system call exit_group(42) called with the same integer stored in the rdi register.

Now, we will modify exit.asm code to invoke the "write" system call. From "write" man page, we know it takes three arguments: a file descriptor to write to, a buffer containing the test to write from, and a count variable to tell write how many bytes to write to fd from buffer. How do we pass those parameters in assembly? The file "linux-5.15.14/arch/x86/entry/entry_64.S" contains this info:

```
...
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
 * (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
...
```

So, in rax we must put the "write" call number, rdi the fd, rsi the test buffer, and rdx count. We can find the "write" call number in "linux-5.15.14/arch/x86/entry/syscalls/syscall_64.tbl":

```
...
1 common  write     sys_write
...
```

Given this info, we modify exit.asm to:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, foo
    mov rdx, 11
    syscall
    mov rax, 231
    mov rdi, 3
    syscall

SECTION .data
foo: db "hello there",0xa,0
```

where we call: write(1, "hello there/x0", 11), then exit(3). To verify that this works, we use strace:

```
[@majd 2]$ nasm -f elf64 -o exit.bin  exit.asm
[@majd 2]$ ld -o exit  exit.bin
[@majd 2]$ strace ./exit
execve("./exit", ["./exit"], 0x7ffe5cb09290 /* 34 vars */) = 0
write(1, "hello there", 11hello there)              = 11
exit_group(3)                             = ?
+++ exited with 3 +++
```

We see that there is a "hello there" next to the third argument to write(). This is simply the output of the exit program. Since ls and exit and writing to fd 1 at the same time, the outputs got mixed up. If we run the program withouth strace, we get what we expect:

```
[@majd 2]$ ./exit
hello there
```

3 Test how the kernel handles unexpected syscall arguments

- System calls are, as we've seen, a primary method of interacting with the kernel. One aspect of the kernel's security we might want to test is its ability to handle unexpected system calls or system calls with unexpected arguments. (If the kernel fails to correctly handle a particular system call with particular arguments, we may have found a vulnerability.) Figuring out what system calls and system call arguments might break the

kernel is a difficult task; one way we could make a start on this is to send a bunch of random(ish) as parameters and see if it the kernel crashes.

Write a program that creates a bunch of assembly programs, each of which invokes a different system call with different, random arguments. It should then assemble them, link them, attempt to run them, and summarize the results.

(This is an example of fuzz testing: https://en.wikipedia.org/wiki/Fuzzing.)

We will perform testing using bash. We used https://tldp.org/LDP/abs/html/ as a resource to write the bash program. The intuition is that, from bash, we have to write the Assembly code to a file, compile it, link it and run it. This is easy to do:

```
echo > out.asm  # reset the content of out.asm
# append to out.asm
echo "BITS 64" >> out.asm
echo "GLOBAL _start" >> out.asm
echo "SECTION .text" >> out.asm
echo "_start:" >> out.asm
echo "    mov rax, $[ $RANDOM % 547 + 1 ]" >> out.asm
echo "    mov rdi, 1" >> out.asm
echo "    syscall" >> out.asm
echo "    mov rax, 231" >> out.asm
echo "    mov rdi, 3" >> out.asm
echo "    syscall" >> out.asm
nasm -f elf64 -o out.bin out.asm
ld -o out out.bin
strace ./out 2>res
grep res
```

res is the result file. In order to test random system calls we have to generate a random call number between 1 and 547 (number of system calls from syscall_64.tbl). This is accomplished using $[$RANDOM % 547 + 1]". We are only passing one argument to the random system call generated (which is 1 in rdi). This program only tests one system call at a time, how can we test many? We use a loop:

```
#!/bin/bash
echo > res
x=1
while [ "$x" -le 5 ]
do
  echo > out.asm
  echo "BITS 64" >> out.asm
  echo "GLOBAL _start" >> out.asm
  echo "SECTION .text" >> out.asm
  echo "_start:" >> out.asm
  echo "    mov rax, $[ $RANDOM % 40 + 10 ]" >> out.asm
  echo "    mov rdi, 1" >> out.asm
  echo "    syscall" >> out.asm
  echo "    mov rax, 231" >> out.asm
```

```
15   echo "    mov rdi, 3" >> out.asm
16   echo "    syscall" >> out.asm
17   nasm -f elf64 -o out.bin out.asm
18   ld -o out out.bin
19   strace ./out 2>res
20   grep res
21
22
23   x=$(( $x + 1 ))
24 done
```

The loop is running as long as the variable x is less that or equal to 5. At the end of the loop, we are increasing x by one. We are writing the results of stracing all the 5 system call to res. This is a very basic testing. We have to think carefully about the input and the number of arguments each system call take. We can generate a list of all possible inputs (there are infinitely many- the more to test the merrier), grep the number at type of arg each system call take from the man page, and supply all the possible inputs for each argument looking for when the system call fails. Perhaps I will explore this in the future.