DM510-project2

Danny Jensen
danje14

Group:
Lea Fog-Fredsgaard
Michelle Dung Hoang

March 2018

1 Introduction

In this project the goal is to make a little program which can put messages into a kernel and retrieve these messages. As such most of the code is given but it needs changing so that it throws an error in every imaginable bad situation, instead of writing garbage that can hurt or even destroy the kernel. To do this a whole list of error code is handed out so that every different error is easily recognizable.
As said previously most of the code is all ready given, this code is a put function and get function in user-land which needs to be changed to kernel-land language. Then of cause a main function to test the put and get functions needs to be made, this main is made in user-land and made to interact with the kernel using sys calls.

2 Design

The biggest design choices in this project was to figure out which error codes should be used and when.

2.1 dm510_msgbox_put

Lets look at the errors in the function *dm510_msgbox_put* first:

```
if (length < 0) {
    return -EINVAL;
}
```

So if the user tries to enter a negative string length, an error needs to acur because it is impossible to allocate negive space with *malloc*, or *kmmalloc* in this case.
The error code *EINVAL* is saying that an invalid argument is given which fits perfectly because if a negative argument is given from the user the function needs to stop before *kmalloc* is called trying to allocate negative space.

The next error is if the sting given from the user is not valid:

```
if (buffer == NULL){
    return -ENOMSG;
}
```

So if the string in *buffer* is *NULL* then the error that is returned is *No messageof desired type* which is fitting because there is no message.

There can be many errors even some that is not the users fault, as the next error will show:

```
if (msg==NULL) {
    return -ENOSPC;
}
```

This error cures if *kmalloc* could not allocate the space for *msg*. The cause for this is that there is no more memory that is free and therefor *kmalloc* cant allocate space. The error code is telling *No space left on device* which is fitting for it.

This next error is also not the users fault:

```
if (copy_from_user(msg->message, buffer, length)!=0){
    return -EBADE;
}
```

This error occurs if *copy_from_user* cant put the whole string into the allocated space(this does not mean that if the user typed in *hello* and gave length as 2 that the error would occur, then *copy_from_user* will simply place the 2 first letters inside the kernel). This error is stating that *Invalid exchange* has occured which is fitting because something went wrong while exchanging information from user-land to kernel-land.

The last error in *dm510_msgbox_put* is also int *dm510_msgbox_get* hence the reason why it is the last error mentioned in *dm510_msgbox_put*:

```
 if (access_ok(VERIFY_READ, buffer, length)) {
}else{
    return -EFAULT;
}
```

So what *access_ok* does is it checks if a pointer to a block of memory in user space is valid. if it may not be valid it returns 0 and an error is returned that says *Bad address* because the address of the block of memory is not right.

2.2 dm510_msgbox_get

The first error in *dm510_msgbox_get* is if the user tries to get data out of the kernel but it is empty:

```
if (top == NULL) {
    return -ENODATA;
}
```

In this case the *ENODATA* is pretty self expanatory, *No data available*.

Then the same *access_ok* error is handled the same way.

Now if the length of the space the user allocated for the retrieved message is to short for the actual message, the *dm510_msgbox_get* needs to throw an error:

```
if (length < msg->length){
    return -EMSGSIZE;
}
```

the error code here tells the user *Message too long* this means that the message cant fit into the allocated space in user-land.

Ofcause lust like *copy_from_user*, *copy_to_user* can also produce errors, this error is the same as *copy_to_user*:

```
if (copy_to_user(buffer, msg->message, mlength)!=0){
    return -EBADE;
}
```

Meaning *Invalid exchange* and in this case it means that not all the data could be exchanged between kernel-land and user-land.

3 Implementation

Now that every error is known and how to handle them the implementation is quite easy

3.1 dm510_msgbox_put

first of every error needs an if statement to be caught. These if statements also need to be as soon as possible to both reduce the work load and to ensure that nothing goes wrong inside the kernel. Some of the code has been shown in the Design section so here the parts that are different will only be shown.
The error with the second *kmalloc*, the one that allocates space for the acrual message. If this error is triggered the first *kmalloc* has allocated space for *msg* this space needs to be freed again before the error is returned, or it is just useless space that cant get used. So *kfree* is used here to free *msg* before returning:

```
if (msg->message == NULL){
    kfree(msg);
    return -ENOSPC;
}
```

The same happens if *copy_from_user* gives an error.

```
if (copy_from_user(msg->message, buffer, length)!=0){
    kfree(msg->message);
    kfree(msg);
    return -EBADE;
}
```

Here both the allocated space for the message and the *msg* needs to be freed before the error is returned.

Now to handle the concurrency part, making sure that nothing is interrupted while the message is handled *mutex_lock* is used.

```
mutex_lock(&lock_sys);
if (bottom == NULL) {
    bottom = msg;
    top = msg;
} else {
    msg->previous = top;
    top = msg;
}
mutex_unlock(&lock_sys);
```

this prevents interrupts from other processes.

3.2 dm510_msgbox_get

Just like *dm*510*_msgbox_put* every error handling needs to come as soon as possible. But $mutex_lock$ is activated pretty early in this function, so it need to be deactivated in every error catch. So in

```
if (length < msg->length){
```

and

```
if (copy_to_user(buffer, msg->message, mlength)!=0){
```

Then the only other thing that was needed was a main function to test if the functions could put data into and retrieve data from the kernel. For this *testsystemcall.java* was simply reused. To put data into the kernel a simple *syscall* is used to call *dm*510*_msgbox_put* with a sting and the string length.

```
printf("Test 1, string:\"hello\" length:6 \n");
int out = syscall(__NR_dm510_msgbox_put, ptr, 6);
printf("%s \n\n",strerror(abs(out)));
```

The same goes for getting a message. The only thing that is different is that if an error comes up it needs to check if the returned value is negative, and if it is then write out the error.

```
char msg[50];
printf("Test 4, length:6 \n");
out = syscall(__NR_dm510_msgbox_get, msg, 50);
if (out < 0) {
  printf("%s \n\n",strerror(abs(out)));
}
printf("%s\n\n",msg);
```

4 Test

As shown in the video every test is either successful or it returns the right error line to the user. Sadly it was not possible to test what would happen if another process tried to interrupt the process of putting and retrieving data from the kernel.

5 Discussion

Like already stated it was impossible to get a test where the process was interrupted, but nothing should happen even though some other process tried to interrupt, because *mutex_lock* was used on the critical parts in the program. This means that another process would not be able to interrupt it even if they tried.

6 Conclusion

So all in all everything works as planned and the tests wa successful.
It was a nice small project where one could really learn the basics of writing a kernel implementation code, and get a handle on the importance of error code handling. Also one would learn to write better code because compiling of the kernel takes a long time, so mistakes needed to be fixed before compiling.