

운영체제 실습

[Assignment #2]

Class : 목3
Professor : 최상호 교수님
Student ID : 2021202045
Name : 김예은

Introduction

본 2차 과제는 system call table 336번에 Asmlinkage int os_ftrace(pid_t pid)를 생성한 후(2-1) 커널 재컴파일을 한다. 이 336번 os_ftrace 시스템 콜을 hijack하여 my_ftrace 함수로 대체하는 module을 짜는 프로젝트이다(2-2). 2-3의 경우, 2-2를 통해 wrapping 연습을 한 뒤, 특정 pid에 대하여 파일에 관한 시스템 콜(openat, read, write, lseek, close)을 wrapping하여 trace를 하는 코드를 구현한다.

그렇다면, Kernel Module이란 무엇일까?

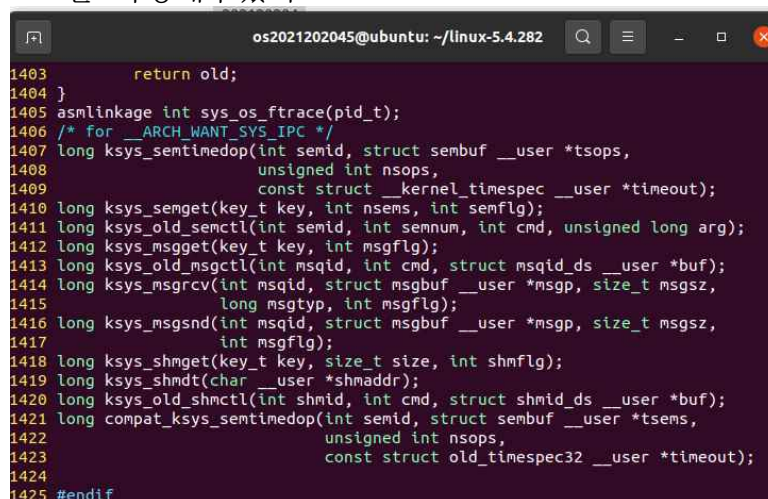
커널 재 컴파일 없이 커널 기능 확장이 가능한, 커널 코드의 일부를 커널이 동작하는 상태에서 load/unload가 가능하다. 내부 main 함수가 없고, init/exit 함수를 통해 모듈 적재 및 모듈 제거를 할 수 있다. 모듈은 때때로 다른 모듈에 의존할 수 있는데, 만약 B 모듈이 A 모듈에 의존한다고 가정하면, A를 먼저 적재 후 B모듈을 적재 해야하며 UNLOAD 시에는, B모듈을 먼저 제거 후, A를 제거해야 한다. A와 B의 의존성은 EXPORT_SYMBOL 및 extern을 통해 이루어질 수 있다.

결과화면

(2-1)

```
333      common io_pgetevents      __x64_sys_io_pgetevents
334      common rseq                __x64_sys_rseq
336      common os_ftrace           __x64_sys_os_ftrace
# don't use numbers 387 through 423, add new calls after the last
# 'common' entry
linux-5.4.282 디렉토리의 system call table에 접근하여 336번에 os_ftrace
라는 system call을 만들어준다.
```

include/linux/syscalls.h에 실습자료와 같은 위치에 위의 os_ftrace system call을 작성해주었다.



```
os2021202045@ubuntu: ~/linux-5.4.282
1403      return old;
1404 }
1405 asmlinkage int sys_os_ftrace(pid_t);
1406 /* for __ARCH_WANT_SYS_IPC */
1407 long ksys_senmtimedop(int semid, struct sembuf __user *tsops,
1408                      unsigned int nsops,
1409                      const struct __kernel_timespec __user *timeout);
1410 long ksys_semget(key_t key, int nsems, int semflg);
1411 long ksys_old_semctl(int semid, int semnum, int cmd, unsigned long arg);
1412 long ksys_msgget(key_t key, int msgflg);
1413 long ksys_old_msgctl(int msqid, int cmd, struct msqid_ds __user *buf);
1414 long ksys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,
1415                 long msgtyp, int msgflg);
1416 long ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz,
1417                 int msgflg);
1418 long ksys_shmget(key_t key, size_t size, int shmflg);
1419 long ksys_shmdt(char __user *shmaddr);
1420 long ksys_old_shmctl(int shmid, int cmd, struct shmid_ds __user *buf);
1421 long compat_ksys_senmtimedop(int semid, struct sembuf __user *tsops,
1422                             unsigned int nsops,
1423                             const struct old_timespec32 __user *timeout);
1424
1425 #endif
```

```

os2021202045@ubuntu: ~/linux-5.4.282
1 #include <linux/kernel.h>
2 #include <linux/syscalls.h>
3
4 SYSCALL_DEFINE1(os_ftrace,pid_t,pid){ //1 parameter
5     printk(KERN_INFO "ORIGINAL ftrace() called! PID is [%d]\n", pid); //print in log level
6     return 0;
7 }

```

system call 함수를 위와 같이 구현해주었는데 SYSCALL_DEFINE1이라는 뜻은 만들 함수는 parameter를 1개 가진다는 뜻이다. 1번째 인자는 만들 함수의 이름이고, 2번째 인자는 os_ftrace 함수의 인자 형, 3번째 인자는 os_ftrace의 인자를 받을 변수명이다.

다음과 같이 커널 Makefile core-y 패턴 검색 후 2번째 결과에 os_ftrace/를 삽입해주었다.

```

953
954 ifeq ($(KBUILD_EXTMOD),)
955 core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ os_
    ftrace/

```

수정된 커널을 컴파일을 한 뒤,

```

LD [M] sound/usb/line6/snd-usb-toneport.ko
LD [M] sound/usb/line6/snd-usb-variax.ko
LD [M] sound/usb/misc/snd-ua101.ko
LD [M] sound/usb/snd-usbmidi-lib.ko
LD [M] sound/usb/snd-usb-audio.ko
LD [M] sound/usb/usx2y/snd-usb-usx2y.ko
LD [M] sound/usb/usx2y/snd-usb-us122l.ko
LD [M] sound/x86/snd-hdmi-lpe-audio.ko
LD [M] sound/xen/snd_xen_front.ko
os2021202045@ubuntu:~/linux-5.4.282$

```

컴파일이 완료된 커널로 재부팅을 하자.

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/syscall.h>
4
5 int main() {
6     pid_t pid = getpid();
7     int result;
8
9     result = syscall(336, pid);
10    if (result == 0) {
11        printf("sys_os_ftrace success! PID: %d\n", pid);
12    } else {
13        perror("sys_os_ftrace failed");
14    }
15
16    return 0;
17 }

```

336번 시스템 콜을 확인하기 위해 test코드를 위와 같이 작성한 뒤, 결과를 확인하면 다음과 같이 336번 시스템 콜 함수를 get pid를 통해 pid값을 가져와 잘 출력하는 것을 확인할 수 있다.

```
os2021202045@ubuntu:~/working$ dmesg | tail -n 3
[ 642.782462] ORIGINAL ftrace() called! PID is [2619]
[ 644.757939] ORIGINAL ftrace() called! PID is [2620]
[ 646.014583] ORIGINAL ftrace() called! PID is [2621]
os2021202045@ubuntu:~/working$
```

(2-2)

2-1에서 만든 os_ftrace를 my_ftrace함수로 wrapping하는 모듈을 제작하여 2-3을 위한 틀을 짤다.

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <linux/syscalls.h>

#ifdef __NR_os_ftrace
#define __NR_os_ftrace 336 // allocate 336
#endif

void **syscall_table;
void *real_os_ftrace; //store original system call(os_ftrace) address

__SYSCALL_DEFINE1(1, my_ftrace, pid_t, pid) {
    printk(KERN_INFO "os_ftrace() hooked! os_ftrace -> my_ftrace");
    return 0;
}

void make_rw(void *addr) { //function that can give authority of reading and writing to the page which includes parameter
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);
    if (pte->pte &~ _PAGE_RW) {
        pte->pte |= _PAGE_RW;
    }
}

void make_ro(void *addr) { //roll back function from giving authority
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);

    pte->pte = pte->pte &~ _PAGE_RW;
}

static int __init os_ftracehooking_init(void) { //call when module loads
    if (!syscall_table) {
        printk(KERN_ERR "NO SYSTEM CALL TABLE\n");
    }

    syscall_table = (void**) kallsyms_lookup_name("sys_call_table"); //find system call table's address
    make_rw(syscall_table); //give authority
    real_os_ftrace = syscall_table[__NR_os_ftrace]; //store original address
    syscall_table[__NR_os_ftrace] = (void *)__x64_sysmy_ftrace; //hooking

    return 0;
}

static void __exit os_ftracehooking_exit(void) {
    syscall_table[__NR_os_ftrace] = real_os_ftrace; //reconnect to original os_ftrace's address
    make_ro(syscall_table); //steal authority
}

module_init(os_ftracehooking_init); //when module loads
module_exit(os_ftracehooking_exit); //when module unloads
MODULE_LICENSE("GPL");
```

위의 코드를 보면, “sys_call_table”을 lookup하여 system call table의 주소를 찾아서 syscall_table 포인터를 통해 가리키게 한다. wrapping 함수란, 기존에 있던 os_ftrace 시스템 콜 함수를 다른 함수로 대체하여 실행시키는 것이다. _SYSCALL_DEFINEx 매크로는 1번째 인자에 만들 함수의 인자 개수, 2번째 인자로 만들 함수의 이름, 3번째 인자로 인자형, 4번째로 인자 변수명을 적어준다. make_rw와 make_ro는 syscall table을 읽고, 쓸 수 있는 권한을 부여하고 부여를 뺏는 함수이다. __init함수와 __exit함수는 module_init/ module_exit에 호출되면서 모듈이 적재될 때, 제거될 때 어떤 코드가 실행 되어야 하는지 적어준다. 위의 코드를 예를 들면, 모듈이 적재될 때는 system call table 주소를 저장한 후, 권한을 부여한다. 그 다음 real 즉, 기존 os_ftrace 함수의 주소를 따로 저장한 뒤, 그 주소에 내가 만든 my_ftrace함수를 대입해 hooking 해준다. 모듈을 제거할 때는 부여했던 권한을 다시 회수한 뒤 기존 os_ftrace 함수의 주소를 반환해준다.

따라서 다음과 같이 system call 336번을 호출하면 os_ftrace가 아닌 my_ftrace함수 내용이 실행되는 것을 확인할 수있다.

꼭 아래와 같이 make로 컴파일 한 후, sudo insmod로 모듈을 적재해주어야한다.

```
os2021202045@ubuntu: ~/ASS2-2$ sudo insmod os_ftracehooking.ko
[sudo] password for os2021202045:
os2021202045@ubuntu: ~/ASS2-2$ gcc os_myftrace_test.c -o os_myftrace_test
os2021202045@ubuntu: ~/ASS2-2$ sudo ./os_myftrace_test
sys_os_ftrace success! PID: 10487
os2021202045@ubuntu: ~/ASS2-2$ sudo ./os_myftrace_test
sys_os_ftrace success! PID: 10489
os2021202045@ubuntu: ~/ASS2-2$ dmesg | tail -n 2
[ 5774.112842] os_ftrace() hooked! os_ftrace -> my_ftrace
[ 5815.862036] os_ftrace() hooked! os_ftrace -> my_ftrace
os2021202045@ubuntu: ~/ASS2-2$
```

(2-3)

2-2에서 미리 짜놓은 코드를 바탕으로 이 코드에 의존하는 iotracehooking.c를 만들어 openat/close/read/write/lseek 함수를 hooking하여 trace하는 코드를 짤다. 우선 위의 5개 함수의 원형을 알기 위해, linux-5.4.282 directory에서 cscope -R 명령어를 사용했다.

```
asmlinkage long sys_fchown(unsigned int fd, uid_t user, gid_t group);
asmlinkage long sys_openat(int dfd, const char __user *filename, int flags,
                           umode_t mode);
462 asmlinkage long sys_lseek(unsigned int fd, off_t offset,
463                          unsigned int whence);
464 asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t
                           count);
465 asmlinkage long sys_write(unsigned int fd, const char __user *buf,
466                           size_t count);
```


C symbol: sys_close

File	Function	Line
0 syscalls.h	<global>	443 asmlinkage long sys_close(unsigned int fd);

아래의 소스코드는 ftracehooking.c의 코드의 일부분으로, iotracehooking.c가 의존하는 c파일이다. my_ftrace함수에서 pid값에 따라 trace 결과를 출력할지, trace할지 케이스를 나누었다. 또한, iotracehooking.c 즉, 실질적으로 원본 함수들을 hooking할 함수들에서 쓰일 변수들을 선언해주고, EXPORT_SYMBOL을 통해 export해주었다. my_trace의 경우 asmlinkage 형으로 선언해주었으므로 2-2와 달리 "syscall_table[__NR_os_ftrace] = my_ftrace"이런 식으로 바로 사용할 수 있다는 장점이 있다. task_struct를 통해 my_ftrace에 들어온 task를 포인트해주고, 그 task의 pid값을 추출하여, 이 값을 trace의 목표 pid로 잡는다.

```
1 #include "ftracehooking.h"
2 #ifndef __NR_os_ftrace
3 #define __NR_os_ftrace 336 // allocate 336
4 #endif
5
6 void **syscall_table;
7 void *real_os_ftrace; //store original system call(os_ftrace) address
8
9 pid_t cur_pid; // pid variable
10 char pc_name[20]; // Process name
11 long read_total_byte = 0;
12 long write_total_byte = 0;
13
14 int open_num = 0;
15 int read_num = 0;
16 int write_num = 0;
17 int lseek_num = 0;
18 int close_num = 0;
19 char file_name[256];
20
21 //value export
22 EXPORT_SYMBOL(read_total_byte);
23 EXPORT_SYMBOL(write_total_byte);
24 EXPORT_SYMBOL(cur_pid);
25 EXPORT_SYMBOL(open_num);
26 EXPORT_SYMBOL(read_num);
27 EXPORT_SYMBOL(write_num);
28 EXPORT_SYMBOL(lseek_num);
29 EXPORT_SYMBOL(close_num);
30 EXPORT_SYMBOL(file_name);
31
32
33 static asmlinkage int my_ftrace(struct pt_regs *regs) {
34     struct task_struct *task;
35     pid_t pid = regs->di; // extract pid from pt_regs
36
37     if (pid == 0) {
38         task = current; // get task_struct of current proces
39         if (task) {
40             strncpy(pc_name, task->comm, sizeof(pc_name) - 1);
41             pc_name[sizeof(pc_name) - 1] = '\0'; // null 종료
42         }
43         printk(KERN_INFO "[2021202045] %s file[%s] stats [x] read - %ld / written - %ld\n",
44             pc_name, file_name, read_total_byte, write_total_byte);
45         printk(KERN_INFO "open[%d] close[%d] read[%d] write[%d] lseek[%d]\n", open_num,
46             close_num, read_num, write_num, lseek_num);
47         printk(KERN_INFO "OS Assignment2 ftrace [%d] End\n", cur_pid);
48     } else {
49         task = pid_task(find_vpid(pid), PIDTYPE_PID); // get task_struct from pid
50         if (!task) {
51             printk(KERN_ERR "Failed to find task for pid: %d\n", pid);
52             return -1; // task error
53         }
54         cur_pid = task->pid;
55         printk(KERN_INFO "OS Assignment2 ftrace [%d] Start\n", cur_pid);
56     }
57     return 0;
58 }
```

다음은 iotracehooking.c에 있는 코드 중 일부이다.

다음과 같이 extern을 이용하여 ftracehooking와 의존관계를 맺는다. 그 다음 original함수들을 가리킬 함수형 포인터들을 선언해주었다.

ftrace_openat함수의 경우 openat 함수를 hooking하는 함수로, original_openat함수를 호출함으로써 기존 openat함수 기능을 실현한 뒤, 현재 pid가 trace 목적 pid 라면(ftracehooking.c에서 저장함) open_num을 증가시켰다. 또한 openat함수의 인자 중 filename인자를 추출하여 copy_from_user를 통해 file_name에 복사시켜 trace출력에 사용되도록 하였다.

```
extern pid_t cur_pid;
extern long read_total_byte;
extern long write_total_byte;
extern int open_num;
extern int write_num;
extern int lseek_num;
extern int read_num;
extern int close_num;

extern char file_name[256];

//point to original function
asm linkage long (*original_openat)(const struct pt_regs *);
asm linkage ssize_t (*original_read)(unsigned int, char __user *, size_t);
asm linkage off_t (*original_lseek)(unsigned int, off_t, unsigned int);
asm linkage long (*original_close)(unsigned int);
asm linkage ssize_t (*original_write)(unsigned int, const char __user *, size_t);

//hooking openat function
asm linkage long ftrace_openat(const struct pt_regs *regs) {
    long fd = original_openat(regs); // call original openat

    if (current->pid == cur_pid) { // check pid which is traced
        open_num++; //increase open number
        const char __user *user_filename; //original filename
        user_filename = (const char __user *)regs->si;
        if (user_filename) {
            if (copy_from_user(file_name, user_filename, sizeof(file_name)-1)==0) { //copy
                file_name[sizeof(file_name) - 1] = '\0'; // null termination
            }
        }
    }

    return fd; //return original openat function's return value
}
```

다음은 ftrace_read, ftrace_write, ftrace_lseek, ftrace_close에 대한 코드로, openat과 마찬가지로 현재 pid가 cur_pid와 같다면 해당 함수부분의 num을 증가시켜 호출 횟수를 업데이트해주었다. 실제 원본 함수들을 호출해준다음, 해당 호출한 함수의 반환값을 그대로 반환해주어 최대한 기존 커널에 영향이 덜 가게끔 해주었다. write_total_byte와 read_total_byte는 해당 함수들이 호출 될 때마다 계속 누적되게끔 코드를 짜주었다. 또한 init과 exit함수에서 2-2에서 했던 것처럼 기존 함수로 다시 가리키게끔 안전하게 처리해주었다.

```

//hooking read function
asmlinkage ssize_t ftrace_read(unsigned int fd, char __user *buf, size_t count) {
    ssize_t read_count = original_read(fd, buf, count);
    if (current->pid == cur_pid) {
        read_num++;
        if (read_count > 0) {
            read_total_byte = read_total_byte + read_count; //read count
        }
    }
    return read_count; //return original read function's return value
}

//hooking write function
asmlinkage ssize_t ftrace_write(unsigned int fd, const char __user *buf, size_t count) {
    ssize_t write_count = original_write(fd, buf, count);
    if (current->pid == cur_pid) {

        /*printf(KERN_INFO "ftrace_write called: fd=%d, count=%zu, write_num=%d\n", fd,
        write_count, write_num);*/

        write_num++; //increase write_num
        if (write_count > 0) {
            write_total_byte = write_total_byte + write_count; //written count
        }
    }
    return write_count; //return original write function's return value
}

//hooking lseek function
asmlinkage off_t ftrace_lseek(unsigned int fd, off_t offset, unsigned int whence){
    off_t new_offset = original_lseek(fd,offset,whence); //call original lseek function
    if(current->pid == cur_pid){
        lseek_num++;
    }
    return new_offset; //return original offset function's return value
}

//hooking close function
asmlinkage long ftrace_close(unsigned int fd){
    long result = original_close(fd); //call original close function
    if(current->pid == cur_pid){
        if(result==0){ //complete close function
            close_num++;
        }
    }
    return result; //return original close function's return value
}

```

test.c 파일은 다음과 같다.

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <errno.h>
9 #include <sys/syscall.h>
10
11 int main()
12 {
13     syscall(336,getpid());
14     int fd=0;
15     char buf[50];
16     fd=open("abc.txt",O_RDWR);
17     for(int i=1;i <=4; ++i)
18     {
19         read(fd,buf,5);
20         lseek(fd,0,SEEK_END);
21         write(fd,buf,5);
22         lseek(fd,i*5,SEEK_SET);
23     }
24
25     lseek(fd,0,SEEK_END);
26     write(fd,"HELLO",6);
27     close(fd);
28     syscall(336,0);
29     return 0;
30 }
31

```


이에 따라 프로그램을 실행해보자.

아래와 같이 make를 해준 다음, insmod를 통해 부모 모듈을 먼저 load를 해준 뒤, 자식 module을 load해준다. test.c에 대한 컴파일명은 test로 하였고 그에 따라 ./test를 해주었다.

```
make[1]: Leaving directory '/home/os2021202045/linux-5.4.282'
os2021202045@ubuntu:~/ASS2-3$ sudo insmod ftracehooking.ko
[sudo] password for os2021202045:
os2021202045@ubuntu:~/ASS2-3$ sudo insmod iotracehooking.ko
os2021202045@ubuntu:~/ASS2-3$ gcc -o test test.c
os2021202045@ubuntu:~/ASS2-3$ ./test
```

그럼 다음과 같이 결과가 나오는 걸 확인할 수 있다.

getpid()로 가져온 pid값과 함께 trace가 시작됐고, test.c에 호출된 만큼 open, close, read, write, lseek의 num값이 잘 출력된다. 또한 read는 5바이트씩 for문 4번, write는 5바이트씩 for문 4번에 마지막 “HELLO”를 쓸 때, 6 byte를 write했으므로 총 byte 누적 수가 각각 20, 26으로 잘 나오는 걸 확인할 수 있다.

```
os2021202045@ubuntu:~/ASS2-3$ dmesg | tail -n 4
[33047.244330] OS Assignment2 ftrace [47283] Start
[33047.244401] [2021202045] test file[abc.txt] stats [x] read - 20 / written - 26
[33047.244402] open[1] close[1] read[4] write[5] lseek[9]
[33047.244402] OS Assignment2 ftrace [47283] End
os2021202045@ubuntu:~/ASS2-3$
```

고찰

2-1에서 커널 재컴파일을 할 때, 시간도 오래 걸리고 중간에 오타가 있는 상태로 컴파일을 해서 에러가 났었다. 다음부터는 조심해야겠다. 2-1, 2-2까지는 수월하게 코드 작성 및 실행을 했지만, 2-3부터는 무한루프인지 iotracehooking 모듈을 넣자마자 리눅스가 멈추고 먹통이 되었다. 그때마다 snapshot을 찍었던 terminal상태로 돌아갔는데, 이 기능이 없었으면 우분투가 다 터졌을 것이다. iotracehooking과 ftracehooking 둘 다에 make_ro/make_rw 함수를 넣어주니 자꾸 rmmod ftracehooking을 할 때 에러가 나서 reboot를 해줘야했다. 그러다 dmesg로 커널을 확인한 결과 syscall table부분에서 문제가 생긴 걸 발견하고 make_ro/make_rw부분이 문제일 것 같아서 iotracehooking.c에서 이 부분을 제거해 주니 잘 해결됐다. 그 다음으로 copy_from_user부분이 계속 안되다가 그 이유가 openat함수에서 filename은 두 번째 인자인가 그런데 자꾸 다른 위치의 인자를 참조해서 그랬던 것임을 발견했다. 다음부터는 이런 실수는 하지 말아야겠다.