

운영체제 실습

[Assignment #2]

Class : 목 3
Professor : 최상호
Student ID : 2020202031
Name : 김재현

Introduction

이번 과제에서는 시스템콜 등록 및 hooking 을 실습합니다.

시스템콜 등록은 다음과 같은 순서로 진행됩니다.

1. **커널 모듈 작성:** 시스템 콜을 등록하기 위한 커널 모듈을 작성합니다. 이 모듈은 시스템 콜 함수와 이를 연결할 수 있는 엔트리 포인트를 정의합니다.
2. **sys_call_table 접근:** 커널에서 시스템 콜 테이블에 접근하여 해당 테이블의 주소를 얻습니다. 이를 통해 시스템 콜이 추가될 위치를 알 수 있습니다.
3. **시스템 콜 추가 및 해제:** 새로운 시스템 콜을 등록하고, 모듈을 제거할 때 시스템 콜을 해제하는 방법을 실습합니다.

hooking 은 다음과 같은 순서로 진행됩니다.

1. **원본 시스템 콜 백업:** 기존의 시스템 콜을 가로채기 전에, 해당 시스템 콜의 주소를 백업하여 나중에 원상 복구할 수 있도록 합니다.
2. **훅킹 함수 작성:** 훅킹할 시스템 콜의 새로운 동작을 정의하는 함수를 작성합니다. 예를 들어, 파일 열기 시스템 콜을 훅킹하여 특정 파일에 접근하는 것을 차단할 수 있습니다.
3. **시스템 콜 대체:** sys_call_table 에서 기존 시스템 콜의 주소를 새로운 훅킹 함수의 주소로 대체합니다.
4. **복원 및 정리:** 실습을 마치면, 원본 시스템 콜을 복원하고 훅킹된 상태를 해제하는 과정도 포함됩니다.

2-1 은 시스템콜 등록

2-2 는 시스템콜 hooking

2-3 은 시스템콜 hooking 을 통한 trace 구현

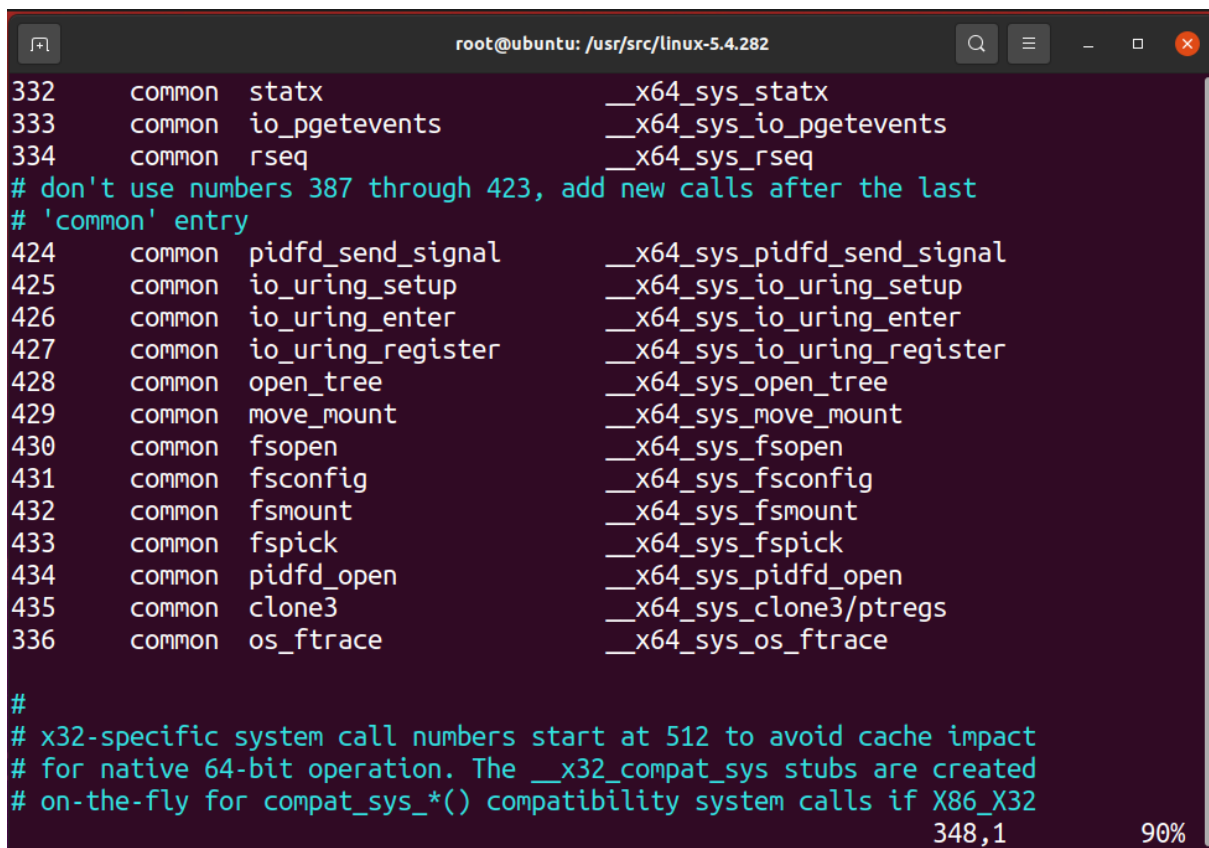
입니다.

결과화면

```
os2020202031@ubuntu:~$ sudo su
root@ubuntu: /home/os2020202031# cat /proc/kallsyms | grep sys_call_table
ffffffff82200260 R sys_call_table
ffffffff82201220 R ia32_sys_call_table
root@ubuntu: /home/os2020202031# cat /boot/System.map-$(uname -r) | grep sys_call
table
ffffffff82200260 R sys_call_table
ffffffff82201220 R ia32_sys_call_table
root@ubuntu: /home/os2020202031#
```

두 주소가 동일한 것을 확인할 수 있다.

2-1.



```
root@ubuntu: /usr/src/linux-5.4.282
332    common  statx          __x64_sys_statx
333    common  io_pgetevents  __x64_sys_io_pgetevents
334    common  rseq           __x64_sys_rseq
# don't use numbers 387 through 423, add new calls after the last
# 'common' entry
424    common  pidfd_send_signal  __x64_sys_pidfd_send_signal
425    common  io_uring_setup     __x64_sys_io_uring_setup
426    common  io_uring_enter     __x64_sys_io_uring_enter
427    common  io_uring_register  __x64_sys_io_uring_register
428    common  open_tree          __x64_sys_open_tree
429    common  move_mount         __x64_sys_move_mount
430    common  fsopen             __x64_sys_fsopen
431    common  fsconfig           __x64_sys_fsconfig
432    common  fsmount           __x64_sys_fsmount
433    common  fspick           __x64_sys_fspick
434    common  pidfd_open        __x64_sys_pidfd_open
435    common  clone3            __x64_sys_clone3/ptregs
336    common  os_ftrace         __x64_sys_os_ftrace

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*( ) compatibility system calls if X86_X32
348,1 90%
```

syscall table 에 추가하기

\$ vi arch/x86/entry/syscalls/syscall_64.tbl 를 입력하여 다음과 같이 추가합니다.


```
root@ubuntu: /usr/src/linux-5.4.282
obj-y = os_ftrace.o
~
~
~
```

\$ vi os_ftrace/Makefile 를 통해 Makefile 을 생성하고

다음과 같은 코드를 작성

```
root@ubuntu: /usr/src/linux-5.4.282
else
    SKIP_STACK_VALIDATION := 1
    export SKIP_STACK_VALIDATION
endif
endif

PHONY += prepare0

export MODORDER := $(extmod-prefix)modules.order

ifeq ($(KBUILD_EXTMOD),)
core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ os_ftrace/

vmlinux-dirs := $(patsubst %/,%, $(filter %/, $(init-y) $(init-m) \
    $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
    $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))

vmlinux-alldirs := $(sort $(vmlinux-dirs) Documentation \
    $(patsubst %/,% $(filter %/, $(init-y) $(core-y) \
```

vi Makefile 에서 /core-y 와 n 으로 해당 단어가 존재하는 부분으로 이동 후 마지막 부분에 os_ftrace/ 추가

sudo make, sudo make modules_install, sudo make install 명령어를 차례로 입력한다.

```
os2020202031@ubuntu: ~/working
#include <stdio.h>

#include <unistd.h> /* syscall() */
#include <sys/syscall.h> /* syscall() */
#include <sys/types.h> /* pid_t */

int main(void)
{
    pid_t pid;
    int ret;

    pid = getpid();

    ret = syscall(336, pid);
    printf("%d\n", ret);

    return 0;
}
~
~
~
~
~
"os_ftrace_test.c" 18L, 295C 18,1 All
```

~/working/os_ftrace_test.c 파일을 생성 후 다음과 같이 타이핑

syscall 에 pid 를 전달하여 반환된 return 값을 출력하는 test 코드입니다.

```
os2020202031@ubuntu: ~/working
os2020202031@ubuntu:~$ cd working/
os2020202031@ubuntu:~/working$ vi os_ftrace_test.c
os2020202031@ubuntu:~/working$ rm app
os2020202031@ubuntu:~/working$ gcc os_ftrace_test.c -o app
os2020202031@ubuntu:~/working$ ./app
Terminal
2547
os2020202031@ubuntu:~/working$ dmesg | tail -n 3
[ 41.749643] audit: type=1400 audit(1728201421.758:52): apparmor="DENIED" operation="open" profile="snap.snap-store.snap-store" name="/etc/PackageKit/Vendor.conf" pid=1884 comm="snap-store" requested_mask="r" denied_mask="r" fsuid=1000 ouid=0
[ 42.550210] audit: type=1400 audit(1728201422.542:53): apparmor="DENIED" operation="open" profile="snap.snap-store.snap-store" name="/etc/appstream.conf" pid=1884 comm="snap-store" requested_mask="r" denied_mask="r" fsuid=1000 ouid=0
[ 73.827766] ORIGINAL ftrace() called! PID is [2547]
os2020202031@ubuntu:~/working$
```

syscall 이 호출되면서 syscall 함수의 printk 또한 잘 출력되는 것을 확인할 수 있습니다.

2-2.

```
os2020202031@ubuntu: ~/moduleTest

#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h> /* kallsyms_lookup_name() */
#include <linux/syscalls.h> /* __SYSCALL_DEFINEx() */
#include <asm/syscall_wrapper.h> /* __SYSCALL_DEFINEx() */

void **syscall_table;
void * real_os_ftrace;

/**
 * Function to be hooked
 * - Low-level macro of the "SYSCALL_DEFINEx" (n: 1-6)
 * - Parameters of this macro
 *   --> 1      : Number of parameters of this system call
 *   --> my_ftrace : Name of the new system call hooked
 *   --> pid_t    : First parameter type of the "my_ftrace" system call
 *   --> pid      : First parameter name of the system call
 */
__SYSCALL_DEFINEx(1, my_ftrace, pid_t, pid)
{
    printk(KERN_INFO "os_ftrace() hooked! os_ftrace -> my_ftrace\n");
    return pid;
}

void make_rw(void *addr)
{
    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)
{
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);

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

static int __init hooking_init(void)
{
    /* Find system call table */
    syscall_table = (void**) kallsyms_lookup_name("sys_call_table");

    /*
     * Change permission of the page of system call table
     * to both readable and writable
     */
    make_rw(syscall_table);

    real_os_ftrace = syscall_table[__NR_os_ftrace];
    syscall_table[__NR_os_ftrace] = __x64_sysmy_ftrace;

    return 0;
}

static void __exit hooking_exit(void)
{
    syscall_table[__NR_os_ftrace] = real_os_ftrace;

    /* Recover the page's permission (i.e. read-only) */
    make_ro(syscall_table);
}

module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");
~
~
~
```

Windows 정품 인증

[설정]으로 이동하여 Windows를 정품 인증합니다.

24,78 All

~/os_ftracehooking.c 파일을 생성하고 다음과 같은 코드를 작성합니다.

```

os2020202031@ubuntu: ~/moduleTest
obj-m := os_ftracehooking.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(CURDIR)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
~

```

Makefile 입니다.

```

os2020202031@ubuntu: ~/moduleTest
#include <stdio.h>

#include <unistd.h> /* syscall() */
#include <sys/syscall.h> /* syscall() */
#include <sys/types.h>

int main(void)
{
    pid_t pid;
    int ret;

    pid = getpid();
    ret = syscall(336, pid);
    printf("%d\n", ret);

    return 0;
}

```

app.c 코드입니다. 커널모듈 적재 전 후 출력값을 비교하여 hooking 이 잘됐는지를 확인합니다.

```

os2020202031@ubuntu:~/moduleTest$ ./app
4008
os2020202031@ubuntu:~/moduleTest$ dmesg | tail -n 3
[ 1073.248729] ORIGINAL ftrace() called! PID is [4003]
[ 1074.750759] ORIGINAL ftrace() called! PID is [4004]
[ 1087.710256] ORIGINAL ftrace() called! PID is [4008]

```

os_ftrace 가 잘 실행됩니다.


```

os2020202031@ubuntu:~/moduleTest$ sudo make
[ Visual Studio Code ]rd for os2020202031:
make -C /lib/modules/5.4.282-os2020202031/build M=/home/os2020202031/moduleTest modules
make[1]: Entering directory '/usr/src/linux-5.4.282'
  CC [M] /home/os2020202031/moduleTest/os_ftracehooking.o
Building modules, stage 2.
MODPOST 1 modules
  CC [M] /home/os2020202031/moduleTest/os_ftracehooking.mod.o
  LD [M] /home/os2020202031/moduleTest/os_ftracehooking.ko
make[1]: Leaving directory '/usr/src/linux-5.4.282'
os2020202031@ubuntu:~/moduleTest$ sudo insmod os_ftracehooking.ko
os2020202031@ubuntu:~/moduleTest$ lsmod | grep os_ftracehooking
os_ftracehooking      16384  0
os2020202031@ubuntu:~/moduleTest$ ./app
4578
os2020202031@ubuntu:~/moduleTest$ dmesg | tail -n 3
[ 1074.750759] ORIGINAL ftrace() called! PID is [4004]
[ 1087.710256] ORIGINAL ftrace() called! PID is [4008]
[ 1118.478053] os_ftrace() hooked! os_ftrace -> my_ftrace

```

os_ftrace 시스템콜을 hijack 하여 my_ftrace 함수로 대체하여 실행했더니 my_ftrace 가 잘 실행되는 것을 확인할 수 있습니다.

```

os2020202031@ubuntu:~/moduleTest$ sudo rmmod os_ftracehooking
os2020202031@ubuntu:~/moduleTest$ lsmod | grep
app                os_ftracehooking.c                .os_ftracehooking.mod.cmd
app.c              os_ftracehooking.ko                os_ftracehooking.mod.o
Makefile           .os_ftracehooking.ko.cmd           .os_ftracehooking.mod.o.cmd
modules.order      os_ftracehooking.mod               os_ftracehooking.o
Module.symvers     os_ftracehooking.mod.c             .os_ftracehooking.o.cmd
os2020202031@ubuntu:~/moduleTest$ lsmod | grep os_ftracehooking
os2020202031@ubuntu:~/moduleTest$ ./app
4610
os2020202031@ubuntu:~/moduleTest$ dmesg | tail -n 3
[ 1087.710256] ORIGINAL ftrace() called! PID is [4008]
[ 1118.478053] os_ftrace() hooked! os_ftrace -> my_ftrace
[ 1192.669588] ORIGINAL ftrace() called! PID is [4610]
os2020202031@ubuntu:~/moduleTest$ █

```

hooking module 을 삭제하고 다시 실행하니 다시 os_ftrace 가 작동하는 것을 확인할 수 있습니다.

2-3.

ftracehooking.c

```
1 #include "ftracehooking.h"
2
3 static void **syscall_table;
4 asmlinkage long (*real_os_ftrace)(const struct pt_regs *);
5
6 pid_t pid = 0; // pid traced
7 char file_name[256]; // name of opened file
8
9 // the number of times that system calls was called
10 int n_openat = 0;
11 int n_read = 0;
12 int n_write = 0;
13 int n_lseek = 0;
14 int n_close = 0;
15
16 int read_byte = 0; // read bytes
17 int write_byte = 0; // write bytes
18
19 static asmlinkage int my_ftrace(const struct pt_regs *regs)
20 {
21     // trace start
22     if (regs->di != 0)
23     {
24         pid = regs->di;
25         n_openat = n_read = n_write = n_lseek = n_close = read_byte = write_byte = 0;
26         memset(file_name, 0, 256);
27         printk(KERN_INFO "OS Assignment 2 ftrace [%d] Start\n", pid);
28     }
29     // trace termination condition
30     if (regs->di == 0)
31     {
32         // show trace result
33         printk(KERN_INFO "[2020202031] %s file[%s] stats [x] read - %ld / written - %ld\n", current->comm, file_name, read_byte, write_byte);
34         printk(KERN_INFO "open[%d] close[%d] read[%d] write[%d] lseek[%d]\n", n_openat, n_close, n_read, n_write, n_lseek);
35         printk(KERN_INFO "OS Assignment2 ftrace [%d] End\n", pid);
36     }
37     return pid;
38 }
39
40 EXPORT_SYMBOL(pid);
41 EXPORT_SYMBOL(file_name);
42 EXPORT_SYMBOL(n_openat);
43 EXPORT_SYMBOL(n_read);
44 EXPORT_SYMBOL(n_write);
45 EXPORT_SYMBOL(n_lseek);
46 EXPORT_SYMBOL(n_close);
47 EXPORT_SYMBOL(read_byte);
48 EXPORT_SYMBOL(write_byte);
49
```

n_openat, n_read, n_write, n_lseek, n_close 는 각각 openat, read, write, lseek, close 시스템콜이 호출된 횟수를 count 하는 변수입니다. read_byte 는 read 가 읽은 바이트 수, write_byte 는 write 가 읽은 바이트 수입니다. pid 는 ftrace 가 추적하고자 하는 프로세스의 id 이며, file_name 은 openat 이 open 한 파일명입니다.

EXPORT_SYMBOL 을 통해 iotracehooking.c 에서 이 변수들을 사용 가능하도록 합니다.

my_ftrace 는 인자로 전달된 수가 0 이 아니면 이를 pid 로 인식하여 해당 pid 를 추적합니다. 인자로 전달된 수가 0 이면 추적을 종료하고 추적 결과를 출력합니다.

```

static int __init hooking_init(void)
{
    /* Find system call table */
    syscall_table = (void**) kallsyms_lookup_name("sys_call_table");

    /*
     * Change permission of the page of system call table
     * to both readable and writable
     */
    make_rw(syscall_table);

    real_os_ftrace = syscall_table[__NR_os_ftrace];
    syscall_table[__NR_os_ftrace] = my_ftrace;

    make_ro(syscall_table);
    return 0;
}

static void __exit hooking_exit(void)
{
    make_rw(syscall_table);
    syscall_table[__NR_os_ftrace] = real_os_ftrace;

    /* Recover the page's permission (i.e. read-only) */
    make_ro(syscall_table);
}

module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");

```

hooking_init 은 시스템콜 테이블에서 os_ftrace 위치에 my_ftrace 로 대체함으로써 os_ftrace 를 my_ftrace 로 hooking 합니다. hooking_exit 은 이를 원상복구합니다.

따라서 os_ftrace syscall 이 호출되면 my_ftrace 가 실행되게 되고, 인자로 전달된 pid 를 my_ftrace 가 trace 하게 됩니다.

iotracehooking.c

```
1  #include "ftracehooking.h"
2
3  static void **syscall_table;
4
5  asmlinkage long (*real_openat)(const struct pt_regs *);
6  asmlinkage long (*real_read)(const struct pt_regs *);
7  asmlinkage long (*real_write)(const struct pt_regs *);
8  asmlinkage long (*real_lseek)(const struct pt_regs *);
9  asmlinkage long (*real_close)(const struct pt_regs *);
10
11  // the number of times that system calls was called
12  extern int pid;
13  extern char file_name[256];
14
15  // the number of times that system calls was called
16  extern int n_openat;
17  extern int n_read;
18  extern int n_write;
19  extern int n_lseek;
20  extern int n_close;
21
22  extern int read_byte; // read bytes
23  extern int write_byte; // write bytes
24
```

openat, read, write, lseek, close 시스템콜을 hooking 하기 위해 위와 같이 asmlinkage long 형 함수포인터로 시스템콜의 주소를 저장해둡니다. void*가 아닌 asmlinkage long*을 사용한 이유는 시스템콜의 원형은 asmlinkage long 형이므로 void*로 저장한 시스템콜을 호출하려면 어차피 asmlinkage long 형으로 형변환을 진행해야하기 때문입니다.

extern 으로 함수를 정의하여 ftracehooking.c 에서 EXPORT_SYMBOL 을 통해 전달한 변수들을 사용합니다.

openat, read, write, lseek, close 시스템콜을 hooking 할 함수들을 정의합니다.

그러기 위해선 위 시스템콜의 원형을 찾아 살펴봐야합니다.

arch/x86/entry/syscalls/syscall_64.tbl 로 가서 시스템콜 테이블에서 시스템콜 원형의 이름을 찾아봅니다.

0	common	read	__x64_sys_read
1	common	write	__x64_sys_write
3	common	close	__x64_sys_close
8	common	lseek	__x64_sys_lseek
257	common	openat	__x64_sys_openat

include/linux/syscall.h 로 가서 시스템콜 원형의 타입과 매개변수를 확인합니다.

```
asmlinkage long sys_openat(int dfd, const char __user *filename, int flags,
                           umode_t mode);
```

매개변수를 4 개 받는 것을 알 수 있습니다. SYSCALL_DEFINE4(name,) 매크로를 사용하면 sys_##name 함수가 정의되므로 name 은 openat 임을 알 수 있습니다.

cscope 를 통해 시스템콜 원형의 정의를 찾아봅니다.

```
Text string: SYSCALL_DEFINE4(openat)
File      Line
0 open.c  1125 SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
1 open.c  1148 COMPAT_SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags, umode_t, mode)

SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
                 umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(dfd, filename, flags, mode);
}
```

똑같은 방법으로 read, write, lseek, close 시스템콜의 원형도 확인합니다.

```
asm linkage long sys_read(unsigned int fd, char __user *buf, size_t count);
```

Text string: SYSCALL_DEFINE3(read

	File	Line
0	read_write.c	595 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
1	read_write.c	1122 SYSCALL_DEFINE3(readv, unsigned long, fd, const struct iovec __user *, vec,
2	read_write.c	1214 COMPAT_SYSCALL_DEFINE3(readv, compat_ulong_t, fd,
3	stat.c	428 SYSCALL_DEFINE3(readlink, const char __user *, path, char __user *, buf,
4	readahead.c	605 SYSCALL_DEFINE3(readahead, int, fd, loff_t, offset, size_t, count)

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    return ksys_read(fd, buf, count);
}
```

```
asm linkage long sys_write(unsigned int fd, const char __user *buf,
                             size_t count);
```

Text string: SYSCALL_DEFINE3(write

	File	Line
0	read_write.c	620 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
1	read_write.c	1128 SYSCALL_DEFINE3(writev, unsigned long, fd, const struct iovec __user *, vec,
2	read_write.c	1323 COMPAT_SYSCALL_DEFINE3(writev, compat_ulong_t, fd,

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                  size_t, count)
{
    return ksys_write(fd, buf, count);
}
```

```
asm linkage long sys_lseek(unsigned int fd, off_t offset,
                             unsigned int whence);
```

Text string: SYSCALL_DEFINE3(lseek

	File	Line
0	read_write.c	322 SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, whence)
1	read_write.c	328 COMPAT_SYSCALL_DEFINE3(lseek, unsigned int, fd, compat_off_t, offset, unsigned int, whence)

```
SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, whence)
{
    return ksys_lseek(fd, offset, whence);
}
```

```
asmlinkage long sys_close(unsigned int fd);
```

Text string: SYSCALL_DEFINE1(close

	File	Line
0	open.c	1198 SYSCALL_DEFINE1(close, unsigned int, fd)

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
    int retval = __close_fd(current->files, fd);

    /* can't restart close syscall because file table entry was cleared */
    if (unlikely(retval == -ERESTARTSYS ||
                 retval == -ERESTARTNOINTR ||
                 retval == -ERESTARTNOHAND ||
                 retval == -ERESTART_RESTARTBLOCK))
        retval = -EINTR;

    return retval;
}
```

위에서 찾은 함수원형을 참고하여 regs 에 들어있는 변수들을 알 수 있습니다.

```
25 static asmlinkage long ftrace_openat(const struct pt_regs *regs)
26 {
27     if(pid == current->pid)
28     {
29         n_openat++;
30
31         // filename copy
32         copy_from_user(file_name, (char*)regs->si, sizeof(regs->si) - 1);
33         file_name[sizeof(regs->si)] = NULL;
34     }
35
36     return real_openat(regs);
37 }
38
```

ftrace_openat 은 원형 openat 시스템콜을 hooking 하여 ftracehooking 에서 추적하고자 하는 pid 와 openat 을 호출한 pid 가 같다면 카운트합니다. 그리고 파일이름을 복사합니다. 마지막으로 원형 openat 시스템콜을 호출합니다.

```
39 static asmlinkage long ftrace_read(const struct pt_regs *regs)
40 {
41     long size_read = real_read(regs);
42     if(pid == current->pid)
43     {
44         n_read++;
45         read_byte += size_read;
46     }
47
48     return size_read;
49 }
```

ftrace_read 은 원형 read 시스템콜을 hooking 하여 ftracehooking 에서 추적하고자 하는 pid 와 read 을 호출한 pid 가 같다면 카운트합니다. 그리고 read 한 바이트 수를 누적합니다. 마지막으로 원형 read 시스템콜을 호출합니다.


```

51 static asmlinkage long ftrace_write(const struct pt_regs *regs)
52 {
53     long size_write = real_write(regs);
54     if(pid == current->pid)
55     {
56         n_write++;
57         write_byte += size_write;
58     }
59
60     return size_write;
61 }

```

ftrace_write 는 원형 write 시스템콜을 hooking 하여 ftracehooking 에서 추적하고자 하는 pid 와 write 를 호출한 pid 가 같다면 카운트합니다. 그리고 write 한 바이트 수를 누적합니다. 마지막으로 원형 write 시스템콜을 호출합니다.

```

63 static asmlinkage long ftrace_lseek(const struct pt_regs *regs)
64 {
65     if(pid == current->pid)
66         n_lseek++;
67
68     return real_lseek(regs);
69 }

```

ftrace_lseek 는 원형 write 시스템콜을 hooking 하여 ftracehooking 에서 추적하고자 하는 pid 와 lseek 를 호출한 pid 가 같다면 카운트합니다. 그리고 마지막으로 원형 lseek 시스템콜을 호출합니다.

```

71 static asmlinkage long ftrace_close(const struct pt_regs *regs)
72 {
73     if(pid == current->pid)
74         n_close++;
75
76     return real_close(regs);
77 }

```

ftrace_close 는 원형 close 시스템콜을 hooking 하여 ftracehooking 에서 추적하고자 하는 pid 와 close 를 호출한 pid 가 같다면 카운트합니다. 그리고 마지막으로 원형 close 시스템콜을 호출합니다.

```

99  static int __init hooking_init(void)
100  {
101      /* Find system call table */
102      syscall_table = (void**) kallsyms_lookup_name("sys_call_table");
103
104      /*
105       * Change permission of the page of system call table
106       * to both readable and writable
107       */
108      make_rw(syscall_table);
109
110      real_openat = syscall_table[__NR_openat];
111      syscall_table[__NR_openat] = ftrace_openat;
112
113      real_read = syscall_table[__NR_read];
114      syscall_table[__NR_read] = ftrace_read;
115
116      real_write = syscall_table[__NR_write];
117      syscall_table[__NR_write] = ftrace_write;
118
119      real_lseek = syscall_table[__NR_lseek];
120      syscall_table[__NR_lseek] = ftrace_lseek;
121
122      real_close = syscall_table[__NR_close];
123      syscall_table[__NR_close] = ftrace_close;
124
125      make_ro(syscall_table);
126
127      return 0;
128  }
129
130  static void __exit hooking_exit(void)
131  {
132      make_rw(syscall_table);
133
134      syscall_table[__NR_openat] = real_openat;
135      syscall_table[__NR_read] = real_read;
136      syscall_table[__NR_write] = real_write;
137      syscall_table[__NR_lseek] = real_lseek;
138      syscall_table[__NR_close] = real_close;
139
140      /* Recover the page's permission (i.e. read-only) */
141      make_ro(syscall_table);
142  }
143
144  module_init(hooking_init);
145  module_exit(hooking_exit);
146  MODULE_LICENSE("GPL");
147

```

hooking_init 은 openat, read, write, lseek, close 시스템콜을 real_##name 에 저장해두고 해당 시스템콜 테이블 위치를 ftrace_##name 로 대체합니다. hooking_exit 은 이를 원상복구합니다.

```
obj-m += ftracehooking.o
obj-m += iotracehooking.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(CURDIR)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean

~
```

Makefile 입니다.

```
os2020202031@ubuntu:~/hooking$ ls
abc.txt ftracehooking.c ftracehooking.h iotracehooking.c Makefile Test.c
os2020202031@ubuntu:~/hooking$ make
make -C /lib/modules/5.4.282-os2020202031/build M=/home/os2020202031/hooking modules
make[1]: Entering directory '/usr/src/linux-5.4.282'
CC [M] /home/os2020202031/hooking/ftracehooking.o
Building modules, stage 2.
MODPOST 2 modules
CC [M] /home/os2020202031/hooking/ftracehooking.mod.o
LD [M] /home/os2020202031/hooking/ftracehooking.ko
CC [M] /home/os2020202031/hooking/iotracehooking.mod.o
LD [M] /home/os2020202031/hooking/iotracehooking.ko
make[1]: Leaving directory '/usr/src/linux-5.4.282'
```

```

os2020202031@ubuntu:~/hooking$ lsmod | grep hooking
os2020202031@ubuntu:~/hooking$ sudo insmod ftracehooking.ko
[sudo] password for os2020202031:
os2020202031@ubuntu:~/hooking$ sudo insmod iotracehooking.ko
os2020202031@ubuntu:~/hooking$ lsmod | grep hooking
iotracehooking          16384  0
ftracehooking           16384  1 iotracehooking
os2020202031@ubuntu:~/hooking$

os2020202031@ubuntu:~/hooking$ gcc Test.c
os2020202031@ubuntu:~/hooking$ ./a.out
os2020202031@ubuntu:~/hooking$ dmesg | tail -n 5
[ 7930.502270] ftracehooking: module verification failed: signature and/or required key missing - tainting kernel
[ 7976.141117] OS Assignment 2 ftrace [4542] Start
[ 7976.152342] [2020202031] a.out file[abc.txt] stats [x] read - 0 / written - 0
[ 7976.152344] open[1] close[1] read[4] write[5] lseek[9]
[ 7976.152346] OS Assignment2 ftrace [4542] End
os2020202031@ubuntu:~/hooking$

os2020202031@ubuntu:~/hooking$ gcc Test.c
os2020202031@ubuntu:~/hooking$ ./a.out
os2020202031@ubuntu:~/hooking$ dmesg | tail -n 5
[ 8616.724782] OS Assignment2 ftrace [5199] End
[ 8642.656064] OS Assignment 2 ftrace [5250] Start
[ 8642.656252] [2020202031] a.out file[abc.txt] stats [x] read - 20 / written - 26
[ 8642.656254] open[1] close[1] read[4] write[5] lseek[9]
[ 8642.656255] OS Assignment2 ftrace [5250] End
os2020202031@ubuntu:~/hooking$

```

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
    syscall(336, getpid());
    int fd = 0;
    char buf[50];
    fd = open("abc.txt", O_RDWR);
    for (int i = 1; i <= 4; ++i)
    {
        read(fd, buf, 5);
        lseek(fd, 0, SEEK_END);
        write(fd, buf, 5);
        lseek(fd, i+5, SEEK_SET);
    }
    lseek(fd, 0, SEEK_END);
    write(fd, "HELLO", 6);
    close(fd);

    syscall(336, 0);

    return 0;
}

```

Test.c 입니다.

syscall(336, getpid());를 실행한 후 open 은 1 번, read 는 4 번, write 는 5 번 lseek 는 9 번 close 는 1 번 실행되는 것을 예상할 수 있는데 dmesg 로 확인한 결과 역시 이와 동일한 것을 알 수 있습니다.

또한 read bytes 는 $5 \times 4 = 20$, write bytes 는 $5 \times 4 + 6 = 26$ 임을 알 수 있습니다.

고찰

`printk(KERN_INFO "");` 코드를 작성할 때 ""는 끝에 개행문자를 넣지 않은 채 컴파일과 모듈적재를 진행했습니다. 그랬더니 테스트파일을 실행시키고 `dmesg` 를 통해 출력을 확인하면 이전 결과가 나오는 문제가 있었습니다.

`printk` 역시 `print` 류의 함수이므로 출력버퍼가 존재할 것입니다. 출력버퍼는 개행문자를 만나야 버퍼를 출력하므로 개행문자를 넣어줬더니 문제가 해결됐습니다.

이를 통해 문자열에 꼭 개행문자를 붙여줘야한다는 것을 알게됐습니다.

`static asmlinkage int my_ftrace(const struct pt_regs *regs)`을 사용하라는 조건 때문에 `SYSCALL_DEFINE` 의 정의를 찾아봤습니다. `SYSCALL_DEFINE` 매크로 역시 결국은 `asmlinkage long` 형으로 함수를 정의하는 매크로였다는 것을 알게됐습니다.

Reference