Kernel Lab Report

2019-11730 신현지

Screenshot

```
hyeonji@hyeonji-Standard-PC-Q35-ICH9-2009:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.5 LTS
Release:        20.04
Codename:       focal
```

```
hyeonji@hyeonji-Standard-PC-Q35-ICH9-2009:~$ uname -ar
Linux hyeonji-Standard-PC-Q35-ICH9-2009 5.15.0-67-generic #74~20.04.1-Ubuntu SMP
 Wed Feb 22 14:52:34 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
```

```
[ 6785.609805] dbfs_ptree module initialize done
[ 6793.724557] dbfs_ptree module exit
```

```
[ 6865.190890] dbfs_paddr module initialize done
[ 6867.692659] dbfs_paddr module exit
```

Goal of Kernel Lab

Part #1

Part #1 에서는, 특정 pid로부터 프로세스 트리를 추적하는 Kernel Module을 프로그래밍한다. 구체적으로, `echo [pid] >> input` 으로 write를 한 후 `cat ptree` 로 read 하면,

```
init (1)
xfce4-panel (2306)
xfce4-terminal (2408)
bash (2413)
sudo (2881)
```

와 같이 출력되어야 한다.

Part #2

Part #2 에서는, virtual address를 사용해 physical address를 찾는 Kernel Module을 프로그래밍한다. 구체적으로, `./app` 으로 테스트 프로그램을 실행할 경우

```
[TEST CASE] PASS
```

이 출력되어야 한다.

두 파트를 통해 Debug File System interface를 기반으로 하는 Linux Kernel Module 프로그래밍을 이해할 수 있다.

Implement

Part #1

write_pid_to_input

```
if (copy_from_user(buffer, user_buffer, length)) {
        return -EFAULT;
}
sscanf(buffer, "%u", &input_pid);
```

1. user space에서 kernel space로 값을 복사한 후, pid를 `input_pid`에 저장한다.

```
curr = pid_task(find_get_pid(input_pid), PIDTYPE_PID);
if (!curr) {
        return -EINVAL;
}
```

2. `find_get_pid`로 `pid_struct`를 가져온 후, `pid_task`로 `task_struct`를 가져온다. task_struct가 없는 경우 `EINVAL`을 반환한다.

```
memset(data, 0, 1024);
```

3. 결과 문자열을 초기화한다.

```
while(curr) {
        sprintf(temp_buffer, "%s (%d)\n", curr->comm, curr->pid);
        strcat(temp_buffer, data);
        strcpy(data, temp_buffer);

        if (curr->pid != 1) {
                curr = curr->parent;
        } else {
                break;
        }
}
```

4. `task_struct`에서 점차 부모 task로 올라가며, command와 pid로 결과 문자열을 생성한다.

file_operations

```
static const struct file_operations dbfs_fops = {
        .write = write_pid_to_input,
};
```

5. 파일에 write를 시도하면 `write_pid_to_input` 함수를 호출하도록 한다.

dbfs_module_init

```
dir = debugfs_create_dir("ptree", NULL);
if (!dir) {
        printk("Cannot create ptree dir\n");
        return -1;
}
```

6. `ptree` directory를 생성한다. 에러가 발생한 경우 로그를 출력하고 중단한다.

```
inputdir = debugfs_create_file("input", 0222, dir, NULL, &dbfs_fops);
if (!inputdir) {
        printk("Cannot create input file\n");
        return -1;
}
```

7. `input` file을 생성한다. write 권한만 필요하므로 mode는 0222, 부모 디렉토리는 dir(`ptree` directory), file_operations는 5에서 정의한 struct를 사용한다. 에러가 발생한 경우 로그를 출력하고 중단한다.

```
data = (char *)kmalloc(1024 * sizeof(char), GFP_KERNEL);
blob = (struct debugfs_blob_wrapper *)kmalloc(sizeof(struct debugfs_blob_wrapper),
GFP_KERNEL);

blob->data = data;
blob->size = 1024 * sizeof(char);

ptreedir = debugfs_create_blob("ptree", 0444, dir, blob);
if (!ptreedir) {
        printk("Cannot create ptree file\n");
        return -1;
}
```

8. 프로세스 트리 추적 결과를 담을 문자열 `data`와, debugfs_blob_wrapper `blob`에 메모리를 할당한다. `ptree` file을 생성한다. read 권한만 필요하므로 mode는 0444, 부모 디렉토리는 dir(`ptree` directory)를 사용한다. 에러가 발생한 경우 로그를 출력하고 중단한다.

dbfs_module_exit

```
debugfs_remove_recursive(dir);
```

9. dir(`ptree` directory)과 하위 file을 삭제한다.

```
kfree(blob);
kfree(data);
```

10. `kmalloc`으로 할당한 메모리를 해제한다.

```
module_init(dbfs_module_init);
module_exit(dbfs_module_exit);
```

11. `dbfs_module_init`과 `dbfs_module_exit` 함수를 커널에 등록한다.

Part #2

read_output

```
unsigned long mask = (1ul << 48) - 1;
```

1. 48비트만 남기고 자르기 위해 48비트가 1로 채워진 마스크를 선언한다.

```
if (copy_from_user(&pckt, user_buffer, length)) {
        return -EFAULT;
}
```

2. user space에서 kernel space로 값을 복사해 `pckt`에 저장한다.

```
task = pid_task(find_get_pid(pid), PIDTYPE_PID);
if (!task) {
        return -EINVAL;
}
```

3. `find_get_pid`로 `pid_struct`를 가져온 후, `pid_task`로 `task_struct`를 가져온다. task_struct가 없는 경우 `EINVAL`을 반환한다.

```
mm = task->mm;

pgd = pgd_offset(mm, vaddr);
if (pgd_none(*pgd) || pgd_bad(*pgd)) {
        return -EINVAL;
}

p4d = p4d_offset(pgd, vaddr);
if (p4d_none(*p4d) || p4d_bad(*p4d)) {
        return -EINVAL;
}

pud = pud_offset(p4d, vaddr);
if (pud_none(*pud) || pud_bad(*pud)) {
        return -EINVAL;
}

pmd = pmd_offset(pud, vaddr);
if (pmd_none(*pmd) || pmd_bad(*pmd)) {
        return -EINVAL;
}

pte = pte_offset_kernel(pmd, vaddr);
if (pte_none(*pte)) {
        return -EINVAL;
}
```

4. task_struct의 `mm`으로 page walk를 하며 `pgd`, `p4d`, `pud`, `pmd`, `pte` 순으로 얻어낸다. none이거나 bad인 경우 `EINVAL`을 반환한다.

```
paddr = (pte_val(*pte) & PAGE_MASK) | (vaddr & ~PAGE_MASK);
paddr = paddr & mask;
```

5. `pte_val(*pte) & PAGE_MASK`으로 PPN, `vaddr & ~PAGE_MASK`으로 PPO을 추출해 physical address를 계산한다.

```
pckt.paddr = paddr;

if (copy_to_user(user_buffer, &pckt, sizeof(pckt))) {
        return -EFAULT;
}
```

6. pckt의 paddr을 계산값으로 설정하고, user space로 값을 복사한다.

file_operations

```
static const struct file_operations dbfs_fops = {
        .read = read_output,
};
```

7. 파일에 read를 시도하면 `read_output` 함수를 호출하도록 한다.

dbfs_module_init

```
dir = debugfs_create_dir("paddr", NULL);
if (!dir) {
        printk("Cannot create paddr dir\n");
        return -1;
}
```

8. `paddr` directory를 생성한다. 에러가 발생한 경우 로그를 출력하고 중단한다.

```
output = debugfs_create_file("output", 0444, dir, NULL, &dbfs_fops);
if (!output) {
        printk("Cannot create output file\n");
        return -1;
}
```

9. `output` file을 생성한다. read 권한만 필요하므로 mode는 0444, 부모 디렉토리는 dir(`paddr` directory), file_operations는 7에서 정의한 struct를 사용한다. 에러가 발생한 경우 로그를 출력하고 중단한다.

dbfs_module_exit

```
debugfs_remove_recursive(dir);
```

10. dir(`paddr` directory)과 하위 file을 삭제한다.

```
module_init(dbfs_module_init);
module_exit(dbfs_module_exit);
```

11. `dbfs_module_init`과 `dbfs_module_exit` 함수를 커널에 등록한다.

Difficult

Part #1

1. `copy_from_user`를 사용하지 않고 바로 `sscanf`를 실행하려 했을 때, 어떤 오류 메시지도 나타나지 않고 바로 종료되는 문제가 있었다. 이로 인해 버그의 원인을 찾고 수정하는 것이 상당히 어려웠다. 또한 버그 발생 시 매번 재부팅을 시도해야 했기 때문에 디버깅 과정도 복잡했다.
2. 프로세스 트리의 추적 결과를 어떻게 `ptree` 파일에서 read 동작 시 보여줄 수 있을지에 대한 고민되었다. 처음에는 `ptree` 파일에 `file_operations`를 이용하여 read 동작 시 버퍼에 저장된 결과를 보여주는 방법을 생각했다. 그러나 스켈레톤 코드에서는 `ptree` 파일에 `file_operations`를 사용하지 않아, 다른 방법이 필요할 것으로 생각했다. 이후 debugfs API에서 blob 기능을 발견했고, blob 파일을 읽으면 `data` 포인터가 가리키는 데이터를 볼 수 있음을 알게 되었다. 이를 통해 `file_operations` 없이도 구현이 가능하다는 것을 알게 되었다.

Part #2

1. `app.c` 파일을 보고 입력이 packet struct 형태로 들어온다는 것을 파악하는데 어려움을 겪었다.
2. 페이지 워크 함수를 찾는 것과, include 오류를 해결하는 것이 힘들었다. 슬라이드에서 제공된 include 경로를 참조하여 직접 파일을 열어 함수가 존재하는지 확인하고 include 문을 수정했다.
3. physical address를 계산하는 과정을 찾는 것이 어려웠다. 커널이 `PAGE_MASK`라는 것을 기본적으로 제공한다는 것을 몰랐다.
4. 48비트로 자르지 않으면 결과가 틀리게 나오는 문제가 있었고, 이 원인을 찾는 것이 어려웠다.

Surprising

1. Loadable Kernel Module의 구조를 처음으로 배웠다. `module_init()`과 `module_exit()`를 통해 커널 모듈이 시스템에 삽입되거나 제거될 때 실행할 함수를 등록할 수 있다는 점이 흥미로웠다.
2. Debug File System을 이용하여 user space과 kernel space 간에 정보를 주고받을 수 있다는 사실을 알게 되었다. Debugfs의 기획 의도와는 다르지만, 이를 효율적으로 활용하는 좋은 아이디어라고 생각했다.