System Programming 2019-01
2016-12146 이세리
Kernel Lab

Kernel Lab Report

Parallels에서 Virtual Box로 개발환경을 바꾸면서 캡쳐했던 사진과 다른 UBUNTU 개발환경을 설치하게 되었습니다. (4 level page table)

ubuntu 16.04.1 LTS + kernel 4.4.0

1. Background Research

The goal of Kernel lab is to add Loadable Kernel Module and use Debug File System(debugfs) to understand the difference between kernel-level programming and user-level programming.

1)debugfs

Debug File System(debugfs) is a special file system available in the linux kernel. It provides simple way for kernel developers to make information available to user space. User space developers can access Linu Kernel information easily using debugfs.

Before seeing the debugs APIs, we have to know how to connect the functions to file operations interfaces in Linux Kernel.

```
#include <linux/fs.h>
static ssize_t write_op(struct file *fp, const char __user *user_buffer, size_t length, loff_t *position); //running when file operation is called
static const struct file_operations my_fops = {
        .write = write_op,
}
```

<from https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt>

<from https://www.fsl.cs.sunysb.edu/kernel-api/re464.html>

<from https://www.fsl.cs.sunysb.edu/kernel-api/re471.html>

Unlike /proc, which is only meant for information about a process, or sysfs, which has strict one-value-per-file rules, debugfs has no rules at all. Developers can put any information they want there. debugfs is typically mounted with a command like :

mount -t debugs none /sys/kernel/debug

The debugfs root directory is accessible only to the root user by default. First, create at least one directory to hold a set of debugfs files:

struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);

This call, if successful, will make a directory called name underneath the indicated parent directory. If the parent is null, the directory will be created in the debugfs root.

On success, the return value is a struct dentry pointer which can be used to create files in the directory and to clean it up at the end.

The most general way to create a file within a debugfs directory is with:

struct dentry *debugfs_create_file(const char *name, mode_t mode, struct dentry *parent, void *data, const struct file_operations *fops);

//data : a pointer to something that the caller will want to get to later on. The inode.i_private pointer will point to this value on the open call.

Here, name is the name of the file to create, mode describes the access permissions the file should have, parent indicates the directory which should hold the file, data will be stored in the i_private field of the resulting inode structure, and fops is a set of file operations which implement the file's behavior.

Another option is exporting a block of arbitrary binary data, with this structure and function:

```
struct debugfs_blob_wrapper{
        void *data;
        unsigned long size;
}
```

struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob);

//blob: a pointer to a struct debugfs_blob_wrapper which contains a pointer to the blob data and the size of the data. If the mode variable is so set it can be read from.

//This function creates a file in debugfs with the given name that exports blob->data as a binary blob.

A read of this file will return the data pointed to by the debugfs_blob_wrapper structure. Some drivers use "blobs" as a simple way to return several lines of (static) formatted text output. This function can be used to export binary information, but there does not appear to be any code which does so in the mainline. Note that all files created with debugfs_create_blob() are read-only.

Once upon a time, debugs users were required to remember the dentry pointer for every debugs file they created so that all files could be cleaned up. We live in more civilized times now, though, and debugs users can call

void debugfs_remove_recursive(struct dentry *dentry);

If this function is passed a pointer for the dentry corresponding to the top-level directory, the entire hierarchy below that directory will be removed.

<from https://medium.com/hungys-blog/linux-kernel-process-99629d91423c>

<from http://tuxthink.blogspot.com/2012/07/module-to-find-task-from-its-pid.html>

2) task_struct

Each entity that can be scheduled are allocated a process descriptor task_struct, which is defined in <linux/sched.h>

```
struct task_struct {
        /* ... */
        struct list_head run_list;
        /* ... */
        struct task_struct *real_parent;
}
```

Here is a module which find the task_struct of a process from its pid. To find the task_struct of a process, we can make use of the function pid_tast defined in kernel/pid.c

struct task_struct *pid_task(struct pid *pid, enum pid_type type);

*arguments:

-pid: pointer to the struct pid of the process

-pid_type: PIDTYPE_PID, PIDTYPE_PGID, PIDTYPE_SID, PIDTYPE_MAX

To find the pid structure if we have the pid of a process, we can use the function find_get_pid which is also defined in kernel/pid.c

struct pid *find_get_pid(pid_t nr);

<from https://medium.com/hungys-blog/linux-kernel-process-99629d91423c>

3)list_head

linux kernel defines list_head data structure, whose only fields next and rev represent the forward and back pointers of a generic doubly linked list element, respectively.

It is important to note, however, that the pointers in a list_head field store the addresses of other list_head fields rather than the addresses of the whole data structure in which the list_head structure is included.

```
struct list_head {
        struct list_head *next, *prev;
}
```

<from https://norux.me/18>

4)sprintf, snprintf

while printf prints what was entered in the shell, sprintf prints what was entered in char * buffer. snprintf is sprintf with size parameter added to prevent buffer overflow.

```
int sprintf(char *buffer, const char *format, …)
int snprintf(char *buffer, int buf_size, const char *format, …)
ex) int main(int argc, char ** argv){
        char buf[256];
        int len;
        int i;
        len = sprintf(buf, "Hello,\n");
        for (i = 0; i < 5; i++) {
            len += sprintf(buf+len, "%d ", i);
        }
        puts(buf);
}
```

It adds len to bug because if it does not add the previous char array's length, it will keep writing in the start address.

2. Process Tree Tracing

First, I created dir to create ptree directory. And then I created input and ptree file. This is implemented by the following 3 lines of code.

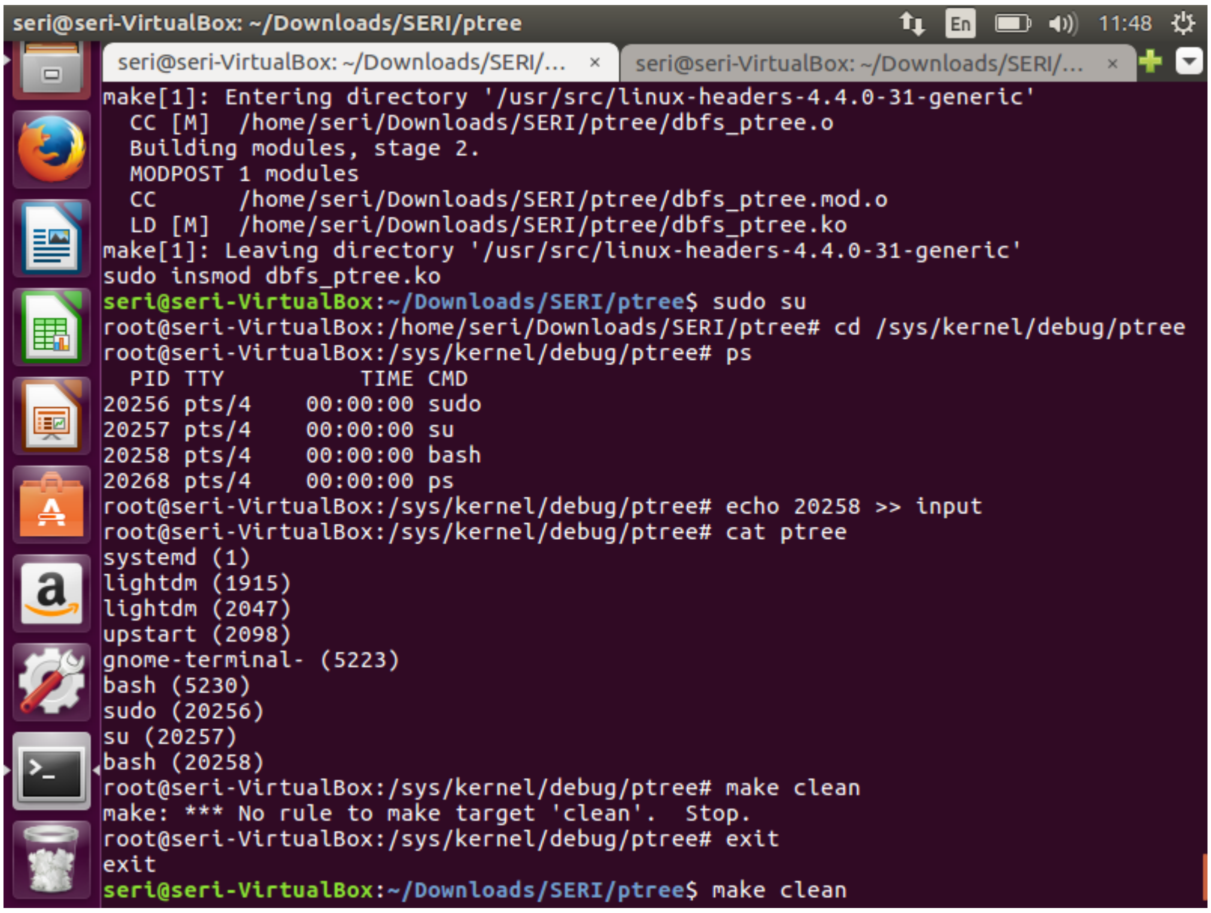
```
debugfs_create_dir("ptree", NULL);
debugfs_create_file("input", 0444, dir, NULL, &dbfs_fops);
debugfs_create_blob("ptree", 0444, dir, &blob);
```

Notice I passed dir as parent parameter to create input and ptree file below the ptree directory. The mode 0444 was chosen because TAs have told me 0777 is going to cause problems because of execution permission. 0444 only allows read and write.

For blob, I created a static buffer area in blob.data and passed that blob as struct debugfs_blob_wrapper parameter. blob.size was set to 0 for initialization that is later needed in other functions.

If the file_operations is written in input file, it reads the pid and prints out the so-called "ptree". Linux's system's task_struct is used to get information about the process. Thankfully, the task_struct contains information about the parent as well, so I made a recursive function to recursively print out the process from init (or in my case, systemd whose pid is 1). sprintf function is used in this case, and I explained the use of this function thoroughly in background research section.

Lastly, I used debusfs_remove_recursive(dir) to erase ptree directory and its contents for the exit module. The result is shown in the captured image below.


seri@seri-VirtualBox: ~/Downloads/SERI/ptree
En
11:48
seri@seri-VirtualBox: ~/Downloads/SERI/...
seri@seri-VirtualBox: ~/Downloads/SERI/...

```
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-31-generic'
  CC [M]  /home/seri/Downloads/SERI/ptree/dbfs_ptree.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seri/Downloads/SERI/ptree/dbfs_ptree.mod.o
  LD [M]  /home/seri/Downloads/SERI/ptree/dbfs_ptree.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-31-generic'
sudo insmod dbfs_ptree.ko
seri@seri-VirtualBox:~/Downloads/SERI/ptree$ sudo su
root@seri-VirtualBox:/home/seri/Downloads/SERI/ptree# cd /sys/kernel/debug/ptree
root@seri-VirtualBox:/sys/kernel/debug/ptree# ps
  PID TTY          TIME CMD
20256 pts/4    00:00:00 sudo
20257 pts/4    00:00:00 su
20258 pts/4    00:00:00 bash
20268 pts/4    00:00:00 ps
root@seri-VirtualBox:/sys/kernel/debug/ptree# echo 20258 >> input
root@seri-VirtualBox:/sys/kernel/debug/ptree# cat ptree
systemd (1)
lightdm (1915)
lightdm (2047)
upstart (2098)
gnome-terminal- (5223)
bash (5230)
sudo (20256)
su (20257)
bash (20258)
root@seri-VirtualBox:/sys/kernel/debug/ptree# make clean
make: *** No rule to make target 'clean'.  Stop.
root@seri-VirtualBox:/sys/kernel/debug/ptree# exit
exit
seri@seri-VirtualBox:~/Downloads/SERI/ptree$ make clean
```

3. Find Physical Address

The creation of paddr dir and output file is implemented with the following two lines.

debugfs_create_dir(“paddr”, NULL);

debugfs_create_file(“output”, 0444, dir, NULL, &dbfs_fops);

I worked on Ubuntu 16.04.1 LTS + kernel 4.4.0 for this lab, and this environment happens to use 4-level page table. The main goal of this part of the assignment is to return a physical address that is mapped to a virtual address. I first looked at app.c file to understand this process. In app.c, pckt's address is sent as a parameter in read function. Therefore, it is fair to say that it would be easier if we could use struct packet in dbfs_paddr.c. So struct packet was defined in dbfs_paddr.c as well and pckt address was read as struct packet in dbfs_paddr.c.

Now that vaddr was acquired, all that was needed to be done was to get the paddr. Thankfully, TAs explained the page walk procedure very kindly in lab8 ppt. I also had to study <asm/pgtable.h> and mm_struct in task_struct. (http://egloos.zum.com/bigs/v/6073753 helped a lot!) (https://stackoverflow.com/questions/20868921/traversing-all-the-physical-pages-of-a-process, https://kldp.org/node/110748 too!)Since it is a 4 level page table, each level entry has variable pid_t, pud_t, pmu_t, pte_t. I got each framepage descriptor using the following code lines:

pgd = pgd_offset(task->mm, pckt->vaddr);

pud = pud_offset(pgd, pckt->vaddr);

pmd = pmd_offset(pud, pckt->vaddr);

pte = pte_offset_map(pmd, pckt->vaddr);

pg = pte_page(*pte);

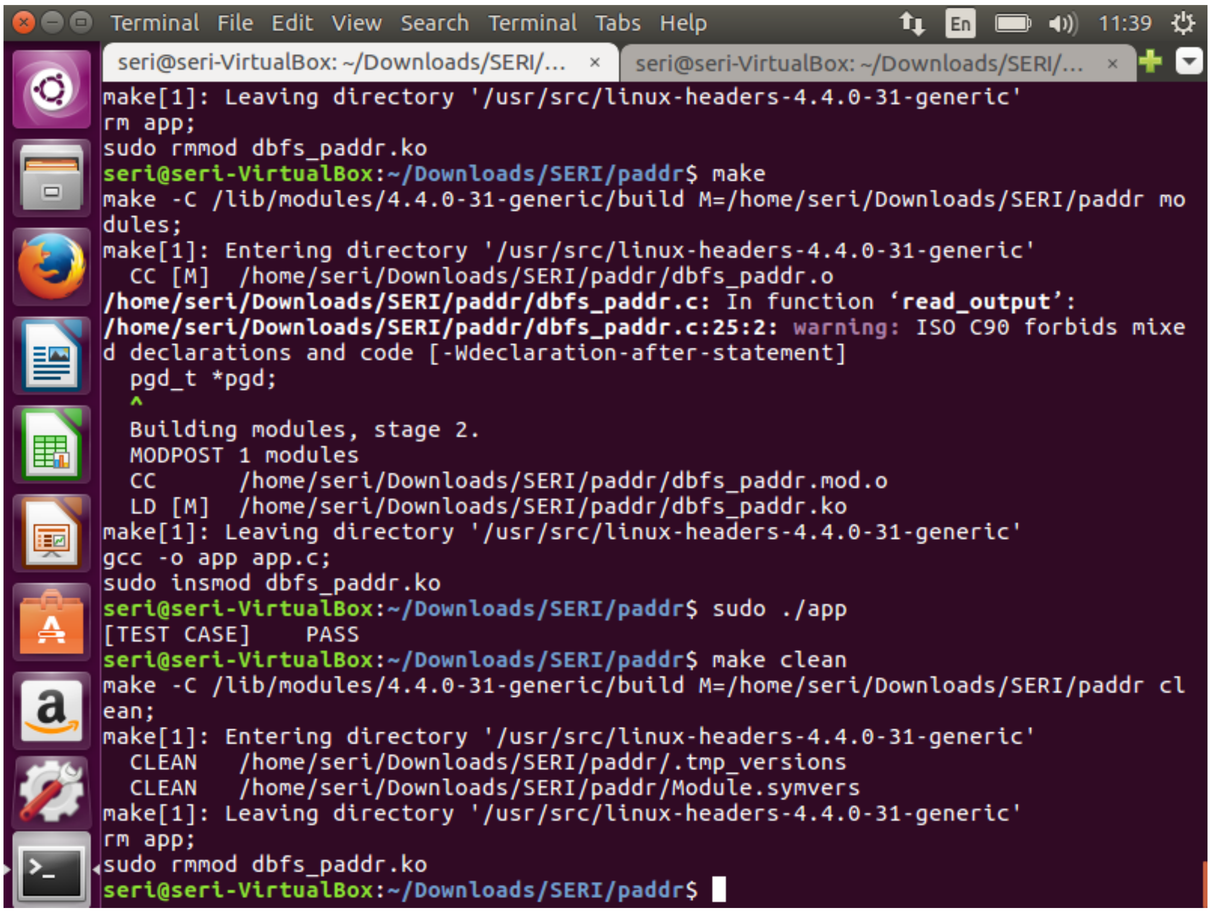
The offset function for example pud_offset function finds pmd_t address. The function pte_page is used to get the page frame descriptor address that pte points to. Thankfully, linux has page_to_phy function that turns struct page* to final physical address. I put the physical address value in pckt->paddr and unmapped pte.

I haven't really tested otherwise, but I was told that it is wise to use pte_offset_map and pte_unmap function for pte offset. This is because PTE page can be in high memory, so temporary mapping is needed to access pte.

pte_offset_map

pte_unmap

Lastly, I used debusfs_remove_recursive(dir) to erase ptree directory and its contents for the exit module. The result is shown in the captured image below.


Terminal File Edit View Search Terminal Tabs Help
11:39
seri@seri-VirtualBox: ~/Downloads/SERI/...
seri@seri-VirtualBox: ~/Downloads/SERI/...

```
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-31-generic'
rm app;
sudo rmmod dbfs_paddr.ko
seri@seri-VirtualBox:~/Downloads/SERI/paddr$ make
make -C /lib/modules/4.4.0-31-generic/build M=/home/seri/Downloads/SERI/paddr mo
dules;
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-31-generic'
  CC [M]  /home/seri/Downloads/SERI/paddr/dbfs_paddr.o
/home/seri/Downloads/SERI/paddr/dbfs_paddr.c: In function 'read_output':
/home/seri/Downloads/SERI/paddr/dbfs_paddr.c:25:2: warning: ISO C90 forbids mixe
d declarations and code [-Wdeclaration-after-statement]
  pgd_t *pgd;
  ^
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seri/Downloads/SERI/paddr/dbfs_paddr.mod.o
  LD [M]  /home/seri/Downloads/SERI/paddr/dbfs_paddr.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-31-generic'
gcc -o app app.c;
sudo insmod dbfs_paddr.ko
seri@seri-VirtualBox:~/Downloads/SERI/paddr$ sudo ./app
[TEST CASE]     PASS
seri@seri-VirtualBox:~/Downloads/SERI/paddr$ make clean
make -C /lib/modules/4.4.0-31-generic/build M=/home/seri/Downloads/SERI/paddr cl
ean;
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-31-generic'
  CLEAN   /home/seri/Downloads/SERI/paddr/.tmp_versions
  CLEAN   /home/seri/Downloads/SERI/paddr/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-31-generic'
rm app;
sudo rmmod dbfs_paddr.ko
seri@seri-VirtualBox:~/Downloads/SERI/paddr$
```

4. Conclusion (what I learned and what was difficult and surprising)

parallels 개발환경에서 정상적으로 동작을 안한다는 사실을 모르고 있어서 parallels 에서 debugging하느라 정말 너무 힘들었다. 이것 저것 고쳐봐도 뭐가 잘못된지 모르겠으니까 또 고치고 돌려보고 고치고 돌려보고 하는 과정의 반복이었다. 처음 3~4일은 parallels 개발환경에서 debugging을 하는 데 소요해서 시간이 굉장히 많이 걸린 랩이었다. 하지만 그 과정에서 이것 저것 찾아보면서 많이 배운 것 같고 kernel이라는 것이 처음에는 너무 생소했는데 이제는 조금 익숙하게 느껴진다. kernel level programming도 user level programming과 debugging하는 방식이 크게 다르지 않다는 것을 깨달았다. (printk를 집어넣어서 어디에 에러가 났는지 계속 확인하는 방식) 막상 코드를 짜보니 그렇게 길지 않지만 코드 한 줄 짤 때마다 많은 양을 찾아보고 공부해야 했어서 코드를 짤 때 암흑 속에서 이리저리 찾아 헤매는 사람이 된 기분이었다. 그래도 하나하나씩 개

념을 이해하기 시작할 때와 코드가 돌아갈 때는 매우 뿌듯했다. 공부한 내용 중 중요하다고 생각되는 부분은 이 보고서 background research와 각 ptree, paddr에 정리해두었다.