CSIE 5374 Assignment 3 (Due on 04/27 11:59)

The page table of a task maps virtual memory addresses to physical memory addresses. In normal cases, page tables are managed by the OS kernel such as Linux, and are inaccessible from userspace. Inspired by Dune, we want to allow userspace tasks to view their own and other userspace task's page tables. As mentioned in the Dune paper, this information has various benefits, such as improving garbage collection. However, this also increases the kernel's attack surface, as one task can manipulate another's page table to carry attacks such as code injection.

In assignment 3, we ask you to explore these fronts by making page tables accessible to userspace tasks. First, you should add a new system call ***expose_pte*** to the Linux kernel v5.4. The system call exposes a given task's page tables to userspace. Next, you are asked to write userspace programs to test the capabilities of the system call. Your modified Linux kernel and the test program must run on the same AArch64 machine emulated by QEMU, as you did in the first two assignments.

This is a group assignment, so you should collaborate with your group members on this assignment.

Development Tips

In the following sections, we assume you already know how to compile the Linux kernel and create a root file system image and run it with `qemu-system-aarch64`. You could refer to the assignment 1 specification for these steps.

Compile Linux

To compile the kernel module, we assume you have downloaded or installed the following prerequisites.

- Prerequisite
  - Linux v5.4 source code
    - `git clone git@github.com:torvalds/linux.git`
    - `git checkout tags/v5.4`
  - Cross compiler
    - gcc-aarch64-linux-gnu (4:9.3.0-1ubuntu2)
    - `sudo apt update && sudo apt install gcc-aarch64-linux-gnu`

NOTE 1: You have to make sure the kernel you use in your VM is compiled from the source you specified above. You should use the provided **defconfig** to compile your Linux kernel.

QEMU shared folder (optional)

Frequently updating kernel module binary to filesystem image can be annoying. So it's nice to have a shared folder.

You can follow the instructions below to have a shared folder in QEMU VM.

First, append these two lines to `run-vm.sh`

```
@@ -9,6 +9,7 @@ FS=cloud.img
 CMDLINE="earlycon=pl011,0x09000000"
 DUMPDTB=""
 DTB=""
+SHARED_DIR=./shared

 usage() {
        U=""
@@ -98,3 +99,4 @@ qemu-system-aarch64 -nographic -machine virt,gic-version=2 -m 1024 -
cpu cortex-a
        -append "console=ttyAMA0 root=/dev/vda rw $CMDLINE" \
        -netdev user,id=net0,hostfwd=tcp::2222-:22 \
         -device virtio-net-pci,netdev=net0,mac=de:ad:be:ef:41:49 \
+       -virtfs
local,path=$SHARED_DIR,mount_tag=shared,security_model=passthrough,readonly \
```

Then create a corresponding directory in host.

```
$ mkdir ./shared
```

Finally, boot your QEMU VM up and mount the shared folder somewhere.

```
$ ./run-vm.sh
...

Ubuntu 18.04.6 LTS ubuntu ttyAMA0

ubuntu login: root

...

root@ubuntu:~# mount -t 9p -o trans=virtio shared /mnt
```

Requirements

Add new system call to Linux: expose_pte (55%)

In this assignment, you are asked add a new system call **expose_pte** to Linux v5.4. The system call exposes a target task's page table needed to translate a given range of userspace virtual memory addresses.

As we discussed in the class, Arch64(Armv8) supports 48-bit addressing mode using four-level page tables. The different levels (from level 0, 1, 2, 3) of the page table are specified with the respective acronyms: PGD, PUD, PMD, PTE.

You are NOT asked to remapping all levels of the page tables as they are to userspace. Instead, you should first create a flattened page table that folds the first three levels of the target task's page tables (e.g. PGD, PUD, PMD), then set the entries in the flattened page table to point to the fourth level page table (PTE). Both the flattened page table and the remapped PTE tables should reside in the system call caller task's

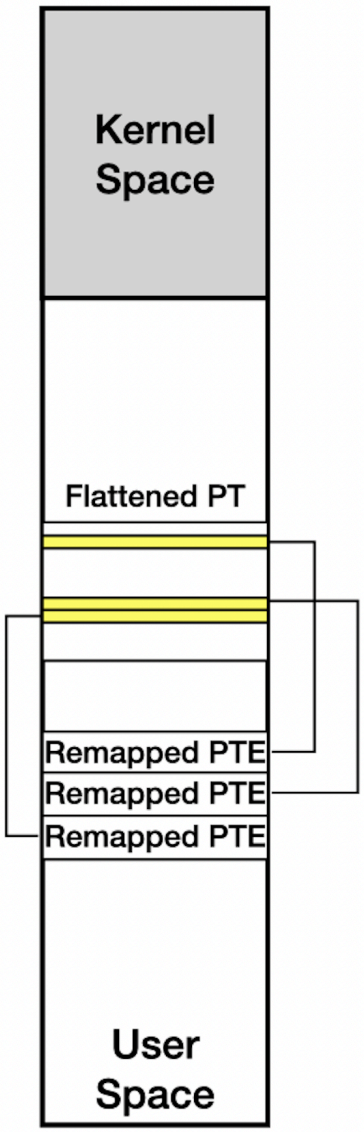
userspace address space.

Entries in the flattened page table are 64-bit long. The system call is responsible of creating the entries in the flattened page table. For a given input virtual memory address (VA), if the mapping to a respective physical memory address (PA) exists, the system call should map the corresponding entry in the flattened page table to the PTE page table that translates VA to PA. Please note that the entry should contain the **remapped virtual address** of the respective PTE page table. If the PTE table that translates PA to VA does not exist, you should store **zero** to the respective entry in the flattened page table.

You can check page 14 here and the reference manual for how 4-level page table translation works in Aarch64.

For the manual, take a look at ***chapter D5.2 The VMSAv8-64 address translation system*** to find out how the how Arm translates a virtual address to a physical address using four-level page tables. Figure D5-3 and D5-7 should include helpful information for you (You can ignore 52-bit support for now and focus on the 48-bit support, and check how Arm translates to a 4KB page). You can find Arm's page table descriptors (page table entries) in ***chapter D5.3 VMSAv8-64 Translation Table format descriptor***. As we discussed in the course, page table descriptors can be used to point to the next level of page table, or points to an entry. For the latter, the descriptor consist of the attribute bits (ex: for permission control) and the pfn bits (physical page frame of the translated physical address).

You can refer to the figure below for how the flattened page table translates virtual memory addresses to physical memory addresses.



The specification of the system call is shown below. The system call takes one parameter, the **struct expose_pte_args**. Please check the embedded comments below for what specific members in the

```
struct expose_pte_args {
  //PID of the target task to expose pte (can be the caller task or others)
        pid_t pid;
  //begin userspace VA of the flattened page table
        unsigned long begin_fpt_vaddr;
  //end userspace VA of the flattened page table
        unsigned long end_fpt_vaddr;
  //begin userspace VA of the remapped PTE table
        unsigned long begin_pte_vaddr;
  //end userspace VA of the remapped PTE table
        unsigned long end_pte_vaddr;
  //begin of userspace VA to expose PTE mappings
        unsigned long begin_vaddr;
  //end of userspace VA to expose PTE mappings
        unsigned long end_vaddr;
};
```

```
SYSCALL_DEFINE1(expose_pte,
                struct expose_pte_args __user *, args)
{
  //handle system call
}
```

The expose_pte system call should perform validation like below. If one of the requirements fail, your system call handler should reject to service and return the error code -EINVAL.

1. If the virtual address range reserved for the flattened page table and PTE tables are reserved in the caller task's address space.
2. If the given task's PID exists.

Detailed Requirements:

- You do not have to deal with the virtual address regions mapped to transparent huge pages (2MB pages).
- You do not need to synchronize page table updates in the kernel to your flattened page tables in userspace. However, kernel updates to the remapped PTE tables should be transparent to the userspace.
- Your system call only has to support input addresses aligned to page size (4K). Your system call could simply reject to service if any of the parameters fail to satisfy the requirement.
- Your system call should ensure that the kernel does not free the target task's resources (ex: page tables, task_struct, etc.) that it accesses.

Hints:

- remap_pfn_range will be of use for you.
- You should consider using the **mmap** system call in your userspace test program to reserve the virtual address space region for the flattened page table and the remapped PTE tables.

Userspace Test Program (30%)

You are asked to write a userspace program to test the **expose_pte** system call. You are asked to implement two features in your test program: Virtual Address Space Inspection, and Code Injection.

Virtual Address Space Inspection (10%)

You should use the new system call to inspect the virtual address space mapping of the target task. Your test program should support the following argument and output like below.

```
shell$>./hw3-test -i pid begin_va end_va
va1 xxx pa1 yyy
va2 xxx pa2 yyy
...
van xxx pan yyy
```

Code Injection (20%)

You should leverage the new system call to carry out code injection attack on a target task. In a stock Linux based system, the kernel enforces strict isolation between task address spaces, ensuring one task cannot arbitrarily modify code or data that belong to other tasks in memory.

We provide a test program template (**hw3.c**) for you in this assignment. This program allocates a memory page and fills in the page with shellcode. Your test program should inject the shellcode to the target process compiled from the provided (**hw3-sheep.c**) and trigger the payload, which launches /bin/sh in the target process.

Bonus (5%)

You will get the bonus points if you could (1) create a shellcode that creates a new file in the /tmp/ directory and writes output to the file, and (2) inject this shellcode to an userspace process other than the sheep process.

Write-up (10%)

Each of the groups are required to provide a write-up about the assignment. The write-up should include the instructions for how to compile and run your test programs, some explanation of your source code (e.g. description for how your code and test program work), and contribution from each of your group members.

Homework submission

You should submit the assignment via NTU Cool.

Submission Format

You are required to submit the kernel patch for Linux v5.4, the source code of your userspace program to test the system call, a Makefile to compile the program, and a write-up file. Compress all the files in hw3.zip, and upload the zip file to NTU Cool.

Your Makefile does not have to deal with copying or deploying the test program to your VM.

```
.
├── source code of your user space test program
├── write-up.md
├── YOUR_5.4_KERNEL.patch
├── any extra file
```

Each of the groups only need to submit **one copy** of the assignment.

Grading criteria

Please run checkpatch against the changes you made to the Linux kernel. You will lose points if checkpatch reports either warnings or errors. Note that you are not asked to run checkpatch against your userspace programs.

In your system call implementation, you are required to properly protect shared memory accesses, manage resources (free allocated memory), handle errors, and return error codes to userspace.

We will also apply your submitted kernel patch to the Linux v5.4 kernel and test the patched kernel in our Ubuntu 20.04 environment. If we fail to apply your patch to the kernel source, or the patched Linux fails to compile, you will get **Zero** points.

Plagiarism policy

You are allowed to reference sources from the internet. If you do, please attach all of your references in the write-up. If we find that your code is similar to other's from the internet, we will count it as plagiarism if we could not find the corresponding references. You will then get zero points for the assignment automatically.

Late Policy

We do not accept late submissions for this assignment. Please start the assignment early.