Lab 7: Kernel Page-Table Isolation

Due at 23:59 Sunday, June 30, 2019

Submit to ftp://rododo:public@public.sjtu.edu.cn/upload/lab7
If you have any questions, feel free to contact Zinan Li (rododo AT sjtu DOT edu DOT cn), who is the TA responsible for lab 7

Introduction

As we learned in class, Meltdown is such a powerful attack that could bypass the permission bit in page table. It allows a user program to read arbitrary pages that marked as present in its page table. Kernel Page-Table Isolation (KPTI) is a technique deployed in kernel that could mitigate Meltdown attack. It makes the kernel not rely on the PTE_U bit in page table. Instead, KPTI removes kernel virtual address space from user page table, using the PTE_P bit as permission control. Then the transient execution in Meltdown will fail due to lack of a valid TLB entry. In this lab, you will implement KPTI on JOS.

Getting Started

Use Git to commit your Lab 6 source (if you haven't already), fetch the latest version of the course repository, and then create a local branch called `lab7` based on our lab7 branch, `origin/lab7`:

```
athena% cd ~/jos-2019-spring/lab
athena% git commit -am 'my solution to lab6'
nothing to commit (working directory clean)
athena% git pull
Already up-to-date.
athena% git checkout -b lab7 origin/lab7
Branch lab6 set up to track remote branch refs/remotes/origin/lab7.
Switched to a new branch "lab7"
athena% git merge lab6
Merge made by recursive.
 fs/fs.c |   42 ++++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
athena%
```

Lab Requirements

This lab does not have challenge problem and annoying question-answer exercise. No document is required. This lab is optional. You have to pass **ALL** tests in this lab to get 5 pts bonus in your final score of this course.

Part A: Isolate kernel virtual address space

The core idea of KPTI is removing the whole kernel virtual address space from user page table. It's very simple to achieve this, although you will soon face lots of problems caused by the "simple" remove.

> **Exercise 1.** Modify the code in `env_setup_vm`. Remove all lines that copy the kernel virtual address space mapping to `env_pgdir`. After correctly removing all critical lines, you should be able to pass the "Isolate env_pgdir and env_kern_pgdir" test.

Part B: Make user program runnable again

After exercise 1, the kernel won't panic due to the newly added check. However, you will see lots of warnings, and the machine will keep rebooting.

If you try to debug the problem, you will find that the machine reboots after you switching page table to `env_pgdir` in `env_run`. Now you should realize what happens to your kernel. In exercise 1, all kernel code/data are removed from user page table. So after the page table switching, CPU could not access the instructions by virtual address. This means that some kernel pages must be mapped to finish the kernel-user switching.

Some mappings are necessary

By tracking the executions after page table switching, you could figure out all pages (including both code and data pages) that will be accessed. All these pages must be mapped in user page table.

Recall the linking process you learned in ICS, all codes are finally grouped into `.text` section, and all data are grouped into `.data` section. No page boundary is enforced, so the whole page will be exposed if one function or variable is necessary in it. To make the information leakage minimal, you should first isolate all functions and variables into dedicated pages. You may want to move the `lcr3` call to make the boundary more reasonable.

We modified the linker script (`kern/kernel.lds`). A dedicated section .user_mapped is added. The section is page aligned so it won't shared any page without contents not in this section. Two macros are added to help you to put functions and global variables in the section. To move a function into the section, add `__user_mapped_text` to function **definition** like this: `__user_mapped_text void foo() { ... }`. To move a global variable into the section, add `__user_mapped_data` to variable **definition** like this: `int foo __user_mapped_data;` or with initialization: `int foo __user_mapped_data = 1;`. For assembly code, use `.section .user_mapped.text` or `.section .user_mapped.data` to move all above lines to user mapped section. Do not forget to switch back to `.text` or `.data` section if necessary.

> **Exercise 2.** Identify and mark all necessary code/data as user mapped.

Now you should modify the `env_setup_vm` again to map the section in user page table. The range of the section is marked as `__USER_MAP_BEGIN__` and `__USER_MAP_END__`, which are declared in `kern/kpti.h`

> **Exercise 3.** Map the `.user_mapped` section in `env_pgdir`.
>
> **Hint:** you may found that the things you marked in execrise 2 are not enough. Use gdb wisely to find out all necessary code/data. Do not forget the data structures that might be accessed by hardware.
>
> Now you should pass the "Put necessary code/data in user mapped area" and "Enter user program" test.

Let me go back to kernel!

In test of exercise 3, the user program we used is a special program called "nosyscall". As the name shows, there is no syscall in the program because any attempt to go back into kernel will lead to a crash.

Now you have to fix the user-kernel switch. Several reasons make the whole machine crash when the user-kernel switch happens. During the switch, CPU will try to load the gate descriptor in IDT and the content of task

segment, which are not mapped currently. The trap handler is also not mapped.

> **Exercise 4.** Mark IDT, task segment and trap handler as user mapped.
>
> If you use gdb and set a breakpoint on your syscall handler, you should see that the CPU gets back to kernel mode, but fails to jump to `trap` in `kern/trap.c`, as it is not marked as user mapped.

Recursively marking all code/data might be used in a syscall is impossible. You will find that soonly the whole kernel is user mapped. Instead of that, we should switch the page table to another one which contains both user and kernel address space mappings. The user space mapping is necessary as during a syscall kernel may directly access user program data through a pointer.

At first, let's build a simple kernel page table for each process, which does not contain the user address space. This should be enough to allow `trap` and `trap_dispatch` to work. But even a simple log printing syscall will fail because the string to be printed is in user address space.

> **Exercise 5.** Allocate and setup an additional page table for kernel use in `env_setup_vm`. The page table should have all kernel address space mapped to allow any code in kernel to work normally.

Before trying to switch the page table, think about which code and data are available in the current page table, and how could we determine the target page table we want. The pointer to the target page table is stored in `struct Env`, which is accessible from `curenv`. However, if you dig into the implementation of `curenv`, you will find that lots of code and data are involved in `cpunum`. We do not want to move the whole implementation of `cpunum` to user mapped area, so now you should implement another way to find out the current CPU core number.

> **Exercise 6.** Finish the code in `switch_and_trap` in `kern/trap.c` and modify `_alltraps` to jump into it instead of `trap`.
>
> **Hint:** one CPU-related variable that you could access in `switch_and_trap` is stack pointer
>
> Now your kernel should be able to handle traps in `trap`. But if you run any user program, there will be a kernel page fault when syscall handler tries to access user program memory.

Keep env_pgdir and env_kern_pgdir consistent

The final step is making `env_pgdir` and `env_kern_pgdir` consistent. Then your kernel will work again.

> **Exercise 7.** Find out all codes that modify `env_pgdir`. Make them modify `env_kern_pgdir`, too.
>
> **Note:** If any of the two modification fails, do not forget to rollback the other one. Otherwise there will be a memory leak in your kernel.
>
> Now you should pass the "Enter kernel again" and "Consistent env_kern_pgdir" test

Exercise 8. Carefully examine codes in `env_free` and free `env_kern_pgdir` correctly. Make sure there is no page leak in your kernel. How should you operate the reference count of pages is highly related to how you map the page in both page table.

Now you should pass the "No memory leak" test. **Note:** This test will keep allocating and forking numerous pages and processes. It takes 22 seconds to finish the test on TA's computer (It's i7 6700K! :). Be patient.

This completes the lab. As usual, don't forget to run `make grade`. Remember that you get the bonus only if you pass **ALL** tests. Before handing in, use `git status` and `git diff` to examine your changes. When you're ready, commit your changes with `git commit -am 'my solutions to lab 7'`, then `make handin` and follow the directions.