Page Table Isolation (KPTI)

Fan Chung

March 7, 2023

Contents

1 Articles

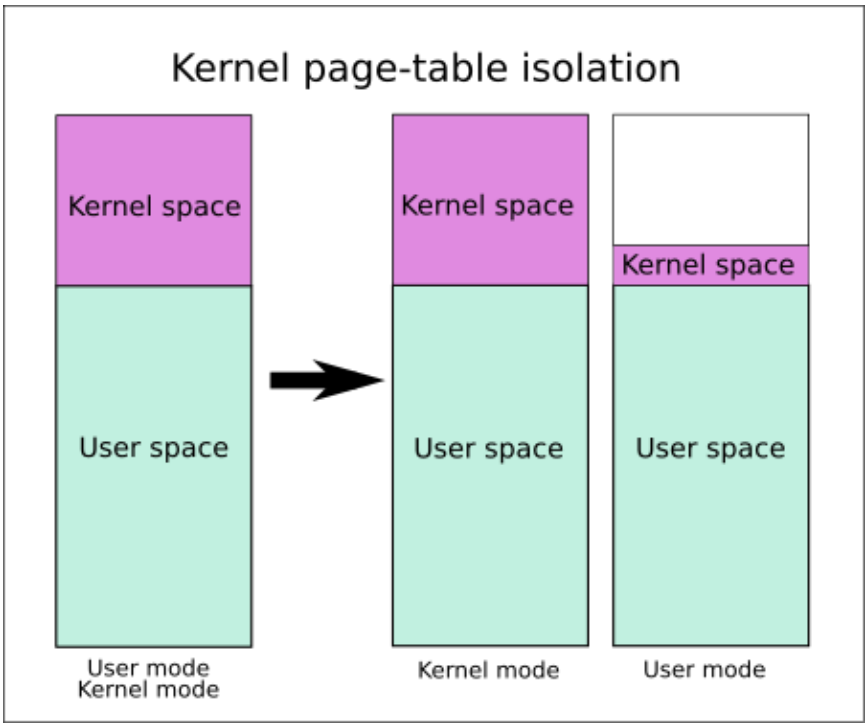
1.1 Background

Before KPTI, for each process, the process page table maps to both the kernel address space and user address space. When doing context switch between the kernel mode and the user mode, the page table don't have to be changed, and the entries in the TLB don't need to be flushed, either.

To defense the meltdown attack, KPTI, based on KAISER, had been proposed. The main concept of KPTI is to hide the kernel address space from the user process. Two page tables are maintained, called kernel-mode page table (corresponds to *normal page table* in KAISER) and user page table (corresponds to *shadow page table* in KAISER).

The kernel-mode page table doesn't change, still contains the kernel space and user space mappings; however, the user-mode page table, contains the user space mappings and only a few kernel space mappings. These kernel space mappings only contain necessary functions, such as entry/exit

functions and IDT (Interrupt Descriptor Table TODO: ??), which we will see them later.



1.1.1 Overhead

KPTI/KAISER Meltdown Initial Performance Regressions

1.2 KPTI implementation in Linux Kernel

1.2.1 Background

On x86-64 architecture, the page table is a radix-tree-like data structure. Each node of the tree represents an page directory entry, the corresponding structures are `pgd_t`, `p4d_t`, `pud_t`, `pmd_t` and `pte_t`. (Assume the 5-level page table (`CONFIG_X86_5LEVEL`) is enabled)

As mentioned previously, KPTI maintains two page tables: kernel page table and user page table. The PGD of the user page table is adjacent to the PGD of the kernel page table, which is offset by 4KB, a size of page. Thus we can toggle the 13th bit on a pgd to switch between the corresponding kernel and user page table.

For example, `kernel_to_user_pgdp` and `user_to_kernel_pgdp` switches between the user and kernel page table by setting/unsetting the 13th bit of the pgdp:

```
/* arch/x86/include/asm/pgtable.h */
#define PTI_PGTABLE_SWITCH_BIT  PAGE_SHIFT

/* arch/x86/include/asm/pgtable_64.h */
static inline pgd_t *kernel_to_user_pgdp(pgd_t *pgdp)
{
  return ptr_set_bit(pgdp, PTI_PGTABLE_SWITCH_BIT);
}

static inline pgd_t *user_to_kernel_pgdp(pgd_t *pgdp)
{
  return ptr_clear_bit(pgdp, PTI_PGTABLE_SWITCH_BIT);
}
```

Both page tables map to the kernel address space and user address space. A page directory have 512 entries. The higher 256 PGD entries map to the kernel address space, while the lower 256 PGD entries map to the user address space.

```
/* arch/x86/include/asm/pgtable_64.h */
static inline bool pgdp_maps_userspace(void *__ptr)
{
  unsigned long ptr = (unsigned long)__ptr;
  return (ptr & ~PAGE_MASK) < (PAGE_SIZE / 2);
}
```

1.2.2 From a high-level perspective

There are mainly three moments that involved KPTI operations:

1. when booting: set up kernel page table and user page table
2. when switching between the kernel/user space: switch between the kernel/user page table
3. when creating a new process: set up the process page table

1. When booting KPTI defines two functions that are called during the early boot stage: `pti_init()` and `pti_finalize()`

   `pti_init()` is called at the end of `mm_init()`, after the kernel page table has been set up. In `pti_init()`, the user page table is mapped some sections from the kernel page table, such as the kernel image (`pti_set_kernel_image_nonglobal`) and the kernel entry (`pti_clone_entry_text()`):

```
/* arch/x86/mm/pti.c */
void __init pti_init(void)
{
  pti_clone_user_shared();

  /* Undo all global bits from the init pagetables in
  ↪  head_64.S: */
  pti_set_kernel_image_nonglobal();
  /* Replace some of the global bits just for shared entry
  ↪  text: */
  pti_clone_entry_text();
  pti_setup_espfix64();
  pti_setup_vsyscall();
}
```

For example, `pti_clone_user_shared()`, shares the ptes where each CPU'S TSS is located:

```
static void __init pti_clone_user_shared(void)
{
  unsigned int cpu;

  pti_clone_p4d(CPU_ENTRY_AREA_BASE);

  for_each_possible_cpu(cpu) {
    /*
     * The SYSCALL64 entry code needs one word of scratch space
     * in which to spill a register.  It lives in the sp2 slot
     * of the CPU's TSS.
     *
     * This is done for all possible CPUs during boot to ensure
     * that it's propagated to all mms.
     */

    unsigned long va = (unsigned long)&per_cpu(cpu_tss_rw, cpu);
    phys_addr_t pa = per_cpu_ptr_to_phys((void *)va);
    pte_t *target_pte;

    target_pte = pti_user_pagetable_walk_pte(va);
    if (WARN_ON(!target_pte))
      return;

    *target_pte = pfn_pte(pa >> PAGE_SHIFT, PAGE_KERNEL);
  }
}
```

`pti_finalize` is called in `reset_init()`. According to the comments, it is required because some of the mappings for the kernel image might

have changed since `pti_init()` cloned them, such as being mapped RO and/or NX.

```
/* arch/x86/mm/pti.c */
void pti_finalize(void)
{
    pti_clone_entry_text();
    pti_clone_kernel_text();

    debug_checkwx_user();
}
```

The relationship can be represented as:

```
start_kernel
  -> mm_init
    -> pti_init
  -> arch_call_reset_init/reset_init
    -> pti_finalize
```

(a) Questions

- ☐ when is the cr3 loaded in x86-64?

(b) Related articles

- ☐ `https://void-shana.moe/posts/kernel-bootup-page-table-initialize-process-64`
- ☐ `https://0xax.gitbooks.io/linux-insides/content/Booting/linux-bootstrap-4.html` - Early page initialization

2. When switching between kernel/user space

   When a user process issues a system call, it switches from the user space to the kernel space. With KPTI enabled, we also need to switch the page table.

   We can take a look at `entry_SYSCALL_64` and swapgs$_{\text{restoreregsandreturntousermode}}$:

   In `entry_SYSCALL_64`, before invoking the system call, it calls `SWITCH_TO_KERNEL_CR3` with the scratch register being `rsp`. The origin value of `rsp` is stored in tss.sp2, which can be accessed in both user and kernel page table (remember `pti_clone_user_shared`?).

```
/* defined in arch/x86/entry/entry_64.S */
SYM_CODE_START(entry_SYSCALL_64)
    UNWIND_HINT_ENTRY

    swapgs

    /* tss.sp2 is scratch space. */
    movq %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
    SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
    movq PER_CPU_VAR(cpu_current_top_of_stack), %rsp

    /* Construct struct pt_regs on stack (skipped) */
    call do_syscall_64

    /* I've simplified the logic here, it jumps to
    ↪  swapgs_restore_regs_and_return_to_usermode whatever */
    jmp swapgs_restore_regs_and_return_to_usermode
SYM_CODE_END(entry_SYSCALL_64)
```

`swapgs_restore_regs_and_return_to_usermode`, basically returns back to the user mode, calls `SWITCH_TO_USER_CR3_STACK` at the end of the procedure, switching back to the user page table.

```
SYM_CODE_START_LOCAL(common_interrupt_return)
SYM_INNER_LABEL(swapgs_restore_regs_and_return_to_usermode,
↪  SYM_L_GLOBAL)
  /* Do restore stuff (skipped) */

  SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

  /* Restore RDI. */
  popq  %rdi
  SWAPGS
  INTERRUPT_RETURN
  /* Other return paths (skipped) */
SYM_CODE_END(common_interrupt_return)
```

So the question is what do `SWITCH_TO_KERNEL_CR3` and `SWITCH_USER_CR3_STACK` exactly do? To figure out what they are, we can take a look at `arch/x86/entry/calling.h`:

We can find `SWITCH_TO_KERNLE_CR3` is a macro definition. With KPTI is enabled (`#ifdef CONFIG_PAGE_TABLE_ISOLATION`), the definition is as follows:

```
.macro SWITCH_TO_KERNEL_CR3 scratch_reg:req
  ALTERNATIVE "jmp .Lend_\@", "", X86_FEATURE_PTI
```

```
  mov  %cr3, \scratch_reg
  ADJUST_KERNEL_CR3 \scratch_reg
  mov  \scratch_reg, %cr3
.Lend_\@:
.endm
```

And `ADJUST_KERNEL_CR3` basically unsets the 13-th bit to switch to the kernel page table:

```
.macro ADJUST_KERNEL_CR3 reg:req
  ALTERNATIVE "", "SET_NOFLUSH_BIT \reg", X86_FEATURE_PCID
  /* Clear PCID and "PAGE_TABLE_ISOLATION bit", point CR3 at
  ↪  kernel pagetables: */
  andq    $(~PTI_USER_PGTABLE_AND_PCID_MASK), \reg
.endm
```

The `SWITCH_TO_USER_CR3_STACK` pushes `rax` to stack and calls `SWITCH_TO_USER_CR3_NOSTACK` with the scratch registers being the origin scratch register and `rax`:

```
.macro SWITCH_TO_USER_CR3_STACK scratch_reg:req
  pushq  %rax
  SWITCH_TO_USER_CR3_NOSTACK scratch_reg=\scratch_reg
  ↪  scratch_reg2=%rax
  popq  %rax
.endm
```

And `SWITCH_TO_USER_CR3_NOSTACK` flips the 13th bit of the PGD:

```
.macro SWITCH_TO_USER_CR3_NOSTACK scratch_reg:req
↪  scratch_reg2:req
  ALTERNATIVE "jmp .Lend_\@", "", X86_FEATURE_PTI
  mov  %cr3, \scratch_reg

  /* Check and do flush (skipped) */
  /* Flip the PGD to the user version */
  orq     $(PTI_USER_PGTABLE_MASK), \scratch_reg
  mov  \scratch_reg, %cr3
.Lend_\@:
.endm
```

3. When creates a new process

1.3 Reference

- Meltdown: Reading Kernel Memory from User Space
- KASLR is Dead: Long Live KASLR
- The current state of kernel page-table isolation - LWN

- PTI(page table isolation)–

2 What ECPT code needs to be modified for KPTI

- pti core functions (`pti.c`)
  - such as `pti_set_user_pgd`, `kernel_to_user_pgdp`
- Switching page table part (`entry_64.S` and `calling.h`)
- ECPT's `pgd_alloc()` (`mm/ECPT.c`)
  - Allocate two page tables?

No interface changes found currently?