Page Table Isolation (PTI)

Overview

Page Table Isolation (pti, previously known as KAISER [1]) is a countermeasure against attacks on the shared user/kernel address space such as the "Meltdown" approach [2].

To mitigate this class of attacks, we create an independent set of page tables for use only when running userspace applications. When the kernel is entered via syscalls, interrupts or exceptions, the page tables are switched to the full "kernel" copy. When the system switches back to user mode, the user copy is used again.

The userspace page tables contain only a minimal amount of kernel data: only what is needed to enter/exit the kernel such as the entry/exit functions themselves and the interrupt descriptor table (IDT). There are a few strictly unnecessary things that get mapped such as the first C function when entering an interrupt (see comments in pti.c).

This approach helps to ensure that side-channel attacks leveraging the paging structures do not function when PTI is enabled. It can be enabled by setting CONFIG_PAGE_TABLE_ISOLATION=y at compile time. Once enabled at compile-time, it can be disabled at boot with the 'nopti' or 'pti=' kernel parameters (see kernel-parameters.txt).

Page Table Management

When PTI is enabled, the kernel manages two sets of page tables. The first set is very similar to the single set which is present in kernels without PTI. This includes a complete mapping of userspace that the kernel can use for things like copy_to_user().

Although _complete_, the user portion of the kernel page tables is crippled by setting the NX bit in the top level. This ensures that any missed kernel->user CR3 switch will immediately crash userspace upon executing its first instruction.

The userspace page tables map only the kernel data needed to enter and exit the kernel. This data is entirely contained in the 'struct cpu_entry_area' structure which is placed in the fixmap which gives each CPU's copy of the area a compile-time-fixed virtual address.

For new userspace mappings, the kernel makes the entries in its page tables like normal. The only difference is when the kernel makes entries in the top (PGD) level. In addition to setting the entry in the main kernel PGD, a copy of the entry is made in the userspace page tables' PGD.

This sharing at the PGD level also inherently shares all the lower layers of the page tables. This leaves a single, shared set of userspace page tables to manage. One PTE to lock, one set of accessed bits, dirty bits, etc...

Overhead

Protection against side-channel attacks is important. But, this protection comes at a cost:

1. Increased Memory Use

   a. Each process now needs an order-1 PGD instead of order-0. (Consumes an additional 4k per process).
   b. The 'cpu_entry_area' structure must be 2MB in size and 2MB aligned so that it can be mapped by setting a single PMD entry. This consumes nearly 2MB of RAM once the kernel is decompressed, but no space in the kernel image itself.

2. Runtime Cost

   a. CR3 manipulation to switch between the page table copies must be done at interrupt, syscall, and exception entry and exit (it can be skipped when the kernel is interrupted, though.) Moves to CR3 are on the order of a hundred cycles, and are required at every entry and exit.
   b. A "trampoline" must be used for SYSCALL entry. This trampoline depends on a smaller set of resources than the non-PTI SYSCALL entry code, so requires mapping fewer things into the userspace page tables. The downside is that stacks must be switched at entry time.
   c. Global pages are disabled for all kernel structures not mapped into both kernel and userspace page tables. This feature of the MMU allows different processes to share TLB entries mapping the kernel. Losing the feature means more TLB misses after a context switch. The actual loss of performance is very small, however, never exceeding 1%.
   d. Process Context IDentifiers (PCID) is a CPU feature that allows us to skip flushing the entire TLB when switching page tables by setting a special bit in CR3 when the page tables are changed. This makes switching the page tables (at context switch, or kernel entry/exit) cheaper. But, on systems with PCID support, the context switch code must flush both the user and kernel entries out of the TLB. The user PCID TLB flush is deferred until the exit to userspace, minimizing the cost. See intel.com/sdm for the gory PCID/INVPCID details.
   e. The userspace page tables must be populated for each new process. Even without PTI, the shared kernel mappings are created by copying top-level (PGD) entries into each new process. But, with PTI, there are now

two kernel mappings: one in the kernel page tables that maps everything and one for the entry/exit structures. At fork(), we need to copy both.

f. In addition to the fork()-time copying, there must also be an update to the userspace PGD any time a set_pgd() is done on a PGD used to map userspace. This ensures that the kernel and userspace copies always map the same userspace memory.

g. On systems without PCID support, each CR3 write flushes the entire TLB. That means that each syscall, interrupt or exception flushes the TLB.

h. INVPCID is a TLB-flushing instruction which allows flushing of TLB entries for non-current PCIDs. Some systems support PCIDs, but do not support INVPCID. On these systems, addresses can only be flushed from the TLB for the current PCID. When flushing a kernel address, we need to flush all PCIDs, so a single kernel address flush will require a TLB-flushing CR3 write upon the next use of every PCID.

Possible Future Work

1. We can be more careful about not actually writing to CR3 unless its value is actually changed.
2. Allow PTI to be enabled/disabled at runtime in addition to the boot-time switching.

Testing

To test stability of PTI, the following test procedure is recommended, ideally doing all of these in parallel:

1. Set CONFIG_DEBUG_ENTRY=y
2. Run several copies of all of the tools/testing/selftests/x86/ tests (excluding MPX and protection_keys) in a loop on multiple CPUs for several minutes. These tests frequently uncover corner cases in the kernel entry code. In general, old kernels might cause these tests themselves to crash, but they should never crash the kernel.
3. Run the 'perf' tool in a mode (top or record) that generates many frequent performance monitoring non-maskable interrupts (see "NMI" in /proc/interrupts). This exercises the NMI entry/exit code which is known to trigger bugs in code paths that did not expect to be interrupted, including nested NMIs. Using "-c" boosts the rate of NMIs, and using two -c with separate counters encourages nested NMIs and less deterministic behavior.

   ```
   while true; do perf record -c 10000 -e instructions,cycles -a sleep 10; done
   ```
4. Launch a KVM virtual machine.
5. Run 32-bit binaries on systems supporting the SYSCALL instruction. This has been a lightly-tested code path and needs extra scrutiny.

Debugging

Bugs in PTI cause a few different signatures of crashes that are worth noting here.

- Failures of the selftests/x86 code. Usually a bug in one of the more obscure corners of entry_64.S
- Crashes in early boot, especially around CPU bringup. Bugs in the trampoline code or mappings cause these.
- Crashes at the first interrupt. Caused by bugs in entry_64.S, like screwing up a page table switch. Also caused by incorrectly mapping the IRQ handler entry code.
- Crashes at the first NMI. The NMI code is separate from main interrupt handlers and can have bugs that do not affect normal interrupts. Also caused by incorrectly mapping NMI code. NMIs that interrupt the entry code must be very careful and can be the cause of crashes that show up when running perf.
- Kernel crashes at the first exit to userspace. entry_64.S bugs, or failing to map some of the exit code.
- Crashes at first interrupt that interrupts userspace. The paths in entry_64.S that return to userspace are sometimes separate from the ones that return to the kernel.
- Double faults: overflowing the kernel stack because of page faults upon page faults. Caused by touching non-pti-mapped data in the entry code, or forgetting to switch to kernel CR3 before calling into C functions which are not pti-mapped.
- Userspace segfaults early in boot, sometimes manifesting as mount(8) failing to mount the rootfs. These have tended to be TLB invalidation issues. Usually invalidating the wrong PCID, or otherwise missing an invalidation.

[1] https://gruss.cc/files/kaiser.pdf

[2] https://meltdownattack.com/meltdown.pdf