Security Part 5
An Important Assumption

• Pages cannot be read if read permission is off
  – on Intel x86-64
    - rings 0-2 can read all mapped pages
    - ring 3 can read only those pages for which read permission is explicitly given (in the page-table entries)
What If the Assumption is Wrong?

- Code in ring 3 can read everything that’s currently mapped
- In Linux, all physical memory is mapped into the kernel’s address space
  - kernel address space mapped into every process
  - every process can read all physical memory
Making the Assumption True

• Processor checks page protection on each access of memory
  – a fault occurs and access is not allowed if page is marked not readable
Speculative Execution

1:  movl  $0x1,%ecx
2:  xorq  %rdx,%rdx
3:  cmpq  %rsi,%rdx
4:  jnl   someplace_else
5:  movl  %esi,%edi
6:  imull (%rax,%rdx,4),%ecx

perhaps execute these instructions
Speculative Execution

• If speculatively executed instruction turns out to be used (branch not taken), its results are retired
  – e.g., if a load into a register, the register is updated
  – if speculatively executed instruction results in exception, exception is taken
• If not used (branch taken), register is not updated
  – exceptions are ignored
Micro Operations

• Machine instructions actually composed of micro operations, which can be executed concurrently by the various functional units

• To fetch from memory into register, (at least) two micro operations are used:
  – load value from memory
    - value goes into cache
  – check if access is allowed
    - if not, register not updated
    - exception occurs
  • unless executed speculatively and instruction not used
Speculative Execution

1:  movq   kernel_address,%rcx
2:  movq   random_value,%rdx
3:  cmpq   %rcx,%rdx
4:  jne   someplace_else  jump is taken
5:  movb  (%rcx),%al      No exception, %rax not modified, but cache is updated

Not a security problem, because there’s no way to determine what went into the cache.

Correct?!?!
Speculative Execution

1: movq kernel_address,%rcx
2: movq random_value,%rdx
3: cmpq %rcx,%rdx
4: jne someplace_else
5: movb (%rcx),%al
6: shl %rax,$0xc  # multiply by 4096
7: movq %rax,0(%rbx,%rax)
   # %rbx contains the address of a
   # 1MB (256 x 4096 byte) array in
   # user-accessible memory
The Array

(%rcx)

0:
1:
2:
3:
...

x:
...

253:
254:
255:
Accessing the Array
Speculative Execution

1: movq kernel_address,%rcx
2: movq random_value,%rdx
3: cmpq %rcx,%rdx
4: jne someplace_else
5: movb (%rcx),%al
6: shl %rax,$0xc  # multiply by 4096
7: movq %rax,0(%rbx,%rax)

# %rbx contains the address of a
# 1MB (256 x 4096 byte) array in
# user-accessible memory

Can we be sure processor will guess jump will not be taken?
Alternative Approach

1: movq kernel_address,%rcx
2: movq random_value,%rdx
3: cmpq %rcx,%rdx
4: movb $1,0 # cause a segfault
5: movb (%rcx),%al
6: shl %rax,$0xc # multiply by 4096
7: movq %rax,0(%rbx,%rax)
   # %rbx contains the address of a
   # 1MB (256 x 4096 byte) array in
   # user-accessible memory
Meltdown

• Can read all of kernel memory on a Linux machine at the rate of 503 KB/sec
  – it’s not dependent on bugs: works on a bug-free kernel
  – achieved by using Intel TSX so that the speculatively executed instructions are not retired
  – slows down to 122 KB/sec if segfaults are used
Countermeasures

• KASLR
  – kernel address-space layout randomization
  – kernel starts at a randomly chosen address
    - 40 bits of randomization
    - try all possibilities …

• KAISER
  – “kernel address-space isolation to have side channels efficiently removed”
  – kernel address space is not mapped when in user mode
  – prevents meltdown!
KAISER requires that there are no mappings of kernel virtual addresses when the processor is running in ring 3 (user mode).

a) This is completely innocuous and has no adverse effects

b) This may “break” some otherwise correct user code

c) This may have noticeable performance effects

d) Both b and c are true