C6 Race Conditions

1. Removable devices

1.1 Automatic Code Execution

AutoRun

- When a drive is mounted, AutoRun is a function that is executed automatically on Windows
  - When a USB drive is inserted, programs specified in `autorun.inf` file specified by the user will be run automatically
- Defence
  - Do not mount USB drives received from someone else; write protect your own USB drives before giving someone else
  - Disable AutoRun
    - Set registry value to 0 (disables Media Change Notification messages)
    - Set NoDriveTypeAutorun registry to 0xFF (disables Autorun on all types of drives; Windows may still execute arbitrary code when user clicks on icon for device)

Shortcuts

- Provide direct links to an executable file (LNK extension)
- Icons displayed on desktop, executable invoked when user clicks on shortcut icon or application parses a shortcut icon

2. Race Conditions

- Multiple computations access shared data in a way that their results depend on the order of access
- Attacker may try to change a value after it has been checked but before it is being used (TOCTTOU time of check to time of use)

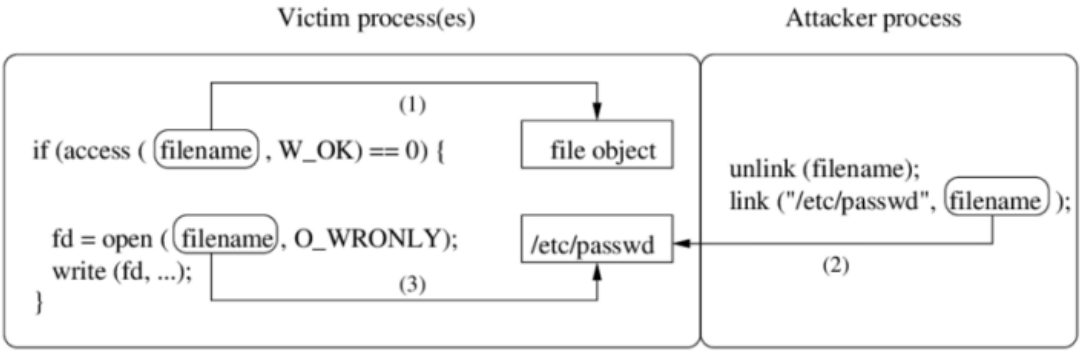
2.1 access / open Races

xterm = X11 Window System terminal emulator, `setuid root` program

- Users can open a log file to record what is being typed
- Log file opened by xterm by
  1. Changes in a subprocess to the user's real uid/gid to test with `access(logfilename, W_OK)` whether the writable file exists and the user has access. If user does not have access, xterm creates the file with user as owner

2. Opens the file for writing by calling as `root open(logfilename, O_WRONLY | O_APPEND)`

- Attacker provides the name of a symbolic link that toggles between a file owned by the attacker and the target file (e.g. /etc/passwd) as `logfilename`
- If access() is called while symbolic link points to the attacker's file and open() is called while it points to the target file, the attacker gets write access to the target file.



Defences

- **Using atomic Code**
  - CZ2005 Operating Systems recap
  - Implemented using semaphores (`synchronized` keyword in Java)
  - Results in a loss of performance
- **TOCTTOU** Defences (File access)
  - Do not take file names as inputs (use file handles / file descriptors instead)
    - `stat()` should not be applied on a file before opening, but should apply `fstat()` directly on file descriptor after opening the file
  - Leave access checking to file system (don't use `access()`)

3. Virtual Memory

- OS assigns each process a virtual address space
- Faster context switch for better performance on linux
- Translation tables map virtual addresses to physical addresses together with permitted access modes
- A CPU register holds address of current table; context switch (change process) updates register value to address of new table

Memory isolation

- Kernel memory can only be accessed if the supervisor bit in the CPU is set (kernel mode)
- Processes running in user space cannot access virtual addresses the kernel is mapped to

Virtual memory access

- Main memory is referenced using a virtual address to load data from main memory into a register
- CPU translates virtual address into physical address using translation table and checks permission bits of virtual address (raise exception if access is not permitted)
    - e.g. accessing illegal memory causes segmentation fault

4. CPU

Traditional CPU

- Serves as interface between hardware and software, microarchitecture is an implementation of ISA

Modern CPU

- Micro-operations are used to implement complex machine instructions as well as to execute actual executions
    - CPU does not directly execute machine instructions
    - Machine language decoded into micro-operations which are then executed

Out of Order Execution (CZ3001 content)

- x86 instructions are fetched by the front-end from memory and decoded into μOPs
    - Reorder Buffer is responsible for register allocation, renaming and retiring of μOPs
    - μOPs are forwarded to Unified Reservation Station, queues the operations on exit ports connected to Execution Units (Functional Units)
- CPUs have branch prediction units that guess which instruction will be executed next before the result of evaluating the branch condition is known
- Instructions on predicted path are independent and can be executed in advance

In-Order Retirement

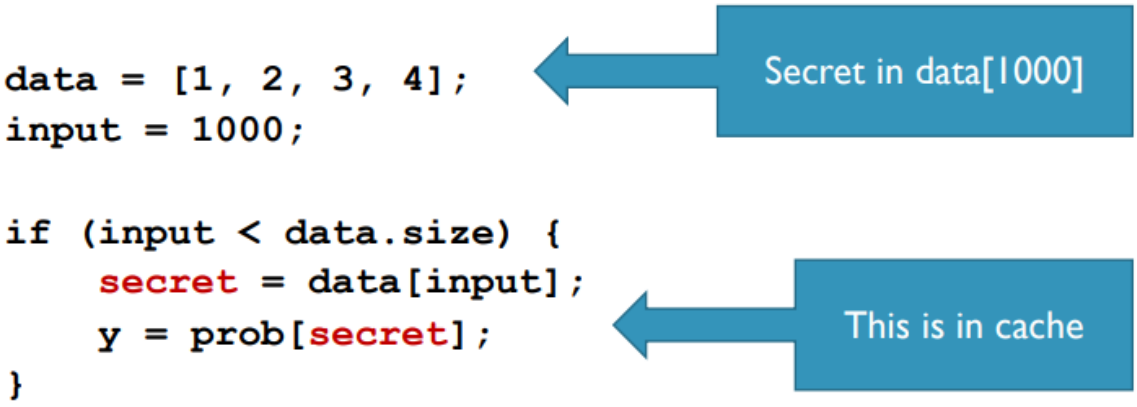
- If prediction was correct, results can be immediately used, else perform a rollback by clearing the reorder buffer and re-initialising the unified reservation station
    - Reverts to saved CPU state
    - Instructions rolled back will be referred to as transient instructions
- μOPs retire in order after execution and results are committed

4.1 Speculative Execution

- Instructions are executed ahead of knowing if they are required to prevent a delay by idling, if it turns out the executed instructions were not needed, changes made are reverted in a way that leaves no trace to the process
- However, **state of microarchitecture has already changed**

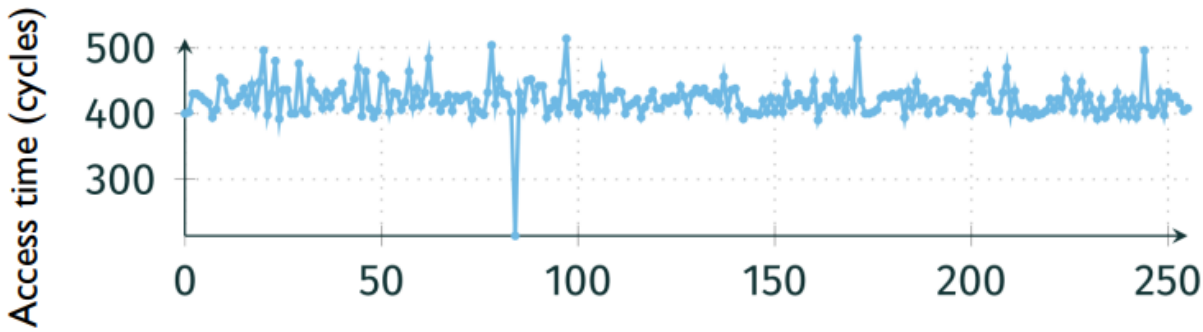
Secret Stealing

- Meltdown and Spectre attacks leverage speculative execution to obtain access to unauthorised information



- Recall cache-based side channel (tutorial 1)
- Can be used to observe other processes running on the same CPU
  - side channel for leaking secret cryptographic keys
- Can be used to observe from user space what had been done while system had been operating in kernel space
  - side channel for leaking secret data (Meltdown)

- Flush+Reload over all entries of the prob array
  - `secret` is revealed
- Out-of-order instructions leave microarchitectural traces
  - We can see them in the cache



4. Meltdown

Dumping memory: https://youtu.be/bReA1dvGJ6Y
Reconstructing images: https://youtu.be/kwnh7q356Jk
Reconstructing photos: https://meltdownattack.com/

4.1 Penetration

1. (load) Transient instruction loads the content of an attacker-chosen illegal memory location into a register
2. (persist) Another transient instruction accesses a cache line based on the content of that register
3. (read) Flush + Reload determines which cache line was accessed; leaks the value stored at the chosen inaccessible memory location

- Repeat to dump kernel memory

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx] ; read a byte of the kernel address
5 shl rax, 0xc ; multiply by 4096 (0xc)
6 jz retry ; retry if the above is zero
7 mov rbx, qword [rbx + rax] ; read offset in the probe array
```

- Target = [rcx] (kernel address); byte value (least significant) at address is loaded into AL
- Line 4, MOV instruction fetched by core, decoded into μOPs, allocated and sent to the reorder buffer
    - Architectural registers (RAX, RCX) mapped to underlying physical registers to prepare for Out-of-Order execution.
- **Step 1 Reading the Secret**
    - Lines 5-7 converted into μOPs and sent to reservation station and wait to be executed once FUs and operand values are available. Kernel address loaded at line 4, decoded into μOP and enters common data bus with μOPs waiting in reservation station for kernel address content to arrive.
    - As soon as this happens, speculative execution of those μOPs can begin.
    - When μOPs finish execution, they retire in order and results are committed to the architectural state; illegal MOV instruction in line 4 raises an exception, and all results of subsequent instructions that were executed out of order are eliminated
- With step 1 alone, attacker manages to load target value into a register which will be reset when the MOV instruction is retired. Attacker needs further transient instructions that encode the target value in the cache state
    - Race condition comes to mind; if attacker manages to execute transient instructions before the exception is raised due to the retirement of MOV, target value can be recorded in cache state.
- **Probe Array**
    - Meltdown allocated a probe array and ensures no part of that array is cached

    - Transient instruction makes an indirect memory access to probe array at an address calculated from the target value in order to encode the target.
    - Target value is multiplied by page size so that accesses to the array have a large spatial separation which also prevents hardware prefetcher from loading adjacent memory locations into the cache (since 1 byte is read, for 4KB pages the probe array is $2^8 \times 4096$ bytes)
- **Step 2 Transmitting the Secret**
    - Line 5 multiplies target value by page size (shift left 12 positions / 4096)
    - Out-of-Order execution has a noise bias towards register value '0', line 6 retries reading target if value is 0 after shifting
    - In line 7, target x 4KB is added to base address of the probe array in RBX and now becomes target address of the side channel
    - Address is read and caches corresponding cache line; transient instruction sequence affects cache state in a way that depends on the target value read in line 4
    - The faster this step is executed, the higher the chance of winning the race against permission check
- **Step 3 Receiving the Secret**
    - When transient instruction of step 2 is executed, one cache line of probe array is cached, position depends on value read in step 1
    - Attacker iterates Flush+Reload over all 256 pages of the probe array and measures the access time for every first cache line on the page; only one cache hit will be observed
    - Target value = page number containing cached memory
- Dumping entire physical memory
    - Attacker can dump entire memory by iterating over all addresses
    - Entire physical memory gets mapped into the kernel address space of the user process, and the entire physical memory of the machine can be read

5. Patching Meltdown

Kaiser

- Does not map kernel memory in user space, addresses therefore cannot be resolved and Meltdown cannot leak any kernel or physical memory other than those few memory locations that have to be mapped in user space (e.g. interrupt handlers)

Stronger Kernel Isolation

- Shadow address space separate kernel space and user space, two copies of page table are kept.
- Shadow page table only keeps the bare minimum required for context switch
- Memory not mapped cannot be accessed

Comment

- Architectural model of the CPU guarantees memory isolation; only this abstract model is specified
  - Programmers writing software for the CPU need this model
- Micro-architectural behaviour is not specified
  - Not needed by programmers
  - Intellectual property of hardware manufacturer relevant for performance that is not revealed to competitors
- Problem is rooted in hardware
  - Race condition between the memory fetch and corresponding permission check
  - Fix: serialize both of them (long-term solution, hard to achieve)

Comment

- Dangers of abstraction: abstract model may not capture observable micro-architectural behaviour
  - Circumventing ASLR is easy if you can read first what you have to guess later
- "A knife sharp enough to cut meat [speculative execution] is sharp enough to cut your finger"