NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 6: Race Conditions

presented by

Li Yi
Assistant Professor
SCSE

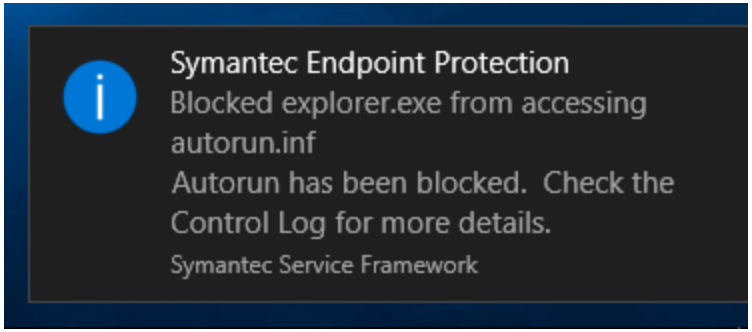
N4-02b-64
yi_li@ntu.edu.sg

Introduction

- This lecture will first look at removable devices as another tainted source of input
- Don't trust your inputs!
- We will then show how race conditions can be exploited to launch attacks, and which defences exist
- Meltdown – final case study in this part of the course where the focus is on the operating system and below

Agenda

- Removable devices
- Race conditions
- Meltdown
- Conclusions



Removable Devices

Automatic Code Execution

- Some functions are executed automatically, e.g., destructors in object-oriented languages
- Another example is the AutoRun feature in Windows that is invoked when a drive is mounted
- If a USB drive is inserted, Windows automatically runs the programs specified in the **`autorun.inf`** file
  - **`autorun.inf`** file specified by owner of USB drive
  - Drive could also host DNS servers or DHCP servers and pretend to be a network …
  - Don't trust your inputs

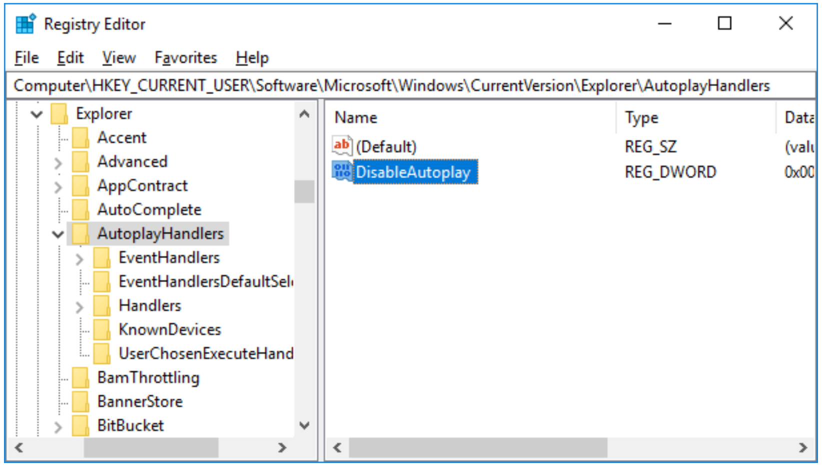
AutoRun (Windows)

- AutoRun exploited to install the Sony rootkit copy-protection software on users' PCs (2005)
  - Installed from CD when user clicked on licence agreement
  - It hides itself so that even many technical computer experts can't find it
- Defence: do not mount USB drives received from someone else; write protect your own USB drives when you give them to someone else
- Defence: disable AutoRun
  - Easier said than done …

"In encoding the discs [with copy-protection spyware] Sony and F4i have decided that their intellectual property is more deserving of protection than the intellectual property and personal information on millions of personal computers around the world"

— Excerpt of class-action lawsuit filed against SONY BMG and First 4 Internet, US District court, New York, 11/14/2005

Disabling AutoRun



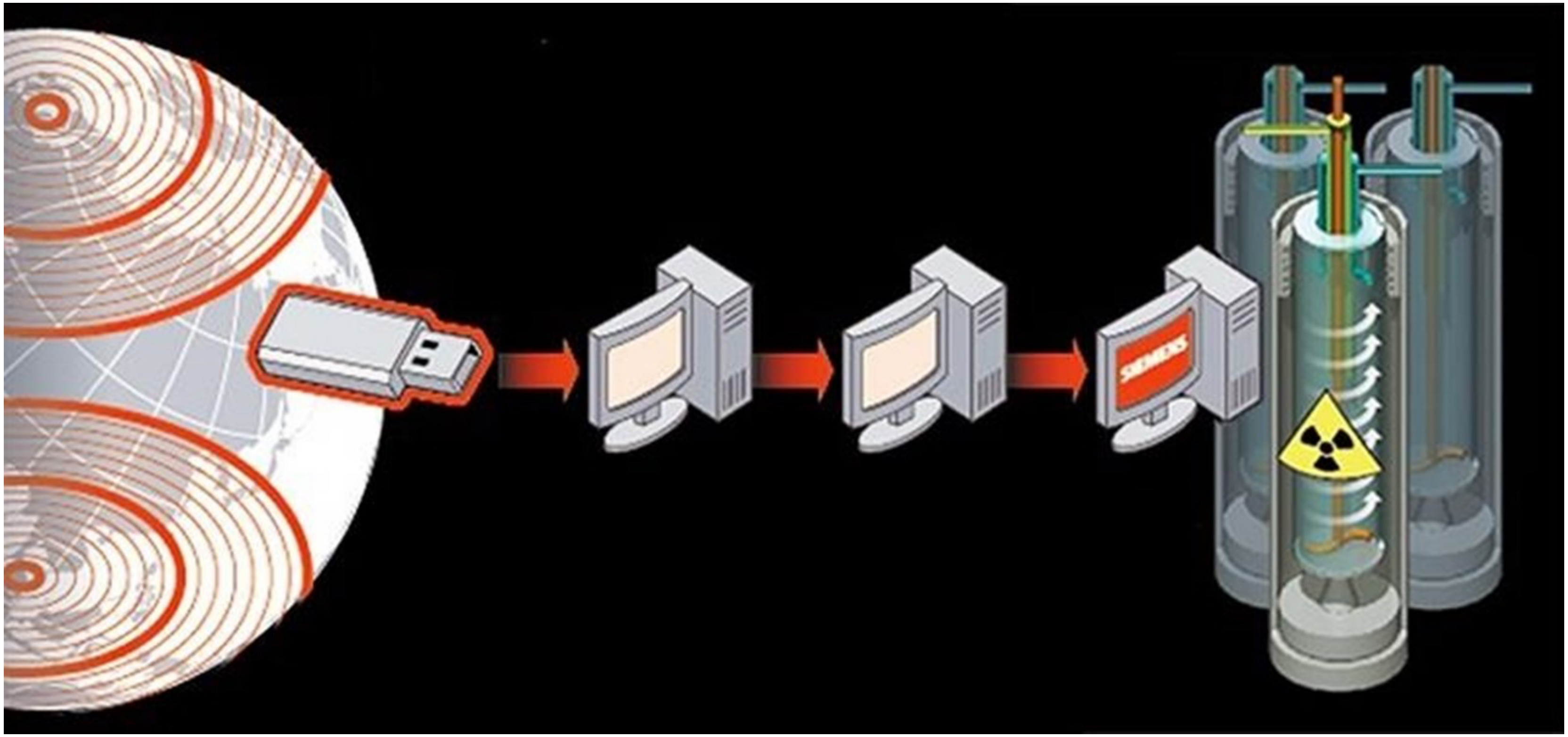
- Set Autorun registry value to 0
  - Only disables Media Change Notification (MCN) messages
- Set NoDriveTypeAutorun registry value to 0xFF:
  - Disables Autoplay on all types of drives
  - But even with this registry value set, Windows may execute arbitrary code when the user clicks the icon for the device
- Information on fixing the problem:
  - https://www.techworld.com/news/security/microsoft-fixes-autorun-windows-vulnerability-111309/ (25.2.2009)
  - http://www.us-cert.gov/cas/techalerts/TA09-020A.html

Lesson

Switching off a feature may be a non-trivial action

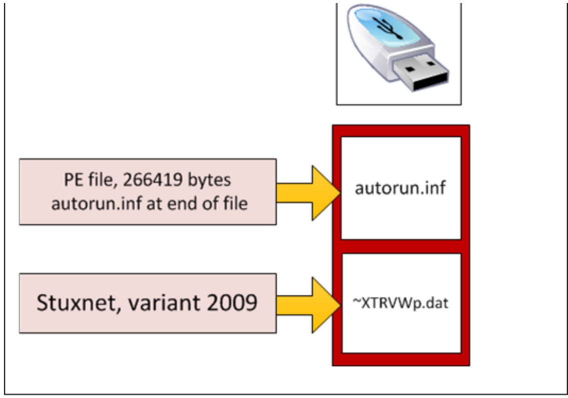
Video: https://youtu.be/7g0pi4J8auQ

Stuxnet: The Virus that Almost Started WW3



http://www.symantec.com/connect/blogs/stuxnet-lnk-file-vulnerability

AutoRun – Stuxnet

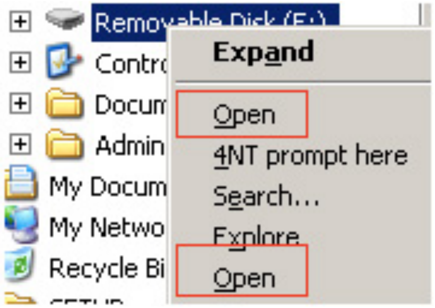


- Early versions of Stuxnet malware spread via AutoRun
- It ceates an **`autorun.inf`** file in the root of removable drives
- Weirdness: **`autorun.inf`** file also contained the shellcode
  - Place shellcode (MZ executable) first within **`autorun.inf`** followed by valid AutoRun commands
  - When parsing **`autorun.inf`** Windows skips characters that are not part of valid AutoRun commands
  - AutoRun commands in **`autorun.inf`** nominate **`autorun.inf`** itself as the file with the executable
  - Shellcode at the start of **`autorun.inf`** would now get executed

https://www.helpnetsecurity.com/2010/09/24/examining-the-stuxnet-worm/

AutoRun – Stuxnet

- Code launched by the local user may run with more privileges than code executed automatically
- How to trick the user?
- AutoRun commands turn off AutoPlay and add a new 'Open' command to the context menu
- A user viewing the context menu for the drive will see two 'Open' commands; the legitimate 'Open' and the 'Open' added by Stuxnet



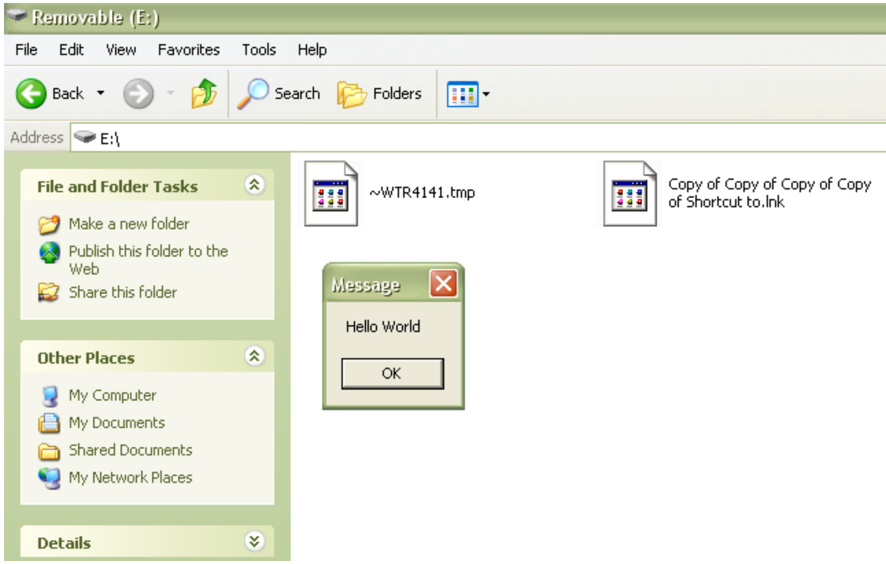
AutoRun – Stuxnet

- Opening the drive via the malign ‘Open’ command triggers execution of the shellcode in `autorun.inf`
- It will also open an Explorer window and display the contents of the drive – as the legitimate command would do
- Success depends on the AutoPlay and AutoRun settings on the computer
- Disabling these features can help to mitigate against this threat

https://www.rubydevices.com.au/blog/stuxnetLnkExploit

Shortcuts (Windows)

- Shortcut files provide direct links to an executable file
  - LNK is the Windows file extension for shortcut files
- Shortcut icons will be displayed on the desktop
- When a user clicks on a shortcut icon or an application parses a shortcut icon, the executable will be invoked



Shortcuts – Stuxnet

- Put malign shortcut and associated shellcode on removable drive
  - Shellcode could also be put on a network share or website
  - Shortcut could also be embedded in some other file
- If a user opens the drive or an application parses the icon of the shortcut file, the shellcode will be invoked
  - Don't trust your inputs
  - Shortcut Icon Loading Vulnerability, CVE-2010-2568
  - Microsoft Security Bulletin MS10-046
  - http://www.f-secure.com/weblog/archives/00001994.html

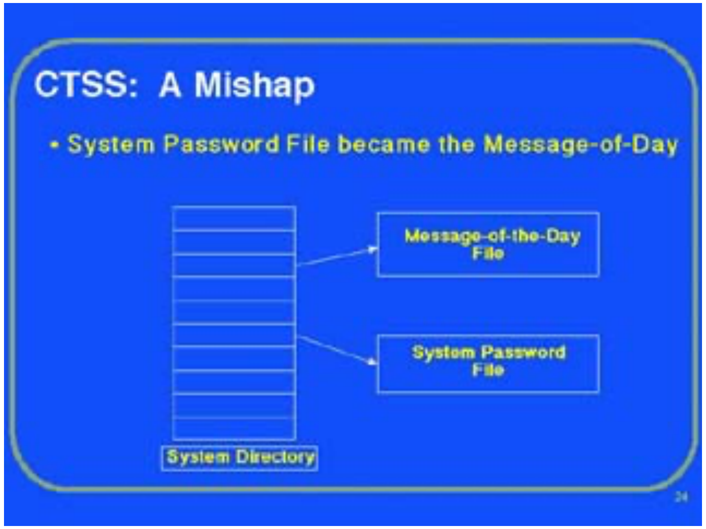
Race Conditions

Race Conditions

- Race condition: multiple computations access shared data in a way that their results depend on the order of accesses
  - Multiple processes accessing the same variable
  - Multiple threads in multi-threaded processes
- Security relevant: attacker may try to change a value after it has been checked but before it is being used
- TOCTTOU (time-to-check-to-time-of use) is a well-known security issue

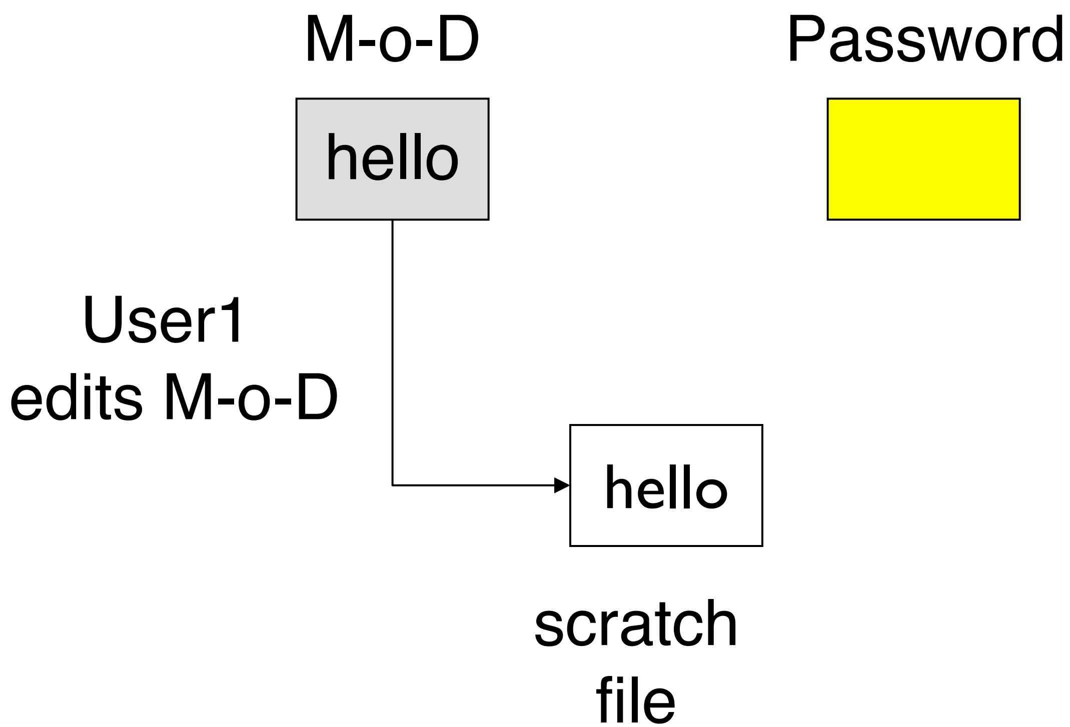
Example – CTSS (1960s)

- One morning, the password file was shown as the message of the day
- Every user had a unique home directory
- When a user invoked the editor, a scratch file with fixed name **`SCRATCH`** was created in this directory
- Innovation: several users could work concurrently as system manager

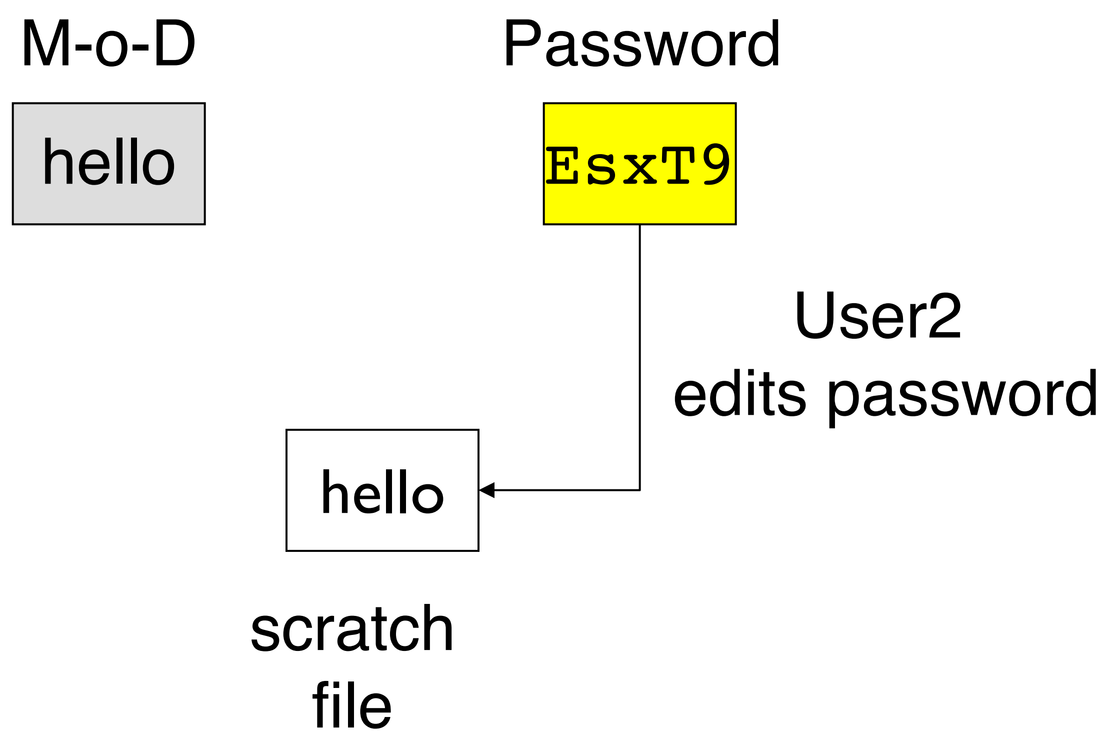


Source: http://larch-www.lcs.mit.edu:8001/~corbato/turing91/

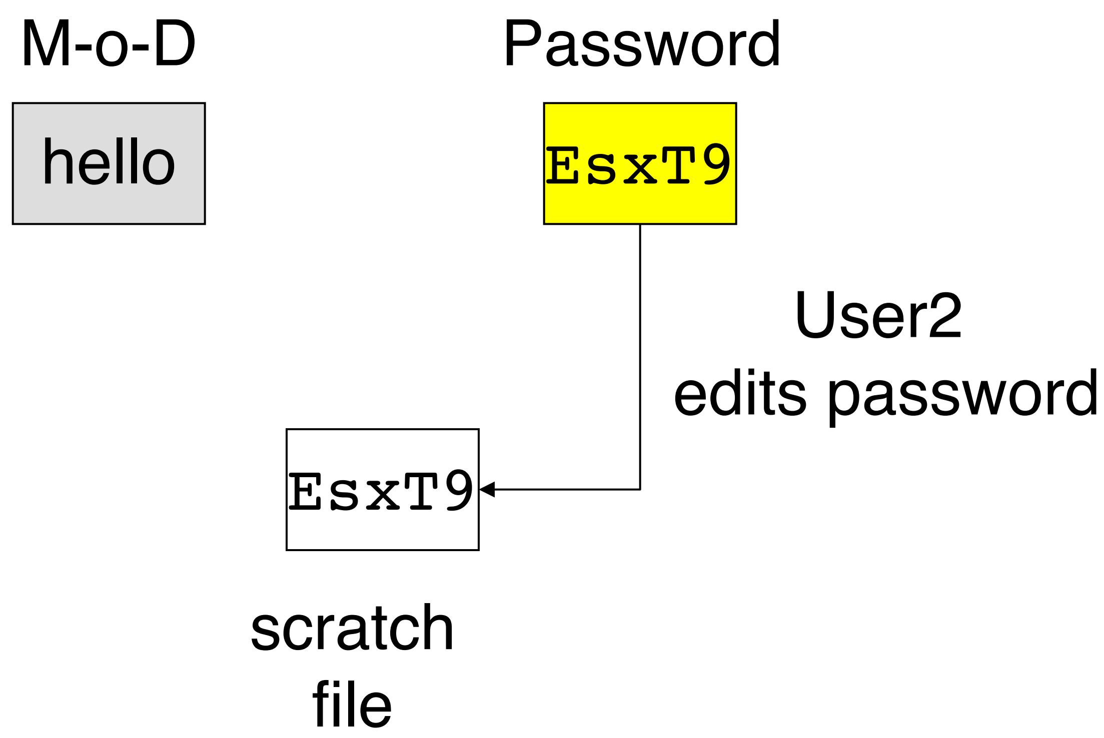
Race Condition



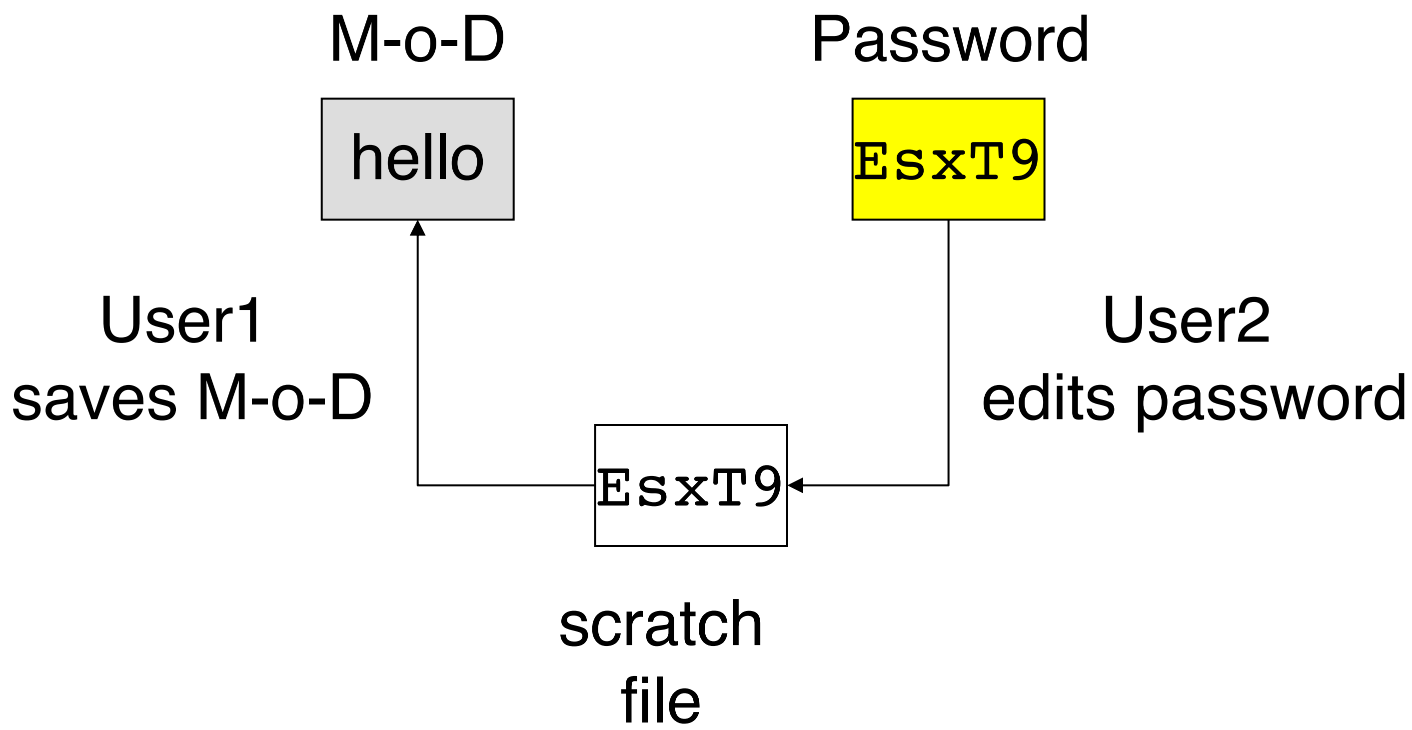
Race Condition



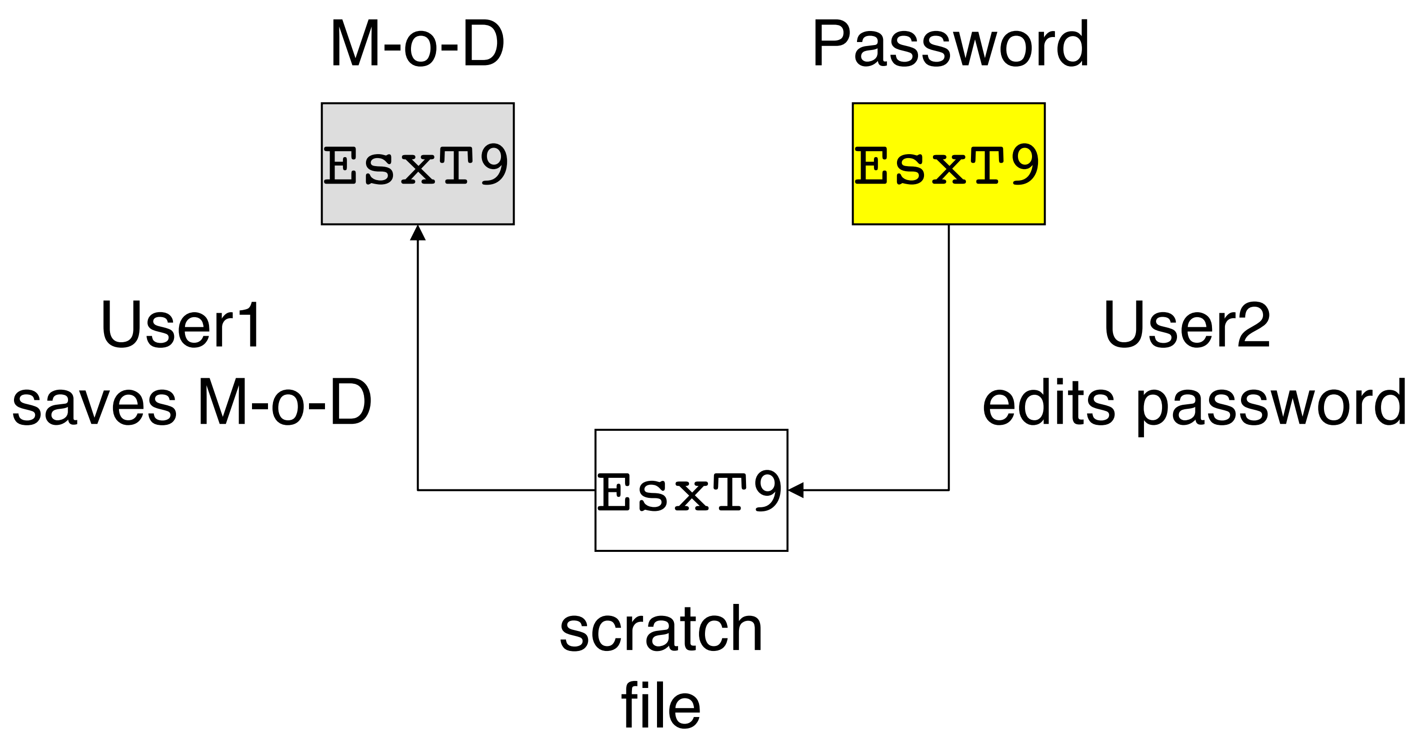
Race Condition



Race Condition



Race Condition



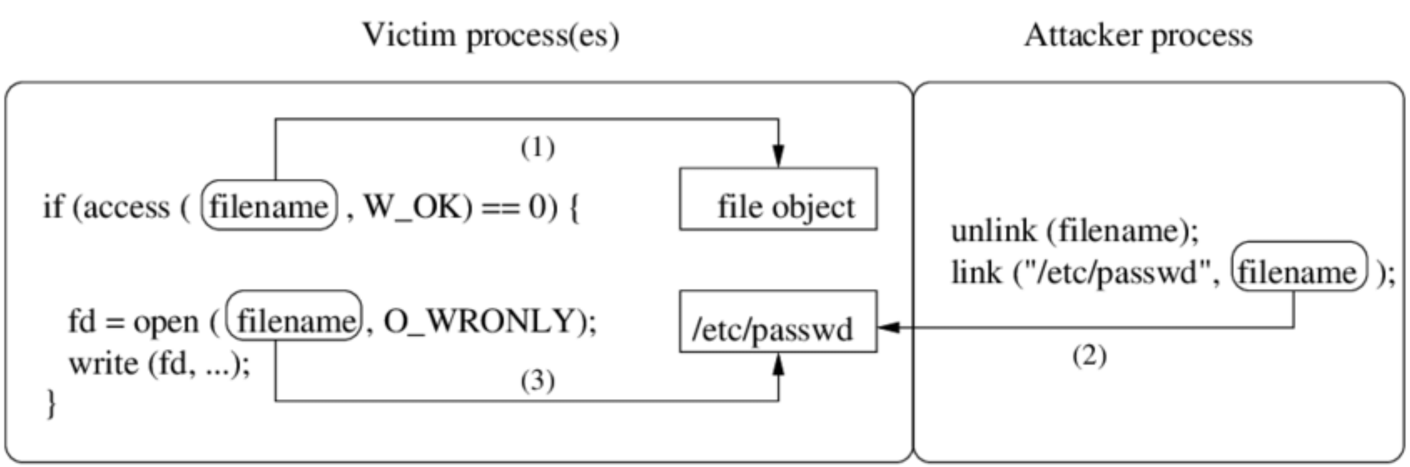
access/open Races

- *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* in two steps (simplified):
    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 not, *xterm* creates the file with the user as the owner
    2. Opens the file for writing by calling as root `open(logfilename, O_WRONLY | O_APPEND)`

Figure source: Lhee & Chapin, Detection of file-based race conditions, International Journal of Information Security 4(1-2):105-119

Example – the *xterm* race

- Attacker provides as **`logfilename`** the name of a symbolic link that toggles between a file owned by the attacker and the target file, e.g., **`/etc/passwd`**
- If **`access()`** is called while the 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

- Atomic code: executes as a single unit; nothing else can occur while the code is being executed
- Implemented using locks (semaphores) that make sure that only one process can have access to a protected object at any point in time
- Example: `synchronized` keyword in Java
  - Important as Java is a multi-threaded language
- Disadvantage: loss of performance

TOCTTOU Defences (File Access)

- Do not take file names as inputs (remove indirection)
- Use file handles or file descriptors instead
  - Don't apply **`stat()`** on a file before opening it, open the file and then apply **`fstat()`** directly on the file descriptor
- Leave access checking to the file system, i.e., do not use **`access()`**

Selected Tools

- Helgrind: thread error detector in Valgrind
  - http://valgrind.org/docs/manual/hg-manual.html
- ThreadSanitizer v2 (Chromium, for Linux)
  - Synchronization error detector based on compiler instrumentation
  - https://www.chromium.org/developers/testing/threadsanitizer-tsan-v2
- ConTest (IBM Research Haifa)
  - "Schedules the execution of program threads such that program scenarios which are likely to contain race conditions, deadlocks and other intermittent bugs … are forced to appear with high frequency
  - https://www.research.ibm.com/haifa/projects/verification/contest/index.html

Sources

- Matt Bishop, Michael Dilger: Checking for race conditions in file accesses, Computing Systems 9(2), pages 131–152,1996
- Prem Uppuluri, Uday Joshi, Arnab Ray: Preventing Race Condition Attacks on File-Systems, Proceedings SAC05, 2005
- Nikita Borisov, Rob Johnson, Naveen Sastry, David Wagner: Fixing Races for Fun and Profit: How to abuse *atime*, 14th USENIX Security Symposium, 2005
- Kyung-suk Lhee, Steve J. Chapin: Detection of File-Based Race Conditions, International Journal of Information Security, vol. 4, 2005
- Martin Abadi, Cormac Flanagan, Stephen N. Freund: Types for safe locking: Static race detection for Java, ACM Transactions on Programming Languages and Systems (TOPLAS), volume 28(2), pages 207 – 255, 2006

Virtual Memory

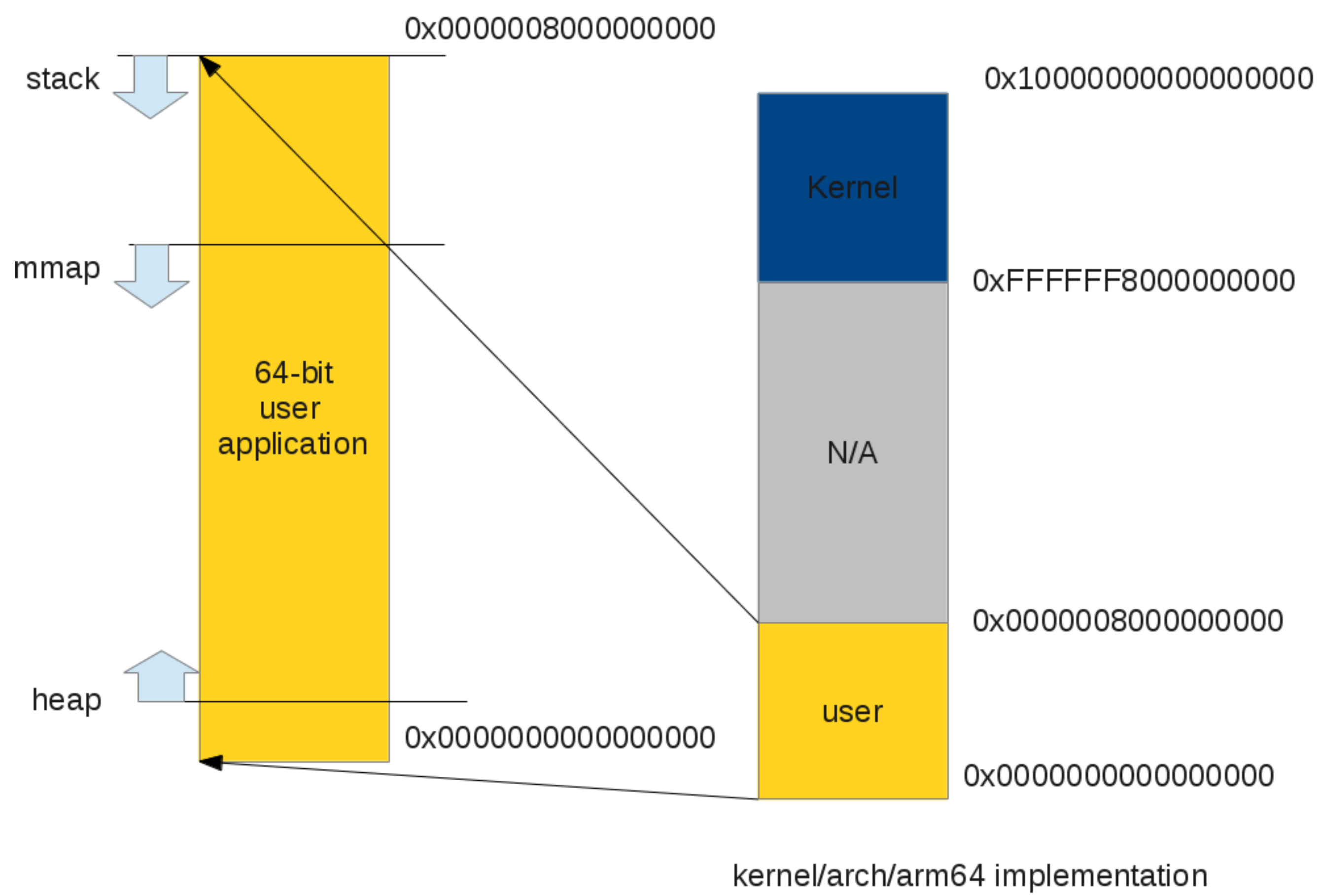
Yet another abstraction

Virtual Address Space

- OS assigns each process a virtual address space
- Feature of modern operating systems: Linux maps kernel into virtual address space of user processes
  - Faster context switch for better performance
  - Windows does it slightly differently
  - Virtual address space contains user data and control data
- Translation tables map virtual addresses to physical addresses, together with permitted access modes
- A CPU register holds address of the current table
  - Context switch updates register content to new table

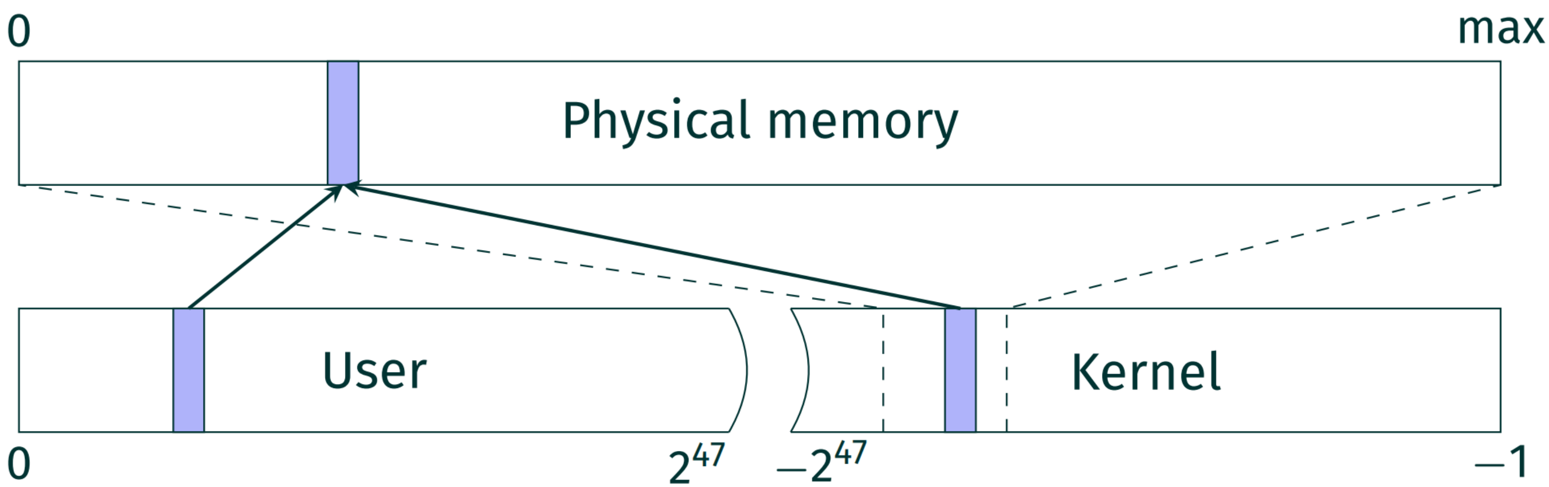
http://thinkiii.blogspot.com/2014/02/

Virtual Address Space



https://www.usenix.org/sites/default/files/conference/protected-files/security18_slides_lipp.pdf

Direct Physical Map

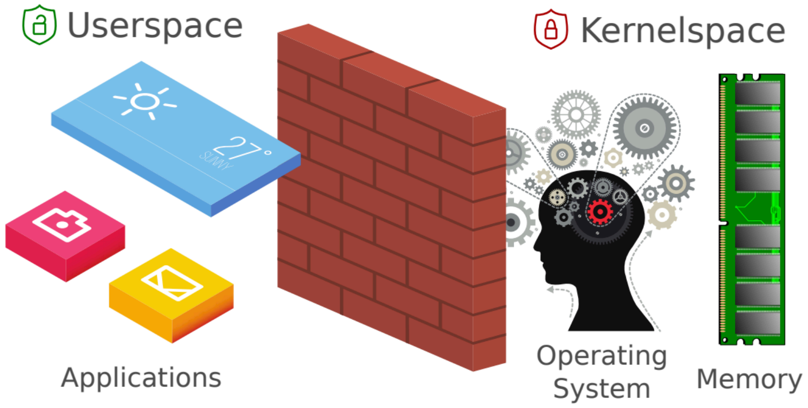


- Kernel is typically mapped into every address space
- Entire physical memory is mapped in the kernel

Memory Isolation

- Kernel memory can only be accessed if the supervisor bit in the CPU is set
- Processes running in user space are not permitted to access virtual addresses the kernel is mapped to
  - Kernel access technically possible but blocked logically

At this level of abstraction, there is no harm in mapping kernel memory into the virtual address space of user processes



Virtual Memory Access

- To load data from the main memory into a register, the data in the main memory is referenced using a virtual address
- CPU translates virtual address into a physical address, checks permission bits of virtual address
- Exception will be raised when access to the given address is not permitted

```
char data = *(char*) 0xffffffff81a000e0;
printf("%c\n", data);
```

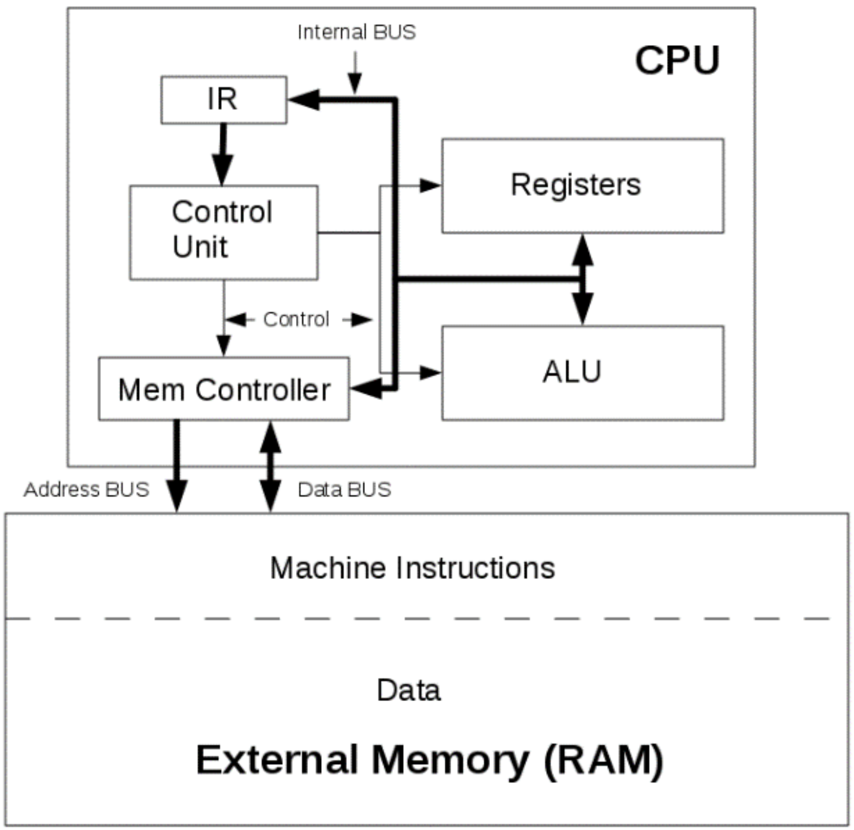
```
segmentation fault
```

Central Processing Unit

Yet another abstraction

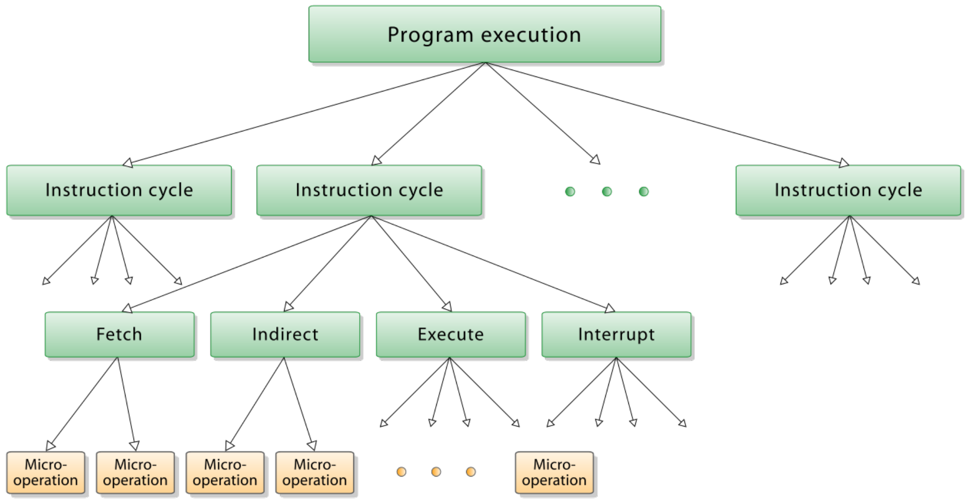
http://www.cs.uwm.edu/classes/cs458/Lecture/HTML/Images/cpu_block.png
https://www.usenix.org/sites/default/files/conference/protected-files/security18_slides_lipp.pdf

"Traditional" CPU



- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, …)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

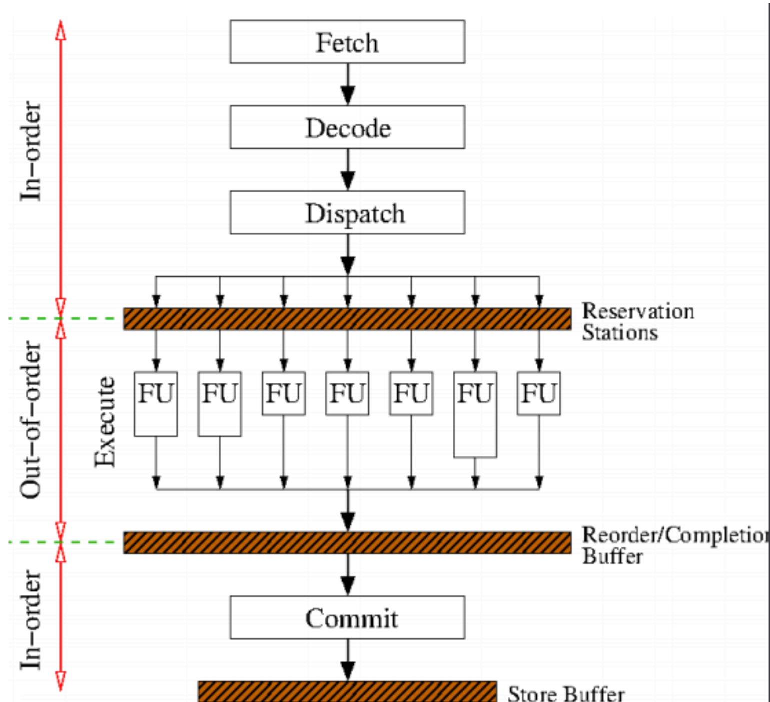
"Modern" CPU



- Micro-operations (μOPs) are low-level instructions used to implement complex machine instructions
- The actual executions are performed by μOPs
  - The CPU no longer directly executes machine instructions
  - Machine instructions are decoded into μOPs and those μOPs are executed

Out-of-Order Execution

- x86 instructions are fetched by the front-end from memory and decoded to micro-operations (µOPs)
  - Reorder Buffer is responsible for register allocation, register renaming and retiring the µOPs
  - µOPs are forwarded to the Unified Reservation Station, queues the operations on exit ports connected to Execution 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 the predicted path that do not have any dependencies can be executed in advance



In-Order Retirement

- If the prediction was correct, results can be immediately used
- If the prediction was incorrect, perform a rollback by clearing the reorder buffer and re-initializing the Unified Reservation Station
  - Rollback reverts to saved CPU state (register contents)
  - Instructions rolled back will be referred to as transient instructions
- When µOPs finish execution, they retire in order and their results are committed
  - During retirement, interrupts and exceptions raised during the execution of the instruction are handled
  - Flush pipeline and recover state in cases of exceptions
- At this level of abstraction, transient executions leave no side effects

Speculative Execution

A common optimization technique in modern processors used to achieve high performance

- Concept: instructions are executed ahead of knowing if they are required, to prevent a delay by idling;
- If it turns out that the work was not needed after all, changes made by the work are reverted in a way that "leaves no trace" to the process
- Even though the execution result may not commit to architecture, the state of microarchitecture is already changed
  - E.g., cache state

Secret Stealing under Speculative Execution

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

```
data = [1, 2, 3, 4];
input = 1000;

if (input < data.size) {
    secret = data[input];
    y = prob[secret];
}
```

Secret in data[1000]

This is in cache

Now the question is: how to get "secret" from cache?

Recall: Cache-Based Side Channels

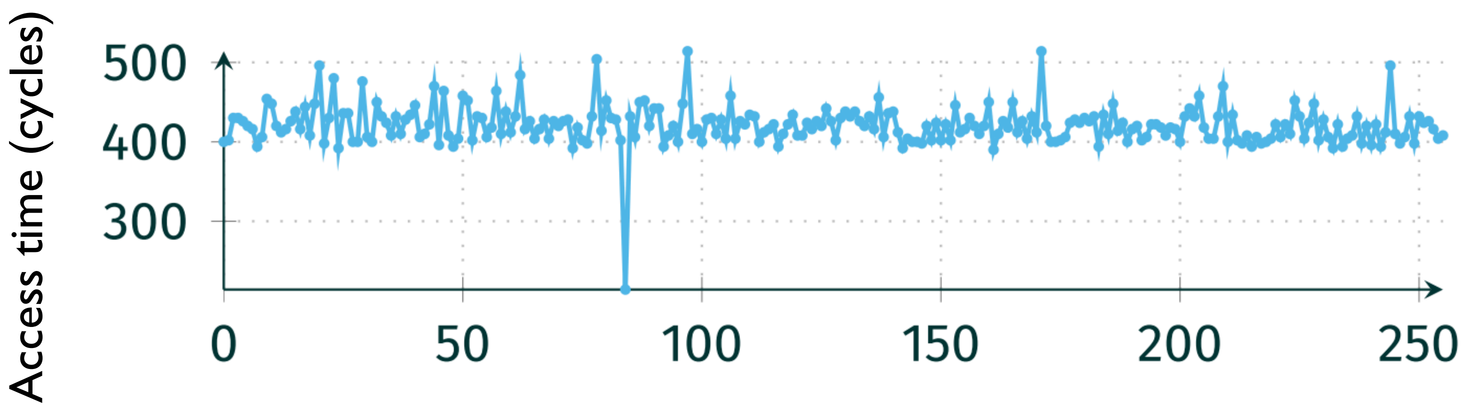
- The "further away" from the CPU data is stored, the longer it takes to read a memory line
- Read time thus tells whether a memory line is cached or not, and where it is cached
- Side channel: move memory line out of the cache, let execution continue, then read the memory line again
  - Flush + Reload: flush target cache line with a suitable instruction (**`CLFLUSH`**)
  - Evict + Reload: write as much data as necessary to the cache so that all previous content is evicted
- Read time tells whether the line had been used by someone else between the flush/evict and the reload

Recall: Cache-Based Side Channels

- Can be used to observe other processes running on the same CPU
  - Cache-based side channels for leaking secret cryptographic keys
- Can be used to observe from user space what had been done while the system had been operating in kernel space
  - Cache-based side channels for leaking secret data (Meltdown)
- Dangers of abstraction:
  - Despite rollback to the saved CPU state, transient executions may have side effects that persist

Cache Side Channel!

- 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

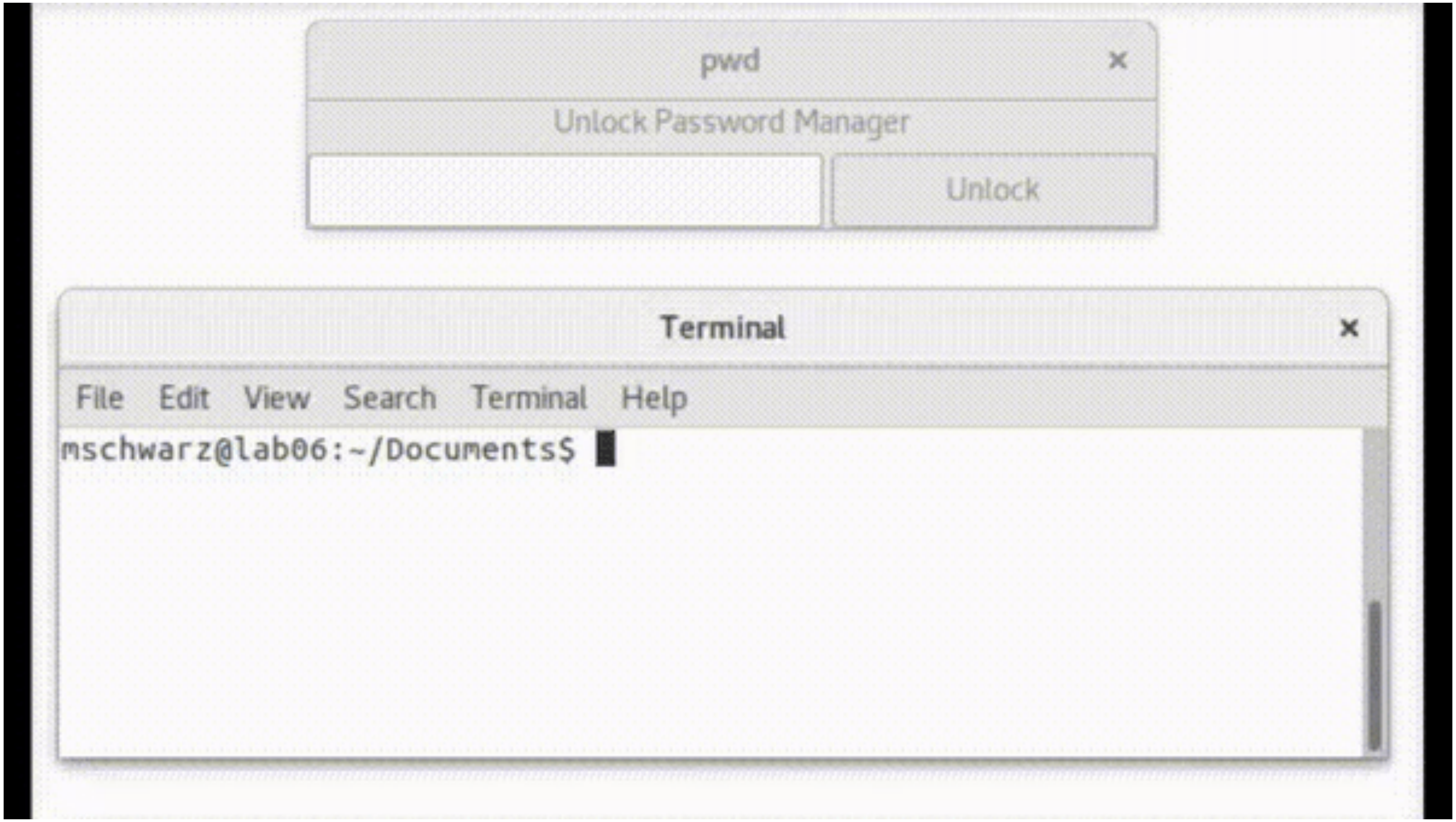


Meltdown

(Penetrate + Patch)^2

Meltdown

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



Recall: Stack Buffer Overflow Attacks

- Stack frame is a data structure that contains user data and control data
- Overflowing a buffer for a local variable (user data) can overwrite return address (control data)
- Attacker may overwrite return address with the address of an '**`eval`**' library function and pass a command as an argument
- **`eval`** will execute that command (return-to-libc)

Recall: Patch – ASLR

- Attacker needs to know the address of the library function
- Randomizing memory layout raises the bar for the attacker; typically applied to user-space
- KASLR (kernel ASLR) randomizes the location of the kernel at boot time
  - https://lwn.net/Articles/569635/

Penetrate – Meltdown

- Step 1 (load). Transient instruction loads the content of an attacker-chosen illegal memory location into a register
  - Illegal: the attacker is not permitted to access the location
- Step 2 (persist). Another transient instruction accesses a cache line based on the content of that register
- Step 3 (read). Flush+Reload determines which cache line was accessed; this reveals (leaks) the value stored at the chosen inaccessible memory location
- Repeat these steps for all memory locations to dump the kernel memory (includes the entire physical memory)

Meltdown – The Code

```
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
```

- Line 4: target kernel address stored in RCX register; byte value at that address is loaded into AL (least significant byte of RAX)
  - MOV instruction is fetched by the core, decoded into µOPs, allocated, and sent to the reorder buffer
  - Architectural registers (e.g., RAX and RCX) are mapped to underlying physical registers to prepare for out-of-order execution

Meltdown – Step 1 (Read the Secret)

```
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
```

- Instructions in lines 5--7 are also decoded and allocated as µOPs
- These µOPs are sent to the reservation station where they are held waiting to be executed
  - µOP can be executed once execution units have spare capacity and all operand values have become available

Meltdown – Step 1 (Read the Secret)

```
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
```

- When the kernel address is loaded in line 4, it is likely that the CPU already had issued subsequent instructions as part of an out-or-order execution, and their μOPs are waiting in the reservation station for the content of the kernel address to arrive
- As soon as the fetched data is observed on the common data bus, speculative execution of those μOPs can begin

Meltdown – Step 1 (Read the Secret)

```
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
```

- When the µOPs finish their execution, they retire in order, and their results are committed to the architectural state
- During the retirement, interrupts and exception raised during the execution of the instruction are handled
- When the illegal MOV instruction in line 4 is retired, the exception is registered; all results of subsequent instructions that were executed out of order are eliminated

Making Side Effect Persistent

- The transient instructions in step 1 have loaded the target value into a register
  - Does not help the attacker; the register content will be reset when the MOV instruction is retired
- Attack needs further transient instructions that encode the target value in the cache state
  - Modifications to the cache are not reverted on rollback!
- Race condition: if these transient instructions get executed before the exception is raised when retiring MOV, the target value is recorded in the cache state

Meltdown – Probe Array

- Meltdown allocates a probe array in memory and ensures that no part of this array is cached
- To encode the target, a transient instruction makes an indirect memory access to the probe array at an address calculated from the target value
- Target value multiplied by page size so that accesses to the array have a large spatial separation
  - Prevents the hardware prefetcher from loading adjacent memory locations into the cache as well
- A single byte is read in one run, hence for 4KB pages the probe array is 256×4096 bytes

Meltdown – Step 2 (Transmitting the Secret)

```
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
```

- Line 5: target value is multiplied by the page size (4 KB)
  - Shift to the left by 12 positions (`0xc`)
- Technicality: out-of-order execution has a noise bias towards register value '0'
- Line 6: when '0' has been read, go back to reading the target

Meltdown – Step 2 (Transmitting the Secret)

```
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
```

- Line 7: target × 4KB is added to the base address of the probe array (in RBX); this is now the target address of the side channel
- This address is read, caching the corresponding cache line
- Thus, the transient instruction sequence affects the cache state in a way that depends on the target value read in line 4
- The faster step 2 is executed, the higher the chance of winning the race against the permission check

Meltdown – Step 3 (Receiving the Secret)

- When the transient instruction sequence of step 2 is executed, one cache line of the probe array is cached
- Position of the cached memory line within the probe array depends on the 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 a single cache hit will be observed
- The number of the page containing the cached memory line equals the target value

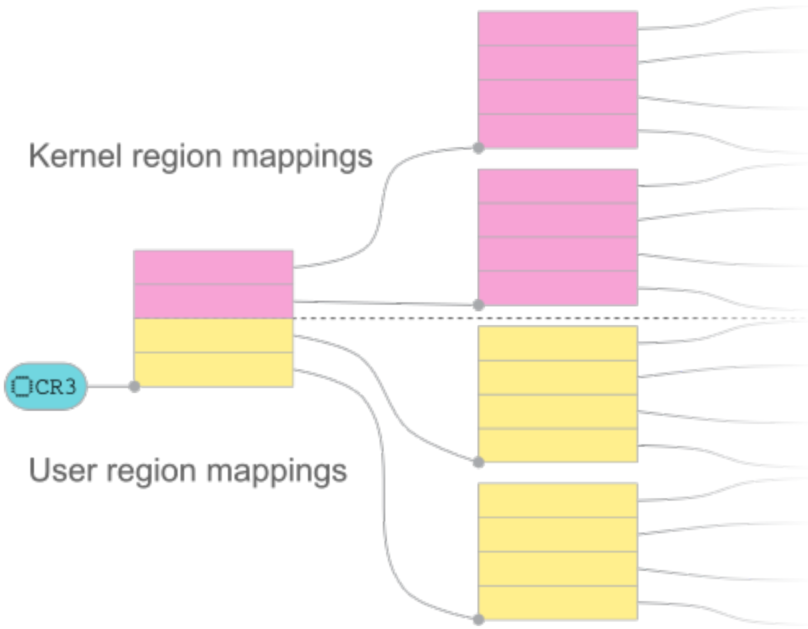
Dumping the Entire Physical Memory

- By repeating all three steps, the attacker can dump the entire memory by iterating over all addresses
- Memory access to a kernel address raises an exception that terminates the program
- Meltdown thus uses a method for handling or suppressing exceptions (several options exist)
- When the entire physical memory is mapped into the kernel address space of a user process, the entire physical memory of the machine can be read
- In particular, one can get the location of the kernel

KAISER

Towards the next patch …

Virtual Address Space

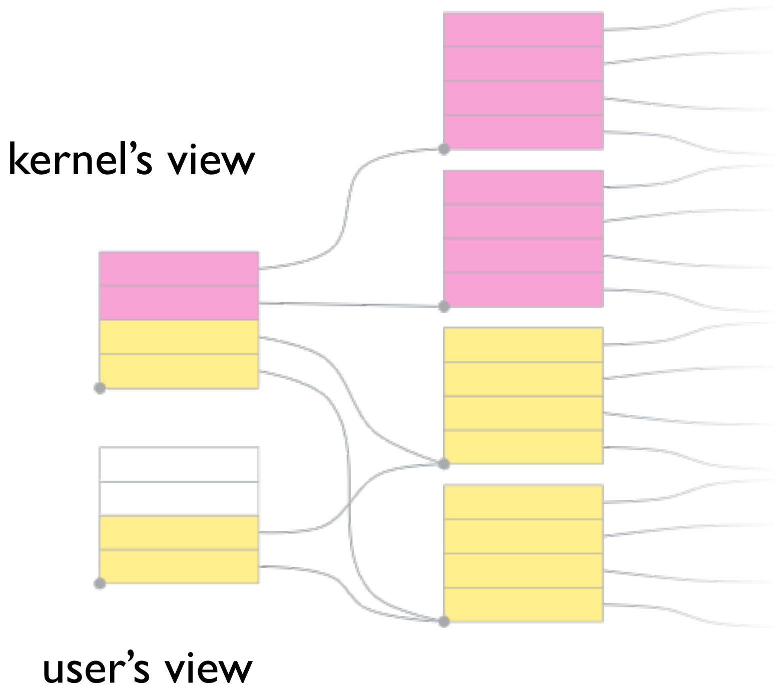


- Page tables are used to store the address of physical pages
  - Each process has its own page table
  - Kernel announces the current address mapping to the CPU by writing to a special register (CR3)
- Operating system maps kernel into the virtual address space of every user process
  - Kernel space protected with permission bits
  - Sometimes may fail: situation encountered in Meltdown

Patch – KAISER

- KAISER does not map kernel memory in the user space, except for certain parts required by the x86 architecture (e.g., interrupt handlers)
- No valid mapping to kernel memory or physical memory (via the direct-physical map) in user space
- Such addresses can therefore not 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

Stronger Kernel Isolation



- Shadow address space to separate kernel space & user space
- Keep two copies of the page table
- Shadow page table only keeps the bare minimum required for context switch
- Memory not mapped cannot be accessed
- Would stop Meltdown

Kernel Page-Table Isolation

- KAISER created before Meltdown was discovered; other KASLR leaks were already known
- Needed to be adapted to defend against Meltdown
- Kernel page-table isolation (KPTI) is the Linux kernel feature defending against Meltdown
- Improves kernel hardening against attempts to bypass KASLR

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"

The Way Ahead?

- Don't increase CPU complexity in order to hang on to 1970s programming languages and execution models
- Design simpler CPUs along with languages that can get most concurrency out of them safely
  - Reduce surprises for programmers that arise out of hidden complexity
  - High-performance computing should focus on GPUs instead of CPUs: simpler processors with many fast data paths
- Remove a layer of abstraction for more clarity

Sources

- Moritz Lipp et al. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018, https://arxiv.org/pdf/1801.01207.pdf
- Paul Kocher et al. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018, https://arxiv.org/pdf/1801.01203.pdf
- http://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/
- https://www.wired.com/story/meltdown-spectre-bug-collision-intel-chip-flaw-discovery/
- https://armkeil.blob.core.windows.net/developer/Files/pdf/Cache_Speculation_Side-channels.pdf

Conclusions

Concurrency

- Concurrency is a useful feature, e.g. for improving performance
- Concurrency can lead to race conditions when the order of accesses to a data item matters
- Race conditions can be security relevant (TOCTTOU)
- Synchronizing access to a data item eliminates race conditions
- Synchronizing access incurs a performance penalty

Dangers of Abstraction

- Abstraction is a main cause for the success of Computer Science
  - Application Program Interfaces (APIs)
  - High level programming languages
  - Machine instructions
  - Even the CPU has become an abstraction
- Gaps between abstraction and implementation can open the door for attacks
- The implementation may behave in ways that do not correspond to any behaviours of the abstraction

Constructive & Analytic Security

- Constructive security is concerned with developing and deploying security mechanisms
  - Cryptography, access control, intrusion detection, …
  - Approach similar to other branches of Computer Science
- Analytic security is concerned with finding weird machines in a system
  - Very different mindset compared to other branches of Computer Science
  - You have to understand how a system can be made to do things it was not designed for

Sources on Vulnerabilities

- Mainstream media will report on major vulnerabilities
  - Tell you that there is a 'big' problem (awareness)
  - At best brief, high level technical explanations
- Security advisories
  - Tell sysadmins how to fix their systems (patch)
  - Sometimes links to technical details are given
- Blackhat talks, whitepapers, professional publications
  - Tell you how a vulnerability could be exploited (understand)
  - Often address readers with a strong technical background