

**Please check your attendance  
using Blackboard!**

# **Lecture 5 – Various Software Vulnerabilities**

[COSE451] Software Security

Instructor: Seunghoon Woo

Spring 2024

# Access Controls and Software Security

- **Privilege escalation**

- Demo: Windows privilege escalation attack

- Exploiting CVE-2017-0213 vulnerability

- Windows allows an elevation privilege vulnerability when an attacker runs a specially crafted application

- <https://www.youtube.com/watch?v=f6x0hBerObM>

# Overview

- **Various Software Vulnerabilities**

# Software Vulnerabilities

- **Vulnerabilities that frequently occur in software ecosystem**
  - Race condition
  - Format string bug

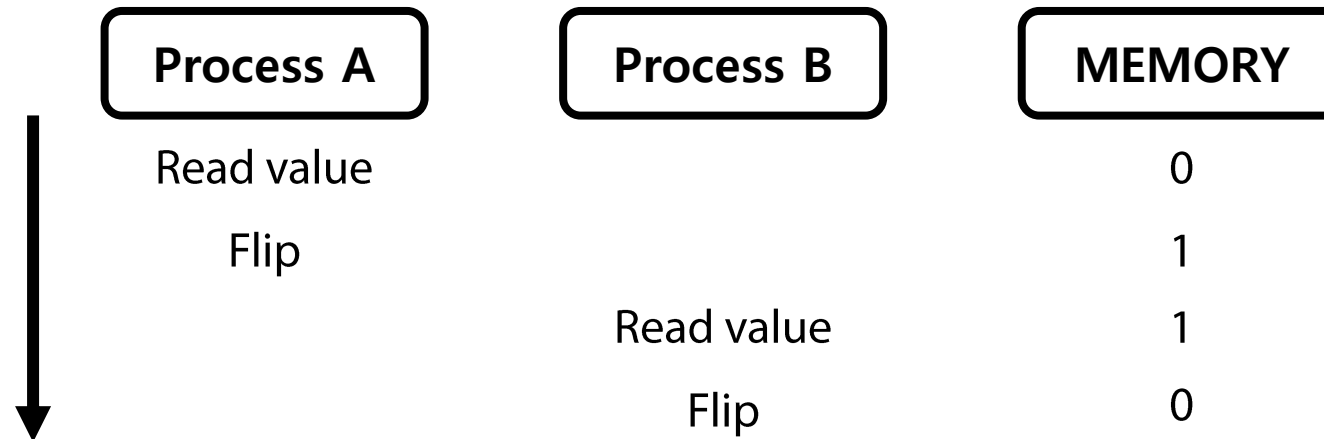
# Race condition

- **A state in which two or more concurrent processes (or threads) compete to access one resource (concurrency error)**
  - Process: a program that runs on a computer
  - Thread: the entity that actually performs work within a process
    - Every process has one or more threads to perform its work
    - A process with two or more threads is called a multi-threaded process
  - CWE-362 (21)
    - Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

# Race condition

- **Example**

- Two processes (A and B) attempting a bit flip
- Case 1:

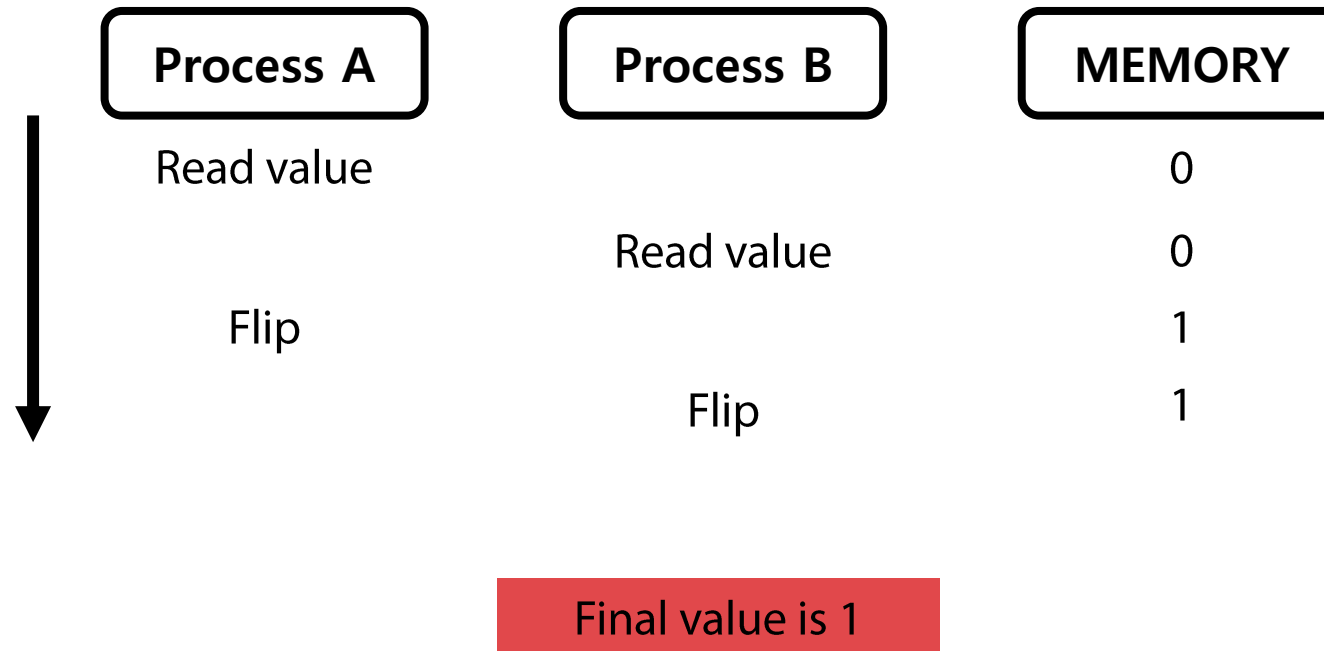


Final value is 0

# Race condition

- **Example**

- Two processes (A and B) attempting a bit flip
- Case 2:





# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - One of the most dangerous vulnerabilities discovered in Linux kernel
  - Exploiting race conditions to cause writes to read-privileged files
    - Abuse of kernel Copy On Write (COW)
    - Any normal user can become the root!



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Copy On Write

- A resource-management technique used in computer programming to efficiently implement a "duplicate" or "copy" operation on modifiable resources

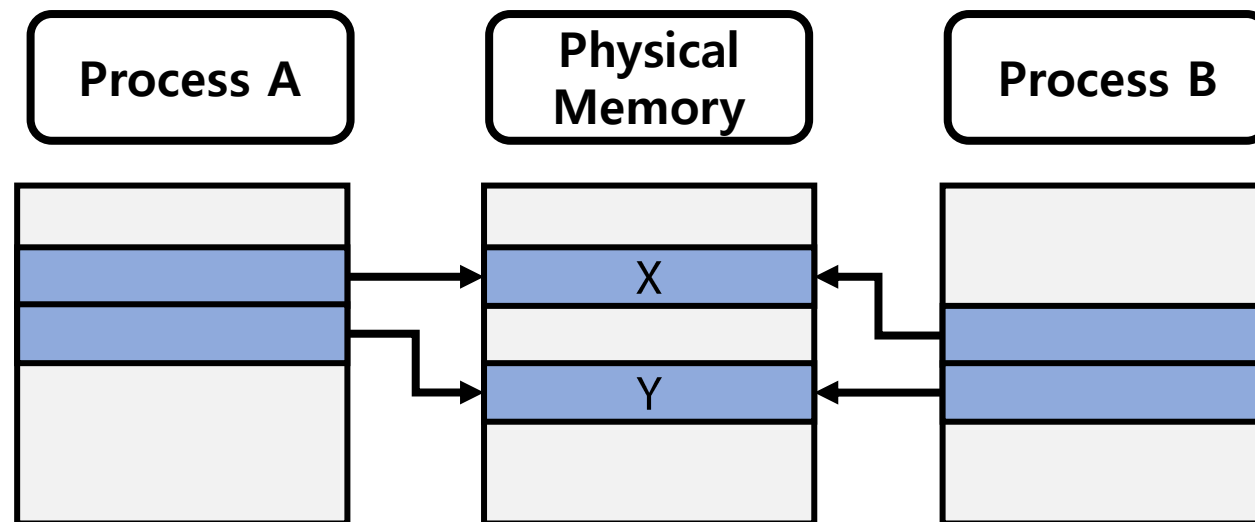


# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Copy On Write

- A resource-management technique used in computer programming to efficiently implement a "duplicate" or "copy" operation on modifiable resources

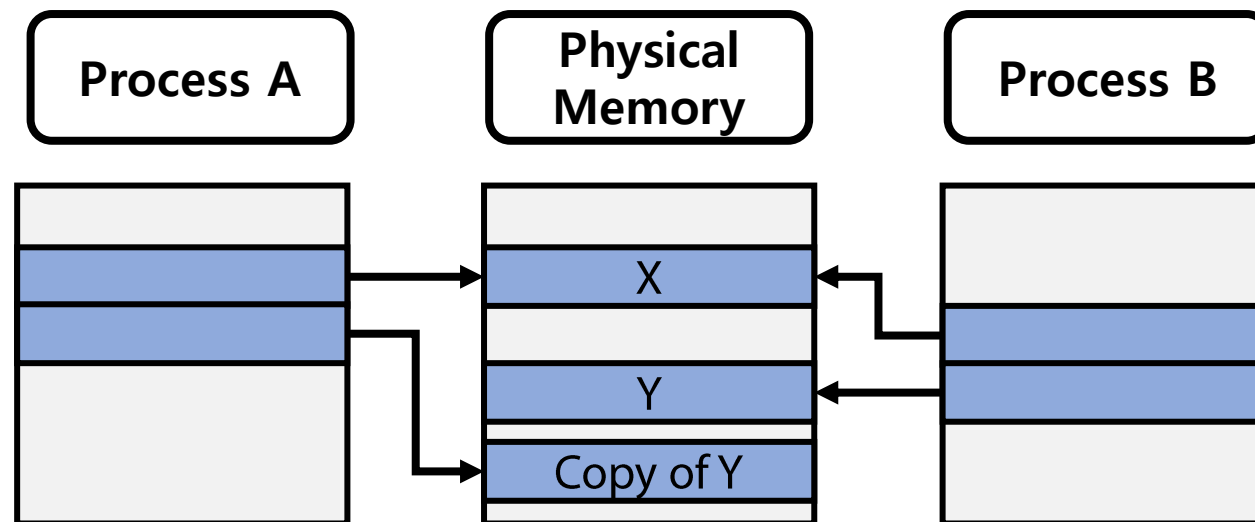


# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Copy On Write

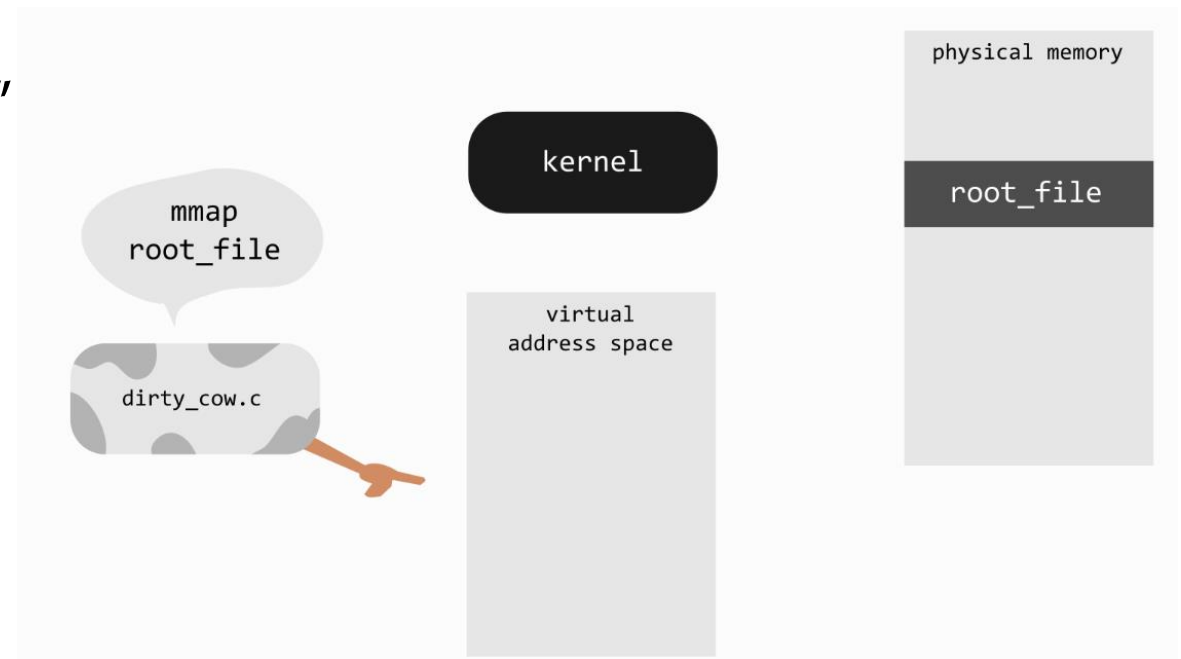
- A resource-management technique used in computer programming to efficiently implement a "duplicate" or "copy" operation on modifiable resources



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

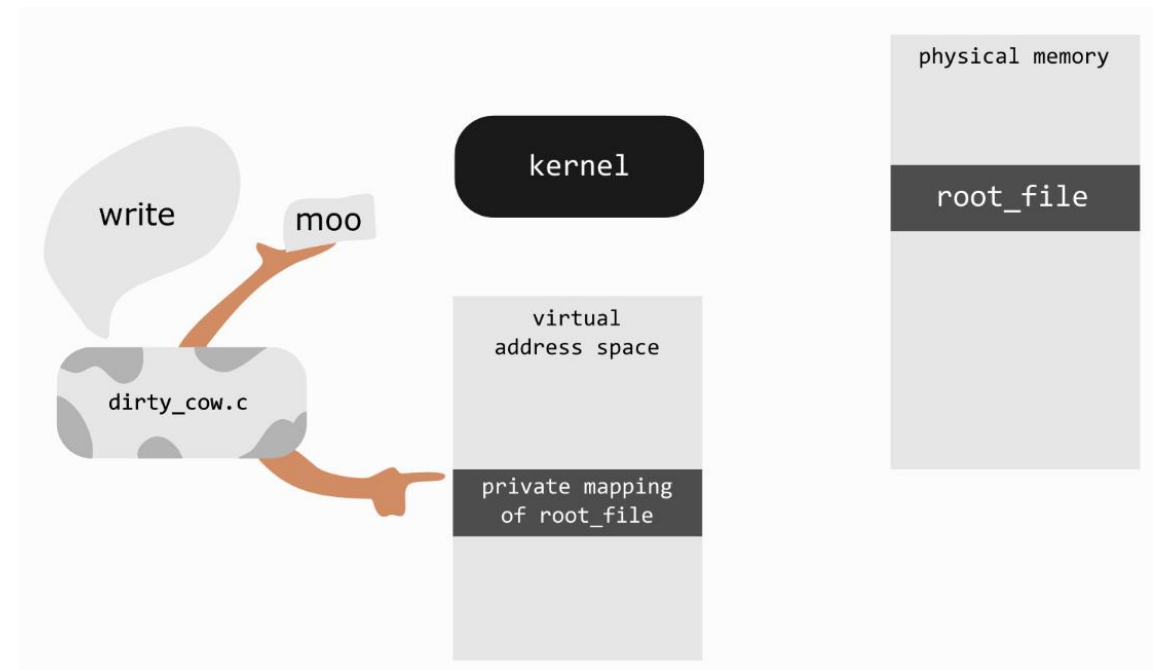
1. We ask the kernel to create a private mapping of “root\_file” (read-only file)



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

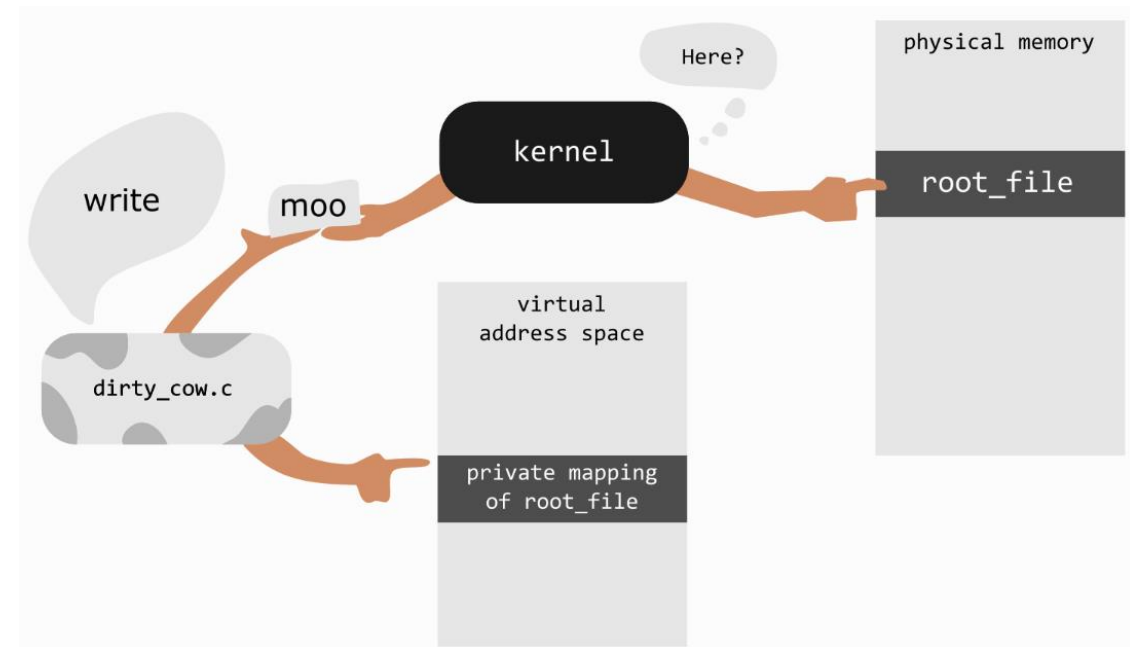
2. We ask the kernel to write “moo” to our private mapping of “root\_file” through “/proc/<PID>/mem”
  - /proc/<PID>/mem: a binary image representing the process's virtual memory
    - Here, a representation of dirty\_cow.c's virtual memory



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

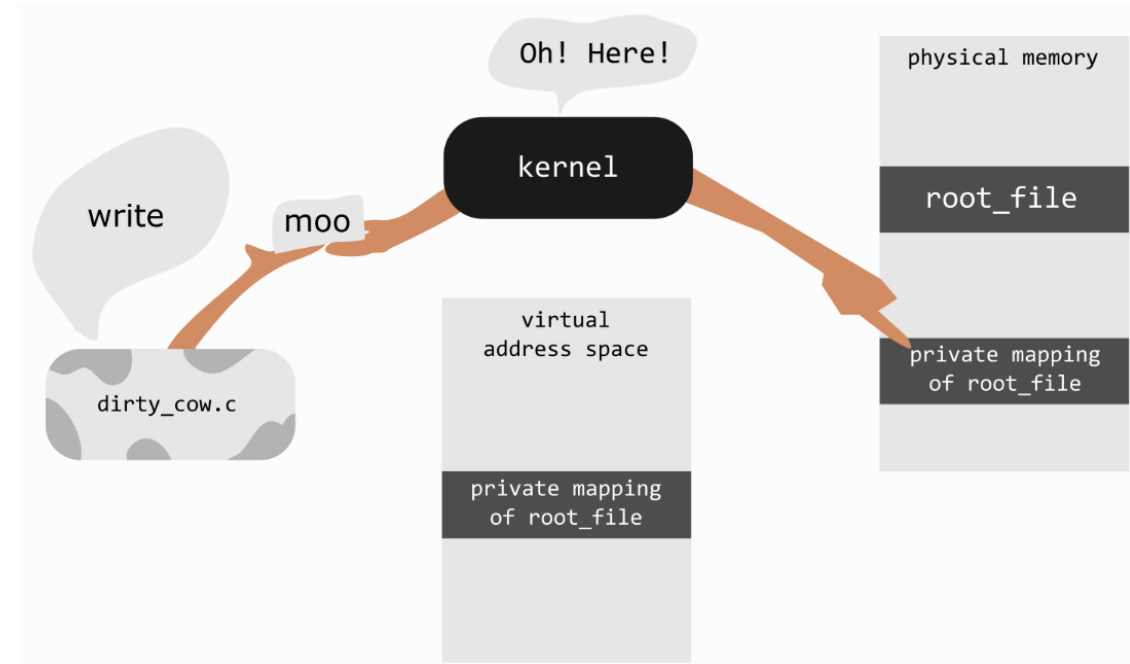
3. The kernel, initially, probes the “root\_file”, but writing to this file is infeasible



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

4. Allocate the copied memory area through Copy On Write

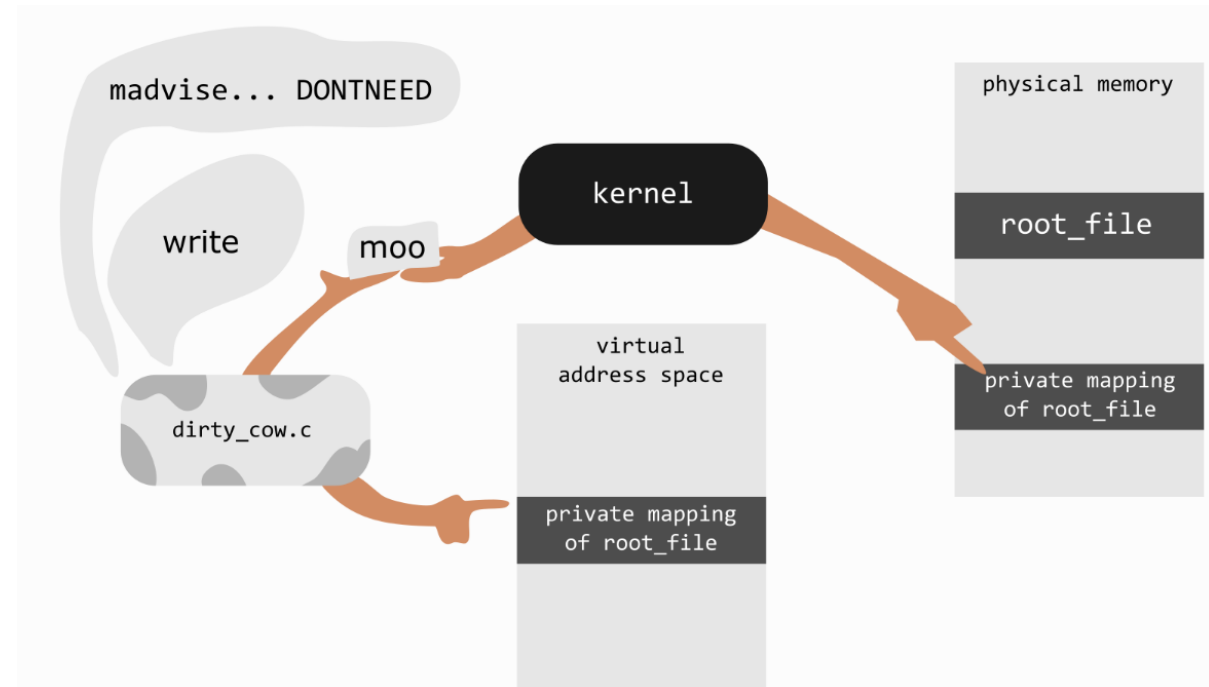




# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

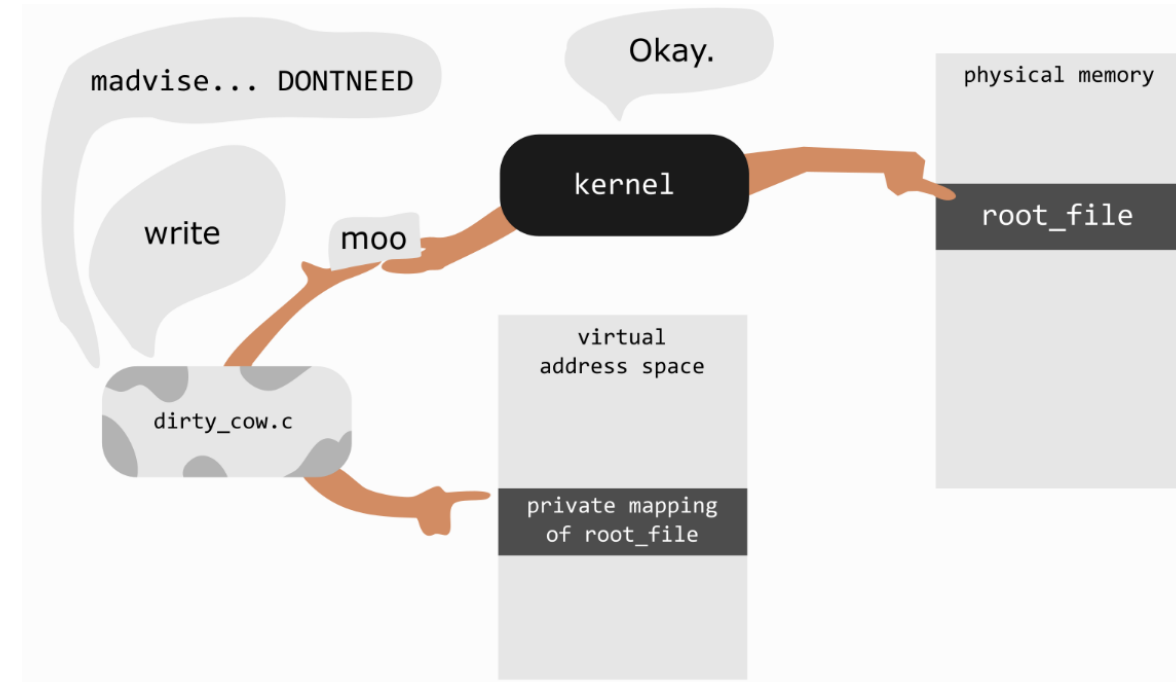
5. We use `madvise` to advise the kernel that we do not need (`MAVD_DONTNEED`) our private mapping anymore
  - Kernel forgets about our private mapping!



# Race condition

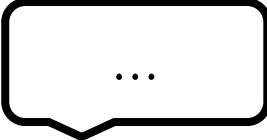
- **Real world example: Dirty COW (CVE-2016-5195)**

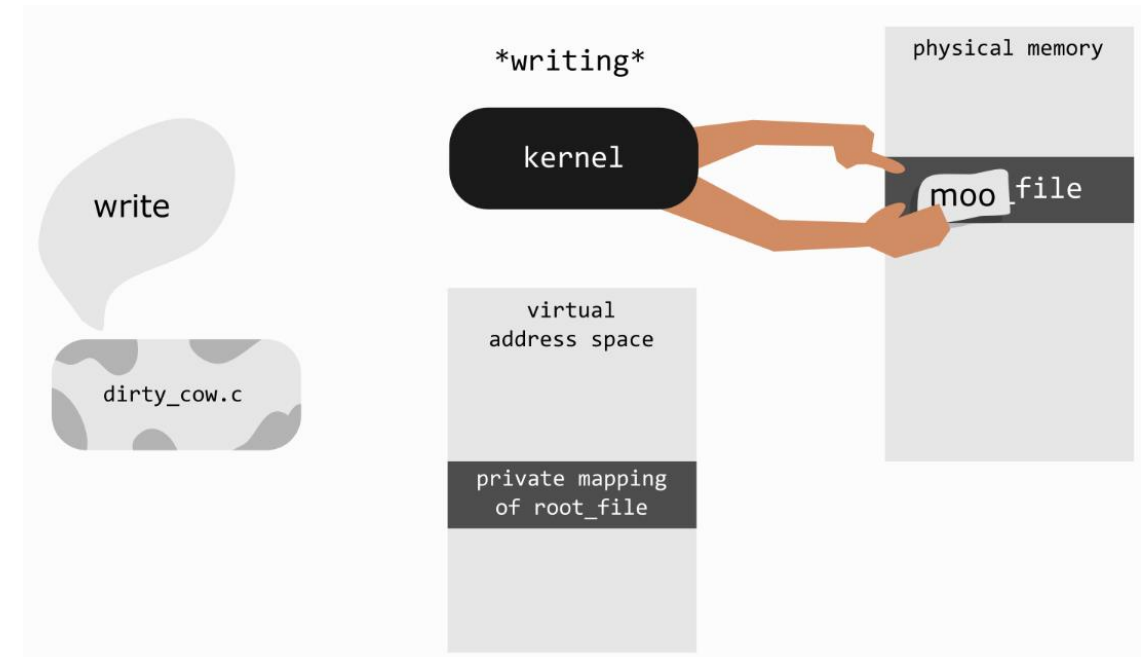
6. The kernel is tricked into thinking our write was for the original "root\_file"



# Race condition

- Real world example: Dirty COW (CVE-2016-5195)

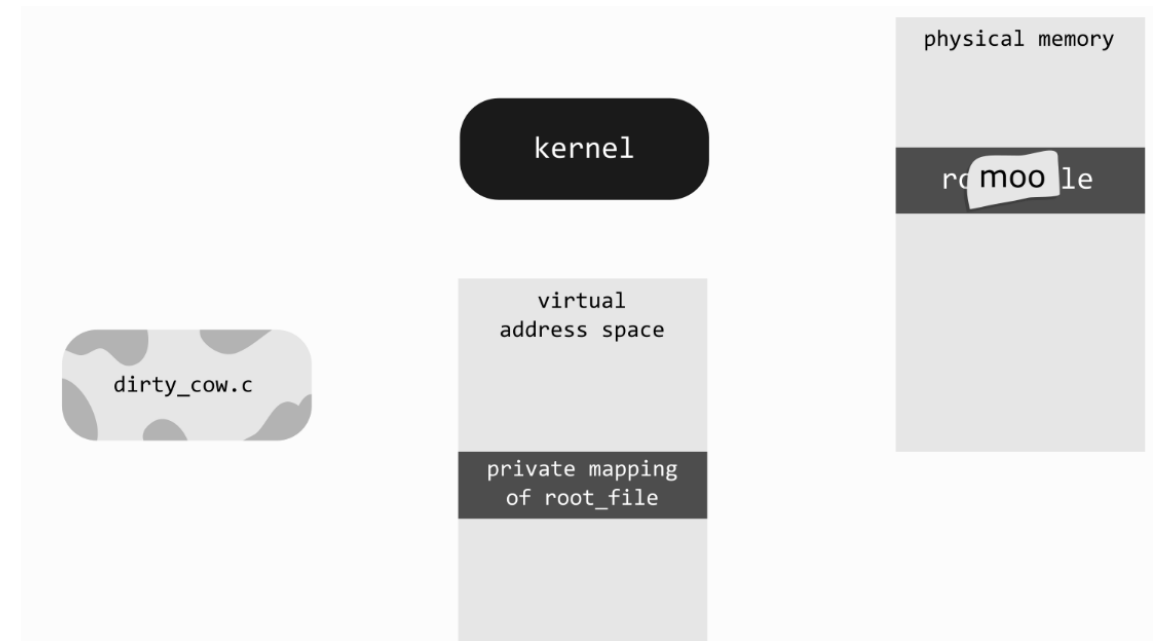
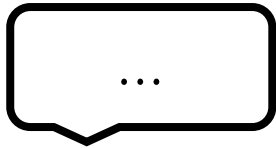
7. 



# Race condition

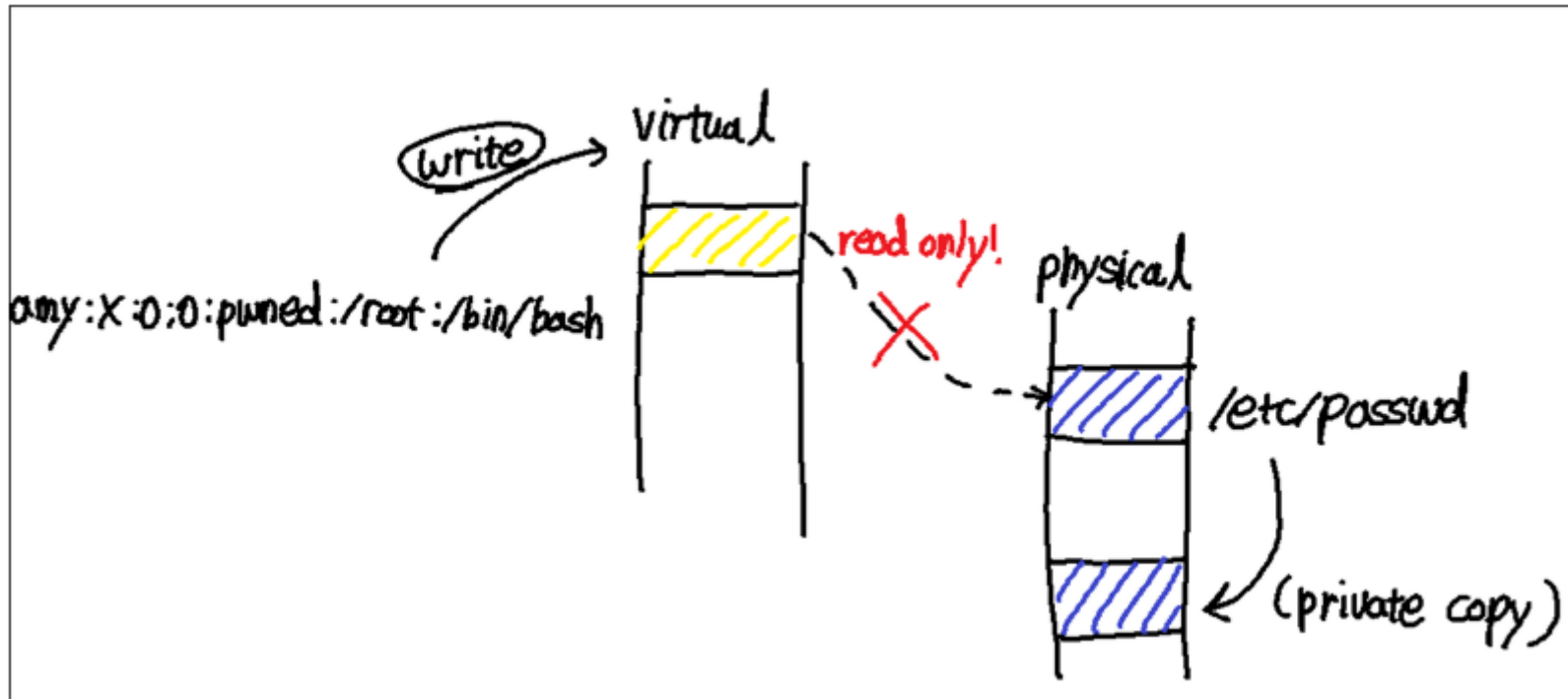
- Real world example: Dirty COW (CVE-2016-5195)

8.



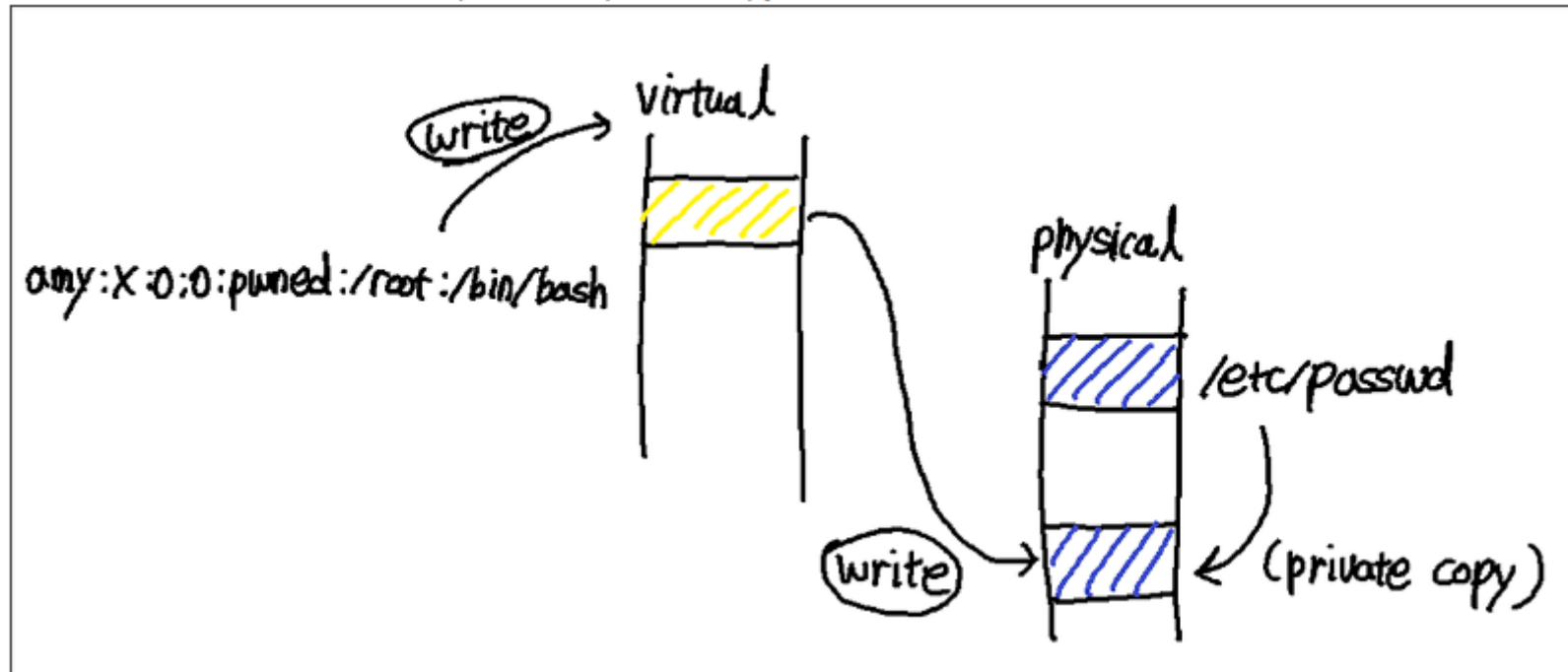
# Race condition

- Real world example: Dirty COW (CVE-2016-5195)
  - Normal case



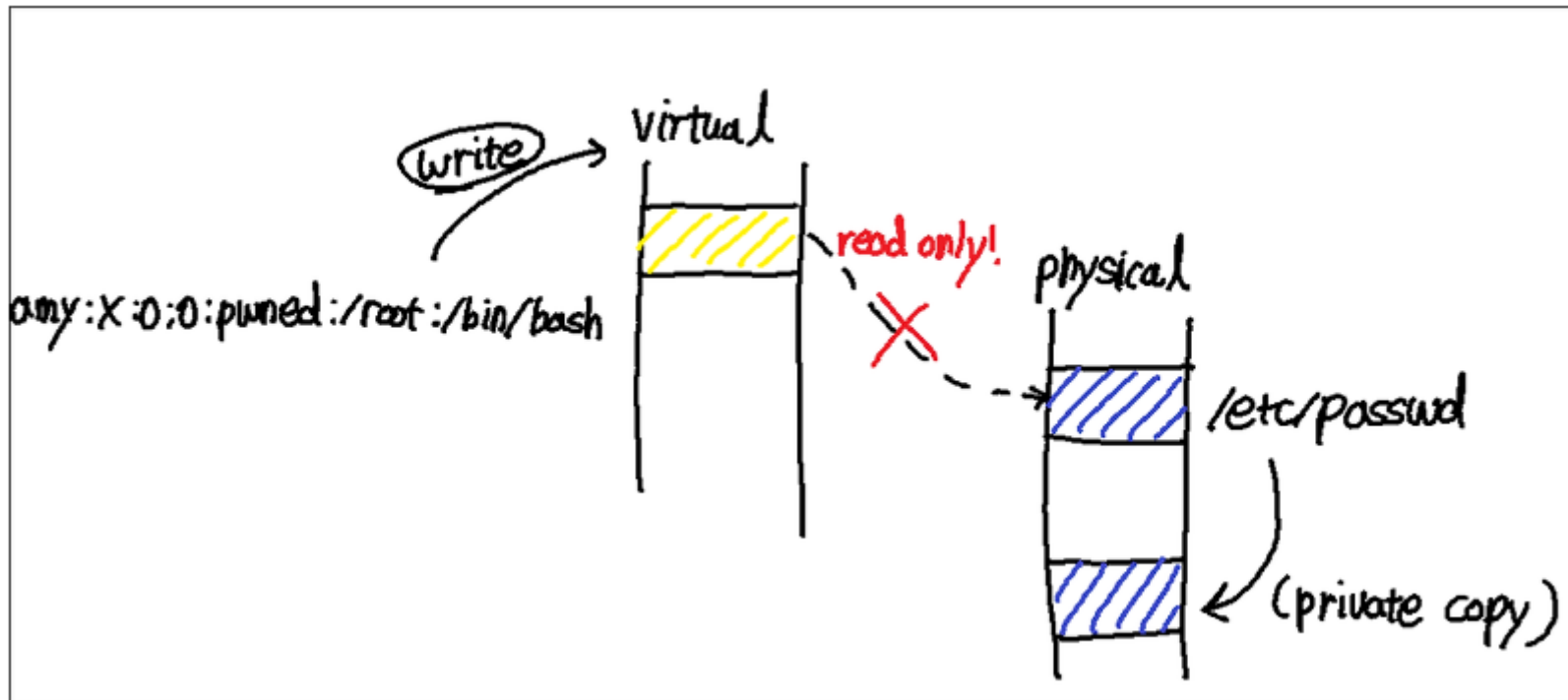
# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - Normal case



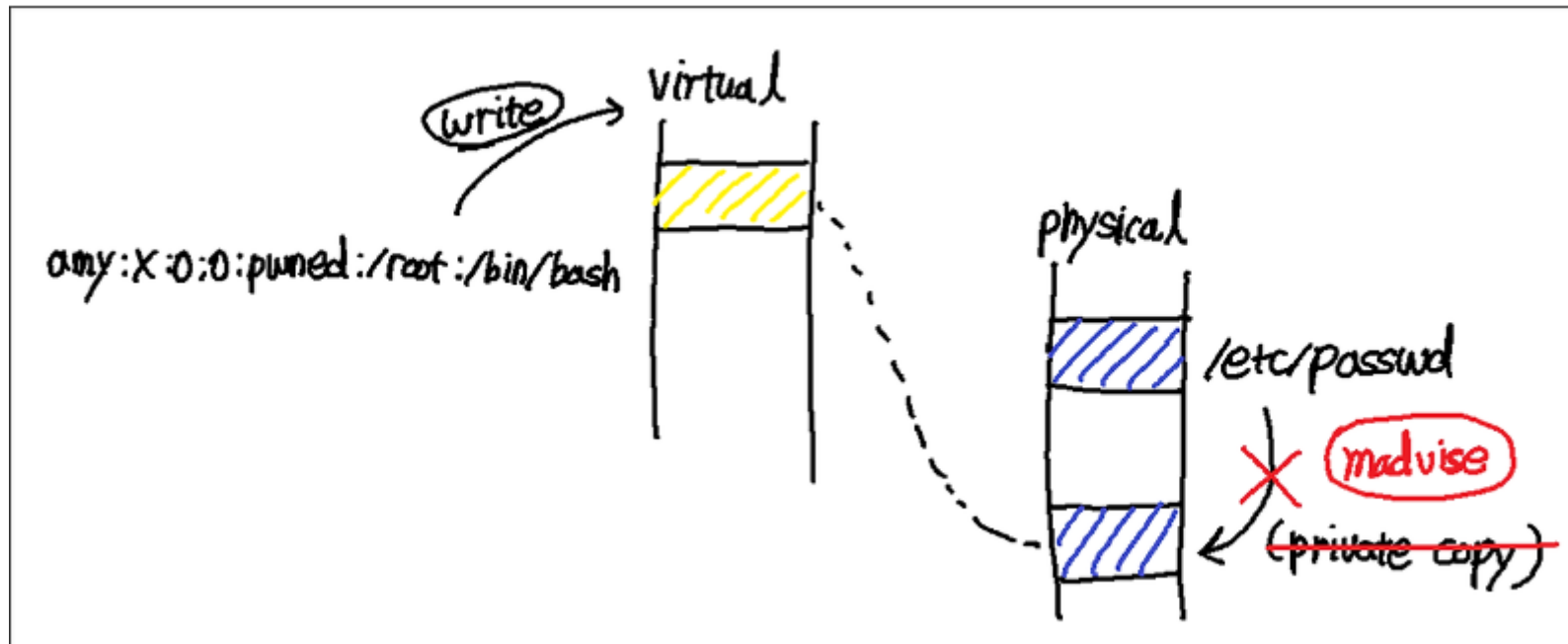
# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - Exploiting Dirty COW



# Race condition

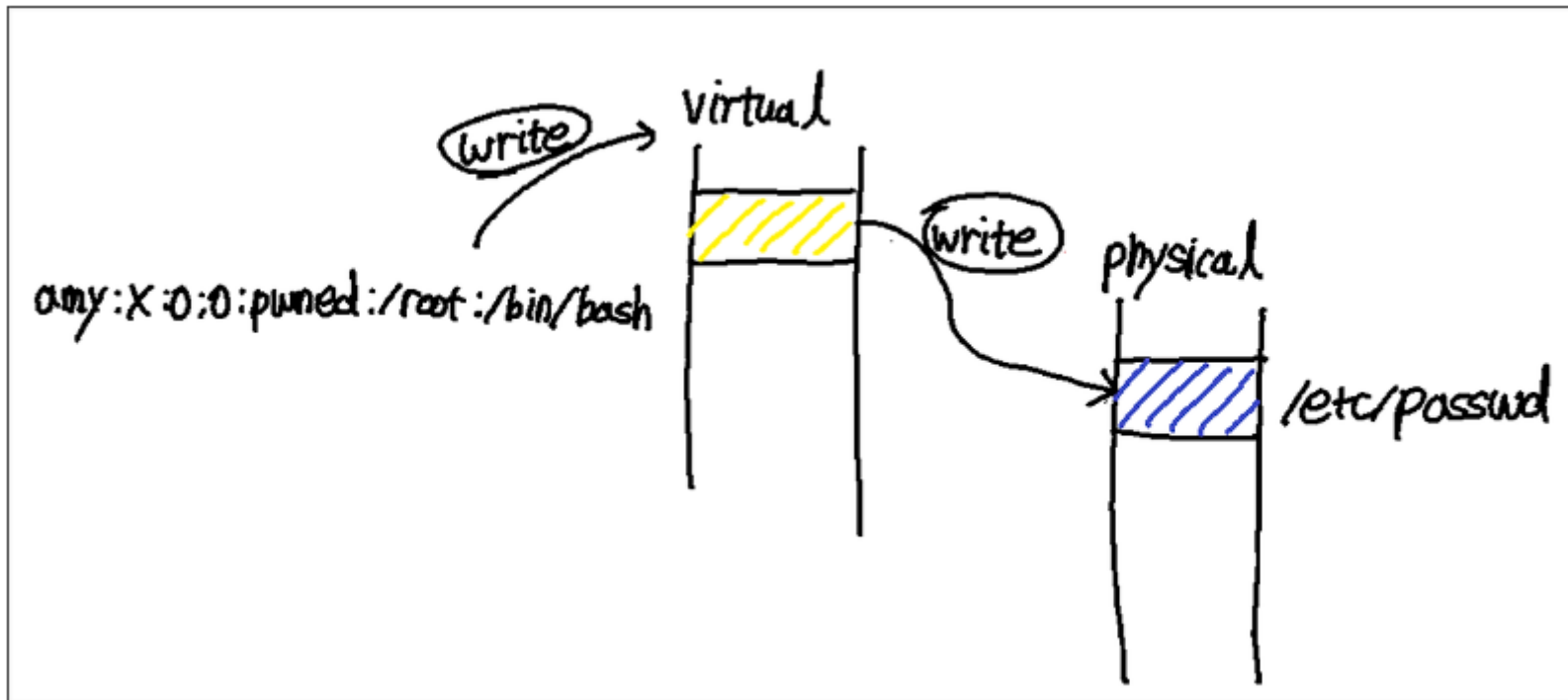
- Real world example: Dirty COW (CVE-2016-5195)
  - Exploiting Dirty COW





# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - Exploiting Dirty COW



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - By exploiting this vulnerability, attackers can add their own root accounts to the “/etc/passwd” file (privilege escalation)

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Vulnerable source code

- When writing through `"/proc/self/mem"`, `__get_user_pages()` is executed

```
1  __get_user_pages(){
2      //Navigate to the page to write
3      do {
4          page = follow_page_mask(vma, start, foll_flags, &page_mask);
5          // Return FALSE by requesting write to read-only file
6
7          if (!page) {
8              int ret;
9              ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);
10             ...
11         }
12         ...
13     }
14 }
```

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Vulnerable source code

- When writing through “/proc/self/mem”, `__get_user_pages()` is executed

```
1  __get_user_pages(){
2      //Navigate to the page to write
3      do {
4          FALSE page = follow_page_mask(vma, start, foll_flags, &page_mask);
5          // Return FALSE by requesting write to read-only file
6
7          if (!page) {
8              int ret;
9              ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);
10             ...
11         }
12         ...
13     }
14 }
```

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Vulnerable source code

- Even writes to Read-only files can be Copy On Write, thus the kernel executes `faultin_page`

```
1 faultin_page(){
2     ...
3     ret = handle_mm_fault(mm, vma, address, fault_flags);
4     // Function to check that it is a COW space
5     // VM_FAULT_WRITE = 1
6     ...
7     if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
8         *flags &= ~FOLL_WRITE;
9         // Because of COW, the kernel removed
10        // the flag to check write permission
11    return 0;
12 }
```

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Vulnerable source code

- Even writes to Read-only files can be Copy On Write, thus the kernel executes `faultin_page`

```
1 faultin_page(){
2     ... COW!
3     ret = handle_mm_fault(mm, vma, address, fault_flags);
4     // Function to check that it is a COW space
5     // VM_FAULT_WRITE = 1
6     ...
7     if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
8         *flags &= ~FOLL_WRITE;
9         // Because of COW, the kernel removed
10        // the flag to check write permission
11    return 0;
12 }
```

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - Vulnerable source code

```
1 faultin_page(){
2     ...
3     ret = handle_mm_fault(mm, vma, address, fault_flags);
4     // Function to check that it is a COW space
5     // VM_FAULT_WRITE = 1
6     ...
7     if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
8         *flags &= ~FOLL_WRITE;
9         // Because of COW, the kernel removed
10        // the flag to check write permission
11    return 0;
12 }
```

**MAVD\_DONTNEED!**

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**
  - Vulnerable source code

```
1  __get_user_pages(){
2      //Navigate to the page to write.
3      do {
4          page = follow_page_mask(vma, start, foll_flags, &page_mask);
5          // Function returns False (the mapped address is not found)
6                          
7          if (!page) {
8              int ret;
9              ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);
10             ...
11         }
12         ...
13     }
14 }
```



# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- Vulnerable source code

- It is judged to be a normal request because the flag to check write permission is missing

```
1 faultin_page(){
2     ...
3     ret = handle_mm_fault(mm, vma, address, fault_flags);
4     // Return the original file
5     ...
6     if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
7         *flags &= ~FOLL_WRITE;
8     return 0;
9 }
```

# Race condition

- **Real world example: Dirty COW (CVE-2016-5195)**

- A part of security patch

- Removed flag to check write permission

- + Add a flag indicating that the request is a COW area

```
@@ -412,7 +422,7 @@ static int faultin_page(struct task_struct *
    * reCOWed by userspace write).
    */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-       *flags &= ~FOLL_WRITE;
+       *flags |= FOLL_COW;
    return 0;
}
```

# Race condition

- **How can we prevent race condition vulnerabilities?**
  - Mutex (mutual exclusion)
  - Semaphore

# Race condition

- **Mutex**

- Algorithm used in concurrent programming to prevent race conditions regarding shareable resources
- Only one process (thread) can acquire the mutex and enter the critical section

# Race condition

- **Mutex**

```
1 mutex = 1;
2
3 void lock () {
4     while (mutex != 1) {
5         // Wait until mutex becomes 1
6     }
7     // Set the mutex to 0 to prevent other processes (threads) from accessing it
8     mutex = 0;
9 }
10
11 void unlock() {
12     // Release the lock to allow other processes access
13     mutex = 1;
14 }
```

# Race condition

- **Semaphore**

- Allow as many processes (or threads) as the number of semaphores to access shared resources

```
1 int S; // Semaphore
2
3 void P(S) {
4     S--;
5     if (S < 0) {
6         // add this process/thread to list
7         // block
8     }
9 }
10 void V(S) {
11     S++;
12     if (S <= 0) {
13         // remove a process p from list
14         // wakeup p
15     }
16 }
```

# Format string

- **Format string**

- A format generally used to accept input from users or output results

- E.g., `printf("%d", integer_value);`

- %d: Integer value (decimal)
- %f: float
- %c: character
- %s: string
- %p: void type pointer
- ...

# Format string

- **Format string attacks**
  - How is this used in attacks?

```
1 #include<stdio.h>
2
3 int main(int argc, char* argv[]){
4     printf ("%s\n", argv[1]);
5     printf (argv[1]);
6 }
```



# Format string

- **Format string attacks**

- (1) Terminating program**

- When we enter "%s", the program tries to read the address of the stack
      - Because the value on the stack is not the address value of the string, it is forced to terminate

```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./format "%s%s%s%s%s%s%s"
%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

# Format string

- **Format string attacks**

- (2) Process stack leak**

- By checking the stack, attackers can determine the memory structure of the process
      - Memory address on the stack at the moment of running this example

```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./format "%p %p %p %p %p"  
%p %p %p %p %p  
0x1 0x1 0x7fa5bd114887 (nil) 0x557b8e12d2a0
```

# Format string

- **Format string attacks**

- How can we prevent format string attacks?

1. Input validation
2. Function check
  - Fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, syslog, etc
3. The use of GCC options including “-Wformat” or FormatGuard

```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ gcc ./format.c -o format -Wformat
./format.c: In function 'main':
./format.c:5:5: warning: format not a string literal and no format arguments [-Wformat-security]
   5 |     printf (argv[1]);
     |           ^~~~~~
```

# Next Lecture

- **Other software vulnerabilities**
  - E.g., integer overflow, command injection