VI. Synchronization and Race Conditions Vulnerabilities

Race condition vulnerability - Description

- **race conditions**
  - conditions / context that allow
  - **uncontrolled**, undetermined, undesired **interference of** different **concurrent actions** (i.e. threads, processes)
  - accessing **shared resources**,
  - which lead to
    - -> **unexpected**, undesired **results**
    - -> inconsistent, **corrupted state of the resources**
- the vulnerability consists in
  - **not taking into account** possible **race conditions**
  - **not (correctly) protecting** the shared resources, i.e. not synchronizing concurrent executions
  - i.e. **not assuring atomicity** of more (logically) related steps
- **language independent**

Race condition vulnerability - Types

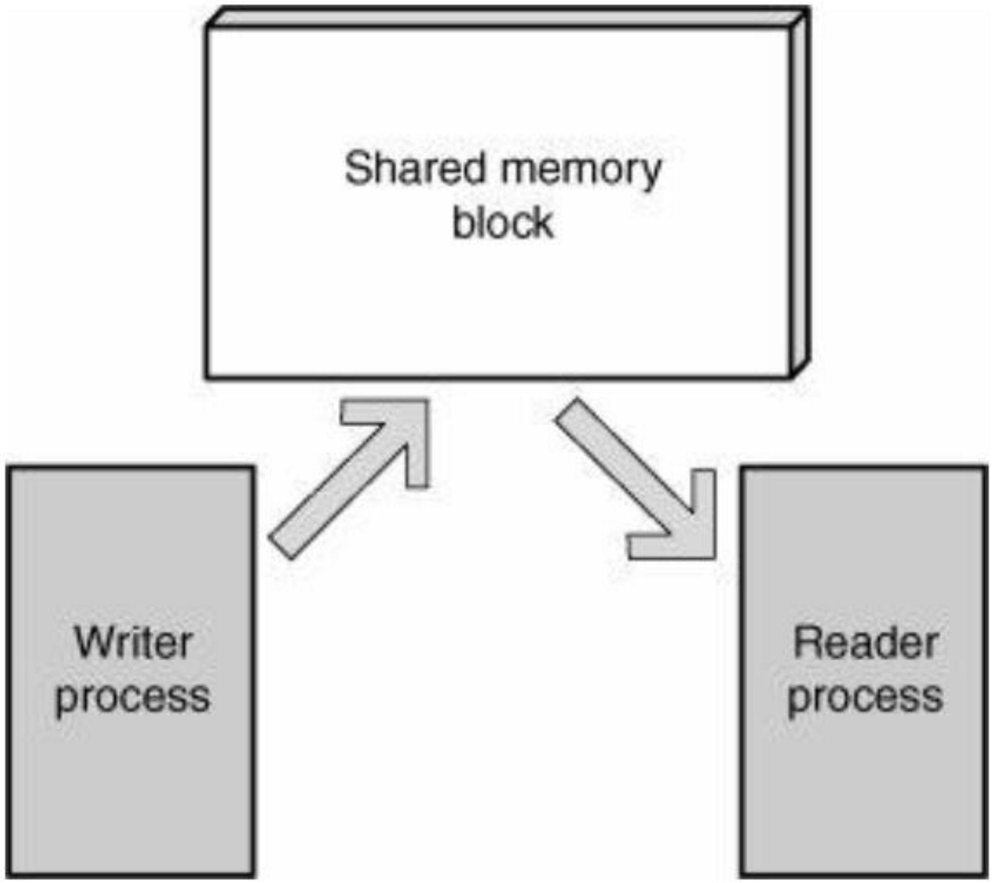
- determined by the ways the interfering attacker's code sequence could be triggered

1. **trusted**: **internal to the application**, e.g. an application's thread
    - cannot be modified by the attacker
    - can only be invoked indirectly
2. **untrusted**: **external to the application**
    - can be authored directly by the attacker
    - based on an environment's component controlled by the attacker

Race condition vulnerability – Attacks and Effects

- **can have security implications** when the expected synchronization is in security-critical code
  - e.g. recording whether a user is authenticated
  - e.g. modifying important state information that should not be influenced by an outsider
- **the attacker**
  - (could) **trigger an action concurrent with other application's actions**
  - due to race conditions could gain advantages over that application
- possible security effects
  - crash the application, i.e. **denial of service** (DoS), affects **availability**
  - **information leakage**, affects **confidentiality**
  - **data corruption**, affects **integrity**
  - **privilege escalation**

Synchronization problems - Atomicity



Reentrancy and asynchronous-safe code

- Reentrancy – function's capability to work correctly even when it is interrupted by another running thread that calls the same function
- i.e. multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states

Reentrancy and asynchronous-safe code (II)

```
struct list *global_list;
int global_list_count;

int list_add(struct list *element) {
  struct list *tmp;
  if(global_list_count > MAX_ENTRIES)
    return -1;
  for(list = global_list; list->next; list = list->next);
  list->next = element;
  element->next = NULL;
  global_list_count++;
  return 0;
}
```

Reentrancy and asynchronous-safe code (III)

```
struct CONNECTION {
  int sock;
  unsigned char * buffer;
  size_t bytes_available, bytes_allocated;
}
client;
size_t bytes_available(void) {
  return client -> bytes_available;
}
int retrieve_data(char * buffer, size_t length)
    {
  if (length < bytes_available()) memcpy(buffer
      , client -> buffer, length);
  else
    memcpy(buffer, client -> buffer,
        bytes_available());
  return 0;
}
```

Race conditions

```
struct element *queue;
int queueThread(void) {
struct element *new_obj, *tmp;
for(;;) {
wait_for_request(); new_obj = get_request();
       if(queue == NULL)
       {
          queue = new_obj;
continue; }
for(tmp = queue; tmp->next; tmp = tmp->next) ;
       tmp->next = new_obj;
   }
}
int dequeueThread(void) {
for(;;) {
       struct element *elem;
       if(queue == NULL)
          continue;
       elem = queue;
       queue = queue->next;
       .. process element ..
   }
}
```

Starvation and Deadlocks

```
Int thread1(void)
{
   lock(mutex1);
.. code ..
   lock(mutex2);
   .. more code ..
   unlock(mutex2);
   unlock(mutex1);
return 0; }
```

```
int thread2(void)
{
   lock(mutex2);
.. code ..
   lock(mutex1);
   .. more code ..
   unlock(mutex2);
   unlock(mutex1);
return 0; }
```

Race condition vulnerability – CWE References

- CWE-361: “Time and State”
  - improper management of time and state in an environment that supports simultaneous or near-simultaneous computation
- CWE-691: “Insufficient Control Flow Management”
  - code does not sufficiently manage its control flow during execution,
  - creating conditions in which the control flow can be modified in unexpected ways
- CWE-364: “Signal Handler Race Condition”
  - software uses a signal handler that introduces a race condition

Race condition vulnerability - CWE References (2)

- **CWE-362**: “Concurrent Execution using Shared Resource with Improper Synchronization (**Race Conditions**)”
  - a code sequence that can run concurrently with other code, and
  - the code sequence requires temporary, exclusive access to a shared resource, but
  - a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently

```c
void f(pthread_mutex_t *mutex)
{
    pthread_mutex_lock(mutex);

    /*access shared resource */

    pthread_mutex_unlock(mutex);
}
```

```c
int f(pthread_mutex_t *mutex)
{
    int result;

    result = pthread_mutex_lock(mutex);
    if (0 != result)
        return result;

    /*access shared resource */

    return pthread_mutex_unlock(mutex);
}
```

Race condition vulnerability - CWE References (3)

```c
#include <sys/types.h>
#include <sys/stat.h>

int main(argc, argv)
{
    struct stat * sb;
    time_t timer;
    lstat("bar.sh", sb);
    printf("%d\n", sb->st_ctime);
    switch (sb->st_ctime % 2)
    {
        case 0:
            printf("One option\n");
            break;
        case 1:
            printf("another option\n");
            break;
        default:
            printf("huh\n");
            break;
    }
    return 0;
}
```

CWE-365: “Race Condition in Switch”

code contains a switch statement in which the switched variable can be modified while the switch is still executing, resulting in unexpected behavior

Race condition vulnerability - CWE References (4)

- CWE-366: “Race Condition within a Thread”
  - if two threads of execution use a resource simultaneously,
  - there exists the possibility that resources may be used while invalid
  - making the state of execution undefined

```
int foo = 0;
int storenum(int num)
{
    static int counter = 0;
    counter++;
    if (num > foo) foo = num;
    return foo;
}
```

Race condition vulnerability - CWE References (5)

- CWE-367: “Time-of-check Time-of-use (TOCTOU)”
  - software checks the state of a resource before using it, but
  - the resource’s state can change between the check and the use in a way that invalidates the results of the check

```
struct stat * sb;
...
// it has not been updated since the last time it was read
lstat("...", sb);
printf("stated file\n");
if (sb->st_mtimespec == ...)
{
    print("Now updating things\n");
    updateThings();
}
```

Race condition vulnerability - CWE References (6)

- CWE-368: “Context Switching Race Condition”
  - performs a series of non-atomic actions to switch between contexts that cross privilege or other security boundaries, but
  - a race condition allows an attacker to modify or misrepresent the product’s behavior during the switch
  - e.g. while a Web browser is transitioning from a trusted to an untrusted domain, an attacker can perform certain actions
- CWE-421: “Race Condition During Access to Alternate Channel”
  - open a channel to communicate with an authorized user, but
  - the channel is accessible to other actors before the authorized users

Race condition vulnerability – (some) vulnerability faces

- unsynchronized (or wrongly synchronized) code
- wrong handling of UNIX signals
- interactions with the file system
- time of check to time of use (TOCTOU)

Race condition vulnerability – related vulnerabilities

- not using proper access control
  - gives the attacker the possibility to interfere with the application
- unfounded trust in application's environment
- generating bad random numbers
  - used for creating files with unpredicted names in public area
  - Let the attacker guess names of the files

Race condition vulnerability – identify the vulnerability

- identify shared resources (between threads or processes)
  - determine if they can be accessed (read, written) concurrently
- identify creation of files (objects) in publicly accessible areas
  - determine possible concurrent external actions
- check for signal handling
- identify non-reentrant functions in multithreaded applications or signal handlers
  - working with global or local static variables

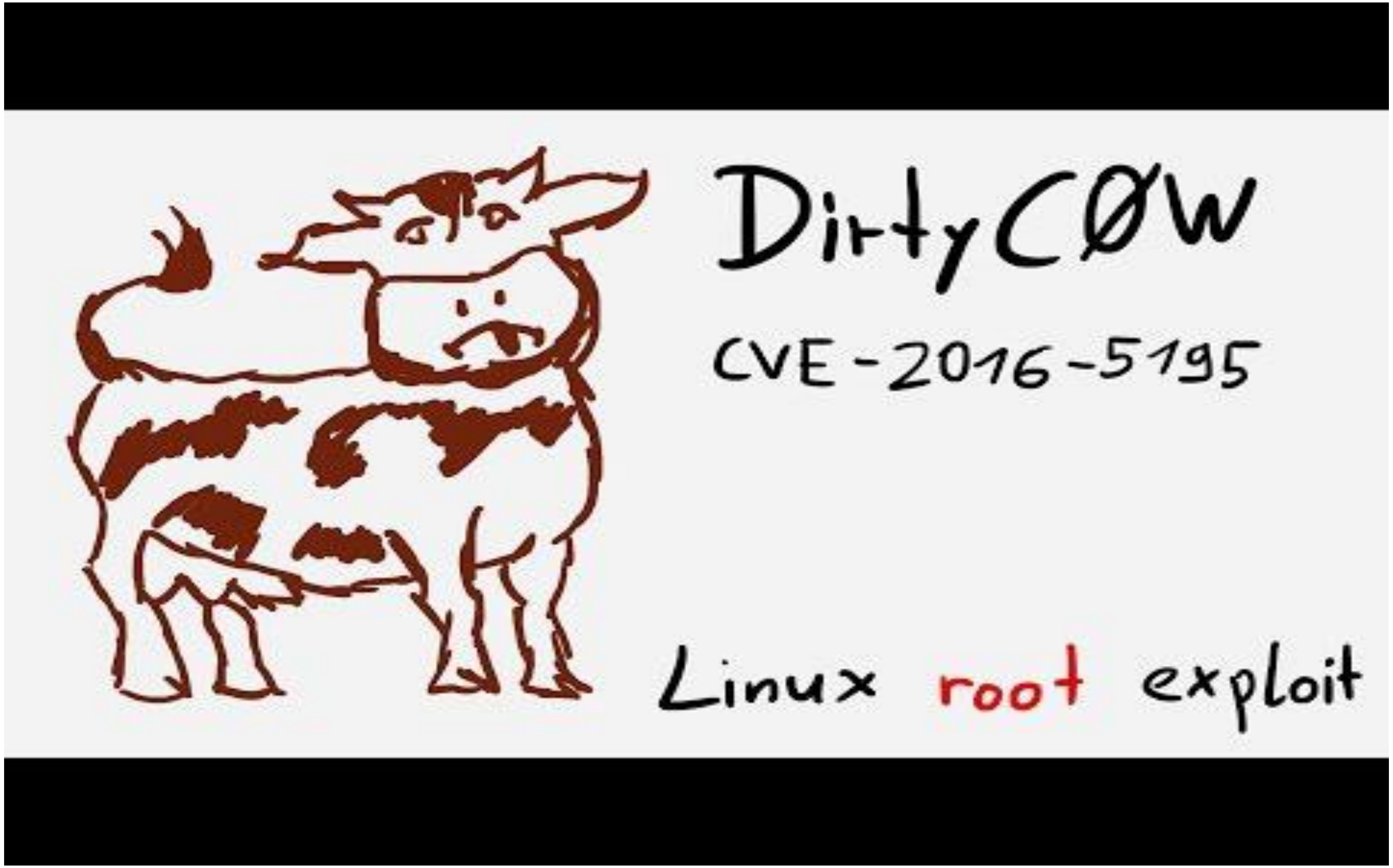
Race condition vulnerability – redemption steps

- understand how to correctly write reentrant code
- understand how to correctly use synchronization mechanisms
- make safe operations in signal handlers
- avoid TOCTOU operations

Race condition vulnerability – detection methods

- black box testing
- white box testing
- automated dynamic analysis
- automated static analysis
- manual code review
- formal methods

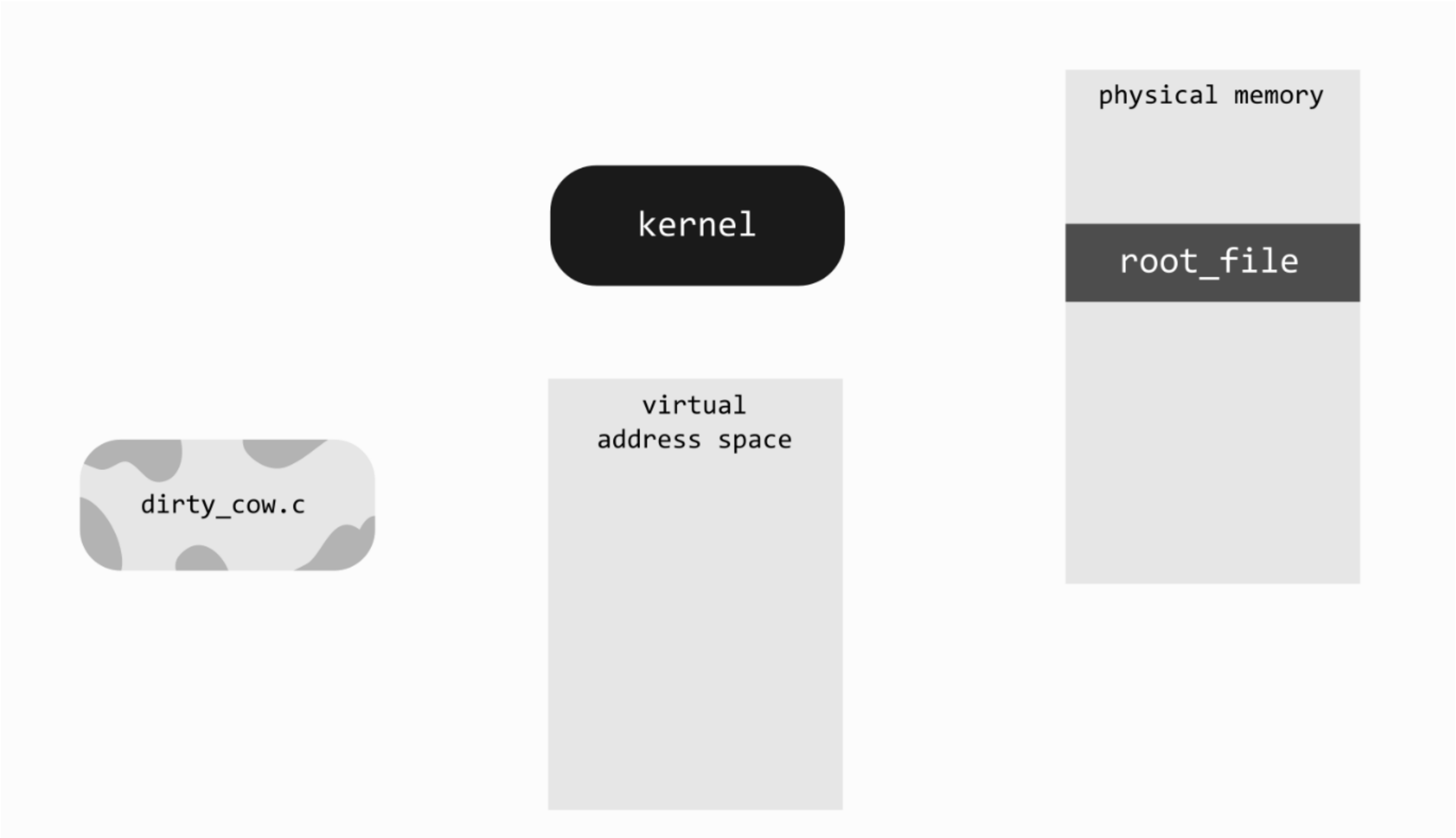
Race condition vulnerability – recent vulnerability: dirty COW



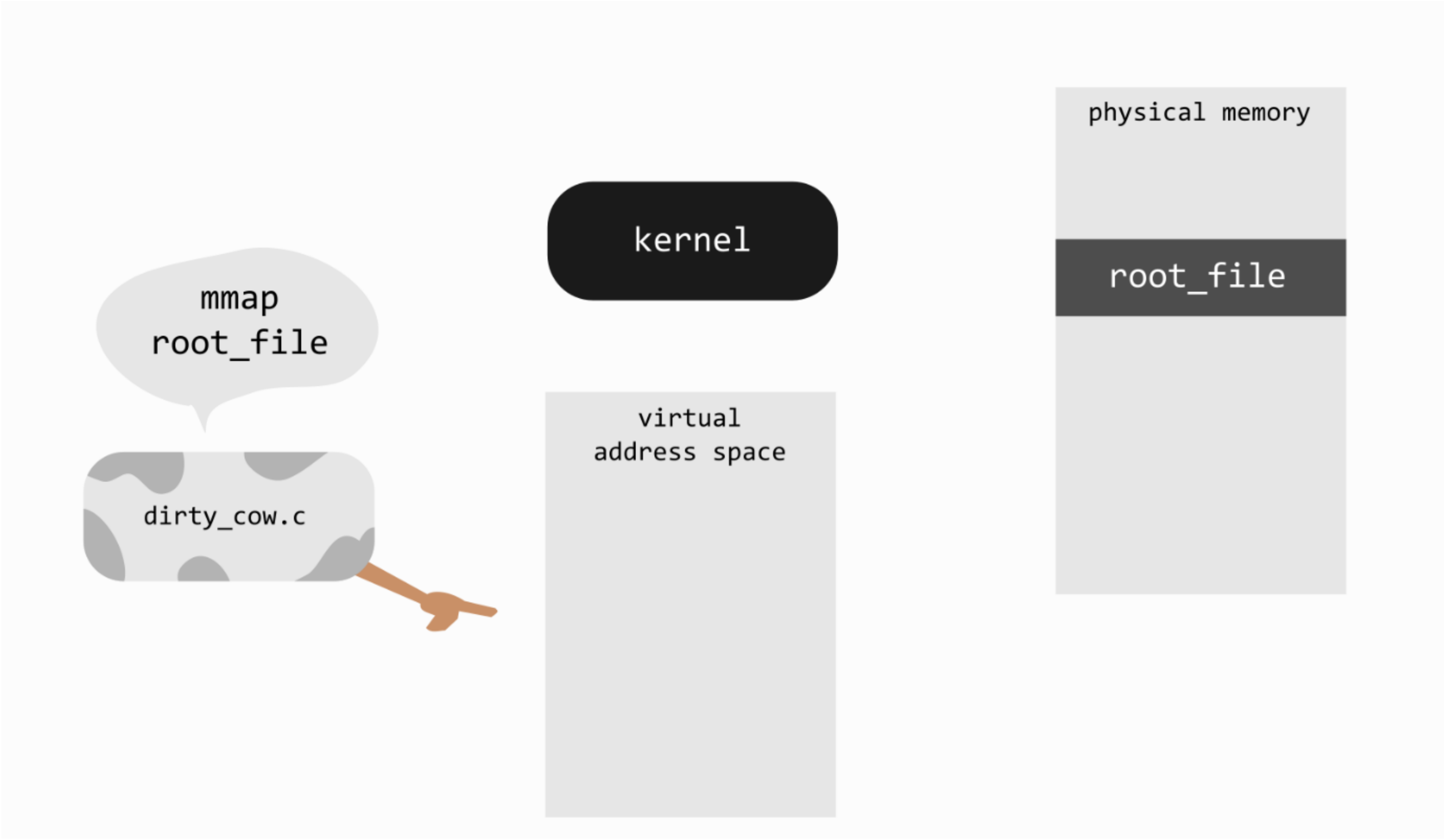
Race condition vulnerability – recent vulnerability: dirty COW (2)

- CVE-2016-5195: **Dirty COW** (i.e. COW = copy-on-write)
  - see https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195
  - see https://dirtycow.ninja/
  - published on 2016-11-10
- allow for **Linux kernel privilege escalation** vulnerability
- due to **a race condition** in Linux kernel's memory subsystem
  - incorrect handling of a copy-on-write (COW) feature
  - to write to a private read-only memory mapping
- explained exploit
  - https://www.youtube.com/watch?v=kEsshExn7aE
  - https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/demo.html

dirty COW - 1

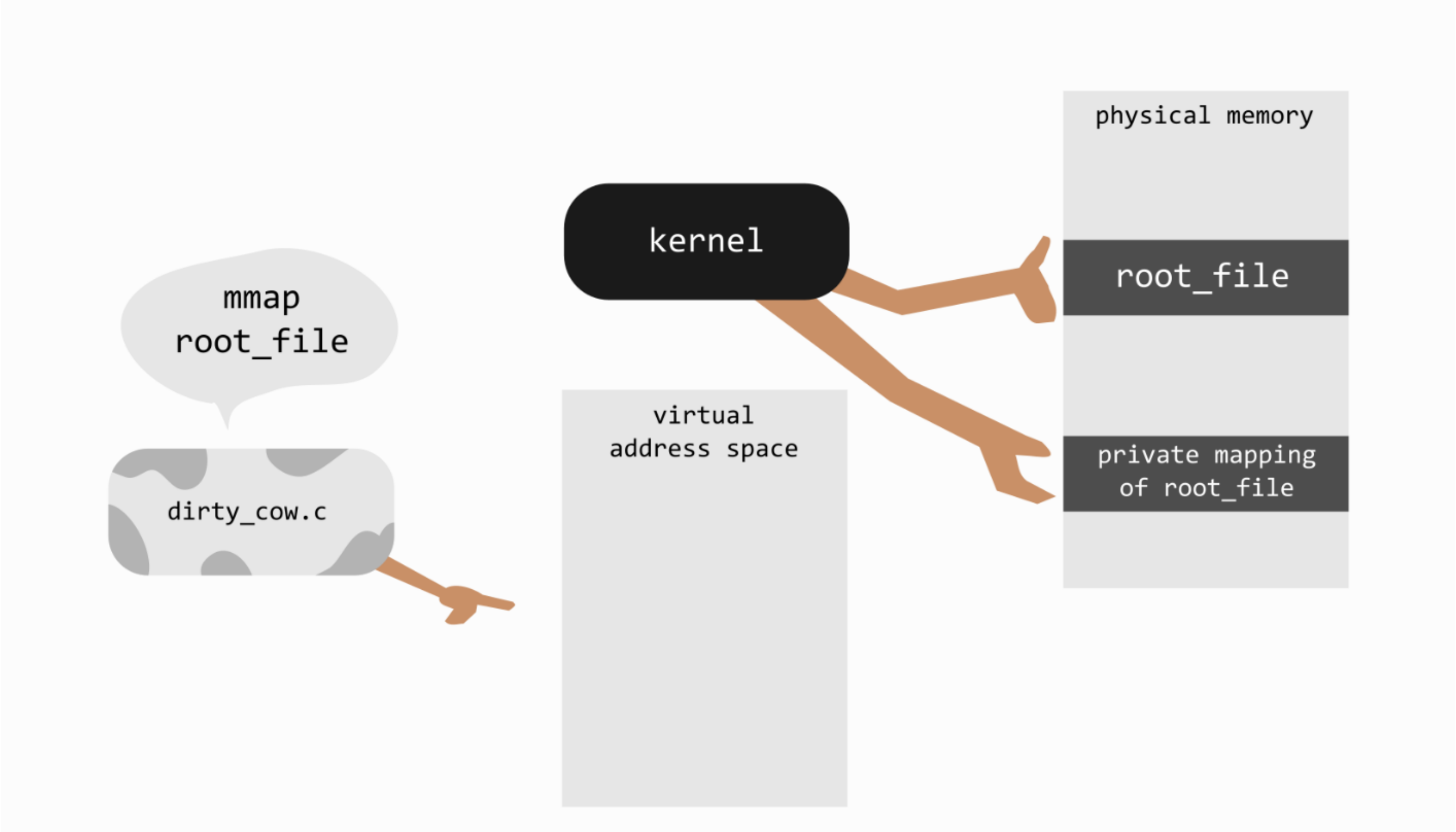

physical memory
kernel
root_file
virtual
address space
dirty_cow.c

dirty COW - 2



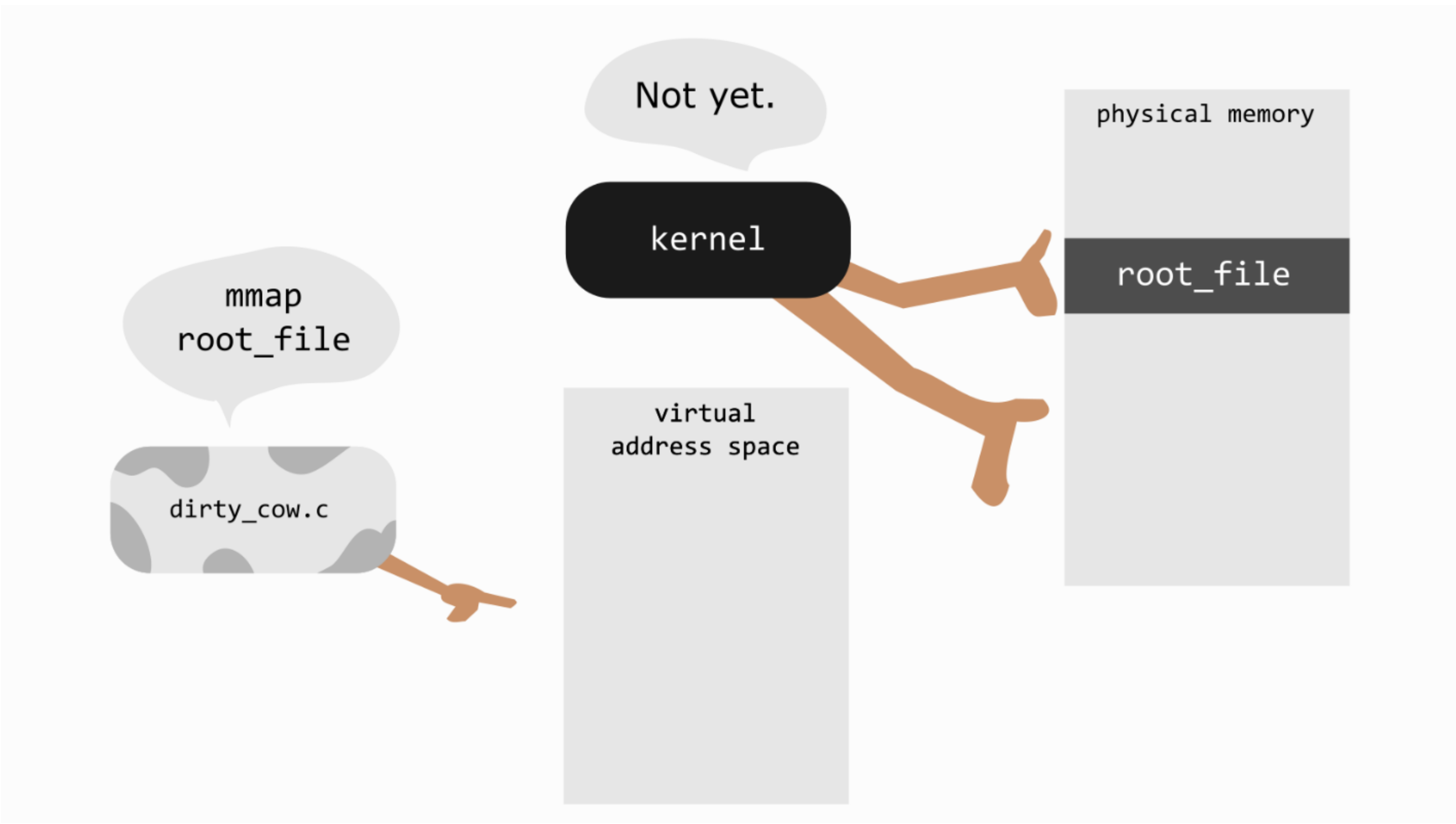
physical memory
kernel
root_file
mmap
root_file
virtual
address space
dirty_cow.c

dirty COW - 3

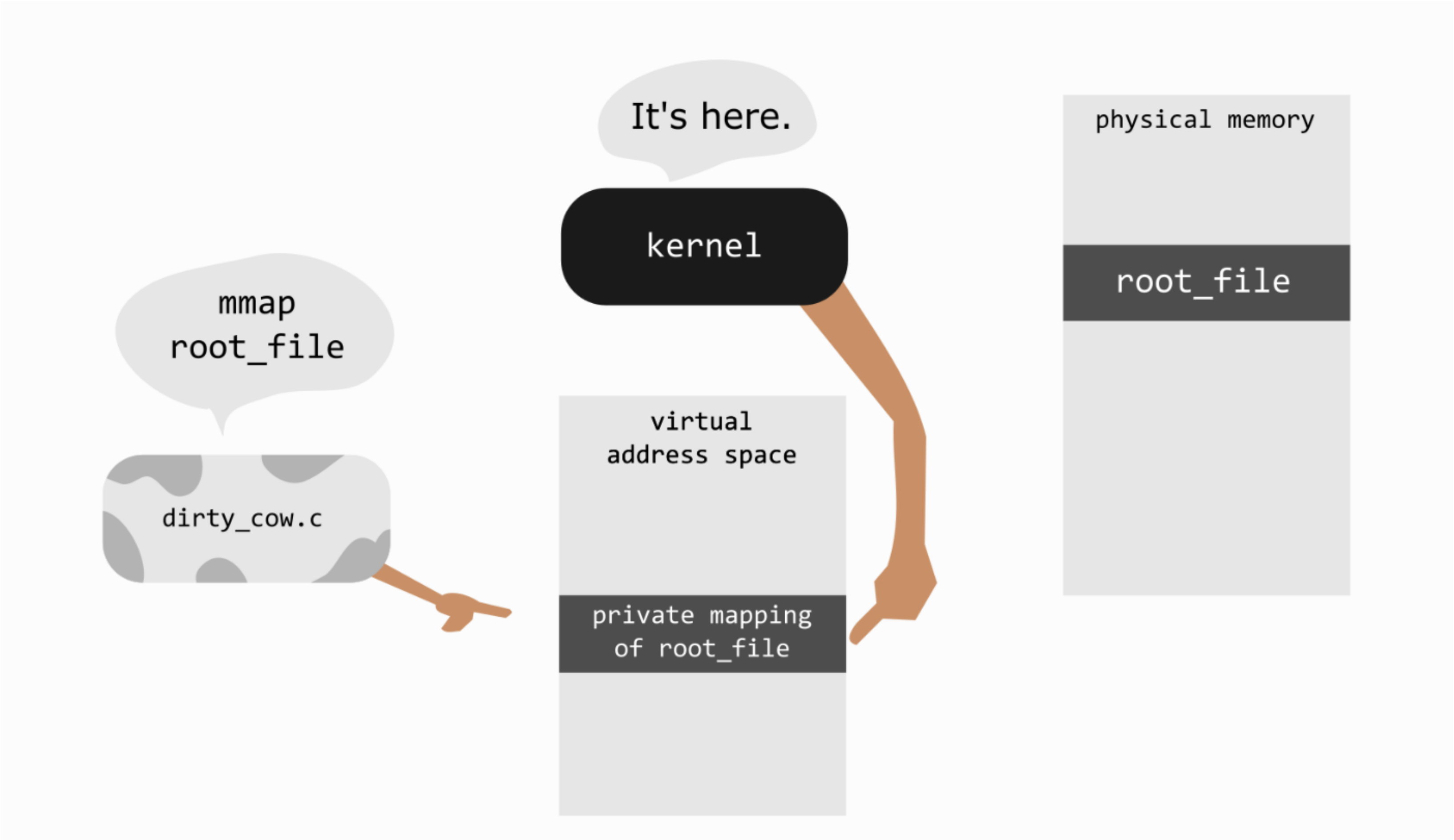


physical memory
kernel
root_file
mmap
root_file
virtual
address space
private mapping
of root_file
dirty_cow.c

dirty COW – 4

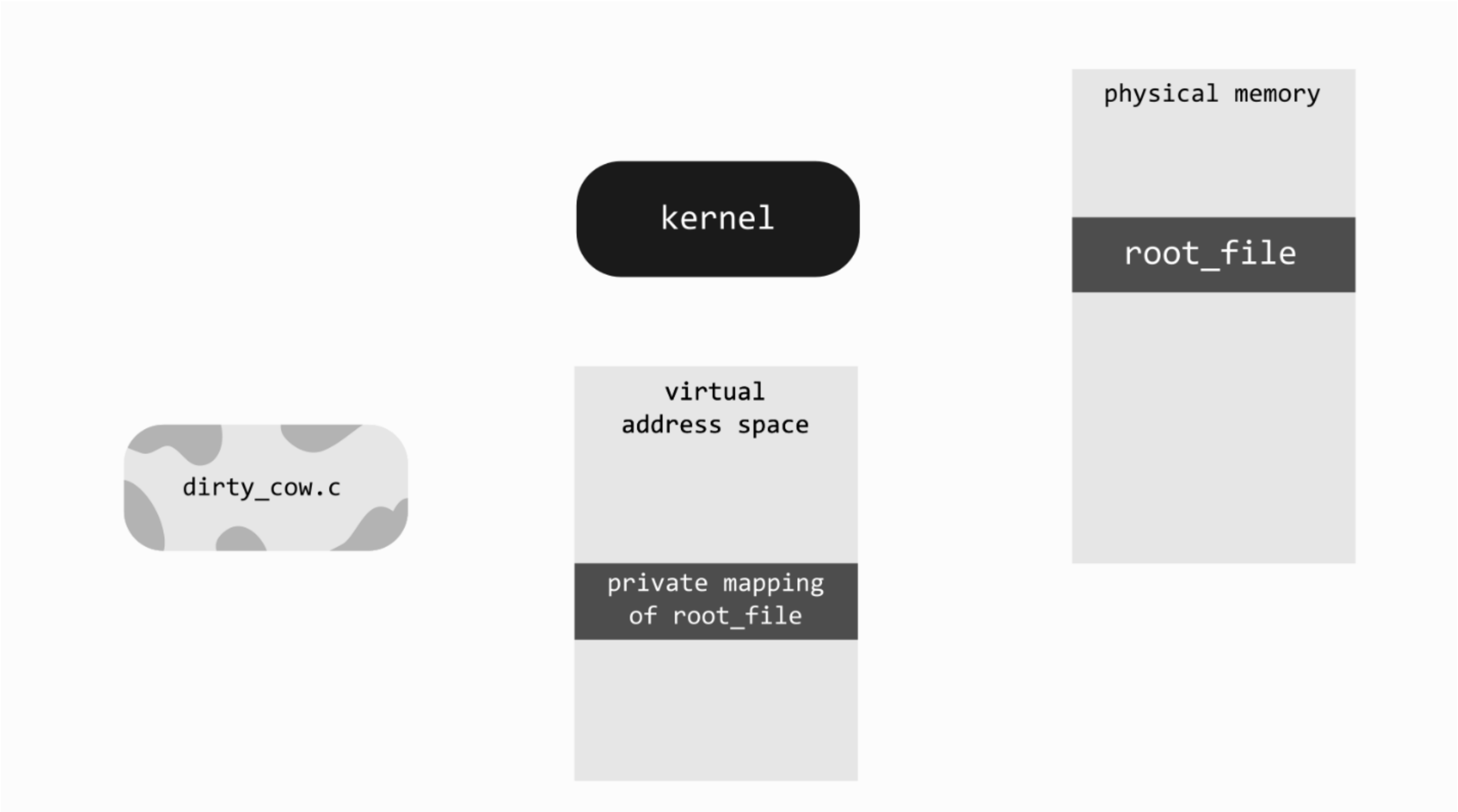

Not yet.
physical memory
kernel
root_file
mmap
root_file
virtual
address space
dirty_cow.c

dirty COW - 5



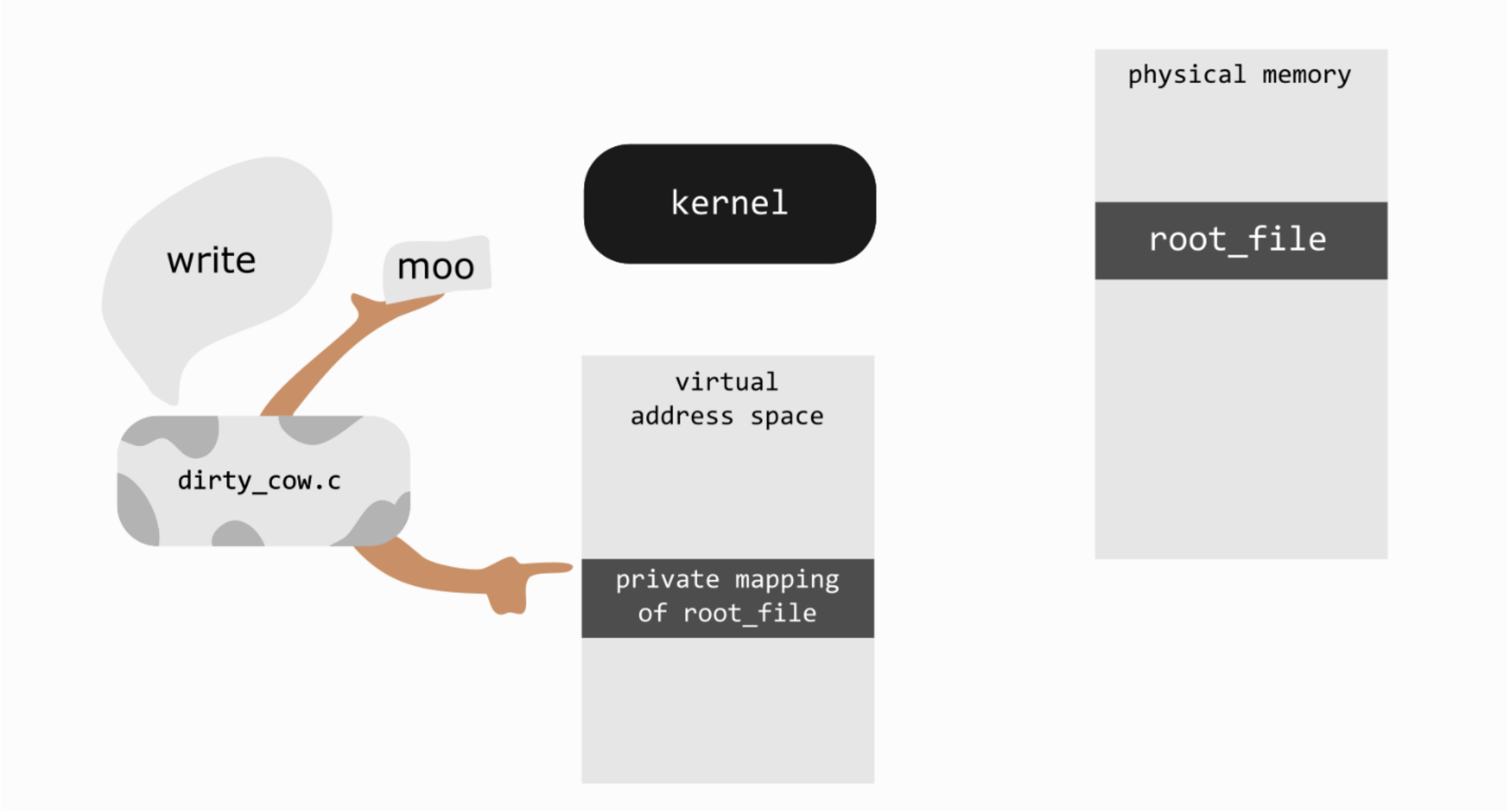
It's here.
kernel
physical memory
root_file
mmap
root_file
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 6



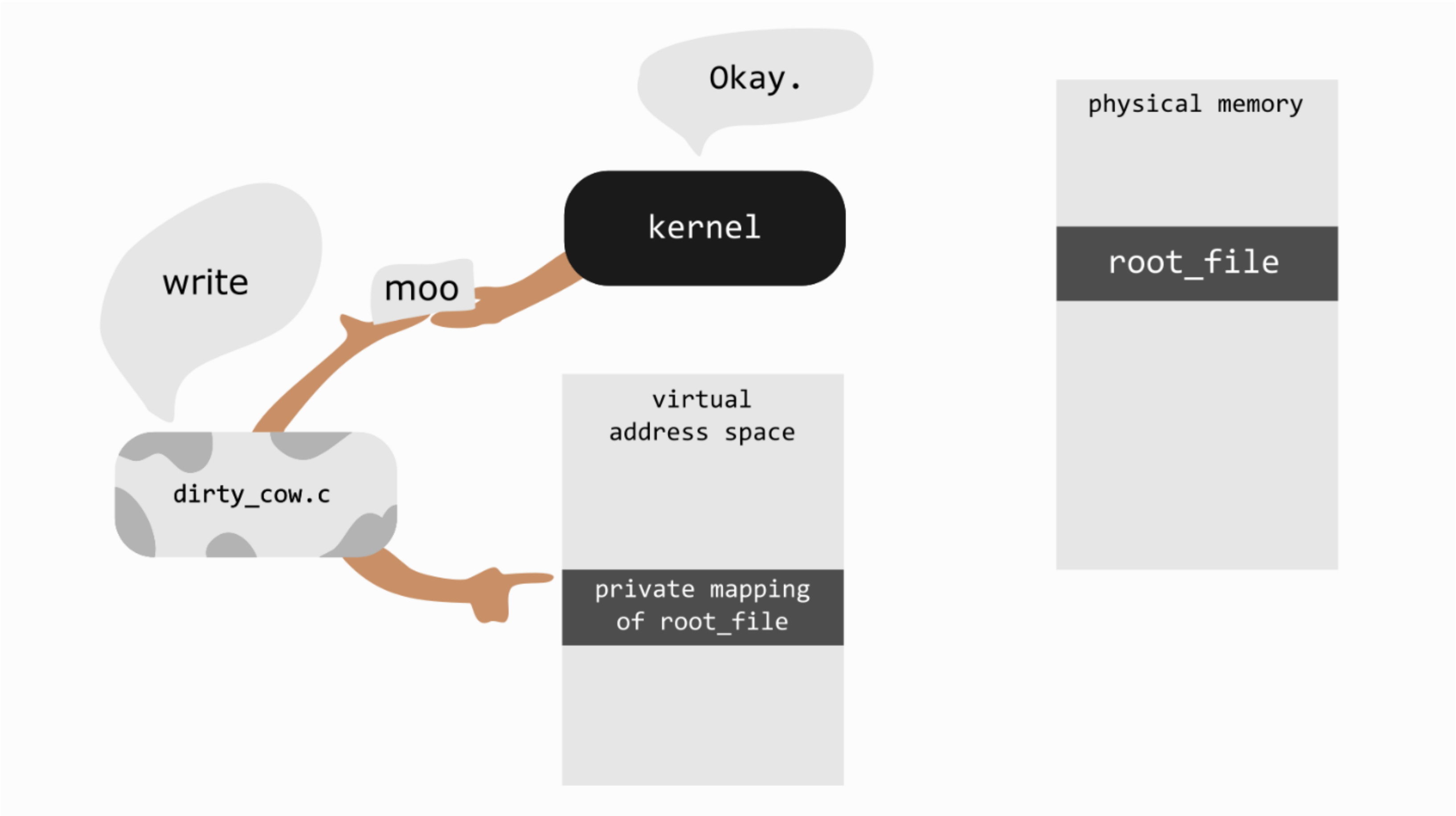
physical memory
kernel
root_file
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 7



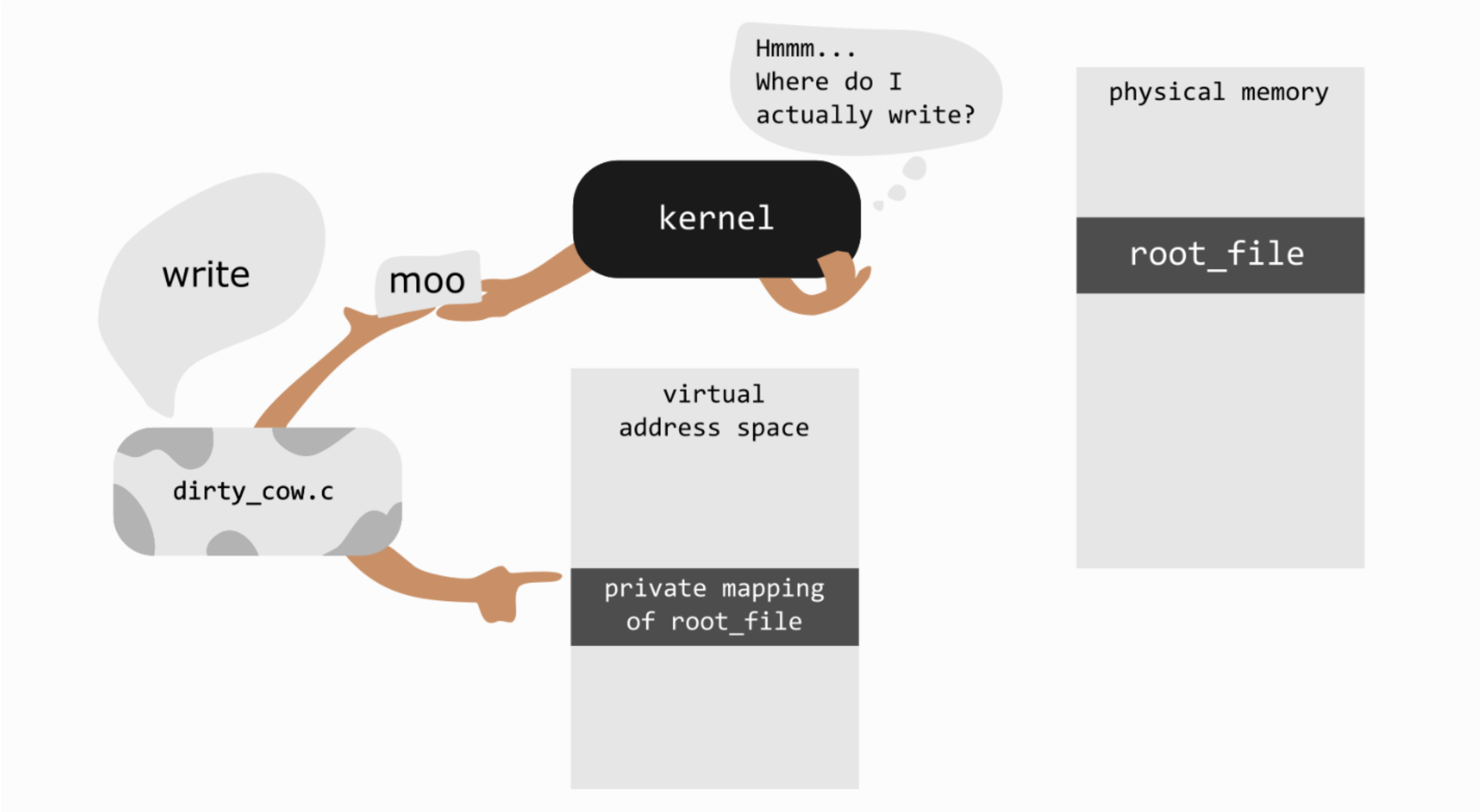
write
moo
dirty_cow.c
kernel
virtual
address space
private mapping
of root_file
physical memory
root_file

dirty COW - 8

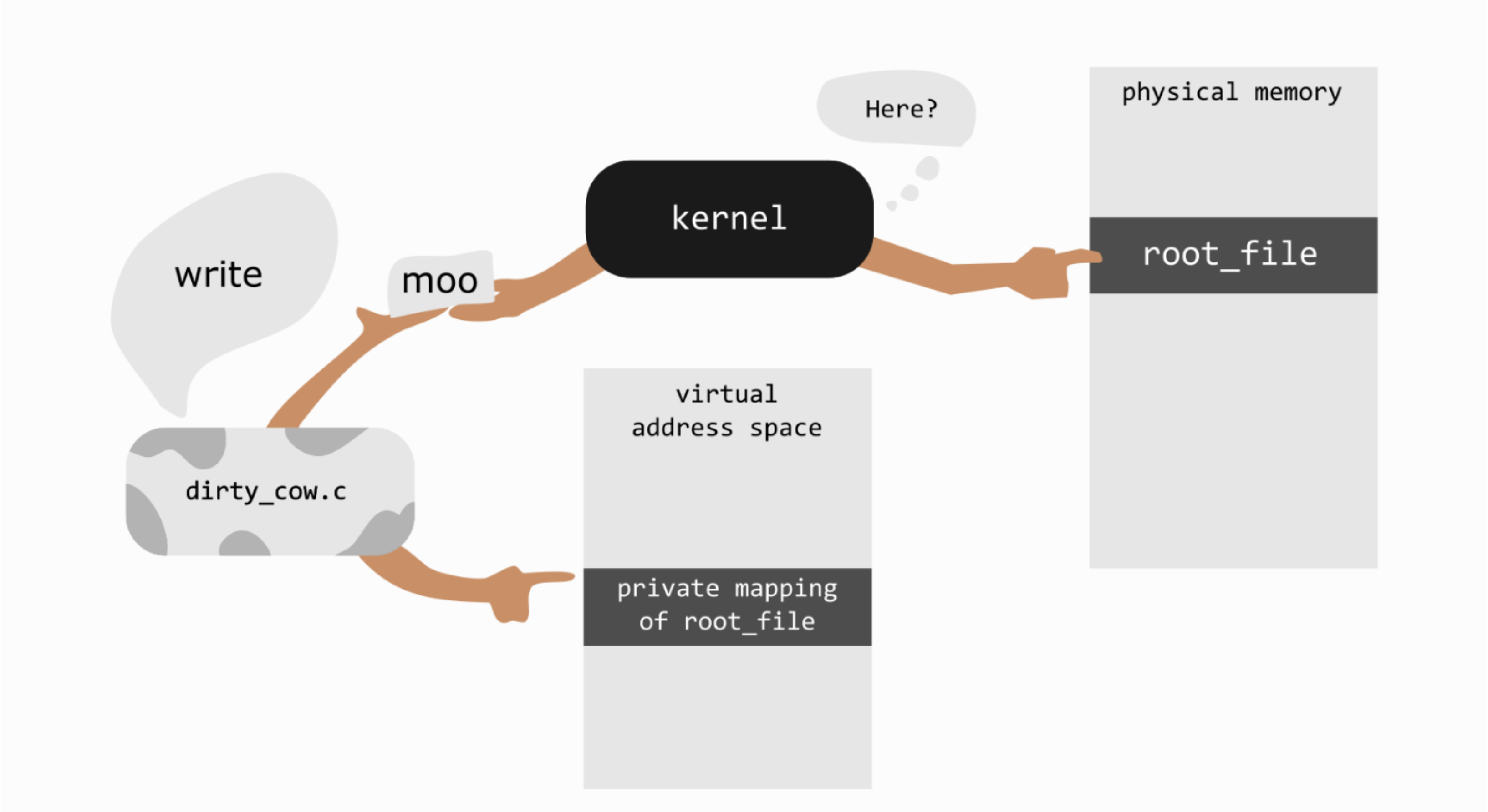


Okay.
physical memory
kernel
root_file
write
moo
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 9

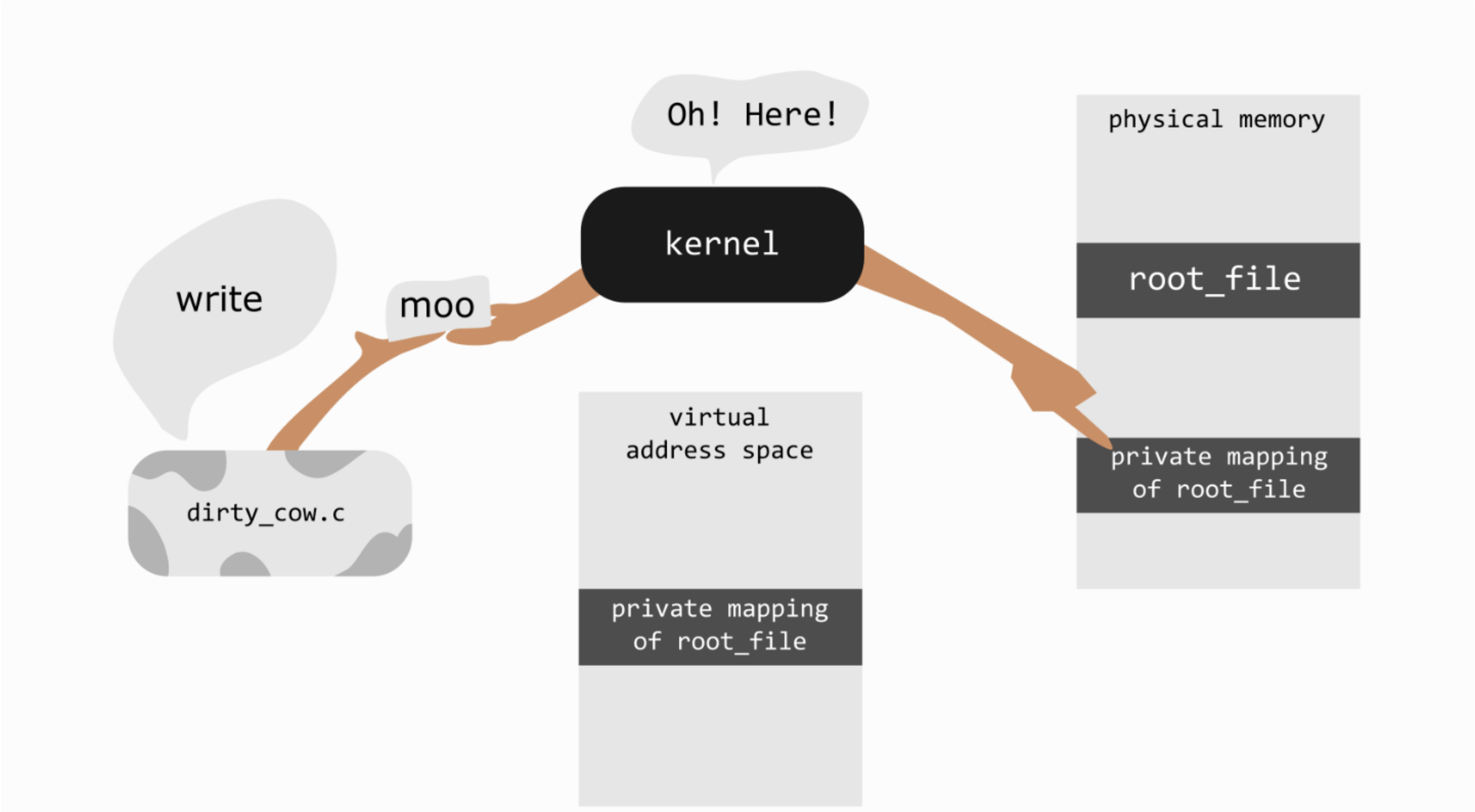

Hmmm...
Where do I
actually write?
physical memory
kernel
root_file
write
moo
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 10



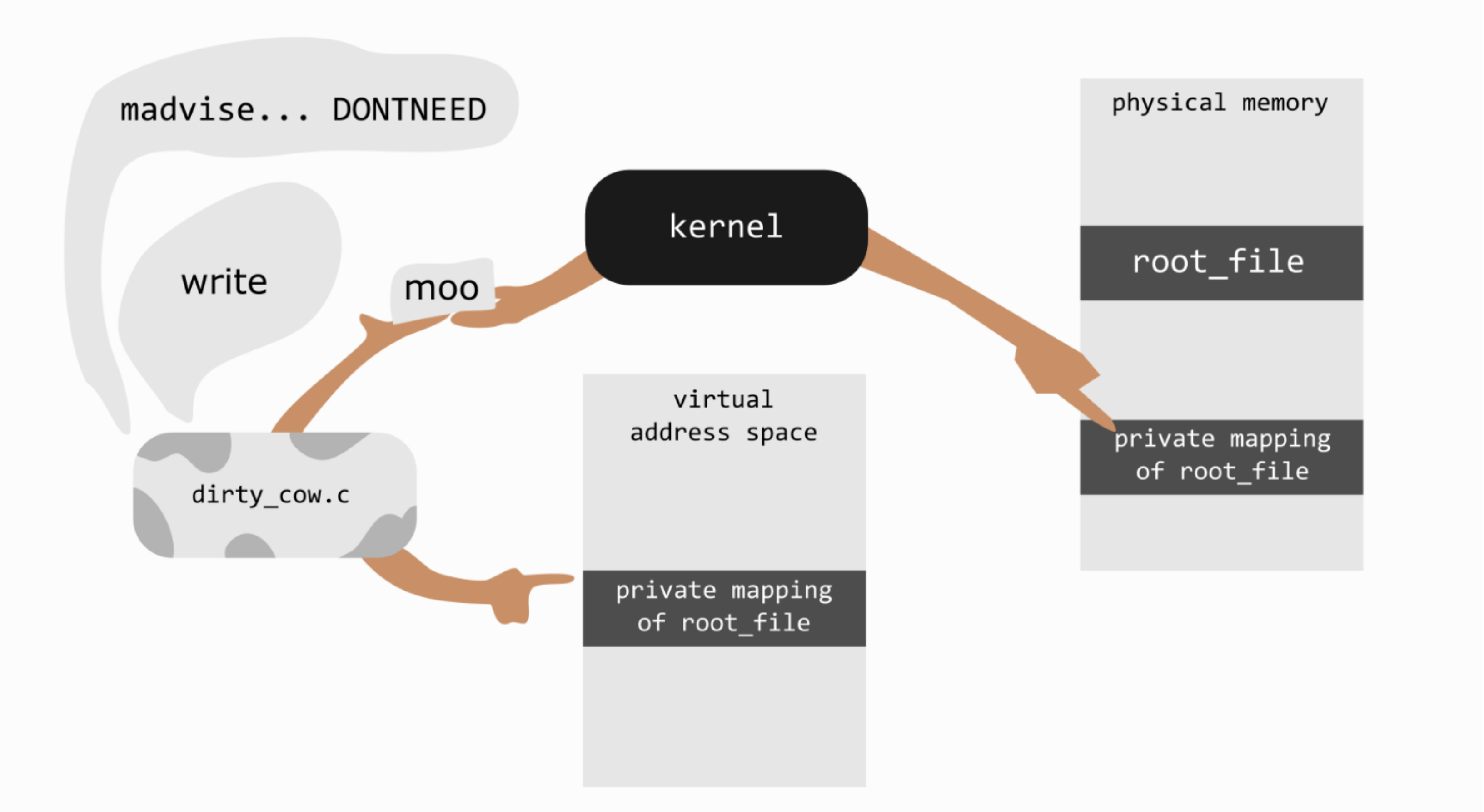
Here?
physical memory
kernel
write
moo
root_file
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 11



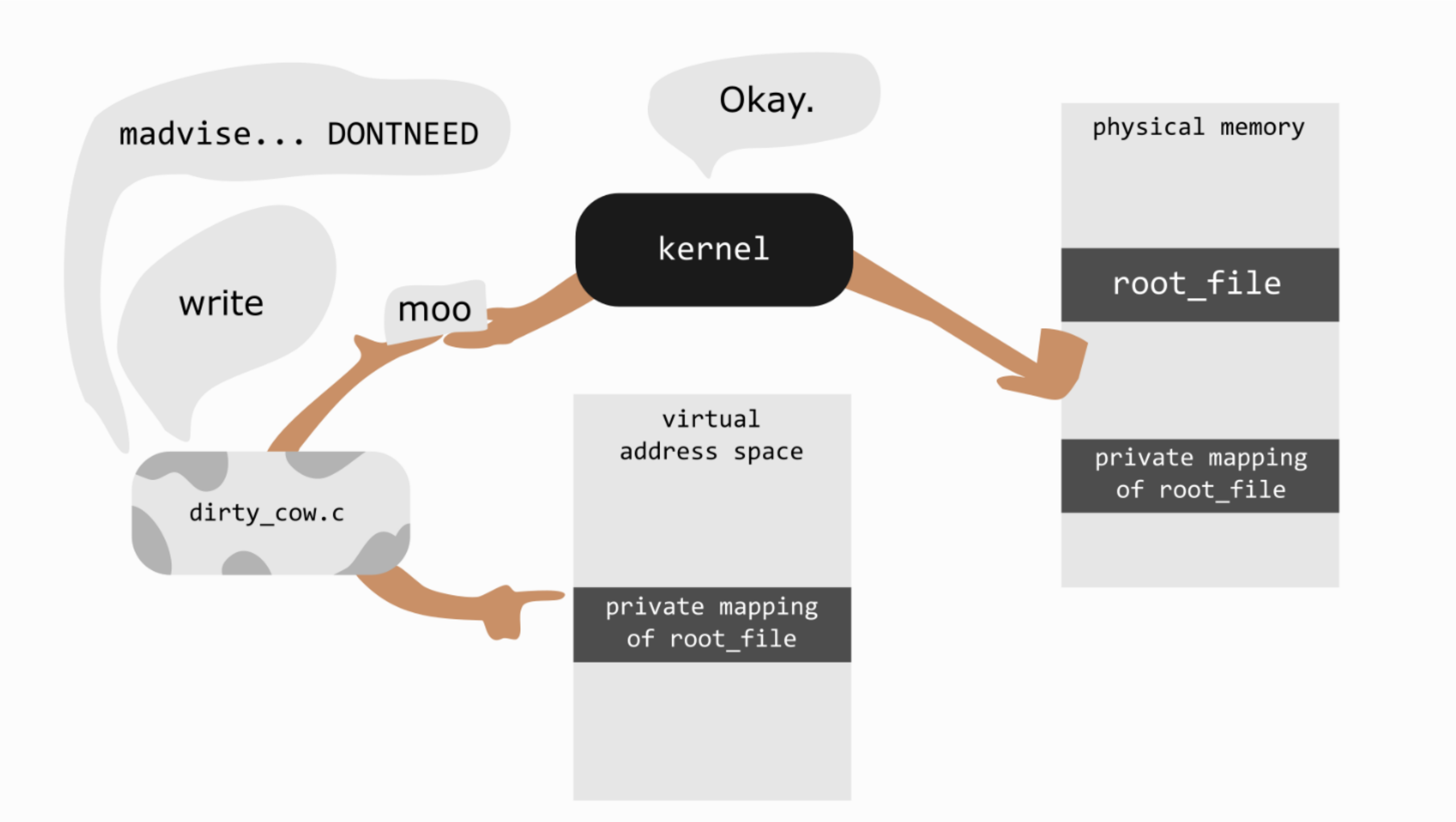
Oh! Here!
write
moo
kernel
dirty_cow.c
virtual
address space
private mapping
of root_file
physical memory
root_file
private mapping
of root_file

dirty COW - 12



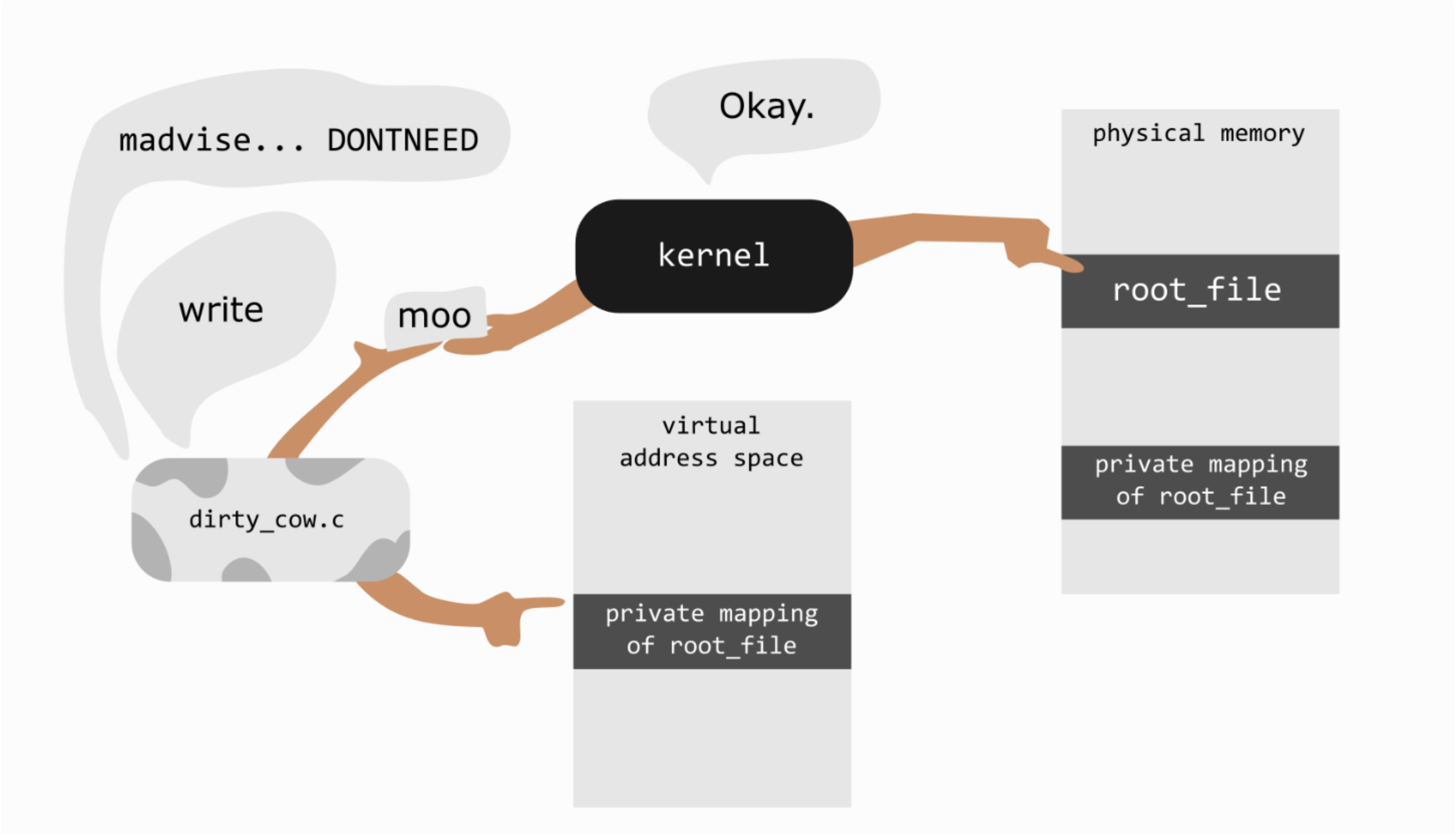
madvise... DONTNEED
write
moo
kernel
dirty_cow.c
virtual
address space
private mapping
of root_file
physical memory
root_file
private mapping
of root_file

dirty COW - 13



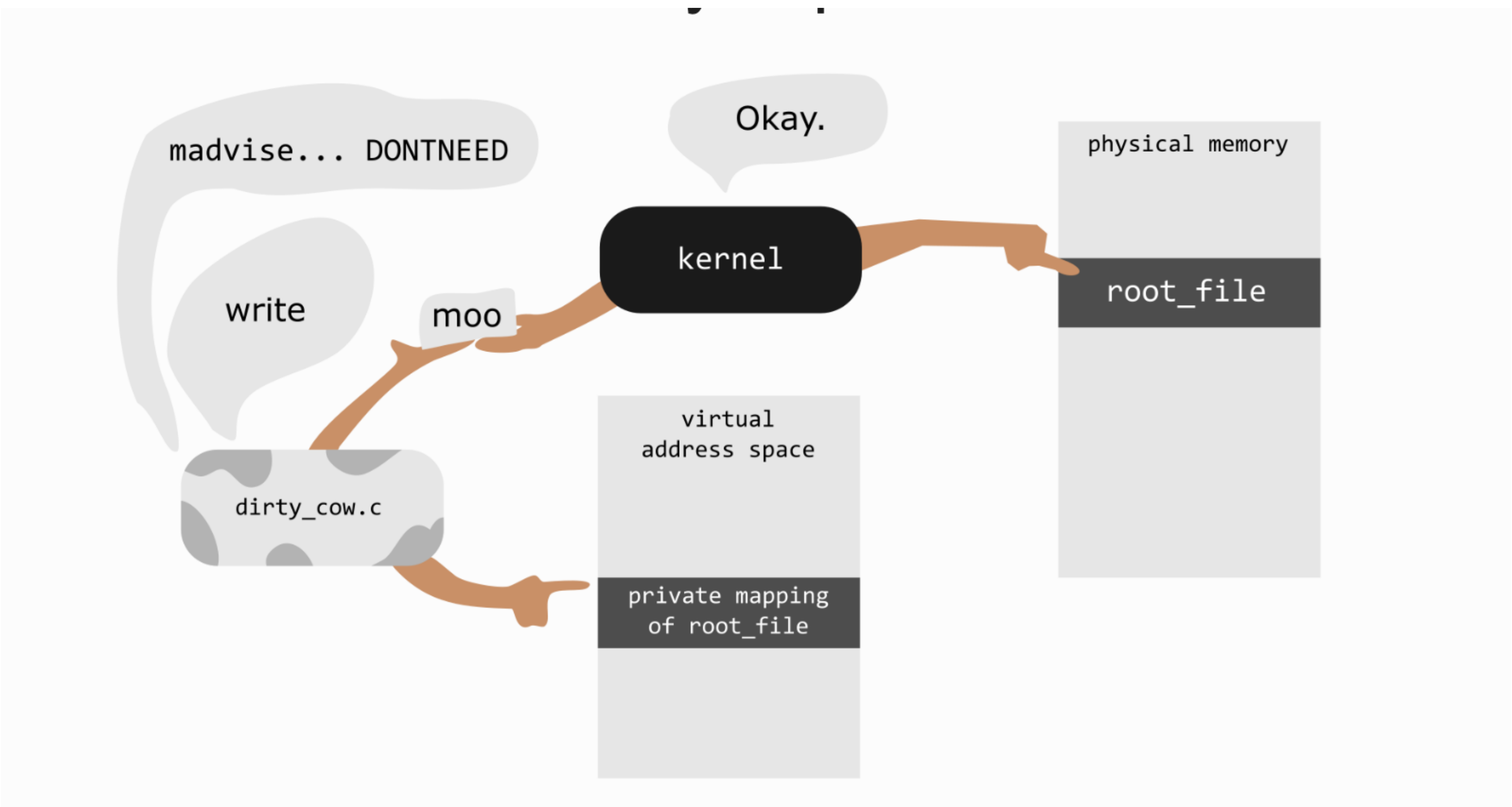
madvise... DONTNEED
Okay.
physical memory
kernel
root_file
write
moo
virtual
address space
private mapping
of root_file
dirty_cow.c
private mapping
of root_file

dirty COW - 14



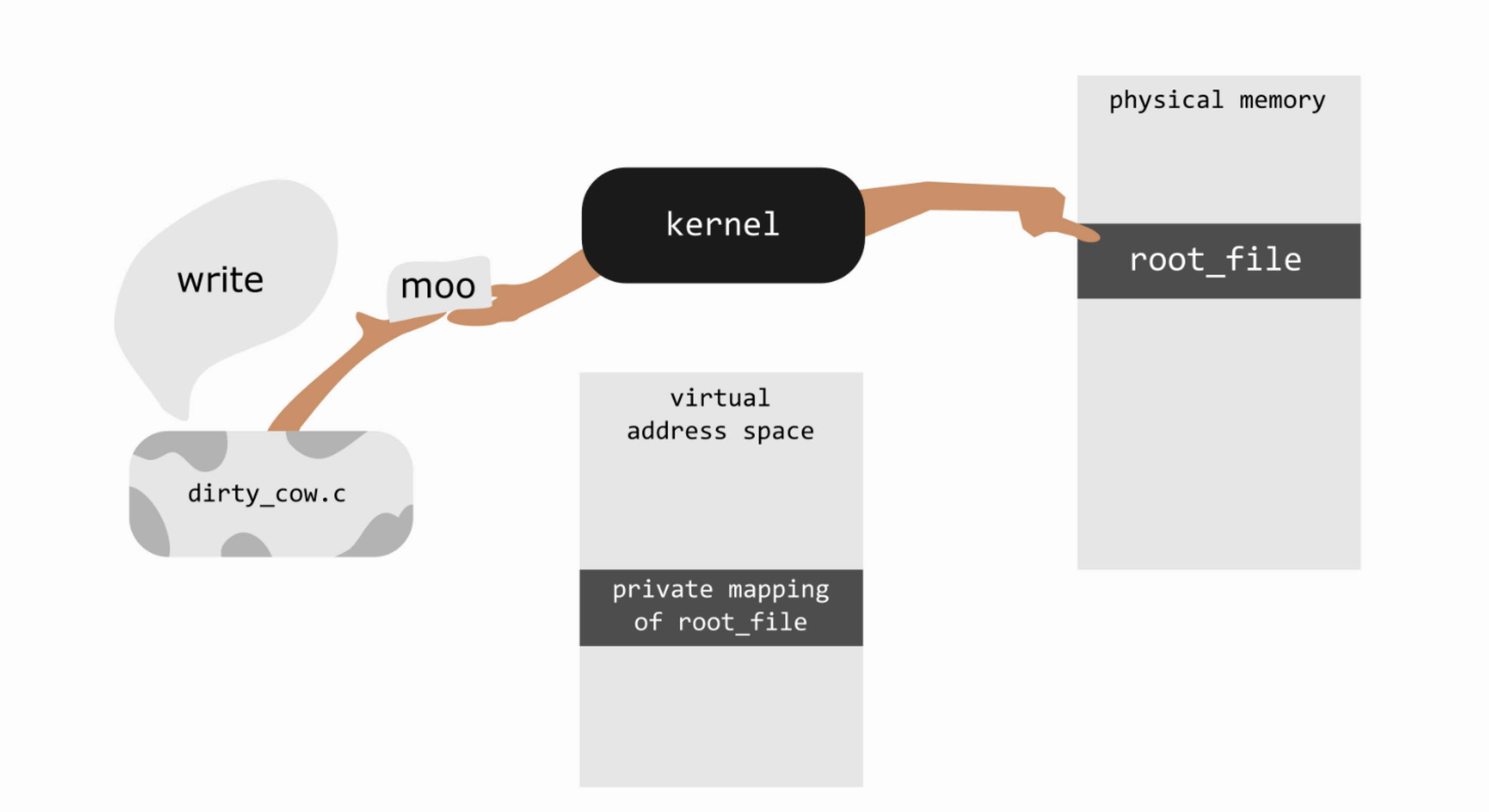
madvise... DONTNEED
Okay.
physical memory
write
kernel
root_file
moo
virtual address space
private mapping of root_file
dirty_cow.c
private mapping of root_file

dirty COW - 15



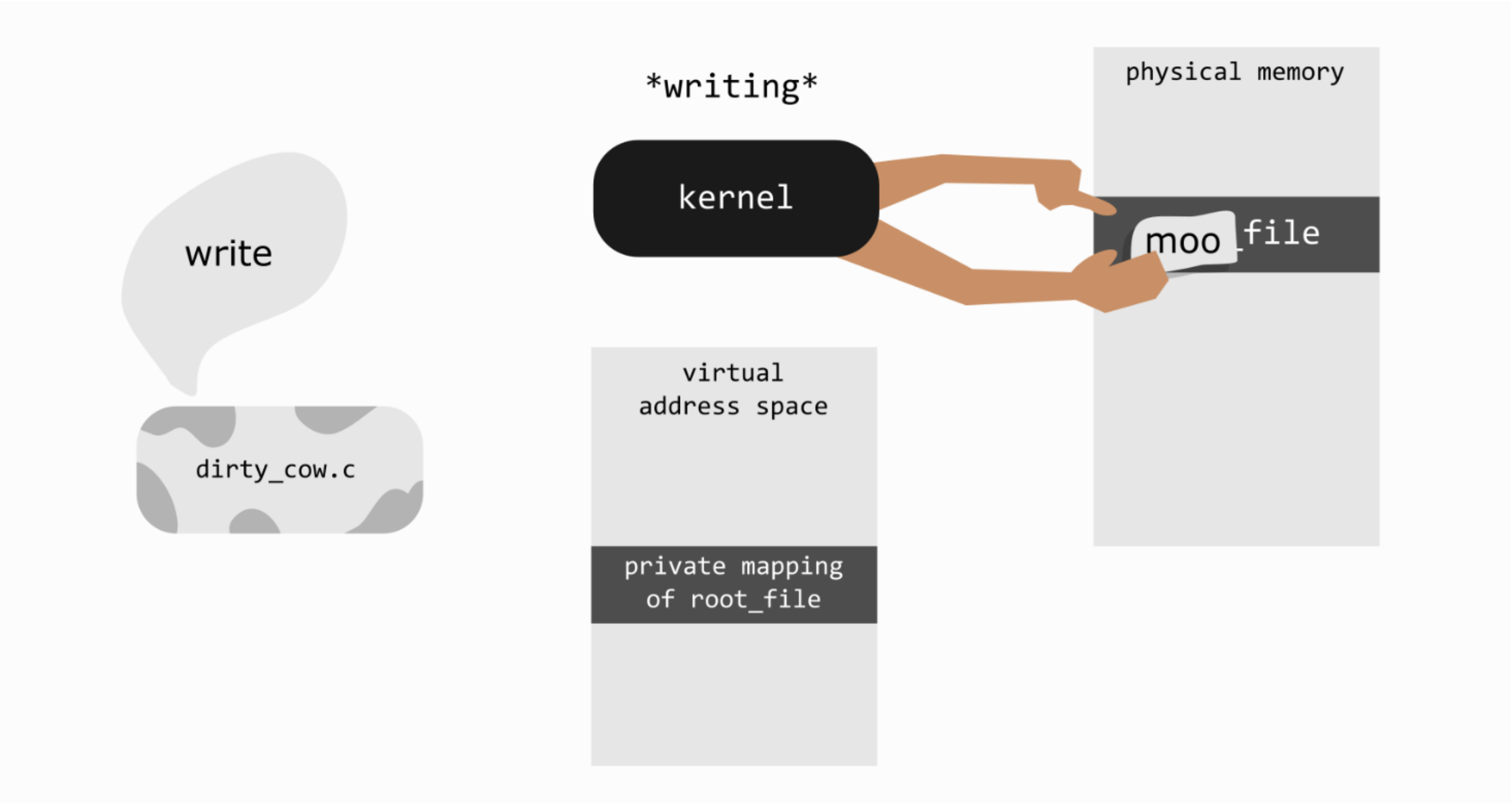
madvise... DONTNEED
Okay.
physical memory
kernel
write
moo
root_file
virtual address space
dirty_cow.c
private mapping of root_file

dirty COW - 16



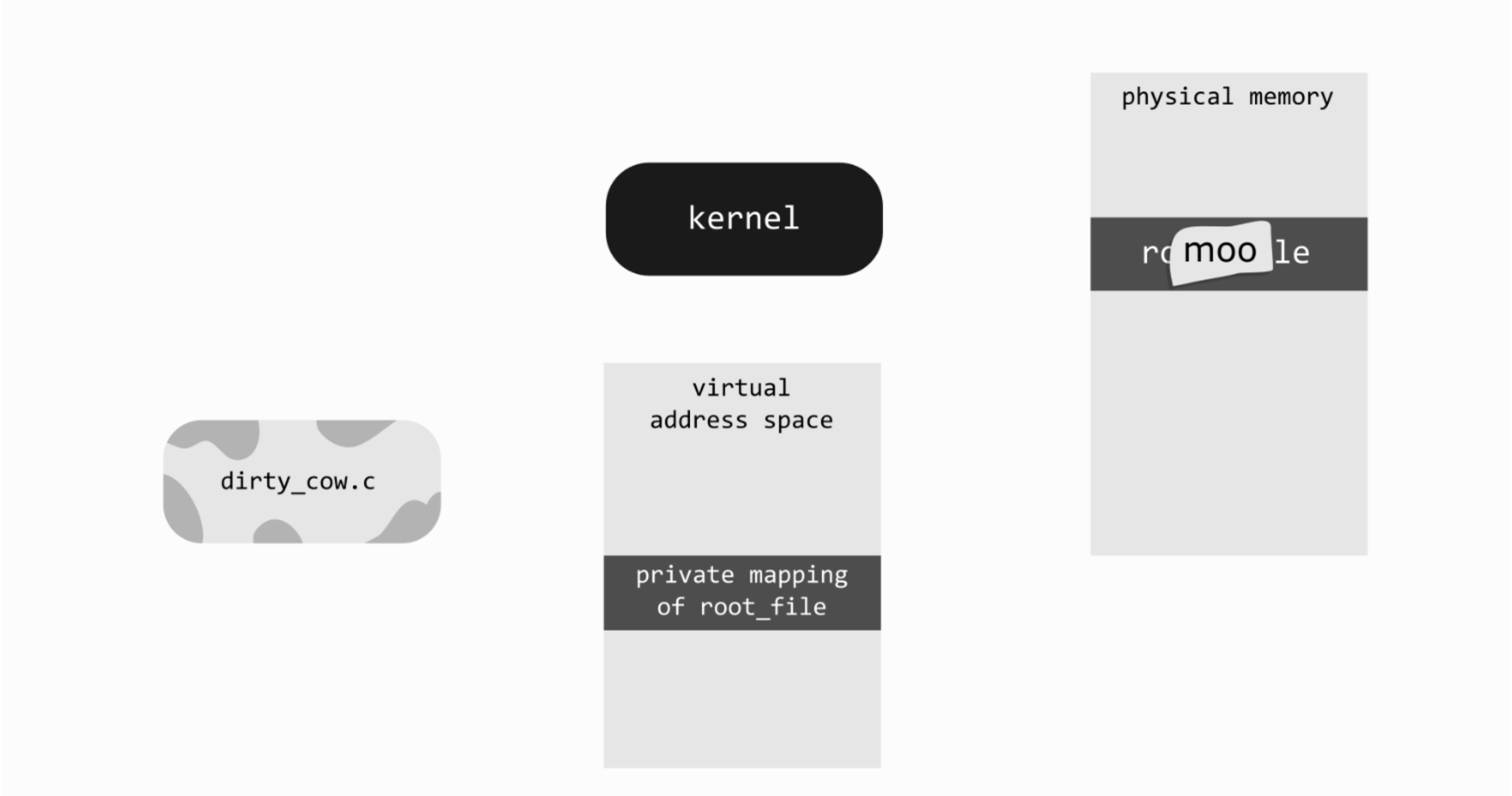
physical memory
kernel
root_file
write
moo
virtual
address space
dirty_cow.c
private mapping
of root_file

dirty COW - 17



writing
physical memory
kernel
moo
file
write
dirty_cow.c
virtual
address space
private mapping
of root_file

dirty COW - 18



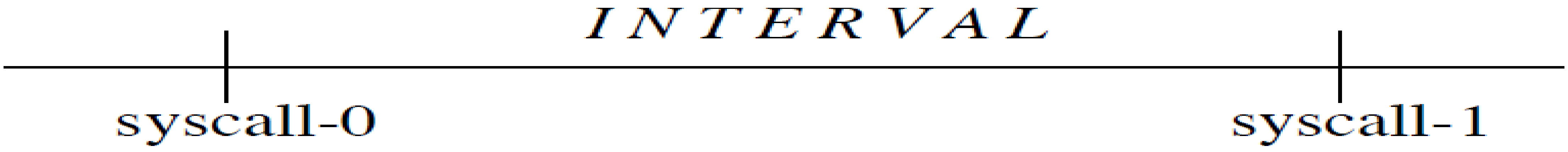
physical memory
kernel
rc moo le
virtual
address space
dirty_cow.c
private mapping
of root_file

Race condition vulnerability – Real life examples

- see all at http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html
- see https: //web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-362
- CVE-2016-7916
  - Linux kernel
  - published on 2016-11-16
  - allows local users to obtain sensitive information from kernel memory
- CVE-2016-3914
  - race condition in Android 4.x, 5.x, 6.x
  - published on 2016-10-10
  - allows attackers to gain privileges via a crafted application that modifies a database between two open operations

Time-of-Check to Time-of-Use (TOCTOU)

- steps
  1. check the state of a resource before using it
  2. use the resource if state is good
- problem: resource’s state changed between check and use
- vulnerability: attacker change the resource’s state to take some advantage
- see Matt Bishop, Michael Dilger, “Checking for Race Conditions in File Accesses”, 1996



TOCTOU - Overview

- existence of such an interval: programming condition
- programming interval: the interval itself
- environmental condition: the attacker be able to affect the assumptions created by the program’s first action
- **-> both conditions must hold for an exploitable TOCTTOU**
- **binding flaw**

TOCTOU - Example

- context: a privileged (SUID) application checks if real UID has access to a file
- file could be changed between access() and open()
- called TOCTOU binding flaw

```
void main(int argc, char **argv) {
    int fd;
    if (access(argv[1], W_OK) != 0)
        exit(1);
    fd = open(argv[1], O_RDWR);
    /* Use fd... */
}
```

TOCTOU – Example (2)

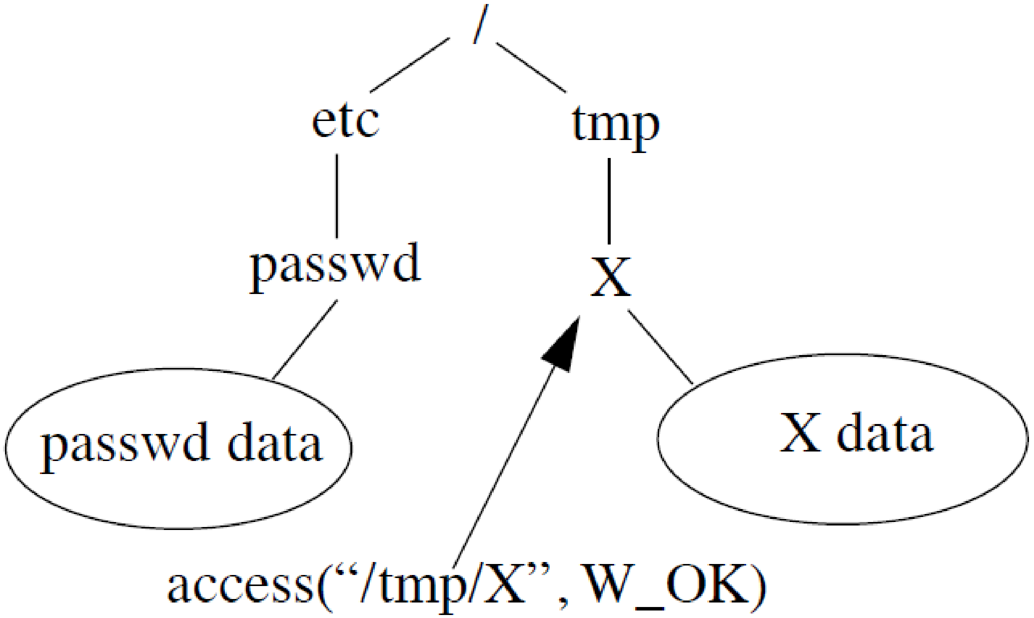


Figure 1a.

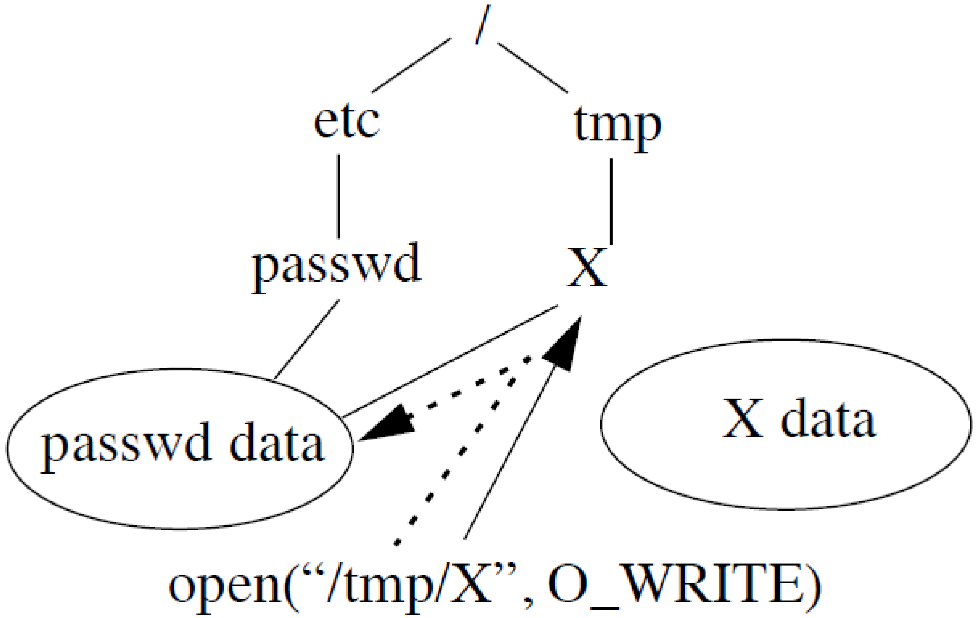


Figure 1b.

TOCTOU – Example (3)

- move dir while the programming is traversing the sub-tree beneath dir,
- -> cause the program to delete files it did not intend to delete

```
void deltree(char *dir) {
   chdir(dir);
   /* Recursively delete
   contents of dir ... */
   chdir("..");
}
```

TOCTOU – Example (4)

- create the file before the victim does
- -> control the permissions and owner of the file
- -> cause the program to open some other file that already exists on the system

```
int mktmpfile(char *fname) {
   int fd = -1;
   struct stat buf;
   if (stat(fname, &buf) < 0)
      fd = open(fname, O_CREAT, S_IRWXU);
   return fd;
}
```

TOCTOU – Example (5)

- modifies the symbolic link exe either immediately before or after the last call to lstat
- -> can execute arbitrary code as another user

```
int run(char *exe) {
  struct stat s[3];
  lstat(exe, &s[0]);
  stat(exe, &s[1]);
  if (s[0].st_uid != s[1].st_uid)
    exit(1);
  lstat(exe, &s[2]);
  setreuid(s[2].st_uid, s[2].st_uid);
  execl(exe, NULL);
}
```

TOCTOU – other examples

root	attacker
	mkdir("/tmp/etc")
	creat("/tmp/etc/passwd")
readdir("/tmp")	
lstat("/tmp/etc")	
readdir("/tmp/etc")	
	rename("/tmp/etc","/tmp/x")
	symlink("/etc","/tmp/etc")
unlink("/tmp/etc/passwd")	

(a) garbage collector

root	attacker
lstat("/mail/ann")	
	unlink("/mail/ann")
	symlink("/mail/ann","/etc/passwd")
fd = open("/mail/ann")	
write(fd,...)	

(b) mail server

root	attacker
access(filename)	
	unlink(filename)
	link(sensitive,filename)
fd = open(filename)	
read(fd,...)	

(c) setuid

TOCTOU - Symlinks and Cryogenic Sleep

- context: reopen files in “/tmp”
- attack
  1. create the expected regular file in “/tmp”
  2. stop application (sending it SIGSTOP) between lstat() and open()
  3. record the device and inode number of the regular file, remove it, and ...
  4. **wait** (possibly very long) until another file with the same values is created
  5. resume application (by sending it SIGCONT)
  6. there could be techniques to increase the chance

```
if (lstat(fname, &stb1) >= 0 && S_ISREG(stb1.st_mode)) {
   fd = open(fname, O_RDWR);
   if (fd < 0 || fstat(fd, &stb2) < 0
         || ino_or_dev_mismatch(&stb1, &stb2))
      raise_big_stink();
   } else {
   /* do the O_EXCL thing */
}
```

Windows process synchronization

- mechanisms to synchronize threads of a process or processes in the system
- synchronization objects
  - types: mutexes, events, semaphores, waitable timers
  - states: signaled and unsignaled
- could be named or unnamed
- share the same namespace with jobs and file-mappings

Windows process synchronization – lack of use

- missing using synchronization objects when needed could lead to unexpected results
- could lead to user array corruption
    - overwrite a (privileged) user with another (non-privileged) one
    - overflow the array

```
char *users[NUSRES];
int crt_idx = 0;
DWORD phoneConferenceThread(SOCKET s) {
    char *name;
    name = readString(s);
    if ((NULL == name) || (crt_idx >= NUSERS))
        return 0;
    users[crt_idx] = name;
    crt_idx++;
    ...
}
```

Lack of use – example (2)

```
function withdraw($amount) {
    $balance = getBalance();
    if($amount <= $balance) {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setBalance($balance);
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

Lack of use – example (2)

Thread 1	Thread 2

Thread 1

```
                          ($10)
function withdraw($amount)
{   ($10,000)
   $balance = getBalance();
   if($amount <= $balance)
   {    ($9,990)
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
```

Thread 2

```
                          ($10)
function withdraw($amount)
{   ($10,000)
   $balance = getBalance();
   if($amount <= $balance)
   {    ($9,990)
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
       setBalance($balance); ($9,990)
   }
   else
   {
       echo "Insufficient funds.";
   }
}
```

Thread 1

```
       setBalance($balance); ($9,990)
   }
   else
   {
       echo "Insufficient funds.";
   }
}
```

Incorrect Use of Synchronization Objects

- application specific
- could lead to data corruption and/or deadlock, even without an attacker interference
- the attacker could try to create the race condition context to gain advantage from
- variant: do not check the return value (success or not) of the synchronization functions

Squatting With Named Synchronization Objects

- context
  - creation of a new synchronization object
  - a synchronization object with the same name could already exist
- case 1: do not check for new object creation success
  - the attacker creates before the application an object with the same name
  - -> could take ownership of the synchronization object
  - change the synchronization objects (e.g. take locks, change semaphores values, signal events etc.)
  - -> control /corrupt the application execution

Squatting With Named Synchronization Objects (2)

- example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if (NULL == hMutex)
   return -1;
...
ReleaseMutex(hMutex);
```

- example 2 (Linux)

```
int semid = semget(ftok("/home/user/file", 'A'), 10, IPC_CREATE | 0600);
...
```

- case 2: check for new object creation success
  - attacker could cause denial of service
  - example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if ((NULL == hMutex) ||
   (GetLastError() == ERROR_ALREADY_EXISTS))
return FALSE;
```

- example 2 (Linux)

Squatting With Named Synchronization Objects (3)

```
int semid = semget(ftok("/home/user/file", 'A'), 10,
                   IPC_CREATE | IPC_EXCL | 0600);
if (semid < 0)
   return -1;
...
```

- case 3: create the object with too much permissions
- attacker could change the synchronization object
- example (Linux)

```
int semid = semget(IPC_PRIVATE, 10, IPC_CREATE | 0666);
if (semid < 0)
   return -1;
...
```

Code review

1. synchronization object scoreboards
    - object name
    - object type
    - using purpose
    - instantiated
    - instantiation parameters
    - permissions
    - used by
    - notes
2. lock matching
    - check for execution paths not releasing a lock
    - limitations: applicable only for locks

Bibliography

1. “The Art of Software Security Assessments”, chapter 13, “Synchronization and State”, pp. ... – ...
2. “The 24 Deadly Sins of Software Security”, chapter 13, pp. 205 –215
3. 3 CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (’Race Condition’), http://cwe.mitre.org/data/definitions/362.html
4. 4 CWE-364: Signal Handler Race Condition, https://cwe.mitre.org/data/definitions/364.html
5. 5 “Delivering Signals for Fun and Profit”, http://lcamtuf.coredump.cx/signals.txt
6. 6 “Symlinks and Cryogenic Sleep”, http://seclists.org/bugtraq/2000/Jan/16