Linux Kernel Memory Management Experiment

518030910031 陳韋廷

1. listvma

a. code

```
void listvma(void) {
  struct task_struct* task;
  struct mm_struct* mm;
  struct vm_area_struct* vma_iter;
  unsigned long perm_flags;
  char perm_str[4];
  perm_str[3] = '\0';

  for_each_process(task) if (strncmp(task -> comm, "test", 4) == 0) break;
  pr_info("[mtest.listvma] Process = %s, PID = %d\n", task -> comm, task -> pid);
  mm = task -> mm;
  if (mm != NULL) {
    vma_iter = mm->mmap;
    while (vma_iter) {
      perm_flags = vma_iter -> vm_flags;
      if (perm_flags & VM_READ) perm_str[0] = 'r';
      else perm_str[0] = '-';
      if (perm_flags & VM_WRITE) perm_str[1] = 'w';
      else perm_str[1] = '-';
      if (perm_flags & VM_EXEC) perm_str[2] = 'x';
      else perm_str[2] = '-';
      pr_info("[mtest.listvma] 0x%lx 0x%lx %s", vma_iter -> vm_start, vma_iter -> vm_end, per
m_str);
      vma_iter = vma_iter -> vm_next;
    }
  }
}
```

b. explanation

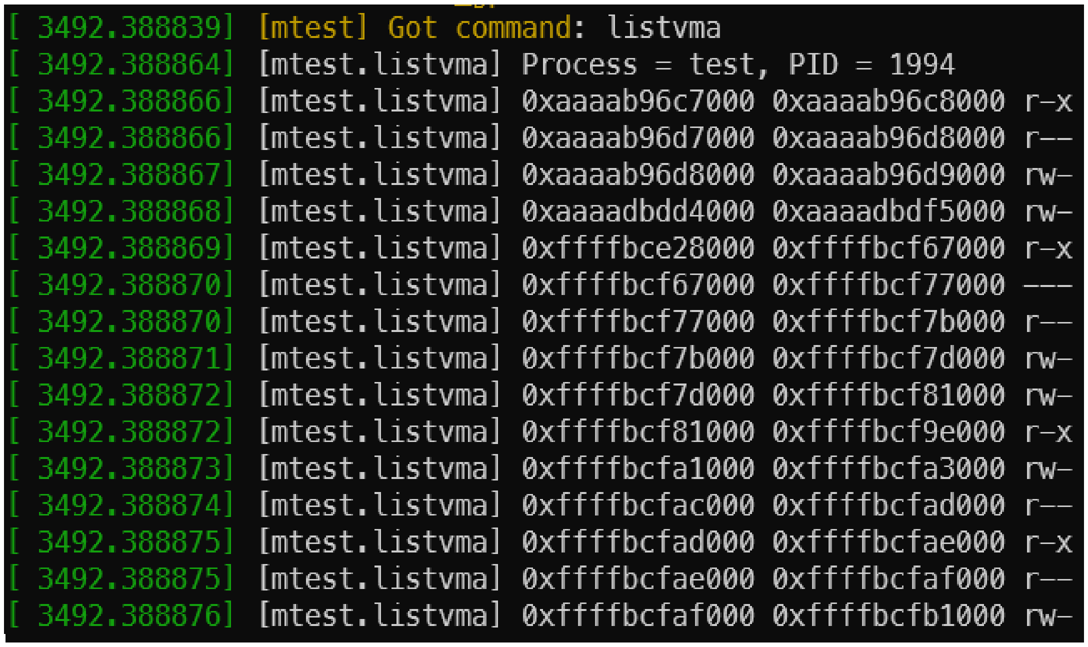
I created a test.c which will allocate some memory to make sure that I would have a process whose memory mapping is consistent during development.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char* dummy = (char*) malloc(64 * sizeof(char));
  memset(dummy, 'A', 64 * sizeof(char));
  printf("Ready\n");
  while (1);
}
```

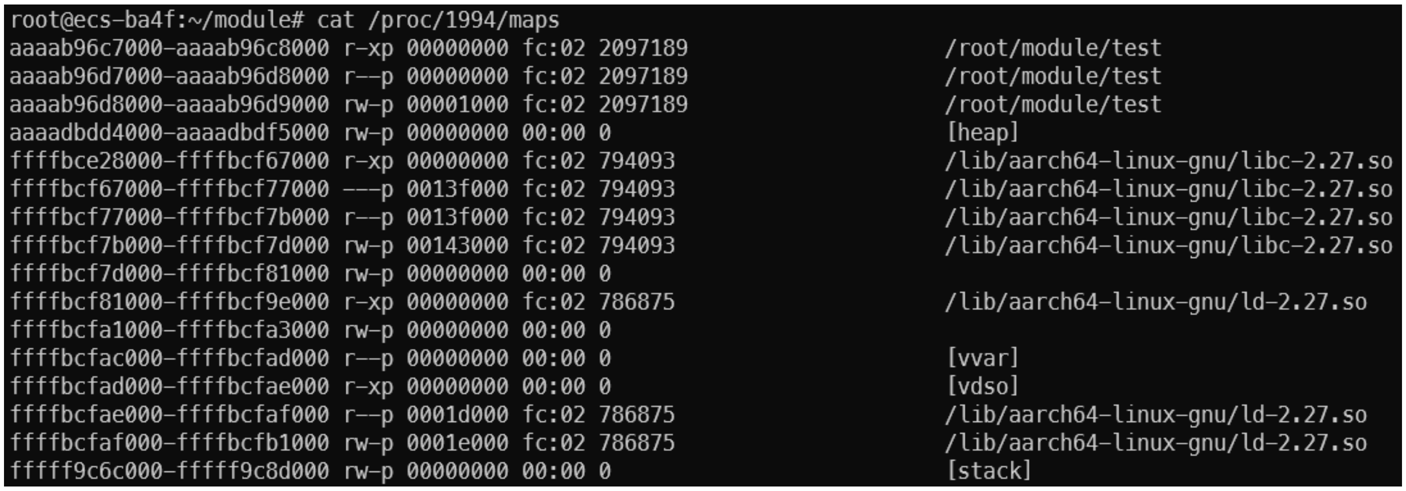
The idea is to find the specific task_struct whose comm is "test" using for_each_process, get its "mm" member, which points to the memory management structure, and go through each doubly-linked vm_area_struct at "mmap".

c. demonstration



```
[ 3492.388839] [mtest] Got command: listvma
[ 3492.388864] [mtest.listvma] Process = test, PID = 1994
[ 3492.388866] [mtest.listvma] 0xaaaab96c7000 0xaaaab96c8000 r-x
[ 3492.388866] [mtest.listvma] 0xaaaab96d7000 0xaaaab96d8000 r--
[ 3492.388867] [mtest.listvma] 0xaaaab96d8000 0xaaaab96d9000 rw-
[ 3492.388868] [mtest.listvma] 0xaaaadbdd4000 0xaaaadbdf5000 rw-
[ 3492.388869] [mtest.listvma] 0xffffbce28000 0xffffbcf67000 r-x
[ 3492.388870] [mtest.listvma] 0xffffbcf67000 0xffffbcf77000 ---
[ 3492.388870] [mtest.listvma] 0xffffbcf77000 0xffffbcf7b000 r--
[ 3492.388871] [mtest.listvma] 0xffffbcf7b000 0xffffbcf7d000 rw-
[ 3492.388872] [mtest.listvma] 0xffffbcf7d000 0xffffbcf81000 rw-
[ 3492.388872] [mtest.listvma] 0xffffbcf81000 0xffffbcf9e000 r-x
[ 3492.388873] [mtest.listvma] 0xffffbcfa1000 0xffffbcfa3000 rw-
[ 3492.388874] [mtest.listvma] 0xffffbcfac000 0xffffbcfad000 r--
[ 3492.388875] [mtest.listvma] 0xffffbcfad000 0xffffbcfae000 r-x
[ 3492.388875] [mtest.listvma] 0xffffbcfae000 0xffffbcfaf000 r--
[ 3492.388876] [mtest.listvma] 0xffffbcfaf000 0xffffbcfb1000 rw-
```

output from dmesg



```
root@ecs-ba4f:~/module# cat /proc/1994/maps
aaaab96c7000-aaaab96c8000 r-xp 00000000 fc:02 2097189                    /root/module/test
aaaab96d7000-aaaab96d8000 r--p 00000000 fc:02 2097189                    /root/module/test
aaaab96d8000-aaaab96d9000 rw-p 00001000 fc:02 2097189                    /root/module/test
aaaadbdd4000-aaaadbdf5000 rw-p 00000000 00:00 0                          [heap]
ffffbce28000-ffffbcf67000 r-xp 00000000 fc:02 794093                     /lib/aarch64-linux-gnu/libc-2.27.so
ffffbcf67000-ffffbcf77000 ---p 0013f000 fc:02 794093                     /lib/aarch64-linux-gnu/libc-2.27.so
ffffbcf77000-ffffbcf7b000 r--p 0013f000 fc:02 794093                     /lib/aarch64-linux-gnu/libc-2.27.so
ffffbcf7b000-ffffbcf7d000 rw-p 00143000 fc:02 794093                     /lib/aarch64-linux-gnu/libc-2.27.so
ffffbcf7d000-ffffbcf81000 rw-p 00000000 00:00 0
ffffbcf81000-ffffbcf9e000 r-xp 00000000 fc:02 786875                     /lib/aarch64-linux-gnu/ld-2.27.so
ffffbcfa1000-ffffbcfa3000 rw-p 00000000 00:00 0
ffffbcfac000-ffffbcfad000 r--p 00000000 00:00 0                          [vvar]
ffffbcfad000-ffffbcfae000 r-xp 00000000 00:00 0                          [vdso]
ffffbcfae000-ffffbcfaf000 r--p 0001d000 fc:02 786875                     /lib/aarch64-linux-gnu/ld-2.27.so
ffffbcfaf000-ffffbcfb1000 rw-p 0001e000 fc:02 786875                     /lib/aarch64-linux-gnu/ld-2.27.so
fffff9c6c000-fffff9c8d000 rw-p 00000000 00:00 0                          [stack]
```

corresponding proc file

2. findpage

a. code

```c
void findpage(unsigned long vaddr) {
  struct task_struct* task;
  pgd_t* pgd; p4d_t* p4d; pud_t* pud; pmd_t* pmd; pte_t* pte;
  unsigned long paddr; unsigned long page_addr; unsigned long page_offset;
  pr_info("[mtest.findpage] vaddr = 0x%lx", vaddr);
  // pr_info("[mtest.findpage] pgtable_l5_enabled = %u\n", pgtable_l5_enabled());
  for_each_process(task) if (strncmp(task -> comm, "test", 4) == 0) break;
  pr_info("[mtest.findpage] current = %s, PID = %d", task -> comm, task -> pid);

  pgd = pgd_offset(task -> mm, vaddr);
  pr_info("[mtest.findpage] pgd_val = 0x%lx\n", pgd_val(*pgd));
  pr_info("[mtest.findpage] pgd_index = %lu\n", pgd_index(vaddr));
  if (pgd_none(*pgd)) {
    pr_info("[mtest.findpage] not mapped in pgd\n");
    return;
  }
  /* p4d = p4d_offset(pgd, vaddr);
  pr_info("[mtest.findpage] p4d_val = 0x%lx\n", p4d_val(*p4d));
  pr_info("[mtest.findpage] p4d_index = %lu\n", p4d_index(vaddr));
  if (p4d_none(*p4d)) {
    pr_info("[mtest.findpage] not mapped in p4d\n");
    return;
  }*/

  pud = pud_offset(pgd, vaddr);
  pr_info("[mtest.findpage] pud_val = 0x%lx\n", pud_val(*pud));
  pr_info("[mtest.findpage] pud_index = %lu\n", pud_index(vaddr));
  if (pud_none(*pud)) {
    pr_info("[mtest.findpage] not mapped in pud\n");
    return;
  }

  pmd = pmd_offset(pud, vaddr);
  pr_info("[mtest.findpage] pmd_val = 0x%lx\n", pmd_val(*pmd));
  pr_info("[mtest.findpage] pmd_index = %lu\n", pmd_index(vaddr));
  if (pmd_none(*pmd)) {
    pr_info("[mtest.findpage] not mapped in pmd\n");
    return;
  }

  pte = pte_offset_kernel(pmd, vaddr);
  pr_info("[mtest.findpage] pte_val = 0x%lx\n", pte_val(*pte));
  pr_info("[mtest.findpage] pte_index = %lu\n", pte_index(vaddr));
  if (pte_none(*pte)) {
    pr_info("[mtest.findpage] not mapped in pte\n");
    return;
  }
```

```
    page_addr = pte_val(*pte) & PAGE_MASK;
    page_offset = vaddr & ~PAGE_MASK;
    paddr = page_addr | page_offset;
    pr_info("[mtest.findpage] page_addr=0x%lx, page_offset = 0x%lx\n", page_addr, page_offset);
    pr_info("[mtest.findpage] vaddr = 0x%lx -> paddr = 0x%lx\n", vaddr, paddr);
}
```

b. explanation

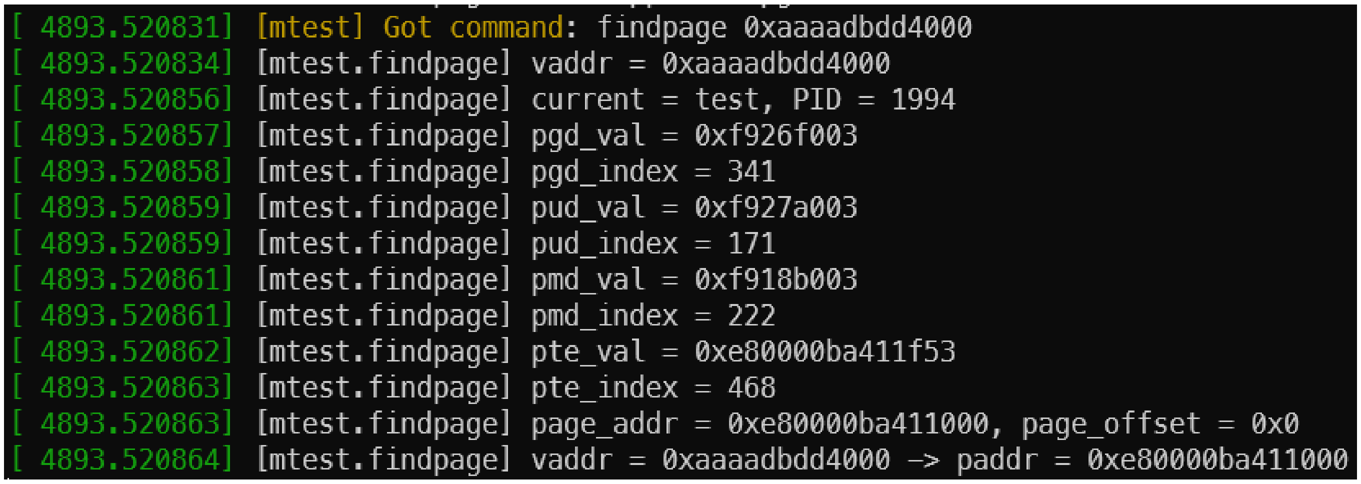
The idea is to go through each level of paging directory and do some calculation to get the address.

When I first tested on my vm, which ran Ubuntu Server 20.04 with x86-64 5.4.0-73-generic kernel, the number of level was 5 (in fact 4 because the default kernel shipped with it was compiled with $CONFIG\_PGTABLE\_LEVELS = 4$), so going through each level would look like this: $pgd \rightarrow p4d \rightarrow pud \rightarrow pmd \rightarrow pte$

However, even in newer kernel, on aarch64 4-level paging is still used, so no level $p4d$, which is why there are commented code. (At least I got compile-time error during module compilation.)

Also, during the first try I was using the global "current" task_struct identifier, and wondering why I couldn't find the corresponding mapping even if I had code identical to others. I finally recalled that each process own its virtual address mapping, so it is impossible to find a memory mapping of other process'.

c. demonstration



```
[ 4893.520831] [mtest] Got command: findpage 0xaaaadbdd4000
[ 4893.520834] [mtest.findpage] vaddr = 0xaaaadbdd4000
[ 4893.520856] [mtest.findpage] current = test, PID = 1994
[ 4893.520857] [mtest.findpage] pgd_val = 0xf926f003
[ 4893.520858] [mtest.findpage] pgd_index = 341
[ 4893.520859] [mtest.findpage] pud_val = 0xf927a003
[ 4893.520859] [mtest.findpage] pud_index = 171
[ 4893.520861] [mtest.findpage] pmd_val = 0xf918b003
[ 4893.520861] [mtest.findpage] pmd_index = 222
[ 4893.520862] [mtest.findpage] pte_val = 0xe80000ba411f53
[ 4893.520863] [mtest.findpage] pte_index = 468
[ 4893.520863] [mtest.findpage] page_addr = 0xe80000ba411000, page_offset = 0x0
[ 4893.520864] [mtest.findpage] vaddr = 0xaaaadbdd4000 -> paddr = 0xe80000ba411000
```

output from dmesg

3. writeval

a. code

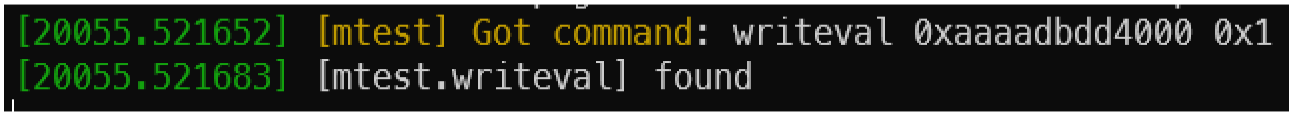
```
void writeval(unsigned long vaddr, unsigned long val) {
  struct vm_area_struct* vma;
  pte_t* pte; void* addr;
  struct page* page;

  vma = _findvma(vaddr);
  if (!vma) {
    pr_info("[mtest.writeval] vma not found\n");
    return;
  }
  if (_vmacanwrite(vma) < 0) {
    pr_info("[mtest.writeval] vma has no write permission\n");
    return;
  }
  pte = _findpte(vaddr);
  if (!pte) {
    pr_info("[mtest.writeval] pte not found\n");
    return;
  }
  page = pte_page(*pte);
  addr = page_address(page);
  pr_info("[mtest.writeval] found\n");
  memset(addr, val, 1);
}
```

b. explanation

After the first and second part, finding the corresponding physical address from a virtual address is possible. To get real access, use pte_page to get the page struct, and finally, page_address to get the pointer to the physical memory. functions that start with underline are self-defined helper function by changing some code from part one and two.

c. demonstration



```
[20055.521652] [mtest] Got command: writeval 0xaaaadbdd4000 0x1
[20055.521683] [mtest.writeval] found
```

output from dmesg

4. final thought

This is a quite tricky experiment because there are not many resources on the internet, especially kernel memory api documentation. There are also traps, such as what the global “current” really points to, which writable memory I can really write to (I once accidentally overwrote the shared libc segment, which made any other command, even ls, clear, unable to continue, resulted in system restart).