CVE-2019-18683: Exploiting a Linux kernel vulnerability in the V4L2 subsystem

1. 漏洞描述

在vivid\_stop\_generating\_vid\_cap()，vivid\_stop\_generating\_vid\_out()和sdr\_cap\_stop\_streaming()中实现了相同的错误锁定方法。

这些函数在流停止过程中被调用，并且锁定了vivid\_dev.mutex。 并且他们在停止其kthread时都犯了相同的错误，这也需要锁定此互斥锁。 请参见vivid\_stop\_generating\_vid\_cap（）中的示例：

/\* shutdown control thread \*/

vivid\_grab\_controls(dev, false);

mutex\_unlock(&dev->mutex);

kthread\_stop(dev->kthread\_vid\_cap);

dev->kthread\_vid\_cap = NULL;

mutex\_lock(&dev->mutex);

但是，当这个互斥锁被解锁时，另一个vb2\_fop\_read()可以锁定它，而不是vivid\_thread\_vid\_cap()，并操作缓冲区队列。这将导致use-after-free。

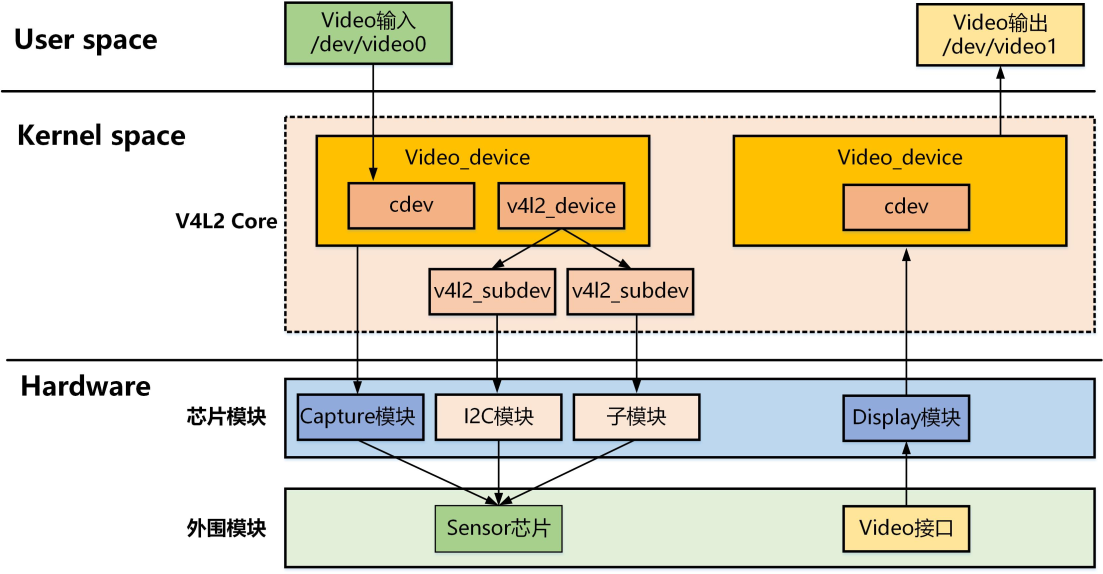
为了解决这些问题，让我们:

1. 避免在vivid\_stop\_generating\_vid\_cap()、vivid\_stop\_generating\_vid\_out()和sdr\_cap\_stop\_streaming()中解锁互斥;

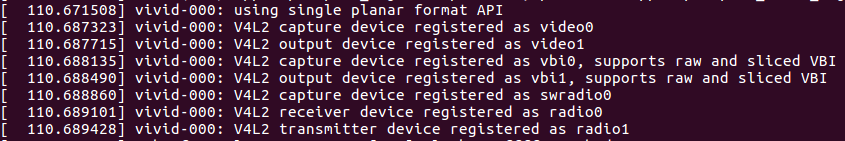
2. 在vivid kthread处理程序的循环中使用mutex\_trylock()和schedule\_timeout()。

1. 漏洞分析

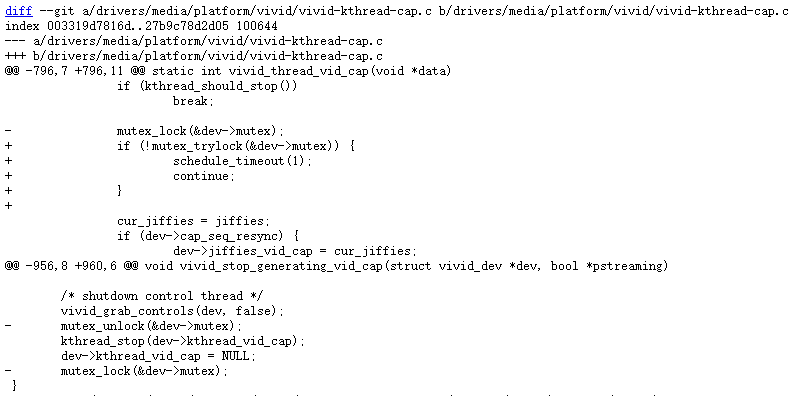
V4L2(Video for Linux two)是Linux内核中关于**视频设备**的内核**驱动框架**，为上层的访问底层的视频设备提供了统一的接口。V4L2框架如下图:



V4L2子系统驱动框架名称是vivid，系统默认不加载该驱动。使用modporbe加载vivid驱动，可以同时将其他支持驱动全部加载到内核中。加载成功如下：



该漏洞类型是use-after-free，由进行错误的锁操作导致的，根据前面漏洞描述可知，在三个函数中进行了错误的锁定操作，以vivid\_stop\_generating\_vid\_cap()函数为例，漏洞补丁如下：

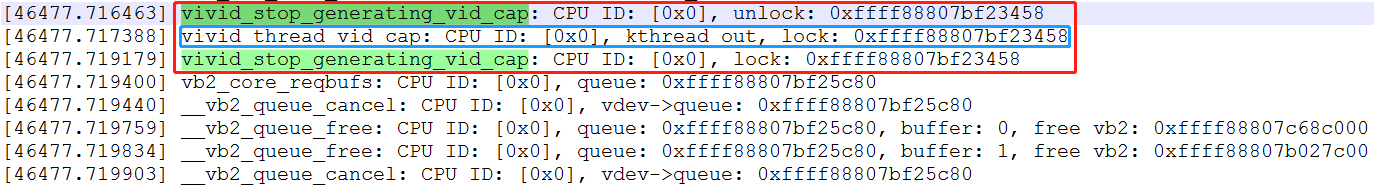


这里vivid\_thread\_vid\_cap（）是由vivid\_start\_generating\_vid\_cap（）创建的kthead函数，vivid\_start\_generating\_vid\_cap（）和vivid\_stop\_generating\_vid\_cap（）是一对，前者负责进行streaming过程中数据填充，后者负责结束这个过程。

先看vivid\_start\_generating\_vid\_cap（）的补丁，将mutex\_lock换成了mutex\_try\_lock，mutex\_lock是阻塞的，如果加锁失败，将一直休眠等待，而mutex\_try\_lock是非阻塞的，它将进行尝试性加锁，如果未加锁成功，将直接返回0。

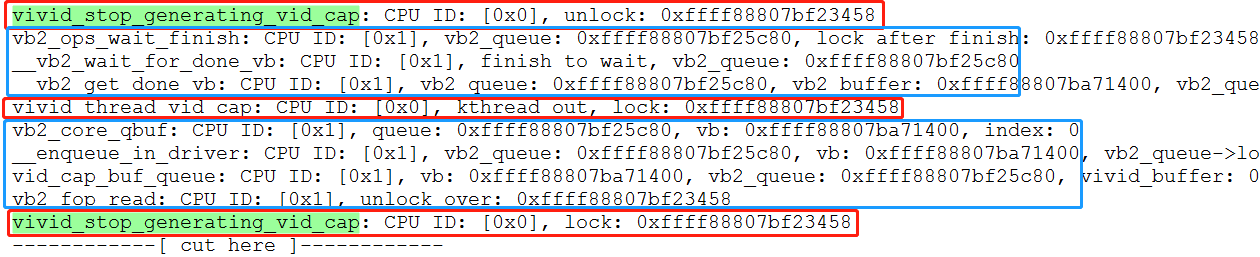
vivid\_stop\_generating\_vid\_cap（）中去掉了加锁和减锁操作。该函数在最后要通知vivid\_thread\_vid\_cap（） kthread函数可以结束了，先进行mutex\_unlock释放锁，这样可以保证vivid\_thread\_vid\_cap（）不会因为加锁失败而进行睡眠最后成为僵尸线程。然后调用kthread\_stop（）给内核线程发送结束信号，最后在mutex\_lock加锁进行后续释放操作。

从设计上看是很合理的，但是实际是不合理的。当vivid\_stop\_generating\_vid\_cap（）中进行mutex\_unlock后，到底谁能获取lock，这是不能保证的。正常情况下应该是这种：

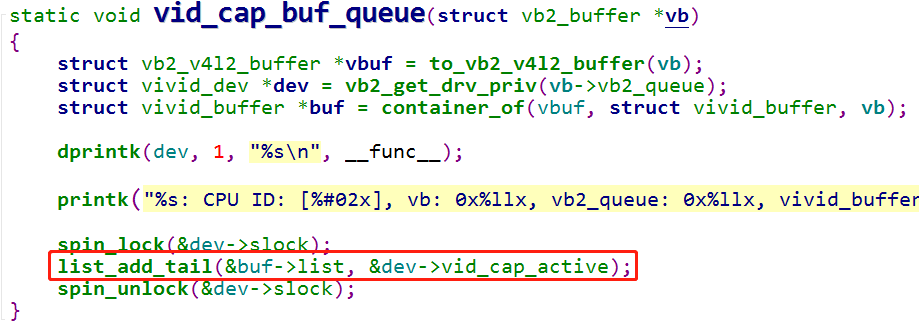


解锁，kthread退出，加锁。三个过程一气呵成，而且都是在同一个任务中进行。

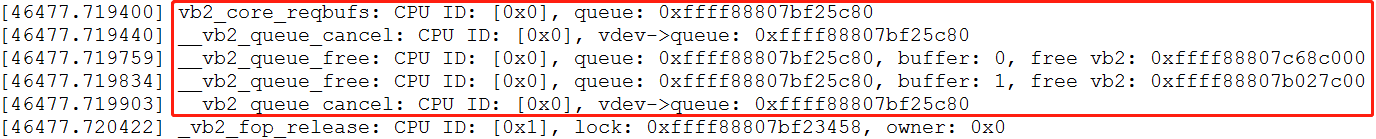
非正常情况下，根据漏洞描述可知，这个锁有可能会被vb2\_fop\_read()拿到，vb2\_fop\_read（）拿到锁后，会对队列进行一系列操作。



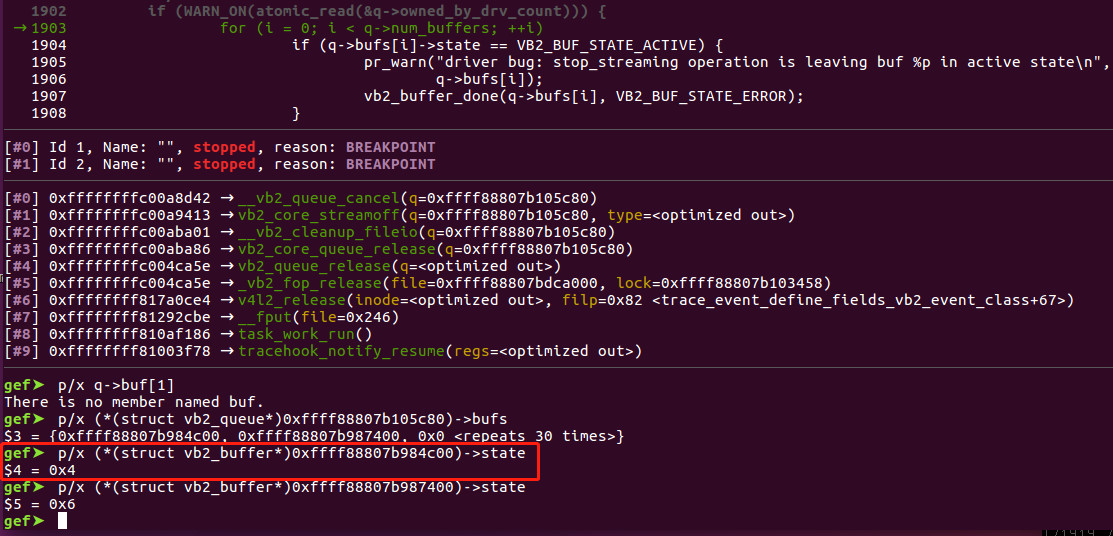
红色框中为CPU0线程，蓝色框中为CPU1线程，发现已经发生了抢锁。蓝框CPU1线程调用vb2\_core\_qbuf（）函数，该函数是用于将vb2\_buffer入队的,然后调用\_\_enqueue\_in\_driver()将将vb2\_buffer放入驱动程序中进行处理，最终会调用vid\_cap\_buf\_queue（）进行处理，实现如下：



将vb2\_buffer添加到dev->vid\_cap\_active链表中，提供给驱动程序去处理。当蓝框CPU1线程处理完释放锁后，红框CPU0线程拿到锁开始继续streamoff的过程，然后继续后续释放vb2\_buffer操作。

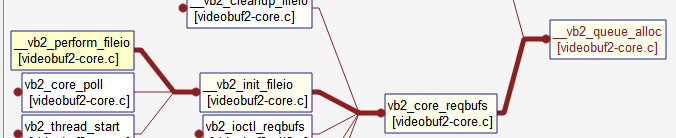


调用\_\_vb2\_queue\_cancel停止或暂停streaming，从驱动程序的队列中删除所有排队的缓冲区，并从videobuf的队列中删除由用户空间排队的所有缓冲区，reqbufs之后返回状态。先通过q->owned\_by\_drv\_count判断queue是否还处于被驱动使用，如果是，再对queue中的vb2\_buffer状态进行判断，

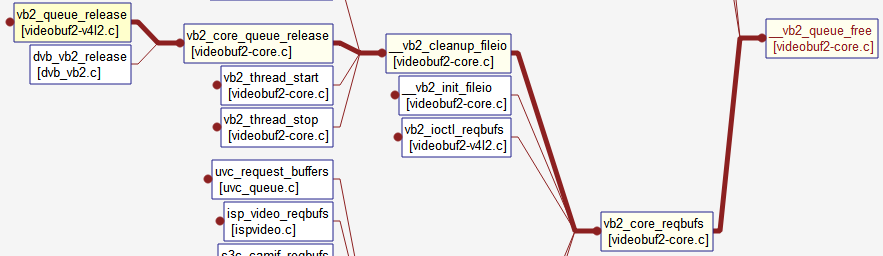


如果状态为VB2\_BUF\_STATE\_ERROR，就会发出警告，提醒目前queue中还有vb2\_buffer被驱动处理使用，进行释放的话，会导致错误。当红框线程结束释放锁后，这时候可能存在蓝框线程拿到锁继续后续的streamon操作，就有可能操作vb2\_buffer，导致use-after-free漏洞造成crash。该漏洞是一个在条件竞争的情况下，错误操作互斥锁导致uaf的漏洞。

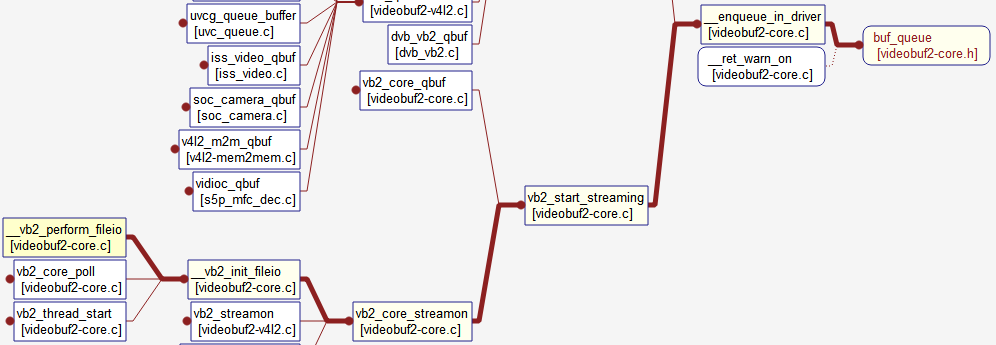
Alloc



Free



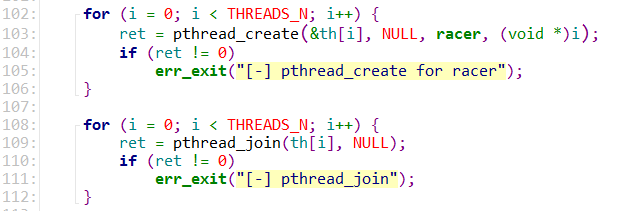
Use



1. PoC和Exploit

1、PoC分析：

启动两个线程用于条件竞争。



竞争函数racer从/dev/video1设备中读取数据，线程绑定到CPU上，方便进行竞争。然后多次循环进行open，read，close操作。

