**内核double fetch型漏洞的利用方法分析：**

**以CVE-2016-6516为例**

今年安全顶会USENIX Security 2017上的一篇文章[1]将内核double fetch型漏洞带入大众视野。Double fetch型漏洞属于竞争类漏洞，能造成内核越界访问或缓冲区溢出，从而导致内核崩溃、提权、信息泄露等严重后果。文章[1]采用静态模式匹配的方法从Linux内核中发掘了6个未知的double fetch型漏洞并提出防御策略，但并没有对其利用方法做详细描述。本文针对double fetch型漏洞，以CVE-2016-6516为例，详述其利用方法及关键技术。

# 1 内核double fetch型漏洞简介

Macintosh HD:Users:wpf:Desktop:princeple-new.eps

图1 Double Fetch原理

现代操作系统普遍采用虚拟内存机制，虚拟内存地址空间被划分为内核空间和用户空间。内核空间运行内核代码，权限高，负责提供系统基础功能。而用户进程则运行在用户空间，通过syscall来使用内核提供的功能。通常，用户提供参数或用户数据给syscall，由syscall验证数据有效性并启动内核处理。当用户数据较为复杂时，内核通常只引用其指针，而将数据留在用户空间。此时，用户数据存在风险，因为其可能被其它恶意用户线程所篡改，造成内核使用的用户数据不一致，从而影响内核正常运行。

如图1所示，double fetch问题的触发需要两个用户线程，第一个线程（trigger thread）为syscall准备用户数据并启动syscall，然后内核对用户提供给syscall的数据进行两次读取，第一次用来检查数据有效性（例如验证指针是否为空，缓冲区大小是否合适等），第二次才会真正使用数据。与此同时，另一个用户线程（flipping thread）通过创造竞争条件（race condition），在两次内核读取之间对用户数据进行篡改（例如将数据长度变量变大造成缓冲区溢出等）。Double fetch型漏洞引发的常见后果是数组访问越界和缓冲区溢出，从而造成内核崩溃或者提权，此外也可造成内核信息泄露等后果[2]。

Serna[3]在2008年第一个为这种类型的漏洞取名为“double fetch”。2013年，Bochpwn[4]采用动态访存追踪的方法针对Windows中的double fetch问题进行研究并发掘了一系列漏洞。2017年，Wang等人采用静态模式匹配的方法针对Linux中（尤其是驱动程序中）的double fetch问题进行研究并发掘了一系列未知漏洞。目前还没有详细的double fetch漏洞利用教程，本文将以CVE-2016-6516为例，详述double fetch漏洞的利用方法及关键技术。

# 2 CVE-2016-6516介绍

CVE-2016-6516发生在Linux kernel-4.5到4.7版本中的ioctl()系统调用中（位于文件fs/ioctl.c中）。ioctl()是设备驱动程序中对设备的I/O通道进行管理的函数，其通过不同的命令参数（cmd）对设备的一些特性进行控制。当cmd = FIDEDUPERANGE时，ioctl()实际调用的是ioctl\_file\_dedupe\_range()函数，其作用是合并映射多个文件中相同的部分以此来节省内存物理空间[5]。

Macintosh HD:Users:wpf:Dropbox:finished paper:CCPE-0906:ccpe_R3:pics:ioctl-h-eps-converted-to.pdf

图2 CVE-2016-6516介绍

如图2所示，在函数ioctl\_file\_dedupe\_range()中，用户数据arge->dest\_count在第579行被内核第一次读取，并用来计算size变量（第584行）。与典型double fetch情景略有不同的是，此处并没有马上发生用户数据的第二次读取，而是对用户的memory region进行了复制（第586行），并将复制后的memory region作为参数传递给了函数vfs\_dedupe\_file\_range()。在此函数中，dest\_count值被从复制的memory region中再一次读取（第1578行），并被作为控制for循环的条件变量使用（第1606和1611行）。当dest\_count在两次内核读取之间被攻击者利用线程间竞争篡改为比实际更大的数值时，将会造成内核访问越界，造成memory corruption，最终导致内核denial of service。

# 3 CVE-2016-6516利用关键技术

当用户调用ioctl(int src\_fd, FIDEDUPERANGE, struct file\_dedupe\_range \*arg)时，除了命令参数FIDEDUPERANGE和源文件描述符src\_fd之外，只需传递一个struct file\_dedupe\_range结构指针指向的用户数据。

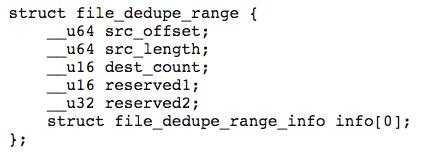


图3 file\_dedupe\_range结构介绍

如图3 所示，file\_dedupe\_range结构主要包括数据块在源文件的起始位置（src\_offset），数据块长度（src\_length），目标文件的数目（dest\_count），以及一个用来描述目标文件的数组file\_dedupe\_range\_info[]。reserved1和reserved2项必须为0。

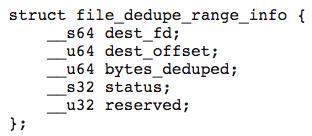


图4 file\_dedupe\_range\_info结构介绍

如图4所示，file\_dedupe\_range\_info结构描述了每一个目标文件的文件描述符（dest\_fd）和数据块起始处（dest\_offset），reserved项也是0。

ioctl\_file\_dedupe\_range()负责检查这些destination sections和source section 是否完全一致，如果一致，则free掉destination section，并将source section映射到对应的destination file。其中bytes\_depued代表成功匹配并被释放的字节数，status存储了操作结束后的状态码。

## 3.1 插桩内核

为了便于观察漏洞利用过程中内核的运行状态，我们对内核中与漏洞相关的函数进行插桩，实时打印状态信息并监视程序执行路径。

我们选取存在漏洞的Linux kerne-4.6.1版本进行实验。需要插桩的函数主要有ioctl\_file\_dedupe\_range()和vfs\_dedupe\_file\_range()两个，分别位于文件fs/ioctl.c以及fs/read\_write.c中。插桩内容如图5和图6所示，一方面是输出关键变量（count）的值，查看其是否被恶意用户线程成功篡改；另一方面是监控程序的执行路径，查看其是否会产生访问越界。

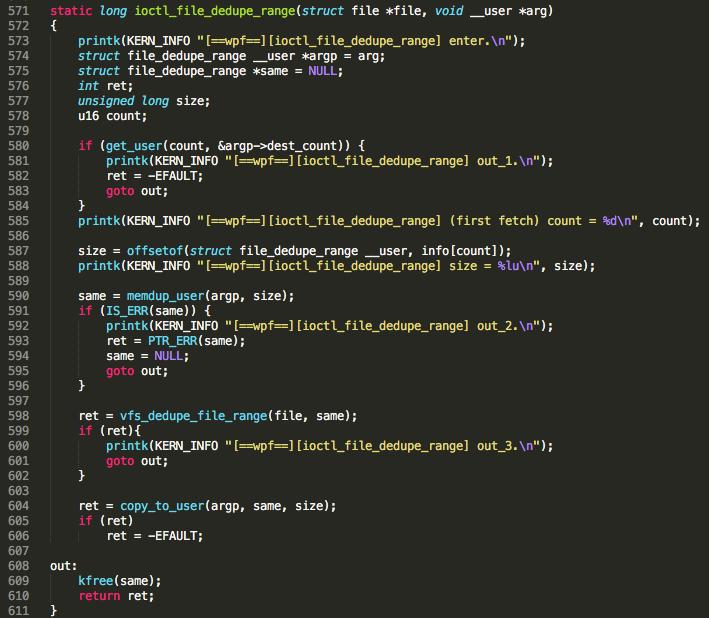


图5 函数ioctl\_file\_dedupe\_range()函数插桩

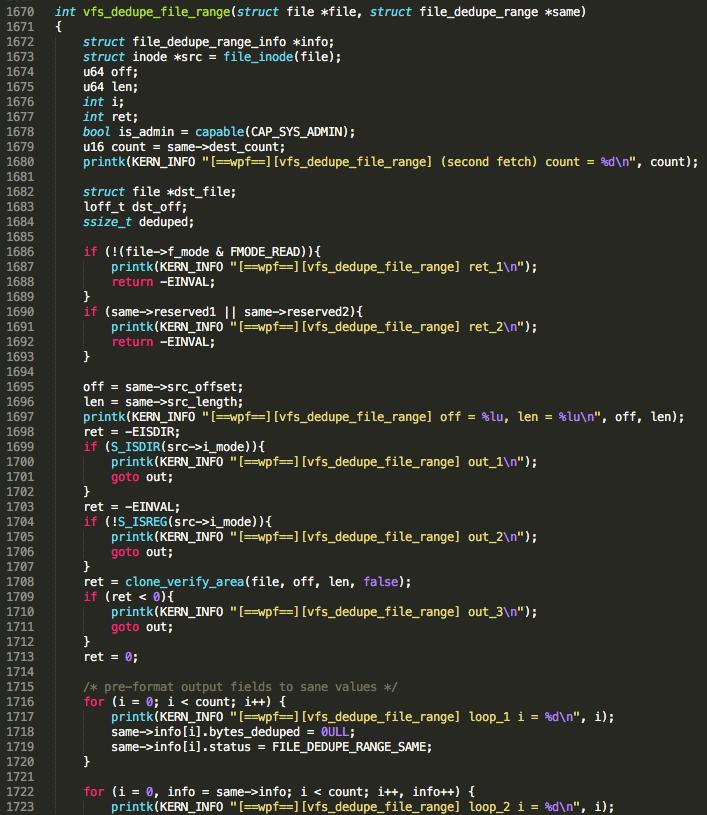


图6 函数vfs\_dedupe\_file\_range()插桩

## 3.2 编译安装新内核

实验所用系统为Ubuntu 16.04 x64。插桩过的Linux kernel-4.6.1需要按照如下步骤重新编译并安装才能使用。

（1）将新内核源文件拷贝到/usr/src/目录下

sudo cp -r ~/Desktop/linux-4.6.1/ /usr/src/

（2）安装必要工具

sudo apt-get install libncurses-dev

sudo apt-get install libssl-dev

（3）编译内核

cd /usr/src/linux-4.6.1/

make mrproper

sudo make menuconfig (直接选exit，save后退出)

sudo make -j2

sudo make modules\_install

sudo make install

安装结束后重启，用uname –a 查看内核版本，确保运行存在漏洞的内核版本。

## 3.3构造exploit

所构造的exploit中，主要包括如下几部分。

（1）相关数据结构的声明

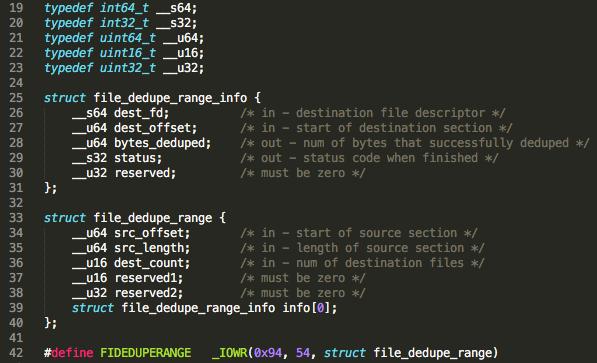


图7 声明数据结构

首先，我们直接在exploit文件中声明需要用到的整数类型和ioctl()所使用的用户数据结构，从而避免引用相应的头文件，简化操作。其中命令参数FIDEDUPERANGE需要用宏\_IOWR声明，详细介绍参照（<http://blog.csdn.net/hepeng597/article/details/7721885>）。其余数据结构如图7所示。

（2）用户数据初始化

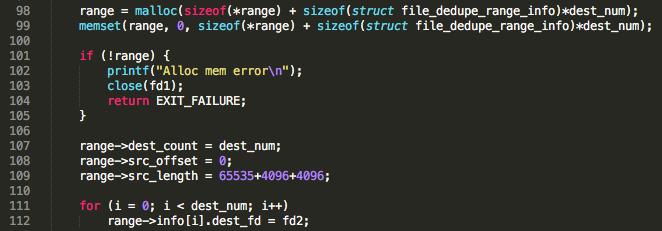


图8 初始化数据

为数据结构分配内存并初始化。这里我们只对关键字段进行初始化，即目标文件数目range->dest\_count，目标文件描述符range->info[i].dest\_fd，以及源文件中的数据块长度range->src\_length。其余字段置0。

（3）trigger thread的实现

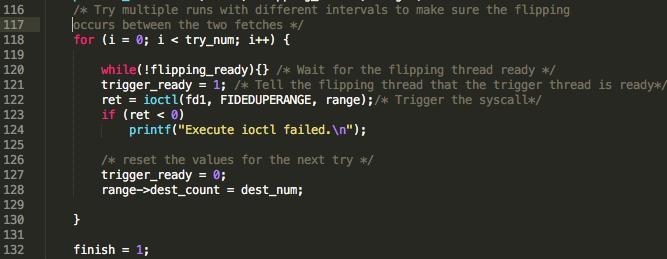


图9 trigger thread实现情况

double fetch型漏洞的利用原理是在同一用户进程内启动两个线程，一个trigger thread负责准备数据并启动目标syscall，另一flipping thread负责篡改数据。利用成功的关键是flipping thread的篡改必须发生在对用户数据的两次内核读取之间。为了达到此目的，我们需要进行多次尝试，使得篡改操作恰好位于两次读取之间。

图9所示为trigger thread的实现，我们将主线程作为trigger thread，并使用循环多次调用ioctl()进行尝试。

（4）flipping thread实现

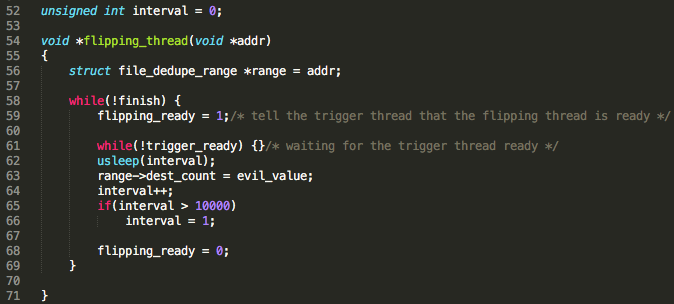


图10 flipping thread实现情况

然后用pthread\_create()创建一个子线程作为flipping thread。我们采用usleep()延迟篡改操作，从0开始，每次尝试递增延时量，直到延时后的篡改操作正好发生于两次内核读操作之间。篡改后的值设为evil\_value，可取任意值。通过实验验证，将尝试次数设置为1000以上，可以达到接近100%的成功率。

（5）线程间同步

图9和图10中的flipping\_ready和trigger\_ready为自己设立的flag变量，用来同步flipping thread和trigger thread，使其从同一时间点开始执行，提高成功率。

# 4. 运行结果

编译 gcc exploit –lpthread –o ex

执行./ex

系统会进入denial of service，但是分为如下两种情况。

# 4.1 内存破坏（memory corruption）

当exploit所篡改的值比较小时，例如将8修改为1024，系统不会立即crash，而是等几秒钟之后才会crash。这是因为CVE-2016-6516漏洞的利用造成了越界访问，在错误的位置写入了错误的数据（如图2所示），导致memory corruption，但是并没有马上引发错误。因此，ioctl()能够执行完毕，并输出相关信息，然后memory corruption会触发系统后续运行的crash。

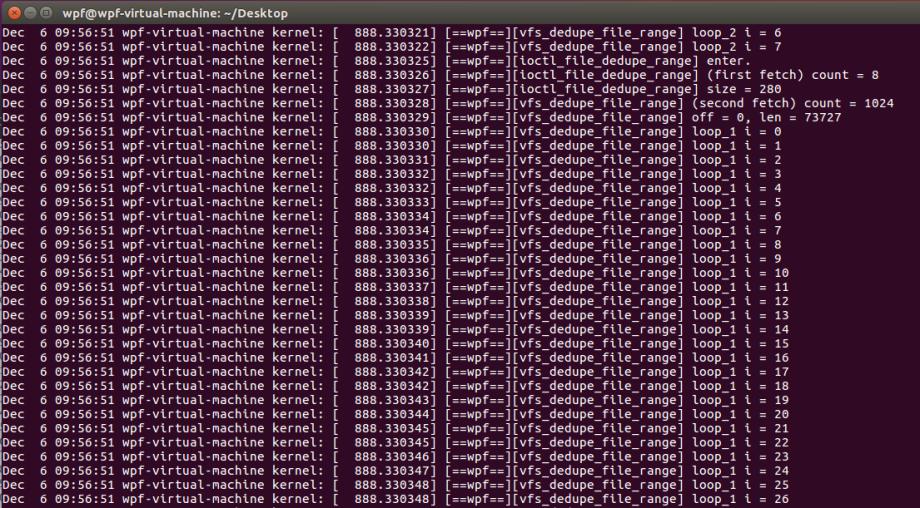


图11 篡改成功后的输出（依靠3.1节中的插桩）

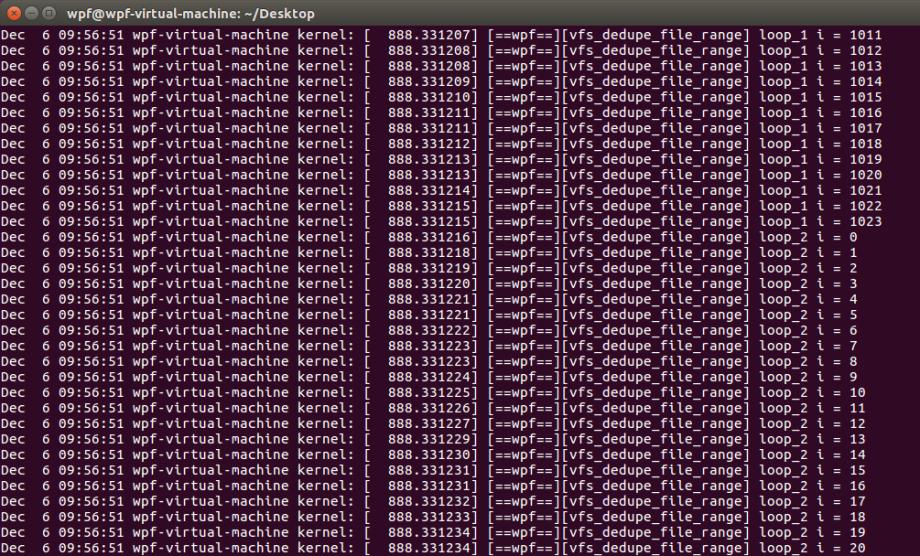


图12 访问越界后的循环依然执行完毕

系统crash掉并重启后，通过命令cat /var/log/syslog 查看系统日志，我们可以从中找到内核插桩后的输出信息。如图11所示，我们可以清楚的看到first fetch的时候 count = 8，到second fetch的时候，count = 1024，说明exploit对rang->dest\_count篡改成功了。但是此时系统并没有马上crash，从图12可以看到，由count =1024控制的循环能够正常执行完毕并输出信息。

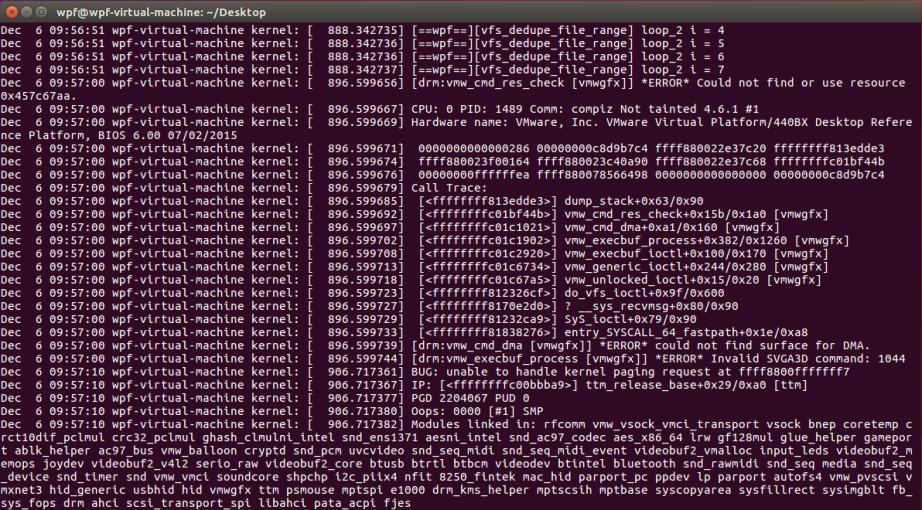


图 13 系统crash信息

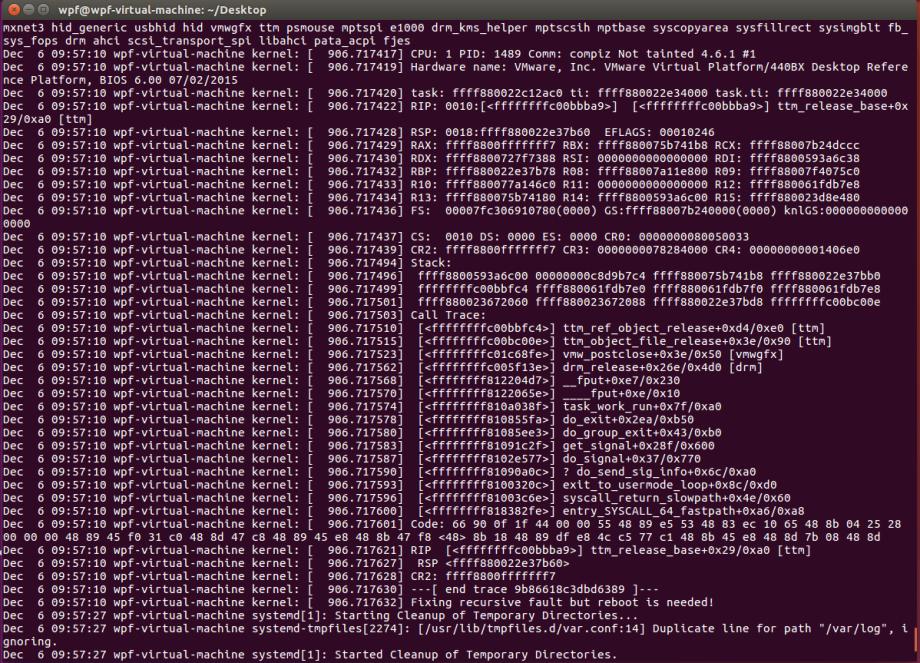


图14 系统crash信息

继续往后找可以找到系统crash后的信息，如图13和14所示。错误信息是“[drm:vmw\_cmd\_dma[vmwgfx]] \*ERROR\* Could not find or use resource 0x457c67aa”

“[drm:vmw\_cmd\_res\_check[vmwgfx]] \*ERROR\* Could not find surface for DMA”

“[drm:vmw\_execbuf\_process[vmwgfx]]” \*ERROR\* Invalid SVGA3D command: 1044”

Oops: 0000 [#1] SMP

# 4.2 地址访问越界（out-of-boundary access）

当exploit所篡改的值比较大时，例如将range->dest\_count由8修改为65535，系统会立即crash。这是因为其访问地址超出了内核的内存页，访问到了没有加载的内存页，造成了访问地址缺页（segment fault）。

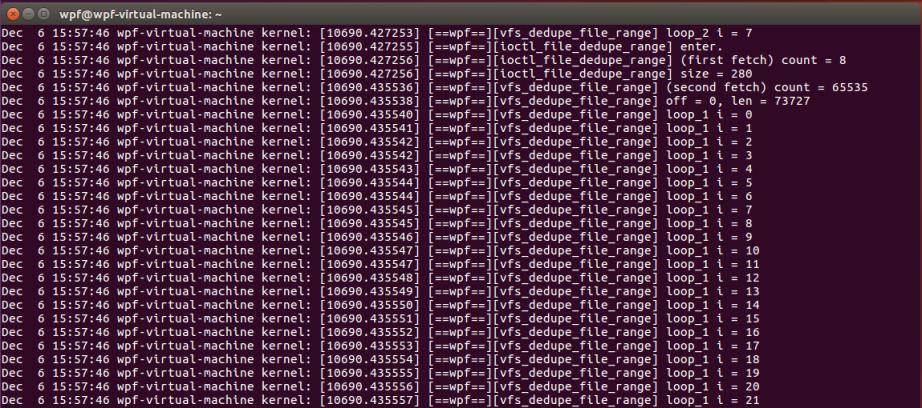


图15 篡改range->dest\_num值为65535时的输出

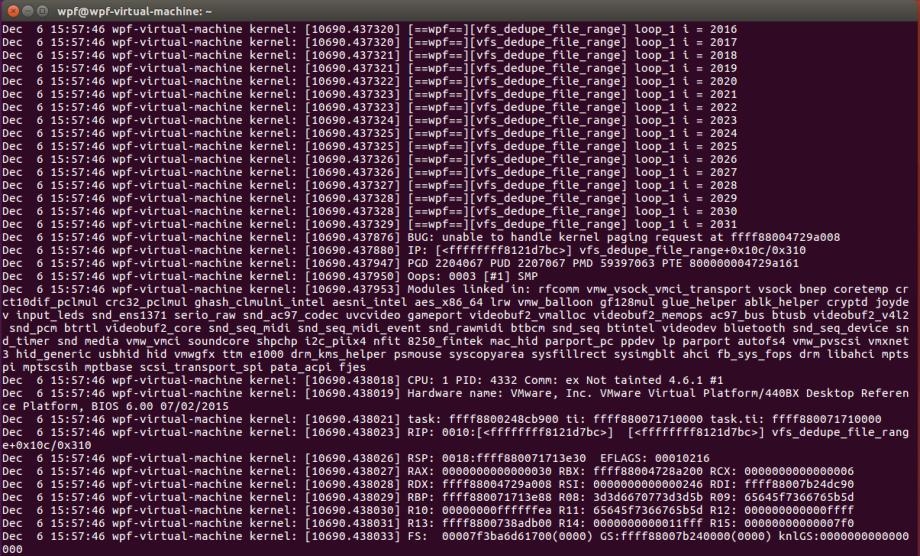


图16 缺页造成的内核denial of service

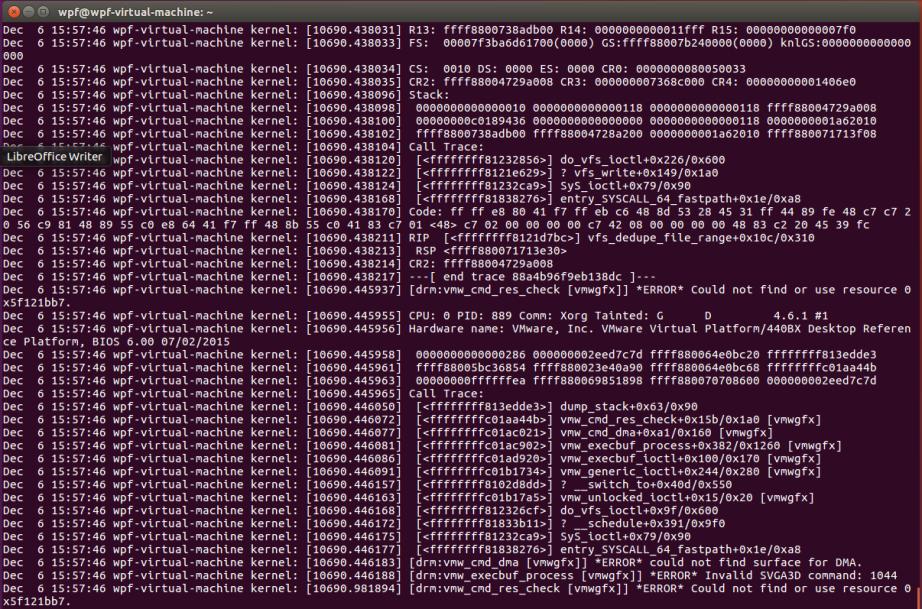


图17 内核crash信息

如图15所示，读取的range->dest\_count值由first fetch时的8成功改为second fetch时的65535。然后程序继续执行，直到循环到2031次时遇到缺页（如图16所示），此时系统输出“BUG: unable to handle kernel paging request at ffff88004729a088”。之后然后内核crash，输出信息如图16和图17所示。

# 5 总结

内核中的double fetch漏洞是一类竞争型漏洞，能够造成内核崩溃、提权、信息泄露等多种严重后果。其存在广泛，在Widows，Linux，Android，FreeBSD等主流操作系统中均发现过double fetch漏洞的踪影。此外，程序开发人员的疏忽极易导致在代码的移植、更新、修改过程中产生double fetch漏洞。文中的CVE-2016-6516漏洞就是在代码移植过程中产生的[6]。相似原因造成的漏洞还包括CVE-2016-5728，在代码更新过程中由于程序员的疏忽，额外又引入了一次对用户数据的读操作而造成了double fetch。

**(注：文中所用exploit部分参考了Scott Bauer的PoC[7]并进行了优化，另文中所有文件可通过**[**https://github.com/wpengfei/CVE-2016-6516-exploit**](https://github.com/wpengfei/CVE-2016-6516-exploit)**获取)**

# 参考文献：

[1] P. Wang and J. Krinke, “How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel,” in USENIX Security Symposium, 2017.

[2]P. Wang, K. Lu et al., “A Survey of the double-fetch vulnerabilities”, 2017.

[3] Serna, F. J. MS08-061: the case of the kernel mode double fetch, 2008. <https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/>.

[4] M. Jurczyk, G. Coldwind et al., “Identifying and exploiting windows kernel race conditions via memory access patterns,” 2013.

[5] <http://man7.org/linux/man-pages/man2/ioctl_fideduperange.2.html>

[6]<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/fs/ioctl.c?h=v4.5&id=54dbc15172375641ef03399e8f911d7165eb90fb>

[7] <http://www.openwall.com/lists/oss-security/2016/07/31/6>