**ecryptfs\_write\_lower()**

**前言**

当用户层发起写操作后，write()函数会使得系统由用户层陷入到内核层完成系统调用。首先，写操作会由VFS（虚拟文件系统）抽象层完成写操作的基本操作，由VFS将具体的写操作转向给具体的文件系统ecryptfs，ecryptfs层会将由用户层传入的数据（char \*buf）进行加密，并将数据划分为若干个extent，每一块需要调用ecryptfs\_write\_lower()函数。

ecryptfs\_write\_lower()函数调用VFS机制，即将加密后的用户层的数据传回给VFS，由VFS抽象层将加密的数据传向至磁盘文件系统（ubuntu下多以ext4文件系统为主）。

ecryptfs\_write\_lower()函数存在如下几个问题：

1. fs\_save = get\_fs()和set\_fs(fs\_save)语句
2. set\_fs(get\_ds());语句
3. vfs\_write()函数
4. 函数的具体参数（ecryptfs\_inode, data, offset, size）

后文均以x86体系作介绍。

**问题一**

1. fs\_save = get\_fs();语句
2. get\_fs()函数会调用current\_thread\_info()函数（该函数头文件为#include <linux/thread\_info.h>）获取当前进程描述符（struct thread\_info），从而找到其成员addr\_limit，该成员是一个结构体，其成员seg保存了当前进程所在的栈的最大地址（内核栈或者用户栈）。此语句表示系统保存当前的状态（即最大地址）。
3. 通过调试程序得到该fs\_save的值为0000 7fff ffff f000（见图（一）），说明此处栈为用户栈，而使用get\_ds()函数获取到的最大地址为ffff ffff ffff ffff（见图（二）），说明如果是内核栈，则addr\_limit.seg为ffff ffff ffff ffff。
4. set\_fs(fs\_save);语句
5. 该语句表示将addr\_limit恢复到原来的addr\_limit。即current\_thread\_info()->addr\_limit = fs\_save；
6. 调试程序可得由ffff ffff ffff ffff恢复到原来的0000 7fff ffff f000。
7. 其它

当系统创建进程时，会为该进程在用户态和内核态分别申请用户栈和内核栈。系统如果在用户栈是无法访问内核栈中的信息，反之亦然。当系统调用时，进程是在用户态，addr\_limit表示的是用户栈的最大地址，系统需要传入数据以供内核使用，而在用户态的进程是无法访问内核栈的区域，所以需要扩大该进程能够访问的范围，扩大范围则需要修改addr\_limit值为内核栈的最大地址（切换模式），很明显，修改该值会影响原来的运行状态，甚至会导致安全问题，所以修改完后立即需要使用set\_fs(fs\_save);语句恢复之前的状态。

内核栈的范围是：0 - 0xFFFFFFFF；用户栈的范围是：0 - 0xBFFFFFFF。

而至于系统如何修改addr\_limit切换模式（mode）以扩大范围，请见问题二。

附图：



图（一）



图（二）

**问题二**

1. get\_ds()函数

在x86中，get\_ds()是宏定义，为KERNEL\_DS；而KERNEL\_DS也是宏定义，为MAKE\_MM\_SEG(-1UL)；MAKE\_MM\_SEG(x)函数将x转化为mm\_segment\_t类型（保持和addr\_limit类型相同）；-1UL即为ffff ffff ffff ffff（在64位机器上，unsigned long为8字节；在32位机器上，unsigned long为4字节）。

1. set\_fs(get\_ds());语句
2. 该语句表示将addr\_limit设置为KERNEL\_DS,其addr\_limit为ffff ffff ffff ffff。即current\_thread\_info()->addr\_limit = ffff ffff ffff ffff；
3. 调试程序可得：由0000 7fff ffff f000设置为内核栈的ffff ffff ffff ffff。
4. 其它

数据在计算机中（磁盘和内存中）是以二进制形式存放，并且是以数据的补码方式存放。正数的原码、反码和补码均为正数其本身，而负数则不同，-1的补码为ffff ffff ffff ffff（8字节）。

附图：



图（三）



图（四）

**问题三**

vfs\_write()函数

1. 系统调用写操作流程：

write() -> VFS(1) -> eCryptfs -> VFS(2) -> DiskFS

1. 实验测试
2. 测试一：

如果在vfs\_write()函数中加入调试语句，编译后测试屏幕会出现大量printk语句，这样非常影响阅读，无法找到关键信息。这样测试，会发现buf中有明文信息存在（如果你在文件中写入hello，则buf内容打印出来为hello）、也有密文信息，明文信息即为在vfs传入ecryptfs的过程中传入用户键入的数据，而密文信息即为ecryptfs加密数据后向vfs传入密文的数据。测试三将验证这一现象。

1. 测试二：

利用file结构体找到magic魔数（file->f\_path.dentry->d\_sb->d\_magic），并判断其是否等于ecryptfs文件系统的魔数。这样测试，会发现buf中有明文信息存在（如果你在文件中写入hello，则buf内容打印出来为hello）。可以想象，明文信息即为在vfs传入ecryptfs的过程中传入用户键入的数据。

结果：

调试后的结果为I am jason（编译内核后，挂载上ecryptfs，并用vi创建并写入”I am jason”，保存退出），写操作调用VFS的vfs\_write()函数操作（检查地址是否合法后），buf参数保存的内容为i am jason，见如图（五）。

1. 测试三：在fs/read\_write.c文件中，复制函数vfs\_write()，命名为jason\_test\_vfs\_write()函数；修改ecryptfs\_write\_lower()函数中的vfs调用，将vfs\_write()函数改为jason\_test\_vfs\_write()函数；并在include/linux/fs.h中添加extern ssize\_t jason\_test\_vfs\_write(…)省略号中的内容和vfs\_write()保持一致。这样的好处，可以单一验证ecryptfs调用VFS写操作的过程。

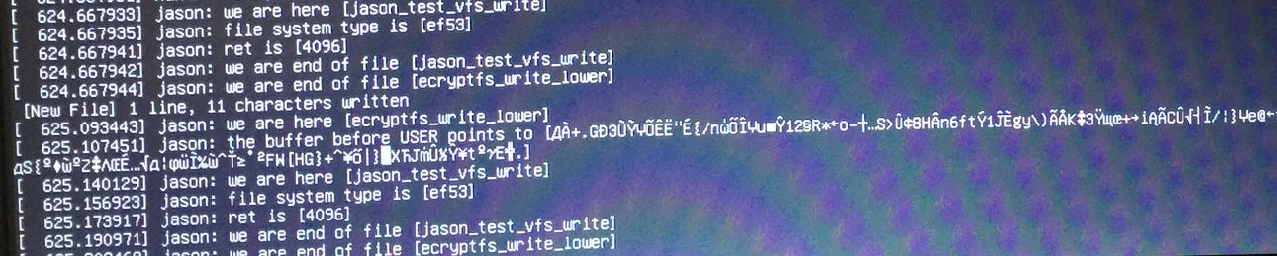
测试中，发现系统调用jason\_test\_vfs\_write()函数，传入的buf都是不可读的信息（密文信息），而没有明文信息，这就能够验证了ecryptfs调用vfs、传入vfs加密的数据这一说法了，见图（六/八）。Ecryptfs调用VFS传入的file结构体对应的magic魔数为ext4（磁盘文件系统），而不是ecr文件系统，见图（九）。

1. 测试四：为了更好的验证测试三的结论，在ecryptfs\_write\_lower()函数中打印data参数，并在jason\_test\_vfs\_write()函数中打印buf参数，进行对比。见图（七）。以上两个函数分别打印buf/data指向区域的内容和buf/data指向区域的地址。实验结果显示：二者无论是地址还是内容都是一模一样。原因：\_\_user并不影响二者，而只有在使用sparse（内核编译的时候使用）

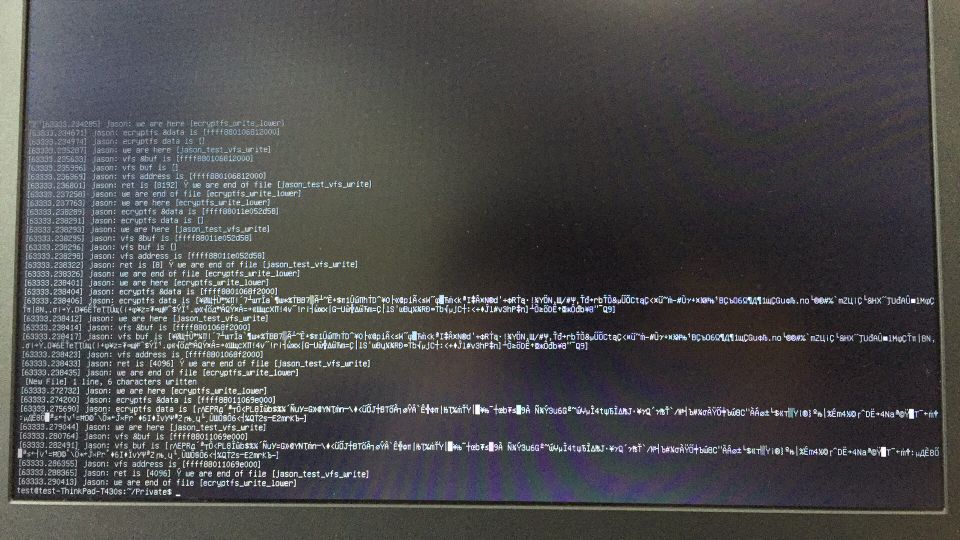
附图：

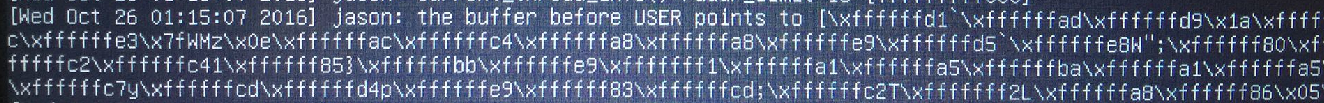


图（五）

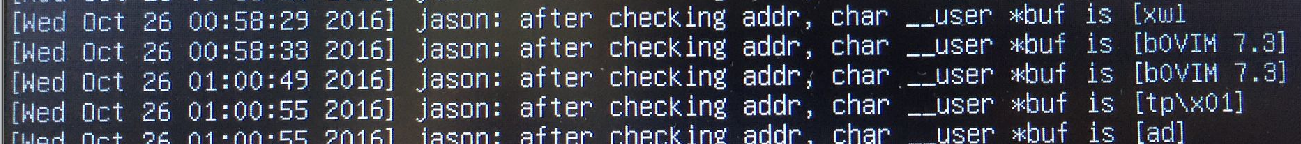


图（六）

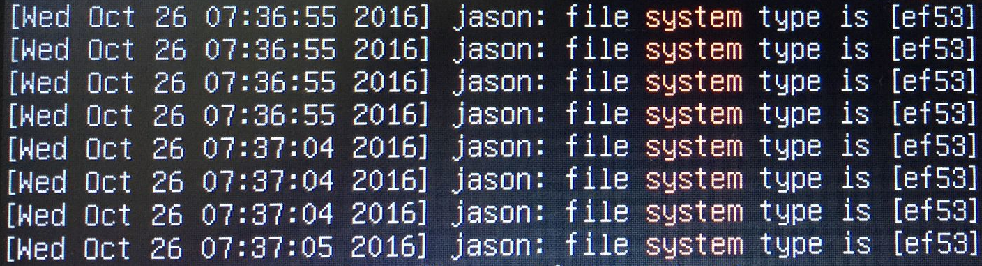




图（七）



图（八）



图（九）

**问题四**

ecryptfs\_write\_lower()函数参数解释：

测试：

1. 测试一：

根据参数ecryptfs\_inode找到magic魔数（ecryptfs\_inode->i\_sb->s\_magic），并打印魔数，其对于ecryptfs文件系统。

1. 测试二：

通过ecryptfs\_inode\_to\_private()函数将ecryptfs\_inode转化为vfs\_inode，并找到magic魔数

（ecryptfs\_inode\_to\_private(ecryptfs\_inode)->vfs\_inode.i\_sb->s\_magic），并打印魔数，其对于ecryptfs文件系统。

1. 测试三：

通过ecryptfs\_inode\_to\_private()函数将ecryptfs\_inode转化为vfs\_inode，并找到magic魔数

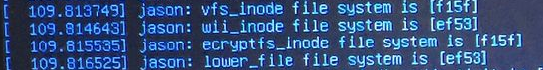
（ecryptfs\_inode\_to\_private(ecryptfs\_inode)->wii\_inode->i\_sb->s\_magic），并打印魔数，其对于ecryptfs文件系统。

实验测试结果见图（十）；

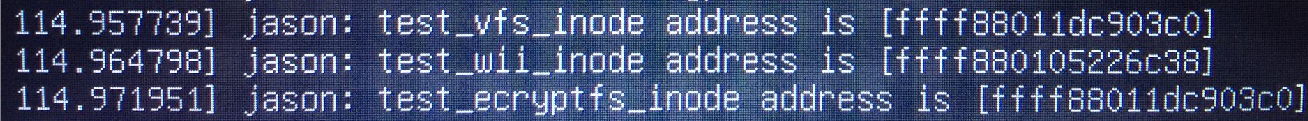
1. 测试四：

wii\_inode所指向的inode结构体即为ecryptfs\_inode\_info结构体的成员vfs\_inode，结果见图（十一）。

附图：



图（十）



图（十一）

**问题杂项**

1. 结构体尽量不要直接赋值
2. 结构体赋值尽量使用memcpy()函数，如果使用a = b的方式去赋值，会出现问题。
3. 结构体的成员中包含了数组、指针则需要小心对待。
4. 结构体（不是结构体指针）不能再if里进行判断
5. 如if(a == b)这是错误的，但是如果需要比较，可以比较其结构体地址，如：if(&a == &b)。