FUSE

Definitions

Userspace filesystem:
: A filesystem in which data and metadata are provided by an ordinary userspace process. The filesystem can be accessed normally through the kernel interface.

Filesystem daemon:
: The process(es) providing the data and metadata of the filesystem.

Non-privileged mount (or user mount):
: A userspace filesystem mounted by a non-privileged (non-root) user. The filesystem daemon is running with the privileges of the mounting user. NOTE: this is not the same as mounts allowed with the "user" option in /etc/fstab, which is not discussed here.

Filesystem connection:
: A connection between the filesystem daemon and the kernel. The connection exists until either the daemon dies, or the filesystem is umounted. Note that detaching (or lazy umounting) the filesystem does *not* break the connection, in this case it will exist until the last reference to the filesystem is released.

Mount owner:
: The user who does the mounting.

User:
: The user who is performing filesystem operations.

What is FUSE?

FUSE is a userspace filesystem framework. It consists of a kernel module (fuse.ko), a userspace library (libfuse.*) and a mount utility (fusermount).

One of the most important features of FUSE is allowing secure, non-privileged mounts. This opens up new possibilities for the use of filesystems. A good example is sshfs: a secure network filesystem using the sftp protocol.

The userspace library and utilities are available from the FUSE homepage:

Filesystem type

The filesystem type given to mount(2) can be one of the following:

fuse
: This is the usual way to mount a FUSE filesystem. The first argument of the mount system call may contain an arbitrary string, which is not interpreted by the kernel.

fuseblk
: The filesystem is block device based. The first argument of the mount system call is interpreted as the name of the device.

Mount options

fd=N
: The file descriptor to use for communication between the userspace filesystem and the kernel. The file descriptor must have been obtained by opening the FUSE device ('/dev/fuse').

rootmode=M
: The file mode of the filesystem's root in octal representation.

user_id=N
: The numeric user id of the mount owner.

group_id=N
: The numeric group id of the mount owner.

default_permissions
: By default FUSE doesn't check file access permissions, the filesystem is free to implement its access policy or leave it to the underlying file access mechanism (e.g. in case of network filesystems). This option enables permission checking, restricting access based on file mode. It is usually useful together with the 'allow_other' mount option.

allow_other
: This option overrides the security measure restricting file access to the user mounting the filesystem. This option is by default only allowed to root, but this restriction can be removed with a (userspace) configuration option.

max_read=N
: With this option the maximum size of read operations can be set. The default is infinite. Note that the size of read requests is limited anyway to 32 pages (which is 128kbyte on i386).

blksize=N
: Set the block size for the filesystem. The default is 512. This option is only valid for 'fuseblk' type mounts.

Control filesystem

There's a control filesystem for FUSE, which can be mounted by:

```
mount -t fusectl none /sys/fs/fuse/connections
```

Mounting it under the '/sys/fs/fuse/connections' directory makes it backwards compatible with earlier versions.

Under the fuse control filesystem each connection has a directory named by a unique number.

For each connection the following files exist within this directory:

waiting
: The number of requests which are waiting to be transferred to userspace or being processed by the filesystem daemon. If there is no filesystem activity and 'waiting' is non-zero, then the filesystem is hung or deadlocked.

abort
: Writing anything into this file will abort the filesystem connection. This means that all waiting requests will be aborted an error returned for all aborted and new requests.

Only the owner of the mount may read or write these files.

Interrupting filesystem operations

If a process issuing a FUSE filesystem request is interrupted, the following will happen:

- If the request is not yet sent to userspace AND the signal is fatal (SIGKILL or unhandled fatal signal), then the request is dequeued and returns immediately.
- If the request is not yet sent to userspace AND the signal is not fatal, then an interrupted flag is set for the request. When the request has been successfully transferred to userspace and this flag is set, an INTERRUPT request is queued.
- If the request is already sent to userspace, then an INTERRUPT request is queued.

INTERRUPT requests take precedence over other requests, so the userspace filesystem will receive queued INTERRUPTs before any others.

The userspace filesystem may ignore the INTERRUPT requests entirely, or may honor them by sending a reply to the *original* request, with the error set to EINTR.

It is also possible that there's a race between processing the original request and its INTERRUPT request. There are two possibilities:

1. The INTERRUPT request is processed before the original request is processed
2. The INTERRUPT request is processed after the original request has been answered

If the filesystem cannot find the original request, it should wait for some timeout and/or a number of new requests to arrive, after which it should reply to the INTERRUPT request with an EAGAIN error. In case 1) the INTERRUPT request will be requeued. In case 2) the INTERRUPT reply will be ignored.

Aborting a filesystem connection

It is possible to get into certain situations where the filesystem is not responding. Reasons for this may be:

a. Broken userspace filesystem implementation
b. Network connection down
c. Accidental deadlock
d. Malicious deadlock

(For more on c) and d) see later sections)

In either of these cases it may be useful to abort the connection to the filesystem. There are several ways to do this:

- Kill the filesystem daemon. Works in case of a) and b)
- Kill the filesystem daemon and all users of the filesystem. Works in all cases except some malicious deadlocks
- Use forced umount (umount -f). Works in all cases but only if filesystem is still attached (it hasn't been lazy unmounted)
- Abort filesystem through the FUSE control filesystem. Most powerful method, always works.

How do non-privileged mounts work?

Since the mount() system call is a privileged operation, a helper program (fusermount) is needed, which is installed setuid root.

The implication of providing non-privileged mounts is that the mount owner must not be able to use this capability to compromise the system. Obvious requirements arising from this are:

A. mount owner should not be able to get elevated privileges with the help of the mounted filesystem
B. mount owner should not get illegitimate access to information from other users' and the super user's processes
C. mount owner should not be able to induce undesired behavior in other users' or the super user's processes

How are requirements fulfilled?

A. The mount owner could gain elevated privileges by either:

   1. creating a filesystem containing a device file, then opening this device
   2. creating a filesystem containing a suid or sgid application, then executing this application

   The solution is not to allow opening device files and ignore setuid and setgid bits when executing programs. To ensure this fusermount always adds "nosuid" and "nodev" to the mount options for non-privileged mounts.

B. If another user is accessing files or directories in the filesystem, the filesystem daemon serving requests can record the exact sequence and timing of operations performed. This information is otherwise inaccessible to the mount owner, so this counts as an information leak.

   The solution to this problem will be presented in point 2) of C).

C. There are several ways in which the mount owner can induce undesired behavior in other users' processes, such as:

   1. mounting a filesystem over a file or directory which the mount owner could otherwise not be able to modify (or could only make limited modifications).

      This is solved in fusermount, by checking the access permissions on the mountpoint and only allowing the mount if the mount owner can do unlimited modification (has write access to the mountpoint, and mountpoint is not a "sticky" directory)

   2. Even if 1) is solved the mount owner can change the behavior of other users' processes.

      i. It can slow down or indefinitely delay the execution of a filesystem operation creating a DoS against the user or the whole system. For example a suid application locking a system file, and then accessing a file on the mount owner's filesystem could be stopped, and thus causing the system file to be locked forever.
      ii. It can present files or directories of unlimited length, or directory structures of unlimited depth, possibly causing a system process to eat up diskspace, memory or other resources, again causing *DoS*.

      The solution to this as well as B) is not to allow processes to access the filesystem, which could otherwise not be monitored or manipulated by the mount owner. Since if the mount owner can ptrace a process, it can do all of the above without using a FUSE mount, the same criteria as used in ptrace can be used to check if a process is allowed to access the filesystem or not.

      Note that the *ptrace* check is not strictly necessary to prevent B/2/i, it is enough to check if mount owner has enough privilege to send signal to the process accessing the filesystem, since *SIGSTOP* can be used to get a similar effect.

I think these limitations are unacceptable?

If a sysadmin trusts the users enough, or can ensure through other measures, that system processes will never enter non-privileged mounts, it can relax the last limitation with a 'user_allow_other' config option. If this config option is set, the mounting user can add the 'allow_other' mount option which disables the check for other users' processes.

Kernel - userspace interface

The following diagram shows how a filesystem operation (in this example unlink) is performed in FUSE.

```
 |  "rm /mnt/fuse/file"               |  FUSE filesystem daemon
 |                                    |
 |                                    |  >sys_read()
 |                                    |    >fuse_dev_read()
 |                                    |      >request_wait()
 |                                    |        [sleep on fc->waitq]
 |                                     |
 |  >sys_unlink()                      |
 |    >fuse_unlink()                   |
 |      [get request from              |
 |       fc->unused_list]              |
 |      >request_send()                |
 |        [queue req on fc->pending]   |
 |        [wake up fc->waitq]          |        [woken up]
 |        >request_wait_answer()       |
 |          [sleep on req->waitq]      |
 |                                     |      <request_wait()
 |                                     |      [remove req from fc->pending]
 |                                     |      [copy req to read buffer]
 |                                     |      [add req to fc->processing]
 |                                     |    <fuse_dev_read()
 |                                     |  <sys_read()
 |                                     |
 |                                     |  [perform unlink]
 |                                     |
 |                                     |  >sys_write()
 |                                     |    >fuse_dev_write()
 |                                     |      [look up req in fc->processing]
 |                                     |      [remove from fc->processing]
 |                                     |      [copy write buffer to req]
 |          [woken up]                 |      [wake up req->waitq]
 |                                     |    <fuse_dev_write()
 |                                     |  <sys_write()
 |        <request_wait_answer()       |
 |      <request_send()                |
 |      [add request to                |
 |       fc->unused_list]              |
 |    <fuse_unlink()                   |
 |  <sys_unlink()                      |
```

Note

Everything in the description above is greatly simplified

There are a couple of ways in which to deadlock a FUSE filesystem. Since we are talking about unprivileged userspace programs, something must be done about these.

Scenario 1 - Simple deadlock:

```
 |  "rm /mnt/fuse/file"               |  FUSE filesystem daemon
 |                                    |
 |  >sys_unlink("/mnt/fuse/file")     |
 |    [acquire inode semaphore        |
 |     for "file"]                    |
 |    >fuse_unlink()                  |
 |      [sleep on req->waitq]         |
 |                                    |  <sys_read()
 |                                    |  >sys_unlink("/mnt/fuse/file")
 |                                    |    [acquire inode semaphore
 |                                    |     for "file"]
 |                                    |    *DEADLOCK*
```

The solution for this is to allow the filesystem to be aborted.

Scenario 2 - Tricky deadlock

This one needs a carefully crafted filesystem. It's a variation on the above, only the call back to the filesystem is not explicit, but is caused by a pagefault.

```
 |  Kamikaze filesystem thread 1      |  Kamikaze filesystem thread 2
 |                                    |
 |  [fd = open("/mnt/fuse/file")]     |  [request served normally]
 |  [mmap fd to 'addr']               |
 |  [close fd]                        |  [FLUSH triggers 'magic' flag]
 |  [read a byte from addr]           |
 |    >do_page_fault()                |
 |      [find or create page]         |
 |      [lock page]                   |
 |      >fuse_readpage()              |
 |         [queue READ request]       |
 |         [sleep on req->waitq]      |
 |                                    |  [read request to buffer]
 |                                    |  [create reply header before addr]
 |                                    |  >sys_write(addr - headerlength)
 |                                    |    >fuse_dev_write()
 |                                    |      [look up req in fc->processing]
 |                                      |       [remove from fc->processing]
 |                                      |       [copy write buffer to req]
 |                                      |         >do_page_fault()
 |                                      |            [find or create page]
 |                                      |            [lock page]
 |                                      |            * DEADLOCK *
```

The solution is basically the same as above.

An additional problem is that while the write buffer is being copied to the request, the request must not be interrupted/aborted. This is because the destination address of the copy may not be valid after the request has returned.

This is solved with doing the copy atomically, and allowing abort while the page(s) belonging to the write buffer are faulted with get_user_pages(). The 'req->locked' flag indicates when the copy is taking place, and abort is delayed until this flag is unset.