# Authentication Infrastructure for Linux kNFSd

Neil Brown

February 7, 2002

**Abstract**

One issue that is important to networking in general and to NFS in particular, but is not addresses particularly well in the Linux kNFSd server, is authentication. The current implementation works fairly well if you have reasonable control of the network, but completely lacks any option for cryptographic authentication, or any obvious way to include such authentication.

This paper discusses the details of authentication in NFS and introduces some modifications to the Linux kNFSd server to help support full authentication.

*Note: This paper will also be available after the conference at* `http://www.cse.unsw.edu.au/~neilb/conf/lca2002/`.

## 1 Introduction

As we all know, the internet is inherently insecure. No self-respecting program should simply accept requests that arrive over the internet and act upon them without some mechanism for checking their origin, and limiting the actions that the request can perform based on that origin.

An NFS server, and the Linux Kernel kNFSd server in particular, is no exception. In fact, as an NFS server has, by its nature, total control of a filesystem, there is a particular need for good authentication and authorisation mechanisms with NFS.

Unfortunately, partly due to the NFS server's location in the kernel, and partly due to the need for speed, high quality authentication has not always been a part of all NFS implementations. The Linux Kernel NFS server implementation (kNFSd) does not have strong authentication capabilities and does not provide any support to add such capabilities.

This paper discusses the authentication mechanisms available in NFS, how these mechanisms present a challenge for the server, and outlines a re-write of the authentication layer in kNFSd to make strong authentication easier to integrate. The re-write does not improve authentication yet, but does resolve some awkwardnesses with the current minimal authentication, and makes it easier to integrate better authentication.

## 2 Background: how authentication works in NFS

Following the tradition of Unix, an NFS service is provided using a combination of a number of discrete components. To understand authentication in NFS, we need to understand how authentication relates to each of these components.

The components are all sub-protocols. They are:

**ONC/RPC** (Open Network Computing/Remote Procedure Call): sometimes known as SUN/RPC because Sun Microsystems developed it. This protocol underlies all other parts of NFS.

**MOUNTD:** a protocol to allow clients to find a filesystem on a server, and to tell it when the filesystem is mounted or unmounted on the client.

**NFS:** the core protocol for accessing files. It is essentially stateless.

**NLM** — the Network Lock Management protocol: A stateful adjunct to NFS for file and record locking.

**STATMON** — The Status Monitoring protocol: An adjunct to NLM to enable detection of system restarts.

Every ONC/RPC (henceforth "RPC") request (and hence every NFS related request) contains a *credential* and a *verifier* that are intended to be used for authentication. The credential contains information about the source of the request, and the verifier contains information which can be used by the server to verify the authenticity of the credential.

A purely hypothetical example might include a username and NIS domain name in the credential, and the verifier might contain an MD5 checksum of the credential combined with some shared secret.

The reply to an RPC request also contains a verifier which the client can use to confirm the the reply came from the actual server.

In practice, the vast majority of RPC implementations use one of two credential schemes and precisely one verifier scheme.

The credential schemes are AUTH_UNIX (aka AUTH_SYS) which contains a machine name, a UID, and a list of GIDs; and AUTH_NONE (aka AUTH_NULL) which contains no identifying information.

Both of these schemes only support one verifier scheme, which also has the name AUTH_NONE. You can guess what this does.

Other credential schemes that are in use to some extent are AUTH_DH (aka AUTH_DES) and AUTH_KERB.

AUTH_DH uses a Diffie-Hellman public key mechanism to sign all requests. This is used in "SecureRPC" and "SecureNFS" products available from SUN Microsystems and other vendors.

AUTH_KERB is based on the Kerberos version 4 security infrastructure to provide a reasonable level of security for RPC requests.

Quite clearly AUTH_UNIX combined with AUTH_NULL verification is not worth the paper that it is written on (or bits it is encoded in). An NFS server that simply trusted such a credential would not last long. Hence it is common practice with AUTH_UNIX based services to rely on the source IP address and port number to act as a verifier (though the source port number has not always been included in the calculation). This is still not perfectly secure, but in a closed network with suitable firewalling, it can work reasonably well.

As you can imagine, simply applying some authentication (credential + verifier) scheme uniformly to each of the protocols used for an NFS server (MOUNT, NFS, NLM, STAT) would provide that level of authentication over the whole system. But it does not or, at least, has not always worked that way.

While MOUNT, NLM, and STAT are often handled by user-space daemons, the core NFS is usually handled by kernel code. Running in the kernel, it is not particularly easy to check if a given IP address maps to a host name that matches a given pattern, or a host that is in a particular netgroup. As I understand it, early implementations of NFS (before Linux) did not do any source based authentication, but rather did content based authentication.

Every (non-trivial) NFS request must contain a filehandle. This is a 32 byte (or up to 64 byte in NFSv3) opaque identifier that the server provides and the client uses to identify a particular file. Clearly not every filehandle will map to an existant file, and if the server can generate filehandles in such a way that the filehandle is not guessable, then any request that contains a valid file handle can be assumed to come from a client with a right to know that file handle. Some systems come with a program `fsirand` which has the purpose of injecting randomness into the inodes in a filesystem so that the filehandles generated for those inodes will not be guessable.

Systems which rely on this content based authentication for core NFS requests put the onus on MOUNT to protect filehandles, as this is the protocol whereby a client gets its first filehandle. I believe that many early NFS systems, including ones that use AUTH_DH or AUTH_KERB for authentication, took this approach of only protecting the MOUNT protocol, and trusting to content based authentication to protect the core NFS protocol.

The NLM and STATMON protocols are much less significant when considering authentication as little harm can be achieved by abusing them. At least one major Operation System (Tru64 from Compaq) still uses AUTH_NONE for all NLM requests and does not seem to suffer for it.

## 3 Midground: Where does Linux kNFSd fit

As mentioned in the introduction, Linux kNFSd is not particularly sophisticated in its handling of authentications, but neither is it totally backward.

The only credential mechanisms that are supported are AUTH_UNIX and AUTH_NONE. This means that we need to put all our trust in the IP headers and the source machine.

kNFSd does do source address (and source port) checking for core NFS requests, and for NLM requests (though these latter can be disabled for Tru64 clients). To achieve this it needs to be given a list of IP addresses of all clients that are allowed to make requests, and which filesystems they are allowed to access.

There are two particular difficulties with keeping this list of IP addresses in the kernel:

- The first is that the list of allowed addresses might be very large (e.g. a whole class B subnet) while the list of active clients might be much smaller. Keeping the whole list in the kernel would be a waste of space and search time.

- The other is that the list might be subject to change. As hosts are added to, or removed from, the DNS or from a "netgroup" map in a NIS database, the correct list of IP addresses might change. Keeping the in-kernel list up-to-date with all these changes would be difficult.

Linux NFS approaches these difficulties by only informing the kernel about IP addresses that are explicitly given in the configuration (`/etc/exports`) and IP addresses of clients that are known to currently have the filesystem mounted.

Thus, whenever a MOUNT request comes from a client that wishes to mount a filesystem, the kernel is told about that client's IP address (assuming it is allowed access) before the filehandle is returned to the client.

To provide continued service over a reboot, `mountd` maintains a file

```
/var/lib/nfs/rmtab
```

that lists which clients have particular filesystems mounted. This is added to when a client mounts a filesystem, and deleted from when a client advises that it has unmounted a filesystem.

Unfortunately, the "unmount" request in the MOUNT protocol is not reliable (if a client crashes it may never be sent) and so `rmtab` tends to collect a lot of records which are no longer valid. This is true with NFS implementations from all vendors.

So, in summary, Linux kNFSd provides about the best AUTH_UNIX authentication that is possible. It does not rely on content for authentication and always checks the source IP address of requests. However it does not provide for any strong cryptographic authentication. Furthermore even the AUTH_UNIX authentication is problematic as it is hard to reliably keep track of which clients have which filesystems mounted.

# 4 Foreground: Where would we like to be tomorrow

Now that we know where we stand, it would help to know where we want to be. Certainly we would like to be able to provide strong, cryptographic authentication with kNFSd; but what should that authentication look like?

An obvious choice might seem to be to use AUTH_DH or AUTH_KERB, however I do not believe that they are very fruitful directions to follow. Partly, this is because they are getting quite old. AUTH_DH is based on DES which, while still popular, is losing favour. AUTH_KERB is based on Kerberos version 4 which has been replaced by version 5.

Furthermore these mechanisms do not provide as strong a protection as you might like. While the verifier securely verifies the credential, it does **not** verify that the operation in the request matches the credential. It would be possible to catch a request in-flight and change the operation while leaving the credential intact. Thus they are open to a man-in-the-middle attack which replaces the operation while preserving the credentials and verifier.

The current future of RPC authentication seems to be a mechanism known as RPCSEC/GSS which is described in RFC2203. This is a mechanism to allow RPC to make use of any authentication scheme which supports the GSS-API (Generic Security Services Application Program Interface) as described in RFC2078. Using appropriate GSS-API facilities, RPC requests can be fully authenticated, integrity checked and, if necessary, encrypted.

Of the GSS-API schemes that are available the one that appeals to me personally is one called LIPKEY, which stands for Low Infrastructure Public KEY. This is in some ways a parallel to SSL (the secure socket layer) that is used for security on the web. However it is packet based rather than connection based.

As a brief outline, a LIPKEY session goes something like this:

1. The client requests a Public Key Certificate from the server, which the server provides.

2. The client verify this certificate to make sure that it is talking to the right host.

3. The client randomly chooses a symmetric session key, encrypts it with the servers public key, and send it to the server. Server and client now share a private session key.

4. The client encrypts the user name and password of the person (principal) that it is acting for and send these to the server.

5. The server verifies the password and then allows the client to perform requests as the given user, with each request securely signed using the session key.

This allows secure, reliable communication without requiring every user to have their own public key certificate. They only need a password which the server can verify.

Of course another aspect of our overall goal would be for the NFS client to be able to speak the same authentication protocol, but that is a whole different story.

# 5 Underground: What would we need to make it work

In order to provide full support for high quality authentication in the kernel some infrastructure needs to be provided to allow different authentication modules to be able to get their job done. The following sections introduce some elements of that infrastructure.

## 5.1 Flavour registration

A *Flavour* in the context of ONC/RPC is a particular style of authentication. AUTH_UNIX and AUTH_NONE are both flavours. Each flavour has a number. This number appears in each RPC request to identify the flavour used, and also in some requests in the MOUNT protocol to allow the client to discover what flavours the server supports and prefers.

To be able to support a number of different authentication flavours we must allow new flavours to be registered with the RPC subsystem. There is nothing particularly new about this, as many subsystems in the Linux kernel already allow registration of instances.

The RPC subsystem does have a concept of different flavours with different handlers (so that it can deal with both AUTH_NONE and AUTH_SYS) but it does not currently allow this to be extended.

### 5.2 Enhanced system call

The kNFSd server has its own system call that is used to start the NFS server and to provide information about exported filesystems to that server.

With different authentication flavours, and with changes in the way that the current flavours are managed, there will need to be some new functionality added to this system call.

Fortunately, the arguments to the syscall are very flexible: an integer command and two pointers, one for arguments and one for results. The only values currently used for the command number are 0 to 8. This leaves lots of room for extension.

What we will do is divide the 32bit command space into 8bit sections. The first section will be for legacy calls to support the current schema. Other sections will be allocated for:

- RPC layer calls such as giving information to authentication flavour modules.
- New kNFSd calls to export filesystems in a way that interacts well with the new authentication setup.
- Possible sections for other RPC related modules such as NFS-client and `lockd`.

### 5.3 Out-Calls or Up-Calls

The most fundamental facility needed is for the `nfsd` threads that are running in the kernel to be able to request something from user-space, such as to confirm whether a given IP address has access to a given filesystem. Without this facility, all information that the kernel might need must be given proactively and as we have seen this is problematic even with a single simple authentication scheme (AUTH_SYS).

This control flow from the kernel out to user-space is the reverse of the normal control flow in which user-space processes request services or information from the kernel. As such, there is no clear established practice of how to enable this communication.

There are in fact several quite different mechanisms to meet this sort of need in Linux and elsewhere. We will briefly review them here.

One of the simplest out-call mechanisms is to fork a new process to run a program in user-space. This mechanism is already in use by `kmod` to auto-load modules (by running `/sbin/modprobe`) and by subsystems such as `usb` to handle hot-plugging of devices by running `/sbin/hotplug`. In the first case the caller waits for `modprobe` to exit and then tries again. In the second, the caller assumes that process will just take care of everything.

Another mechanism is used by `autofs` to tell the automounting daemon that a particular mountpoint has been accessed. The daemon should respond by mounting the filesystem that should be there.

The mechanism used here is a simple pipe. The daemon creates a pipe and passes the write end down to the kernel via a system call. The `autofs` module then writes requests to this pipe as needed. This approach avoids the overhead of starting a new process which is especially significant as the automounter would normally cache a lot of information which could not be done if it was called independently for each mount.

Other mechanisms use sockets to communicate. `kerneld`, which was a precursor of `kmod`, used unix domain sockets to communicate requests. The current networking code can use (as I understand it) sockets in a special "netlink" address family for handling routing lookup and MAC address lookup.

SUN already has high quality authentication for their NFS server. Their kernel sends requests to user-space by using RPC message, though I don't know what sort of connection they use (loop-back socket or pipe I suspect).

As can be seen, there are many options and no clear winner.

The first choice is between starting a new process or sending a request over an established connection. The first is conceptually simpler and easier to prototype as a shell script could do much of the work needed. The second is probably more efficient, particularly where user-space can benefit by pre-loading and caching lots of information.

For RPC authentication, I have chosen to support both approaches, at least initially. The kernel will form each request into a list of textual words, and then pass them up either as arguments to a command, or as a message over an established connection to a daemon.

As to the question of what sort of connection to use, any of pipe, Internet socket or netlink socket could be argued to be appropriate for network related code such as this. I have chosen to use a pipe primarily because it is simple.

To allow for maximum flexibility, each different authentication flavour will be able to use a separate pipe for sending requests to user-space. Whenever a module needs to make a request, it will first check if a pipe is registered for that module and if so, will use it. It will then check if there is a pipe registered for the RPC module as a whole and will use that if available. Finally it checks to see if a path name has been registered for out-calls. If so, it is run. If none of these checks succeed, then the module will act as though the information is not available.

## 5.4 Comprehensive caching

When an out-call is made to gather some information, and the information is returned, it must be possible to keep that information in the kernel in case it is needed again soon. It should also be easy for the kernel to discard such information that has not been used for a long time as it is a waste of space. This calls for a comprehensive caching scheme.

Firstly, there are quite a variety of sorts of information that will need to be cached. The AUTH_SYS flavour will need to store a mapping from IP address to some form of host or host-group identifier.

The `nfsd` module will need to store a mapping from host-group identifier and filesystem to export options.

A LIPKEY module would need to store mappings from a textual username to information such as: a password, the user id and group id list, and the current session key.

Secondly, there will be a substantial amount of commonality among the features of these caches. Every cache element will need access and expiry times. Each element will need flags indicating whether it is valid, and whether an out-call has been made recently to try to fill in or revalidate the entry. Also, each cache will need to be regularly cleaned of any old data that has not been used or revalidated recently.

These two aspects suggest an object oriented/polymorphic approach to implementing the cache. A class could be defined that handles all the common aspects of the caches, and a sub-class could be created for each specific cache.

As the Linux kernel is written in "C" which does not have any explicit support for object oriented programming, we need to create our own. I will not go into the details here (you can always look at the code), but by embedding a common header in all the cached structures, by used a few slightly ugly casts, and by using `#define` to emulate a `C++` template, I have created an object oriented collection of caches for the RPC module and its client.

## 5.5 Request deferral

When an RPC (typically NFS) request arrives, the kernel can go about the task of unwrapping and interpreting it based on the information stored in various caches. If it finds that some information that is needed is not in a cache, it can make an out-call to user-space to get the information. But what then?

1. The thread could block waiting for the cache entry to be filled in. This would be simple but not ideal. There are a limited number of kernel threads that handle NFS requests and if several were blocked waiting for a cache item to fill, they would not be able to proceed with other requests.

2. The whole request could be passed out to user-space on the understanding that it will be passed back in (somehow) when the needed information is available. This would be fairly easy for the kernel, but would mean duplicating a lot of code in kernel and user space.

3. The request could be dropped on the assumption that the client will resend it. This certainly could work. However it would introduce noticeable delays. While it could take a long time for user-space to respond to a request (and hence the desire not to block) it will usually be quite quick, much quicker than the resend timeout for the client (especially if the client is using TCP). So this is not desirable.

4. The request could be filed away to be re-tried when the cache item has been filled in. This, while more complicated, is by far the preferred option and this is what we do.

When a request causes an out-call to be made for a particular cache item, the request is stored in a table of deferred requests, and is marked as being associated with that cache item.

If and when that cache item gets set by a system call from user-space, the table of deferred requests is checked to see if any are waiting on this item. If any are, they are rescheduled for processing. If requests remain in this table for too long, they get automatically discarded.

## 5.6 Immutable requests

If requests are to be deferred and retried, then it is important that the request does not get changed while it is being processed.

This may seem obvious, however it currently is not the case. RPC requests are encoded in XDR (eXternal Data Representation) format. Strings in this format are preceded by a character count and are not `nul` terminated. Linux prefers `nul` terminated strings and when a string is found in an RPC request, it is (sometimes) copied backwards a little bit (over-writing the length) and a `nul` is appended. This effectively corrupts the request so that a re-try on it would get confused.

We could take a copy of a request before processing it, but that would be a waste of time in most cases.

There are two sorts of strings that appear in NFS requests: path name components and symlink contents.

Path name components can be handled as an unterminated string with a length thanks to the `lookup_one_len` call, and since 2.4.11 kNFSd has not corrupted requests because of path name components.

However symlink contents (as used when creating a symlink) must be `nul` terminated when passed to the filesystem. In NFSv2, the symlink content will always be followed by a `nul` byte (the high byte of a 16 bit number stored in a 32 bit field) and in NFSv3 the symlink content always comes at the end of a request, and so appending a `nul` should be fairly straight-forward.

To stop the packet corruption that currently happens when extracting symlink names, we pass the content into `nfsd` as an unterminated string and a length, and `nfsd` will check if it is, in fact, `nul` terminated before passing it to the filesystem. If it is not `nul` terminated, it will be copied to a temporary buffer.

However this is only half the story. The other half of the story involves encrypted requests.

RPCSEC/GSS allows requests to be encrypted for privacy, so they will need to be decrypted before being passed to `nfsd`. For space management reasons it might be best to allow the request to be decrypted in-place. However this would make replay of deferred requests awkward. It might be possible to flag a deferred request as being decrypted so as to avoid the confusion. However for now we just require that requests are decrypted into a separate buffer so that the original remains unchanged.

## 5.7 Backwards compatibility

As can be imagined, the internal detail of how export information is accessed in the kernel is changing dramatically. However it would be nice if existing user-level support tools (nfs-utils) continue to work, though they would not expect to have access to the new functionality.

This can be achieved reasonably well by supporting the old system calls for setting export information and having them create cache entries with an infinite life time. Combining this with the rule that if no out-call mechanism has been explicitly set then the lookup fails means that the caches can look a lot like the previous mechanism.

One piece of functionality supported by the old setup which is awkward to provide in the the scheme is that of telling the kernel to forget about a particular client. However the current user-space tools never call that function, so not supporting it should not cause any problems.

# 6 Playground: random thoughts

While designing and developing this code a number of issues relating to kernel design and development had to be addressed and some of these issues could be of general interest, so I thought I would share them here.

The first relates to the naming of kernel objects, and the second to patterns for kernel code.

## 6.1 Naming of kernel objects

This issue came up when considering how best to handle out-calls for passing requests from the kernel to user-space. In each case where requests are passed out, the most important piece of information that is passed out is the name of something in the kernel.

It might be a device which has been plugged or unplugged. It might be a (potential) mountpoint that has been accessed. It might be an entry in a cache that needs to be updated. It might be a device that some process is trying to access.

The request also contains, or implies, a verb which indicates what needs to be done with the named object, but the name of the object seems most significant.

If we had a uniform hierarchical naming scheme for all kernel objects, then we could provide a uniform mechanism for handling out-calls.

If the kernel names appeared in a filesystem, as they should, then the out-call process could probably be hooked into the directory change notification mechanism quite effectively.

If a process wanted to manage all changes in one part of the kernel namespace, then it requests directory notification for that subtree. If an out-call is required and there are no processes waiting for relevant directory notifications, then a program can be run giving the full kernel name for the object in question. It can then find and run the right tool to deal with the issue.

While this is, as yet, not a very well studied idea, I believe it holds promise for introducing a lot of consistency and transparency into some aspects of kernel design.

## 6.2 Kernel Design Patterns

Design Patterns have been much talked about in some circles as a means for codifying and then sharing certain software techniques, thus allowing software engineers to make use of the work of others to improve both the speed and the correctness of the development process.

Having access to such patterns is particularly valuable when working with complex or unfamiliar areas.

Developing for the Linux kernel requires proper handling of concurrency issues introduced by Multiprocessor (SMP) machines. It also requires very strict attention to managing memory allocation and release correctly. There is no room for shortcuts (such as having a server re-execute itself periodically, and thus automatically free up any unused memory).

For this reason, it would seem that the Linux kernel would be a fruitful place to seek out and extract important design patterns. Having such design patterns would quite possibly have been helpful to me when I set out to re-implement the authentication infrastructure for kNFSd.

The area that was most challenging was in managing the caches in which the various modules can store information. The most problematic issue was how to manage the changing of information in a cache item given that it is quite possible that several other threads are accessing a cache item at the same time that we are trying to modify it.

One obvious approach would be to require the element, or the whole cache, to be locked whenever changing or accessing an element. In a cache that is read-mostly, this would cause a lot of un-necessary locking. So the question is, how to manage updates to items without requiring a lock for every read.

In exploring this question, I noticed a similarity to the dcache and icache in the Linux VFS layer. A valuable observation from the handling of those caches is that a dentry (an object in the dcache which represents a name in a directory in a filesystem) can either be negative, meaning that there is no inode associated with the name, or it can be positive and point to a particular inode. The only change that is allowed is that a negative dentry can be made positive by associating it with an inode. Once that association has been made it can never be broken. The dentry can never be associated with a different inode, nor can the dentry be made negative again.

When a file is renamed, the inode is not moved from one dentry to another as ou might expect. Rather the name of one dentry is changed to reflect the rename. Thus if a thread holds a reference to a dentry, it need never worry about the inode possibly changing.

The general rule, which could be included in a Design Pattern, seems to be:

1. If a cache item is negative (purports to contain no information) then it is safe to make it positive providing that change appears atomic. Any reader should check a single field to see if the item is negative or not, and should not reference other fields if negative. A writer should set this single field last in the process of validating the cache item.

2. If a cache item, when positive, contains atomic fields that are independent of all other fields, and are not references to other reference counted objects (i.e. references to a slave cache), then they can be changed without concern for readers.

3. If a cache item contains fields that are not atomic, that are dependent on other fields, or that are references to external objects, then they must not be changed once set. Instead of changing them, a new cache item should be created and atomically swapped into the cache in place of the old item.

Had I been fully aware of this design pattern when I started writing the code, I would have made the generic cache-lookup template take a replacement element which, if set, gets atomically swapped into the cache with the correct index. I did eventually do this, but not until very close to the end of the development process.

# 7 Groundwork: current state of the code

At the time of writing most of the core code has been written and some of it tested.

Cache misses do cause out-calls, at least when forking a process, and cache entries can be set from user-space.

The deferring and replaying of requests has not yet been tested. No user-level code has been written (or even designed) to interact with this functionality.

The code, which grew in spurts over about 9 months, needs to be reviewed and re-organised. This will probably happen as part of the process of dividing the code up into minimal functional parts to be submitted to Linus.

The current code can be found by hunting around in

`http://www.cse.unsw.edu.au/~neilb/patches/linux-stable/`

Look for patches with names like `NfsAuthWIP`.

# 8 Ground Swell - Other things that need doing

More as an appendix than as a final section, here is a list of some other changes that are planned or underway for kNFSd. Some should get into 2.4. Others may only be in 2.5/2.6.

**Get most of kNFSd out from under the Big Kernel Lock** When Linux first started supporting SMP machines, everything in the kernel was placed under the Big Kernel Lock so that code which assumed it was the only thing running would still work. Bit by bit code has been audited, fixed, and moved out of the BKL. kNFSd has not made that transition yet.

There are a few data structures used by kNFSd which need SMP protection (the reply cache, the read-ahead cache, etc). These will be given their own fine grained locking and then the BKL will be removed. Patches to do this are already available but need a bit more testing.

**Stop relying on device numbers to identify filesystems in the filehandle** kNFSd currently uses the device number (major and minor) of the device holding a filesystem to identify that filesystem, particularly in filehandles. This is problematic, largely because device numbers are not always stable across reboots.

Patches are almost ready for kernel and nfs-utils which allow the admin to specify a number to be used to identify the filesystem instead of relying on the device number. This trades a bit of admin overhead for correctness.

**Improve interface with filesystems** kNFSd needs to interact with filesystems to find the file (dentry) identified by a given filehandle, and to generate a filehandle for a dentry.

The current interface between kNFSd and filesystems is adequate for some filesystems, but not for all and is not as efficient as it could be.

An improved interface is available in a patch which can handle filesystems that set `d_ops` properly and can allow the filesystem to lookup the name for a given inode, rather than having kNFSd going through the readdir interface.

**Support TCP connections to kNFSd** kNFSd (or more accurately, the RPC layer beneath it) can handle TCP connections at the moment (with a one line change before compiling to enable it) but there are possibilities of denial of service and incorrect behaviour during network congestion.

I have a collection of patches that address most of these issues, but they have not received much testing yet.

**Improve handling of export-point crossing** This issue addresses the question of what happens when a LOOKUP request finds an object (directory) that is under a different export-point than the export-point which exported the original directory.

There are two flavours of this, one where the new directory is on a different filesystem, and one where it is on the same filesystem.

NFS normally does not support crossing filesystems. If a client wants to access two different filesystems on the server, it needs to mount them separately. However some people find it administratively convenient to only mount the parent filesystem and to get all the descendant filesystems accessible automatically. There are issues with doing this as there will be no guarantee of inode number uniqueness, but it can be made to work fairly well.

Linux kNFSd supports a `nohide` export option which allows the server to cross mountpoints. This currently is not supported very well by nfs-utils, and not really supported at all by the new authentication infrastructure. Current out-calls to request export information only make the request based on a fragment on the filehandle. We need also to be able to make requests based on a path-name.

kNFSd does not currently allow both a directory and an ancestor of that directory on the same filesystem to be exported to the same client. This is because it could lead to ambiguity as to how the lower directory is really exported. However this functionality is often wanted by sysadmins, for example to export "/" read-only, but "/tmp" read-write.

For this to be supportable, kNFSd needs to know, while walking around the filetree, when it hits an export-point. This requires, at the very least, that dentrys be flagged to say if they are export-points.

More work is needed on this.

**Fix Bugs** There have been surprisingly few real bugs in kNFSd lately, though I suspect I can fix that with some of the changes described above. The only outstanding ones that I can think of relate to exporting FAT filesystems and are fairly minor and will be fixed soon.

# 9 Conclusion

There is still a lot of work to be done in order to provide secure authentication for Linux kNFSd. However with the changes described here, we are well on the way. The enxt steps will be to implement the RPCSEC/GSS layer, the LIPKEY layer, and SPKM-3 on which LIPKEY depends.