Reprinted from the

Proceedings of the Linux Symposium

Volume One

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

NFSv4 and `rpcsec_gss` for linux

J. Bruce Fields
University of Michigan
bfields@umich.edu

Abstract

The 2.6 Linux kernels now include support for version 4 of NFS. In addition to built-in locking and ACL support, and features designed to improve performance over the Internet, NFSv4 also mandates the implementation of strong cryptographic security. This security is provided by rpcsec_gss, a standard, widely implemented protocol that operates at the rpc level, and hence can also provide security for NFS versions 2 and 3.

1 The rpcsec_gss protocol

The rpc protocol, which all version of NFS and related protocols are built upon, includes generic support for authentication mechanisms: each rpc call has two fields, the credential and the verifier, each consisting of a 32-bit integer, designating a "security flavor," followed by 400 bytes of opaque data whose structure depends on the specified flavor. Similarly, each reply includes a single "verifier."

Until recently, the only widely implemented security flavor has been the auth_unix flavor, which uses the credential to pass uid's and gid's and simply asks the server to trust them. This may be satisfactory given physical security over the clients and the network, but for many situations (including use over the Internet), it is inadequate.

Thus rfc 2203 defines the rpcsec_gss protocol, which uses rpc's opaque security fields to carry cryptographically secure tokens. The cryptographic services are provided by the GSS-API ("Generic Security Service Application Program Interface," defined by rfc 2743), allowing the use of a wide variety of security mechanisms, including, for example, Kerberos.

Three levels of security are provided by rpcsec_gss:

1. Authentication only: The rpc header of each request and response is signed.
2. Integrity: The header and body of each request and response is signed.
3. Privacy: The header of each request is signed, and the body is encrypted.

The combination of a security level with a GSS-API mechanism can be designated by a 32-bit "pseudoflavor." The mount protocol used with NFS versions 2 and 3 uses a list of pseudoflavors to communicate the security capabilities of a server. NFSv4 does not use pseudoflavors on the wire, but they are still useful in internal interfaces.

Security protocols generally require some initial negotiation, to determine the capabilities of the systems involved and to choose session keys. The rpcsec_gss protocol uses calls with procedure number 0 for this purpose. Normally such a call is a simple "ping" with no side-effects, useful for measuring round-trip

latency or testing whether a certain service is running. However a call with procedure number 0, if made with authentication flavor rpcsec_gss, may use certain fields in the credential to indicate that it is part of a context-initiation exchange.

2 Linux implementation of rpcsec_gss

The Linux implementation of rpcsec_gss consists of several pieces:

1. Mechanism-specific code, currently for two mechanisms: krb5 and spkm3.
2. A stripped-down in-kernel version of the GSS-API interface, with an interface that allows mechanism-specific code to register support for various pseudoflavors.
3. Client and server code which uses the GSS-API interface to encode and decode rpc calls and replies.
4. A userland daemon, gssd, which performs context initiation.

2.1 Mechanism-specific code

The NFSv4 RFC mandates the implementation (though not the use) of three GSS-API mechanisms: krb5, spkm3, and lipkey.

Our krb5 implementation supports three pseudoflavors: krb5, krb5i, and krb5p, providing authentication only, integrity, and privacy, respectively. The code is derived from MIT's Kerberos implementation, somewhat simplified, and not currently supporting the variety of encryption algorithms that MIT's does. The krb5 mechanism is also supported by NFS implementations from Sun, Network Appliance, and others, which it interoperates with.

The Low Infrastructure Public Key Mechanism ("lipkey," specified by rfc 2847), is a public key mechanism built on top of the Simple Public Key Mechanism (spkm), which provides functionality similar to that of TLS, allowing a secure channel to be established using a server-side certificate and a client-side password.

We have a preliminary implementation of spkm3 (without privacy), but none yet of lipkey. Other NFS implementors have not yet implemented either of these mechanisms, but there appears to be sufficient interest from the grid community for us to continue implementation even if it is Linux-only for now.

2.2 GSS-API

The GSS-API interface as specified is very complex. Fortunately, rpcsec_gss only requires a subset of the GSS-API, and even less is required for per-packet processing.

Our implementation is derived by the implementation in MIT Kerberos, and initially stayed fairly close the the GSS-API specification; but over time we have pared it down to something quite a bit simpler.

The kernel gss interface also provides APIs by which code implementing particular mechanisms can register itself to the gss-api code and hence can be safely provided by modules loaded at runtime.

2.3 RPC code

The RPC code has been enhanced by the addition of a new rpcsec_gss mechanism which authenticates calls and replies and which wraps and unwraps rpc bodies in the case of integrity and privacy.

This is relatively straightforward, though somewhat complicated by the need to handle discontiguous buffers containing page data.

Caches for session state are also required on both client and server; on the client a preexisting rpc credentials cache is used, and on the server we use the same caching infrastructure used for caching of client and export information.

2.4 Userland daemon

We had no desire to put a complete implementation of Kerberos version 5 or the other mechanisms into the kernel. Fortunately, the work performed by the various GSS-API mechanisms can be divided neatly into context initiation and per-packet processing. The former is complex and is performed only once per session, while the latter is simple by comparison and needs to be performed on every packet. Therefore it makes sense to put the packet processing in the kernel, and have the context initiation performed in userspace.

Since it is the kernel that knows when context initiation is necessary, we require a mechanism allowing the kernel to pass the necessary parameters to a userspace daemon whenever context initiation is needed, and allowing the daemon to respond with the completed security context.

This problem was solved in different ways on the client and server, but both use special files (the former in a dedicated filesystem, rpc_pipefs, and the latter in the proc filesystem), which our userspace daemon, gssd, can poll for requests and then write responses back to.

In the case of Kerberos, the sequence of events will be something like this:

1. The user gets Kerberos credentials using kinit, which are cached on a local filesystem.

2. The user attempts to perform an operation on an NFS filesystem mounted with krb5 security.

3. The kernel rpc client looks for the a security context for the user in its cache; not finding any, it does an upcall to gssd to request one.

4. Gssd, on receiving the upcall, reads the user's Kerberos credentials from the local filesystem and uses them to construct a null rpc request which it sends to the server.

5. The server kernel makes an upcall which passes the null request to its gssd.

6. At this point, the server gssd has all it needs to construct a security context for this session, consisting mainly of a session key. It passes this context down to the kernel rpc server, which stores it in its context cache.

7. The server's gssd then constructs the null rpc reply, which it gives to the kernel to return to the client gssd.

8. The client gssd uses this reply to construct its own security context, and passes this context to the kernel rpc client.

9. The kernel rpc client then uses this context to send the first real rpc request to the server.

10. The server uses the new context in its cache to verify the rpc request, and to compose its reply.

3 The NFSv4 protocol

While rpcsec_gss works equally well on all existing versions of NFS, much of the work on rpcsec_gss has been motivated by NFS version 4, which is the first version of NFS to make rpcsec_gss mandatory to implement.

This new version of NFS is specified by rfc 3530, which says:

"Unlike earlier versions, the NFS version 4 protocol supports traditional file access while integrating support for file locking and the mount protocol. In addition, support for strong security (and its negotiation), compound operations, client caching, and internationalization have been added. Of course, attention has been applied to making NFS version 4 operate well in an Internet environment."

Descriptions of some of these features follow, with some notes about their implementation in Linux.

3.1 Compound operations

Each rpc request includes a procedure number, which describes the operation to be performed. The format of the body of the rpc request (the arguments) and of the reply depend on the program number. Procedure 0 is reserved as a no-op (except when it is used for rpcsec_gss context initiation, as described above).

The NFSv4 protocol only supports one non-zero procedure, procedure 1, the compound procedure.

The body of a compound is a list of operations, each with its own arguments. For example, a compound request performing a lookup might consist of 3 operations: a PUTFH, with a filehandle, which sets the "current filehandle" to the provided filehandle; a LOOKUP, with a name, which looks up the name in the directory given by the current filehandle and then modifies the current filehandle to be the filehandle of the result; a GETFH, with no arguments, which returns the new value of the current filehandle; and a GETATTR, with a bitmask specifying a set of attributes to return for the looked-up file.

The server processes these operations in order, but with no guarantee of atomicity. On encountering any error, it stops and returns the results of the operations up to and including the operation that failed.

In theory complex operations could therefore be done by long compounds which perform complex series of operations.

In practice, the compounds sent by the Linux client correspond very closely to NFSv2/v3 procedures—the VFS and the POSIX filesystem API make it difficult to do otherwise—and our server, like most NFSv4 servers we know of, rejects overly long or complex compounds.

3.2 Well-known port for NFS

RPC allows services to be run on different ports, using the "portmap" service to map program numbers to ports. While flexible, this system complicates firewall management; so NFSv4 recommends the use of port 2049.

In addition, the use of sideband protocols for mounting, locking, etc. also complicates firewall management, as multiple connections to multiple ports are required for a single NFS mount. NFSv4 eliminates these extra protocols, allowing all traffic to pass over a single connection using one protocol.

3.3 No more mount protocol

Earlier versions of NFS use a separate protocol for mount. The mount protocol exists primarily to map path names, presented to the server as

strings, to filehandles, which may then be used in the NFS protocol.

NFSv4 instead uses a single operation, PUTROOTFH, that returns a filehandle; clients can then use ordinary lookups to traverse to the filesystem they wish to mount. This changes the behavior of NFS in a few subtle ways: for example, the special status of mounts in the old protocol meant that mounting `/usr` and then looking up `local` might get you a different object than would mounting `/usr/local`; under NFSv4 this can no longer happen.

A server that exports multiple filesystems must knit them together using a single "pseudofilesystem" which links them to a common root.

On Linux's nfsd the pseudofilesystem is a real filesystem, marked by the export option "fsid=0". An adminstrator that is content to export a single filesystem can export it with "fsid=0", and clients will find it just by mounting the path "/".

The expected use for "fsid=0", however, is to designate a filesystem that is used just a collection of empty directories used as mountpoints for exported filesystems, which are mounted using `mount ---bind`; thus an administrator could export `/bin` and `/local/src` by:

```
mkdir -p /exports/home
mkdir -p /exports/bin/
mount --bind /home /exports/home
mount --bind /bin/ /exports/bin
```

and then using an exports file something like:

```
/exports *.foo.com(fsid=0,crossmnt)
/exports/home *.foo.com
/exports/bin *.foo.com
```

Clients in `foo.com` can then mount `server.foo.com:/bin` or `server.foo.com:/home`. However the relationship between the original mountpoint on the server and the mountpoint under `/exports` (which determines the path seen by the client) is arbitrary, so the administrator could just as well export `/home` as `/some/other/path` if desired.

This gives maximum flexibility at the expense of some confusion for adminstrators used to earlier NFS versions.

3.4 No more lock protocol

Locking has also been absorbed into the NFSv4 protocol. In addition to advantages enumerated above, this allows servers to support mandatory locking if desired. Previously this was impossible because it was impossible to tell whether a given read or write should be ordered before or after a lock request. NFSv4 enforces such sequencing by providing a stateid field on each read or write which identifies the locking state that the operation was performed under; thus for example a write that occurred while a lock was held, but that appeared on the server to have occurred after an unlock, can be identified as belonging to a previous locking context, and can therefore be correctly rejected.

The additional state required to manage locking is the source of much of the additional complexity in NFSv4.

3.5 String representations of user and group names

Previous versions of NFS use integers to represent users and groups; while simple to handle, they can make NFS installations difficult to manage, particularly across adminstrative domains. Version 4, therefore, uses string names of the form `user@domain`.

This poses some challenges for the kernel im-

plementation. In particular, while the protocol may use string names, the kernel still needs to deal with uid's, so it must map between NFSv4 string names and integers.

As with rpcsec_gss context initation, we solve this problem by making upcalls to a userspace daemon; with the mapping in userspace, it is easy to use mechanisms such as NIS or LDAP to do the actual mapping without introducing large amounts of code into the kernel. So as not to degrade performance by requiring a context switch every time we process a packet carrying a name, we cache the results of this mapping in the kernel.

3.6 Delegations

NFSv4, like previous versions of NFS, does not attempt to provide full cache consistency. Instead, all that is guaranteed is that if an open follows a close of the same file, then data read after the open will reflect any modifications performed before the close. This makes both open and close potentially high latency operations, since they must wait for at least one round trip before returning–in the close case, to flush out any pending writes, and in the open case, to check the attributes of the file in question to determine whether the local cache should be invalidated.

Locks provide similar semantics—writes are flushed on unlock, and cache consistency is verified on lock—and hence lock operations are also prone to high latencies.

To mitigate these concerns, and to encourage the use of NFS's locking features, delegations have been added to NFSv4. Delegations are granted or denied by the server in response to open calls, and give the client the right to perform later locks and opens locally, without the need to contact the server. A set of callbacks is provided so that the server can notify the client when another client requests an open that would confict with the open originally obtained by the client.

Thus locks and opens may be performed quickly by the client in the common case when files are not being shared, but callbacks ensure that correct close-to-open (and unlock-to-lock) semantics may be enforced when there is contention.

To allow other clients to proceed when a client holding a delegation reboots, clients are required to periodically send a "renew" operation to the server, indicating that it is still alive; a client that fails to send a renew operation within a given lease time (established when the client first contacts the server) may have all of its delegations and other locking state revoked.

Most implementations of NFSv4 delegations, including Linux's, are still young, and we haven't yet gathered good data on the performance impact.

Nevertheless, further extensions, including delegations over directories, are under consideration for future versions of the protocol.

3.7 ACLs

ACL support is integrated into the protocol, with ACLs that are more similar to those found in NT than to the POSIX ACLs supported by Linux.

Thus while it is possible to translate an arbitrary Linux ACL to an NFS4 ACL with nearly identical meaning, most NFS ACLs have no reasonable representation as Linux ACLs.

Marius Eriksen has written a draft describing the POSIX to NFS4 ACL translation. Currently the Linux implementation uses this mapping, and rejects any NFS4 ACL that isn't exactly in the image of this mapping. This en-

sures userland support from all tools that currently support POSIX ACLs, and simplifies ACL management when an exported filesystem is also used by local users, since both nfsd and the local users can use the backend filesystem's POSIX ACL implementation. However it makes it difficult to interoperate with NFSv4 implementations that support the full ACL protocol. For that reason we will eventually also want to add support for NFSv4 ACLs.

4 Acknowledgements and Further Information

This work has been sponsored by Sun Microsystems, Network Appliance, and the Accelerated Strategic Computing Initiative (ASCI). For further information, see `www.citi.umich.edu/projects/nfsv4/`.