RPCSEC_GSS for the Linux Kernel

William A. Adamson

Center for Information Technology Integration
University of Michigan
Ann Arbor

{andros}@citi.umich.edu

1 Introduction

An implementation of the RPCSEC_GSS Protocol (RFC 2203) is a required piece of the NFS version 4 Open Source Reference Implementation project sponsored by Sun Microsystems. This document describes the initial implementation.

Note that this project uses the following RFC's, and that to chase down proper behavior requires jumping between the behaviors therein described.

- *RFC 1964 The Kerberos Version 5 GSS-API Mechanism*
- *RFC 2078 Generic Security Service Application Program Interface, Version 2*
- *RFC 2203 RPCSEC_GSS Protocol Specification*

2 RPCSEC_GSS Architecture

Kerberos v5 is one of the required mechanism for the NFSv4 RPCSEC_GSS implementation. This is convenient, because Kerberos v5 from MIT includes a userlevel GSS implementation. There is however, no Kerberos implementation for the Linux kernel. Following Sun Microsystems lead, we implemented a userlevel daemon called GSSD to call userland Kerberos v5 gss functionality, and added the RPCSEC_GSS infrastructure to the Linux kernel Sun ONC implementation. This architecture allows us to get early functionality, and decide on a feature by feature basis what portions of the Kerberos v5 code (if any) belongs in the Linux kernel.

Our first implementation left all the gss work to GSSD, and used the RPCSEC_GSS additions to the Linux Kernel Sun ONC RPC simply as a transport. Although this implementation performed Kerberos v5 mutual authentication over and RPCSEC_GSS channel, it was not fully RFC 2203 compliant as it did not perform the per packet hashing and verification required by RPCSEC_GSS.

The hashing and verification is needed for every packet with the RPC_AUTH_GSS authentication flavor. We felt that an upcall to GSSD for each packet to perform hashing and verification would be too great a performance hit, so we decided to add this functionality to the Linux kernel.

This decision means adding code to the Sun ONC RPC kernel implementation that can do the following:

- *Switch on GSS security mechanisms.*
- *Import and cache gss_context on both the client and server*
- *Create and verify GSS tokens*
- *Understand Kerberos v5 GSS token payloads.*
- *Call kernel crypto routines.*

The rest of this document details the changes and additions to the Linux kernel Sun ONC RPC that implement the above features. The changes are based on the Linux 2.4.4 kernel ../net/sunrpc and ../include/linux/sunrpc code. We based our userland work on MIT's krb5-1.2.1 Kerberos v5 code.

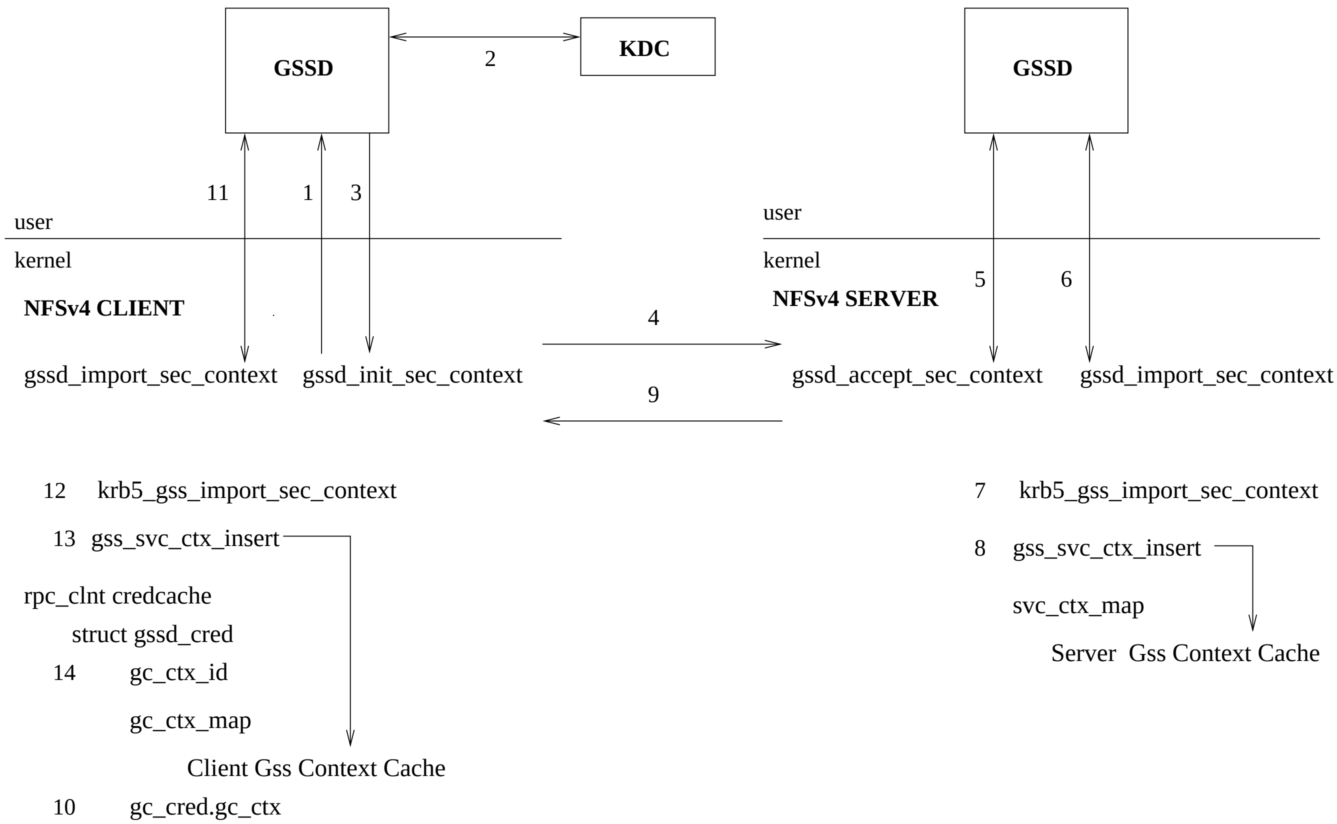

Figure 1: **GSS Context Creation and Caching.** This figure shows the process of obtaining a GSS Context and caching it in the Linux Kernel

3 Sun ONC RPC Interface to RPCSEC_GSS

Our first task was to examine and improve MIT's krb5-1.2.1 rpcsec_gss implementation, and bring it to compliance with RFC 2203. This is mostly done. We then based our Linux kernel port on this code.

The entry point for creating a gss_context is the client rpcauth_create() call with the RPC_AUTH_GSS flavor. In the NFSv4 client, this occurs as a result of a security negotiation with the server, or at client creation via the rpc_create_client() call.

On the client, there are two changes to *auth.c*: the addition of authgss_ops to the struct rpc_authops and reverting to a previous interface to rpcauth_lookup_credcache.

This is the original 2.4.4 interface which will not work for rpcsec_gss which needs the struct task.

```
static struct rpc_cred *
rpcauth_lookup_credcache(
          struct rpc_auth *auth,
          int taskflags)
```

Here is the changed interface:

```
static struct rpc_cred *
rpcauth_lookup_credcache(
          struct rpc_task *task)
```

On the server, the only change to the existing Sun ONC RPC code is an entry in *svcauth.c* authtab for the gss flavor, and a gss cache initialization function in *stats.c*.

4 GSSD

GSSD is an Sun ONC RPC service that runs on the localhost ethernet interface on a privileged port and demands root uid/gid values. It is compiled against the MIT krb5-1.2.1 code, and acts as a gss function translator, receiving gss function call requests from NFSv4 which it runs and returns the result.

Referring to Figure 1 for an example, the NFSv4 client will initiate a gss connection with by calling gssd_init_sec_context (step 1) which communicates with GSSD over its RPC interface. GSSD then calls gss_init_sec_context which, for the Kerberos v5 mechanism, contacts the KDC (step 2). GSSD then bundles the gss_init_sec_context results and returns an RPC to the NFSv4 client (step 3). The NFSv4 client packages the gssd_init_sec_context result in an rpcsec_gss null rpc which is sent to the server (step 4). The server performs similar steps calling gssd_accept_sec_context with data from the step 4 null rpc, and returns the results to the client in step 9.

5 Switch on GSS Security Mechanisms

The Kerberos v5 gss code base includes a mech_glue sub-directory that contains code to switch on security mechanisms, of which Kerberos v5 is one. The mech_glue code exports the full set of gss functions. These functions add a mechanism OID to data structures such as a gss_oid data structure and then uses this mechanism OID to locate the mechanism specific gss function. By default, the mech_glue code is not used, and a mechanism OID of zero is interpreted as the Kerberos OID by the MIT code. We have debugged and added to the mech_glue layer which is used by GSSD.

The Linux kernel implementation will need to perform the same task, and I have added a portion of the mech_glue layer to the kernel rpc code base (*gss_union.c*). I have yet to implement the function table lookup piece of the mech_glue layer in the kernel.

6 Import and Cache gss_context

Referring to Figure 1, after steps 1 through 5 have occurred, the negotiated context's reside in the respective GSSDs. The context contains information necessary to perform crypto such as negotiated algorithms. The context needs to be imported into the kernel so they can be used.

Context importation is done on both the NFSv4 client and server with the gssd_import_sec_context upcall to GSSD (step 6). GSSD then calls gss_export_sec_context which for the Kerberos v5 mechanism calls krb5_export_sec_context. This results in the context being removed from the userland GSSD Kerberos v5 context cache. GSSD then returns the exported context in the gssd_import_sec_context call. step 6 is performed only after the gssd_accept_context call (step 5) succeeds.

One caveat - the Kerberos v5 code returns the context in a form similar to XDR or ASN1 in that it's 'flattened' - but it's in their special 'serialized' form. I chose to import the serialization decoding code into the Linux kernel so that I could decode the Kerberos v5 gss context into a readable form (*gss_k5ser.c,gss_k5serialize.c* and *gss_intern_ctx.c*). Step 7 and step 12 call krb5_gss_import_sec_context to de-serialize the context.

Step 8 creates a gss context cache entry (struct gss_ctx_cacheent), and inserts the new context into the server gss context cache. On the NFSv4 server, the global gss_svc_ctx_mapping struct holds gss_ctx_cacheents hashed by cacheent pointer. The gss_ctx_cacheent pointer is put in the gc_ctx field of the struct rpc_gss_cred in the return null rpc (step 9) and is used by the client to reference the server gss_context in future communications.

```
auth_gss.h:

/* server gss context cache */
struct gss_svc_ctx_mapping {
  rwlock_t        lock;
  list_head gss_ctx_cache[GSS_HASH_SIZE];
};

struct gss_ctx_cacheent {
  GSS_OID     gcc_mech;
  u32     gcc_qop;
  void    *gcc_ctx;

/* fields after this point are private,
 * for use by the gss cache */
  atomic_t    gcc_refcount;
  list_head  gss_ctx_cache;
};
```

The server gss caching code is in *gss_svc_cache.c*.

The client keeps all gss cache info in struct gss_cred which is stored in the rpc cred cache, rpc_clnt-¿cli_auth-¿aui_credcache. If the status of the gss_accept_sec_context call, step 9, indicates success, the pointer to the server side gss_ctx_cacheent is copied from the on the wire struct rpc_gss_cred gc_ctx into the gss_cred.gc_cred.gc_ctx (step 10).

The NFSv4 client then calls gssd_import_sec_context (step 11) and krb5_gss_import_sec_context (step 12) to obtain it's negotiated gss_context. The context is stored in a gss_ctx_cacheent in the client gss context cache (step 13), and a pointer to the newly created client gss_ctx_cacheent is stored in the gss_cred gc_ctx_id field.

The client gss cache is implemented as a list of struct gss_ctx_cacheent's hanging off the gss_cred stuct with a pointer to the current gss_context stored in gss_cred.gc_ctx_id. This design allows for the fact that there may be multiple gss contexts needed for a single user, for example, when a server exports two file systems, one with Kerberos v5 security and one with LIPKEY security.

```
auth_gss.h:

/* client gss context cache */
struct gss_ctx_mapping {
  rwlock_t                lock;
  struct list_head        gss_ctx_cache;
};


struct gss_cred {
  struct rpc_cred         gc_base;
  u32                     gc_established;
  GSS_BUFFER_T            gc_wire_verf;
  GSS_BUFFER_T            gc_service_name;
  GSS_CTX_ID_T            gc_ctx_id;
  struct gss_ctx_mapping  gc_ctx_map;
  u32                     gc_win;
  struct rpc_gss_cred     gc_cred;
};
```

The client gss caching code is in *gss_cache.c*.

7 Create and Verify GSS tokens

All gss communications are wrapped in gss token headers and followed by mechanism specific payloads. This includes the gss_verify_mic and gss_get_mic used to hash portions of the gss header to verify message integrity. Note that this verification is part of each gss data message and is separate from the data integrity calculations.

The gss token header is ASN1 encoded, and it's construction is therefore non-trivial. I imported the Kerberos v5 routines to create and verify the token headers (*gss_generic_token.c*).

gss_mic.c contains the mechanism independent gss_get_mic and gss_verify_mic function calls, which hard code the calls to the Kerberos v5 specific calls kg_seal (in *gss_k5seal.c*), and kg_unseal (in *gss_k5unseal.c*). These Kerberos v5 calls construct the gss token payload. They are also used for data integrity and privacy tokens. They have code that switches on algorithms stored in the gss context. I've implemented the Kerberos v5 default algorithms which use md5 and des cbc. The md5 algorithm is used to calculate and verify the gss header checksums.

An important part of the gss protocol is the correct creation of sequence numbers. The functions in *gss_generic_ordering.c* perform this task. The Kerberos v5 token payload encrypts the calculated sequence number using des cbc. These functions exist in *gss_k5util_seqnum.c*.

8 Kerberos 5 Mechanism and the Linux CryptoAPI

I use the Linux kernel crypto patch, currently cryptoapi-2.4.10.diff. The cryptoapi uses strings to locate digests and ciphers. Kerberos v5 gss sends integers as algorithm identifiers. I constructed static lists that map Kerberos v5 algorithm identifiers to linux crypto names, and functions to locate and lookup buffer lengths (*gss_util_crypto.c*).

I then replace the (three or so deep) Kerberos v5 crypto interface with calls into the Linux kernel cryptoapi. *gss_k5encrypt.c*, *gss_k5decrypt.c*, and *gss_k5hash_md5.c* all contain pieces of this code.