GSS Policy with Kerberos 5 Mechanism HLD

Eric Mei

February 7, 2008

1 Requirements

- Implement GSS policy for Secure PTLRPC framework.
- Implement Kerberos 5 mechanism for GSS policy.

Note: the overall structure is borrowed from NFSv4 implementation in Linux.

2 Functional Specification

2.1 GSS policy

The structure has been discussed in “security API HLD”. As to the general Lustre security API framework, the GSS is one policy implementation, realize all the required API functions. GSS itself is also an general dispatch layer, it prescribed a set of GSSAPI functions and dispatch function calls to correct underlying mechanisms, and provide general support for all of them.

On the other hand, GSS is equivalent of standard user-space GSSAPI library, but implemented in kernel, and with some modifications (only a subset of the functionality is needed, and some change on interface) in favor of kernel environment.

2.1.1 interface to security API

GSS implemented all prescribed API functions, include policy functions on client & server side, and context functions. It will implemente 2 kind of data protection level: *integrity* and *privacy*.

Beside the upper level lustre message data, GSS policy should be able to extend to pack arbitrary extra data into request and reply message, transparently to PTLRPC layer. Those extra data will also under protection of integrity or privacy.

2.1.2 support for mechanisms

GSS policy layer will provide general support for all underlying mechanisms:

- A unified pack/unpack method as well as internal buffer layout & arrangement, respectively to integrity mode and privacy mode, will be used for all RPCs.
- A unified on-wire data format will be used for all RPCs.
- A mechanism of protection from replay attack will be implemented in GSS policy layer, independent to any underlying mechanisms.
- GSS policy layer will handle all interactions with user space daemons, if needed, for all mechanisms.
- Secure reply ACK might be able to implemented in GSS policy layer, as an extra data payload.

2.1.3 security context negotiation

There's 2 user space daemons which help security context negotiation: *lgssd* running on client side, and *lsvcgssd* running on server side.

lgssd will accept request from kernel, authenticate with authentication servers (e.g. KDC in kerberos 5 enviroment), prepare gss negotiation token, verify reply token, and generate client side context for kernel.

lsvcgssd will accept and parse gss negotiation token, prepare reply token, and generate server side context for kernel.

Currently lgssd and lsvcgssd only support Kerberos 5 mechanism, but they should be able to be extended to support other mechanisms in the future.

For the message exchange during negotiation, user daemon only pass to kernel by ioctl(), and kernel will responsible to sent the message using normal PTLRPC.

2.2 Kerberos 5 mechanism

Kerberos 5 is one backend mechanism of GSS policy, realize the prescribed GSS-API functions. Driven by GSS policy, Kerberos 5 mechanism could:

- recongnize Kerberos 5 security context data passed from user-space.
- perform security transform upon given data, using security context.
- pack/unpack data in gss/krb5 specific way.

3 Use Cases

3.1 Context establishment of gss/krb5

1. client: refresh a context, call into gss_refresh_ctx().
2. client: gss policy do upcall to user space daemon lgssd.
3. client: lgssd authenticate with kerberos KDC, prepare message token, call ioctl into kernel.
4. client: send the token to lustre server via PTLRPC.
5. server: parse the incoming request, hand it to gss policy.
6. server: gss policy do upcall to user space daemon lsvcgssd.
7. server: lsvcgssd verify the token, generate reply token and server side security context, call ioctl into kernel.
8. server: gss policy install the security context via krb5 mechanism, and send back reply token to client.

9. client: return the reply token to user space daemon lgssd.
10. client: lgssd verify the reply token, generate client side security context, call ioctl into kernel.
11. client: gss policy install the krb5 security context via krb5 mechanism.

4 Logic Specification

4.1 buffer arrangement

General rules:

- on wire data must conform to lustre_msg_v2, we use embedded lustre_msg_v2 to pack any extra data.
- no matter what data transform we need, we should try to prevent data copy around as possible as we can.

In integrity mode, signature will be the last segment of lustre_msg_v2, and computed from all the precede segments. We allocate the whole data buffer at beginning, and compute pointers to point to respect portion in the buffer, thus we totally avoid unnecessary data copy.

In privacy mode, as the initial implementation we allocate separate buffers for clear text and cipher text, for simplicity. This might add considerable memory consumption, some RPC (e.g. part of a transaction) will live for a long time thus also keep all the buffers for a long time. Later we might consider reduce the memory usage by using only one buffer. In any cases, the wire message only contains one segment which is the cipher text.

4.2 context negotiation

The procedure is essentially the same as normal gss negotiation. It usually take one or more message exchange between 2 peers to establish a security context. The transferred data is prepared by user space daemons, and specific to the actual

authentication mechanism it use. The kernel ptlrpc only responsible to send those message to target place and receive reply as blobs of data.

When user space call ioctl into kernel, it come with information about local client obd and remote node, then kernel could find a proper obd_import, and using normal RPC path to sent out the request and receive reply. A special flag should be added on ptlrpc_request to indicate bypassing all further security transforming. Just like other RPCs, the negotiation RPCs has the same impact on recovery.

After context negotiation complete successfully, server will send back a "context handle", which is simply a number or so. For subsequent RPC requests, client must present the "context handle" in the message, thus server could find the correct corresponding security context to handle the request.

4.3 Kerberos 5 environment

The service name for MDS is "*lustre_mds*", for OSS is "*lustre_oss*". According to Kerberos 5, each service principal must bind with FQN where the service resident. So for each MDS, sysadmin should add a service principal "*lustre_mds/hostname@REALM*" in KDC, and "*lustre_oss/hostname@REALM*" for each OSS.

For each of the service principal, sysadmin also should generate a service *keytab* file, and install on each server node, usually at */etc/krb5.keytab*. This is the credential which will be used to verify context negotiation requests.

On client node, a user must log on Kerberos 5 network at first, which usually involves invoke */bin/kinit* and type in password, and got a Ticket Grant Ticket (TGT) from KDC, and cached at /tmp for further use.

Suppose user *Alice* triggered a context negotiation, following is what will happen:

1. kernel do upcall to lgssd, with Alice's uid, service type (mds or oss), target NID, and client obd's uuid.
2. lgssd obtain server's hostname via target NID, construct service principal, construct request and encrypt using cached TGT, and send request to KDC.
3. KDC: generate sessions keys, encrypt with service key and Alice's key respectively, and send reply back.

4. lgssd decrypt reply got session key. Construct request which include the session key encrypted with service key, send to server node (via kernel ptlrpc service).

5. lsvcgssd verify the request, decrypt and got session keys. Construct reply and encrypt with session key, send back to client node (via kernel ptlrpc service).

6. lsvcgssd notify kernel (server side) to install the context with the session keys.

7. lgssd verify reply, notify kernel (client side) to install the context with the session key.

8. Now the in-kernel security context has been established.

We can see Kerberos 5 is somewhat bind to TCP/IP environment, which means each lustre node also must has TCP/IP environment.

Currently there's 2 major implementations of Kerberos 5: *MIT* version which is the standard in RedHat/Fedora, Debian, and many other districutions; *Heimdal* version which is standard in SuSE, maybe more. Firstly we only focus on MIT Kerberos, Heimdal support maybe added later.

4.4 context cache, and user space daemons

Here we follow what NFSv4 is doing. We adapt the NFS daemons to be used as lustre gss daemon, and use the same mechanism to communicate between user space and kernel: On client we use pipefs to send context negotiation request to and receive client side context from user space; On server the "nfsd" filesystem is used to do the job. On server side, we also use the general cache facility from NFS to cache the server contexts, just like NFS4.

We omit more description about the NFS4 stuff: pipefs, nfsd filesystem, general cache.

For above reason, Lustre might have some dependence on NFS: NFS should be enabled in kernel, "nfsd" and "pipefs" filesystem should be mounted, and so on.

4.5 replay attack protection

A simple algorithm which specified in RFC 2203 is used to protect replay attach. Each request contains a sequence number, and server maintained a sequence number window to track whether a sequence number have already been handled or not. Client also make sure the reply sequence match the request sequence.

4.6 secure reply ACK

to be added.

4.7 credential of root user

This is for kerberos 5 case only. Instead of using "root@REALM", a special service principal "*lustre_root/node@REALM*" should be created in KDC database, and generate service keytab file to be installed on all client nodes. Client daemon lgssd will use the keytab for root user. By doing this, root user don't need password to mount lustre.

This introduce a problem that any guy who has root permission on any of the client nodes will have the root privilege on lustre filesystem. So MDS should decide treat principal "*lustre_root/node@REALM*" as real root user or map it to another less privileged user.

4.8 GSSAPI interface

1. ***__u32 gss_import_sec_context(struct kvec *token, struct gss_ctx *ctx)***

    - @token: token which encoded security context.
    - @ctx: gss context.
    - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
    - Might sleep on memory allocation.

    Import security context which encoded in @token into gss context @ctx.

2. ***__u32 gss_delete_sec_context(struct gss_ctx *ctx)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Can't sleep.

   Destroy gss context @ctx.

3. ***__u32 gss_copy_reverse_context(struct gss_ctx *ctx, struct gss_ctx *ctx_new)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

   Copy gss context @ctx to @ctx_new, and make it be reverse one.

4. ***__u32 gss_inquire_context(struct gss_ctx *ctx, unsigned long *endtime)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Can't sleep.

   Obtain the expire time of gss context @ctx.

5. ***__u32 gss_get_mic(struct gss_ctx *ctx, struct kvec *msg, int msg_count, struct kvec *mic_token)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

   Generate MIC on data (@msg_count of vector @msg), and store in @mic_token.

6. ***__u32 gss_verify_mic(struct gss_ctx *ctx, struct kvec *msg, int msg_count, struct kvec *mic_token)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

Verify MIC in @mic_token upon data (@msg_count of vector @msg).

7. ***__u32 gss_wrap(struct gss_ctx *ctx, struct kvec *msg, struct kvec *out_token)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

   Encrypt data @msg and store cipher text in @out_token.

8. ***__u32 gss_unwrap(struct gss_ctx *ctx, struct kvec *in_token, struct kvec *msg)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

   Decrypt cipher text in @in_token and store clear text data in @msg.

9. ***__u32 gss_get_mic_bulk(struct gss_ctx *ctx, lnet_kiov_t *data, int vecsize, struct kvec *mic_token)***

   - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
   - Might sleep on memory allocation.

10. ***__u32 gss_verify_mic_bulk(struct gss_ctx *ctx, lnet_kiov_t *data, int vecsize, struct kvec *mic_token)***

    - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
    - Might sleep on memory allocation.

11. ***__u32 gss_wrap_bulk(struct gss_ctx *ctx, lnet_kiov_t *in_data, int in_vecsize, lnet_kiov_t *out_data, int out_vecsize, struct kvec *out_token)***

    - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.

- Might sleep on memory allocation.

12. ***__u32 gss_unwrap_bulk(struct gss_ctx *ctx, lnet_kiov_t *in_data, int in_vecsize, lnet_kiov_t *out_data, int out_vecsize, struct kvec *in_token)***

    - Return GSS_S_COMPLETE if succeed, otherwise return gss error code.
    - Might sleep on memory allocation.

4.9 Kerberos 5 mechanism

This part we also follow NFS4 implementation, using the same way and pack/unpack data, except we should support more cipher algorithms, at least for those mostly used algorithms used in Kerberos 5.

5 State Management

6 Alternatives

7 Focus of Inspection