High Level Design of GSS

Peter Braam, Eric Mei

Feb 27, 2005

1 Requirements

- Support GSSAPI framework in lustre.
- Support kerberos 5 as a mechanism of GSSAPI.
- Support user authentication and integrity/privacy protection for ptlrpc messages between clients and MDS's.

Most of the design came from NFSv4 project, and so far quite similar.

2 Functional Specification

From the whole picture, the newly added security facilities are actually add a security protection upon the ptlrpc connections between lustre nodes, tighly coupled with ptlrpc functions, so logically it's a part of PTLRPC module. But we try to separate the API as much as possible, and make it below ptlrpc layer, thus to avoid any high level logic complication, such as recovery, etc.

We choose to implement a general security API which utilized by ptlrpc to protect their messages, and implement the original mechanism (without strong authentication and message protection) as a instance of the security. Thus user could still choose to not use strong security to achieve high performance.

The first strong authentication mechanism we need support is Kerberos 5. It involves operations such as authenticating with KDC, some Kerberos internal data structure parsing, etc. It's hard, and not necessary, to put everything into kernel. So we resort to user space daemons, which simply call kerberos libraries to accomplish those tasks.

2.1 General security API

The API is splitted into two parts: client and server. It's not lustre client and server node, instead it is about ptlrpc connection: the client is the side which send out request is the client, which will later receive reply; and the other side of ptlrpc connection is the server. So, currently all MDS's, OSS's and lustre clients could be both security server and client.

From now on, the rpc request/reply message buffers will be managed by security module, since different security policy might require different buffer layout/manipulation scheme. But it is transparent to rest of ptlrpc and upper layer, the message buffer pointers are all they know.

2.1.1 Client side API

- Each ptlrpc_import must grab a security handler at the initialization.
- Each ptlrpc_request must hold a valid credential at first before doing anything.
- ptlrpc must call security API to allocate request buffer before filling into request data.
- ptlrpc must call security API to do message transform on request message before send out.
- ptlrpc must call security API to alloc reply buffer before really submit the RPC.
- ptlrpc must call security API to do message transform/verify on incoming reply message before parsing it.
- Each ptlrpc_request must drop the credential before be destroied.
- Each ptlrpc_import must release security handler before be destroied.

So the main client side API are:

```
/* import interaction */
int ptlrpcs_import_get_sec(obd_import *imp);
void ptlrpcs_import_drop_sec(obd_import *imp);

/* credential APIs */
int ptlrpcs_req_get_cred(ptlrpc_request *req);
void ptlrpcs_req_drop_cred(ptlrpc_request *req);
int ptlrpcs_req_refresh_cred(ptlrpc_request *req);

/* buffer manipulation */
int ptlrpcs_cli_alloc_reqbuf(ptlrpc_request *req,
                             int msgsize);
int ptlrpcs_cli_alloc_repbuf(ptlrpc_request *req,
                             int msgsize);
void ptlrpcs_cli_free_reqbuf(ptlrpc_request *req);
void ptlrpcs_cli_free_repbuf(ptlrpc_request *req);

/* rpc message transform */
int ptlrpcs_cli_wrap_request(ptlrpc_request *req);
int ptlrpcs_cli_unwrap_request(ptlrpc_request *req);
```

For all of them, return 0 means success, otherwise is error number.

2.1.2 Server side API

- Each incoming request must go through security checking/transform before be parsed.
- ptlrpc must call security API to allocate reply buffer before filling in reply data.
- Each reply must be performed transform before be sent out.
- Each ptlrpc_request must call security API to cleanup security related staff.

So the main server API are:

```
/* security checking for each incoming request */
int svcsec_accept(ptlrpc_request *req);

/* perform transform on reply message */
int svcsec_authorize(ptlrpc_request *req);

/* alloc reply buffer */
int svcsec_alloc_repbuf(ptlrpc_request *req,
                        int msgsize);

/* cleanup request */
void svcsec_cleanup_req(ptlrpc_request *req);
```

For all of them, return 0 means success, otherwise is error number.

2.2 Internal security API

The internal of security module could be devided into 2 layers. The upper level is the general layer, which just defined several sets of functions and rules. In the lower layer, we can implement several different security policies as the backend of the general layer. When an external API get called, this layer simply deliver the control to approriate security backend to accomplish the real things.

Correspond to external API, the internal function sets also are devided into client and server parts. At client side, there are mainly two types of security objects: "ptlrpc_sec" and "ptlrpc_cred". At server side the main object are "ptlrpc_svcsec".

2.2.1 Client side: ptlrpc_sec

A ptlrpc_sec represent an instance of the whole security facility. Each obd_import must hold a handle of a sec, and all following security activities associated with

the import will happen inside the context of this sec. Function sets defined by ptlrpc_sec are the basic security service which the security backend must implement, mainly are:

- Create and destroy a sec instance.
- Create a credential for a ptlrpc_request.
- Allocate and free request/reply buffers for a ptlrpc_request.
- The management of credential cache.

So the main API are defined as:

```
struct ptlrpc_secops {
  ptlrpc_sec * (*create_sec)
                      (sec_flavor_t flavor);
  void (*destroy_sec) (ptlrpc_sec *sec);

  ptlrpc_cred * (*create_cred)
                      (ptlrpc_sec *sec,
                       ptlrpc_request *req,
                       vfs_cred *cred);

  int  (*alloc_reqbuf)(ptlrpc_sec *sec,
                       ptlrpc_request *req,
                       int msgsize);
  void (*free_reqbuf) (ptlrpc_sec *sec,
                       ptlrpc_request *req);
  int  (*alloc_repbuf)(ptlrpc_sec *sec,
                       ptlrpc_request *req,
                       int msgsize);
  void (*free_repbuf) (ptlrpc_sec *sec,
                       ptlrpc_request *req);
};
```

2.2.2 Client side: ptlrpc_cred

A ptlrpc_cred represent a credential of a certain user. Each ptlrpc_request must hold a credential before doing anything. The function sets defined by ptlrpc_cred must implement by each security backend, mainly are:

- Credential management.
- Security transform for a message.

So the main API are defined as:

```
struct ptlrpc_credops {
  /* credential management */
  int  (*refresh) (ptlrpc_cred *cred);
  int  (*match)   (ptlrpc_cred *cred,
                   ptlrpc_request *req,
                   vfs_cred *vcred);
  void (*destroy) (ptlrpc_cred *cred);

  /* data transform for integrity protection */
  int  (*sign)    (ptlrpc_cred *cred,
                   ptlrpc_request *req);
  int  (*verify)  (ptlrpc_cred *cred,
                   ptlrpc_request *req);
  /* data transform for privacy protection */
  int  (*seal)    (ptlrpc_cred *cred,
                   ptlrpc_request *req);
  int  (*unseal)  (ptlrpc_cred *cred,
                   ptlrpc_request *req);
};
```

2.2.3 Server side: ptlrpc_svcsec

A ptlrpc_svcsec represent an instance of the server side security facility. It dose not need to be associated with certain export structure, since the security services are based on each incoming request, and no per-export status need to be maintained. The function set defined by ptlrpc_svcsec must be implemented by each security backend, mainly are:

```
struct ptlrpc_svcsec {
  /* security transform */
  int  (*accept)   (ptlrpc_request *req);
  int  (*authorize)(ptlrpc_request *req);

  /* buffer manipulation */
  int  (*alloc_repbuf)(ptlrpc_svcsec *svcsec,
                       ptlrpc_request *req,
                       int msgsize);
  void (*free_repbuf)(ptlrpc_svcsec *svcsec,
                      ptlrpc_reply_state *rs);

  /* cleanup associated security staff */
  void (*cleanup_req)(ptlrpc_svcsec *svcsec,
                      ptlrpc_request *req);
};
```

2.3 Security backend: NULL

NULL represent "no security", which is the simplest backend of the internal security API. It implemented all the APIs described above, but nullified all the operations like authentication, credit management, message transform, etc. So it's actually fallback to oringinal mode: no authenticaion, no message protection. So NULL security could be also considered as the proof of concept of the security framework.

2.4 Security backend: GSS

GSS implemented a small part of GSSAPI in the kernel, with some changes to reflect some rules of kernel programming. Like NULL security, GSS module is a backend of the general security framework, by implementing all internal security API in gss specific way. At the same time, GSS itself is also an other level of abstraction layer. It defines a set of functions and rules to be implemented by specific security mechanism, like kerberos 5. The API roughly are as following. For each function, return 0 means success, otherwise is gss error code.

```
struct gss_api_ops {
  /* context init/fini/query */
  u32 (*import_sec_ctxt)(rawobj_t *in_token,
                         gss_ctx  *ctx);
  u32 (*inquire_context)(gss_ctx *ctx,
                         time_t endtime);
  u32 (*delete_sec_ctxt)(gss_ctx *ctx);

  /* msg integrity transform */
  u32 (*get_mic)        (gss_ctx *ctx,
                         rawobj_t *msg,
                         rawobj_t *mic);
  u32 (*verify_mic)     (gss_ctx *ctx,
                         rawobj_t *msg,
                         rawobj_t *mic);

  /* msg privacy transform */
  u32 (*wrap)           (gss_ctx *ctx,
                         rawobj_t *in_token,
                         rawobj_t *out_token);
  u32 (*unwrap)         (gss_ctx *ctx,
                         rawobj_t *in_token,
                         rawobj_t *out_token);
};
```

The above gss_api_ops which will be implemented by backend mechanism are all about message protection, no authentication functions included. This is because the authentication part is not suitable be put into kernel, we'll use user

level daemons to accomplish it, and just tell kenel the final result of security context, which will be notify the specific mechanism by import_sec_ctxt() in gss_api_ops set.

There is no split upon the gss mechanism interface, they are equally on both client and server.

The general GSS module implement the all common part for all mechanisms, such as:

- Interface with general security layer.
- Mechanism management (register/deregister, etc.). Select proper mechanism according to various conditions.
- Authentication initiation.
- Interaction with user space daemons.
- Security context cache & management.

2.5 GSS mechanism: krb5

The gss_krb5 module simply implemented gss_api_ops, mainly are:

- Generate & verify MIC for data buffers.
- Encrypt & Decrypt for data buffers.

All transform upon data must conform to kerberos 5 standard.

2.6 lgssd & lsvcgssd

There's two kind of user level daemons: lgssd running on every client nodes; lsvcgssd running on every server nodes. They mainly perform the part of authentication & security context establishment which is not suitable be put into kernel, as mentioned before. Each daemon should be flexible enough to deal with different kind of authentications, kerberos 5 is one of them.

When needed, client gss module will issue request to lgssd, with information about who need authentication, what type of service, which target server. lgssd do all the things like authentication with authentication server, obtain security tokens, and notify kernel the final result of security context.

Server gss module will issue request to lsvcgssd, with information of the security initialization data. lsvcgssd will verify the whether the request are valid or not, compose a reply to client, and notify kernel the final result of security context.

We obey the standard of GSSAPI, which require data exchange between server and client during the context establishing phase. In our design lgssd and lsvcgssd will use in-kernel ptlrpc staff to do data exchange instead of all in user space.

2.7 User interface

The only interface to users is the mount parameters. Mount will accept options to determine what kind of security policy will be forced on the connections between client and MDS's:

```
-o sec=sec_flavor
```

"sec_flavor" must be one of:

- null: NULL security mode.
- krb5i: kerberos 5 authentication with integrity protection on rpc messages.
- krb5p: kerberos 5 authentication with privacy protection on rpc messages.

Without specify parameter "sec" means using default NULL security. On a client, connections to each MDS must have the same security type.

3 Use Cases

3.1 NULL security case

1. A user on a client access a lustre file which lead to an rpc must be sent to MDS.
2. Lustre client generate a ptlrpc_request, and call ptlrpcs_req_get_cred() to grab a cred.
3. Generic sec module found a matched valid cred.
4. Lustre client call ptlrpcs_cli_alloc_reqbuf() to allocate request buffer.
5. Generic sec module pass request to null_sec.
6. null_sec allocate the buffer, as normal way.
7. Lustre client fill in request data. and call ptlrpcs_cli_wrap_request().
8. Generic sec module pass request to null_sec.
9. null_sec do nothing and return.
10. Lustre client call ptlrpcs_req_alloc_repbuf() to allocate reply buffer.
11. Generic sec module pass request to null_sec.
12. null_sec allocate the buffer, as normal way.
13. Lustre client submit the rpc.
14. MDS get the request, call svcsec_accept() to perform security checking.

15. Generic sec module pass request to null_svcsec.
16. null_svcsec do nothing and return.
17. MDS parse request, and pass to normal mds handler.
18. MDS call svcsec_alloc_repbuf() to allocate reply buffer.
19. Generic svcsec module pass request to null_svcsec.
20. null_svcsec allocate the buffer, as normal way.
21. MDS fill in reply data, and call svcsec_authorize() to perform security transform.
22. Generic svcsec module pass request to null_svcsec.
23. null_svcsec do nothing and return.
24. MDS send out the reply, call svcsec_cleanup_req() to do cleanup before be destroied.
25. Generic svcsec module pass request for null_svcsec.
26. null_svcsec do nothing and return.
27. Lustre client get reply, call ptlrpcs_cli_unwrap_reply() to do security transform.
28. Generic sec module pass request to null_sec.
29. null_sec do nothing and return.
30. Lustre client parse the reply, do proper things accordingly, call ptlrpcs_req_drop_cred() before be destroied.

3.2 GSS/krb5 security case 1: first use

1. Suppose a client has mounted as krb5 mode, lgssd and lsvcgssd running on clients and servers.
2. Alice on this client access a lustre file at her first time, which lead to an rpc must be sent to MDS.
3. Lustre client generate a ptlrpc_request, and call ptlrpcs_req_get_cred() to grab a cred.
4. Generic sec module can't find a valid cred, create a new one for Alice, call into gss_sec to refresh it.
5. gss_sec send request to lgssd, with information of Alice's uid, service type, and target node.

6. lgssd prepare the context initialization data, pass back to Lustre client kernel.
7. Lustre client kernel send the initialization data to MDS.
8. MDS call svcsec_accept() into svcsec to handle the request.
9. Generic svcsec module pass the request to gss_svcsec.
10. gss_svcsec send the initialization data to lsvcgssd.
11. lsvcgssd verify the incoming data, generate the security context for server and reply message to client, pass down to MDS kernel.
12. gss_svcsec install the server side context, by calling service of gss_krb5, and cache the context in the kernel. Finally send reply message back to lustre client.
13. Lustre client get the reply, pass back to lgssd.
14. lgssd verify the reply data, generate security context for client side, pass down to lustre client kernel.
15. gss_sec install the context passed down. Now an security context between lustre client and MDS has been established, which will be represented by a valid cred on lustre client.
16. Lustre client call ptlrpcs_cli_alloc_reqbuf() to allocate request buffer.
17. Generic sec module pass request to gss_sec.
18. gss_sec allocate the buffer, according to the specific security service type.
19. Lustre client fill in request data. and call ptlrpcs_cli_wrap_request().
20. Generic sec module pass request to gss_sec.
21. gss_sec pass request to gss_krb5.
22. gss_krb5 sign or encrypt the message.
23. Lustre client call ptlrpcs_req_alloc_repbuf() to allocate reply buffer.
24. Generic sec module pass request to gss_sec.
25. gss_sec allocate the buffer, according to the specific security service type.
26. Lustre client submit the rpc.
27. MDS get the request, call svcsec_accept() to perform security checking.
28. Generic sec module pass request to gss_svcsec.

29. gss_svcsec parse the incoming request, find corresponding cached context. Then call service of gss_krb5 to verify the message.
30. gss_krb5 decrypt or verify the incoming message.
31. MDS parse request, and pass to normal mds handler.
32. MDS call svcsec_alloc_repbuf() to allocate reply buffer.
33. Generic svcsec module pass request to gss_svcsec.
34. gss_svcsec allocate the buffer, according to the specific security service type.
35. MDS fill in reply data, and call svcsec_authorize() to perform security transform.
36. Generic svcsec module pass request to gss_svcsec.
37. gss_svcsec call service of gss_krb5.
38. gss_krb5 sign or encrypt the reply message.
39. MDS send out the reply, call svcsec_cleanup_req() to do cleanup before be destroied.
40. Generic svcsec module pass request for gss_svcsec.
41. gss_svcsec cleanup the security related stuff.
42. Lustre client get reply, call ptlrpcs_cli_unwrap_reply() to do security transform.
43. Generic sec module pass request to gss_sec.
44. gss_sec call service of gss_krb5.
45. gss_krb5 verify or decrypt reply message.
46. Lustre client parse the reply, do proper things accordingly, call ptlrpcs_req_drop_cred() before be destroied.

3.3 GSS/krb5 security case 2: normal use

1. Suppose security context has been established for Alice, i.e. she has ever successfully accessed lustre filesystem.
2. Alice on this client access a lustre file again, which lead to an rpc must be sent to MDS.
3. The event sequence is the same as in 3.2, except there's no context initialization procedure anymore because we can find the security context in kernel cache. Which means no interaction with user space daemons are needed for the whole procedure.

3.4 GSS/krb5 security case 3: destroy

1. Suppose security context has been established for Alice, i.e. she has ever successfully accessed lustre filesystem.
2. Alice before logout, tell lustre kernel to flush her security context.
3. gss_sec find Alice's contexts, for each of them send destroy notification rpc to MDS.
4. MDS get the requests, hand to svcsec by calling svcsec_accept().
5. Generic svcsec module pass the request to gss_svcsec.
6. gss_svcsec find cached contexts for each request, destroy them, and send back replies.
7. gss_sec get the replies, also destroy local cached context.

3.5 0-conf mount and umount, in GSS/krb5 mode

1. root on a client mount lustre by: mount -t lustre -o sec=krb5p mds1:/mds1/client /mnt/lustre
2. Lustre client prepare an import to MDS, create a ptlrpc_sec associated with the import.
3. Lustre client prepare MDS_CONNECT rpc to MDS.
4. A security context initialize procedure for root will be done.
5. Lustre client send MDS_CONNECT request to MDS, and got reply.
6. Lustre client fetch the client startup log from MDS.
7. Lustre client destroy the import, which lead to procedure of destroying the existing security context.
8. Lustre replay the startup log, which will constrct new connections to MDS's, and lead to security context be established accordingly.
9. Mount finish successfully.
10. root do umount by : umount /mnt/lustre
11. Lustre client prepare another import to MDS, create a ptlrpc_sec associated with the import.
12. Lustre client prepare MDS_CONNECT message to MDS.
13. A security context initialize procedure for root will be done.
14. Lustre client send MDS_CONNECT request to MDS, and got reply.

15. Lustre client fetch the client shutdown log from MDS.
16. Lustre client destroy the import, which lead to procedure of destroying the existing security context.
17. Lustre replay the shutdown log, which will destruct all connections to MDS's, and lead to security context be destroied accordingly.
18. Umount finish successfully.

3.6 GSS/krb5 context expiration

1. Suppose security context has been established for Alice, i.e. she has ever successfully accessed lustre filesystem.
2. Some time later, Alice's context on MDS expired, and be destroied.
3. Alice on this client access a lustre file again, which lead to an rpc be sent to MDS.
4. MDS failed to find the context, send error reply back.
5. Lustre client drop the context, re-establish a new context.
6. Lustre client resent former rpc with the new context.
7. The rpc finish successfully.

3.7 GSS/krb5 client reboot

1. A client reboot, remount lustre filesystem or not.
2. MDS's will keep the old security contexts, since they'v no idea whether the corresponding contexts on client exist or not.
3. Later those context expired and then be destroied.

3.8 MDS reboot and recovery

1. A MDS crashed and re-setup.
2. A gss/krb5 client send a request as normal.
3. MDS can't find proper security context, send back error reply.
4. The client drop the old context, re-establish a new security context with the MDS.
5. The client re-send former rpc with the new context.
6. MDS return ENOTCONN, thus initiate recovery procedure.

4 Logic Specification

4.1 Wire data format

Security subsystem know nothing about the internal structure of lustre_msg, but prepend a security header to every on-wire ptlrpc packet:

```
struct ptlrpcs_wire_hdr {
  u32  secflavor;  /* NULL/GSS */
  u32  sectype;    /* none/integrity/privacy */
  u32  msg_len;    /* length of lustre message */
  u32  sec_len;    /* length of security payload */
};
```

All fields are stored in little-endian format. The layout of every on-wire packet will be:

```
struce wire_packet {
    ptlrpc_wire_hdr;  /* 16 bytes */
    lustre_msg;       /* 0 - any bytes */
    security_payload; /* 0 - any bytes */
};
```

- In NULL security mode, the security payload is always 0 bytes.
- In gss security mode, the security payload is always non-zero bytes.
- In gss/privacy mode, the lustre_msg is always 0 bytes, because they have been encoded into the security payload section.
- At any cases, the whole packet must be 8-bytes aligned.

Right now only gss have security payload. Each security payload start with a gss header:

```
struct gss_wire_hdr {
  u32      version; /* GSS version */
  u32      proc;    /* procedure */
  u32      seq;     /* sequence number */
  u32      svc;     /* service */
};
```

All fields are stored in little-endian format. "proc" means gss control procedure, could be INIT, INIT_CONTINUE, DATA, DESTROY, etc. "seq" is for the sequence number checking algorithm from RFC 2203, to prevent replay attack. The whole security payload format will be:

```
struct gss_security_payload {
  gss_wire_hdr;
  context_handle;
  mech_payload;
};
```

The "context_handle" is let the server find proper security context cached. The "mech_payload" is the actual signature or ciphertext made by specific gss mechanism, which is transparent to generic gss layer.

4.2 GSS context

The gss context is divided into two parts: generic gss context and mechanism specific context. The generic part is in fact quite simple, but not symmetric for client and server. On the client side, they mainly are:

- Control procedure. Could be INIT, DATA, etc.
- Sequence number.
- Peer context handle. This will be sent to server in each request, used by server to address the correponding context.

On the server side, they mainly are the facility to implement sequence number algorithm. The purpose is to prevent replay attack: Suppose a bad guy could eavesdrop the network, and record an rpc packet transfered across the network, and some time later re-send the packet again to the same machine. In this case the target machine should be able to detect this is an replay attack and drop it.

The mechanism specific context could only be interpreted and used by certain gss mechanism. They are generated by lgssd or lsvcgssd and then installed in the kernel. For gss_krb5 mechanism, they are mainly:

- Algorithm used in signature/verification.
- Algorithm used in encryption/decryption.
- Valid time.
- Other krb5 specific staff.

4.3 gss context creation and management

The rpc implementation in standard 2.6 kernel contains a general cache and upcall code, which is used by NFSv4 server to interact with server side daemon and cache security context in the kernel; And "rpc_pipefs" mechanism which allow kernel communicate with user space as message basis, NFSv4 client use it to interact with client side daemon. At this part we follow what NFSv4 does, and even use the server cache and rpc_pipefs at extact the same way as what NFSv4 use them. Please refers to NFSv4 implementation for the whole details, here we only outline the basics.

4.3.1 Security client

At client side, rpc_pipefs is required to be mounted, usually at /var/lib/nfs/rpc_pipefs. When each instance of ptlrpc_sec is created, an pipe which is the outlet to user daemon will also be created in the pipefs.

When in-kernel gss is asked to create security context for certain user, it simply pump an simple message, which contains uid, service name, target uuid, target nid, etc. into pipe, and wait for the reply from user space. The correct reply will contains:

- General gss context. This will be installed in generic gss layer.
- Mechanism specific context. GSS will call mechanism's import_sec_context() to install into mechanism layer.

Each ptlrpc_sec structure contains a context hash table, each context entry has it's own expire time. Expired entry will be dropped once be found expired. When we drop a valid context by force, an notification RPC will be sent to server to also destroy server side of the context. The whole client side cache management will be quite simple and straightforward.

4.3.2 Security server

An "nfsd" filesystem must be mounted at /proc/fs/nfsd, which is to communicate between user space and kernel. This require NFSv4 muct be enabled in the kernel.

We use the general cache management code provided by rpc. The interaction between gss and cache is roughly as:

- We need implement two kinds of cache type in the framework of general cache manager: one for context intialization, one for context. They are all about define how to submit request and parse reply.
- When an context initialize request comes in, gss generate a context init cache entry, assoiated with the request data, submit to cache manager. Then wait for the reply.
- Cache manager will create an context cache entry, reply the entry handler to GSS. And submit all data to user space daemon.
- Then GSS will wait until the context entry is filled.

4.4 lgssd and lsvcgssd

Since we only support kerberos 5 as the mechanism of GSS, all following discussions are assuming krb5 case. Both daemons are built on GSSAPI, which means they call service of gssapi to negotiate security context. So on both sides, user space GSSAPI enviroment must be properly configured, mostly of which is configure kerberos 5 as the mechanism of GSSAPI. The GSSAPI library is

usually static library, must be prepared when build lgssd and lsvcgssd, but not necessary for running lgssd and lsvcgssd.

Beside GSSAPI, our daemons, especially lgssd, will exploit some features of kerberos 5 directly, so kerberos 5 development enviroment must also be properly configured.

4.4.1 lgssd

Each client node will have one lgssd running. It constantly monitor the change and event at certain directory in rpc_pipefs. When lgssd got an context initialize request from rpc_pipefs, it at first compose the service pincipal, e.g. "lustre_mds@CN.CFS". Then find out whether there's already cached ticket of it for this user. If not found, then obtain the ticket from KDC, using kerberos 5 API. This require that node already have kerberos 5 TGT cached. If succeefully got the new ticket, lgssd will also cache it locally.

After that, lgssd call GSSAPI init_sec_context() to prepare the initialize request data, then pass down to kerenl which will in turn send to server. In kerberos 5 case, only one data exchange is needed for initial negotieation. So the reply either contains error notification, or GSSAPI specific reply data. lgssd will parse the reply, form the suitable context and passdown to kernel.

Note that the lgssd must know certain internal strucutre of gss and kerberos 5 to be able to parse the reply.

4.4.2 lsvcgssd

At startup, lsvcgssd will prepare it's service credential, which is about parse kerberos 5 service keytab, prepare for the service. And enter a loop to monitor event of nfsd filesystem. When a request comes up, it read the incoming gss request, and pass into GSSAPI accept_sec_context(), which will check the request using service credential. If succeed, the server side context and gss reply to client will be generated. lsvcgssd will pass all of them into kernel, which will in turn install the server side context and send the rest to client. Note lsvcgssd also must know certain internal structure of gss the kerberos 5.

4.5 RPC of security initialization

The security context negotiation is done by kernel ptlrpc layer. But this RPC is special because all other normal RPCs will go through series of security checking/transform which is not needed for this one. So we treat it as raw RPC and initiate it by:

```
int ptlrpc_do_rawrpc(struct obd_import *imp,
                     char *reqbuf, int reqlen,
                     char *repbuf, int replen,
                     int timeout);
```

It simply send "reqlen" of "reqbuf" to destination described by peer of "imp", and got maximum "replen" of reply into "repbuf". ptlrpc_do_rawrpc() use some ptlrpc facilities like callback/wakeup, etc, but will not trigger any recovery process since we consider the whole initialize procedure is happen underlying the ptlrpc layer.

4.6 Krb5 mechanism

This module is a backend of in-kernel GSS. It's all about the details of how to sign/verify and encrypt/decrypt messages in kerberos 5 standard way. We also follow what NFSv4 does, so please refers to NFSv4 implementation for detials. As of NFSv4, currently we only support DES and MD5 Algorithm.

4.7 Reverse RPC

The procedure of establishing gss context is complex as we described. Client need to obtain kerberos ticket, while server need to be installed an service keytab. In most cases, the server side of security activity is the lustre server, e.g. MDS, and client is the lustre client. But some RPCs, e.g. LDLM ASTs, some llog RPCs, are initiated by lustre server node. Apparently we can't use the same procedure to initialize security context for those reverse connections.

It looks not good to use any context which already be established between client and server, since those contexts could be expired or dropped at anytime, and AST rpc is crucial in lustre, fail to send out will lead to client be evicted. So the reverse context must be always valid. And the situation is made more complex by the fact that a key will be considered insecure if it is used to encrypt too many messages.

Currently we force all reverse connections use NULL security, which is always valid but of course not secure. Although AST and llog RPC itself didn't contains any confidential of user message, but malicious users might use fake rpc packet to attack lustre filesystem. Later we may have to design a secure way for reverse connection.

5 State Management

5.1 Impact on recovery

We try to put the security part under the layer of ptlrpc, to avoid disturbing recovery procedure. There's some consideration here:

- Client initiate the security negotiation using "raw rpc", to avoid any recovery related staff.
- Server handle the negotiation request without trigger any ptlrpc recovery staff.

- Sometime client might get reply of server that the security context is invalid, after the context is re-established, client need send the original request again. In this case, client need make sure this request is sent as it was, especially without changing any flags like RESENT, etc.

5.2 Prerequisite on server and client

Security identities on lustre client side is based users. Each user want to access lustre filesystem must at first authenticate with authentication server, which is KDC in kerberos case, and cache the credential locally. Lustre client use cached credential of appropriate user to authenticate with MDS.

On lustre server side the identities is based on service. In Kerberos 5 case, we need create a service principal, and create a kaytab for it. The kaytab is usually a regular file stored on disk, and must be installed/parsed before providing any services.

All lustre clients must have lgssd running, while all MDS's must have lsvcgssd running.

5.3 GSS context pairs

There will be a lot of gss contexts be established in a typical lustre system. Each user on each lustre client will have a gss context to each MDS. If we use GSS also in client-OSS, MDS-OSS or inter MDS's, the number will be even much bigger. Suppose we setup a large cluster, the time of mounting tens of thousands of client will be much longer.

5.4 Others

No disk format changes. No changes on current network API.

6 Alternatives

- Now the null security is at the same layer as gss: both are two security policies under framework of generic security API. It _might_ be possible to remove the generic security API, and make gss be the generic layer, and implement null security as a special mechanism of gss. This could reduce the security layering by one.

- Currently at client side, each import has an ptlrpc_sec, which contains hash table of cached contexts. Probably we can use a single large pool of context cache for the whole client, just like single hash table for cached contexts at server side.

7 Focus of Inspection

- Are there oversights on recovery issue?
- How could the gss scheme scale?
- What's thought about reverse import?
- What impact on us to use NFSv4 staff, both in-kernel cache/rpc_pipefs and user space daemons?