Security API & Null Policy DLD

Eric Mei

February 8, 2008

1 Functional Specification

1.1 new data structures

struct ptlrpc_sec_policy: security policy module.

struct ptlrpc_sec: on client side, security policy instance owned by an *obd_import*.

struct ptlrpc_cli_ctx: the client side half of a user's security context on a ptlrpc connection.

struct ptlrpc_svc_ctx: the server side half of a user's security context on a ptlrpc connection.

1.2 security flavor

Security flavor is an unsigned integer number, in which encoded following information:

- security policy number
- sub-flavor
- security service type, e.g. authentication only, encryption, etc.
- misc flags

1.3 policy registration

1. ***int sptlrpc_register_policy(struct ptlrpc_sec_policy *policy)***

   - @policy: security policy.
   - Return 0 when succeed, otherwise indicate an error.
   - Can not sleep.

   Register a security policy to system. Register a policy twice will return error.

2. ***void sptlrpc_unregister_policy(struct ptlrpc_sec_policy *policy)***

   - @policy: security policy.
   - Can not sleep.

   Unregister a security policy from system. If the policy is already unregistered, it is ignored silently.

1.4 APIs for import

1. ***int sptlrpc_import_get_sec(struct obd_import *imp, struct ptlrpc_svc_ctx *ctx, __u32 flavor, unsigned long flags)***

   - @ctx: service security context. It will be mostly NULL except the @imp is a reverse import.
   - @flavor: security flavor.
   - @flags: special flags, such as reverse, etc.
   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation or possible interaction with user-space.

   The affected policy is supposed to create a new or reuse a existed ptlrpc_sec structure, and assign to @imp->imp_sec. If all succeed, this function will hold a reference to the @imp, and also increase the policy module's refcount.

2. ***void sptlrpc_import_put_sec(struct obd_import *imp)***

   - Might sleep on memory allocation or possible interaction with user-space.

   Release @imp->imp_sec from the import. The ptlrpc_sec structure will be destroied if its refcount dropped to 0. We also release a reference of @imp and affected policy module.

3. ***int sptlrpc_import_check_ctx(struct obd_import *imp)***

   - return 0 when found a valid security context, otherwise indicate an error.
   - Might sleep on RPC waiting or memory allocation.

   Given an import, check whether current user has a valid security context. It possibly trigger context refreshing procedure and wait for it complete, so it might sleep and return after relatively long time.

1.5 APIs for request

1. ***int sptlrpc_req_get_ctx(struct ptlrpc_request *req)***

   - return 0 when obtained a context, otherwise indicate an error.
   - Might sleep on memory allocation.

   Find a security context of current user on the @req->rq_import, or create a new one if cache missed, assign to @req->rq_cli_ctx. The request will hold a reference of the context. The context don't need to be uptodate, and we are not supposed to wait its uptodate.

   Caller need to make sure @req->rq_import is not NULL, and @req->rq_cli_ctx is NULL.

2. ***void sptlrpc_req_put_ctx(struct ptlrpc_request *req)***

   - Might sleep on memory allocation or interaction with user-space.

Dereference context @req->rq_cli_ctx, and set @req->rq_cli_ctx to be NULL. According to the context and corresponding ptlrpc_sec status, the context might be simply put back to cache or be destroied. In rare case, it probably cause the corresponding ptlrpc_sec structure be destroied too.

Usually called before @req is about to be freed.

3. ***int sptlrpc_req_refresh_ctx(struct ptlrpc_request *req, long timeout)***

   - @timeout: max time caller wish to wait:
     - > 0 : wait at most @timeout seconds.
     - = 0 : wait until context uptodate or fatal error happened.
     - < 0 : don't wait.
   - return 0 when succeed, otherwise indicate an error.
   - might sleep on RPC waiting or memory allocation.

   Caller should guarantee @req->rq_cli_ctx is not NULL. If the context is already uptodate or is during upcall, simply return 0; If the context is in error status, return error code. If the context is dead, replace it with a new one, trigger a refresh; and wait @timeout as described.

1.6 RPC message manipulation

1. ***int sptlrpc_cli_alloc_reqbuf(struct ptlrpc_request *req, int msgsize)***

   - @msgsize: maximum size of request rpc message.
   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Allocate request message buffer for a request. Caller don't need to know any details of the request buffer, but upon this return successfully, call could access @req->rq_reqmsg which point to a request buffer sizes at least @msgsize.

   Caller should guarantee @req->rq_cli_ctx not NULL, and only call this once for a request.

2. ***int sptlrpc_cli_alloc_repbuf(struct ptlrpc_request *req, int msgsize)***

- @msgsize: maximum size of anticipated reply message.
- return 0 when succeed, otherwise indicate an error.
- Might sleep on memory allocation.

Allocate reply message buffer for a reply. Caller don't need to know any details of the reply buffer, and @req->rq_repmsg is not supposed to be accessable even after this is called.

3. ***void sptlrpc_cli_free_reqbuf(struct ptlrpc_request *req)***

   - Can not sleep.

   Free request message buffer of a request. Caller don't need to know any details of the request buffer, but @req->rq_reqmsg can't be accessed after this is called. Usually called before @req is about to be freed.

4. ***void sptlrpc_cli_free_repbuf(struct ptlrpc_request *req)***

   - Can not sleep.

   Free reply message buffer of a request. Caller don't need to know any details of the request buffer, but @req->rq_repmsg can't be accessed after this is called. Usually called before @req is about to be freed.

5. ***int sptlrpc_cli_wrap_request(struct ptlrpc_request *req)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Perform security data transform on the request message pointed by @req->rq_reqmsg. Caller should guarantee that @req->rq_reqmsg which sizes @req->rq_reqlen contains the whole request message want to be sent out. Caller should not access req->rq_reqmsg anymore after this is called, unless it was instructed to resend again.

6. ***int sptlrpc_cli_unwrap_reply(struct ptlrpc_request *req)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

Verify or decrypt the incoming reply data. Upon return successfully, @req->rq_repmsg should point to the reply message which sizes @req->rq_replen. Only call this in *after_reply()*.

1.7 server side

1. ***int sptlrpc_svc_unwrap_request(struct ptlrpc_request *req)***

   - @req: incoming request.
   - return security handling code.
   - Might sleep on memory allocation, interaction with user-space.

   Called in *ptlrpc_server_handle_request()*, to verify or decrypt the incoming request data @req->rq_reqbuf which sizes @req->rq_reqdata_len. Return code a following:

   - SECSVC_OK: Success, and @req->rq_reqmsg should point to the request message which sizes @req->rq_reqlen. Caller should continue normal handling by target device.
   - SECSVC_FINISHED: Security policy has finished this handling, and reply has been prepared. caller should simply send this reply out by calling *ptlrpc_send_reply()*.
   - SECSVC_DROP: Security policy has detect unrecoverable error in the request, caller should drop this request.

2. ***int sptlrpc_svc_wrap_reply(struct ptlrpc_request *req)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Caller should make sure @req->rq_repmsg which sizes @req->rq_replen is the whole reply data it want to send out. This function sign or decrypt the outgoing reply message. @req->repmsg is not supposed to be accessable after this is called.

3. ***int sptlrpc_svc_alloc_rs(struct ptlrpc_request *req, int msgsize)***

- @req: request.
- @msgsize: the maximum size of reply message.
- return 0 when succeed, otherwise indicate an error.
- Might sleep on memory allocation.

Allocate reply state and reply message buffer. Caller don't need to know the details of the buffers, but upon return successfully, @req->rq_reply_state should point to a structure *ptlrpc_reply_state*, and @req->rq_repmsg should point to reply message buffer which sizes at least @msgsize.

4. ***void sptlrpc_svc_free_rs(struct ptlrpc_reply_state *rs)***

   - @rs: reply state.
   - Can not sleep.

   Free reply state and reply message buffer. Caller don't need to know the details of the buffers.

5. ***void sptlrpc_svc_cleanup_req(struct ptlrpc_request *req)***

   - @req: request.
   - Can not sleep.

   It's just a hook function, called before server freeing @req, give security module a chance to cleanup security data structures which might allocated for this request.

1.8 reverse context

1. ***int sptlrpc_svc_install_rvs_ctx(struct obd_import *imp, struct ptlrpc_svc_ctx *ctx)***

   - @imp: the reverse import.
   - @ctx: the normal security context used to handle the request.
   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

Derive a reverse client security context from a normal service context @ctx, and install it into the policy instance associated with @imp. It's usually called on the reverse import when server finished normal handling of an *OBD_CONNECT*.

2. ***int sptlrpc_cli_install_rvs_ctx(struct obd_import *imp, struct ptlrpc_cli_ctx *ctx)***

   - @imp: import.
   - @cred: an valid client security context.
   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Derive a reverse service security context from a normal client context @ctx and install it. It's usually called when client get the success reply of *OBD_CONNECT* and finished the normal handling of that.

1.9 Bulk I/O

1. ***int sptlrpc_bulk_write_cli_wrap(struct ptlrpc_request *req, struct ptlrpc_bulk_desc *desc)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Called on client side before bulk write rpc be sent out. It checksum, sign, or encrypt the outgoing bulk data, and pack result into request. In encryption case, we probably need allocate extra pages which are tracked by @desc, and are used for the later bulk pages register.

2. ***int sptlrpc_bulk_write_svc_unwrap(struct ptlrpc_request *req, struct ptlrpc_bulk_desc *desc)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

Called on server side after bulk get completed. It verify checksum or signature, or decrypt the bulk data. It also pack necessary data into reply if needed. Called should guarantee ptlrpc_pack_reply() has been called upon the @req.

3. ***void sptlrpc_bulk_write_cli_verify(struct ptlrpc_request *req, struct ptlrpc_bulk_desc *desc)***

   - Might sleep on memory allocation.

   Called on client side after bulk write rpc get reply. It only give warnings if it found server & client side checksum mismatch.

4. ***int sptlrpc_bulk_read_svc_wrap(struct ptlrpc_request *req, struct ptlrpc_bulk_desc *desc)***

   - return 0 when succeed, otherwise indicate an error.
   - Might sleep on memory allocation.

   Called on server side before bulk read be sent out. It checksum, sign, or encrypt the outgoing bulk data, and pack result into request. In encryption case, we probably need allocate extra pages which are tracked by @desc, and are used for the later bulk pages register.

5. ***int sptlrpc_bulk_read_cli_unwrap(struct ptlrpc_request *req, struct ptlrpc_bulk_desc *desc)***

   - Might sleep on memory allocation.

   Called on client side after bulk read rpc get reply. It verify checksum or signature, or decrypt the bulk data.

1.10 specify security flavor

1.10.1 RPC security flavor

User can specify which security flavor be used separately between lustre nodes: client, MDS, OSS, etc., by supplying parameters to *mount.lustre* or *lconf*.

on client node, the parameters for *mount.lustre* are:

- *mds_sec=flavor*
- *oss_sec=flavor*

on mds node, the parameters for *lconf* are:

- *–mds_oss_sec=flavor*

The parameter *flavor* must be a valid flavor name, e.g. "*null*" means *null* policy. In any case, if omit these parameters lustre will use the default *null* policy for all connections.

1.10.2 bulk security flavor

Currently we are not very sure how bulk I/O security (checksum, signature, and encryption) will affect performance, how should it connect to RPC security flavor, and how end user willing to use it. So we allow user specify it with mount options:

- *bulk_sec=bulk_flavor*

The parameter *bulk_flavor* could be following strings:

- *none*: no protection.
- *csum_alg*: checksum with certain algorithm to protect integrity. This could be "*csum_crc32*", "*csum_md5*", "*csum_sha1*", etc.
- *sign*: sign on bulk data using RPC security context. Invalid for *null* policy.
- *priv*: encryption bulk data using RPC security context. Invalid for *null* policy.

2 Use Cases

2.1 life cycle of *ptlrpc_sec*

1. client allocate a new obd_import, call sptlrpc_import_get_sec() to allocate a security policy instance for this import.
2. prepare MDS_CONNECT rpc. allocate a ptlrpc_request structure, call sptlrpc_req_get_ctx() to obtain a credential.
3. call sptlrpc_req_refresh_ctx() to make context uptodate, server will also cache a correponding context.
4. after proper preparation, send out the MDS_CONNECT rpc.
5. mds handle the request, using the cached context.
6. mds allocate a reverse import, call sptlrpc_import_get_sec() to allocate a reverse security policy instance for this import, which will internally call sptlrpc_svc_install_rvs_ctx() to install a reverse context.
7. mds send back reply.
8. client handle the reply, call sptlrpc_cli_install_rvs_ctx() to install a reverse security context. the ptlrpc connection is fully established by then.
9. client send MDS_CLOSE rpc.
10. mds call ptlrpcs_import_put_sec() on the reverse import, dereference & destroy the reverse ptlrpc_sec, which cause all reverse contexts be flushed.
11. mds destroy the reverse import and export, send back reply.
12. client call sptlrpc_import_put_sec() on the import, dereference & destroy the ptlrpc_sec, which cause all contexts be flushed.
13. client destroy the import.

2.2 forced reconnect on a FULL ptlrpc connection

1. client send OBD_CONNECT for a import.
2. server handle the connect, and call sptlrpc_svc_install_rvs_ctx() to install a reverse cred.
3. server send reply to client.
4. client finish the connect, and call sptlrpc_cli_install_rvs_ctx() to install a reverse security context.

3 Logic Specification

3.1 data structures and internal APIs

3.1.1 user credential (identity)

A user's credential, i.e. its identity, from VFS is represented by:

```
struct vfs_cred {
  uid_t    uid;
}
```

Currently uid is the only key between different users. In the future we might add more fields in for new features like PAG.

3.1.2 buffer arrangement

As a requirement, the wire rpc data should always conform to *lustre_msg* format. So the security transform is actually embedding the real lustre request/reply messages into an wrapper *lustre_msg* structure.

On rpc client, *ptlrpc_request* will contains following fields:

```
struct ptlrpc_request {
  ...
  /* existing fields */
  struct lustre_msg   *rq_reqmsg;
  struct lustre_msg   *rq_repmsg;
  int                  rq_reqlen;
  int                  rq_replen;
  /* newly added fields */
  struct lustre_msg   *rq_reqbuf;
  struct lustre_msg   *rq_repbuf;
  int                  rq_reqbuf_len;
  int                  rq_reqdata_len;
  int                  rq_repbuf_len;
  int                  rq_repdata_len;
  ...
}
```

The *rq_reqmsg*/*rq_repmsg* and *rq_reqlen*/*rq_replen* still means the lustre request/reply message buffer and length, in clear text, used in rest part of ptlrpc and upper layer. So the existing logic could keep almost unchanged.

The other 6 new fields are managed by security policy modules, upper logic should avoid using them directly. The *buf*/*buf_len*/*data_len* respectively means buffer pointer, total buffer length, and actual length of data filled in.

On rpc server, *ptlrpc_reply_state* will contain following fields:

```
struct ptlrpc_reply_state {
  ...
  /* existing fields */
  struct lustre_msg   *rs_msg;
  /* newly added fields */
  struct lustre_msg   *rs_repbuf;
  int                  rs_repbuf_len;
  int                  rs_repdata_len;
  ...
}
```

The meaning of them are the same as above described.

Some request and reply buffers are pre-allocated, e.g. request pool and reply state pool. We should add the maximum length of security payload into the pre-allocated buffer, and make use of it instead of allocate extra buffers.

3.1.3 security policy

Each policy module should implement & register following structure:

```
struct ptlrpc_sec_policy {
  char   *policy_name;
  __u32   policy_number;
  struct policy_cops *client_ops;
  struct policy_sops *server_ops;
}
```

The *client_ops* and *server_ops* respectively point to function set of client side and server side. A policy module must implement both of them. The client side function set roughly include following functions:

1. ***struct ptlrpc_sec *create_sec(struct obd_import *imp, struct ptlrpc_svc_ctx *ctx, __u32 flavor, unsigned long flags);***

   - @ctx: service security context. It is NULL except the @imp is a reverse import.
   - @flavor: security flavor.
   - @flags: security flags such as reverse, etc.

   Return a security structure, otherwise return NULL.

2. ***void destroy_sec(struct ptlrpc_sec *sec);***

   Destroy the given client security structure.

3. ***struct ptlrpc_cli_ctx *create_cli_ctx(struct ptlrpc_sec *sec, struct vfs_cred *vcred);***

   - @vcred: vfs identity of an user.

Create a context structure corresponding to user @vcred and return. The context don't need to be uptodate at that time. Return NULL if any error happen.

4. ***int install_reverse_ctx(struct obd_import *imp, struct ptlrpc_sec *sec, struct ptlrpc_cli_ctx *ctx);***

   - @ctx: client context of the user who perform the *OBD_CONNECT*.

   Install reverse security context according to a normal context in @ctx. Details hide in security policy implementation.

5. ***int alloc_reqbuf(struct ptlrpc_sec *sec, struct ptlrpc_request *req, int msgsize);***

   - @msgsize: maximum request message size.

   Allocate request message buffer for a request at @req->rq_reqbuf. Return 0 means success, otherwise return error code. After successfully return, @req->rq_reqmsg should point to buffer which sizes at least @msgsize, to allow upper layer fill in request data. @req->rq_reqmsg could simply point to a inner part of @req->rq_reqbuf or another separate buffer. Whatever the buffer layout is, security policy should try to avoid data be copied around as possible as it can.

6. ***int alloc_repbuf(struct ptlrpc_sec *sec, struct ptlrpc_request *req, int msgsize);***

   - @msgsize: maximum reply message size.

   Allocate reply message buffer for a request at @req->rq_repbuf which at least sizes @msgsize + maximum possible security payload. Return 0 means success, otherwise return error code.

7. ***void free_reqbuf(struct ptlrpc_sec *sec, struct ptlrpc_request *req);***

   Free request message buffer which allocated at @req->rq_reqbuf.

8. ***void free_repbuf(struct ptlrpc_sec *sec, struct ptlrpc_request *req);***

   Free reply message buffer which allocated at @req->rq_repbuf.

The server side function set roughly include:

1. ***int unwrap_request(struct ptlrpc_request *req);***

   Perform security transform and verify upon incoming on-wire request message pointed by @req->rq_reqbuf. Return SECSVC_OK means success and @req->rq_reqmsg should point to clear text of request message; Return SECSVC_COMPLETE means security module has prepared the reply and only need send it out; Return SECSVC_DROP means error happens and caller should drop this request.

2. ***int wrap_reply(struct ptlrpc_request *req);***

   Perform security transform upon outgoing reply message pointed by @req->rq_repmsg, length @req->rq_replen. Return non-zero is anything wrong; otherwise return 0 and @req->rq_reply_state->rs_repbuf point to the on-wire reply message which ready to be sent out.

3. ***int alloc_rs(struct ptlrpc_request *req, int msgsize);***

   - @msgsize: the maximum reply message length.

   Allocate reply_state structure and reply message buffer. If succeed, @req->rq_reply_state should point to the allocated reply_state, req->rq_repmsg should point to reply message buffer.

4. ***void free_rs(struct ptlrpc_reply_state *rs);***

   Free the reply state buffer pointed by @rs, and reply message buffer pointed by @rs->rs_repbuf.

5. ***void cleanup_req(struct ptlrpc_request *req);***

   Give security policy a chance to cleanup security data allocated for handling the request. Details hide in policy module.

6. ***int install_reverse_ctx(struct obd_import *imp, struct ptlrpc_svc_ctx *ctx);***

   - @ctx: security service data which contains normal security context.

   Derive a reverse credential from a normal service context in @ctx, and installed in the reverse *ptlrpc_sec* of @imp. Details hide in specific policy module.

3.1.4 client security facility

At client side each import point to a security structure as following:

```
struct ptlrpc_sec {
  struct ptlrpc_sec_policy  *policy;
  __u32                      default_flavor;
  struct obd_import         *imp;
  struct list_head          *ctx_cache;
}
```

The *ctx_cache* is the hash table of contexts. Other fields are self-explanatory.

3.1.5 client context

```
struct ptlrpc_cli_ctx {
  struct list_head       hash;
  atomic_t               refcount;
  struct vfs_cred        vcred;
  struct ptlrpc_sec     *sec;
  struct ptlrpc_ctx_ops *ctx_ops;
  unsigned long          expire_time;
  unsigned long          flags;
}
```

Each ptlrpc_cred linked into the hash table which *sec* point to. The *flags* indicate following context status:

- *NEW*: newly created.
- *UPTODATE*: valid and usable.
- *DEAD*: expired or somehow invalidated.
- *ERROR*: some fatal error happened during refresh or handling.

Once a context is marked *DEAD* or *ERROR*, it will not be able to return to other status, only will be destroyed after its last user release it.

The *ctx_ops* is a function set roughly as following:

int match(struct ptlrpc_cli_ctx *ctx, struct vfs_cred *vcred);

Return 1 if the user represented by @vcred matches with @ctx, otherwise return 0.

int refresh(struct ptlrpc_cli_ctx *ctx);

Refresh the given context. Usually it's asynchronous, only start the procedure of refreshing. Upon return the @ctx is not guaranteed be uptodate. Actually this function should return as soon as possible once the refresh get started. Return non-zero if any error happens.

void destroy(struct ptlrpc_cli_ctx *ctx);

Destroy the given context. Caller should guarantee there's no users hold the @ctx.

int sign(struct ptlrpc_cli_ctx *ctx, struct ptlrpc_request *req);

Using @ctx to make signature upon outgoing request message of @req->rq_reqlen bytes at @req->rq_reqmsg. Return 0 if succeed, otherwise error code. After successfully returned, size @req->rq_repdata_len bytes of data at @req->rq_repbuf is the final message will hit wire.

int seal(struct ptlrpc_cli_ctx *ctx, struct ptlrpc_request *req);

Using @ctx to encrypt the outgoing request message of @req->rq_reqlen bytes at @req->rq_reqmsg. Return 0 if succeed, otherwise error code. After successfully returned, size @req->rq_repdata_len bytes of data at @req->rq_repbuf is the final message will hit wire.

int verify(struct ptlrpc_cli_ctx *ctx, struct ptlrpc_request *req);

Using @ctx to verify the signature of reply message received of @req->rq_nob_received bytes of data at @req->rq_repbuf from server. Return 0 if succeed, otherwise return error code. After successfully return, @req->rq_replen bytes of data at @req->rq_repmsg is the actual reply lustre message.

During execution of *verify()*, if it found server return some special policy-specific error code which notify client that the credential used for the request has expired, it might return success but set @req->rq_resend as 1, in this case caller should try to obtain another credential and resend the request.

int unseal(struct ptlrpc_cli_ctx *ctx, struct ptlrpc_request *req);

Using @ctx to decrypt the reply data received of @req->rq_nob_received bytes of data at @req->rq_repbuf from server. Return 0 if succeed, otherwise return error code. After successfully return, @req->rq_replen bytes of data at @req->rq_repmsg is decoded reply lustre message.

During execution of *unseal()*, if it found server return some special policy-specific error code which notify client that the context used for the request has expired, it might return success but set @req->rq_resend as 1, in this case caller should try to obtain another credential and resend the request.

3.2 context management

All context related to a import are linked into hash table in a corresponding *ptlrpc_sec*. Security API layer provide functions to manage the hash table.

An issue: on an extremely heavily loaded system, suppose we have > 10k users accessing lustre filesystem, the hash table can't scale with it since we use a fixed hash size. [maybe use something like rbtree???]

A context can only be refreshed once. The thread which create a ptlrpc_cli_ctx structure should start the refresh procedure by calling ctx_ops->refresh().

3.2.1 garbage collection

At any time there might be some contexts get expired. There's no separate thread which periodically do the garbage collection and remove dead context etc., instead we check context on-the-fly when the credential hash table is accessed.

- Each time when we get a context from hash table, we need check whether its expired or not.
- Each *ptlrpc_sec* has a field "garbage_collect_time", indicate the next time of date when we should perform garbage collection. Each time when we access hash table, if current time is after the "garbage_collect_time", it should scan the whole hash table and release dead contexts, and set "garbage_collect_time" to next proper time.

3.2.2 logic of get_ctx()

The *get_ctx()* is the function we obtain a context for a certain user, and it's also the heart of context management.

```
struct ptlrpc_cli_ctx *
get_ctx(struct ptlrpc_sec *sec,
        struct vfs_cred *vcred)
{
  /* check garbage collection */
  if (current_time > sec->garbage_collect_time) {
    scan_clean_dead_ctx(sec);
    sec->garbage_collect_time = next_proper_gc_time;
  }
  /* check expire of each encountered entry */
  for_each_entry_in_hash(entry) {
    if (entry_is_expired_check(entry))
      continue;
    if (entry->match(entry, vcred))
      break;
  }
  if (!found) {
    entry = create_new();
    entry->refresh();
  }
  return entry;
}
```

A normal context's life time should be at least several hours, and usually not too many credentials in the hash table, so the function should mostly return quickly.

3.2.3 asynchronous refresh

The *ptlrpc_request* is added a new field of *struct list_head*, which link the request on the *ptlrpc_cli_ctx* it is waiting for; also *ptlrpc_cli_ctx* has a chain to hold the request list.

For normal RPCs which performed in its own thread context, it usually wait infinitely the refresh return. Before it go to sleep the *ptlrpc_request* should be linked into the waited *ptlrpc_cli_ctx*, and the thread be added into *request->rq_reply_waitq* (here we simply re-use *rq_reply_waitq* instead of adding a new one).

For RPCs performed by ptlrpcd, send_new_req() will set rq_waiting as 1 and simply return if the cred is not uptodate; the following check_set() should take care of this situation, make sure the context is uptodate before really send out request.

The thread which set the ptlrpc_cli_ctx state is responsible to wake up all threads waiting on this context. If any fatal error happened during refreshing, the associated requests's rq_ctx_err should be set to indicate caller abandoning this request. To avoid a refresh procedure never be finished, the security policy module should implement its own timeout mechanism.

The main logic of *ptlrpcs_req_refresh_ctx(req, timeout)* will be:

```
struct ptlrpc_cli_ctx *ctx = req->rq_cli_ctx;
while (1) {
  if (!ptlrpcs_ctx_check_uptodate(ctx))
    return 0;
  if (ctx->flags & CTX_ERROR)
    return -EPERM;
  add_into_wait_list(req, ctx);
  if (timeout < 0)
    return -EWOULDBLOCK;
  rc = l_wait_event(req->rq_reply_waitq, check_ctx());
  del_from_wait_list(req, ctx);
  if (ctx->flags & (CRED_ERROR|CRED_UPTODATE|CRED_DEAD))
    return rc;
}
```

3.3 import forced reconnect

After a successful connect on an import, security policy module should find out when the root context which used for the connection expire, and set *import->imp_next_reconnect* as the next time when we need reconnect this import by force, with a new credential.

The *pinger* thread should check next reconnect time on each import, if the time is up, *ptlrpc_connect_import()* will be called by force, even if the import status is *FULL*. After the reconnect finished, security policy will set the next reconnect time again for this import.

3.4 null policy

Only one *ptlrpc_sec* and one *ptlrpc_cli_ctx* structure will be statically allocated and initialized at module loading time, which will be shared by all users, including reverse contexts. The context will always be valid, never expire, thus no upcall is needed.

3.4.1 normal RPC

No data transform will be performed upon request & reply RPC messages, no embedded message structure at all.

3.4.2 bulk RPC

The *SIGN* and *PRIV* mode are not supported.

To maintain wire-data compatible, there's also no embedded message structure. If checksum is enabled, an extra segment will be appended at the tail of the request/reply RPC message, and should contain informations like checksum algorithm, data length, or other necessary ones.

4 State Management

Structure *ptlrpc_cli_ctx* might be concurrent accessed by multiple threads, its status need to be protected by locks.

Only one OBD_CONNECT could be issued on a import at a time. Before pinger send out a forced reconnect, make sure there's no other CONNECT is ongoing, which could be done by check whether import status is *FULL* or not. If it's not *FULL*, someone else must has taken care of this import, thus pinger should give up reconnect.

There's no recovery issue need special attention.

5 Environment

The *null* policy apply zero data transform, so the wire data is compatible between new and old nodes, and there’s no real authentication between client and server, all the security stuff is done within its own node, so both old client and new server and vice versa should be no problem.