Security API & Null Policy HLD

Eric Mei

2006-03-06

1 Requirements

- General framework of Secure PTLRPC (*sptlrpc*).
- A policy module which implement the current "no security" scenario, i.e. *null* policy.
- Be able to support GSS and other future policies.

2 Functional Specification

2.1 Overview

The PTLRPC security is at per-user basis. For each individual user, before he could use normal ptlrpc service to communicate with server side, he must authenticate with server and then establish a security context between the two peers. It like a virtual private rpc channel, although all these channels actually use the same "physical" PTLRPC connection. All following rpc messages will be sent/received within this context, in one of the 3 forms:

- *none*: no data transform applied.
- *auth*: a checksum is attached to protect the message's integrity, and also prove the genuine sender.

- *priv*: message is encrypted, and also prove the genuine sender.

Lustre should be able to use the wide range of existing authentication mechanisms, in most cases it should be mutual authentication between the user at client node and the server. We hope most security enforcement be applied within PTLRPC, and transparent to upper layer. Besides normal performance & administration penalty, Lustre security structure should be able to scale to large clusters.

The overall hierarchical structure is like following graph:

The *Security API* is a thin layer which provide security service to PTLRPC, and is actually tightly incorporated into PTLRPC. What it does is:

- Provide a set of security service APIs which be called throughout of PTLRPC.
- Implement some general facilities which are needed by all security policies.
- Dispatch security calls to the actual underlying policy modules.

Each security policy should implement a internal security function set which is prescribed by the general security API layer, in order to achieve its own security scheme. We should at least design 2 policies:

- *null*: It actually didn't apply any security check at all: no authentication, no data protection. It act as a fall-back to current situation, and should be wire-compatible with precedent versions of Lustre.
- *gss*: It's a subset (and modified) of normal user-space GSSAPI implementation in kernel. It doing the following:
  - Implement the function set prescribed by general security API.
  - Implement general facilities for all underlying security mechanisms.
  - Dispatch security calls to correct underlying security mechanisms.

Each security mechanism should implement function set prescribed by GSS policy, in the mechanism-specific way. Usually the real data transform, signature checking etc happened in this layer. Our first target mechanism is Kerberos 5.

We have another effort which will take care of the privacy of bulk data transfer and keep on-disk file data encrypted on OST. But here the security API also provide extra support for security of bulk data transfer: it could be checksumed to ensure data integrity, signed to ensure authenticated origin and data integrity, or encrypted to ensure privacy (only for on-wire privacy, on-disk data still in clear text).

In this HLD we mostly focus on the general security API layer, and the internal of *null* policy. Details about GSS and Kerberos 5 mechanism will be discussed in a separate HLD.

2.2 The general security APIs

As general description:

- Each import owns a *ptlrpc_sec* structure, which describe necessary security facilities exclusively used by this import.
- Each request on client node must associate with a security context, refered by structure *ptlrpc_cli_ctx*, which has to be refreshed and valid before be use for RPC.
- Each request on server node must also associate with a security context, refered by structure *ptlrpc_svc_ctx*, which is established during initial context negotiation.
- Security modules will be responsible to allocate & free rpc message buffers, because different security flavor might have special requirement on the message buffers.

2.2.1 Security policy registration

1. *sptlrpc_register_policy(ptlrpc_sec_policy *)*

   Can not sleep.

   Register a security policy module. It should be called by each policy module at initializing time (module loading, etc.).

2. *sptlrpc_unregister_policy(ptlrpc_sec_policy *)*

   Can not sleep.

   Unregister a security policy module. It should be called by each policy module at finalizing time (module unloading, etc.).

2.2.2 Import

1. *int sptlrpc_import_get_sec(obd_import *)*

   Might sleep on memory allocation.

   Given an import, obtain a ptlrpc_sec structure and associated with the import.

2. *void sptlrpc_import_put_sec(obd_import *)*

   Might sleep on memory allocation.

   Detach the ptlrpc_sec structure from a import, and usually destroy it.

3. *int sptlrpc_import_check_ctx(obd_import *)*

   Might sleep for a long period waiting on RPC completion.

   Find out whether current user has a valid context to this import's peer server.

2.2.3 Client request

1. *int sptlrpc_req_get_ctx(ptlrpc_request *)*

   Might sleep on memory allocation.

   Obtain a context of current user and associate with a request.

2. *void sptlrpc_req_put_cred(ptlrpc_request *)*

   Might sleep on memory allocation.

   Detach the credential from a request.

3. *int sptlrpc_req_refresh_cred(ptlrpc_request *, timeout)*

   Might sleep for a long period waiting on RPC completion.

   Start refreshing credential of a request, caller could choose to wait it finish, wait certain amount of time, or no wait at all.

4. *int sptlrpc_cli_alloc_reqbuf(ptlrpc_request *)*
5. *int sptlrpc_cli_alloc_repbuf(ptlrpc_request *)*

   Might sleep on memory allocation.

   Allocate request or reply buffers for a request.
6. *void sptlrpc_cli_free_reqbuf(ptlrpc_request *)*
7. *void sptlrpc_cli_free_repbuf(ptlrpc_request *)*

   Can not sleep.

   Free request or reply buffers for a request.
8. *int sptlrpc_cli_wrap_request(ptlrpc_request *)*

   Might sleep on memory allocation.

   Perform security transform upon request message which about to be sent out.
9. *int sptlrpc_cli_unwrap_reply(ptlrpc_request *)*

   Might sleep on memory allocation.

   Perform reverse security transform upon reply message received.

2.2.4 Server handling

1. *int sptlrpc_svc_unwrap_request(ptlrpc_request *)*

   Might sleep on memory allocation.

   Perform security verify/transform upon a incoming request, as long as handling context initialization & destruction request.
2. *int sptlrpc_svc_alloc_rs(ptlrpc_request *)*

   Might sleep on memory allocation.

   Allocate reply state for a request.
3. *int sptlrpc_svc_wrap_reply(ptlrpc_request *)*

   Might sleep on memory allocation.

   Perform a security transform upon a reply which will be sent out.

4. *void sptlrpc_svc_free_rs(ptlrpc_reply_state *)*

   Can not sleep.

   Free reply state buffer.

5. *void sptlrpc_svc_cleanup_req(ptlrpc_request *)*

   Can not sleep.

   Cleanup security service data which associate with a request.

2.2.5 Bulk transfer

1. *int sptlrpc_bulk_write_cli_wrap(ptlrpc_request *, ptlrpc_bulk_desc *)*

   Might sleep on memory allocation.

   Called by client side during bulk write, to perform checksum, signature, or encryption on the outgoing bulk data, and pack the result into request.

2. int sptlrpc_bulk_write_cli_verify(ptlrpc_request *, ptlrpc_bulk_desc *)

   Might sleep on memory allocation.

   Called by client side during bulk write, to verify there's no mismatch between the client checksum and server checksum, or verify server signature is correct, or decrypt data.

3. int sptlrpc_bulk_write_svc_unwrap(ptlrpc_request *, ptlrpc_bulk_desc *)

   Might sleep on memory allocation.

   Called by OSS side during bulk write, to verify there's no mismatch between the client checksum and OSS computed, or verify client signature is correct, or decrypt data, and pack necessary result into reply.

4. int sptlrpc_bulk_read_svc_wrap(ptlrpc_request *, ptlrpc_bulk_desc *)

   Might sleep on memory allocation.

   Called by OSS side during bulk read, to perform checksum, signature, or encryption on the outgoing bulk data, and pack the result into reply.

5. int sptlrpc_bulk_read_cli_unwrap(ptlrpc_request *, ptlrpc_bulk_desc *)

   Might sleep on memory allocation.

Called by client side during bulk read, to verify there's no mismatch between the server checksum and client computed, or verify server signature is correct, or decrypt data.

3 Use Cases

3.1 A full RPC cycle on client side

1. In context of user A, an RPC to server node is needed, a ptlrpc_request is created.
2. call sptlrpc_req_get_cred() to obtain an credential for user A.
3. call sptlrpc_cli_alloc_reqbuf() to allocate request message buffer.
4. request data is filled into request buffer.
5. call sptlrpc_req_refresh_cred() to refresh associated credential until we get a valid credential.
6. call sptlrpc_cli_wrap_request() to perform security transform upon request message.
7. call sptlrpc_cli_alloc_repbuf() to prepare reply buffer.
8. send out the rpc.
9. got reply from server.
10. call sptlrpc_cli_unwrap_reply() to perform security transform upon received reply message.
11. interpret the reply, finally to destroy this ptlrpc_request.
12. call sptlrpc_cli_free_reqbuf() and ptlrpcs_cli_free_repbuf() to release request/reply message buffer.
13. call sptlrpc_req_drop_cred() to release current credential.
14. destroy ptlrpc_request structure.

3.2 A full RPC cycle on server side

1. a request arrives, call sptlrpc_svc_unwrap_request() to verify/transform the request message.
2. interpret the request.
3. call sptlrpc_svc_alloc_rs() to allocate reply state and reply message buffer.
4. fill reply data into reply buffer.
5. call sptlrpc_svc_wrap_reply() to perform security transform upon reply message.
6. send reply out.
7. call sptlrpc_svc_cleanup_req() to cleanup security service data associated with the ptlrpc_request.
8. destroy ptlrpc_request structure.
9. call sptlrpc_svc_free_rs() to finally release reply state and reply message buffer.

3.3 Protect local cached data on client

1. A user's context get expired after some period.
2. This use access a local cached object, find a matched DLM lock.
3. call sptlrpc_import_check_ctx() to find out this user's credential to corresponding server has expired.
4. release DLM lock and deny the access.

3.4 File read checksum

1. Client: simply go through normal path, the underlying security policy will pack the checksum flag into request, and send out request.

2. OSS: prepare reply buffer, read data off disk.
3. OSS: call sptlrpc_bulk_read_svc_wrap() to pack checksum into reply.
4. OSS: start & finish bulk transfer, send reply back.
5. Client: call sptlrpc_bulk_read_cli_unwarp() to verify checksum.

3.5 File write checksum

1. Client: before send out request, call sptlrpc_bulk_write_cli_wrap() to pack checksum into request, and send out request.
2. OSS: prepare reply buffer, start & finish bulk transfer.
3. OSS: call sptlrpc_bulk_write_svc_unwarp() to verify checksum, which also pack bounce checksum into reply.
4. OSS: write data on disk.
5. Client: call sptlrpc_bulk_write_cli_verify() to bounce checksum, give warning if found mismatch.

4 Logic Specification

4.1 Data structures

A *ptlrpc_sec_policy* is a implementation of a certain security policy, by a set of functions which conformed to the internal security API.

A *ptlrpc_cli_ctx* is a certain user's security context to a certain server node. The main fields are: user identity, expire time, mechanism-specific data, etc.

A *ptlrpc_svc_ctx* represent the half of a security context on server side.

A *ptlrpc_sec* represent the overall security facility which apply on this ptlrpc connection. It mainly hold a hash table of ptlrpc_cli_ctx of various users.

- Each import owns one and only one *ptlrpc_sec*.

- On client, each ptlrpc_request associates with one and only one *ptlrpc_cli_ctx*; A *ptlrpc_cli_ctx* might be used by multiple ptlrpc_request, thus by multiple thread, at the same time.
- *ptlrpc_cli_ctx* can not be recycled: after a context get expired, it will be released, and a new *ptlrpc_cli_ctx* will be create and refreshed.
- One system user could have multiple *ptlrpc_cli_ctx* at the same time, each one correspondant to a different server node.
- On server, each incoming request will be associate with a correct *ptlrpc_svc_ctx* which will be used throughout the lifecycle of the request.
- General security API layer will implement a simple algorithms to maintain credential hash tables, periodically scan and reap expired credentials, etc. .

At server side, there's no specific structure corresponding to each peer *ptlrpc_sec*, all the security contexts are created and cached by a central cache management system. As a matter of fact, we use in-kernel cache service from NFSv4, so we put all of that in GSS policy module, the API layer didn't do anything on this.

4.2 Wire data format

The security API layer don't specify the details wire data format, it is different from policy to policy. But the overall rule is every on-wire RPC message must conform to the structure *lustre_msg*. One lustre_msg could be embedded into another one by just becoming a data segment of the "container" lustre_msg. So if any security policy want to add its own data on a RPC, it need construct a container lustre_msg, like following:

```
struct lustre_msg container {
    ...
    bufcount = n;
    bufs[0] = struct lustre_msg embedded {
        ...
        bufcount = m;
        bufs[0] = ...;
        bufs[1] = ...;
```

```
        ...
        bufs[m-1] = ...;
    };
    bufs[1] = security payload 1;
    bufs[2] = security payload 2;
    ...
    bufs[n-1] = ...;
}
```

4.3 Context negotiation

The simplest context negotiation process is roughly like following:

1. Client side prepare a bunch of data, sent to server.
2. Server side verify the request, install a context, and prepare a reply send to client.
3. Client side verify the reply, install a context.

After that, client and server will have agreement on a pair of security context. Depend on the actual authentication scheme used, usually it require one or more message exchange between client and server node to let a security context be established.

In most cases, the security context negotiation are complex enough that can't totally fit into kernel, so we have to resort to user space tools. The content of the negotiation messages are specific to certain authentication mechanism, and in a secure manner.

We'll use the normal kernel PTLRPC service to perform the message exchange, and also the corresponding security policies in kernel might want to pack/unpack the RPC, to encode/decode with its own special data.

Those RPCs are special, it could happen when an import is in any status except *CLOSED*, the normal ptlrpc call path should be modified to support it. In error cases, context negotiation RPCs will return "fatal" or "non-fatal" to caller. The security API layer only provide the message transportation, caller itself should responsible to detect & recovery from errors/attacks like data be snooped, modified, or lost.

4.4 RPCs through a secure channel

After a successful context negotiation, server will cache a context half (*ptlrpc_svc_ctx*) and return a handle to client; client also cache a corresponding context half (*ptlrpc_cli_ctx*). For each following normal RPC, client will perform security transform upon the request (signing or encrypting) using *ptlrpc_cli_ctx*, and send out with the context handle to server.

Server at first find the cached *ptlrpc_svc_ctx* by received context handle, and perform reverse security transform (verify signature or decrypting) upon the request. Before the reply be sent out, *ptlrpc_svc_ctx* will again be used to signing/encrypt the reply message, and finally send out.

Client again will use *ptlrpc_cli_ctx* to verify/decrypt the reply. Here we don't need handle because one ptlrpc_request only could associated with one client context.

4.5 Context refresh

Context is refreshed by demand: it happened when a RPC is constructed and no valid context has been found, usually that's the first time of a user sending RPCs to certain server node. Usually refreshing happens between *RQ_PHASE_NEW* and *RQ_PHASE_RPC*, in rare cases it might also happens within *RQ_PHASE_RPC*. We don't define new phases of RPC, to keep minimal changes of current code.

It might take a long time to refresh a context, so *sptlrpc_req_refresh_ctx()* provide both "sync" and "async" modes. The RPCs which go through *ptlrpc_queue_wait()* normally refresh context in sync mode, in which caller put itself into waiting queue until be waken up when the refresh finished or error happened. Some critical threads like *ptlrpcd* should never be blocked by refreshing a single context, so it must be called in async mode. The *ptlrpcd* only make sure the refresh has started, put the request into a notification list of the context, and return to the main loop to check next on-list request. When something happen on the context, it will go through all the request in the notification list, and wake them up. If the request belongs to a *ptlrpc_set* then the set manager (i.e. *ptlrpcd* in this case) will be waken up.

Refreshing a context might return various kind of errors, some of them are fatal, e.g. the user has been rejected by server; some are transient problems, e.g. temporarily network partition. The API layer should treat them differently. In

former case, all *ptlrpc_request* waiting this context will abort and return failure immediately, otherwise issue another refresh.

4.6 Context timeout

We don't track each context's expire time and do refresh before certain amount of time ahead of that. We just keep using it until we found its expired and then obtain a new context. It's bad that a context get expired between sending request and receiving reply which thus can't be verified. Actually it's not necessary to let a context get expired at its exact expire time. Our rule is: when a ptlrpc_request first time found its context is valid, it will be able to keep using it until the RPC finished and release the context. And for this reason, timeout is totally managed by API layer, security policy module don't do any further expiry checking when performing data transforming.

Because system time might be different between nodes, it's possible that a security context get expired on server but client doesn't know it. In this case server will return a special error, policy module's *unwrap_reply*() should detect it and notify upper layer resend the RPC. This also could happen when server crashed and come up again, all the previous security contexts have been lost.

4.7 Reverse context

We don't simply use normal context negotiation for reverse connection, because:

- It will not add any extra security strength.
- What we need is just a secure data extrange channel, extra authentication has no point.
- It could add non-negligible administration and performance overhead.

Reverse import also owns a *ptlrpc_sec* structure, and there's at least one (and uaually only one) context of root user, because DLM callback only be issued within root's context. This context is directly derived from the normal context at *OBD_CONNECT* time, instead of using normal security negotiation, and has the same expiry as its sibling context.

On client side, *pinger* thread could be use to check every import periodically. When it found root's context of an import is about to expire, a new context will be obtained and issue a *OBD_CONNECT* using the new context, thus server will have the chance to refresh the context in its reverse import.

Client should do the *OBD_CONNECT* at least earlier than the maximum allowed time difference between nodes before the context expiry time, otherwise server might have a period which has no valid context available thus possibly lead to the client be evicted.

4.8 Managing message buffers

Now the message buffer allocation and free are delegated to policy layer. The policy module could freely arrange the buffer layout, but has to conform to following rules:

- The buffers must be big enough to hold all the data, including security payload, for request and reply.
- Provide consistant view to PTLRPC layer:
  - After sptlrpc_cli_alloc_reqbuf(), the request->rq_reqmsg must point to a buffer which allow caller fill in request message.
  - After sptlrpc_cli_unwrap_reply(), the request->rq_repmsg must point to the clear text of reply message sent from server.
  - After sptlrpc_svc_unwarp_request(), the request->rq_reqmsg must point to the clear text of request message sent from client.
  - After sptlrpc_svc_alloc_rs(), an reply_state structure must be allocated, and request->rq_repmsg must point to a buffer which allow caller fill in reply message.
- All the allocation and free must most mess with pre-allocated urgent buffers, on both client and server.

4.9 Protect locally cached data

In MDC and OSC layer, whenever we found a matched DLM lock, we need also check whether current user owns a valid context to server node. If not, we create a new credential and try to refresh it, until we have a valid credential or encountered fatal errors. In latter case the DLM lock will be released and return failure to caller.

4.10 Secure reply ACK

Currently the ACK of reply in LNET layer, which some transaction depend on, is insecure. Now we replace it with a higher level ACK in PTLRPC layer, which is guaranteed to be secure.

At server side, each "difficult" reply state has a ACK number, which is unique within an export. This number could be a simple ever increasing natural number, and the all the reply_state of an export somehow get sorted and could be indexed by the ACK number. This ACK number is filled in reply message and send to client.

At client side, all pending ACK numbers are record inside import. For normal outgoing request, it pack the pending ACK numbers into the request, and be protected by the security transform. The ACK numbers should be separate kept in case of resend. Then server could finally release those pending reply_state and associated locks.

The ACK notification should not be going with context negotiation RPC because not secure at that time, and also not OBD_CONNECT (even in recovery) because connection not supposed to be established yet.

For a very inactive connection, the longest time that a reply_state hanging on server side is the *obd_timeout*, when the next pinger will arrive with ACK notification. (***FIXME: whether it's acceptable to delay the ACK notification to next time pinger wait? How bad will this affect server?***)

For some RPC which is marked as "no resend" (e.g. ping), if the RPC failed we should put back the ACK numbers it carrying to import, in order to allow other requests pick them up later.

Note here we has an assumption that we trust that client kernel is not modified. The ACK notifications could be sent with different poeple's context, which the server has to trust.

It would be the best to allow both two kind of ACK. The *null* policy applies no security, and it need to 100% compatible with old version of Lustre, so we should be able to allow PTLRPC connections running with *null* policy still use the LNET layer ACK; while other connections with strong security automatically switch to PTLRPC layer ACK mechanism.

4.11 Protect bulk transfer

Currently lustre has a CRC32 checksum implementation against bulk I/O, now we'll replace it with a more general mechanism in security API layer. The shared secret between the secure ptlrpc connection peers can't be used to encrypt on-disk file data, because there's no permanent keys available. But we can use it to protect file data transfering across insecure network. There's following grades of protection:

- *NONE*: no protection.
- *CSUM*: exchange checksum of bulk data, make sure data integrity.
- *SIGN*: signature on bulk data, make sure the authenticated origin and data integrity.
- *PRIV*: encrypt bulk data.

In *CSUM* or *SIGN* mode, the computed checksum or signature will be packed into RPC's security payload, transparent to upper layer lustre_msg, bulk data will remain unchanged; In *PRIV* mode, encrypted file data will be transfered as bulk data, but extra related data will be packed into RPC's security payload. Something special in *PRIV* mode:

- *write*:
  - *client*: need allocate extra pages to hold the encrypted data, which participate in the bulk transfer.
  - *server*: decrypt could be done in-place.
- *read*:

- *client*: decrypt could be done in-place.
- *server*: need allocate extra pages to hold the encrypted data, which participate in the bulk transfer.

Besides CPU overhead, the extra pages in *PRIV* mode should not bring prohibited memory pressure, because normally at any given time, the percent of pages which during I/O should not be too high.

We perhaps could encode a flag into security flavor which indicate whether it need some kind of protection on bulk I/O, thus underlying security policy will pack all the data we need and be transparent to upper layers.

4.12 Impact on recovery

As described abovely, the security API has minimum changes on recovery.

For context negotiation RPCs: If they are sent before the import connected or in recovery status, then the failure will not trigger any recovery event; If they are sent when the import is connected, the failure may trigger recovery on the import; In any case, the context negotiation RPC should never be automatically resent, just return error to caller and let caller decide how to proceed.

An request might be resend multiple times, and it could be handed to security module for wrapping for several times. Each ptlrpc_request has a flag to indicate whether the security transform has been applied on the message or not, thus the security policy module could take different action on the message according to the flag.

In case of server reboot, all the old contexts will be lost, thus server can't unpack any RPCs from client nodes, including *OBD_CONNECT* request. In this case server will return an error notification, but must in clear text. Client then flush the old context, establish a new one and continue the rest of recovery.

4.13 Possible attacks

4.13.1 Replay attack

The *null* policy is vulnerable to replay attack, because itself is not secure at all. GSSAPI standard required a mechanism described in RFC 2203 to be implemented to prevent replay attack. We'll describe it in another HLD.

4.13.2 Snoop or modifying data

For a real security context upon a connection implemented under GSS or other policies, snoop could be prevented by encrypt data, and both signature and encryption could prevent data be modified by a man in the middle. Again the *null* policy is of course vulnerable to the attack.

4.13.3 DOS attack

The only clear text sent across network without protection is the error notification from server to client, described in section 4.10. A malicious guy could intercept reply and forge the error messages to force client flush old contexts, thus achieve a kind of DOS attack.

There are probably many other kinds of DOS attacks. For example, a user repeatedly create tons of security negotiation RPCs from many client nodes to a single server, to cause server be overloaded.

Many security system can't resist DOS attack by itself, so does Lustre security API. It could be addressed by some extra mechanisms not covered by this HLD.

4.14 *null* policy

The *null* policy module will be extremely simple:

- All users share a single *ptlrpc_cli_ctx*, which never expired, no refresh is needed, no error notification will be sent by server nodes.
- All imports share a single *ptlrpc_sec* structure.
- Security transform is actually did nothing upon request/reply messages, no extra checking either.
- Still use reply ACK service from LNET layer, no ACK notification needed.
- Support "NONE" and "CSUM" mode for bulk I/O.
- The wire protocol is 100% compatible with current versions of Lustre.

5 State Management

The *ptlrpc_cli_ctx* might be concurrently accessed by multiple threads, make sure the operations, e.g. checking expiry etc., are atomic.

No disk format changes involved.

No serious impact on recovery.

6 Alternatives

7 Focus of Inspection