Lustre Client GSS with Linux Keyrings

Eric Mei <ericm@clusterfs.com>

2008.07.22

1 Introduction

This document describes how Lustre use keyring to populate and manage the client gss context, to replace current mechanism which use pipefs.

2 Requirements

- Use keyring on Lustre client for gss/kerberos context acquiring.
- Try to still keep pipefs mechanism a functional option.
- Although keyring is only available in Linux at this moment, we need to keep as much portable as we can.
- Make it possible to support the special authentication requirement from NRL.

3 Functional Specification

3.1 Keyring Services

Keyring is relatively new to Linux kernel. Although it's carefully designed & implemented, there's no real user of keyring so far in or outside of kernel so far as we know. So there's chances that some of keyring doesn't fit Lustre very well. We have to "work around" it, patching the kernel is not acceptable at least for now.

Keyring provides following services for distributed file systems:

- Kernel upcall into user space, this is generally used by obtaining secure user context.
- System call API & user space tools to manipulate keys.

- In-kernel key management.
- Well-organized thread/process/session keyrings. This may help us implement PAG functionality in the future.

3.2 Authentication at Navy Research Labs

NRL has following situations about Kerberos authentication:

- No file-based credential anymore. A customized credential cache that talks to a special credential cache server started by the user as part of the initial login process is used.
- Only processes that are children of the initial login process can access the user's credentials.

3.3 Lustre GSS with Keyrings

- A new instance of GSS policy will be defined. It share the same *ptlrpc_sec_sops* with current one, but redefine *ptlrpc_sec_cops* and *ptlrpc_ctx_ops* to use services of upcall and context management from keyring. We don't need our own hash table of contexts anymore.
- Current client side daemon *lgssd* will be obsolete.
- A user space program *lgss_keyring* will be implemented, which is called when kernel request a new security context. This program should finish all GSS/Kerberos context negotiation, and feed the final data back to the kernel.
- To work in computing environment at NRL, some extra stuff is needed, but we don't implement them immediately until we are told to:
  - A user space tool will be implemented, which manipulating Kerberos credentials just like what *kinit* is doing, but store in kernel space via keyring interface.
  - The *lgss_keyring* should be modified that store/retrieve any credential into/from kernel via keyring interface.
  - In kernel, key request function should be able to use in-kernel Kerberos TGT as a temporary authentication key.

4 Use Cases

4.1 Create a new context

- Upper level request a new *ptlrpc_cli_ctx*.
- Keyring search existed contexts, create a new key K.
- Keyring issue an upcall to *lgss_keyring* to refresh K.
- The *lgss_keyring* negotiate with GSS server, and pass down the final context data into kernel, then exit.
- Keyring instantiate K, as well as the *ptlrpc_cli_ctx* structure.
- The new *ptlrpc_cli_ctx* returned back to upper layer.

4.2 Create a new context timeout

- Upper level request a new *ptlrpc_cli_ctx*.
- Keyring create a new key K, issue an upcall to *lgss_keyring* to refresh K.
- The *lgss_keyring* stuck with network, or get killed before finish.
- After waiting certain time, keyring revoke K and return error to upper level.

5 Logic Specification

5.1 General Considerations

In general, we keep current security API mostly unchanged and fit keyring into the framework. Current pipefs mechanism will be transformed to one kind of gss policy instance, and keyring will become another instance. Some kind of user-supplied option to determine which instance to use.

- Structure *key* could be considered a header managed by general keyring code. Structure *ptlrpc_cli_ctx* and underlying mechanism data will be separately allocated, and point to each other. Destroy a key will lead to destroy associated *ptlrpc_cli_ctx* as well.
- Upper layer only see *ptlrpc_cli_ctx*, they have no idea whether keyring or pipefs are used. It also means that a small amount of context information, e.g. status, expiry, etc., will be duplicated in both *ptlrpc_cli_ctx* and *key*. We think it's acceptable as a small price of portability.

- Keyring treat a key with zero refcount as unused and intend to release it immediately. We have to hold a "base" reference for any key which still in use (ongoing initiation, initiated, ongoing destroy), and drop the reference when we find it is not usable anymore.
- Keyring trees are global, so UID is not enough anymore to label a key. The unique ID should somehow composed by part of UID, target UUID, client UUID, connect count, etc..
- All *ptlrpc_cli_ctx* should be linked together, or linked into associated *ptlrpc_sec* structure, to be iterated through easily. Because keyring doesn't have interface to operate on multiple entries in one iteration.
- Context hash table which now lives in structure *ptlrpc_sec* should be moved to another place, as well as the associated management functions, thus pipefs (or maybe more) policy instance can utilize them.
- Reverse context on server has a special flag to indicate it doesn't associate with any *key*.

5.2 API definition

5.2.1 ptlrpc_sec_cops

Some of the API functions keep unchanged, but maybe add some sanity checking code which specifically related to keyring:

- create_sec() / destroy_sec()
- install_rctx()
- alloc_reqbuf() / free_reqbuf() / enlarge_reqbuf()
- alloc_repbuf() / free_repbuf()

Redefine following API functions:

- *lookup_ctx()*: Directly call key_request() to search, create, refresh a context.

Add following API functions:

- *release_ctx()*: Destroy the context. Called when the user drop the reference of ctx to 0.
- *flush_ctx_cache()*: Flush out contexts which satisfy the selected criterion

Remove following API functions:

- *create_ctx()*: Same as before, but make sure the context and key are associated with each other.
- *destroy_ctx()*: Same as before, but taking care of the association between context and key.

5.2.2 ptlrpc_ctx_ops

Some of the API functions keep almost unchanged, but again need more keyring related sanity checking:

- sign() / verify()
- seal() / unseal()
- wrap_bulk() / unwrap_bulk()

Redefine following API functions:

- *display()*: Also display associated key information.
- *match()*: Find the key, and compare.
- *refresh()*: Initiate keyring upcall to user space (which seems likely has been fired off at this point??).

Add following API functions:

- *validate()*: Call to determine the context is valid to be used.
- *die()*: Call to invalidate the context by force.

5.3 Asynchronous Upcalls

Thread performing keyring upcall won't return until the upcall finished, but we'd like the upcall be asynchronous. To work around it, when *lgss_keyring* is called, it forked into two processes; One process notify kernel immediately to instantiate the key and exit, with this chance kernel populate the key with general *ptlrpc_cli_ctx* data, and return to caller (Note at this moment keyring think the key has been instantiated, but Lustre know it's not); The other process continue the gss negotiation with server, and update kernel key with the final context data.

Only the parent process can assume authority of target key, which will get lost by fork(). So the parent process need obtain all the additional authentication data needed before fork the child.

One drawback of this mechanism is an extra *fork()* of *lgss_keyring* is needed, hope it's not too big deal because half of them are very short lived.

5.4 Upcall Timeout

Keyring has no built-in mechanism to timeout an ongoing upcall. To solve this, when key get first notification and populate generic *ptlrpc_cli_ctx* data, we set the key timeout a suitable value which act like a upcall timeout.

There's still a chance that the *lgss_keyring* process crashed before notifying kernel, thus the calling process will hang there forever. But firstly the chance is extremely small; Secondly we still can add our own timeout code before calling request_key().

TODO: coordinate the timeout with Adaptive Timeouts.

5.5 lgss_keyring

After the *fork()*, the parent notify kernel and exit immediately; The child firstly daemonize itself (completely detach from its parent), it might be necessary because we want to let kernel think the parent process is the all of the upcall, thus when parent process end the upcall is considered end. Then the child do context negotiation similar to lgssd, and parse context and feed final data back to kernel and exit.

The *lgss_keyring* will be called in root's context, upcall parameter should indicate which user we are serving. Also there are parameters about thread/process/session keyring of the caller, which allow us access auth_key and other related information of the real user. Maybe we can setuid to make it running a little safer, not so important though.

5.6 Expiry detection

Keyring has no active expiry detection mechanism. It only cleanup dead keys when found one key is dead. We mark the context dead when *key_validate*() return -EKEYEXPIRED, and drop the base reference of key. This leave a chance that a key without user get expired but still be cached until another key dies. An alternative is a dedicated thread iterating through all contexts periodically.

5.7 Process Keyring control

By default key will be linked to process's session keyring if it present. This is not good for root user, because we want all root processes share the same key, no matter which session it belongs to, even in PAG mode. But we can use another behavior of keyring, which is when session keyring is not present, a default user session keyring will be used. So we can do following to solve this:

- When starting Lustre internal threads (ptlrpcd, ptlrpc service, pinger, etc.), drop current .session keyring.

- When normal root process (e.g. mount.lustre) request a key, we temporarily drop its session keyring; After we hold the key the original session keyring will be restored.

5.8 Cleanup Keys

Keys will be destroyed when its reference dropped to zero. But during key creation, it might be linked into various kind of keyrings which hold references on key and might not be released automatically when Lustre cleanup. So we'd better record which keyring the key linked to, and unlink them when revoking the key.

6 State Management

Will not hurt scalability, performance, or recovery. No wire or disk format changes.

7 Alternatives

N/A.

8 Focus For Inspections

N/A.