Lustre Client GSS with Linux Keyrings

Eric Mei <ericm@clusterfs.com>

2007.04.26

1 Functional Specification

- A gss policy instance with full set of API implementation is constructed.
- Keep old pipefs mechanism as another gss policy.
- Implement a key type under the Linux keyring framework.
- Use keyring upcall to user space to do gss/krb5 context negotiation.

2 Use Cases

2.1 normal context negotiation

1. upper layer request a new context.
2. request_key() is called to obtain corresponding key, which result a upcall be issued to user space.
3. user space tools do gss context negotiation with gss server, and return back to kernel.
4. a new key and coupling context is initialized, and context is returned to caller.

2.2 request an existed context

1. upper layer request a context.
2. request_key() is called to obtain corresponding key, which is found in the cache.
3. obtain the coupled context through the key.
4. return the context to caller.

2.3 upcall timeout

1. upper layer request a new context.
2. request_key() is called to obtain corresponding key, which result a upcall be issued to user space.
3. user space tools somehow crashed or stucked during negotiation.
4. after sometime, caller detect the timeout and invalid the key & context.

3 Logic Specification

3.1 General rules of using keyring

Keyring should be used only in gss policy layer, transparent to upper layers. Only normal general structures, like *ptlrpc_cli_ctx*, *ptlrpc_sec*, are exposed to users.

A mature client context (a ptlrpc_cli_ctx instance) is 1:1 correspond to a key, they each hold a reference of the other. Without intervene from outside, a context-key pair will never be released.

A context is linked into a single list on its belonging ptlrpc_sec, to facilitate iteration operations. A global dedicated kernel thread responsible to walk through all ptlrpc_sec periodically, cleanup dead contexts and release coupled keys.

Each key requires a unique string to describe itself. Instead of using excessive long string which comprise of target, client, connection generation, uid, etc., we assign a unique 32 bits *secid* to each ptlrpc_sec, and the description string would be as simple as "*uid@secid*".

Keyring upcall is totally synchronous which is unacceptable for Lustre. So we break the upcall into two phases: The upcall process instantiate key with null data and exit, this will make keyring consider the upcall has finished so the waiting threads could be unblocked; Then the forked the child process could proceed and finish the rest of gss context negotiation.

3.2 Process keyring control

Linux keyring service is fine grained with precisely thread/process/session/user controls. Plenty of subtle issues need to be considered carefully.

By default a constructed key is linked to process's session keyring. But in any case, we hope root user use a single security context for a certain ptlrpc connection. There's 2 reaons for this:

1. There could be _many_ lustre kernel threads, it doesn't make much sense to create unique context for each one.
2. Root user is supposed to authenticate with server automatically (with proper krb5 keytab credential installed). So be a root process already means the right to authenticate with server successfully.

So when root context was requested, we firstly search cached contexts directly without help of keyring facility; If we did't find one, then back to the normal way to request a key which lead to the context be constructed. The request has to be protected from multiple thread issue the request at the same time. Furthermore, we hope the root key is not linked to any keyring thus Lustre can totally control it. A little bit trouble is that Linux keyring will automatically link any key into some specific keyring, we'd better unlink it after the key/context was constructed.

For normal users we follow the default behavior, that is installing key on process's session keyring, which makes the key only visible for processes belonging to the same session. Note this result in the fact that more than one context for a certain user could be linked into one ptlrpc_sec, which coupled with different key but with the same key description. This need to be considerd when implementing "flush my context".

A key won't be released if it linked to a keyring. which may lead to lustre constructed keys hanging there forever. Lustre should precisely control what keyring a key linked to. But unfortunately they are mostly built-in Linux keyring implementation and not controlable by outsiders. As the result we have following behavior:

- Root key will not be linked to any keyring, as we described above.
- Normal user key will be linked to session keyring, which will only be break when this user session ended.

Keyring upcall process is executed in root context, with a temporary new session keyring linked by a authentication key, with which the process could assume the authority on target user context. But a important note a forked child process will lost the authority, so any needed authentication data should be prepared in parent process context.

For rootonly or reverse sec, we can't rely on key searching from linux kernel to obtain desired key in whatever case. We can choose only construct the context which doesn't associate with any key. An alternative is construct a fake one to unify the code.

3.3 PAG & NRL consideration

The above default behavior should be able to satisfy requirement from NRL. It also act like PAG has been enabled.

We perhaps should also provide alternative behavior as PAG disabled, which is no sessions based access control. To achieve this, we only need to what we did for root context to normal users: first search context directly, if not found then issue keyring upcall to populate one.

3.4 Context and sec structures

Same mechanism is used to context and sec structures.

```
struct gss_sec_keyring {
    struct gss_sec      gsk_base;           /* base data */
    int                 gsk_id;             /* unique sec ID */
    struct hlist_head   gsk_ctx_list;       /* all contexts linked here */
    struct mutex        gsk_upcall_lock;    /* protect concurrent upcall */
    ...
}
struct gss_cli_ctx_keyring {
    struct gss_cli_ctx  gck_base;           /* base data */
    struct key         *gck_key;            /* associated key */
    ...
}
```

3.5 Sec API

- *create_sec()*: Create gss_sec_keyring structure and initialize.
- *destroy_sec()*: Destroy gss_sec_keyring structure.
- *lookup_ctx()*: Drop related keyrings temporarily (as above description), construct target key description, call request_key() to obtain a key. If the key hasn't coupled with any context yet, we construct a context and couple them together.
- *release_ctx()*: Destroy the gss client context.
- *flush_ctx_cache()*: Destroy selected gss client context from context list.

3.6 Context API

- *refresh()*: Do nothing, because the upcall must already on the way.
- *validate()*: Verify the context is uptodate, remove expired/dead context.
- *die()*: Force a context to be expired.

3.7 Key type API

- *instantiate()*: Do nothing, because we can't get enough information to do anything here.
- *update()*: If called with context data, actually initialize the gss context; otherwise invalidate the context and key.
- *match()*: Simply compare the key description.
- *destroy()*: Do nothing. At this time the key and context must have been decoupled.
- *describe()*: Print out key/context information.

3.8 User space upcall

This user space tool is called with parameters indicating key ID, description, callout info, uid, gid, session keyring. The callout info is Lustre specific infomation, which at least include service type, target nid, target uuid. After fork to a child process to do the actual gss context negotiation, the parent process should exit shortly.

The main process proceed like following:

1. Parsing parameters.
2. Immediately notify kernel by *keyctl_instantiate()* with null data.
3. Obtain whatever information needed for the authentication. Currently we do nothing about this. In the future if user store its Kerberos 5 TGT in kernel, this is the chance that we can fetch the TGT.
4. Fork a child process to proceed further authentication.
5. Exit.

The child process proceed like following:

1. Call *setuid()*/*setgid()* to desired user. This is necessary because here we already lost the authorization to access original user context.
2. Call *keyctl_link()* to link the key to session keyring, which allow further negotiation RPC could find the correct key.
3. Prepare gss negotiation data, do negotiation RPC, just like what previously lgssd was doing.
4. Update kernel context data with *keyctl_update()*.
5. Call *keyctl_unlink()* to unlink the key to session keyring.
6. Exit.

3.9 Upcall timeout

A suitable timeout value is set on any pre-mature context, which will be checked by any process which accessing the context or by the gss worker thread.

A key during upcall is also set with an timeout value, which will be checked by any process which accessing the key.

4 State Management

4.1 Context & Key locking

Both key and context use refcount to track its usage, and be released when usage drop to 0. Because a coupled key and context each hold a reference on the other, so we can access a context through a key or access a key through a context without worrying be released.

Couple or Decouple a key/context pair is a rare happened operation, and it is protected by the key semaphore in write mode.

A context enlist/delist to/from its belonging ptlrpc_sec is protected by another spinlock.

5 Environment

Require installation & configuration of package *keyutils*, *libgssapi*.

No disk format changes.

No RPC wire protocol changes.