Common framing

MLS IETF 104 Prague

Goals and security properties

- Meta-data protection, roster can be derived from looking at HS messages
- Add messages: identity, credentials
- Remove messages: indexes

Conflicting goals

- Server assist needs at least public keys, possibly commit hashes of tree hash
- Needs to be optional in case the server should know the roster

Message framing

```
struct {
      opaque group_id<0..255>;
      uint32 epoch;
      uint32 sender;
      ContentType type;

      select (MLSPlaintext.type) {
      case handshake:
            Handshake handshake;
      case application:
            opaque application_data<0..2^32-1>;
      }

      opaque signature<1..2^16-1>;
} MLSPlaintext;
```

```
enum {
      invalid(0),
      handshake(1),
      application(2),
      (255)
} ContentType;
```

Message framing 1

```
struct {
      opaque content[MLSPlaintext.length];
      uint8 signature[MLSInnerPlaintext.sig_len];
      uint16 sig_len;
      ContentType type;
      uint8 zero_padding[length_of_padding];
} MLSInnerPlaintext;
```

```
struct {
      opaque group_id<0..255>;
      uint32 epoch;
      uint32 sender;
      uint32 generation;
      opaque cipertext<0..2^32-1>;
} MLSCiphertext;
```

Message framing 2: application message

```
struct {
      opaque content[MLSPlaintext.length];
      uint8 signature[MLSInnerPlaintext.sig_len];
      uint16 sig_len;
      ~~ContentType type;~~
      uint8 marker = 1;
      uint8 zero_padding[length_of_padding];
} MLSInnerPlaintext;

struct {
      opaque group_id<0..255>;
      uint32 epoch;
      ContentType type;
      opaque sender_data_nonce<0..255>;
      opaque encrypted_sender_data<0..255>;
      opaque ciphertext<0..2^32-1>;
} MLSCiphertext;
```

```
struct {
      uint32 sender;
      uint32 generation;
} SenderData;

struct {
      opaque group_id<0..255>;
      uint32 epoch;
      ContentType type;
} SenderDataAAD;

struct {
      opaque group_id<0..255>;
      uint32 epoch;
      ContentType type;
      uint32 sender;
      uint32 generation;
} CiphertextAAD;
```

Encryption keys

- Application messages: regular key from application key schedule
- Handshake messages: Derive a new key from the key schedule

Authentication

Common AEAD / signature scheme between Handshake and Application

1. AEAD authenticates sender's membership in the group
2. Signature authenticates specific sender identity
3. [Handshake only] Confirmation MAC proves agreement on group state

Signature-covers-MAC instead of MAC-covers-Signature (as in TLS)

TODO: Analysis to confirm this is OK

Note: SIGMA is agnostic; neither covers the other

Open questions

- Are we fine with revealing ContentType?
- Is it worth encrypting sender & generation?
- Are there any obvious problems with switching the order of signature and MAC (relative to TLS)?

Lazy updates

MLS IETF 104 Prague

Context

- Current problem: bottleneck when issuing an Update in all groups
- Awareness raised at IETF 103, initial discussion at the last interim
- Reminder: most groups in messengers are inactive
- Reminder 2: Time is a scarce resource when enrolling a new device
- Possible solution: LazyUpdate

How does it work?

- Similar to the first phase of a regular update

```
struct {
      DirectPath path;
} Update;


struct {
      HPKEPublicKey public_key;
} LazyUpdate;
```

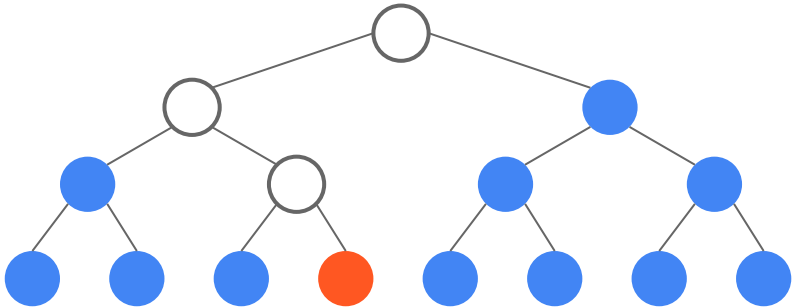
How does it work?

1. The sender creates a new leaf node (NLN), but doesn't "hash it up".

2. The NLN is sent to the group in a new HS message (LazyUpdate)

3. Everyone in the group replaces the old leaf node (OLN) of the sender with the new leaf node (NLN)

4. Everyone blanks the sender's direct path

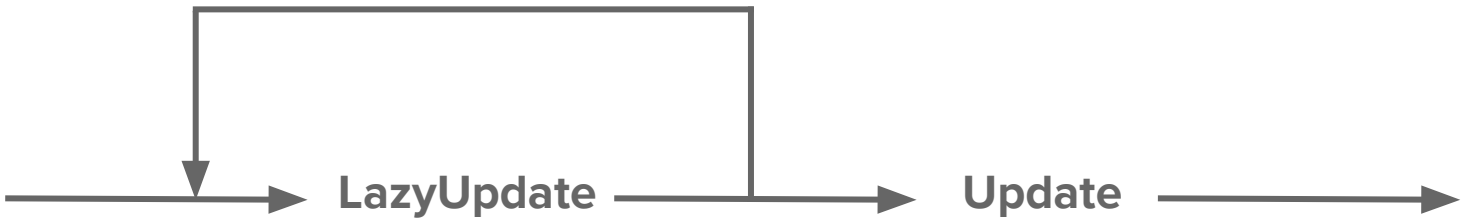


Net result

- No crypto is done, other than generating the new leaf node (fast)
- New leaf nodes can potentially be re-used among groups
- New leaf node can be used by everyone for HPKE

- No security yet: No FS or PCS at this point

Rules



- A LazyUpdate can be followed by a LazyUpdate
- The last LazyUpdate must be followed by a regular update
- The regular update can be issued by a different sender

Cost shifting

- The sender of the LazyUpdate postpones the cost of an Update: O(1) instead of O(log n) - O(n)
- Either the sender terminates the LazyUpdate (e.g. on a different device, or later on the same device)
- Or the LazyUpdate is terminated by another member
- Potential for saving: If a member frequently sends LazyUpdates in the same leaf (in particular in inactive groups)
- Cost model shifted from “eager” to “lazy”
- Operations might be more expensive, but fewer of them might be needed
- Bottlenecks are avoided

Security considerations

- LazyUpdates introduce a new “in between epochs” state
- The old epoch is no longer valid, but the new one is not there yet
- Only the terminating regular Update introduces a new epoch
- No other operation should occur (including sending application messages) before the new epoch is there
- This might complicate analysis

More context

- LazyUpdates are very similar to Adds
- Differences: Adds don’t require PCS, only FS

```
struct {
      uint32 index;
      UserInitKey init_key;
} Add;


struct {
      HPKEPublicKey public_key;
} LazyUpdate;
```

More context

- There could be LazyRemove handshake message as well

```
struct {
      uint32 removed;
      DirectPath path;
} Remove;
```

```
struct {
      uint32 removed;
} LazyRemove;
```

Open questions

- Are there better ways to address bottlenecks?
- Are we fine with the (yet not fully understood) security implications?