# Low-Level Secure Scuttlebutt Packet Spec

*draft 2022-03-23*

## A) General Packet Layout (128 Bytes)

- 8B are reserved for link- or connection-specific purposes (e.g, cloaking, encryption)
- 120B are either *blobs* or *log entries*
- log entries have:
  - a demultiplexing field (DMX), resembling port or application-type selectors
  - a 1B algorithm and type field
  - a 48B payload
  - a 64B signature

```
<------------------- 128B ------------------->

  8B               120B
+-----+-------------------..------------------+
| RND | cloaked packet content                | frame
+-----+-------------------..------------------+

      where uncloaked PACKET content is one of:

      +---------------------------------------+
      | blob                                  |
      +---------------------------------------+

         7B           113B
      +-----+--------..----------------------+
      | DMX | demux content                  |
      +-----+--------..----------------------+

             where demux content is one of:

              1B     48B            64B
            +---+----..---+--------..--------+
            | 0 | payload | crypto-signature | 48B payload
            +---+----..---+--------..--------+

            +---+----..---+--------..--------+
            | 1 | L|C|PTR | crypto-signature | start of blob chain
            +---+----..---+--------..--------+
                  L= overall content length
                  C= first part of content
                  PTR = 20B hash of first blob in the chain

             ...

            +---+----..----------------------+
            | x | frontier/repl mgmt cmd/etc | and other packet types ...
            +---+----..----------------------+
```

## B) Replicating Append-Only Logs “With Context”

In classic Secure Scuttlebutt, a log entry is self-describing in the sense that it contains all relevant bit fields like `feedID` (ed25519 public key), the entry’s sequence number and the hash value of the previous log entry, in order to verify the entry’s signature. Together with some “trust anchor state” (feedID and hash of first log entry), a consumer can assert the integrity, authenticity and trustworthiness of a log entry . . . if the previous log entry was already trusted.

Based on the observation that trustworthiness of a new entry anyway depends on the trustworthiness of past entries as well as other external state, Low-Level SSB only includes those bits in the wire format that *cannot* be procured from such context. The feedID, for example, is implicit through the signature: a consumer can validate the signature by trying out all trusted public keys (because a key that is not part of the trust anchor cannot lead to trustworthiness by definition). Similarily, the sequence number is very much predictable: either it is immediately following the hightest trusted number we know about, or the potential entry has to be dropped anyway.

*Not* sending a field does not mean that it has no relevance. When computing the signature, for example, these fields must be part of the “virtual log entry” that is signed together with the bits of the packet, but these fields are not manifested on the wire.

```
                     context state (exists at producer  |    a) trust anchors
                             as well as consumer side)  |    b) per feed 'head':
                                                        |      - feedId
                                                        |      - latestSeqNo
                                                        |      - latestMsgID
                                                         `---------|-----------
                                                                   |
virtual part of log entry              +-----log entry-----+       |
(used to compute the signature         .                   . <-----'
 and demux field)                      .    feedID         .
                                       .    seqNo          .
                                   <---.--- prevMsgID      .
. . . . . . . . . . . . . . . . . . . .+ . . . . . . . . . +
                                       |    demux          |
manifested part of log entry           |    algo + type    |
(bits 'on the wire' as well            |    payload        |
 as replicated on disk)                |    signature      |
                                       +-------------------+
```

Figure 1: virtual vs manifested parts of a log entry

The following section explains, starting from the packet format (wire bits), how the various fields are computed.

## Log Entry – Computing the (virtual) 64 Bytes *Name of a Log Entry*

*virtual* means that a value is only used internally i.e., it is not transmitted as actual bytes on the wire.

```
LOG_ENTRY_NAME = PFX + FID + SEQ + PREV

  where PFX   = 8B   'llssb-v0', for versioning packet formats
        FID   = 32B  ed25519 public key (feed ID)
        SEQ   = 4B   sequence number in big endian format
        PREV  = 20B  message ID (MID) of preceding log entry

        The PREV field is all-zeros for the log's genesis block (seq=0)
```

## Log Entry – Computing the 7 Bytes *Demultiplexing Field*

```
DMX = sha256(LOG_ENTRY_NAME)[:7]
```

## Log Entry – Computing the (virtual) 120 Bytes *Expanded Log Entry*

```
EXPANDED_LOG_ENTRY = LOG_ENTRY_NAME + DMX + T + PAYL
                   = PFX + FID + SEQ + PREV + DMX + T + PAYL

  where T     = 1B   signature algorithm and packet type
        PAYL  = 48B  payload
```

## Log Entry – Computing the 64 Bytes *Signature*

```
SIG = ed25519_signature(secretkey, EXPANDED_LOG_ENTRY)
```

## Log Entry – Computing the (virtual) 184 Bytes *Full Log Entry*

```
FULL_LOG_ENTRY = EXPANDED_LOG_ENTRY + SIG
               = PFX + FID + SEQ + PREV + DMX + T + PAYL + SIG

  where SIG   = 64B  signature
```

## Log Entry – Computing the (virtual) 20 Bytes *Message ID*

```
MID = sha256(FULL_LOG_ENTRY)[:20]
```

The MessageID of a log entry is referenced in the subsequent log entry as PREV field (that is never transmitted).

## Log Entry – Computing its 120 Bytes *Wire Format*

```
PACKET = DMX + T + PAYL + SIG
```

## Blob – Computing its 20 Bytes *Hash Pointer*

```
PTR = sha256(BLOB)[:20]

  where BLOB   = 120B
```

## C) Type 1 – Content Stored in a Side Hashchain

If more than the fixed-size 48B payload should be added to the log (type-0), a sidechain can be used (type-1). The payload field serves to encode (a) the total length of content, (b) some bytes of the content, plus (c) a hash pointer to the first data blob. Inside each blob, 100B are for content and 20B are for a continuation pointer, linking to the next blob.

```
<--------------120B packet content---------->


        <------48B payload----->
+----+-+------------------------+- . . . ---+
| OHT|T| len | content1 | hptr1 | signature |   log entry
+----+-+-----+---------+-------+- . . . ---+
                            |
   ------------------------' start of side chain (hash of blob 1)
  /
 v     100B                          20B
+----------. . . . . . . . . .------+-------+
| content2                          | hptr2 |   blob 1, identfied by hptr1
+----------. . . . . . . . . .------+-------+
                                       |
                      - - -------------' continuation of side chain
   ------------- - - -
  /
 v     100B                          20B
+----------. . . . . . . . . .------+-------+
| contentN         |  padding       | 0---0 |  last blob N-1
+----------. . . . . . . . . .------+-------+
```

Figure 2: Side chain to have log entries
with content of arbitrary length

When forming the chain, the last blob has to be created first, from which the last pointer value can be derived which goes into the second-last blob etc. The total content length (field `len`) is encoded in Bitcoin's varInt format and its length must be considered when filling the last packet.

Note that Type-0 log entries have a fixed-length 48B payload while Type-1 log entries can have any length, even from 0 to 48 bytes. When the length is less than 28, the `hptr1` field consists of zeros as no blobs are necessary.

## D) Filtering Received Packets via "Expectation Tables"

A node has precise knowledge about what packets are acceptable, given its current state. The DMX field was introduced for letting packets declare their profile and letting nodes filter on these announcement (before performing the computationally expensive verification of the signature). A second, independent filter is based on a requested packet's hash value. A node therefore maintains two "expectation tables", DEMUX_TBL and CHAIN_TBL.

The DEMUX_TBL table is populated by `(DMX,feedID)` tuples for each feed that a node subscribed for: the node can exactly predict which `feedID`, `SeqNo` and `prevMsgID` a packet must have in order to be a valid extension of the local log replica. As soon as a valid entry was received, the old DMX value can be removed from the DEMUX_TBL and is replaced by the next one, based on the updated feed. Note that a publisher can stream log entries back-to-back while the consumer simply rotates the corresponding DMX entry.

Similarily, if a log entry contains a sidechain and the subscription asked for replication WITH its sidechains, the node can add an expected hash pointer value to the CHAIN_TBL table (which is the first hash pointer of this chain, found in the payload of the received log entry). When the first blob is received, the node can replace this expectation with the next hash pointer found in the received blob, etc

Overall, the pseudo code for packet reception looks like this:

```
on_receive(frame f):

  pkt = uncloak(f)
  hptr = hash(pkt)
  if hptr in CHAIN_TBL:
    remove hptr from CHAIN_TBL
    deliver_blob(pkt)                 # may add new values to CHAIN_TBL and DEMUX_TBL
    return

  if pkt.DMX in DEMUX_TBL:
    for feedID associated with pkt.DMX:
      if verify(full_log_entry(feedID,pkt), pkt.signature)
        remove pkt.DMX from DEMUX_TBL
        deliver_log_entry(feedID, pkt) # may add new values to DEMUX_TBL and CHAIN_TBL
        return

  otherwise drop this frame
```

---