Chatwoot webhook security, loop prevention, and replay

This reference defines the complete inbound boundary implemented by the reviewed Chatwoot host. It covers authentication, freshness, replay state, loop suppression, event identity, local persistence, and central publication.

SECTION	Connectors
TYPE	Connector
PROTOCOL	AIP 1.0
SOURCE	docs/connectors/chatwoot/reference/webhook-security-loop-prevention-and-replay.md

A successful webhook response proves local acceptance or deliberate suppression. It does not always prove that central AIP acknowledged or consumed the event.

Ingress boundary

Webhook ingress is opt-in. The standalone host exposes:

```
TEXT
POST /webhooks/chatwoot
```

only when `AIP_CHATWOOT_WEBHOOK_SECRET_FILE` is configured. In that mode, `AIP_CONNECTOR_HOST_EVENT_ENDPOINT` is also required. An incomplete configuration fails startup.

If the webhook secret is omitted, the host is outbound-only and does not create this route. Provider webhook records do not change that host behavior.

The route accepts an original body of at most 1 MiB. The bytes must be valid UTF-8 because the frozen delivery type stores the exact body as a string before verification and JSON parsing.

Acceptance pipeline

One delivery follows this order:

1. read the three required headers;
2. parse the timestamp as a signed integer;
3. retain the exact UTF-8 request body;
4. verify timestamp freshness and HMAC over the raw body;
5. atomically observe the delivery ID in instance-scoped replay state;
6. parse the raw body as the supported Chatwoot subset;
7. suppress connector-originated loop messages;
8. map a non-looping delivery to one Channel Message;
9. derive one deterministic AIP Event ID;
10. append or recover that event in the local event log;

11. durably enqueue the event for central AIP;
12. return the local and central acceptance outcome.

Signature verification happens before JSON parsing. Reformatting, parsing, or re-serializing the body before HMAC verification changes the signed bytes and is invalid.

Required headers

HTTP header	Accepted value	Used for
`X-Chatwoot-Delivery`	Non-empty string	Replay identity and deterministic Event ID
`X-Chatwoot-Timestamp`	Base-10 signed integer	Freshness and signed input
`X-Chatwoot-Signature`	`sha256=<hex>`	HMAC-SHA256 comparison

HTTP header names are case-insensitive. Missing, non-text, or empty headers produce `400 webhook.missing_header`. A non-integer timestamp produces `400 webhook.invalid_timestamp`.

The connector does not derive the delivery identity from the JSON body. Keep the provider-supplied delivery header unchanged across a legitimate retry.

Signature construction

Let:

- `secret` be the exact bytes from the owner-only webhook-secret file;
- `timestamp` be the decimal header text after integer parsing;
- `raw_body` be the exact request body bytes.

The signed byte sequence is:

```
TEXT
ascii(decimal_timestamp) || "." || raw_body
```

The signature value is:

```
TEXT
"sha256=" || lowercase_or_uppercase_hex(HMAC-SHA256(secret, signed_bytes))
```

Hex decoding accepts valid hex digits; the prefix itself is exactly `sha256=`. The computed and supplied MACs are compared in constant time after their lengths match.

The host accepts a timestamp at most 300 seconds in the past or future relative to its UTC clock. A larger absolute difference fails authentication.

Clock synchronization is therefore part of availability. Do not widen the window in a proxy or replace the timestamp before forwarding, because either change breaks the source-owned verification contract.

Raw-body rules

Authentication binds exact bytes, not a JSON value. These bodies can represent the same JSON while producing different signatures:

- different whitespace;
- different member order;
- different string escapes;
- a final newline;

- a different Unicode byte representation.

A reverse proxy can terminate TLS, but it must preserve the body bytes and the three headers. Content transformation, decompression policy, and request-size limits must be tested against this byte-level boundary.

The connector parses only after successful HMAC verification. The supported top-level subset requires `event`, `account`, and `conversation`; `message` and `contact` are optional.

Replay state is a repair fence

The production host builds a `ProfileStateChatwootReplayStore` scoped as:

```
TEXT
instance:{connector_instance_id}
```

The state namespace is `aip.connector.chatwoot.webhook_replay.v1`. Compare-and-set gives a shared instance one atomic delivery-ID set across replicas using the same durable profile state.

Property	Reviewed value
Freshness and retention cutoff	300 seconds
Maximum retained delivery identities	10,000
Compare-and-set attempts	32
Production scope	Logical connector instance

Before inserting a new ID, the store removes entries whose recorded delivery timestamp is older than the current cutoff. At capacity, it removes the oldest recorded timestamps until the new ID fits.

A duplicate ID returns `false` from the replay store. The ingestion path deliberately does not reject or drop on that Boolean. It reconstructs the same deterministic Event identity so a provider retry can repair a crash between replay observation and event persistence.

This distinction matters:

- replay state records that an authenticated delivery was observed;
- deterministic Event identity prevents a second durable AIP event;
- the event store returns the first stored event when that ID already exists;
- the durable publication outbox can retry central delivery independently.

The error variant `connector.chatwoot.webhook_replay` exists in the connector error mapping, but the reviewed delivery path does not construct it for an ordinary duplicate.

Deterministic Event identity

A non-looping delivery becomes an Event with:

Field	Mapping
`kind`	`chatwoot.channel_message`
`id`	`evt_chatwoot_` plus lowercase SHA-256 hex of the delivery ID
local `actor`	Principal derived from Chatwoot account and contact IDs
`data.channel_message`	Normalized Channel Message
delivery metadata	Original delivery ID
raw payload summary	Chatwoot event name

The Channel Message carries the provider conversation as an external reference, the message object, text content, and a Chatwoot channel descriptor. Missing numeric account, conversation, or contact IDs map to the string `unknown` rather than creating provider facts.

Before central publication, the signed connector-host publisher moves the provider-originated actor into upstream data. The authenticated central ingress owns the protocol Event actor and binds it to the host signing identity.

Loop prevention

Messages created by the three connector composites contain:

```
JSON
{
  "content_attributes": {
    "aip": {
      "connector_id": "chatwoot",
      "loop_prevention": true,
      "action_id": "act_..."
    }
  }
}
```

Inbound mapping suppresses a message when either condition is true:

- `content_attributes.aip.loop_prevention` is Boolean `true`;
- `content_attributes.aip.connector_id` equals `chatwoot`.

Suppression happens after authentication, replay observation, and JSON parsing, but before Channel Message and Event construction. The host returns:

```
JSON
{
  "accepted_events": 0,
  "central_delivery": null
}
```

No AIP event or central outbox record is created for that delivery.

Loop prevention is marker-based. It is not a general bot-sender rule and does not prove that unmarked automations are safe to reprocess.

Local persistence and central publication

For a non-looping delivery, the host first calls the local Event Log's atomic `append_with_outcome`. The durable PostgreSQL implementation claims unique Event IDs and returns the existing stored event for an idempotent replay.

The host then passes the exact accepted Event to the connector event publisher on channel `chatwoot-webhooks`. The publisher:

- validates the admitted channel route;
- normalizes the Event for signed publication;
- validates bounded Event size;
- stores a content-addressed batch in durable profile state;
- attempts immediate delivery to the fixed central endpoint;
- retains transient failures for the background publisher.

The outbox accepts one through 100 Events per batch and retains at most 10,000 pending batches. The Chatwoot route submits one Event at a time.

Successful responses

For a non-looping delivery, HTTP `200` returns one of:

```
JSON
{
  "accepted_events": 1,
  "central_delivery": "centrally_acknowledged"
}

JSON
{
  "accepted_events": 1,
  "central_delivery": "durably_queued"
}
```

`central_delivery`	Established fact
`centrally_acknowledged`	Central AIP acknowledged the batch before the webhook response
`durably_queued`	Local durable outbox owns later central delivery
`null`	No Event was produced, normally because loop prevention matched

`durably_queued` is a successful local acceptance. It does not prove central consumption, subscriber processing, or business completion.

Error responses

HTTP status and code	Boundary that failed	Retry decision
`400 webhook.missing_header`	Required header absent, empty, or not text	Correct sender or proxy; do not invent a header
`400 webhook.invalid_timestamp`	Timestamp was not an integer	Correct header encoding
`400 webhook.invalid_utf8`	Body was not valid UTF-8	Preserve evidence and correct sender encoding
`401 webhook.authentication_failed`	HMAC verification, JSON decoding, or connector-level webhook mapping returned `InvalidWebhook`	Distinguish clock, bytes, signature, secret, and payload shape in bounded host diagnostics; do not bypass verification
`400 webhook.invalid_delivery`	Another connector-level delivery rejection occurred	Preserve delivery ID and inspect bounded host diagnostics
`500 webhook.invalid_mapping`	Channel Message or Event mapping failed	Preserve authenticated delivery evidence and escalate
`503 webhook.storage_unavailable`	Replay state, Event Log, or durable outbox failed	Repair durable state, then replay the same provider delivery

The public HTTP mapping deliberately groups several internal errors. Never put the secret, raw customer payload, or unrestricted provider message into an error response.

Crash-window recovery

Last established point	Provider retry with same delivery ID
HMAC failed	Fails again until clock, bytes, signature, or secret is corrected
HMAC passed; replay state failed	Can be retried after durable state recovers
Replay ID stored; process crashed before Event append	Reconstructs the deterministic Event ID and can append it
Event appended; process crashed before outbox enqueue	Returns the existing Event and can enqueue it
Outbox durable; central delivery failed	Background publisher owns retry; provider retry is not required
Central acknowledged; local outbox cleanup crashed	Central Event identity deduplicates a repeated signed batch
Loop message suppressed	Repeats the same zero-Event outcome while the signature remains fresh

Use the same provider delivery ID, timestamp, signature, and exact raw body when replaying within the authentication window. A newly signed delivery with a new ID is a new ingress identity, even when its JSON content matches.

Do not delete replay state or local Event data to force acceptance. Repair the failed durable component and let deterministic identities settle the original delivery.

Verification checklist

Verify all of the following for a webhook-enabled deployment:

- the secret is an owner-only file and is not present in environment values;
- the central event endpoint is fixed, allowlisted, and signed by the host;
- the public proxy preserves required headers and exact body bytes;
- host and provider clocks remain within the five-minute boundary;
- all replicas for one logical instance share durable profile state;
- Event storage enforces durable Event-ID uniqueness;
- `durably_queued` is monitored as pending delivery, not central success;
- looped connector messages return zero accepted Events;
- retries retain the original delivery identity and raw bytes;
- logs, metrics, and retained evidence redact customer payloads and secrets.

These checks establish conformance to the reviewed source path. They do not prove provider reachability, live delivery, or production qualification.

Related documentation

- Chatwoot connector configuration
- Inboxes, integrations, and webhooks
- AIP composite operations
- Connector fleet architecture
- Error reference