WeSeeDo Webhooks — authentication guide for implementation partners

How to receive WeSeeDo webhooks and verify that they really came from WeSeeDo.

1. What you receive

WeSeeDo sends an HTTP `POST` to the URL configured for your site / company / dealer / reseller.

```
POST /hooks/weseedo HTTP/1.1
Content-Type: application/json
X-Webhook-Timestamp: 1786612353123
X-Webhook-Nonce: 0f1c3a2e-4b5d-4e6f-8a9b-0c1d2e3f4a5b
X-Webhook-Hashed-Body: 0d92547d3bd5e2fe8883ba9f0de72f36beffe83db62068763addb7d79d7aff19
X-Webhook-Signature: 0f809d4cb54661317538d1aaf0d622746d722bc86e74e875883b405f1fdceade

{"id":"5f450d7b6bbe2f0540961982","created":"2026-08-
13T09:12:33.123Z","attempt":1,"type":"MEETING_ENDED","data":
{"id":"68b1c0f2a1b2c3d4e5f60718","name":"Café demo"}}
```

Field	Meaning
`id`	Unique id of this webhook. **Stays the same across retries** — use it as your idempotency key.
`created`	When the event was queued (ISO 8601, UTC).
`attempt`	Delivery attempt, `1` – `5`.
`type`	Event type, e.g. `MEETING_ENDED`.
`data`	The entity, in the same shape as the corresponding REST API response — except for `*_DELETED`, see below.

Event types:

`MEETING_CREATED`, `MEETING_STARTED`, `MEETING_RESTARTED`, `MEETING_ENDED`, `MEETING_CHANGED`, `MEETING_TAKEOVER`, `MEETING_WAIT`, `MEETING_DECLINED`, `MEETING_FORWARDED`, `MEETING_REMINDER`, `MEETING_DELETED`, `MEETING_SCREENSHOT_FAILED`, `MEETING_SCREENSHOT_UPLOADED`, `MEETING_SCREENSHOT_INDIRECT_UPLOAD_AUTHENTICATION_SUCCEEDED`, `MEETING_SCREENSHOT_INDIRECT_UPLOAD_AUTHENTICATION_FAILED`, `USER_CREATED|CHANGED|DELETED`, `SITE_CREATED|CHANGED|DELETED`, `COMPANY_CREATED|CHANGED|DELETED`, `DEALER_CREATED|CHANGED|DELETED`.

`*_DELETED` payloads are reduced and do **not** follow the REST shape:

- `MEETING_DELETED` → `{ "id", "company_id", "site_id", "participants": [{ "user", "email", "name" }] }` — note the flat `company_id` / `site_id`, where other events carry nested `company` / `site` objects.
- all other `*_DELETED` → `{ "id" }` only.

2. Authentication modes

Configured per entity (site, company, dealer, reseller):

Mode	What WeSeeDo sends	When to use
No authentication	Only the JSON body over HTTPS.	Never for production. Anyone who learns your URL can post fake events.
HMAC	The four `X-Webhook-*` headers described below.	Always.

That is the complete list — there is no Basic auth, bearer token, mTLS or per-field payload encryption on webhooks. Confidentiality comes from TLS; the HMAC gives you **authenticity + integrity** (it proves the payload came from WeSeeDo and was not modified).

The secret: minimum 16 characters. It is set on the same entity as the webhook URL and shared with you out of band. It is never transmitted with the webhook — only the HMAC computed with it. Use a long random string (32+ chars), store it as an environment variable/secret, never in your repo.

The URL: a complete `https://…` URL including the scheme, publicly reachable, with a valid certificate. Use an all-lowercase path.

3. The signature scheme

Two hex values are computed, both lowercase:

```
X-Webhook-Hashed-Body = SHA256( raw request body bytes, UTF-8 )
X-Webhook-Signature   = HMAC-SHA256( secret, timestamp + path + nonce + hashedBody )
```

The signed string is a plain concatenation — **no separators, no newlines** — of:

Part	Source	Example
`timestamp`	`X-Webhook-Timestamp`, verbatim. Unix time in **milliseconds**.	`1786612353123`
`path`	The path of the webhook URL configured in WeSeeDo. No scheme, no host, no query string.	`/hooks/weseedo`
`nonce`	`X-Webhook-Nonce`, verbatim. A UUID v4, unique per attempt.	`0f1c3a2e-…`
`hashedBody`	`X-Webhook-Hashed-Body`, verbatim (or your own recomputed value — verify they match first).	`0d92547d…`

So for the example above the signed string is:

```
1786612353123/hooks/weseedo0f1c3a2e-4b5d-4e6f-8a9b-
0c1d2e3f4a5b0d92547d3bd5e2fe8883ba9f0de72f36beffe83db62068763addb7d79d7aff19
```

Path edge cases: a URL with no path signs `/`; a trailing slash is kept; a percent-encoded path is signed in its encoded form (`/h%20k`); a query string is never part of the signature (`https://you.example.com/hooks/wsd?src=x` signs `/hooks/wsd`).

Two rules that cause 90% of the integration problems

1. **Hash the raw bytes.** Hash the body exactly as it arrived, before any JSON parse/re-serialize. Parsing and re-stringifying changes key order, spacing and unicode escaping, and the hash will never match.
2. **Use the configured path, not the received path.** If a reverse proxy, load balancer or framework rewrites or strips a prefix, `req.path` is no longer what WeSeeDo signed. Hardcode/configure the path exactly as it is configured in WeSeeDo, and coordinate any later change of the URL with your WeSeeDo contact.

4. Verification checklist

In this order:

1. All four headers present → else reject with `401`.
2. `|now − X-Webhook-Timestamp| ≤ 5 minutes` (remember: **milliseconds**) → else reject. Retries are re-signed with a fresh timestamp, so this is safe. Treat a malformed timestamp as a rejection, not as an exception.
3. `X-Webhook-Nonce` not seen before within that window (replay protection) → else reject.
4. `SHA256(raw body) == X-Webhook-Hashed-Body`.
5. `HMAC-SHA256(secret, timestamp+path+nonce+hashedBody) == X-Webhook-Signature`.
6. Compare in **constant time** (`timingSafeEqual` / `hash_equals` / `compare_digest`).
7. Deduplicate on the body's `id` before doing any work. Never deduplicate on the body hash — it is not stable across attempts, and it is not unique either.

5. Node.js / Express

Complete, runnable receiver. Only this example implements the nonce cache of step 3; the examples in §6 leave it out for brevity and you should port it across.

```
const express = require('express');
const crypto = require('crypto');

const SECRET = process.env.WESEEDO_WEBHOOK_SECRET;
const WEBHOOK_PATH = '/hooks/weseedo'; // must match the path configured in WeSeeDo
const TOLERANCE_MS = Number(process.env.TOLERANCE_MS || 5 * 60 * 1000);

if (!SECRET) throw new Error('WESEEDO_WEBHOOK_SECRET is not set');

const app = express();
```

```
const seenNonces = new Map(); // nonce -> expiry. Use Redis if you run more than one instance.

function safeEqual(a, b) {
    const bufA = Buffer.from(String(a), 'utf8');
    const bufB = Buffer.from(String(b), 'utf8');
    return bufA.length === bufB.length && crypto.timingSafeEqual(bufA, bufB);
}

function verifyWebhook(req) {
    // express.raw yields a Buffer. Anything else means another body parser ran first
    // and the exact bytes WeSeeDo signed are already gone.
    if (!Buffer.isBuffer(req.body)) return 'raw body not available';

    const signature = req.get('X-Webhook-Signature');
    const timestamp = req.get('X-Webhook-Timestamp');
    const nonce = req.get('X-Webhook-Nonce');
    const hashedBody = req.get('X-Webhook-Hashed-Body');

    if (!signature || !timestamp || !nonce || !hashedBody) return 'missing headers';

    const signedAt = Number(timestamp);
    if (!Number.isFinite(signedAt)) return 'malformed timestamp';
    if (Math.abs(Date.now() - signedAt) > TOLERANCE_MS) return 'stale timestamp';

    const now = Date.now();
    for (const [key, expiry] of seenNonces) if (expiry < now) seenNonces.delete(key);
    if (seenNonces.has(nonce)) return 'replayed nonce';

    const expectedHash = crypto.createHash('sha256').update(req.body).digest('hex');
    if (!safeEqual(expectedHash, hashedBody)) return 'body hash mismatch';

    const expectedSignature = crypto.createHmac('sha256', SECRET)
        .update(`${timestamp}${WEBHOOK_PATH}${nonce}${hashedBody}`)
        .digest('hex');
    if (!safeEqual(expectedSignature, signature)) return 'signature mismatch';

    // Anchor expiry to the signed timestamp: the cache then covers the whole window.
    seenNonces.set(nonce, signedAt + TOLERANCE_MS);

    return null;
}

// Mount this route before any global express.json(); the raw bytes are what is signed.
// Keep the size limit generous: some events carry sizeable payloads.
app.post(WEBHOOK_PATH, express.raw({ type: '*/*', limit: '5mb' }), async (req, res) => {
    const error = verifyWebhook(req);
```

```
    if (error) {
        console.warn('[weseedo] rejected webhook:', error);
        return res.sendStatus(401);
    }

    const event = JSON.parse(req.body.toString('utf8'));

    try {
        // Persist before acknowledging: a 200 means "delivered" and there is no retry after it.
        // Ignore the event if you have already stored this id.
        await enqueue(event);
    } catch (err) {
        console.error('[weseedo] could not store webhook:', err);
        return res.sendStatus(500); // WeSeeDo will retry
    }

    res.sendStatus(200);
});

app.listen(3000);
```

Process the stored events out of band, keyed on `event.type` and deduplicated on `event.id`.

6. Same thing in other stacks

Verification only — add the nonce cache and the persist-before-`200` handling from §5.

PHP

```
<?php
$secret = getenv('WESEEDO_WEBHOOK_SECRET');
if ($secret === false || $secret === '') {   // never verify with an empty key
    http_response_code(500);
    exit;
}

$path = '/hooks/weseedo';                     // as configured in WeSeeDo
$raw  = file_get_contents('php://input');

$h = [];                                      // getallheaders() is not available on every SAPI
foreach ($_SERVER as $key => $value) {
    if (strpos($key, 'HTTP_') === 0) {
        $h[strtolower(str_replace('_', '-', substr($key, 5)))] = $value;
    }
}
```

```
$timestamp  = $h['x-webhook-timestamp']   ?? '';
$nonce      = $h['x-webhook-nonce']       ?? '';
$hashedBody = $h['x-webhook-hashed-body'] ?? '';
$signature  = $h['x-webhook-signature']   ?? '';

$fail = null;
if (!$timestamp || !$nonce || !$hashedBody || !$signature) {
    $fail = 'missing headers';
// compared as float: a 13-digit millisecond timestamp overflows int on 32-bit builds
} elseif (!is_numeric($timestamp)
    || abs(round(microtime(true) * 1000) - (float) $timestamp) > 300000) {
    $fail = 'stale or malformed timestamp';
} elseif (!hash_equals(hash('sha256', $raw), $hashedBody)) {
    $fail = 'body hash mismatch';
} elseif (!hash_equals(
    hash_hmac('sha256', $timestamp . $path . $nonce . $hashedBody, $secret), $signature)) {
    $fail = 'signature mismatch';
}

if ($fail !== null) {
    error_log('[weseedo] rejected webhook: ' . $fail);
    http_response_code(401);
    exit;
}

// store $raw, then:
http_response_code(200);
```

Python (Flask)

```
import hashlib, hmac, os, time
from flask import Flask, request

SECRET = os.environ["WESEEDO_WEBHOOK_SECRET"].encode()
WEBHOOK_PATH = "/hooks/weseedo"          # as configured in WeSeeDo
TOLERANCE_MS = int(os.environ.get("TOLERANCE_MS", 5 * 60 * 1000))

app = Flask(__name__)

@app.post(WEBHOOK_PATH)
def weseedo_webhook():
    raw = request.get_data()             # raw bytes, not request.json
    timestamp = request.headers.get("X-Webhook-Timestamp", "")
    nonce = request.headers.get("X-Webhook-Nonce", "")
    hashed_body = request.headers.get("X-Webhook-Hashed-Body", "")
    signature = request.headers.get("X-Webhook-Signature", "")
```

```
    if not (timestamp and nonce and hashed_body and signature):
        return "", 401

    try:
        signed_at = int(timestamp)
    except ValueError:
        return "", 401
    if abs(int(time.time() * 1000) - signed_at) > TOLERANCE_MS:
        return "", 401

    # compare_digest on bytes: header values are not guaranteed to be ASCII
    if not hmac.compare_digest(hashlib.sha256(raw).hexdigest().encode(), hashed_body.encode()):
        return "", 401

    expected = hmac.new(
        SECRET,
        f"{timestamp}{WEBHOOK_PATH}{nonce}{hashed_body}".encode(),
        hashlib.sha256,
    ).hexdigest()
    if not hmac.compare_digest(expected.encode(), signature.encode()):
        return "", 401

    # store raw / request.json, deduplicated on event["id"], then:
    return "", 200
```

C# (ASP.NET Core minimal API, .NET 6+)

```
using System.Security.Cryptography;
using System.Text;

const string WebhookPath = "/hooks/weseedo";   // as configured in WeSeeDo
const long ToleranceMs = 5 * 60 * 1000;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var secret = Encoding.UTF8.GetBytes(
    Environment.GetEnvironmentVariable("WESEEDO_WEBHOOK_SECRET")
        ?? throw new InvalidOperationException("WESEEDO_WEBHOOK_SECRET is not set"));

static bool HexEquals(string a, string b) =>
    CryptographicOperations.FixedTimeEquals(
        Encoding.UTF8.GetBytes(a), Encoding.UTF8.GetBytes(b));

app.MapPost(WebhookPath, async (HttpRequest req) =>
```

```
{
    using var ms = new MemoryStream();
    await req.Body.CopyToAsync(ms);
    var raw = ms.ToArray();   // the stream is now consumed: parse from raw, not from req

    var timestamp  = req.Headers["X-Webhook-Timestamp"].ToString();
    var nonce      = req.Headers["X-Webhook-Nonce"].ToString();
    var hashedBody = req.Headers["X-Webhook-Hashed-Body"].ToString();
    var signature  = req.Headers["X-Webhook-Signature"].ToString();

    if (timestamp.Length == 0 || nonce.Length == 0
        || hashedBody.Length == 0 || signature.Length == 0)
        return Results.Unauthorized();

    // compared without subtracting, which avoids an overflow on a crafted timestamp
    var now = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();
    if (!long.TryParse(timestamp, out var signedAt) ||
        signedAt < now - ToleranceMs || signedAt > now + ToleranceMs)
        return Results.Unauthorized();

    if (!HexEquals(Convert.ToHexString(SHA256.HashData(raw)).ToLowerInvariant(), hashedBody))
        return Results.Unauthorized();

    using var hmac = new HMACSHA256(secret);
    var expected = Convert.ToHexString(
        hmac.ComputeHash(Encoding.UTF8.GetBytes($"{timestamp}{WebhookPath}{nonce}{hashedBody}")))
        .ToLowerInvariant();

    if (!HexEquals(expected, signature))
        return Results.Unauthorized();

    // store raw, deduplicated on the payload id, then:
    return Results.Ok();
});

app.Run();
```

On .NET 5 and older, replace `Convert.ToHexString(...)` with `BitConverter.ToString(...).Replace("-", "").ToLowerInvariant()` and `SHA256.HashData(raw)` with `SHA256.Create().ComputeHash(raw)`.

7. Test vector

Use this to unit-test your verifier without waiting for a real event. These values are what WeSeeDo produces.

```
secret     : a-very-long-test-secret
path       : /hooks/weseedo
timestamp  : 1786612353123
nonce      : 0f1c3a2e-4b5d-4e6f-8a9b-0c1d2e3f4a5b

raw body   : {"id":"5f450d7b6bbe2f0540961982","created":"2026-08-
13T09:12:33.123Z","attempt":1,"type":"MEETING_ENDED","data":
{"id":"68b1c0f2a1b2c3d4e5f60718","name":"Café demo"}}

X-Webhook-Hashed-Body : 0d92547d3bd5e2fe8883ba9f0de72f36beffe83db62068763addb7d79d7aff19
X-Webhook-Signature   : 0f809d4cb54661317538d1aaf0d622746d722bc86e74e875883b405f1fdceade
```

The body contains a non-ASCII character on purpose: if your framework re-encodes the body, the hash will not match.

The timestamp is fixed, so a correct verifier will reject the vector as stale. Raise the tolerance while replaying it — the examples above read `TOLERANCE_MS` from the environment for exactly this reason:

```
TOLERANCE_MS=999999999999 WESEEDO_WEBHOOK_SECRET=a-very-long-test-secret node receiver.js

curl -i -X POST http://localhost:3000/hooks/weseedo \
  -H 'Content-Type: application/json' \
  -H 'X-Webhook-Timestamp: 1786612353123' \
  -H 'X-Webhook-Nonce: 0f1c3a2e-4b5d-4e6f-8a9b-0c1d2e3f4a5b' \
  -H 'X-Webhook-Hashed-Body: 0d92547d3bd5e2fe8883ba9f0de72f36beffe83db62068763addb7d79d7aff19' \
  -H 'X-Webhook-Signature: 0f809d4cb54661317538d1aaf0d622746d722bc86e74e875883b405f1fdceade' \
  --data-binary '{"id":"5f450d7b6bbe2f0540961982","created":"2026-08-
13T09:12:33.123Z","attempt":1,"type":"MEETING_ENDED","data":
{"id":"68b1c0f2a1b2c3d4e5f60718","name":"Café demo"}}'
```

Expected: `200` for the vector, `401` if you replay it a second time (nonce), `401` if you change a single byte of the body.

8. Responding and retries

- **Respond with HTTP `200`.** Only `200` counts as delivered — `201`, `202`, `204` and every error are treated as failures and trigger retries.
- **Answer promptly**, as soon as you have stored the event. Do your processing afterwards, out of band.
- **Do not redirect.** Configure the exact final URL. Redirects are followed by the sender and the delivery may then be recorded as successful without your endpoint receiving the payload. Watch for the usual accidental redirects: `http` → `https`, apex→ `www`, added or stripped trailing slash.
- Failed deliveries are retried on a schedule measured from `created`: immediately, then **+1 min, +10 min, +1 h, +24 h** — 5 attempts in total, after which the webhook is dropped. Retries can land slightly later than these times.

- Every attempt is re-signed: same `id`, incremented `attempt`, new timestamp, new nonce, new signature. The body normally changes between attempts because `attempt` is part of it, so the body hash is not a reliable delivery identifier.
- **Be idempotent**, keyed on `id`. A duplicate can also arrive when your `200` is lost in transit.

9. Operational notes

- **One endpoint per level.** Webhook URLs can be set on the site, company, dealer and reseller. If the same URL is configured on more than one level you receive one delivery per level, each signed with *that* entity's own secret — and a level with authentication switched off delivers the same event unsigned. Prefer configuring a single level.
- **Secret rotation.** WeSeeDo stores one secret per entity, so accept both the old and the new secret in your verifier, then have the secret changed, then drop the old one.
- **Multiple secrets.** If you receive webhooks for several customers/sites on one endpoint, key your secrets by the configured path or use a distinct URL per site — the payload level is not identified in the headers.
- **Firewalling.** Verifying the signature is the security boundary; IP allow-listing is not offered, since delivery originates from the WeSeeDo API nodes and those addresses can change.

10. Troubleshooting

Symptom	Cause
`body hash mismatch`	You hashed parsed-and-re-serialized JSON instead of the raw bytes, or a middleware (an earlier body parser, compression, charset conversion) altered or consumed the body.
Hash matches but `signature mismatch`	Wrong `path` (proxy rewrite, trailing slash, query string included), or the timestamp/nonce were normalised instead of used verbatim, or the wrong entity's secret.
Everything gets retried although you handle it	You return something other than `200`.
Large events never arrive, small ones do	Your framework's body-size limit rejects the request before your handler runs.
Events look delivered but you never received them	Your URL redirects; see §8.
Nothing arrives	The URL is not publicly reachable over HTTPS, the scheme is missing from the configured URL, or the certificate is not trusted.
Signature headers missing	Webhook authentication is off, or no secret is set, for the entity that produced this delivery.

If you need the delivery status of a specific event, ask your WeSeeDo contact.