ChatGPT

Fixing Meta Instagram Webhook Signature Verification on Render with Flask

Background and key findings

Meta documents webhook authenticity using `X-Hub-Signature-256` (format `sha256=<hex>`) and expects you to compute an HMAC-SHA256 over the exact payload bytes using your App Secret, then compare. [1]

The "transparent gzip decompression" hypothesis does not fit the most directly observable evidence available for real Instagram/Messenger webhook deliveries: captured requests commonly include `Accept-Encoding: deflate, gzip` but do **not** include `Content-Encoding: gzip`, and the body is plain JSON with a normal `Content-Length` on the order of a few hundred bytes. [2] That header pattern indicates the sender is stating what **response encodings** it can accept (via `Accept-Encoding`), not that the request payload it sent was compressed. [3]

Render's public edge commonly sits behind Cloudflare (Render staff have stated Cloudflare is "in front of ALL services" deployed to Render), which means there is at least one reverse proxy hop before your container/process. [4] However, request-body decompression is not a default behavior in many common reverse proxies (and is uncommon on the public web generally), and there is no strong evidence in public Render documentation or community material that Render transparently decompresses request bodies in a way that would strip `Content-Encoding` before your app sees the request. [5]

A more likely root cause, based on how Meta describes signing and on common failure modes in real deployments, is that the bytes you hash in production are not exactly the bytes Meta signed—most often due to inadvertent body parsing/rewriting, Unicode escape handling, or reading the body after another layer has already consumed/decoded it. [6]

What Meta signs and how `X-Hub-Signature-256` verification actually works

Meta's webhook verification model (across multiple Meta webhook products) is described as:

- Meta signs Event Notification payloads and places the signature in `X-Hub-Signature-256`, prefixed with `sha256=`. [1]
- Verification is done by generating your own SHA-256 HMAC using the payload and your App Secret, then comparing to the header value (after `sha256=`). [1]

A crucial nuance repeatedly referenced in Meta webhook discussions is that the calculation is performed over a **raw escaped-Unicode representation** of the payload with lowercase hex digits; examples of

required escaping commonly called out include Unicode characters (e.g., `äöå` → `\u00e4\u00f6\u00e5`) and additional escapes for certain characters like `<`, `%`, and `@`. [7]

Two practical consequences follow from this:

1. **You must compute the HMAC over the exact bytes of the HTTP request body as received**, not over a re-serialized JSON object. Even "equivalent" JSON (different whitespace, different escaping, different ordering, different normalization) yields a totally different HMAC. [8]
2. **If you ever decode and re-encode the body (or reconstruct it from parsed JSON),** you risk changing escape sequences and therefore changing the signed bytes (especially when messages contain emojis or other non-ASCII characters). [9]

Whether Meta sends webhook bodies gzip-compressed

Publicly visible request captures of real Instagram webhook deliveries show:

- `Accept-Encoding: deflate, gzip`
- **No** `Content-Encoding`
- `Content-Type: application/json`
- A small `Content-Length` (e.g., ~402 bytes)
- The body is readable JSON as transmitted. [2]

That combination is consistent with **uncompressed request bodies** and with Meta indicating it can accept compressed **responses** from your server. [3]

This aligns with a broader HTTP reality: request-body compression (e.g., a client sending `Content-Encoding: gzip` on POST bodies) is comparatively uncommon because the sender cannot reliably know whether the receiver supports request decompression, and because enabling request decompression can magnify certain resource-exhaustion risks. [10]

From the above, the most supportable conclusion is:

- Meta webhook bodies are typically **sent uncompressed**, and Meta's signature is computed for the **raw JSON payload bytes as sent** (including whatever escaping Meta's serializer uses), not for a compressed-bytes variant. [11]

Render's proxy layer and the likelihood of request-body mutation

Render's platform commonly places Cloudflare in front of services, which confirms that webhook requests traverse at least one intermediary proxy hop before reaching your Flask process. [4] Cloudflare documents that it passes request headers to the origin and may add additional headers, and (separately) it is widely known to apply compression/decompression behaviors for **responses** based on `Accept-Encoding` negotiation. [12]

What matters for your case is **request** body handling. The strongest web-available indicators point away from "transparent request decompression" as a default proxy behavior:

- In common reverse-proxy deployments, incoming gzipped request bodies are typically forwarded as-is unless explicit request-decompression support is configured; for example, nginx is commonly observed passing `Content-Encoding: gzip` requests through to upstreams without uncompressing the body. [13]
- When request decompression *does* occur in application stacks, it is generally an explicit feature/middleware (not an implicit universal behavior); for example, the ASP.NET Core request decompression middleware wraps the request body in a decompression stream and removes the `Content-Encoding` header only when configured to do so. [14]

Given those patterns, plus the direct evidence that Meta's webhook deliveries include `Accept-Encoding` but not `Content-Encoding`, the decompression hypothesis is low-probability for Meta → Render → Flask. [15]

What Render-specific documentation does support is that users can optionally interpose their own Cloudflare proxy in front of Render (custom domains), and Render even documents scenarios where Cloudflare proxying changes routing behavior during domain verification. [16] If your Instagram webhook endpoint is behind multiple proxy layers (for example, your own Cloudflare in front of Render's Cloudflare), then "something in the middle changed something" becomes more plausible—but there is still no concrete public evidence that such changes would rewrite JSON bodies in-flight. [17]

Concrete steps to get verification working in Flask on Render

Make the byte source unambiguous: one authoritative `raw_body` buffer

In Flask/Werkzeug, the safest approach is to read the body exactly once as bytes using `request.get_data()` **before** anything else touches the body (e.g., before `request.get_json()`, before any code that might trigger form parsing). [18]

A minimal verification pattern that matches Meta's described scheme is:

```
import hashlib
import hmac
from flask import Flask, request, abort

APP_SECRET = b"...your app secret bytes..."  # ensure no trailing newline

app = Flask(__name__)

@app.post("/webhook")
def webhook():
    sig = request.headers.get("X-Hub-Signature-256", "")
    body = request.get_data(cache=True)  # raw bytes, cached for the request
lifetime
```

```
    expected = "sha256=" + hmac.new(APP_SECRET, body,
hashlib.sha256).hexdigest()

    # Constant-time compare
    if not hmac.compare_digest(expected, sig):
        abort(403)

    return "", 200
```

This approach is consistent with how Meta describes validating the payload (HMAC-SHA256, hex digest, `sha256=` prefix). [19]

Verify you are not accidentally hashing "a different view" of the payload

Because an HMAC changes completely if even a single byte differs, you need a way to prove the byte stream you hash is stable end-to-end. A practical technique is to log a **non-secret** fingerprint of the received bytes:

- `len(body)`
- `hashlib.sha256(body).hexdigest()` (plain SHA-256 over payload bytes, not keyed)
- a short hex prefix of `body[:32]` and suffix `body[-32:]` (in hex), for debugging.

The reason to prefer this over re-serializing JSON is that parsing/reserializing can change whitespace and escaping, which Meta explicitly warns affects signature validation. [20]

Eliminate the most common Flask/Werkzeug "raw body got consumed" trap

A known pitfall in Flask/Werkzeug is that some access patterns can consume the underlying input stream such that later attempts to read "raw data" do not retrieve the original bytes. The recommended pattern is to call `request.get_data()` as the canonical raw read, and not rely on properties that may trigger parsing behavior first (especially for form content types). [21]

Even though Meta webhooks are JSON, this still matters if you have any shared middleware that tries to parse bodies, log structured payloads, or enforce schema checks before your verification step. [22]

Avoid adding your own "escapeUnicode" step unless you can prove the incoming body differs

Meta's guidance about "escaped Unicode" is best understood as "hash the payload exactly as delivered (including any escapes)". For most servers, that means you should hash the raw bytes you receive, not a transformed representation. [9]

If you add additional escaping (for example, escaping all non-ASCII characters yourself) you risk diverging from the actual on-the-wire payload unless you can show (via byte fingerprints) that the request body you receive is *not* already in the escaped form Meta signed. [9]

If a proxy really is changing the body: what solutions are valid, and which are not

The non-negotiable requirement for signed webhooks

Webhook signature schemes generally require that the signature be computed over an exact canonical string/byte sequence, and providers emphasize that the body must not be changed before verification.

- Stripe explicitly frames signature failures around “the request body must be the body string … without any changes.” [23]
- entity["company","GitHub","code hosting platform"] explicitly calls out that if signature verification fails, you should ensure that proxies/load balancers do not modify the payload or headers. [24]

Meta’s scheme, as described, is in the same class: if an intermediary rewrites the JSON bytes, you can’t “recover” the originally signed bytes from the modified payload. [25]

“Re-compress it and verify” is not a reliable workaround

Even if Meta *had* signed a compressed request body, re-compressing the decompressed JSON to reproduce the original compressed byte stream is generally not dependable:

- The gzip format includes header fields such as timestamps (`MTIME`) and can include other metadata, meaning two gzip streams containing the same underlying JSON can differ byte-for-byte. This is described in the gzip file format specification (RFC 1952) from the entity["organization","IETF","internet standards body"] and is widely discussed as a cause of differing checksums for gzipped files. [26]
- It is a common principle of lossless compression that decompressing and recompressing does not guarantee identical output bytes across implementations/settings/metadata choices. [27]

So, if the platform did decompress a gzipped request body before your app could read it, the signature would be effectively unverifiable in-app unless the platform also preserved the original compressed bytes somewhere accessible (which is not a standard feature). [28]

Practical options if body mutation is proven

If you can prove (via stable byte fingerprints) that the body bytes reaching your Flask app differ from what Meta signed, the viable solutions are constrained by the requirement above:

- Ensure the endpoint is reachable without intermediary transformations that could rewrite bodies (for example, avoiding extra CDN/proxy layers in front of the webhook endpoint). Render explicitly supports configurations where Cloudflare is added in front of a service; reducing proxy layers reduces opportunities for mutation. [29]
- Terminate the webhook request at a component you control that can both (a) preserve the raw bytes and (b) forward the request downstream after verification, without changing the signed bytes. The key requirement is that verification must happen before any transformation. [30]

Summary answers to the research questions

Meta webhook bodies: Based on real webhook request captures, Meta sends JSON webhook bodies that are not gzip-compressed (`Accept-Encoding` is present, `Content-Encoding` is not), and signatures are intended to validate the integrity of the raw payload bytes sent, compared against `X-Hub-Signature-256: sha256=<hex>`. [11]

Render behavior: Render commonly places Cloudflare in front of services, but publicly available information does not show Render transparently decompressing inbound request bodies; request-body decompression is typically an explicit feature and request compression itself is uncommon on the public web. [31]

Concrete path to restore verification: Treat the request body as an opaque byte string; read it exactly once (e.g., `request.get_data()`), compute `sha256=` + HMAC-SHA256 hex digest using the App Secret, and compare in constant time—ensuring no parsing/reserialization or additional escaping occurs before verification. [32]

---

[1] [19] [25] [32] https://communityforums.atmeta.com/discussions/dev-general/how-to-verify-a-webhook-request-sign/1171086
https://communityforums.atmeta.com/discussions/dev-general/how-to-verify-a-webhook-request-sign/1171086

[2] [11] [15] https://github.com/chatwoot/chatwoot/issues/11983
https://github.com/chatwoot/chatwoot/issues/11983

[3] [5] https://medium.com/%40abhinav.ittekot/why-http-request-compression-is-almost-never-supported-5de68067b245
https://medium.com/%40abhinav.ittekot/why-http-request-compression-is-almost-never-supported-5de68067b245

[4] [17] [31] https://community.render.com/t/static-site-on-render-cloudflare-firewall-rules/4448
https://community.render.com/t/static-site-on-render-cloudflare-firewall-rules/4448

[6] [8] [20] [22] https://stackoverflow.com/questions/60875202/validating-payload-from-facebook-webhook
https://stackoverflow.com/questions/60875202/validating-payload-from-facebook-webhook

[7] [9] https://stackoverflow.com/questions/67699684/x-hub-signature-on-facebooks-messenger-platform-webhook
https://stackoverflow.com/questions/67699684/x-hub-signature-on-facebooks-messenger-platform-webhook

[10] https://stackoverflow.com/questions/46596899/why-is-gzip-compression-of-a-request-body-during-a-post-method-uncommon
https://stackoverflow.com/questions/46596899/why-is-gzip-compression-of-a-request-body-during-a-post-method-uncommon

[12] https://developers.cloudflare.com/fundamentals/reference/http-headers/
https://developers.cloudflare.com/fundamentals/reference/http-headers/

[13] https://serverfault.com/questions/334215/how-do-i-configure-nginx-to-accept-gzip-requests
https://serverfault.com/questions/334215/how-do-i-configure-nginx-to-accept-gzip-requests

[14] Request decompression in ASP.NET Core
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/request-decompression?view=aspnetcore-10.0&utm_source=chatgpt.com

16 https://render.com/docs/configure-cloudflare-dns
https://render.com/docs/configure-cloudflare-dns

18 21 https://stackoverflow.com/questions/10999990/get-raw-post-body-in-python-flask-regardless-of-content-type-header
https://stackoverflow.com/questions/10999990/get-raw-post-body-in-python-flask-regardless-of-content-type-header

23 Resolve webhook signature verification errors
https://docs.stripe.com/webhooks/signature?utm_source=chatgpt.com

24 30 https://docs.github.com/en/webhooks/testing-and-troubleshooting-webhooks/troubleshooting-webhooks
https://docs.github.com/en/webhooks/testing-and-troubleshooting-webhooks/troubleshooting-webhooks

26 https://datatracker.ietf.org/doc/html/rfc1952
https://datatracker.ietf.org/doc/html/rfc1952

27 28 https://stackoverflow.com/questions/66955482/why-does-recompressing-a-file-using-gzip-produces-a-different-output
https://stackoverflow.com/questions/66955482/why-does-recompressing-a-file-using-gzip-produces-a-different-output

29 https://community.render.com/t/can-i-proxy-via-cloudflare-to-my-render-app/288
https://community.render.com/t/can-i-proxy-via-cloudflare-to-my-render-app/288