Verificación de Webhooks

Firmas HMAC y Prevención de Ataques de Replay

PuigFlows — Josué Puig

Guía · Nivel Intermedio · Proyectos Reales

Índice general

Versión en Español

Aprende a verificar que un webhook entrante viene de verdad de quien dice y no fue alterado, usando firmas HMAC. Cubre el esquema marca-de-tiempo+cuerpo, comparación en tiempo constante, prevención de replay con tolerancia e idempotencia, e implementaciones listas para producción en Python (FastAPI) y Node (Express).

Un webhook es una de las integraciones más útiles que puedes construir. En lugar de preguntar cada pocos segundos si algo cambió, dejas que el servicio externo te avise con un POST en el instante exacto en que ocurre el evento. El problema es que esa URL queda expuesta: cualquiera que la descubra puede enviarte un pago falso, un pedido inventado o un evento de cancelación que jamás sucedió. La verificación de firmas es lo que separa un webhook seguro de una puerta abierta de par en par.

Qué problema resuelve la firma

Cuando llega un webhook necesitas responder dos preguntas antes de confiar en él: ¿de verdad lo envió quien dice enviarlo, y llegó intacto? Una firma HMAC responde a las dos. El emisor y tú comparten un secreto que nunca viaja por la red. El emisor calcula un hash del cuerpo del mensaje con ese secreto y lo adjunta en una cabecera. Tú repites el mismo cálculo con el mismo secreto y, si tu resultado coincide con el recibido, sabes dos cosas a la vez: el mensaje proviene de alguien que conoce el secreto y nadie alteró ni un byte por el camino.

Cómo funciona HMAC en la práctica

Los proveedores serios (Stripe, GitHub, Shopify) rara vez firman solo el cuerpo: firman la combinación de una marca de tiempo y el cuerpo, y envían ambas piezas en la cabecera. Un formato habitual luce así:

```
X-Signature: t=1718200000,v1=5257a869e7ec...firma_en_hex
```

El campo `t` es el momento en que se generó el evento y `v1` es el HMAC-SHA256 de la cadena formada por la marca de tiempo, un punto y el cuerpo crudo. Incluir ese timestamp dentro de la firma es justo lo que más adelante te permitirá frenar los ataques de repetición.

Cada proveedor firma a su manera

El esquema de arriba es genérico; en producción te adaptarás al formato de cada emisor, y casi siempre cambian solo dos cosas: el nombre de la cabecera y la codificación de la firma. Stripe la envía en la cabecera `Stripe-Signature` con el formato `t=...,v1=...` en hexadecimal. GitHub usa `X-Hub-Signature-256` con el valor `sha256=` seguido del HMAC en hexadecimal, así que debes quitar ese prefijo antes de comparar. Shopify, en cambio, manda el HMAC en `X-Shopify-Hmac-SHA256` codificado en Base64, no en hexadecimal, así que ahí comparas bytes en Base64. La lógica de fondo —recalcular el HMAC sobre los bytes crudos y comparar en tiempo constante— no cambia; solo ajustas de dónde lees la firma y cómo la decodificas.

Implementación en Python (FastAPI)

La regla de oro es verificar antes de parsear. Necesitas los bytes exactos que llegaron, porque si dejas que el framework convierta el JSON en un diccionario y luego lo vuelva a serializar, el orden de las claves o un simple espacio cambiarán y la firma nunca coincidirá.

```python
import json, hmac, hashlib, time
from fastapi import FastAPI, Request, HTTPException

app = FastAPI()
SECRET = b"whsec_tu_secreto"  # cárgalo desde una variable de entorno
TOLERANCIA = 300  # 5 minutos en segundos

def verificar(cuerpo: bytes, cabecera: str) -> None:
    partes = dict(p.split("=", 1) for p in cabecera.split(","))
    timestamp = int(partes["t"])
    recibida = partes["v1"]

    # 1. Anti-replay: rechaza eventos demasiado viejos
    if abs(time.time() - timestamp) > TOLERANCIA:
        raise HTTPException(400, "Marca de tiempo fuera de tolerancia")

    # 2. Recalcula el HMAC sobre timestamp + cuerpo crudo
    firmado = f"{timestamp}.".encode() + cuerpo
    esperada = hmac.new(SECRET, firmado, hashlib.sha256).hexdigest()

    # 3. Compara en tiempo constante
    if not hmac.compare_digest(esperada, recibida):
        raise HTTPException(400, "Firma invalida")

@app.post("/webhooks/proveedor")
async def recibir(request: Request):
    cuerpo = await request.body()  # bytes crudos, sin parsear
    firma = request.headers.get("X-Signature", "")
    verificar(cuerpo, firma)
    evento = json.loads(cuerpo)
    # procesa de forma idempotente con evento["id"]
    return {"recibido": True}
```

Cada paso cuenta: primero rechazamos los eventos viejos, luego recalculamos la firma sobre los mismos bytes que firmó el emisor y comparamos en tiempo constante. Solo cuando todo cuadra parseamos el JSON y procesamos el evento.

Implementación en Node (Express)

En Express el detalle crítico es no aplicar `express.json()` en esta ruta. Ese middleware consume el cuerpo y te deja únicamente el objeto ya parseado, así que pierdes los bytes originales. Usa `express.raw()` para conservar el Buffer intacto:

```javascript
const crypto = require("crypto");
const express = require("express");
const app = express();

const SECRET = process.env.WEBHOOK_SECRET;
const TOLERANCIA = 300; // segundos
```

```
// Importante: cuerpo crudo, NO express.json()
app.post(
  "/webhooks/proveedor",
  express.raw({ type: "application/json" }),
  (req, res) => {
    const cabecera = req.get("X-Signature") || "";
    const partes = Object.fromEntries(
      cabecera.split(",").map((p) => p.split("="))
    );
    const timestamp = Number(partes.t);

    // 1. Anti-replay
    if (Math.abs(Date.now() / 1000 - timestamp) > TOLERANCIA) {
      return res.status(400).send("Marca de tiempo fuera de tolerancia");
    }

    // 2. Recalcula el HMAC sobre timestamp + cuerpo crudo
    const firmado = Buffer.concat([Buffer.from(`${timestamp}.`), req.body]);
    const esperada = crypto.createHmac("sha256", SECRET)
      .update(firmado).digest("hex");

    // 3. Compara en tiempo constante (verifica la longitud primero)
    const a = Buffer.from(esperada);
    const b = Buffer.from(partes.v1 || "");
    if (a.length !== b.length || !crypto.timingSafeEqual(a, b)) {
      return res.status(400).send("Firma invalida");
    }

    const evento = JSON.parse(req.body);
    res.json({ recibido: true });
  }
);
```

Los tres errores que rompen la seguridad

1. Parsear el cuerpo antes de verificar

Es el fallo más habitual. Como el framework ya te entrega el cuerpo convertido en objeto, resulta tentador firmar sobre esa versión re-serializada. Pero volver a serializar no garantiza el mismo orden de claves, ni los mismos espacios, ni el mismo escape de Unicode que envió el emisor. Un solo byte distinto y el HMAC cambia por completo. Trabaja siempre sobre los bytes crudos.

2. Comparar con el operador normal en vez de en tiempo constante

Comparar `esperada == recibida` parece inofensivo, pero filtra información. Una comparación de cadenas se detiene en el primer carácter que difiere, así que tarda un poquito más cuando los primeros caracteres ya coinciden. Midiendo miles de intentos, un atacante puede reconstruir la firma carácter a carácter. Por eso `hmac.compare_digest` en Python y `crypto.timingSafeEqual` en Node tardan siempre lo mismo, sin importar dónde esté la diferencia.

3. Ignorar la marca de tiempo

Si solo validas la firma, quien intercepte un webhook legítimo puede reenviarlo mil veces: cada copia lleva una firma perfectamente válida. Rechazar los eventos cuya marca de tiempo se aleje más de cinco minutos del reloj actual cierra esa ventana. Combínalo con idempotencia —guardar el identificador de cada evento ya procesado— para que ni siquiera un reenvío dentro de la ventana provoque un cobro doble.

Buenas prácticas para producción

- Carga el secreto desde una variable de entorno; nunca lo escribas en el código ni lo dejes caer en los logs.
- Responde con un 2xx lo antes posible y procesa el evento en una cola o tarea en segundo plano: los proveedores reintentan si tardas demasiado, y eso multiplica los duplicados.
- Devuelve 400 ante una firma inválida y 200 ante un duplicado ya procesado, para que el emisor deje de reintentar algo que ya recibiste.
- Soporta rotación de secretos aceptando dos claves a la vez durante el periodo de cambio; así rotas sin tiempo de inactividad.
- Sirve el endpoint solo por HTTPS: la firma protege la integridad, pero TLS protege la confidencialidad.

Conclusión

Verificar firmas es una de las defensas con mejor relación esfuerzo-beneficio que existen: unas veinte líneas de código y tu endpoint deja de aceptar eventos de cualquier desconocido. Junto con la idempotencia y el rate limiting, la verificación de webhooks completa el trío básico para exponer una API a terceros sin perder el sueño.

English Version

Learn how to verify that an incoming webhook truly comes from who it claims to and was not tampered with, using HMAC signatures. Covers the timestamp+body scheme, constant-time comparison, replay prevention with tolerance and idempotency, and production-ready implementations in Python (FastAPI) and Node (Express).

A webhook is one of the most useful integrations you can build. Instead of polling every few seconds to ask whether something changed, you let the external service notify you with a POST the very instant the event happens. The catch is that the URL is now exposed: anyone who discovers it can send you a fake payment, a made-up order, or a cancellation event that never occurred. Signature verification is what separates a secure webhook from a wide-open door.

What the signature solves

When a webhook arrives, you need to answer two questions before trusting it: did the sender really send it, and did it arrive intact? An HMAC signature answers both. The sender and you share a secret that never travels over the network. The sender computes a hash of the message body using that secret and attaches it in a header. You repeat the same computation with the same secret, and if your result matches the one you received, you know two things at once: the message comes from someone who knows the secret, and not a single byte was altered along the way.

How HMAC works in practice

Serious providers (Stripe, GitHub, Shopify) rarely sign the body alone: they sign the combination of a timestamp and the body, and send both pieces in the header. A common format looks like this:

`X-Signature: t=1718200000,v1=5257a869e7ec...signature_in_hex`

The `t` field is when the event was generated, and `v1` is the HMAC-SHA256 of the string made of the timestamp, a dot, and the raw body. Putting the timestamp inside the signature is exactly what lets you shut down replay attacks later on.

Every provider signs its own way

The scheme above is generic; in production you adapt to each sender's format, and almost always only two things change: the header name and the signature's encoding. Stripe sends it in the `Stripe-Signature` header with the `t=...,v1=...` format in hexadecimal. GitHub uses `X-Hub-Signature-256` with the value `sha256=` followed by the HMAC in hex, so you must strip that prefix before comparing. Shopify, by contrast, sends the HMAC in `X-Shopify-Hmac-SHA256` encoded in Base64, not hex, so there you compare Base64 bytes. The underlying logic —recompute the HMAC over the raw bytes and compare in constant time— never changes; you only adjust where you read the signature and how you decode it.

Implementation in Python (FastAPI)

The golden rule is verify before you parse. You need the exact bytes that arrived, because if you let the framework turn the JSON into a dictionary and then re-serialize it, the key order or a single space will change and the signature will never match.

```
import json, hmac, hashlib, time
from fastapi import FastAPI, Request, HTTPException
```

```python
app = FastAPI()
SECRET = b"whsec_your_secret"  # load it from an environment variable
TOLERANCE = 300  # 5 minutes in seconds

def verify(body: bytes, header: str) -> None:
    parts = dict(p.split("=", 1) for p in header.split(","))
    timestamp = int(parts["t"])
    received = parts["v1"]

    # 1. Anti-replay: reject events that are too old
    if abs(time.time() - timestamp) > TOLERANCE:
        raise HTTPException(400, "Timestamp out of tolerance")

    # 2. Recompute the HMAC over timestamp + raw body
    signed = f"{timestamp}.".encode() + body
    expected = hmac.new(SECRET, signed, hashlib.sha256).hexdigest()

    # 3. Compare in constant time
    if not hmac.compare_digest(expected, received):
        raise HTTPException(400, "Invalid signature")

@app.post("/webhooks/provider")
async def receive(request: Request):
    body = await request.body()  # raw bytes, unparsed
    signature = request.headers.get("X-Signature", "")
    verify(body, signature)
    event = json.loads(body)
    # process idempotently using event["id"]
    return {"received": True}
```

Every step matters: we reject old events first, recompute the signature over the same bytes the sender signed, and compare in constant time. Only when everything checks out do we parse the JSON and process the event.

Implementation in Node (Express)

In Express the critical detail is not to apply `express.json()` on this route. That middleware consumes the body and leaves you only the already-parsed object, so you lose the original bytes. Use `express.raw()` to keep the Buffer intact:

```javascript
const crypto = require("crypto");
const express = require("express");
const app = express();

const SECRET = process.env.WEBHOOK_SECRET;
const TOLERANCE = 300; // seconds

// Important: raw body, NOT express.json()
app.post(
```

```
  "/webhooks/provider",
  express.raw({ type: "application/json" }),
  (req, res) => {
    const header = req.get("X-Signature") || "";
    const parts = Object.fromEntries(
      header.split(",").map((p) => p.split("="))
    );
    const timestamp = Number(parts.t);

    // 1. Anti-replay
    if (Math.abs(Date.now() / 1000 - timestamp) > TOLERANCE) {
      return res.status(400).send("Timestamp out of tolerance");
    }

    // 2. Recompute the HMAC over timestamp + raw body
    const signed = Buffer.concat([Buffer.from(`${timestamp}.`), req.body]);
    const expected = crypto.createHmac("sha256", SECRET)
      .update(signed).digest("hex");

    // 3. Compare in constant time (check length first)
    const a = Buffer.from(expected);
    const b = Buffer.from(parts.v1 || "");
    if (a.length !== b.length || !crypto.timingSafeEqual(a, b)) {
      return res.status(400).send("Invalid signature");
    }

    const event = JSON.parse(req.body);
    res.json({ received: true });
  }
);
```

The three mistakes that break security

1. Parsing the body before verifying

This is the most common failure. Since the framework already hands you the body as an object, it's tempting to sign over that re-serialized version. But re-serializing doesn't guarantee the same key order, the same whitespace, or the same Unicode escaping the sender used. One different byte and the HMAC changes completely. Always work on the raw bytes.

2. Comparing with the normal operator instead of constant time

Comparing `expected == received` looks harmless, but it leaks information. A string comparison stops at the first character that differs, so it takes a touch longer when the leading characters already match. By measuring thousands of attempts, an attacker can reconstruct the signature character by character. That's why `hmac.compare_digest` in Python and `crypto.timingSafeEqual` in Node always take the same time, no matter where the difference is.

3. Ignoring the timestamp

If you only validate the signature, anyone who intercepts a legitimate webhook can replay it a thousand times: every copy carries a perfectly valid signature. Rejecting events whose timestamp is more than five minutes off the current clock closes that window. Combine it with idempotency —storing the ID of every event already processed— so that not even a replay within the window causes a double charge.

Production best practices

- Load the secret from an environment variable; never hard-code it or let it leak into your logs.
- Respond with a 2xx as soon as possible and process the event on a queue or background task: providers retry if you take too long, and that multiplies duplicates.
- Return 400 on an invalid signature and 200 on an already-processed duplicate, so the sender stops retrying something you already received.
- Support secret rotation by accepting two keys at once during the changeover; that way you rotate with zero downtime.
- Serve the endpoint over HTTPS only: the signature protects integrity, but TLS protects confidentiality.

Conclusion

Verifying signatures is one of the best effort-to-payoff defenses there is: about twenty lines of code and your endpoint stops accepting events from any stranger. Together with idempotency and rate limiting, webhook verification completes the basic trio for exposing an API to third parties without losing sleep.