Validating webhook deliveries

In this article

You can use a webhook secret to verify that a webhook delivery is from GitHub.

About validating webhook deliveries

Once your server is configured to receive payloads, it will listen for any delivery that's sent to the endpoint you configured. To ensure that your server only processes webhook deliveries that were sent by GitHub and to ensure that the delivery was not tampered with, you should validate the webhook signature before processing the delivery further. This will help you avoid spending server time to process deliveries that are not from GitHub and will help avoid man-in-the-middle attacks.

To do this, you need to:

1. Create a secret token for a webhook.
2. Store the token securely on your server.
3. Validate incoming webhook payloads against the token, to verify that they are coming from GitHub and were not tampered with.

Creating a secret token

You can create a new webhook with a secret token, or you can add a secret token to an existing webhook. When creating a secret token, you should choose a random string of text with high entropy.

- *To create a new webhook with a secret token*, see "Creating webhooks."
- *To add a secret token to an existing webhook*, edit the webhook's settings. Under "Secret", type a string to use as a `secret` key. For more information, see "Editing webhooks."

Securely storing the secret token

After creating a secret token, you should store it in a secure location that your server can access. Never hardcode a token into an application or push a token to any repository. For more information about how to use authentication credentials securely in your code, see

"Keeping your API credentials secure."

Validating webhook deliveries

GitHub Enterprise Server will use your secret token to create a hash signature that's sent to you with each payload. The hash signature will appear in each delivery as the value of the `X-Hub-Signature-256` header. For more information, see "Webhook events and payloads."

In your code that handles webhook deliveries, you should calculate a hash using your secret token. Then, compare the hash that GitHub sent with the expected hash that you calculated, and ensure that they match. For examples showing how to validate the hashes in various programming languages, see "Examples."

There are a few important things to keep in mind when validating webhook payloads:

- GitHub Enterprise Server uses an HMAC hex digest to compute the hash.
- The hash signature always starts with `sha256=`.
- The hash signature is generated using your webhook's secret token and the payload contents.
- If your language and server implementation specifies a character encoding, ensure that you handle the payload as UTF-8. Webhook payloads can contain unicode characters.
- Never use a plain `==` operator. Instead consider using a method like `secure_compare` or `crypto.timingSafeEqual`, which performs a "constant time" string comparison to help mitigate certain timing attacks against regular equality operators, or regular loops in JIT-optimized languages.

Testing the webhook payload validation

You can use the following `secret` and `payload` values to verify that your implementation is correct:

- `secret`: "It's a Secret to Everybody"
- `payload`: "Hello, World!"

If your implementation is correct, the signatures that you generate should match the following signature values:

- signature: `757107ea0eb2509fc211221cce984b8a37570b6d7586c22c46f4379c8b043e17`
- X-Hub-Signature-256: `sha256=757107ea0eb2509fc211221cce984b8a37570b6d7586c22c46f4379c8b043e17`

Examples

You can use your programming language of choice to implement HMAC verification in your code. Following are some examples showing how an implementation might look in various programming languages.

Ruby example

For example, you can define the following `verify_signature` function:

```ruby
def verify_signature(payload_body)
  signature = 'sha256=' + OpenSSL::HMAC.hexdigest(OpenSSL::Digest.new('sha256'),
ENV['SECRET_TOKEN'], payload_body)
  return halt 500, "Signatures didn't match!" unless
Rack::Utils.secure_compare(signature, request.env['HTTP_X_HUB_SIGNATURE_256'])
end
```

Then you can call it when you receive a webhook payload:

```
post '/payload' do
  request.body.rewind
  payload_body = request.body.read
  verify_signature(payload_body)
  push = JSON.parse(payload_body)
  "I got some JSON: #{push.inspect}"
end
```

Python example

For example, you can define the following `verify_signature` function and call it when you receive a webhook payload:

```
import hashlib
import hmac
def verify_signature(payload_body, secret_token, signature_header):
    """Verify that the payload was sent from GitHub by validating SHA256.

    Raise and return 403 if not authorized.

    Args:
        payload_body: original request body to verify (request.body())
        secret_token: GitHub app webhook token (WEBHOOK_SECRET)
        signature_header: header received from GitHub (x-hub-signature-256)
    """
    if not signature_header:
        raise HTTPException(status_code=403, detail="x-hub-signature-256 header
is missing!")
    hash_object = hmac.new(secret_token.encode('utf-8'), msg=payload_body,
digestmod=hashlib.sha256)
    expected_signature = "sha256=" + hash_object.hexdigest()
    if not hmac.compare_digest(expected_signature, signature_header):
        raise HTTPException(status_code=403, detail="Request signatures didn't
match!")
```

JavaScript example

For example, you can define the following `verifySignature` function and call it in any JavaScript environment when you receive a webhook payload:

```
let encoder = new TextEncoder();

async function verifySignature(secret, header, payload) {
    let parts = header.split("=");
    let sigHex = parts[1];

    let algorithm = { name: "HMAC", hash: { name: 'SHA-256' } };

    let keyBytes = encoder.encode(secret);
    let extractable = false;
    let key = await crypto.subtle.importKey(
        "raw",
        keyBytes,
        algorithm,
        extractable,
        [ "sign", "verify" ],
    );

    let sigBytes = hexToBytes(sigHex);
    let dataBytes = encoder.encode(payload);
    let equal = await crypto.subtle.verify(
```

```
        algorithm.name,
        key,
        sigBytes,
        dataBytes,
    );

    return equal;
}

function hexToBytes(hex) {
    let len = hex.length / 2;
    let bytes = new Uint8Array(len);

    let index = 0;
    for (let i = 0; i < hex.length; i += 2) {
        let c = hex.slice(i, i + 2);
        let b = parseInt(c, 16);
        bytes[index] = b;
        index += 1;
    }

    return bytes;
}
```

Typescript example

For example, you can define the following `verify_signature` function and call it when you receive a webhook payload:

JavaScript

```
import * as crypto from "crypto";

const WEBHOOK_SECRET: string = process.env.WEBHOOK_SECRET;

const verify_signature = (req: Request) => {
  const signature = crypto
    .createHmac("sha256", WEBHOOK_SECRET)
    .update(JSON.stringify(req.body))
    .digest("hex");
  let trusted = Buffer.from(`sha256=${signature}`, 'ascii');
  let untrusted =  Buffer.from(req.headers.get("x-hub-signature-256"), 'ascii');
  return crypto.timingSafeEqual(trusted, untrusted);
};

const handleWebhook = (req: Request, res: Response) => {
  if (!verify_signature(req)) {
    res.status(401).send("Unauthorized");
    return;
  }
  // The rest of your logic here
};
```

Troubleshooting

If you are sure that the payload is from GitHub but the signature verification fails:

- Make sure that you have configured a secret for your webhook. The `X-Hub-Signature-256` header will not be present if you have not configured a secret for your webhook. For more information about configuring a secret for your webhook, see "Editing webhooks."
- Make sure you are using the correct header. GitHub recommends that you use the `X-Hub-Signature-256` header, which uses the HMAC-SHA256 algorithm. The `X-Hub-`

`Signature` header uses the HMAC-SHA1 algorithm and is only included for legacy purposes.

- Make sure that you are using the correct algorithm. If you are using the `X-Hub-Signature-256` header, you should use the HMAC-SHA256 algorithm.
- Make sure you are using the correct webhook secret. If you don't know the value of your webhook secret, you can update your webhook's secret. For more information, see "Editing webhooks."
- Make sure that the payload and headers are not modified before verification. For example, if you use a proxy or load balancer, make sure that the proxy or load balancer does not modify the payload or headers.
- If your language and server implementation specifies a character encoding, ensure that you handle the payload as UTF-8. Webhook payloads can contain unicode characters.

Further reading

- "Handling webhook deliveries"
- "Best practices for using webhooks"