http-signatures-php Documentation

Liam Dennehy

Aug 05, 2019

Contents:

PHP implementation of Signing HTTP Messages draft IETF specification, allowing cryptographic signing and verifying of PHP PSR-7 messages.

CHAPTER 1

Quickstart

This page provides a quick introduction to HTTP Signatures PHP library and introductory examples.

If you have not already installed HTTP Signatures PHP library head over to the *Installation* page.

1.1 Signing a message

Once you have a PSR-7 message ready to send, create a Context with:

- your chosen algorithm
- the list of headers to include in the signature
- the key you will use to sign the message

For these examples we will sign the method + URI (indicated by `(request-target)`) and the `Content-Type` header. This provides a very basic level of protection, and you should consider the headers you sign in your application carefully. These may also be specified by the verifier (most often a server hosting an API or web service).

Note also that this does not apply only to HTTP requests sent by a client. Servers can add a signature to responses that the client can verify.

1.1.1 Shared Secret Context (HMAC)

This type of signature uses a secret key known to you and the verifier.

```
use HttpSignatures\Context;

$context = new HttpSignatures\Context([
  'keys' => ['key12' => 'your-secret-here'],
  'algorithm' => 'hmac-sha256',
  'headers' => ['(request-target)', 'Content-Type'],
]);
```

1.1.2 Private Key Context (RSA)

This type of signature uses a private key known only to you, which can be verified using a public key that is known to anyone who wants to verify the message.

The key file is assumed to be an unencrypted private key in PEM format.

```
use HttpSignatures\Context;

$context = new Context([
  'keys' => ['key43' => file_get_contents('/path/to/privatekeyfile')],
  'algorithm' => 'rsa-sha256',
  'headers' => ['(request-target)', 'Date', 'Accept'],
]);
```

1.1.3 Signing the Message:

```
$context->signer()->sign($message);
```

Now *$message* contains the `Signature` header:

```
$message->headers->get('Signature');
// keyId="examplekey",algorithm="hmac-sha256",headers="...",signature="..."
```

1.1.4 Adding a Digest header while signing

Include a `Digest` header automatically when signing to also protect the payload (body) of the message in addition to the request-target and headers:

```
$context->signer()->signWithDigest($message);
$message->headers->get('digest');
// SHA-256=<base64SHA256Digest>
```

1.2 Verifying a Signed Message

Most parameters are derived from the Signature in the signed message, so the Context can be created with fewer parameters.

It is probably most useful to create a Context with multilpe keys/certificates. the signature verifier will look up the key using the keyId attribute of the Signature header and use that to validate the signature.

1.2.1 Verifying a HMAC signed message

A message signed with an hmac signature is verified using the same key as the one used to sign the original message:

```
use HttpSignatures\Context;

$context = new HttpSignatures\Context([
  'keys' => ['key300' => 'some-other-secret',
             'key12' => 'secret-here']
```

(continues on next page)

(continued from previous page)

```
]);

$context->verifier()->isSigned($message); // true or false
```

1.2.2 Verifying a RSA signed message

An RSA signature is verified using the certificate associated with the Private Key that created the message. Create a context by importing the X.509 PEM format certificates in place of the 'secret':

```
use HttpSignatures\Context;

$context = new HttpSignatures\Context([
  'keys' => ['key43' => file_get_contents('/path/to/certificate'),
             'key87' => $someOtherCertificate],
$context->verifier()->isSigned($message); // true or false
]);
```

1.2.3 Verifying a message digest

To confirm the body has a valid digest header and the header is a valid digest of the message body:

```
$context->verifier()->isValidDigest($message); // true or false
```

An all-in-one validation that the signature includes the digest, and the digest is valid for the message body:

```
$context->verifier()->isSignedWithDigest($message); // true or false
```

1.3 Symfony compatibility

Symfony requests normalize query strings which means the resulting request target can be incorrect. See https://github.com/symfony/psr-http-message-bridge/pull/30

When creating PSR-7 requests you use *withRequestTarget* to ensure the request target is correct. For example

```
use Symfony\Bridge\PsrHttpMessage\Factory\DiactorosFactory;
use Symfony\Component\HttpFoundation\Request;

$symfonyRequest = Request::create('/foo?b=1&a=2');
$psrRequest = (new DiactorosFactory())
      ->createRequest($symfonyRequest)
      ->withRequestTarget($symfonyRequest->getRequestUri());
```

CHAPTER 2

The HTTP Signature

This section is based on the definitions and descriptions in Signing HTTP Messages IETF draft RFC version 10.

Table of Contents

2.1 Signature Line

```
keyId="abc123",algorithm="rsa-sha256",headers="(request-target) date",signature=
↪"base64string"
```

The Signature line is the component of a signature header that describes the parameters of how a message was signed as well as the actual digital signature.

These parameters together should provide any verifier with the information required to prove the validity of a signature against the HTTP message it accompanies.

The parameters of the Signature Line are described here

2.1.1 keyId

As desribed in the draft RFC, the `keyId` parameter is used by the verifier to look up the key that can be used to verify the provided signature.

- In the HMAC case these are the same key - the shared secret.
- In the RSA or EC case, this is the public component of the key.

Note that the RFC is not specific about the meaning of the parameter's value. This could be a fingerprint of the certificate containing the key, the e-mail address of the signer, or even no value at all if the verifier can determine which key to use by another means entirely e.g. if the key/certificate is provided in a dedicated header.

The value of `keyId` must therefore be agreed before the message is transmitted - either by agreeing an explicit value, or the format of the value acceptable to the verifier if it not distinct. This is typically found in the API documentation for the resource.

2.1.2 algorithm

The `algorithm` parameter informs the verifier which hash algorithm was used to generate the hash signed by the signature ("hash" algorithm), and which cryptographic algorithm was used to sign that resulting hash ("signature algorithm").

The hash algorithm cannot be deduced simply by looking at the key and signature, so must be provided in this parameter.

However the verifier should not rely on the signature algorithm part of the `algorithm` parameter alone to determine which signature algorithm to use. Rather the "metadata" (e.g. which elliptic curve algorithm the key is designed for) associated with the key should be relied on separate from the signed message.

This arises as some types of keys can be used in multiple modes, and selecting the wrong mode for verification may introduce security issues.

In any case the signer and verifier should agree which hash and signature algorithms are acceptable for a given request/response.

2.1.3 headers

The `headers` parameter is a space-delimited list of the headers that are included in the signature itself. These headers are specified in lowercase, and let the verifier know which order to place the headers in when the signature is verified - so this order cannot be altered.

The signer and verifier(s) need to agree on which headers should be included in any signature, especially if there are minimum headers that must be included and any that are forbidden.

2.1.4 signature

The `signature` parameter is simply a base64-encoded string representing the raw digital signature (which is likely encoded with unprintable characters).

The verifier can use this string, along with the other parameters and headers in the HTTP message, to verify the contents of the message (specifically the message's *headers*) have not been altered since the signer generated the signature.

2.2 Headers

2.2.1 Authorization header

```
Authorization: Signature <signatureline>
```

The `Authorization` header is described in **RFC 7235#section-4.2** and provides a way for a HTTP client to "authenticate itself with an origin server". This gives a hint that the header is used almost exclusively by a client when talking to a server.

The first parameter of an `Authorization` header is the authorization type, of which many have been defined. When the type is `Signature`, the server will expect the next parameters to be a *Signature Line* according to the specifications of https://tools.ietf.org/html/draft-cavage-http-signatures

Since this header is involved primarily with authenticating a client to a server, this header is not typically used to protect the content of a message, and is not useful in a HTTP Response.

2.2.2 Signature header

```
Signature: <signatureline>
```

The `Signature` header is a new HTTP header proposed in https://tools.ietf.org/html/draft-cavage-http-signatures.

The value of the header is simply the ref:*header-signatureline*.

This header is more versatile than the `Authorization` header as it can be used:

- by both the client *and* server (HTTP request and HTTP respnse respectively)
- to prove the identity of the signer (similar to the `Authorization` header in `Signature` mode
- in addition to an `Authorization` header when needed

2.2.3 Digest header

```
Digest: SHA-256=<base64string>
```

The `Digest` header is a way to determine the integrity of the payload (aka body) of a HTTP request. Including the `Digest` in the signature's *signature* allows the integrity of the payload to be included in the signature itself.

CHAPTER 3

API Reference

Table of Contents

3.1 Class: Context

```
new Context($args)
```

CHAPTER 4

Usage

Add liamdennehy/http-signatures-php to your `composer.json`. Full instructions can be found in *Installation*

To quickly see how a message is signed, take a look in *Signing a message* in the Quickstart guide.

CHAPTER 5

Requirements

1. PHP 5.6 (PHP >7.0 recommended)
2. Composer for full autoloading of class loading
3. Understanding of PSR-7 HTTP message handling

5.1 Installation

The recommended way to install http-signatures-php is with Composer. Composer is a dependency management tool for PHP that allows you to declare the dependencies your project needs and installs them into your project.

```
# Install Composer
curl -sS https://getcomposer.org/installer | php
```

You can add http-signatures-php as a dependency using the `composer.phar` CLI:

```
php composer.phar require liamdennehy/http-signatures-php
```

Alternatively, you can specify http-signatures-php as a dependency in your project's existing `composer.json` file:

```
{
  "require": {
     "liamdennehy/http-signatures-php": "~6.0"
  }
}
```

After installing, you need to require Composer's autoloader in your project to be able to locate the library within PHP:

```
require __DIR__ . '/vendor/autoload.php';
```

You can find out more on how to install Composer, configure autoloading, and other best-practices for defining dependencies at getcomposer.org.

CHAPTER 6

Contributing

Pull Requests are welcome.

CHAPTER 7

License

Index

R