HTTP
Internet-Draft
Intended status: Standards Track
Expires: 11 October 2026

D. Hardt
Hell
T. Meunier
Cloudflare

9 April 2026

HTTP Signature Keys
draft-hardt-httpbis-signature-key-04

Abstract

This document defines two HTTP header fields and one Accept-Signature parameter for use with HTTP Message Signatures as defined in RFC 9421. The Signature-Key request header distributes public keys used to verify signatures, with five initial key distribution schemes: pseudonymous inline keys (hwk), self-issued key delegation via JWK Thumbprint JWTs (jkt-jwt), identified signers with JWKS URI discovery (jwks_uri), JWT-based delegation (jwt), and X.509 certificate chains (x509). The sigkey parameter extends Accept-Signature (RFC 9421 Section 5) to indicate the type of Signature-Key the server requires. The Signature-Error response header provides structured error information when signature verification fails. Together, these mechanisms enable flexible trust models ranging from privacy-preserving pseudonymous verification to horizontally-scalable delegated authentication and PKI-based identity chains.

Discussion Venues

Note: This section is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/dickhardt/signature-key (https://github.com/dickhardt/signature-key).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Conventions and Definitions

{::boilerplate bcp14-tagged}

2. Introduction

HTTP Message Signatures [RFC9421] provides a powerful mechanism for creating and verifying digital signatures over HTTP messages. To verify a signature, the verifier needs the signer's public key. While RFC 9421 defines signature creation and verification procedures, it intentionally leaves key distribution to application protocols, recognizing that different deployments have different trust requirements.

This document defines:

* *Signature-Key* (Signature-Key HTTP Request Header (#signature-key-http-request-header)) — a request header that distributes public keys for HTTP Message Signature verification. The header supports five schemes, each designed for different trust models and operational requirements:

    1. *Header Web Key (hwk)* - Self-contained public keys for pseudonymous verification
    2. *JKT JWT (jkt-jwt)* - Self-issued key delegation via JWK Thumbprint JWTs ("jacket jot")
    3. *JWKS URI (jwks_uri)* - Identified signers with key discovery via metadata
    4. *JWT (jwt)* - Delegated keys embedded in signed JWTs for horizontal scale
    5. *X.509 (x509)* - Certificate-based verification with PKI trust chains

Additional schemes may be defined through the IANA registry established by this document.

* *sigkey* (Accept-Signature sigkey Parameter (#accept-signature-sigkey-parameter)) — a parameter for the Accept-Signature header ([RFC9421], Section 5) that indicates the type of Signature-Key the server requires. This extends RFC 9421's existing mechanism for requesting signatures rather than defining a new header.

* *Signature-Error* (Signature-Error HTTP Response Header (#signature-error-http-response-header)) — a response header that provides structured error information when signature verification fails, enabling clients to diagnose and correct signing issues.

The Signature-Key header works in conjunction with the Signature-Input and Signature headers defined in RFC 9421, using matching labels to correlate signature metadata with keying material.

3. Signature-Key HTTP Request Header

The Signature-Key header provides the public key or key reference needed to verify an HTTP Message Signature. It is a Structured Field Dictionary [RFC8941] keyed by signature label, where each member describes how to obtain the verification key for the corresponding signature.

Format:

Signature-Key: <label>=<scheme>;<parameters>...

Where: - <label> (dictionary key) matches the label in Signature-Input and Signature headers - <scheme> (token) identifies the key distribution scheme - <parameters> are semicolon-separated key-value pairs whose values are structured field strings or byte sequences, varying by scheme

Multiple keys are comma-separated per the dictionary format. See [RFC8941] for definitions of dictionary, token, string, and byte sequence.

Example:

```
Signature-Input: sig=("@method" "@authority" "@path" "signature-key"); created=1732210000
Signature: sig=:MEQCIA5...
Signature-Key: sig=hwk;kty="OKP";crv="Ed25519";x="JrQLj..."
```

Label Correlation:

Labels are correlated by equality of label names across Signature-Input, Signature, and Signature-Key. Signature-Key is a dictionary keyed by label; Signature-Input and Signature are the sources of what signatures are present; Signature-Key provides keying material for those labels.

Verifiers MUST:

1. Parse Signature-Input and Signature per RFC 9421 and obtain the set of signature labels present. The verifier determines which labels it is attempting to verify based on application context and RFC 9421 processing.

2. Parse Signature-Key as a Structured Fields Dictionary

3. For each label being verified, select the Signature-Key dictionary member with the same name

4. If the Signature-Key header is present and the verifier is attempting to verify a label using it, but the corresponding dictionary member is missing, verification for that signature MUST fail

| *Note:* A verifier might choose to verify only a subset of labels
| present (e.g., the application-required signature); labels not
| verified can be ignored.

Signatures whose keys are distributed through mechanisms outside this specification (e.g., pre-configured keys, out-of-band key exchange) are out of scope. A Signature-Key header is not required for such signatures, and verifiers MAY use application-specific means to obtain the verification key.

3.1. Label Consistency

If a label appears in Signature or Signature-Input, and the verifier attempts to verify it using Signature-Key, the corresponding member MUST exist in Signature-Key. If Signature-Key contains members for labels not being verified, verifiers MAY ignore them.

3.2. Multiple Signatures

The dictionary format supports multiple signatures per message. Each signature has its own dictionary member keyed by its unique label:

```
Signature-Input: sig1=(... "signature-key"), sig2=(... "signature-key")
Signature: sig1=:...:, sig2=:...:
Signature-Key: sig1=jwt;jwt="eyJ...", sig2=jwks_uri;id="https://example.com";dwk="eg-conf
ig";kid="k1"
```

Most deployments SHOULD use a single signature. When multiple signatures are required, the complete Signature-Key header (containing all keys) MUST be populated before any signature is created, and each signature MUST cover signature-key. This ensures all signatures protect the integrity of all key material. See Signature-Key Integrity (#signature-key-integrity) in Security Considerations. Alternative key distribution mechanisms outside this specification may be used for scenarios requiring independent signature addition.

3.3. Header Web Key (hwk)

The hwk scheme provides a self-contained public key inline in the header, enabling pseudonymous verification without key discovery. The parameter names and values correspond directly to the JWK parameters defined in [RFC7517].

Parameters by key type:

OKP (Octet Key Pair):

* kty (REQUIRED, String) - "OKP"
* crv (REQUIRED, String) - Curve name (e.g., "Ed25519")
* x (REQUIRED, String) - Public key value

Signature-Key: sig=hwk;kty="OKP";crv="Ed25519";x="JrQLj5P..."

EC (Elliptic Curve):

* kty (REQUIRED, String) - "EC"
* crv (REQUIRED, String) - Curve name (e.g., "P-256", "P-384")
* x (REQUIRED, String) - X coordinate
* y (REQUIRED, String) - Y coordinate

Signature-Key: sig=hwk;kty="EC";crv="P-256";x="f83OJ3D...";y="x_FEzRu..."

RSA:

* kty (REQUIRED, String) - "RSA"
* n (REQUIRED, String) - Modulus
* e (REQUIRED, String) - Exponent

Signature-Key: sig=hwk;kty="RSA";n="0vx7agoebGcQ...";e="AQAB"

Constraints:

* The alg parameter MUST NOT be present (algorithm is derived from the key type and curve)
* The kid parameter SHOULD NOT be used

Use cases:

* Privacy-preserving agents that avoid identity disclosure

* Experimental or temporary access without registration

* Rate limiting and reputation building on a per-key basis

3.4. JKT JWT Self-Issued Key Delegation (jkt-jwt)

The jkt-jwt scheme (pronounced "jacket jot") provides self-issued key delegation using a JWT whose signing key is embedded in the JWT header. This enables devices with hardware-backed secure enclaves to delegate signing authority to ephemeral keys, avoiding the performance cost of repeated enclave operations while maintaining a cryptographic chain of trust rooted in the enclave key.

Many devices — mobile phones, laptops, IoT hardware — include secure enclaves or trusted execution environments (e.g., Apple Secure Enclave, Android StrongBox, TPM) that can generate and store private keys with strong protection guarantees. However, signing operations using these enclaves are comparatively slow and may require user interaction (biometric confirmation, PIN entry).

For HTTP Message Signatures, where every request requires a signature, this creates a tension between security and performance. The jkt-jwt scheme resolves this by allowing the enclave key to sign a JWT that delegates authority to a faster ephemeral key:

1. The enclave generates a long-lived key pair (the identity key)
2. The device generates an ephemeral key pair in software (the signing key)
3. The enclave signs a JWT binding the ephemeral key via the cnf claim
4. HTTP requests are signed with the fast ephemeral key
5. The JWT proves the ephemeral key was authorized by the enclave key

The enclave key's JWK Thumbprint URI (urn:jkt:<hash-algorithm>:<thumbprint>) serves as a stable, pseudonymous device identity. Verifiers build trust in this identity over time (TOFU — Trust On First Use [RFC7435]).

Parameters:

* jwt (REQUIRED, String) - Compact-serialized JWT

JWT requirements:

Header:

* typ (REQUIRED) - Identifies the thumbprint hash algorithm. Defined values: jkt-s256+jwt (SHA-256), jkt-s512+jwt (SHA-512). Implementations MUST support jkt-s256+jwt and MAY support additional algorithms.

* alg (REQUIRED) - Signature algorithm used by the enclave key

* jwk (REQUIRED) - JWK public key of the enclave/identity key (the key that signed this JWT)

Payload:

* iss (REQUIRED) - JWK Thumbprint URI of the signing key, in the format urn:jkt:<hash-algorithm>:<thumbprint> where the thumbprint is computed per [RFC7638]. The hash algorithm in the URN MUST match the algorithm indicated by the JWT typ. The verifier knows the hash algorithm from the typ it accepted, computes the thumbprint of the header jwk, prepends the known urn:jkt:<hash-algorithm>: prefix, and compares to iss by string equality.

* iat (REQUIRED) - Issued-at timestamp

* exp (REQUIRED) - Expiration timestamp

* cnf (REQUIRED) - Confirmation claim [RFC7800] containing jwk: the ephemeral public key delegated for HTTP message signing

The sub claim is not used. The identity is the enclave key itself, fully represented by the iss thumbprint.

JWT Type Values:

The typ value encodes both the purpose and the thumbprint hash algorithm:

typ	Hash Algorithm	iss prefix
jkt-s256+jwt	SHA-256	urn:jkt:sha-256:
jkt-s512+jwt	SHA-512	urn:jkt:sha-512:

Table 1

The jkt- prefix indicates a self-issued delegation JWT: the signing key is embedded in the JWT header as a JWK, the issuer is identified by the key's thumbprint, and the JWT delegates signing authority to the key in the cnf claim. The suffix (s256, s512) identifies the hash algorithm used for the thumbprint. The typ and iss prefix MUST be consistent.

These types are independent of the Signature-Key header and MAY be used in other contexts where self-issued key delegation is needed. Additional hash algorithms can be supported by registering new typ values following the jkt-<alg>+jwt pattern.

Example:

Signature-Key: sig=jkt-jwt;jwt="eyJ..."

JWT header:

```
{
  "typ": "jkt-s256+jwt",
  "alg": "ES256",
  "jwk": {
    "kty": "EC",
    "crv": "P-256",
    "x": "f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU",
    "y": "x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0"
  }
}
```

JWT payload:

```
{
  "iss": "urn:jkt:sha-256:NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs",
  "iat": 1732210000,
  "exp": 1732296400,
  "cnf": {
    "jwk": {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "JrQLj5P_89iXES9-vFgrIy29clF9CC_oPPsw3c5D0bs"
    }
  }
}
```

In this example, the enclave holds a P-256 key (signed via hardware) and delegates to an Ed25519 ephemeral key (signed in software). The identity is urn:jkt:sha-256:NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs.

Verification procedure:

1. Parse the JWT without verifying the signature
2. Check the typ header (e.g., jkt-s256+jwt). Reject if the type is not supported.
3. Determine the hash algorithm and iss prefix from the typ (e.g., jkt-s256+jwt → SHA-256, urn:jkt:sha-256:)
4. Extract the jwk from the JWT header
5. Compute the JWK Thumbprint ([RFC7638]) of the header jwk using the determined hash algorithm
6. Construct the expected iss value by prepending the known prefix to the computed thumbprint
7. Verify the iss claim matches the constructed value by string equality
8. Verify the JWT signature using the header jwk
9. Validate exp and iat claims per policy
10. Extract the ephemeral public key from cnf.jwk
11. Verify the HTTP Message Signature using the ephemeral key

Use cases:

* Devices with hardware-backed secure enclaves delegating to fast ephemeral keys
* Persistent pseudonymous identity without requiring registration or authority
* Mobile apps, laptops, and IoT devices with enclave-backed identity

3.5. JWKS URI Discovery (jwks_uri)

The jwks_uri scheme identifies the signer and enables key discovery via a metadata document containing a jwks_uri property.

Parameters:

* id (REQUIRED, String) - Signer identifier (HTTPS URL)

* dwk (REQUIRED, String) - Dot well-known metadata document name under /.well-known/

* kid (REQUIRED, String) - Key identifier

Discovery procedure:

1. Fetch {id}/.well-known/{dwk}

2. Parse as JSON metadata

3. Extract jwks_uri property

4. Fetch JWKS from jwks_uri

5. Find key with matching kid

Example:

```
Signature-Key: sig=jwks_uri;id="https://client.example";dwk="example-configuration";kid="
key-1"
```

Use cases:

* Identified services with stable HTTPS identity

* Search engine crawlers and monitoring services

* Services requiring explicit entity identification

3.6. JWT Confirmation Key (jwt)

The jwt scheme embeds a public key inside a signed JWT using the cnf (confirmation) claim [RFC7800], enabling delegation and horizontal scale.

Parameters:

* jwt (REQUIRED, String) - Compact-serialized JWT

JWT requirements:

* MUST contain cnf.jwk claim with embedded JWK

* SHOULD contain iss claim (HTTPS URL of the issuer) — using SHOULD rather than MUST allows existing JWT infrastructure to be used without modification

* SHOULD contain dwk claim (dot well-known metadata document name) — the verifier constructs {iss}/.well-known/{dwk} to discover the issuer's jwks_uri. Using SHOULD allows deployments where the verifier already knows the issuer's keys.

* SHOULD contain standard claims: sub, exp, iat

* Verifiers SHOULD verify the JWT typ header parameter has an expected value per deployment policy, to optimize for a quick rejection

| *Note:* The mechanism by which the JWT is obtained is out of scope
| of this specification.

Verification procedure:

1. Parse the JWT parameter value per [RFC7519] Section 7.2. Reject if the value is not a well-formed JWT. This and subsequent pre-signature checks allow the verifier to fail early without expensive cryptographic operations or network fetches.

2. Verify the JWT typ header parameter has an expected value per policy. Reject if unexpected.

3. Validate exp claim if present. Reject if the token has expired.

4. Verify required claims are present (cnf.jwk, plus any claims required by deployment policy). Reject if a required claim is missing.

5. If iss and dwk claims are present, fetch {iss}/.well-known/{dwk}, parse as JSON metadata, extract jwks_uri. Fetch JWKS from jwks_uri, find key matching kid in JWT header. If iss or dwk is absent, the verifier MUST obtain the issuer's key through an application-specific mechanism.

6. Verify JWT signature using the discovered key

7. Validate remaining JWT claims per policy (iss, sub, etc.)

8. Extract JWK from cnf.jwk

9. Verify HTTP Message Signature using extracted key

Example:

```
Signature-Key: sig=jwt;jwt="eyJhbGciOiJFUzI1NiI..."
```

JWT payload example:

```
{
  "iss": "https://issuer.example",
  "dwk": "example-configuration",
  "sub": "instance-123",
  "exp": 1732210000,
  "cnf": {
    "jwk": {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "JrQLj5P_89iXES9-vFgrIy29clF9CC_oPPsw3c5D0bs"
    }
  }
}
```

Use cases:

* Distributed services with ephemeral instance keys

* Delegation scenarios where instances act on behalf of an authority

* Short-lived credentials for horizontal scaling

3.7. X.509 Certificates (x509)

The x509 scheme provides certificate-based verification using PKI trust chains.

Parameters:

* x5u (REQUIRED, String) - URL to X.509 certificate chain (PEM format, [RFC7517] Section 4.6)

* x5t (REQUIRED, Byte Sequence) - Certificate thumbprint: SHA-256 hash of DER-encoded end-entity certificate

Verification procedure:

1. Check cache for certificate with matching x5t

2. If not cached or expired, fetch PEM from x5u

3. Validate certificate chain to trusted root CA

4. Check certificate validity and revocation status

5. Verify x5t matches end-entity certificate

6. Extract public key from end-entity certificate

7. Verify signature using extracted key

8. Cache certificate indexed by x5t

Example:

```
Signature-Key: sig=x509;x5u="https://client.example/.well-known/cert.pem";x5t=:bWcoon4QTV
n8Q6xiY0ekMD6L8bNLMkuDV2KtvsFc1nM=:
```

Use cases:

* Enterprise environments with PKI infrastructure

* Integration with existing certificate management systems

* Scenarios requiring certificate revocation checking

* Regulated industries requiring certificate-based authentication

4. Accept-Signature sigkey Parameter

[RFC9421] Section 5 defines the Accept-Signature response header for requesting HTTP Message Signatures. This document extends Accept-Signature with a sigkey parameter that indicates the type of Signature-Key the server requires.

4.1. Parameter Definition

The sigkey parameter is an Item parameter on each member of the Accept-Signature Dictionary. Its value is a Token ([RFC8941], Section 3.3.4) with three defined values:

Value	Meaning	Acceptable Signature-Key schemes
jkt	Pseudonymous key identified by JWK Thumbprint	hwk, jkt-jwt
uri	Key identified by a URI	jwks_uri, jwt, x509 (with URI SAN)
x509	Key from an X.509 certificate chain	x509

Table 2

These values represent ordered levels of identification. A server requesting sigkey=uri accepts any scheme that provides a URI-based identifier. A server requesting sigkey=x509 specifically requires PKI infrastructure.

When sigkey is present, the keyid parameter ([RFC9421], Section 5) SHOULD NOT be included and MUST be ignored by the client. Key identification is handled by the Signature-Key header schemes, not by keyid. The algs and tag parameters remain applicable alongside sigkey.

4.2. Label Binding

The signature label in Accept-Signature ties together all four headers on the signed request. When a server requests:

```
Accept-Signature: sig1=("@method" "@path" "@authority");
    alg="ecdsa-p256-sha256";sigkey=uri
```

The client responds with matching labels:

```
Signature-Key: sig1=jwks_uri;id="https://client.example";dwk="example-configuration";kid=
"key-1"
Signature-Input: sig1=("@method" "@path" "@authority" "signature-key");
    created=1732210000;keyid="https://client.example"
Signature: sig1=:MEQCIA5...:
```

The signature-key covered component is added by the client per this specification’s requirement that signature-key appear in covered components. The server does not need to list it in Accept-Signature.

4.3. Response Status Codes

Accept-Signature with a sigkey parameter can be set for any response. Below is a list of what it MAY mean on responses with the following status codes:

Status	Meaning	Legacy client behavior	Signature-aware client behavior
401	Authentication required	Falls back to WWW-Authenticate	Signs request with appropriate Signature-Key scheme
402	Payment + authentication required	Processes payment mechanism	Signs request AND processes payment
429	Rate limited	Respects Retry-After, slows down	Signs request, gets higher per-key rate limit

Table 3

The 429 case is particularly important for incremental adoption: a server can add Accept-Signature with sigkey to its existing 429 responses with zero risk. Legacy clients ignore the unknown header and respect Retry-After. Signature-aware clients sign with a pseudonymous key, giving the server a stable key thumbprint for per-client rate limiting — and the client gets a higher rate limit in return.

4.4. sigkey Semantics

4.4.1. jkt

The server requires a signed request using a pseudonymous Signature-Key scheme (hwk or jkt-jwt). The server can track the client by JWK Thumbprint ([RFC7638]) without knowing its identity. This is useful for rate limiting anonymous requests, tracking repeat visitors by key thumbprint, spam prevention without requiring verified identity, and hardware-backed pseudonymous identity.

4.4.2. uri

The server requires a signed request with a URI-identified Signature-Key (jwks_uri, jwt, or x509 with a URI SAN). This is useful for API access policies based on known clients, webhook signature verification, and allowlisting trusted clients for elevated rate limits.

4.4.3. x509

The server requires a signed request using an X.509 certificate chain (x509 scheme). This is useful for enterprise environments with PKI infrastructure, regulated industries requiring certificate-based authentication, and scenarios requiring certificate revocation checking.

[RFC9421] Section 5.2 defines the processing of Accept-Signature by the client. If the sigkey parameter is unsupported, the client MAY ignore it.

If a client already knows the server’s sigkey requirement (from a previous interaction or metadata), it MAY sign the initial request directly without waiting for a challenge response.

4.5. Incremental Adoption

Accept-Signature with sigkey is designed for zero-coordination deployment. The sigkey parameter is unknown to legacy clients and ignored per Structured Fields semantics — servers can add it to existing responses without breaking anything.

Stage 1 — Rate limiting (429): A server adds Accept-Signature with sigkey=jkt to its 429 responses. Legacy clients slow down as before. Signature-aware clients sign requests and get higher per-key rate limits. The server gains per-client rate limiting without requiring registration or API keys.

Stage 2 — Authentication (401): The server starts requiring signatures on some paths, returning 401 with Accept-Signature and sigkey=jkt. It can include WWW-Authenticate alongside for legacy clients that have other auth mechanisms. Signature-aware clients sign; legacy clients fall back to bearer tokens or other schemes.

Stage 3 — Identity (401): The server upgrades from sigkey=jkt to sigkey=uri on sensitive paths, requiring verifiable client identity via jwks_uri, jwt, or x509 schemes. The server can now make identity-based policy decisions without pre-registration.

Each stage is independently deployable. A server can use stage 1 on all endpoints while using stage 3 on admin endpoints. No bilateral agreements or client coordination required.

4.6. Coexistence with WWW-Authenticate

Accept-Signature and WWW-Authenticate ([RFC9110], Section 11.6.1) are independent header fields; a response MAY include both. A client that understands Signature-Key processes Accept-Signature with sigkey; a legacy client processes WWW-Authenticate. Neither header's presence invalidates the other.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api"
Accept-Signature: sig1=("@method" "@path" "@authority");
    alg="ecdsa-p256-sha256";sigkey=uri
```

A 402 response MAY include a payment mechanism such as x402 [x402] or the Micropayment Protocol ([I-D.ryan-httpauth-payment]) alongside Accept-Signature for authentication:

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="x7Tg2pLq", method="example",
    request="eyJhbW91bnQiOiIxMDAw..."
Accept-Signature: sig1=("@method" "@path" "@authority");sigkey=jkt
```

4.7. Examples

Pseudonymous access:

```
HTTP/1.1 401 Unauthorized
Accept-Signature: sig1=("@method" "@path" "@authority");sigkey=jkt
```

Identity with algorithm restriction:

```
HTTP/1.1 401 Unauthorized
Accept-Signature: sig1=("@method" "@authority" "@path");
    alg="ecdsa-p256-sha256";sigkey=uri
```

Rate limiting with pseudonymous upgrade:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 30
Accept-Signature: sig1=("@method" "@path" "@authority");sigkey=jkt
```

Payment with pseudonymous authentication:

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="x7Tg2pLq", method="example",
    request="eyJhbW91bnQiOiIxMDAw..."
Accept-Signature: sig1=("@method" "@path" "@authority");sigkey=jkt
```

4.8. Client Processing

When a client receives a response containing an Accept-Signature header with a sigkey parameter, it MAY retry the request with an HTTP Message Signature using a Signature-Key scheme appropriate for the indicated sigkey value.

When a 429 response includes both Retry-After and Accept-Signature with sigkey, the client MAY retry one time with a signed request without waiting for the Retry-After interval. Signing the request provides a key thumbprint that enables per-client rate limiting, which may result in a higher rate limit for the client.

A server MAY return a 429 response without Accept-Signature to a signed request when it wants to rate-limit the client regardless of signing. In this case, the client MUST respect Retry-After as usual.

| *Open Issue:* Should this specification define a baseline HTTP Message Signatures profile (minimum covered components, timestamp requirements, verification steps), or is that always the responsibility of the protocol using these headers? See GitHub issue #7 (https://github.com/dickhardt/signature-key/issues/7).

5. Signature-Error HTTP Response Header

When a server rejects a signed request due to a signature-related error, the response SHOULD include the Signature-Error header. The response status code is typically 400 Bad Request, since the signature or keying material is malformed or invalid. A server MAY use 401 Unauthorized for recoverable errors (e.g., unsupported_algorithm, invalid_input) where the client can retry with corrected parameters.

5.1. Header Structure

The Signature-Error header is a Dictionary ([RFC8941], Section 3.2) with the following member:

* error (REQUIRED): A Token ([RFC8941], Section 3.3.4) indicating the error code.

Additional members are defined per error code. Recipients MUST ignore unknown members.

```
Signature-Error: error=unsupported_algorithm,
    supported_algorithms=("ed25519" "ecdsa-p256-sha256")
```

The Signature-Error header is the authoritative source for machine-readable error information. The client MUST NOT depend on the response body for error handling.

5.2. Response Body

Servers SHOULD use Problem Details [RFC9457] (application/problem+json) for the response body when returning Signature-Error. The type member SHOULD be a URN of the form urn:ietf:params:sig-error:<error-code>, where <error-code> matches the error value in the header.

```
{
  "type": "urn:ietf:params:sig-error:unsupported_algorithm",
  "title": "Unsupported signature algorithm",
  "status": 400,
  "detail": "The server does not support rsa-v1_5-sha256"
}
```

Extension members in the Problem Details object (e.g., supported_algorithms) MAY duplicate information from the Signature-Error header for convenience. When the header and body conflict, the header takes precedence.

5.3. Access Denied

When the server successfully verifies the client's signature and identity but denies access based on policy (e.g., the client is not authorized for this resource), the server returns 403 Forbidden. This is not a signature error — the authentication succeeded but authorization was denied. The response MUST NOT include an Accept-Signature header with sigkey or a Signature-Error header.

5.4. Error Codes

5.4.1. unsupported_algorithm

The signing algorithm used by the client is not supported by the server.

* supported_algorithms (REQUIRED): An Inner List of String ([RFC8941], Section 3.1.1) listing the algorithms the server accepts, using identifiers from the HTTP Signature Algorithms registry ([RFC9421], Section 6.2). The registry description for each identifier specifies the corresponding key type and curve. The response MUST include this member.

```
Signature-Error: error=unsupported_algorithm,
    supported_algorithms=("ed25519" "ecdsa-p256-sha256")
```

5.4.2. invalid_signature

The HTTP Message Signature is missing, malformed, or cryptographic verification failed. This includes missing Signature, Signature-Input, or Signature-Key headers, an expired created timestamp, or a signature that does not verify.

```
Signature-Error: error=invalid_signature
```

5.4.3. invalid_input

The Signature-Input is missing required covered components.

* required_input (OPTIONAL): An Inner List of String ([RFC8941], Section 3.1.1) listing the covered components the server requires. The response SHOULD include this member.

```
Signature-Error: error=invalid_input,
    required_input=("@method" "@authority" "@path"
    "signature-key" "content-digest")
```

5.4.4. invalid_request

The request is malformed or missing required information unrelated to signature verification — such as missing query parameters or an unsupported content type.

```
Signature-Error: error=invalid_request
```

5.4.5. invalid_key

The public key in Signature-Key could not be parsed, is expired, or does not meet the server's trust requirements.

```
Signature-Error: error=invalid_key
```

5.4.6. unknown_key

The public key from Signature-Key does not match any key at the client's jwks_uri (applicable when the client uses scheme=jwks_uri). The server SHOULD re-fetch the JWKS once before returning this error, to handle key rotation.

```
Signature-Error: error=unknown_key
```

5.4.7. invalid_jwt

The JWT in the Signature-Key header (when using scheme=jwt or scheme=jkt-jwt) is malformed or its signature verification failed.

Signature-Error: error=invalid_jwt

5.4.8. expired_jwt

The JWT in the Signature-Key header (when using scheme=jwt or scheme=jkt-jwt) has expired (exp claim is in the past).

Signature-Error: error=expired_jwt

6. Security Considerations

6.1. Key Validation

Verifiers MUST validate all cryptographic material before use:

* *hwk*: Validate JWK structure and key parameters per [RFC7517]
* *jwks_uri*: Verify HTTPS transport and validate fetched JWKS per [RFC7517]
* *x509*: Validate complete certificate chain per [RFC5280], check revocation status
* *jwt*: Verify JWT signature per [RFC7519] and validate embedded JWK per [RFC7517]
* *jkt-jwt*: Verify JWT signature per [RFC7519] using header jwk, validate thumbprint matches iss per [RFC7638], validate embedded ephemeral JWK per [RFC7517]

6.2. Caching and Performance

Verifiers MAY cache keys to improve performance but MUST implement appropriate cache expiration:

* *jwks_uri*: Respect cache-control headers, implement reasonable TTLs
* *x509*: Cache by x5t, invalidate on certificate expiry
* *jwt*: Cache embedded keys until JWT expiration

* *jkt-jwt*: Cache embedded keys until JWT expiration; cache by iss thumbprint URI

Verifiers SHOULD implement cache limits to prevent resource exhaustion attacks.

6.3. Scheme-Specific Risks

hwk: No identity verification - suitable only for scenarios where pseudonymous access is acceptable.

jkt-jwt: The security of this scheme depends on the enclave key's private key remaining protected in hardware. If the enclave key is compromised, all delegated ephemeral keys are compromised. Verifiers should be aware that the jkt-jwt scheme implies but does not prove hardware protection — there is no attestation mechanism in this scheme. Unlike the jwt scheme where trust is rooted in a discoverable issuer, jkt-jwt trust is rooted in the key itself. Verifiers MUST understand that any party can create a jkt-jwt — the scheme provides pseudonymous identity, not verified identity. The exp claim on the JWT controls how long the ephemeral key is valid. Shorter lifetimes limit the exposure window if an ephemeral key is compromised. Implementations SHOULD use the shortest practical lifetime. The iss value is a JWK Thumbprint URI — a globally unique, collision-resistant identifier. The verifier MUST always compute the expected iss from the header jwk and compare by string equality — never trust the iss value alone.

jwks_uri: Relies on HTTPS security — vulnerable to DNS/CA compromise. Beyond HTTPS validation, nothing prevents an attacker from copying a client's public keys and serving them from a different domain. Verifiers SHOULD verify that the id parameter in the Signature-Key header matches an expected or authorized origin.

jwt: Delegation trust depends on JWT issuer verification. Verifiers MUST validate JWT signatures and claims before trusting embedded keys.

x509: Requires robust certificate validation including revocation checking. Verifiers MUST NOT skip certificate chain validation.

6.4. Algorithm Selection

The signature algorithm is determined by the key material in Signature-Key, not by the optional alg parameter in Signature-Input ([RFC9421], Section 2.3). For JWK-based schemes (hwk, jkt-jwt, jwks_uri, jwt), the algorithm is identified by the key type and curve (kty + crv) or by the alg parameter in the JWK ([RFC7517]). For the x509 scheme, the algorithm is determined by the certificate's public key type.

If the alg parameter is present in Signature-Input, verifiers MUST verify it is consistent with the key material. If it is absent, verifiers derive the algorithm from the key.

Verifiers MUST:

* Validate the algorithm against policy (reject weak algorithms)

* Ensure the key type is consistent with the derived algorithm

* Reject keys whose type does not match an acceptable algorithm

6.5. Signature-Key Integrity

The Signature-Key header SHOULD be included as a covered component in Signature-Input:

```
Signature-Input: sig=("@method" "@authority" "@path" "signature-key"); created=1732210000
```

If signature-key is not covered, an attacker can modify the header without invalidating the signature. Attacks include:

Scheme substitution: An attacker extracts the public key from an hwk scheme and republishes it via jwks_uri under their own identity, causing verifiers to attribute the request to the attacker.

Identity substitution: An attacker modifies the id parameter in a jwks_uri scheme to point to their own metadata endpoint that returns the same public key, impersonating a different signer.

Verifiers SHOULD reject requests where signature-key is not a covered component.

7. Privacy Considerations

7.1. Pseudonymity vs. Identity

The hwk and jkt-jwt schemes enable pseudonymous operation where the signer's identity is not disclosed. Verifiers should be aware that:

* A server can track a client across requests by JWK Thumbprint ([RFC7638]). If a client uses the same key across multiple servers, those servers could correlate the client's activity. Clients MUST use distinct keys for distinct servers to prevent cross-server correlation of pseudonymous identity.

* The jkt-jwt thumbprint is stable across sessions (tied to the enclave key), enabling long-term tracking even when ephemeral keys rotate.

* Verifiers should not log or retain pseudonymous keys beyond operational necessity.

The jwks_uri, x509, and jwt schemes reveal signer identity. When a client presents its identity via these schemes, the server learns the client's HTTPS URL or certificate subject, revealing which software is making the request. Servers SHOULD NOT disclose client identity information to third parties without the client operator's consent.

7.2. Key Discovery Tracking

The jwks_uri, jwt, and x509 schemes require verifiers to fetch resources from signer-controlled URLs. This creates tracking vectors:

* Signers can observe when and from where keys are fetched. In particular, when a server fetches a client's JWKS from jwks_uri at verification time, the fetch reveals to the JWKS host that someone is verifying signatures for that client.

* Verifiers should cache keys to minimize fetches.

* Verifiers may wish to use shared caching infrastructure to reduce fingerprinting.

7.3. JWT Contents

JWTs in the jwt scheme may contain additional claims beyond cnf. Verifiers should:

* Only process claims necessary for verification

* Not log or retain unnecessary JWT claims

* Be aware that JWT contents are visible to network observers unless using TLS

8. IANA Considerations

8.1. HTTP Field Name Registration

This document registers the following header fields in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" defined in [RFC9110].

Header field name: Signature-Key

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

Header field name: Signature-Error

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

8.2. Signature-Key Scheme Registry

This document establishes the "HTTP Signature-Key Scheme" registry. This registry allows for the definition of additional key distribution schemes beyond those defined in this document.

8.2.1. Registration Procedure

New scheme registrations require Specification Required per [RFC8126].

8.2.2. Initial Registry Contents

Scheme	Description	Reference
hwk	Header Web Key - inline public key	[this document]
jkt-jwt	JKT JWT Self-Issued Key Delegation - enclave-backed delegation	[this document]
jwks_uri	JWKS URI Discovery - key discovery via metadata	[this document]
jwt	JWT Confirmation Key - delegated key in JWT	[this document]
x509	X.509 Certificate - PKI certificate chain	[this document]

Table 4

8.2.3. Registration Template

Scheme Name: The token value used in the Signature-Key header
Description: A brief description of the scheme
Specification: Reference to the specification defining the scheme
Parameters: List of parameters defined for this scheme

8.3. HTTP Signature Metadata Parameters

This document registers the following parameter in the "HTTP Signature Metadata Parameters" registry established by [RFC9421], Section 6.3.

Parameter Name: sigkey

Status: standard

Specification document(s): [this document]

Description: Indicates the type of Signature-Key the server requires. Defined values: jkt (pseudonymous key identified by JWK Thumbprint), uri (key identified by a URI), x509 (X.509 certificate chain).

8.4. URN Sub-namespace Registration

This document registers the following URN sub-namespace in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry defined in [RFC3553].

Registry name: sig-error

Specification: [this document]

Repository: [this document], Section on Error Codes

Index value: Values are registered in the "Signature Error Code" registry defined in this document.

The URN pattern is urn:ietf:params:sig-error:<error-code>, where <error-code> corresponds to a value in the Signature Error Code registry. These URNs are used as Problem Details type values ([RFC9457]) in response bodies accompanying Signature-Error headers.

8.5. Signature Error Code Registry

This document establishes the "Signature Error Code" registry. New values may be registered following the Specification Required policy ([RFC8126]).

8.5.1. Initial Registry Contents

Value	Description	Reference
unsupported_algorithm	Signing algorithm not supported	[this document]
invalid_signature	Signature missing, malformed, or verification failed	[this document]
invalid_input	Missing required covered components	[this document]
invalid_request	Missing required info unrelated to signature	[this document]
invalid_key	Key cannot be parsed or doesn’t meet trust requirements	[this document]

unknown_key	Key not found at jwks_uri	[this document]
invalid_jwt	JWT malformed or signature verification failed	[this document]
expired_jwt	JWT expired	[this document]

Table 5

9. References

9.1. Normative References

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <https://www.rfc-editor.org/info/rfc3553>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8941] Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/info/rfc8941>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/info/rfc9457>.

9.2. Informative References

[I-D.ryan-httpauth-payment] Ryan, B., Moxey, J., Meagher, T., Weinstein, J., and S. Kaliski, "The "Payment" HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-ryan-httpauth-payment-01, 17 March 2026, <https://datatracker.ietf.org/doc/html/draft-ryan-httpauth-payment-01>.

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", November 2014, <https://openid.net/specs/openid-connect-discovery-1_0.html>.

[RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, DOI 10.17487/RFC7435, December 2014, <https://www.rfc-editor.org/info/rfc7435>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[x402] x402 Foundation, "x402: HTTP 402 Payment Protocol", 2025, <https://docs.x402.org>.

Appendix A. Document History

Note: This section is to be removed before publishing as an RFC.

A.1. draft-hardt-httpbis-signature-key-04

* Renamed spec from "HTTP Signature-Key Header" to "HTTP Signature Keys"
* Added sigkey parameter for Accept-Signature (RFC 9421 Section 5) with three values: jkt (pseudonymous), uri (URI-identified), x509 (PKI certificate)
* Added Signature-Error response header for structured signature verification error responses
* Added incremental adoption section describing zero-coordination deployment via 429/401/402 status codes
* Added privacy considerations for key thumbprint tracking, agent identity disclosure, and JWKS fetch side channel
* Registered sigkey in the HTTP Signature Metadata Parameters registry (RFC 9421 Section 6.3)
* Established Signature Error Code Registry

A.2. draft-hardt-httpbis-signature-key-03

* Added jkt-jwt scheme for self-issued key delegation
* Renamed well-known parameter to dwk (dot well-known)
* Added iss and dwk claims to jwt scheme (SHOULD) for issuer key discovery
* Added early validation step to jwt verification procedure (format, typ, exp checks before network fetches)
* Added TOFU reference (RFC 7435) to jkt-jwt scheme
* Added design rationale for jwks_uri vs inline JWKS
* Moved hwk string vs byte sequence design note to rationale appendix
* Reordered schemes
* Added acknowledgments

A.3. draft-hardt-httpbis-signature-key-02

* Changed x5t parameter to byte sequence per reviewer feedback
* Added structured field types to all parameters
* Added design note explaining string vs byte sequence choice for hwk

A.4. draft-hardt-httpbis-signature-key-01

* Initial public draft with four schemes: hwk, jwks_uri, x509, jwt

Appendix B. Design Rationale

B.1. Why jwks_uri Instead of Inline JWKS?

The jwks_uri and jwt schemes reference a jwks_uri property in the .well-known metadata document rather than embedding the JWKS directly in the metadata. This separation of concerns is deliberate:

1. *Independent key rotation*: Keys can be rotated by updating the JWKS endpoint without modifying the .well-known metadata document. This decouples key lifecycle management from configuration management, allowing operations teams to rotate keys on their own schedule without redeploying metadata.

2. *Independent management*: The .well-known metadata document and the JWKS can be hosted, managed, and secured by different systems or teams. For example, an identity team may manage keys while a platform team manages service metadata.

3. *Caching semantics*: The JWKS endpoint can have its own cache-control headers tuned for key rotation frequency (e.g., short TTLs during a rotation event), independent of the .well-known document’s caching policy.

4. *Consistency with existing standards*: This approach mirrors the pattern established by OpenID Connect Discovery [OpenID.Discovery] and OAuth Authorization Server Metadata [RFC8414], which both use jwks_uri in metadata documents for the same reasons.

B.2. Why a Separate Header?

An alternative design would extend Signature-Input with additional parameters to carry key material. This was considered and rejected for several reasons:

1. *Parameter complexity*: Each scheme has a different set of parameters (e.g., hwk needs kty, crv, x, y; jwks_uri needs id, dwk, kid; jwt needs a full JWT string). Overloading Signature-Input with all possible key parameters across all schemes would make the Signature-Input grammar unwieldy and harder to parse.

2. *Separation of concerns*: Signature-Input describes _what_ is signed and _how_ (covered components, algorithm, timestamps). Signature-Key describes _who_ signed it and _where to find the key_. These are distinct concerns, and separating them into distinct headers makes each easier to understand and process independently.

3. *Extensibility*: A separate header with a scheme registry allows new key distribution mechanisms to be added without modifying the Signature-Input grammar. New schemes can define arbitrary parameters without coordination with RFC 9421.

4. *Multiple signatures*: With a dictionary structure keyed by label, each signature can use a different scheme. This is natural in a separate header but would create complex nesting if embedded in Signature-Input.

B.3. Why Schemes Instead of Just a Key and Key ID?

A simpler design would define Signature-Key as carrying only a public key (or key reference) and a key identifier, without the scheme abstraction. This was considered insufficient because:

1. *Trust model varies*: A bare key tells the verifier nothing about the trust model. Is this a pseudonymous key to be evaluated on its own merits (hwk)? A key bound to a discoverable identity (jwks_uri)? A delegated key from an authority (jwt)? A certificate-backed key (x509)? The scheme token tells the verifier which verification procedure to follow and what trust properties the key carries.

2. *Verification procedure differs*: Each scheme has a fundamentally different verification path. hwk requires no external fetches. jwks_uri requires metadata discovery. x509 requires certificate chain validation. jwt requires JWT signature verification before the HTTP signature can be verified. A key-and-ID-only design would push scheme detection to heuristics or out-of-band agreement.

3. *Security properties differ*: Without an explicit scheme, a verifier cannot distinguish between a self-asserted key and a CA-certified key. The scheme makes the trust model explicit, allowing verifiers to enforce policy (e.g., "only accept jwt or x509 schemes").

4. *Interoperability*: Explicit schemes create clear interoperability targets. Two implementations that support the jwt scheme know exactly what to expect from each other. Without schemes, the same key material could be interpreted differently by different implementations.

B.4. Why Strings Instead of Byte Sequences for hwk?

The hwk parameters use structured field strings rather than byte sequences. JWK key values are base64url-encoded per [RFC7517], while structured field byte sequences use base64 encoding per [RFC8941]. Using strings allows implementations to pass JWK values directly without converting between base64url and base64, avoiding a potential source of encoding bugs.

Appendix C. Acknowledgments

The author would like to thank Yaron Sheffer for their feedback on this specification.

Authors' Addresses

Dick Hardt
Hell
Email: dick.hardt@gmail.com

Thibault Meunier
Cloudflare
Email: ot-ietf@thibault.uk