HTTPBIS
Internet-Draft
Intended status: Standards Track
Expires: 23 September 2026

A. Ferro
ApertoID

22 March 2026

ApertoID-Signature: HTTP Request Signing for AI Agent Identity
draft-ferro-httpbis-apertoid-sig-00

Abstract

This document defines the ApertoID-Signature HTTP header field, which enables AI agents to cryptographically prove their identity on each HTTP request. The agent signs the request method, target URL, body hash, and identity metadata using an Ed25519 private key whose corresponding public key is published in DNS via the ApertoID protocol [APERTOID-DNS]. The mechanism provides request-level identity verification, action binding (the signature is tied to the specific method and URL), and replay protection via timestamps and nonces.

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 September 2026.

Copyright Notice

Table of Contents

1. Introduction

The ApertoID protocol [APERTOID-DNS] enables domain owners to declare authorized AI agents in DNS, including publishing Ed25519 public keys for agent identity verification. However, publishing a key in DNS only establishes which key belongs to which agent — it does not prove that a particular HTTP request was made by the holder of that key, nor does it bind the signature to the specific action being performed.

This document defines the ApertoID-Signature HTTP header field, which closes both gaps. When an agent makes an HTTP request (e.g., to an MCP server, an API, or any HTTP service), it includes this header containing an Ed25519 signature over the request method, target URL, body hash, and identity metadata. The receiving service can then verify the signature against the public key published in the agent's ApertoID DNS record, confirming both that the request originates from the authorized agent AND that the signature applies to this specific request — not a different endpoint, not a different method, not a different body.

This mechanism is analogous to DKIM signatures for email: DKIM key records are published in DNS, and DKIM signatures are attached to email messages. Similarly, ApertoID key records are published in DNS (per [APERTOID-DNS]), and ApertoID-Signature headers are attached to HTTP requests (per this document).

1.1. Relationship to HTTP Message Signatures

HTTP Message Signatures [RFC9421] provides a general-purpose framework for signing HTTP messages. ApertoID-Signature does not use RFC 9421 for the following reasons:

* RFC 9421 requires structured headers (RFC 8941) support, component identifiers, algorithm negotiation, and signature metadata — all of which add implementation complexity that is unnecessary for the single-purpose case of agent identity verification with a fixed algorithm.

* ApertoID-Signature uses DNS-based key discovery (the public key is in the agent's DNS TXT record), which does not map to RFC 9421's key resolution model.

* ApertoID-Signature is designed to be implementable as a drop-in middleware in under 100 lines of code in any language, without requiring an RFC 9421 library.

However, ApertoID-Signature follows RFC 9421's principle of binding signatures to specific request components. The signing input includes the HTTP method and request target (path + query), ensuring that a signature is valid only for the specific action it was created for.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. The ApertoID-Signature Header Field

2.1. Header Syntax

The ApertoID-Signature header field contains semicolon-separated tag-value pairs. The formal grammar uses ABNF [RFC5234]:

```
ApertoID-Signature: d=example.com; s=leadhunter;
  t=1711100000; n=a1b2c3d4e5f6;
  sig=<base64-ed25519-signature-88chars>
```

2.2. ABNF Definition

```
apertoid-sig-hdr = "ApertoID-Signature" ":" OWS sig-value OWS
sig-value       = domain-tag ";" SP selector-tag ";"
                  SP timestamp-tag ";" SP nonce-tag ";"
                  SP signature-tag
domain-tag      = "d=" domain-name
selector-tag    = "s=" selector
timestamp-tag   = "t=" 1*DIGIT
nonce-tag       = "n=" 1*16HEXDIG
signature-tag   = "sig=" base64url

domain-name     = label *("." label)
label           = ALPHA *(ALPHA / DIGIT / "-")
selector        = ALPHA *(ALPHA / DIGIT / "-")
base64url       = 1*( ALPHA / DIGIT / "+" / "/" / "=" )
OWS             = *( SP / HTAB )
```

2.3. Header Tags

d (REQUIRED) The domain the agent claims to represent. The verifier uses this to locate the ApertoID Policy Record at "_apertoid.<d>".

s (REQUIRED) The agent selector. Combined with the domain, this identifies the Agent Declaration Record at "<s>._apertoid.<d>" where the public key is published.

t (REQUIRED) The signature timestamp as a Unix timestamp (seconds

since 1970-01-01T00:00:00Z). MUST be within the validity window (default: 300 seconds / 5 minutes) of the verifier's current time. Requests with timestamps outside this window MUST be rejected.

n (REQUIRED) A nonce: a unique, non-repeating value for this request, encoded as 1-16 hexadecimal characters (lowercase). The nonce provides replay protection within the timestamp validity window. Verifiers MUST maintain a nonce cache for the duration of the validity window and MUST reject requests with previously seen nonces.

sig (REQUIRED) The Ed25519 signature over the signing input (Section 3.1), encoded as unpadded Base64 per [RFC4648] Section 4 (88 characters for 64 bytes).

3. Signing Procedure

3.1. Signing Input Construction

The signing input is a byte string constructed by concatenating the following components, each terminated by a newline character (0x0A):

```
signing_input = d_value   LF
                s_value   LF
                t_value   LF
                n_value   LF
                method    LF
                target    LF
                body_hash LF
```

Where:

d_value The value of the d= tag (the domain name, lowercase).

s_value The value of the s= tag (the selector, lowercase).

t_value The decimal string representation of the t= tag (the timestamp).

n_value The value of the n= tag (the nonce, lowercase hex).

method The HTTP request method, uppercase (e.g., "GET", "POST", "DELETE"). This binds the signature to the specific HTTP action. A signature created for a POST request MUST NOT be valid for a GET or DELETE request.

target The request target as sent in the HTTP request line: the path

and query string, without the scheme, host, or fragment (e.g., "/mcp/tools/search?limit=10"). If there is no query string, only the path is included (e.g., "/mcp/tools/search"). This binds the signature to the specific endpoint. A signature created for /mcp/search MUST NOT be valid for /mcp/delete.

body_hash The lowercase hexadecimal SHA-256 hash of the raw HTTP request body. This binds the signature to the specific request content. If the request has no body (e.g., GET, HEAD, DELETE without body), the SHA-256 hash of the empty string MUST be used:

e3b0c44298fc1c149afbf4c8996fb924
27ae41e4649b934ca495991b7852b855

All components MUST be encoded as UTF-8. The signing input MUST be deterministic: the same input parameters MUST always produce the same signing input byte string.

3.2. Producing the Signature

The agent produces the signature as follows:

1. Determine the request method (e.g., "POST") and request target (e.g., "/mcp/tools/search").
2. Compute the SHA-256 hash of the request body (or the empty-body hash for bodyless requests).
3. Generate a unique nonce (RECOMMENDED: 8-16 random hex characters).
4. Record the current Unix timestamp.
5. Construct the signing input as defined in Section 3.1.
6. Sign the signing input using the agent’s Ed25519 private key per [RFC8032], producing a 64-byte signature.
7. Encode the signature as unpadded Base64 per [RFC4648] Section 4.
8. Construct the ApertoID-Signature header with all required tags.
9. Attach the header to the outgoing HTTP request.

3.3. Example

An agent "leadhunter" acting for "example.com" sends:

```
POST /mcp/tools/search HTTP/1.1
Host: api.target.com
Content-Type: application/json

{"query": "find leads in tech sector", "limit": 10}
```

The signing input (each line terminated by LF):

```
example.com
leadhunter
1711100000
a1b2c3d4e5f6
POST
/mcp/tools/search
7d5e4a8b... (SHA-256 of the JSON body)
```

The agent signs this input with its Ed25519 private key and attaches:

```
ApertoID-Signature: d=example.com; s=leadhunter;
  t=1711100000; n=a1b2c3d4e5f6;
  sig=MEUCIQDx4f... (88 base64 characters)
```

4. Verification Procedure

Services that have deployed ApertoID SHOULD inspect incoming HTTP requests for the ApertoID-Signature header. If the header is present, the service SHOULD verify it per this specification. If the header is absent but the agent's domain publishes an ApertoID policy with "p=reject", the service MAY reject the unsigned request.

When a service receives a request with an ApertoID-Signature header, it performs the following verification:

```
VERIFY_APERTOID_SIGNATURE(request):

  1. Extract ApertoID-Signature header from request
  2. Parse d=, s=, t=, n=, sig= tags
     If any required tag is missing: Return "malformed"
  3. Check timestamp t= is within validity window:
     If |current_time - t| > 300: Return "timestamp_invalid"
  4. Check nonce n= against nonce cache:
     If n= is in cache: Return "nonce_reused"
     Add n= to cache with expiry = t + 300
  5. Perform DNS verification per [APERTOID-DNS]:
     Query "_apertoid.<d>" for policy record
     Query "<s>._apertoid.<d>" for agent declaration
     Extract pk= (public key) and check exp=
  6. If DNS verification fails:
     Apply policy p= from policy record
     Return DNS verification result
  7. Reconstruct signing_input from:
     d, s, t, n,
     request.method (uppercase),
     request.target (path + query),
     SHA-256(request.body)
  8. Verify Ed25519 signature sig= against signing_input
     using public key pk= from DNS record
  9. If signature is invalid:
     Apply policy p= from policy record
     Return "sig_invalid"
  10. Return "pass"
```

4.1. Result Values

pass The signature is valid, the agent is authorized, and the signature matches the specific request method, target, and body.

malformed The ApertoID-Signature header is present but cannot be parsed.

timestamp_invalid The timestamp is outside the validity window.

nonce_reused The nonce was already seen within the validity window.

sig_invalid The Ed25519 signature does not match the signing input and public key.

DNS-level results (none, revoked, expired, url_mismatch, key_mismatch, permerror, temperror) are as defined in [APERTOID-DNS].

5. Replay Protection

ApertoID-Signature provides three layers of replay protection:

Timestamp window Signatures are valid for at most 300 seconds (5 minutes). Requests with timestamps outside this window are rejected. This limits the useful lifetime of any intercepted signature.

Nonce uniqueness Within the timestamp window, each nonce may be used only once. Verifiers MUST maintain a nonce cache and reject duplicate nonces. The cache need only retain entries for the duration of the validity window; older entries can be safely evicted.

Action binding The signing input includes the HTTP method and request target. A valid signature for "POST /mcp/search" cannot be replayed against "DELETE /mcp/data" or "POST /mcp/export" — even within the timestamp window and with a fresh nonce, because the signing input would differ.

Verifiers SHOULD use a validity window of 300 seconds (5 minutes). Shorter windows reduce the replay surface but increase sensitivity to clock skew. Verifiers MAY allow configuration of the validity window within the range of 60 to 600 seconds.

6. Security Considerations

6.1. Action Binding Scope

The signing input includes the HTTP method and request target (path + query), preventing cross-endpoint and cross-method replay attacks. However, it does not include the Host header or scheme. This means a valid signature could theoretically be replayed against a different host serving the same path, if the attacker can redirect the request. In practice, this is mitigated by TLS: the agent establishes a TLS connection to a specific host, and the signature is only transmitted over that connection. Services MUST require HTTPS per [RFC9110]; HTTP connections MUST NOT be used with ApertoID-Signature.

6.2. HTTP Headers Not Signed

HTTP headers (other than the request method and target) are not included in the signing input. This means headers such as Content-Type, Authorization, and custom headers can be modified by an intermediary without invalidating the signature. The rationale is that ApertoID-Signature authenticates agent identity and binds it to a specific action and payload — it is not a general-purpose message

integrity mechanism. TLS provides full message integrity in transit. Services requiring header integrity beyond what TLS provides SHOULD use HTTP Message Signatures [RFC9421] in addition to ApertoID-Signature.

6.3. Clock Synchronization

The timestamp-based validity window requires that agents and verifiers maintain reasonably synchronized clocks. Agents and verifiers SHOULD use NTP [RFC5905] or equivalent time synchronization. Clock skew greater than the validity window will cause all requests to fail verification.

6.4. Nonce Cache Requirements

Verifiers MUST maintain a nonce cache for the duration of the timestamp validity window. The cache MUST be shared across all verification instances if the service runs multiple processes or nodes. Failure to maintain a shared nonce cache allows replay attacks across processes. For services running on a single node, an in-memory cache is sufficient. For distributed services, a shared cache (e.g., Redis, Memcached) is RECOMMENDED.

6.5. Private Key Protection

The agent’s Ed25519 private key MUST be protected with the same care as any other signing key. It SHOULD be stored in a hardware security module (HSM), trusted platform module (TPM), or at minimum in encrypted storage with appropriate access controls. If the private key is compromised, the domain owner MUST immediately revoke the agent’s DNS record per [APERTOID-DNS].

6.6. Signature Stripping

An attacker who can intercept and modify HTTP requests could strip the ApertoID-Signature header entirely, causing the request to appear unsigned. Verifiers SHOULD query the agent’s ApertoID policy record to determine whether the domain expects signed requests. If the policy specifies "p=reject", the verifier SHOULD reject unsigned requests from agents claiming to represent that domain.

7. Privacy Considerations

The ApertoID-Signature header reveals the agent's domain (d=) and selector (s=) to the receiving service and to any intermediary that can observe HTTP headers. This is by design — the purpose of the header is to declare agent identity. However, domain owners should be aware that the same d= and s= values appear on all requests from the same agent, creating a correlation identifier that enables request tracking across time and endpoints.

Services that observe ApertoID-Signature headers learn which domains are using AI agents and which specific agents are making requests. This information is inherent to the protocol's purpose and cannot be mitigated without defeating the protocol's goals. Domain owners who wish to limit correlation SHOULD rotate selectors periodically, though this requires publishing new DNS records.

8. IANA Considerations

8.1. HTTP Header Field Registration

This document requests registration of the following HTTP header field in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at <https://www.iana.org/assignments/http-fields>:

Field Name: ApertoID-Signature

Status: permanent

Structured Type: N/A

Reference: [this document]

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8032] Josefsson, S., "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC9110] Fielding, R., "HTTP Semantics", RFC 9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[APERTOID-DNS] Ferro, A., "ApertoID: DNS-Based Agent Identity Declaration Protocol", Work in Progress, Internet-Draft, draft-ferro-dnsop-apertoid-00, March 2026, <https://datatracker.ietf.org/doc/html/draft-ferro-dnsop-apertoid-00>.

9.2. Informative References

[RFC5905] Mills, D., "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC6376] Crocker, D., "DomainKeys Identified Mail (DKIM) Signatures", RFC 6376, September 2011, <https://www.rfc-editor.org/info/rfc6376>.

[RFC9421] Backman, A., "HTTP Message Signatures", RFC 9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

Appendix A. Full Request/Response Example

```
   === Agent sends signed POST request ===

   POST /mcp/tools/search HTTP/1.1
   Host: api.targetservice.com
   Content-Type: application/json
   ApertoID-Signature: d=example.com; s=leadhunter;
     t=1711100000; n=a1b2c3d4e5f6;
     sig=MEUCIQDx4fakebase64signaturehere...88chars

   {"query": "find leads in tech sector", "limit": 10}

   === Signing input that was signed ===

   example.com\n
   leadhunter\n
   1711100000\n
   a1b2c3d4e5f6\n
   POST\n
   /mcp/tools/search\n
   <sha256-hex-of-body>\n

   === Verifier checks ===

   1. Parse header: d=example.com, s=leadhunter
   2. Timestamp 1711100000 within 300s of now: OK
   3. Nonce a1b2c3d4e5f6 not in cache: OK, cache it
   4. DNS: _apertoid.example.com -> policy p=reject
   5. DNS: leadhunter._apertoid.example.com -> pk=MCow...
   6. exp= not passed: OK
   7. Reconstruct signing_input with method=POST,
      target=/mcp/tools/search, body_hash=sha256(body)
   8. Ed25519 verify sig against signing_input with pk: OK
   9. Result: pass

   === Same signature replayed to DELETE endpoint ===

   DELETE /mcp/data/all HTTP/1.1
   ApertoID-Signature: d=example.com; s=leadhunter;
     t=1711100000; n=a1b2c3d4e5f6;
     sig=MEUCIQDx4f... (same signature)

   Verification FAILS at step 8:
     signing_input includes "DELETE" and "/mcp/data/all"
     which differs from original "POST" and "/mcp/tools/search"
     -> Ed25519 verify FAILS -> Result: sig_invalid
```

Appendix B. Implementation Guidance

This appendix is non-normative.

To maximize adoption, implementations SHOULD provide middleware or decorator patterns that require minimal code changes.

```
# Python: Agent side - sign outgoing requests
import hashlib, time, secrets, base64
from nacl.signing import SigningKey

def sign_request(method, url_path, body, domain, selector, key):
    t = str(int(time.time()))
    n = secrets.token_hex(8)
    body_hash = hashlib.sha256(body).hexdigest()
    signing_input = f"{domain}\n{selector}\n{t}\n{n}\n"
    signing_input += f"{method}\n{url_path}\n{body_hash}\n"
    sig = key.sign(signing_input.encode()).signature
    sig_b64 = base64.b64encode(sig).decode().rstrip("=")
    return {
        "ApertoID-Signature":
            f"d={domain}; s={selector}; t={t}; n={n}; sig={sig_b64}"
    }

# Python: Verifier side - verify incoming requests
def verify_request(request):
    header = request.headers.get("ApertoID-Signature")
    if not header:
        return "unsigned"
    tags = parse_tags(header)  # extract d, s, t, n, sig
    # ... check timestamp, nonce, DNS lookup, then:
    body_hash = hashlib.sha256(request.body).hexdigest()
    signing_input = (
        f"{tags['d']}\n{tags['s']}\n{tags['t']}\n{tags['n']}\n"
        f"{request.method}\n{request.path}\n{body_hash}\n"
    )
    pubkey = get_apertoid_pubkey(tags['d'], tags['s'])  # DNS
    return verify_ed25519(pubkey, signing_input, tags['sig'])
```

Reference implementations in Python, Go, and JavaScript are maintained at https://github.com/ApertoID.

Acknowledgements

The signing mechanism in this document was inspired by the DKIM signature scheme [RFC6376]. The principle of binding signatures to specific request components follows the approach established by HTTP Message Signatures [RFC9421], adapted for the single-purpose case of AI agent identity verification with DNS-based key discovery.

Author’s Address

Andrea Ferro
ApertoID
Verona
Italy
Email: irn@irn3.com
URI: https://github.com/ApertoID