TBD
Internet-Draft
Intended status: Informational
Expires: 7 November 2026

D. Hardt
Hell

6 May 2026

# AAuth Bootstrap Guidance
## draft-hardt-aauth-bootstrap-01

## Abstract

This document provides informational guidance for agent providers (APs) on enrolling agents and issuing AAuth agent tokens defined in [I-D.hardt-oauth-aauth-protocol]. It covers per-platform key handling, optional platform attestation, agent identifier strategies, and refresh patterns. The mechanisms described here are not normative protocol — they are common patterns that interoperable AP implementations can adopt or adapt.

## Discussion Venues

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

Discussion of this document takes place on GitHub at https://github.com/dickhardt/AAuth (https://github.com/dickhardt/AAuth). Issues, comments, and pull requests are welcome there. Source for this draft is in the same repository.

## Status of This Memo

Table of Contents

## 1. Introduction

The AAuth Protocol [I-D.hardt-oauth-aauth-protocol] establishes that every agent has its own cryptographic identity — an agent identifier of the form aauth:local@domain, bound to a signing key, and attested by an agent token issued by an agent provider (AP). The protocol defines the agent token format and how agents present that identity to person servers (PSes), resources, and access servers (ASes). It does not specify how an agent comes to hold an agent token in the first place. That step is *bootstrap*, and it is the subject of this document.

### 1.1. What Bootstrapping Is

Bootstrapping is the AP-side ceremony by which an instance of an agent acquires an agent token. The agent generates a signing key on the device or in the browser where it will run, presents whatever evidence the AP requires (a signed-in account, an attested device, a published JWKS, etc.), and receives an agent token whose cnf.jwk is bound to that key and whose sub is an aauth:local@domain identifier the AP has chosen.

After bootstrap the agent can participate in AAuth: it can sign HTTP messages per [I-D.hardt-httpbis-signature-key], identify itself at resources, and present its agent token to a PS so the user can bind the agent to themselves on first interaction.

### 1.2. What Bootstrapping Is Not

* *Not normative protocol.* This document is informational. The AAuth Protocol does not mandate a specific bootstrap ceremony, and conformance does not depend on the patterns described here. APs are free to use other approaches that produce a valid agent token.
* *Not the user-to-agent binding.* Binding an agent to a person is performed by the PS, lazily, on the agent's first interaction with the PS per the AAuth Protocol. Bootstrap produces an agent identity; the PS attaches that identity to a user.
* *Not authorization.* Bootstrap conveys no scope, no resource permission, and no user identity claims. Those are obtained through the flows defined in the AAuth Protocol after bootstrap.
* *Not one-size-fits-all.* Web, mobile, and self-hosted agents have different threat models and different platform primitives available to them. This document offers patterns appropriate to each, not a single prescribed ceremony.

### 1.3. Patterns Covered

* *Per-platform key handling* — where the agent's signing key lives and how strongly it is protected on web, mobile, and self-hosted deployments; desktop and workload coverage is TBD Section 4.
* *Optional platform attestation* — when and why to require WebAuthn, App Attest, or Play Integrity Section 5.
* *Agent identifier strategies* — how to construct the sub claim's local part Section 6.
* *Refresh patterns* — issuing fresh agent tokens for renewal Section 8.

Throughout, when this document refers to "the durable key" and "the ephemeral key" it means the keys defined in Section 4. The ephemeral key's public part appears in agent_token.cnf.jwk and signs HTTP messages from the agent per [I-D.hardt-httpbis-signature-key]; the durable key serves as the AP's stable enrollment anchor and signs only at refresh. APs that use a single durable key for all signatures Section 4 can read references to "the ephemeral key" as referring to that same durable key.

## 2. Conventions and Definitions

{::boilerplate bcp14-tagged}

This document is informational guidance and does not itself impose normative requirements. Normative requirements relevant to bootstrap are defined in [I-D.hardt-oauth-aauth-protocol]; this document references them where helpful but uses lowercase "should" / "must" in its own descriptive prose.

## 3. Terminology

Terms defined in [I-D.hardt-oauth-aauth-protocol] are used here with the same meaning. In particular:

* *Agent Provider (AP)* — issues agent tokens.
* *Agent token* — JWT signed by the AP, carrying iss, sub, cnf.jwk, optionally ps, and other claims.
* *Person Server (PS)* — represents the person; binds agents to a person on first interaction.

This document additionally uses:

* *Durable key* — a signing key whose lifetime is intended to span the agent install (typically the lifetime of an install or browser-storage entry). The durable key is the AP's stable enrollment anchor; it is presented only to the AP at refresh and is not used to sign requests to PSes, resources, or ASes.
* *Ephemeral key* — a signing key generated fresh per agent-token issuance. Its public part appears in agent_token.cnf.jwk. The agent uses it to sign HTTP messages for the agent token's lifetime, then discards it on the next refresh.
* *Platform attestation* — a mechanism by which the runtime platform attests to properties of the agent or its key (WebAuthn, Apple App Attest, Google Play Integrity, etc.).

## 4. Per-Platform Key Handling

On web, mobile, and desktop, APs should use a two-key pattern: a *durable key* that serves as the AP's stable enrollment anchor and is presented only to the AP at refresh, plus an *ephemeral key* generated fresh per agent-token issuance whose public part appears in agent_token.cnf.jwk and which signs HTTP messages for the agent token's lifetime. Refresh chains the new ephemeral key to the durable key via the jkt-jwt scheme [I-D.hardt-httpbis-signature-key]; see Section 8.

This pattern bounds the blast radius of an ephemeral-key leak to one agent token's lifetime, narrows the durable key's attack surface to the AP refresh path (it never signs requests to PSes, resources, or ASes), and accommodates hardware-backed durable keys on platforms that have them today and on platforms that may expose them in the future without protocol change. APs may use a single durable key for all signatures where simplicity outweighs these properties — receivers cannot distinguish the two patterns, since they only verify cnf.jwk against the HTTP signature.

Self-hosted agents Section 4.3 use a single key — the JWKS-published key serves as both the AP signing key and the agent's signing key, since there is no separate AP to refresh against.

### 4.1. Web Apps

The durable key is a non-extractable [WebCryptoAPI] key generated with extractable: false and stored in IndexedDB scoped to the AP's origin. The ephemeral key is also a non-extractable WebCrypto key, regenerated on each refresh and discarded when the next refresh produces its replacement.

Properties of the durable key:

* The private key cannot be read or exported by JavaScript, including by code injected via XSS or malicious browser extensions. JS can only ask the browser to sign with it.
* The key is bound to the origin's IndexedDB storage. Clearing site data destroys it; a new enrollment is required.
* No user-verification gesture is required to sign — operations are fast enough for routine signature use.

Both keys are software-protected (browser sandbox) rather than hardware-protected. The two-key pattern still applies: the durable key signs only the periodic refresh (once per agent-token lifetime), while the ephemeral key signs every HTTP request and rotates on each refresh. APs that want the additional assurance of a WebAuthn ceremony at enrollment time can layer one on top; see Section 5. If browsers later expose hardware-backed credentials suitable for use as the durable key, the pattern accommodates them with no protocol change.

### 4.2. Mobile (iOS and Android)

The durable key is generated and stored in the platform's hardware-backed keystore: the Secure Enclave on iOS or StrongBox on Android (or the Android Keystore where StrongBox is unavailable). The ephemeral key is a software key in app memory, regenerated on each refresh.

Properties of the durable key:

* The private key cannot be exported from the keystore. Cryptographic operations are performed by the keystore on the application's behalf.
* The key is bound to the application install. Reinstalling the app generates a new key and requires re-enrollment.

* The keystore can additionally enforce user authentication (biometric or device passcode) before sign operations, if the AP wants user-presence on each refresh.

The AP typically also requires an attestation ceremony at enrollment to confirm the durable key is real keystore-bound material rather than software-generated. See Section 5. Because the durable key signs only at refresh, any per-op cost (keystore round-trip, optional user verification) is incurred at most once per agent-token lifetime, not on every request.

### 4.3. Self-Hosted Agents

A self-hosted agent runs under a domain the user controls. The agent publishes its AP metadata document at /.well-known/aauth-agent.json per [I-D.hardt-oauth-aauth-protocol]; the JWKS itself is hosted at any HTTPS URL referenced by the metadata's jwks_uri. The corresponding private key should be hardware-bound where the platform supports it: macOS Keychain (Secure Enclave on supported hardware), Windows TPM, or Linux Secret Service.

Self-hosted agents act as their own AP — they self-issue agent tokens signed by the JWKS-published key. There is no separate AP to refresh against, so the two-key pattern does not apply: the JWKS-published key serves both as the AP signing key (signing self-issued agent tokens) and as the key whose public part appears in agent_token.cnf.jwk (signing HTTP messages). Because the trust anchor is a key the user controls and publishes, no platform attestation step exists. Other parties verify the agent token signature against the published JWKS, exactly as they would for any other AP.

### 4.4. Desktop Apps

TBD. Future revisions will cover key handling for native desktop applications, where the durable key would live in a hardware-backed store (macOS Keychain with Secure Enclave on supported hardware, Windows TPM via CNG, Linux Secret Service / TPM2) and the ephemeral key in process memory, following the same pattern as mobile.

### 4.5. Workload

TBD. Future revisions will cover headless workload identity (e.g., SPIFFE/SPIRE SVIDs, WIMSE workload identity, cloud-platform IMDS attestation), where the trust anchor is platform attestation rather than user interaction.

## 5. Optional Platform Attestation

Platform attestation gives the AP cryptographic evidence about the runtime context in which the durable key was generated. It is optional — the AAuth Protocol does not require it. APs choose whether to require attestation based on their threat model.

Common reasons an AP might require attestation:

* *Anti-fraud at enrollment.* Distinguishing real user devices from server-side automation.
* *Hardware-binding evidence.* Confirming the durable key is in a Secure Enclave or StrongBox rather than software.
* *App-integrity evidence.* Confirming the agent is the AP's published app, not a modified or repackaged binary.
* *User-presence evidence.* Confirming a real user gesture authorized this enrollment.

Common reasons an AP might not require attestation:

* The AP serves a trust posture where AP-side fraud detection is not signal-driven (e.g., paid accounts, invitation-only enrollment).
* The AP wants the broadest possible reach, including environments where attestation is unavailable.
* The deployment is self-hosted, where the user is the trust anchor.

### 5.1. WebAuthn (Web Apps)

[WebAuthn] provides user-verification and a hardware-rooted credential on web. APs that require it perform a registration ceremony at enrollment and an assertion ceremony at later sensitive operations. The credential is stored in the user's authenticator (TPM, Secure Enclave, security key, or platform syncing fabric).

Tradeoffs:

* Provides user-verification (touch, face, PIN) the WebCrypto-only approach lacks.
* Provides hardware-protected key material that cannot be lifted even by a fully compromised browser process.
* UX failure modes exist around cross-Chrome-profile use, embedded webviews, syncing-fabric inconsistencies, and cross-device QR ceremonies. APs that adopt WebAuthn should plan for fallback paths when these fail.

### 5.2. App Attest (iOS)

[AppAttest] attests to two things: that the durable key was generated in the device's Secure Enclave, and that the request originates from the AP's published app on a genuine Apple device. The AP receives an attestation object at enrollment which it verifies against Apple's attestation root, then accepts subsequent assertions signed by the same enclave key.

### 5.3. Play Integrity (Android)

[PlayIntegrity] provides a device, app, and account integrity verdict signed by Google. The AP nominates a nonce, the agent invokes the Play Integrity API, and the AP verifies the resulting integrity token. Subsequent assertions can be signed by an Android Keystore (or StrongBox) key that the AP enrolled at the same time.

### 5.4. When to Require Attestation

A practical rule of thumb:

* *Consumer-grade APs*: optional. Most consumer agent providers do not require attestation; the protocol-level binding at the PS is the security gate.
* *Regulated or high-assurance APs*: usually required. Financial services, healthcare, enterprise SSO providers benefit from attestation as part of their AML/KYC or device-management posture.
* *Multi-tenant APs*: often required at higher tiers. An AP may issue weak agent tokens to free tier users and require attestation for paid or enterprise tier users, surfacing the difference to receivers via AP-defined claims in the agent token.

## 6. Agent Identifier Strategies

The agent token's sub is an aauth:local@domain identifier. The domain part is the AP. The local part identifies the agent install at the AP — it must be stable for the lifetime of the install so PSes and other parties can recognize a returning agent.

APs are free to choose any opaque scheme for the local part: a random string assigned at enrollment, a deterministic derivation from the durable key's thumbprint, a sequential identifier, or a human-readable handle. When deriving from a thumbprint, use the durable key's thumbprint — the ephemeral key rotates on each refresh and is not a stable identifier. Receivers treat the identifier as opaque.

### 6.1. Per-Install Identity

Each install's durable key is the basis for one agent identity. A returning user on a new device is a new agent. This keeps the AP minimal — it has no user-account system, no (user, durable_jkt) mappings, and no ability to correlate a single user's activity across their devices.

Multi-device users will see multiple agent entries in their PS dashboard. Grouping or merging those entries belongs at the PS, which already authenticates the user and is the correct layer for cross-device correlation. Rotation of the durable key produces a new agent identity; rotation of the ephemeral key (on every refresh) does not — the agent's sub is stable across ephemeral rotations. PS-side regrouping is the recovery path for durable-key changes.

## 7. Example Agent Token Claims

A typical agent token issued by an AP, illustrating the claims described in [I-D.hardt-oauth-aauth-protocol] and the identifier strategies in Section 6.

JWT header:

```
{ "alg": "EdDSA", "typ": "aa-agent+jwt", "kid": "..." }
```

JWT payload:

```
{
  "iss":  "https://ap.example",
  "dwk":  "aauth-agent.json",
  "sub":  "aauth:k7q3p9n2@ap.example",
  "ps":   "https://ps.example",
  "cnf":  { "jwk": { "kty": "OKP", "crv": "Ed25519", "x": "..." } },
  "iat":  1746316800,
  "exp":  1746320400,
  "jti":  "..."
}
```

## 8. Refresh Patterns

Agent token lifetime is the AP's policy re-evaluation cadence — every refresh is the AP's chance to re-check device posture, attestation freshness, and account status before issuing a new token. A typical lifetime is *1 hour*, matching common practice for proof-of-possession-bound access tokens. APs may use shorter lifetimes (e.g., 515 minutes) for higher-assurance deployments where attestation must be refreshed often, or longer lifetimes up to the AAuth Protocol's

24-hour ceiling for low-policy-churn deployments where refresh chattiness is undesirable.

### 8.1. Two-Key Refresh

On web, mobile, and desktop, refresh chains the new ephemeral key to the durable key via the jkt-jwt scheme [I-D.hardt-httpbis-signature-key]:

1. The agent generates a fresh ephemeral key pair.
2. The agent constructs a JWT signed by the *durable key*, naming the new ephemeral public key. This is the "naming JWT" carried in the Signature-Key header under scheme=jkt-jwt.
3. The agent signs the refresh request with the *ephemeral key* under [RFC9421] HTTP Message Signatures.
4. The AP verifies the durable-key signature on the naming JWT, looks up the enrollment by the durable key's thumbprint, verifies the HTTP signature against the ephemeral public key, applies its policy (device posture, attestation freshness, account status), and returns a new agent token whose cnf.jwk is the ephemeral public key.
5. The agent uses the new agent token and ephemeral key for the agent token's lifetime, then discards the ephemeral key on the next refresh.

Example refresh request:

```
POST /refresh HTTP/1.1
Host: ap.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1746316800
Signature: sig=:...ephemeral-key signature bytes...:
Signature-Key: sig=jkt-jwt;jwt="eyJhbGc..."

{}
```

The jwt parameter value is a JWT signed by the durable key with payload including the ephemeral public key (typically as cnf.jwk) and a jti for replay protection. The HTTP signature is produced by the ephemeral key. The AP correlates the two keys via the naming JWT's payload.

### 8.2. Single-Key Refresh

APs that opt for the single-durable pattern Section 4 sign the refresh request directly with the durable key under the hwk scheme [I-D.hardt-httpbis-signature-key]. The AP verifies the signature, looks up the enrollment by the key's thumbprint, and issues a fresh agent token with a new exp. The same cnf.jwk is carried through; the agent's key is unchanged.

### 8.3. Mobile Refresh Specifics

APs that required platform attestation at enrollment typically do not re-attest on every refresh — the durable-key signature on the naming JWT (or the durable-key HTTP signature in the single-key pattern) is sufficient proof that the same enclave-resident key is making the request. APs that want periodic re-attestation can require a fresh App Attest assertion or Play Integrity verdict on a schedule (e.g., every 30 days) by including a server nonce in the refresh challenge.

### 8.4. Self-Hosted Refresh

Self-hosted agents self-issue agent tokens. There is no separate refresh ceremony — the agent generates a new agent token signed by its JWKS-published key whenever needed. The two-key pattern does not apply Section 4.3.

### 8.5. Key Rotation vs Token Refresh

Refresh issues a new agent token bound to a fresh ephemeral key (or, in the single-key pattern, to the same durable key). *Durable key rotation* generates a new durable key and is a separate, rare event. Under the per-install identity model Section 6.1, a new durable key is a new agent — the PS treats it as new on first interaction, and any cross-device or cross-rotation continuity is handled at the PS by the user.

## 9. Per-Platform Enrollment Sketches

This section sketches a typical end-to-end enrollment for each platform. The sketches are illustrative; APs are free to vary them.

### 9.1. Web App Enrollment

1. User logs into the AP through the AP's normal login.
2. Agent (running in the AP's web origin) generates a non-extractable WebCrypto Ed25519 *durable* key and stores its handle in IndexedDB.

3. Agent posts the durable public key to an AP-internal enrollment endpoint, signed by the new key (hwk scheme).
4. AP optionally performs a WebAuthn registration and verifies it.
5. AP records (ap_user, durable_jkt) and is now ready to issue agent tokens.
6. When the agent needs an agent token directed at PS_X, it generates a fresh *ephemeral* WebCrypto key and calls an AP-internal token-issuance endpoint indicating ps=PS_X, signed via jkt-jwt chaining the durable key to the ephemeral key Section 8. The AP returns an agent token with sub derived per the AP's identifier strategy Section 6 (using the durable key's thumbprint when derivation is used), ps = PS_X, cnf.jwk = the ephemeral public key, and any AP-attested claims.

### 9.2. Mobile App Enrollment

1. User signs into the AP through the app's normal login.
2. App generates a *durable* key in the Secure Enclave (iOS) or StrongBox (Android).
3. App initiates platform attestation: App Attest on iOS, Play Integrity on Android. The AP nominates a nonce.
4. App posts the durable public key, the attestation result, and the nonce to an AP-internal enrollment endpoint.
5. AP verifies the attestation against the platform's trust root.
6. AP records (ap_user, durable_jkt, attestation) and is ready to issue agent tokens.
7. Token issuance proceeds as in the web app sketch — app generates a fresh ephemeral key per agent token and chains it to the durable key via jkt-jwt.

### 9.3. Self-Hosted Enrollment

1. User generates a hardware-bound key on their machine.
2. User publishes an AP metadata document at /.well-known/aauth-agent.json per [I-D.hardt-oauth-aauth-protocol], with jwks_uri pointing to a JWKS containing the public part of that key.
3. The agent self-issues an agent token signed by that key as needed.

There is no separate enrollment step — publication of the JWKS is the enrollment.

## 10. Security Considerations

### 10.1. Trust in the AP

Every AP-attested claim in the agent token is only as trustworthy as the AP that signed the token. Receivers should apply policy proportional to their trust in the AP. An unfamiliar AP making strong attestation claims may warrant additional caution at the PS consent screen.

### 10.2. Ephemeral Key Compromise

An ephemeral-key leak — via memory disclosure, in-page attacker, side channel, or similar — exposes only the signatures the agent makes during the current agent token's lifetime. At the recommended 1-hour lifetime, the blast radius is bounded to roughly that window before natural expiry forces replacement. Agents that detect compromise can decline to refresh, aging out the ephemeral key without explicit revocation. This bounding is the primary security argument for the two-key pattern Section 4.

### 10.3. Durable Key Compromise

Compromise of the durable key compromises the install's agent identity for the durable key's lifetime. The durable key signs only at refresh and is presented only to the AP, so its attack surface is much narrower than the ephemeral key's — but a successful compromise lets the attacker mint refresh requests indefinitely until the AP revokes the enrollment. APs should detect anomalous refresh patterns and provide a way for users to revoke a durable enrollment.

A non-extractable WebCrypto durable key cannot be exfiltrated by page-level attackers, but it can be used by them while they hold execution in the page. APs should pair WebCrypto-only enrollment with normal web hygiene (CSP, subresource integrity, dependency review) and should not treat the non-extractable property as a substitute for keeping the page's JavaScript clean. A hardware-backed durable key (Secure Enclave, StrongBox, TPM) cannot be exfiltrated at all, only used in-place — narrowing the threat to malicious code running in the agent application itself.

### 10.4. Attestation Replay

Platform attestation results (App Attest, Play Integrity, WebAuthn ceremonies) should be bound to a server-nominated nonce that is single-use and short-lived (5 minutes is a reasonable upper bound). The AP should verify that the attestation includes the nonce it issued; without this binding, a captured attestation can be replayed across enrollments.

### 10.5. Self-Hosted JWKS Key Compromise

Compromise of the self-hosted JWKS key allows the attacker to mint agent tokens for that user's domain. Users running self-hosted agents should use hardware-backed keys (Secure Enclave / TPM / StrongBox) and rotate the published JWKS if compromise is suspected.

## 11. Privacy Considerations

### 11.1. Identifier Stability and User Tracking

An agent's sub is the same value at every PS the agent contacts, not a per-PS pairwise identifier. A stable sub lets each PS reliably re-identify the agent across sessions — that is the intended property — but it also means colluding PSes (or any party with cross-PS telemetry) can correlate the agent's activity across them. Under per-install identity Section 6.1, durable-key rotation produces a new sub, giving users a natural "fresh start" capability.

### 11.2. AP Visibility Into Agent Activity

The AP that issued an agent token does not see the agent's subsequent traffic to PSes, resources, or ASes (they verify against the AP's published JWKS, not by calling the AP). The AP's view is limited to enrollment and refresh requests. APs should document their data retention practices for those events.

## 12. IANA Considerations

This document is informational and registers no new media types, JWT claim names, or metadata fields.

## 13. Implementation Status

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

This section records the status of known implementations of the patterns described in this document at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs.

TBD

## 14. Document History

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

* draft-hardt-aauth-bootstrap-01
  - Major rewrite. The document is now informational guidance for AP implementers. The previously-normative PS bootstrap protocol (PS /bootstrap endpoint, bootstrap_token, bootstrap announcement, agent server [now Agent Provider] bootstrap_endpoint / refresh_endpoint / webauthn_endpoint) has been removed. PS-side binding to a person now happens lazily on the agent's first interaction with the PS per the AAuth Protocol; the bootstrap document covers AP-side enrollment patterns only.
  - Removed Agent-Attested Display Values section; the platform and device parameters are defined and described in the AAuth Protocol.
* draft-hardt-aauth-bootstrap-00
  - Initial draft.

## 15. Acknowledgments

TBD.

## 16. References

### 16.1. Normative References

[I-D.hardt-httpbis-signature-key]
Hardt, D., "HTTP Signature Keys", 2026, <https://datatracker.ietf.org/doc/draft-hardt-httpbis-signature-key>.

[I-D.hardt-oauth-aauth-protocol]
Hardt, D., "AAuth Protocol", 2026, <https://github.com/dickhardt/AAuth>.

[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>.

### 16.2. Informative References

[AppAttest]
Apple, "Establishing your app’s integrity (App Attest)", 2024, <https://developer.apple.com/documentation/devicecheck/establishing-your-app-s-integrity>.

[PlayIntegrity]
Google, "Play Integrity API", 2024, <https://developer.android.com/google/play/integrity/overview>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[WebAuthn] W3C, "Web Authentication: An API for accessing Public Key Credentials - Level 3", 2024, <https://www.w3.org/TR/webauthn-3/>.

[WebCryptoAPI]
W3C, "Web Cryptography API", 2017, <https://www.w3.org/TR/WebCryptoAPI/>.

## Author’s Address

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