Obsigil Mandate-Token Format

The obsigil project

version 1.0 draft

Contents

Abstract

Obsigil is a mandate-token format: a credential split into a public, advisory *manifest* and a secret-sealed, authoritative *mandate*, joined into one compact text string. Each half is a deterministically sealed ciphertext; the manifest is sealed under a published key, so a front end can read its *claims* without a secret, while the mandate is sealed under a secret key that both mints and verifies its *clauses*. Verification is symmetric, so obsigil serves the shared-secret use cases of JWT and JWE — a time-bounded, audience-scoped bearer credential — without a signature, a JOSE header, or a JSON wire form. This document defines the wire format, the two registered algorithms, the reserved fields, and a cross-language API conformance profile.

1 Introduction

A mandate token is a JWT-like token split into a public **manifest** and an encrypted **mandate**. Each half is a deterministically sealed ciphertext under an authenticated encryption algorithm — AES-SIV (RFC 5297 [7]) or AES-GCM-SIV (RFC 8452 [6]) — rendered as `b64` or `hex` text. The two halves are joined by a separator that names the shared text encoding (`.` for `b64`, `~` for `hex`), with a single-character algorithm code on each side naming that half's cipher (`0` for AES-SIV, `1` for AES-GCM-SIV). The manifest carries public *claims* a front end reads; because it is sealed under a published key, anyone can open *or forge* a manifest, so a reader MUST NOT trust its claims (§16.7). The mandate carries sealed *clauses* the back-end enforces. Verification is symmetric — the key that mints a mandate also verifies it — so obsigil fits shared-secret (HS256-style) JWT and JWE deployments, not public-key verification.

This document is normative. §4 through §8 define the wire format: the token grammar, how each half is constructed and sealed, the algorithm registry, the canonical CBOR (Concise Binary Object Representation) serialization, and the reserved fields. §9 gives the front-end / back-end reader split and §16 the security model an implementation MUST honor; §11 covers versioning and §12 a cross-language API conformance profile. §13 fixes wire conformance against published test vectors and §15 registers the media type. The design rationale — why the format is shaped this way — is recorded separately in RATIONALE.md and is not needed to build a conformant implementation.

Conventions

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in BCP 14 (RFC 2119 [2], RFC 8174 [12]) when, and only when, they appear in all capitals. Grammar rules are given in ABNF (RFC 5234 [3]).

2 Terminology

- **claim** — a field carried in the manifest. *Asserted*: the manifest is keyless, so anyone can state a claim and a reader MUST NOT trust it.
- **clause** — a field carried in the mandate. *Sealed*: the mandate is authenticated, so a clause is binding within the trust domain.
- **AEAD** — authenticated encryption with associated data: a cipher providing confidentiality *and* integrity. Every algorithm obsigil registers (§6) is an AEAD.
- **keyless** — encryption under a fixed, *publicly known* key. Because the key is public — published, and shipped in every front-end build — keyless encryption provides neither confidentiality (anyone reads it) nor authentication (anyone forges it). It detects accidental, non-adversarial corruption, like a checksum; it is *not* a security layer. The public value it uses — the manifest key — is pinned in §5.2.
- **deterministic** — for a fixed key and algorithm, the same plaintext always produces the same ciphertext, with no random nonce. Obsigil seals both halves deterministically

(§5); the mandate's required, unique `tid` (§8) keeps every mandate plaintext distinct, so determinism leaks no plaintext equality (§16.4).

- **SIV** — AES-SIV (RFC 5297), a misuse-resistant deterministic AEAD; obsigil algorithm code `0`.
- **GCM-SIV** — AES-GCM-SIV (RFC 8452), a nonce-misuse-resistant AEAD; obsigil algorithm code `1`.
- **algorithm code** — the single character (`0`–`9` or `a`–`z`) naming a half's encryption algorithm (`0`=AES-SIV, `1`=AES-GCM-SIV), written in the clear immediately adjacent to the separator on that half's side (§4, §6).
- **NumericDate** — seconds since the Unix epoch (as in JWT).
- **serialization** — the data format of a half's fields. Obsigil fixes this: every half is a canonical CBOR map (RFC 8949 [1], §4.2), which obsigil encodes itself from the supplied field values (§7). Not a per-half choice; carried by no in-band tag.
- **text encoding** — the text encoding of a half's ciphertext: `b64` or `hex`. Selected for the *whole token* by the separator (§4); not sealed.
- **b64** — the URL-safe base64 encoding (RFC 4648 [10], §5; alphabet `A-Za-z0-9-_`) with *no* padding. One of the two text encodings, selected by the `.` separator (§4).
- **hex** — lowercase base16 (RFC 4648 [10], §8). RFC 4648's base16 alphabet is uppercase; obsigil narrows it to lowercase (`0-9a-f`) so that case-folding a `hex` token is well-defined. The other text encoding, selected by the `~` separator (§4).
- **separator** — the single character joining the two halves. It both delimits them and names the token's text encoding: `.` selects `b64`, `~` selects `hex`.

3 Dependencies

Obsigil is built on standard authenticated-encryption primitives and a text encoding. It depends normatively on:

- AES-SIV (RFC 5297 [7]) and, optionally, AES-GCM-SIV (RFC 8452 [6]) — the deterministic AEADs named by the algorithm codes (§6);
- CBOR (RFC 8949 [1]) — the serialization of both halves' fields, in its deterministic encoding (RFC 8949 §4.2, §7);
- the text encodings (RFC 4648 [10]) — `b64` (URL-safe base64, §5, no padding) and `hex` (lowercase base16, §8);
- UUIDv7 (version 7 of the Universally Unique IDentifier, RFC 9562 [4]) — the form of the mandate's required `tid` (§8);
- HKDF (the HMAC-based Extract-and-Expand Key Derivation Function) with HMAC-SHA-256 (RFC 5869 [11]) — the key derivation for algorithm code `1` (§6.1);
- the media-type registration procedures (RFC 6838 [5], BCP 13) — used by the registration in §15;
- ABNF (RFC 5234 [3]) — the grammar notation of §4.

Obsigil's per-half byte layout is specified in full by this document: §5 (construction), §6 (the algorithm codes and the key derivation each one uses), and §5.2 (the public manifest key). The format depends on no particular library. Any two conformant implementations MUST produce byte-identical tokens from identical field values — canonically CBOR-encoded per §7 — under the same key, algorithm, and encoding; the published, language-agnostic test vectors (§13) — not any single implementation — are the reference against which that byte-level conformance is measured.

This document also depends normatively on BCP 14 (RFC 2119, RFC 8174) for the requirement keywords defined in the Conventions above. JWT (JSON Web Token, RFC 7519), CWT (CBOR Web Token, RFC 8392), and COSE (CBOR Object Signing and Encryption, RFC 9052) are referenced *informatively* only: obsigil borrows JWT's registered-claim vocabulary and CBOR's integer-key idiom but is none of them (see §14).

4 Token structure

A token is the two halves joined with a single separator, each present half carrying its algorithm code against the separator:

```
token         = manifest-part SEP mandate-part
manifest-part = [ manifest ALG ]        ; ciphertext, then its code
mandate-part  = [ ALG mandate ]         ; code, then ciphertext
SEP           = "." / "~"               ; "." => b64, "~" => hex
ALG           = %x30-39 / %x61-7A       ; one char 0-9 / a-z (see registry)
```

Each half is a deterministic AEAD ciphertext, already compact text (§5); the grammar gives no production for `manifest` or `mandate` because each is an opaque ciphertext body, fixed by §5, not by syntax. The separator does double duty: it delimits the two halves *and* names the *text encoding* both halves use — `.` for `b64`, `~` for `hex`. Immediately adjacent to the separator, on the side of each *present* half, sits that half's single-character algorithm code (§6): `0` for AES-SIV, `1` for AES-GCM-SIV. *Exactly one separator is always present, and it is the only separator character in a well-formed v1 token.* Either half MAY be empty; an empty half is *absent* and carries no algorithm code. Three shapes are therefore well-formed (shown with `.` and both halves AES-SIV; `~` and code `1` behave identically):

- `manifest0.0mandate` — the full token.
- `manifest0.` — **manifest-only**: advisory claims, no enforceable content (see §16.8).
- `.0mandate` — **mandate-only**: no front-end claims; also the form the front end forwards to the backend (see §9).

The `0` in each shape is the adjacent half's algorithm code, not part of the ciphertext. A parser MUST split a token on its single separator; from a non-empty manifest part it MUST read the *last* character as the algorithm code and the prefix as the manifest ciphertext, and from a non-empty mandate part the *first* character as the algorithm code and the suffix as the mandate ciphertext. Because the code is read positionally — exactly one character against the separator — the split is unambiguous even though those characters (`0`–`9`, `a`–`z`) also occur inside the `b64` and `hex` alphabets. A parser MUST reject a token that: contains zero separator characters, more than one, or a separator outside {`.`, `~`}; carries a positional algorithm-code character outside the ALG set (`0`–`9`, `a`–`z`), or names an algorithm code it

does not implement (§6); has a present half that is a lone algorithm code with empty ciphertext; or has both halves absent (a bare separator) — rather than guess a split. The trailing-versus-leading separator makes the two degenerate shapes structurally distinct, so a parser never confuses a manifest-only token (`manifest0.`) with a mandate that lost its half (`.0mandate`).

The separator invariant is normative. An obsigil text encoding's alphabet MUST exclude both separator characters, so a separator never occurs inside a half and the split is unambiguous before any decoding. Both defined encodings satisfy this: `b64` (URL-safe `A-Za-z0-9-_`, no padding) and `hex` (`0-9a-f`) contain neither `.` nor `~`, and — unlike JWT — neither carries `=` padding. An obsigil implementation MUST NOT pair the separator set with an encoding whose alphabet includes `.` or `~`. The algorithm code needs no such exclusion: it is identified by position, not by being outside the alphabet.

Both encodings are canonical and strict, matching the unpadded forms used throughout: `b64` is URL-safe (`A-Za-z0-9-_`) with no padding, and `hex` is lowercase (`0-9a-f`) of even length. A decoder MUST reject non-canonical input — `=` padding, whitespace, any out-of-alphabet character, a `b64` symbol whose unused trailing bits are non-zero, a `b64` string whose length is 1 modulo 4, or an odd-length `hex` string. Because `hex` case-folds losslessly — unlike `b64`, where case is significant — a deployment MAY lowercase a received *hex* token (separator `~`) before decoding to tolerate case-mangling transport; it MUST NOT lowercase a `b64` token. A producer MUST emit lowercase `hex`.

Because the separator and the algorithm codes *select* the decoding alphabet and the decryption scheme rather than being sealed, altering either in transit is not an attack. For the mandate, sealed under a secret key (§5.1), an attacker cannot re-seal under a different code or encoding, so a half re-read under the wrong alphabet, or decrypted under the wrong algorithm, either fails to decode or fails authentication, which obsigil rejects uniformly (§16.6); the mandate's cleartext signals are thus integrity-protected without being part of the sealed payload. The manifest is keyless (§5.2), so an attacker *can* re-seal it under any code or encoding and its cleartext signals carry no such protection — harmless only because the manifest is non-authoritative (§16.7): a forged or relabeled manifest is already within the attacker's reach and governs nothing.

5 Construction

The two halves are produced *independently*. Each half's fields are encoded as a canonical CBOR map (§7) and sealed with the deterministic AEAD named by its algorithm code; the ciphertext is then text-encoded per the token's separator (§4).

5.1 The mandate

The private clauses are encoded as a canonical CBOR map and sealed under a secret key with the deterministic AEAD named by the mandate's algorithm code (AES-SIV, code `0`, by default). Only the backend holds the key. The mandate MUST carry a unique `tid` (§8), which obsigil generates by default (§8.2); because every mandate plaintext is therefore distinct, deterministic sealing leaks no plaintext equality (§16.4).

The mandate key MUST be 64 bytes that are uniformly random or computationally indistinguishable from uniform: either drawn directly from a cryptographically secure generator, or derived from a high-entropy secret (at least 256 bits) through an approved KDF — for example HKDF (RFC 5869 [11]) — emitting a full 64-byte output. It MUST NOT be derived from a password or other low-entropy secret except through an approved password-based KDF (for example Argon2id, scrypt, or PBKDF2 with parameters meeting current guidance) configured to emit a full 64-byte output, and MUST be distinct from the public manifest key (§5.2). A KDF-derived master is then used exactly as a generator-produced one — the construction consumes those 64 bytes directly (§6.1), so a short or low-quality KDF output would weaken every algorithm's key. A backend's set of candidate mandate keys MUST NOT include the public manifest key — it is published in this specification, so accepting it as a mandate key would let anyone mint valid mandates. A mandate key MUST be used only for obsigil v1 mandates with the semantics defined here; it MUST NOT be shared with another protocol or with a future obsigil version whose semantics differ, since obsigil binds no version into the ciphertext (§11).

5.2 The manifest

The public claims are encoded as a canonical CBOR map and sealed with the keyless method (§2): the deterministic AEAD named by the manifest's algorithm code (AES-SIV, code `0`, by default), under the fixed, public **manifest key** this specification pins below. The manifest MUST be keyless — it is definitionally the public half. Because the manifest key is public — published here, and shipped in every front-end build — anyone can open the manifest *and* anyone can forge one: it is an encoding wrapper, not a security layer, providing neither confidentiality nor origin authentication. A *keyed* manifest would be confidential and unforgeable — different semantics, and unopenable by the public front-end build — and is therefore not a conformant obsigil manifest; an implementation MUST seal the manifest keyless.

The manifest key is the 64-byte value below — a public constant published by this specification. Every implementation MUST use this exact value (128 hex digits, the two rows concatenated):

```
manifest key -- 64 bytes (hex):
381284633d02ea5f35df8596b5cc4218310060468e8b465455a415174ea6e966
a9f48eec4ba446ddfc8b78587895356f45a75a1ab7419454dd9f7aa8a95dbdd5
```

This 64-byte value is a master key. How each algorithm draws its key material from a 64-byte master — the manifest key here, or the secret mandate key (§5.1) — is specified once, per code, in §6. The single 64-byte manifest key is the only published constant; every algorithm's key derives from it, not from a second constant.

A 64-byte key published in a specification would, for any *secret*, be catastrophic — it is public knowledge. For the manifest that is exactly the point: a manifest guards no secret and is meant to be opened and forged by anyone (§16.7), so a universally known key is the right tool, not a misuse.

5.3 Algorithm, serialization, and encoding

Three facets describe each half. Two vary per token and obsigil names each in the clear; the third is fixed by this specification:

- *algorithm* (AES-SIV / AES-GCM-SIV) — the single-character algorithm code in the clear, immediately adjacent to the separator (§4, §6). Obsigil names the algorithm itself; it does *not* rely on the cipher output being self-describing.
- *serialization* (canonical CBOR) — *fixed*: every half is a canonical CBOR map (§7). It is not a per-half choice and carries no in-band tag.
- *text encoding* (`b64` / `hex`) — named in the clear by the separator (§4), and therefore shared by both halves of a token.

Of obsigil's own format choices only two are carried as stored data, both in the clear next to the join: the per-half algorithm code and the token-wide encoding separator. The serialization is a fixed constant of this specification, so nothing names it; nothing else about the format is stored either.

6 Algorithm registry

An algorithm code is a single character (`0`–`9`, `a`–`z`) naming the AEAD that seals a half. It is read positionally against the separator (§4), so it needs no exclusion from the encoding alphabets. Obsigil v1 registers two:

code	algorithm	reference
`0`	AES-SIV	RFC 5297
`1`	AES-GCM-SIV	RFC 8452

Every registered algorithm MUST be an authenticated AEAD; a confidentiality-only or non-authenticated one MUST NOT be registered, so a valid algorithm code always denotes authenticated encryption — the property the mandate relies on (§16.2). Obsigil uses every registered algorithm *deterministically* (§16.4).

So that any conformant party can open any token, every obsigil implementation MUST support code `0` (AES-SIV) for both halves: the mandatory-to-implement default. Code `1` (AES-GCM-SIV) is OPTIONAL; a token using it interoperates only between implementations that both compile it. A verifier presented with an algorithm code it does not implement MUST reject the token under the uniform failure of §16.6.

Later obsigil versions MAY register further codes (`2`–`9`, then `a`–`z`, thirty-four in all); each MUST likewise be an authenticated AEAD. This registry is internal to the obsigil specification — a closed set this and later versions of this document define, extended only by revising the document, not through any external registration procedure; the sole IANA registration obsigil makes is the media type of §15.

6.1 Key material

Both registered algorithms key from a single 64-byte master — the secret mandate key (§5.1) or the public manifest key (§5.2); the derivation is identical for either.

— *Code `0` (AES-SIV).* The full 64-byte master is the AES-SIV key (RFC 5297), used with no derivation. AES-SIV splits it into two 256-bit subkeys: bytes 0–31 as the S2V (CMAC) authentication key and bytes 32–63 as the CTR-mode encryption key — i.e. AES-256-SIV. An implementation built on a combined-key AES-SIV API MUST pass the 64 bytes in that order; one built on lower-level primitives MUST reproduce the same subkey split.

— *Code `1` (AES-GCM-SIV).* The 32-byte AES-256-GCM-SIV key-generating key is derived as

```
key = HKDF-Expand(PRK = master, info = "gcmsiv", L = 32)
```

using HKDF with HMAC-SHA-256 (RFC 5869), where `master` is the 64 decoded master bytes (not any hex form) and `info` is the six ASCII bytes `gcmsiv`. The `HKDF-Extract` step MUST be omitted: the 64-byte master is already uniformly random and serves directly as the pseudorandom key. An implementation MUST compute `HKDF-Expand` only and MUST NOT prepend an `HKDF-Extract` step or substitute an Extract-then-Expand call — a single `HKDF` invocation with a null or empty salt is *not* equivalent and yields different bytes.

6.2 Sealing parameters and output layout

Obsigil uses both algorithms *deterministically* (§16.4): no random nonce, and no associated data. A half's plaintext is the canonical CBOR encoding of its fields (§7); the AEAD encrypts that plaintext and the result is laid out as below, then text-encoded per the separator (§4).

— *Code `0` (AES-SIV).* Invoked with a *zero-element* associated-data vector — S2V is called with no associated-data components, *not* with one empty component, which would differ — and no external nonce; the 16-byte synthetic IV is the S2V output. A sealed half is

```
synthetic-IV (16 bytes) || encrypted plaintext
```

— *Code `1` (AES-GCM-SIV).* Invoked with a fixed all-zero 12-byte nonce and zero-length AAD; the 16-byte authentication tag is appended and the fixed nonce is *not* emitted. A sealed half is

```
encrypted plaintext || authentication tag (16 bytes)
```

The fixed nonce is sound here because the mandate's unique `tid` (§16.4) keeps every plaintext distinct, so the only thing a repeated nonce could leak — plaintext equality — never arises. Reusing one nonce for every message does, however, forgo AES-GCM-SIV's per-nonce key separation: all messages under a key share the same nonce-derived encryption and authentication keys, so the construction reverts to a deterministic AEAD whose safety is bounded by the birthday limit on its 128-bit synthetic IV rather than by fresh per-nonce keys. RFC 8452's message-limit tables (§9) are parameterized by a bounded nonce-reuse count and do not extend to a single nonce reused across every message, so they do not bound this single-fixed-nonce regime; the per-key message ceiling stated below applies.

Both registered algorithms seed encryption from a 128-bit synthetic IV — the AES-SIV S2V output (§6.1) and, for AES-GCM-SIV, the authentication tag — so each carries a birthday bound near 2^{64} distinct mandates per key, independent of the 256-bit cipher key: beyond

it two mandates' synthetic IVs collide with non-negligible probability, weakening the deterministic guarantee (§16.4) for the colliding pair. An implementation SHOULD keep the number of distinct mandates sealed under one mandate key well below that bound — staying under roughly 2^{32} per key holds the collision probability near 2^{-64} — for AES-SIV, the mandatory-to-implement default, as much as for AES-GCM-SIV.
The two cleartext signals — the algorithm code and the encoding separator — are *not* fed as associated data; for the secret-keyed mandate their integrity is the fail-closed property of §4 (a wrong code or encoding fails to decode or to authenticate), not an AAD binding — the keyless manifest gains no such protection but needs none (§16.7). A verifier MUST reject a half whose decoded length is below 17 bytes — the AEAD's 16-byte floor (synthetic IV or tag) plus at least one byte of CBOR plaintext (the shortest being the empty map, `0xa0`), the least a sealed half can occupy — and, per §8, a half whose authenticated plaintext omits a required field; all fail uniformly (§16.6).

7 Serialization

A half's fields are serialized as a single **canonical CBOR** map (RFC 8949 [1]). The serialization is fixed: obsigil offers no per-half format choice and carries no in-band format tag. The sealed plaintext of a half is exactly that map's encoded bytes:

```
plaintext = canonical-CBOR(map of fields)
```

Obsigil owns this encoding. A producer supplies field *values*; obsigil validates them (§8) and emits the canonical CBOR itself before sealing. Canonical here is the core deterministic encoding of RFC 8949 §4.2, applied recursively to every nested item: definite-length items only; the shortest (preferred) serialization of every integer, length, float, and simple value; and map keys sorted by their encoded bytes. Array element order is significant — canonical CBOR orders map keys but never array elements — so an array field's elements are sealed in exactly the order supplied, and two producers obtain identical bytes only when they agree on that order (the test vectors pin it; see §8.4 for `aud`). A floating-point value — which only an application field may hold, as every reserved field is an integer, byte string, text string, or array of text (§8) — additionally takes the shortest of the IEEE 754 half, single, and double widths (subnormals included) that round-trips it exactly. `NaN` is forbidden: it has no single canonical bit pattern across encoders, so a producer MUST NOT emit one and a verifier MUST reject a half that carries one. Because obsigil produces the bytes, two producers that seal the same field values under the same key, algorithm, and encoding obtain byte-identical tokens with no further coordination — determinism is *field-level* for everything obsigil encodes (§16.4). This is the sole point at which obsigil acts as a serializer rather than sealing producer-supplied octets, and it does so only for the fixed CBOR envelope.

Map keys are CBOR integers or text strings, split into two namespaces by sign:

— *reserved keys* — the entire negative-integer space, reserved to obsigil. The reserved fields of §8 take small negative keys (`tid` at `-1`, and so on), single-byte in CBOR through `-24`. Obsigil reads, validates, and enforces these.

— *application keys* — non-negative integers and text strings, for whatever a producer adds for its own use. The non-negative single-byte range `0`–`23` is entirely the application's; obsigil never inspects, places, or enforces these (§8).

The sign *is* the namespace: a verifier reads it from the CBOR major type as it walks the map, with no threshold to configure. This fixes the unknown-key policy (§16.10): an unrecognized *negative* key lies in obsigil's namespace and MUST be rejected — a verifier cannot honor a reserved meaning it does not implement — while an unrecognized non-negative key or text string is opaque application data and is ignored.

A well-formed half uses only these two key types, at every depth. A verifier MUST reject a mandate any of whose CBOR maps — the top-level map or a map nested inside an application value — carries a key that is neither a CBOR integer nor a text string (a byte string, a float, a tag, a boolean or other CBOR simple value, or a compound array or map key), under the uniform failure of §16.6: such a key falls outside both namespaces and has no defined meaning. A manifest carrying such a key is handled as any malformed manifest is (§16.7). The restriction binds map *keys*; an application field's *value* is otherwise unconstrained by type — an application value MAY be any canonical CBOR data item whose maps are themselves so keyed, and structured data with non-text keys travels instead as the byte-string value of an application key (below). That obsigil "never inspects" an application value means it applies no semantic or type policy to it, *not* that it leaves the bytes as supplied: the canonical encoding of this section applies *recursively* to the whole half, so every nested map's keys are sorted by their encoded bytes and every nested integer, length, float, and simple value takes its shortest form, with `NaN` forbidden at every depth.

Because every half obsigil decodes is CBOR — a pure data format whose standard decoder neither evaluates its payload as code nor constructs arbitrary host objects — opening a half is never a code-execution vector, not even for the keyless, attacker-forgeable manifest (§16.7). The capability hazard a free-form serialization would pose is thus foreclosed by construction rather than by a rule an implementation must police. An application that needs a foreign serialization (say a protobuf message it already models) MAY carry it as the byte-string value of an application key; obsigil treats that value as opaque bytes and never decodes it. That value's determinism is the producer's responsibility, and since a decoder for it runs in application code — on authenticated mandate bytes, but *authenticated* is not *safe to evaluate* — the producer SHOULD give it a pure-data decoder too. (For the formats CBOR displaced as an envelope serialization, and why, see RATIONALE.md.)

> *Why one fixed format.* The halves serve disjoint consumers (§9), yet neither needs a format *choice*: both readers open their half through an obsigil library — the front end already needs one to decrypt the keyless manifest — so a single ubiquitous binary format costs neither consumer a native parser, while giving both field-level determinism and a compact integer-keyed encoding. Fixing the format is also what lets obsigil own the canonical encoding, turning byte-identical minting from a "share a serializer" burden on producers into an automatic property.

8 Reserved fields

A field inside a half is a *claim* in the manifest and a *clause* in the mandate. A claim is *asserted*: the manifest is keyless, so anyone can state one and a reader MUST NOT trust it. A clause is *sealed*: the mandate is authenticated, so its contents are binding within the trust domain. The same datum changes status with the half that carries it — `exp` is a binding clause in the mandate and, optionally, an advisory claim in the manifest.

Obsigil reserves a small set of field names with fixed, security-relevant meaning, borrowing JWT's registered-claim vocabulary but giving each a single concrete rule rather than JWT's application-specific interpretation. Names obsigil does not reserve are **opaque application data**: obsigil never inspects, places, or enforces them, in either half.

Placement follows one rule: a field the backend enforces lives in the mandate, as a clause; a field the front end displays lives in the manifest, as a claim; a field both need appears in both, independently, with the mandate's clause authoritative.

A presence requirement is *conditional on the half being present* (§4): an absent half carries no fields and imposes no requirement, but a half that *is* present MUST carry the field marked required for it.

field	key	half	status	meaning
`tid`	`-1`	mandate	required	unique token id (UUIDv7)
`exp`	`-2`	mandate	required	authoritative expiry
`exp`	`-2`	manifest	optional	advisory refresh hint
`aud`	`-3`	mandate	optional (see below)	intended verifiers
`sub`	`-4`	mandate	optional	subject authorized
`iss`	`-5`	manifest	required	issuer, for display
`iss`	`-5`	mandate	optional	issuer, for audit

`exp` is a NumericDate — a CBOR integer counting seconds since the Unix epoch. A verifier MAY allow a small configured leeway for clock skew.

Required means different things by half, matching authority. A missing *mandate* `tid` or `exp` is a hard failure: the verifier rejects the token uniformly (§16.6, §8.3, §8.2). A missing *manifest* `iss` is soft: the front end treats the manifest as malformed and ignores its claims but MUST NOT block or fail enforcement (§8.6), since the manifest is advisory (§16.7). The table marks both *required*; the half decides which enforcement applies.

8.1 Field keys and the reserved namespace

Each field is a map entry whose key is a CBOR integer or text string (§7). The reserved fields above take the negative integer keys in the table; the entire negative-integer space is obsigil's, and these are the only assignments obsigil v1 makes. A later version MAY assign further reserved fields, always at negative keys. Everything else — non-negative integer keys and text-string keys — is opaque application data the producer controls and obsigil ignores.

The sign of the key decides namespace and unknown-key handling, with no configured boundary: a verifier MUST reject a half carrying a negative key it does not recognize (fail closed, §16.6), and MUST ignore a non-negative or text-string key it does not recognize. Because canonical CBOR forbids duplicate map keys, a half whose encoding repeats any key — reserved or application — is non-canonical and MUST be rejected (§16.10); there is no first-wins / last-wins ambiguity to resolve.

A reserved key is defined only for the half the table of §8 assigns it to. A reserved key carried in a half that does not define it cannot be given its reserved meaning there, and MUST be treated exactly as an unrecognized reserved key in that half: a verifier MUST reject such a mandate (uniform failure, §16.6), and a front end MUST treat such a manifest as

malformed and ignore all of its claims (§8.6) — the soft handling the advisory manifest receives for any defect (§16.7). Under the v1 table this can arise only in the manifest, which defines only `exp` and `iss`, so a `tid`, `aud`, or `sub` there is out of place; the mandate defines every reserved field, and a later version that confines a new field to one half inherits the same rule. The stray field is never honored.

Reserved fields are wire-encoded by key, but an implementation MAY surface them to callers by their conventional names (`tid`, `exp`, `aud`, `sub`, `iss`); the name and the key denote the same field (§12).

8.2 `tid`

A unique identifier for the token, and the linchpin of deterministic sealing (§16.4). A mandate, when present, MUST carry a `tid`, and it MUST be a UUIDv7 (RFC 9562) unique per mandate key. It MUST be encoded as the 16-byte binary UUIDv7 — a CBOR byte string of length 16, never the 36-character text form. UUIDv7's 48-bit millisecond timestamp and per-millisecond randomness make that uniqueness structural rather than a bare promise; a verifier MUST reject a mandate whose `tid` is absent or not a well-formed UUIDv7 (uniform failure, §16.6). A *well-formed* UUIDv7 here is a 16-byte value whose 4-bit version field — the high nibble of byte 6 — is `0x7` and whose 2-bit variant field — the top two bits of byte 8 — is `0b10` (RFC 9562); its 48-bit big-endian millisecond timestamp occupies bytes 0–5. A verifier MUST reject a `tid` that is not 16 bytes or whose version or variant field differs, and MAY additionally reject one whose embedded timestamp lies implausibly far in the future. That last rejection is a local hardening policy, not a well-formedness rule: the same token can be accepted by one verifier and rejected by another, so it lies outside the interoperability contract (§13) and is never encoded in a known-answer vector.

By default obsigil *generates* the `tid` when minting a mandate: a fresh UUIDv7 whose 48-bit timestamp comes from the current clock and whose 74 random bits come from a cryptographically secure generator. Generation makes the required uniqueness hold by construction — within a millisecond the 74 random bits collide only with negligible probability — so an issuer cannot accidentally repeat a `tid`. A producer MAY instead supply its own `tid` (to bind the token to an external identifier, or when an application clause must reference it); a supplied `tid` MUST be a well-formed UUIDv7 and the producer then owns its uniqueness. Either way the verifier validates only well-formedness; uniqueness it cannot check (§16.4). Because generation reads a clock and a generator, minting is not a pure function of its field inputs — the deterministic, vector-pinned function is sealing a *given* mandate, `tid` included (§13).

Beyond uniqueness, `tid` does double duty as the issue clock: its 48-bit big-endian Unix-millisecond field *is* the mandate's issue-time, so obsigil defines no separate `iat` clause — a consumer that needs issuance time derives it from the `tid` (read the 48-bit field; for NumericDate semantics, floor to seconds). This supports max-age policies and revocation by epoch (reject any `tid` whose embedded time predates T); and a deployment MAY record spent `tid`s in a denylist or single-use store to detect replay. Because obsigil is symmetric (§16.1), any holder of the mandate key sets the embedded timestamp freely, so revocation by epoch bounds honest issuers but is not a defense against a compromised key. (This is JWT's `jti`, renamed and given a single concrete, enforceable rule: an obsigil token is not a JWT, so the format-branded abbreviation would be incoherent; `tid` is "token id", a neutral coordinate.)

8.3 exp

A mandate, when present, MUST carry an `exp` clause, and the verifier MUST reject a mandate once the current time is at or past its `exp`. A manifest `exp` claim, if present, is advisory only: the front end MAY use it to refresh early, and the backend MUST NOT enforce it. Expiry is the backend's sole responsibility.

8.4 aud

A *non-empty CBOR array of text strings* naming the verifier or verifiers the mandate is for; a mandate bound to a single verifier carries a one-element array. There is no bare-string form, so every verifier runs the same membership test and never branches on shape. If an `aud` clause is present, the verifier MUST reject the mandate unless its own identifier is a byte-exact member of the array; an empty array names no audience and MUST be rejected. The match is over the *decoded* CBOR text string — the characters the decoder yields, not the on-the-wire encoded bytes — and is a raw octet comparison: neither side applies Unicode normalization, case folding, or any other transformation. A deployment using non-ASCII audience identifiers MUST fix a single normalization form out of band; the case-folding tolerance of §4 applies only to the outer text encoding of the whole token, never to an inner field value such as `aud`. `aud` SHOULD be present whenever the mandate key is shared across more than one service or trust domain — without it, a mandate sealed for one service replays at any other holding the same key. Because the match is membership, one mandate MAY name several such services in its `aud` and be accepted by each, so a bearer calling several services carries a single mandate rather than one per service. A producer that derives an `aud` array by filtering MUST treat an empty result as an error rather than emit it: an empty array is rejected by every verifier, whereas omitting `aud` entirely is accepted by any holder of the key.

8.5 sub

The subject the mandate authorizes — a CBOR text string, typically a user id. Reserved with this meaning; omit it for capability mandates that authorize an action rather than identify a principal.

8.6 iss

A manifest, when present, MUST carry an `iss` claim — a CBOR text string — which the front end reads for display. A front end that opens a manifest lacking its required `iss` MUST treat that manifest as malformed and ignore all of its claims (display nothing from it); it MUST NOT block or fail enforcement on this, since the manifest is advisory and the mandate alone governs. Include `iss` as a mandate clause too when the backend needs the issuer for audit — the manifest's copy never reaches the backend. `iss` is *not* a key selector; key selection is by trial decryption (§16.5).

9 Audiences

The split is by reader, and the readers never overlap:

— *manifest → front end.* The front end receives the whole token, opens the manifest (keyless) and reads its claims — issuer, roles, expiry hint — to drive the UI. It treats them as advisory. If the manifest is absent (a `.0mandate` token), the front end has nothing to display.

— *mandate → backend.* The front end forwards the mandate to the backend as a *manifest-absent token*: the token's leading separator, the mandate's algorithm code, and the sealed mandate (e.g. `.0mandate` for a `b64`, AES-SIV mandate). That leading separator and code still name the encoding and algorithm, so the backend reads both from them, strips them, decodes, decrypts with the secret key, and enforces. Forwarding is thus replacing the manifest with an empty half, and the forwarded value is itself a well-formed obsigil token. The backend never sees, parses, or trusts the manifest, and — because it only ever receives the post-separator part — cannot receive manifest content even when one is present.

Because the backend's sole input is the mandate, every field the backend enforces MUST live in the mandate, as a clause; the manifest carries only claims the front end displays. Nothing binds the two halves cryptographically, so a forged or spliced manifest cannot affect backend enforcement — but it *can* drive whatever a client does with manifest claims, which is why clients MUST treat them as advisory only (§16).

10 Worked example

This section is non-normative; it walks one token end to end so the mechanics of §4 through §8 can be checked against concrete bytes. It uses the published manifest key (§5.2) and, for the mandate, the conformance test mandate key — both deliberately public — so the result reproduces exactly. Hex is lowercase; the encoding is `b64` and both halves use AES-SIV (code `0`).

The mandate

A mandate carrying only a fixed `tid` and an expiry:

```
clauses: tid = 019ed29a-378d-72f0-b462-4929cd2bfcad   (UUIDv7)
         exp = 4000000000                             (NumericDate)
```

encodes to the canonical CBOR map (§7). The keys are the negative integers `-1` (`tid`) and `-2` (`exp`), sorted by encoded byte (`0x20` before `0x21`), with `tid` a 16-byte byte string and `exp` a shortest-form integer:

```
a2                                      map(2)
  20                                      -1  (tid)
  50 019ed29a378d72f0b4624929cd2bfcad     bytes(16)
  21                                      -2  (exp)
  1a ee6b2800                               4000000000
```

i.e. the 25 plaintext octets `a22050019ed29a378d72f0b4624929cd2bfcad211aee6b2800`. Sealing these under the mandate key with AES-SIV (§6.2) prepends the 16-byte synthetic IV; text-encoding the result as `b64` gives the mandate ciphertext. As a manifest-absent token — the value a front end forwards to the backend (§9) — it is:

```
.0XEGe0T5Vih7NhiJsXhrEuLHX7SqEoSOY4PSx91evs1qMZav-laAa5Os
```

The manifest

A manifest advertising an issuer:

```
claims: iss = "auth.example"
```

encodes to `a1246c617574682e6578616d706c65` — `a1` map(1), `24` the key `-5` (`iss`), `6c` a 12-byte text string, then the bytes of `auth.example` — and, sealed keyless under the manifest key, yields the manifest ciphertext.

The token

Joining the manifest half and its code, the `.` separator (which selects `b64`), and the mandate code and half gives the full token:

```
Ifjt1gPO2S2soNJQZjtP8Q8zDe5zvPxl2D2OuejeOQ0.0
   XEGe0T5Vih7NhiJsXhrEuLHX7SqEoSOY4PSx91evs1qMZav-laAa5Os
```

This exact token is a conformance vector (§13); an implementation reproduces it byte for byte from the octets above.

11 Versioning

This document specifies obsigil *v1*. A token carries no in-band obsigil-version signal. The algorithm code (§6) gives crypto-agility — the AEAD sealing either half MAY change, per token, with no change to the obsigil format — but that is distinct from format evolution. A future obsigil version that changes the token grammar or field semantics is negotiated out of band. Version negotiation is out of scope for v1.

12 API conformance

This section defines obsigil's **API conformance profile**: the operations every obsigil library exposes, named once, so that code and prose read the same across languages. It is a *second* conformance dimension, independent of the wire. Wire conformance is fixed wholly by §4 through §8 and the test vectors (§13) — an implementation that reproduces every positive vector byte-for-byte and rejects every negative one with the uniform failure of §16.6 is *wire-conformant* whatever API it exposes — and nothing in this section changes a byte on the wire. An implementation is *API-conformant* when it exposes the operations defined below under the names given here, and a library MAY be one without the other: a wire-conformant library with a nonstandard API is still a valid obsigil implementation. This profile MAY be revised independently of the wire format's version.

Within the profile, an implementation MUST expose the operations named below, transformed only for its host language's casing and idiom (§12.7), and MAY add idiomatic conveniences on top; no convenience may weaken a normative property of §16, the uniform failure surface above all. Where a requirement here restates a §16 property — for example, that a diagnostic read MUST stay non-bearer-facing — its binding force is that of §16, repeated here for locality.

The vocabulary is the spec's own (§2), and it fixes the operation names. A *claim* is a manifest field and a *clause* is a mandate field, so `claims` and `clauses` name the decoded fields of each half; *manifest* and *mandate* name the halves, so `manifest` and `mandate` name those halves as they sit on the wire. The two pairs are the spine of the API — a *noun* per (half, fidelity), not a verb per action: there is no `verify`, `open`, or `forward` operation, only the reads that obtain a clause, a claim, or a half.

12.1 Operations, not an object

Obsigil defines a set of *operations*, written here as `name(inputs) -> result`. The notation is abstract: an operation is realized in each language's natural idiom — a free function, a method on a token value, a builder, or a module export — and those realizations are equals, not one canonical shape with exceptions (§12.7). The operations divide by the caller's role and the key it holds, extending the reader split of §9 to the issuer that mints:

- *keyless* (the front end, holding no key): `claims`, `manifest`, `mandate` — open the advisory manifest and obtain the mandate to forward.
- *minting* (the issuer, holding the secret key): `mint`, and the key generator `generate_key`.
- *verifying* (the backend, holding the secret key): `clauses` — authenticate and enforce.

The split is a property of which operation the caller can reach, not three constructors of one object: only the minting and verifying operations take a mandate key, and an implementation SHOULD keep that custody visible, so a key never flows into a keyless read.

Minting. `mint(fields, mandate-key, params) -> token` seals a mandate (and, optionally, a manifest) and returns the whole token string (§5). `fields` are the application clauses; the reserved clauses (`exp`, `tid`, `aud`, `sub`, `iss`) and the optional manifest are supplied through `params`. `exp` is required (§8.3); `tid` is generated as a fresh UUIDv7 unless supplied (§8.2); the algorithm defaults to AES-SIV (code `0`) and the encoding to `b64` (§6, §4). `generate_key() -> key` returns a fresh 64-byte key from a cryptographically secure generator (§5.1). An implementation MAY also offer an `authorization_header(token, scheme) -> header` convenience that wraps `mandate` in an `Authorization` value (§15).

12.2 Reading a half: three fidelities

Each half is reachable at three fidelities — the encoded wire half, the decrypted plaintext octets, and the decoded fields:

fidelity	manifest (keyless)	mandate (keyed)
encoded half	`manifest(token)`	`mandate(token)`
plaintext	`manifest_plaintext(token)`	`mandate_plaintext(token, keys)`
decoded	`claims(token)`	`clauses(token, keys, policy)`

`manifest(token)` and `mandate(token)` each return one half as a *standalone, well-formed token* — the trailing-separator `manifest0.` and the leading-separator `.0mandate` of §4 — so `mandate(token)` *is* the value the front end forwards to the backend (§9), and it needs no key. Throughout, the token string is `mint`'s output and the input to every read, so no separate token accessor is needed. A `*_plaintext` read returns the decrypted canonical CBOR octets (§7) exactly as sealed, parsing nothing; it is OPTIONAL. The *mandate* plaintext read takes a key and authenticates the half (§16.3); the *manifest* plaintext read only decrypts under the public manifest key, detecting corruption but not forgery (§2). The encoded read is the same for both halves, but from the plaintext fidelity down the mandate read takes a key and the manifest read does not, and at the decoded fidelity policy enforcement separates them further (§12.3).

12.3 The two decoded reads

At the decoded fidelity the manifest and the mandate part exactly as their authority does (§16.3). There are two first-class reads, one per half, and they are all most callers ever need:

- `clauses(token, keys, policy) -> clauses` — the *mandate* read: authenticate under a candidate key, then enforce policy (`exp` not past, `aud` membership, `tid` well-formedness, reserved types). It returns the clauses, or fails with the one opaque error of §16.6. This is the trustworthy read; a backend enforcing a mandate uses only this.
- `claims(token) -> claims` — the *manifest* read: open the keyless manifest and return its claims. It is advisory and authenticates nothing, so a reader MUST NOT trust it (§16.7).

`claims` never raises. When there is nothing trustworthy to show — no manifest, a malformed token, a bad encoding, or a manifest that does not open or is otherwise malformed (a missing required `iss`, §8.6; a wrong-half reserved key; or any other defect, §16.7) — it returns the host language's single *absent* value (a null, `None`, `undef`, or an empty optional), never an exception and never a partial map. Reading an advisory manifest that is not there is a non-event, so — unlike `clauses`, which *fails* — `claims` simply yields nothing; the asymmetry mirrors authority (§16.7). It does *not* distinguish "no manifest" from "a manifest present but unreadable or forged": both are equally without authority, so both yield the one absent value. There is no unchecked variant of `claims` — the manifest is keyless, so opening it is already the unvalidated read.

The optional diagnostic tier

A backend sometimes needs the mandate's contents *without* the policy verdict — to read an `exp` and decide whether to refresh, or to log a soon-to-be-rejected token's `tid`. For this an implementation MAY offer, below `clauses`:

— `clauses_unchecked(token, keys) -> clauses` — authenticate and decode the canonical CBOR map (a non-canonical or duplicate-key encoding still fails, §16.10), but do *not* apply the value checks. Returns a map, or fails if the half does not authenticate or is not canonical CBOR.
— `mandate_plaintext(token, keys) -> octets` — authenticate only, returning the raw CBOR octets with no parsing: the only faithful view of a pathological payload, where a duplicate key that `clauses_unchecked` rejects and a parsed map would silently collapse is visible as bytes.

Both are OPTIONAL and both authenticate — neither ever exposes unauthenticated bytes (§16.3). They are backend-internal diagnostics and MUST stay non-bearer-facing: whether one succeeds where `clauses` would have rejected reveals *why* a token failed, which §16.6 forbids signaling outward. The `unchecked` marker names the hazard; an implementation SHOULD keep that marker, or an equally conspicuous one, in the name.

12.4 Reserved-field access

`claims` and `clauses` return a map in which the reserved fields (wire-encoded at negative integer keys, §8.1) carry their §8 meaning — an implementation MAY surface them under their conventional names (`exp`, `tid`, `aud`, `sub`, `iss`) — while every application key is opaque application data. Over that result an implementation SHOULD offer direct access to the reserved clauses — an `exp`, a `tid`, and an `issued_at`, the last derived from the UUIDv7 `tid` (§8.2), which a raw map lookup cannot compute. Single-field access (a `sub` lookup, a map `get`, or similar) follows host idiom.

12.5 Verification configuration

The `clauses` read takes the candidate *mandate keys* and a `policy`. The keys are tried in order by trial decryption (§16.5); the `policy` bundles the remaining parameters of §16:

— this verifier's own audience identifier, for the `aud` membership test (§8.4);
— a clock-skew leeway for `exp`, itself bounded by a fixed maximum (§16.10);
— an injectable "now" for reproducible tests;
— a maximum decoded size, enforced before any trial decryption (§16.10).

The names follow host idiom; the capabilities are the convention. The singular/plural split is deliberate: minting takes the one secret *mandate key*; verifying takes a list of candidate *mandate keys*.

12.6 Failure surface

Every bearer-facing rejection collapses to one opaque error (§16.6). A granular cause MAY be delivered out-of-band — a callback, a logged value, a separate internal field — for telemetry only, never to the bearer, and never through an incidental channel such as a debug or display rendering of the error that ordinary logging would surface to where a bearer can read it.

12.7 Idiomatic realizations

One operation set, several host shapes, all equal. A language MAY realize the operations as free functions, as methods on a token value, as builders, or as separate modules, provided the names and semantics above stay recognizable across the choice:

— *free functions* — `mint`, `clauses`, `claims`, `mandate` as top-level functions, the closest rendering of the operation list (a natural Go or Perl shape).
— *a token value* — one value constructed over a token, exposing the reads as members; the keyless construction holds no key, the verifying construction takes the keys and policy (a natural Python shape).
— *builders* — a typed minting builder and a typed verifying builder, each terminating in the read it serves; the reference Rust implementation uses this shape, with generically typed clauses and claims.
— *modules* — the keyless operations and the keyed operations in separate units, making key custody a structural boundary (a natural TypeScript package split).

None of these is the canonical model with the others as exceptions: they are four spellings of one operation set. An implementation states which it uses, and a reader who knows the operations knows the library.

13 Conformance and test vectors

Cross-implementation interoperability is established by known-answer test vectors maintained in a separate, language-agnostic repository, not inlined in this document. Vectors come at two layers. An *octet* vector pins a complete sealed input — the master key, the algorithm code, the text encoding, and the exact per-half plaintext CBOR octets — to one exact token; because obsigil seals bytes deterministically (§16.4), that mapping is a true function with no nonce to fix, and the vector reproduces without invoking any serializer: an implementation seals the given octets and MUST match the byte string. A *field* vector pins field *values* (with a supplied `tid`, since minting otherwise generates one) to one exact token, and additionally exercises the canonical CBOR encoder of §7: an implementation encodes, seals, and MUST match. The two layers isolate a failure to the encoder or to the seal. The suite MUST also include, at minimum, negative cases for:

— a token with the wrong number of separators (zero, or a separator run longer than one) or an unrecognized separator
— an unrecognized or unsupported algorithm code
— non-canonical CBOR (including a duplicate map key)
— a map key that is neither a CBOR integer nor a text string
— an unrecognized negative key
— a reserved field of the wrong CBOR type (for example an `exp` that is not an integer, or an `aud` that is not a non-empty array of text strings, §16.10)
— an expired `exp`
— an `aud` mismatch
— a mandate missing its required `tid` or carrying a non-UUIDv7 `tid`

— an empty mandate

An implementation is *wire-conformant* when it reproduces every positive vector byte-for-byte and rejects every negative one with the uniform failure of §16.6; the separate API conformance profile (§12) is independent of this and changes no byte on the wire. A malformed manifest is *not* such a rejection: by §16.7 a conforming front end instead ignores all of that manifest's claims — whether the defect is a missing `iss` (§8.6), a wrong-half reserved key (§8.1), a disallowed map-key type, or non-canonical content — and verification of the token is unaffected, so the suite MAY carry such cases as separate front-end behavioral vectors, kept outside the uniform-failure set above.

14 Relationship to JWT, CWT, and COSE

Obsigil borrows JWT's [8] registered-claim names (`exp`, `iss`, `aud`, `sub`) because they are the recurring coordinates of any time-bounded, multi-party credential — not a JWT invention. An obsigil token is *not* a JWT: it has no JOSE (JSON Object Signing and Encryption) header, no signature, and no base64url-of-JSON wire shape.

Because both halves are CBOR, the nearer neighbors are CWT (RFC 8392 [9]) and its underlying COSE (RFC 9052 [13]): a CWT is CBOR claims secured by COSE, and obsigil's claims are integer-keyed for the same compactness. The sign-based reservation is obsigil's own, though: CWT numbers its registered claims with small *positive* integers and COSE spends negative labels on algorithm and key-type parameters, whereas obsigil reserves the whole negative space to the protocol and leaves the non-negative integers (and text strings) to applications. Obsigil is nonetheless a distinct format, not a COSE profile. Three properties set it apart, and their intersection is the niche it occupies: it is *deterministic* (a sealed half is a pure function of its plaintext, no nonce or signature randomness); it is *split-audience* (a public, forgeable manifest for the front end and a sealed, authoritative mandate for the backend, §9); and it is *opaque* — encrypted, with no key identifier and key selection by trial decryption (§16.5), where COSE carries cleartext headers including a `kid`. The references to JWT, CWT, and COSE in this document are informative throughout.

15 IANA Considerations

This document requests that IANA register the `application/vnd.obsigil` media type in the media types registry, in the vendor tree (RFC 6838 §3.2), using the template below.

An obsigil token is a bearer credential carried in transport metadata — an HTTP `Authorization` value, a cookie, a message field — not a file on disk (§4). Like the JWT it descends from, it is identified by a **media type**, not a file extension or a magic number. The registered type is `application/vnd.obsigil`. A producer labeling an obsigil token (for example in a `Content-Type`) MUST use `application/vnd.obsigil`, and a consumer SHOULD accept it. No file-extension, magic-number, or Macintosh-type registration is made, by design: a token is self-delimiting and self-describing — the separator names the encoding, the adjacent code names the algorithm, and the shape (full, manifest-only, mandate-only) is structural — so neither a filename nor a leading sentinel is needed to interpret one.

The registration follows the RFC 6838 registration template:

```
Type name: application
Subtype name: vnd.obsigil
Required parameters: N/A
Optional parameters: N/A
Encoding considerations: 7bit. An obsigil token is printable
  US-ASCII: two halves over the URL-safe base64 alphabet
  (A-Za-z0-9-_) or the lowercase-hex alphabet (0-9a-f), joined by a
  single "." (selecting base64) or "~" (selecting hex) separator,
  each present half carrying a one-character algorithm code
  (0-9a-z) adjacent to the separator. Either half may be empty.
Security considerations: See the Security Considerations of the
  obsigil specification. In summary: an obsigil token is
  a bearer credential and MUST be protected in transit and at rest,
  the mandate half especially, since possession of a valid mandate
  is sufficient to act on it. The manifest half is keyless, public,
  and forgeable by anyone, and MUST NOT be relied on for any
  access-control decision; only the mandate is authoritative.
  Sealing is deterministic, so identical plaintext under one key
  produces identical halves, revealing equality. All decode,
  authentication, and validation failures are reported uniformly.
Interoperability considerations: Algorithm code "0" (AES-SIV) is
  mandatory to implement; code "1" (AES-GCM-SIV) is optional and
  interoperates only between implementations that share it. A token
  carries no in-band version; versions are negotiated out of band.
Published specification: https://obsigil.org/spec
Applications that use this media type: Applications that issue,
  forward, and verify obsigil mandate tokens, typically a web front
  end and the backend services it authenticates to.
Fragment identifier considerations: N/A
Additional information:
  Deprecated alias names for this type: N/A
  Magic number(s): N/A (a token has no fixed leading bytes)
  File extension(s): N/A
  Macintosh file type code(s): N/A
Person & email address to contact for further information:
  The obsigil project, spec@obsigil.org
Intended usage: COMMON
Restrictions on usage: None
Author: The obsigil project
Change controller: The obsigil project, https://obsigil.org
```

`application/vnd.obsigil` sits in the *vendor tree* (RFC 6838 §3.2): it may be registered directly with IANA under Expert Review, by the obsigil project, with no IETF standards action — and the `vnd.` facet correctly signals a format that is one organization's, not yet a community standard. Should obsigil reach broad, multi-implementation adoption, registration in the *standards tree* as `application/obsigil` would be sought through the IETF (RFC 6838 §3.1), which requires a recognized standards process; on that registration `application/vnd.obsigil` becomes a deprecated alias retained for compatibility. Until then `application/vnd.obsigil` is the only correct identifier: a standards-tree name MUST NOT be used before it is registered, so an implementation MUST NOT label tokens `application/obsigil` today.

16 Security Considerations

Obsigil is built for a single trust domain: a backend, or a set of backends sharing one secret key, that both issues and enforces mandates. Within that domain the token is a sealed, self-contained credential; outside it, the assumptions below are load-bearing and MUST be honored.

16.1 The mandate is symmetric

The mandate is sealed with an AEAD under a secret key, and that key both opens *and* mints mandates — so every party that can verify a mandate can also forge one. Obsigil gives no signer/verifier separation: a single compromised holder of the mandate key can issue arbitrary mandates. Asymmetric origin authentication — public verification, private minting — is out of scope for obsigil.

16.2 The mandate must be authenticated

The mandate's integrity is the whole basis of enforcement. Every algorithm in the registry is an authenticated AEAD (§6), so a well-formed mandate is authenticated by construction — there is no confidentiality-only code to mis-select. A verifier MUST nonetheless reject any mandate that fails authentication (tampering, wrong key, wrong algorithm code), yielding no plaintext. Implementations SHOULD enforce the AEAD requirement structurally rather than by a runtime check — the reference implementation compiles only registered AEADs, so an unauthenticated mandate is unrepresentable, not merely rejected.

16.3 Authentication and policy are distinct layers

Verifying a mandate is two separable steps. *Authentication* — the AEAD opening the sealed half under a candidate key (§16.2, §16.5) — is mandatory and cannot be skipped: the mandate is *encrypted*, not merely signed, so there is no plaintext to read until a key authenticates it, and a wrong key or a tampered half yields nothing. *Policy validation* — the value checks of §16.10 and §8 (`exp` not past, `aud` membership, `tid` a well-formed UUIDv7, reserved-field types) — runs only on already-authenticated plaintext. This is the chief departure from a signed-but-cleartext token: there, the payload is readable without the key and only its *trust* is in question; here, the mandate is unreadable without the key, and reading it *is* authenticating it. The manifest (§5.2) is the only half a party can read without a key, and it is non-authoritative by construction (§16.7).

An implementation MAY expose a mandate's authenticated plaintext with the policy checks deliberately *not* applied — to read an `exp` to decide whether to refresh, to log the `tid` or `iss` of a token it is about to reject, or for diagnostics. Such a path MUST still authenticate — it MUST NOT expose unauthenticated mandate contents — and MUST remain non-bearer-facing: whether it succeeds where full verification would have rejected reveals *why* a token failed, the policy-failed versus authentication-failed distinction §16.6 forbids signaling outward. An implementation offering such a path SHOULD make its unvalidated nature evident at the call site. The manifest needs no parallel mode: opening it is, definitionally, the unvalidated read.

16.4 Deterministic sealing requires unique plaintext

Obsigil seals both halves *deterministically* — no random nonce. The value is twofold: there is no nonce to generate, manage, or catastrophically reuse, and sealing a *given* mandate is a pure function of its inputs, which is what lets a known-answer vector pin one exact token with nothing to fix (§13). (Minting a *fresh* mandate is not pure — obsigil generates the `tid`, §8.2 — so it is the sealing function, over a mandate whose `tid` is already chosen, that is deterministic, not the mint convenience above it.) A deterministic AEAD has exactly one property a probabilistic one lacks: identical plaintext under the same key yields identical ciphertext, so an observer can tell when two halves are byte-identical. Obsigil neutralizes this for the mandate through the unique `tid` (§8): every mandate plaintext is then distinct, so below the per-key birthday bound of §6.2 no two mandates collide and the equality channel stays closed. AES-SIV and AES-GCM-SIV are designed for exactly this deterministic, unique-input use (RFC 5297; RFC 8452). The manifest is keyless and public, so equality among manifests leaks nothing of value and needs no `tid`.

This relocates one obligation: the mandate's confidentiality-of-equality now rests on `tid` actually being unique, where a random nonce would have enforced it automatically. Obsigil discharges the obligation by *generating* the `tid` as a fresh UUIDv7 by default (§8.2) — its millisecond timestamp and 74 random bits collide only with negligible probability — so uniqueness holds by construction. A producer that supplies its own `tid` takes that obligation back (§8.2).

Length is not hidden either: a half's encoded length reveals its plaintext length plus the algorithm's fixed overhead, and deterministic sealing keeps that length stable for a stable clause shape. A deployment whose clause sizes or shapes are sensitive SHOULD pad inside the mandate before sealing, or adopt a fixed-size clause profile.

16.5 Key selection is by trial decryption

A verifier that holds more than one candidate mandate key MUST select the key by trial: attempt to open the mandate under each candidate and accept the first that authenticates. The algorithm is already named by the cleartext code, so the trial is over keys alone. Because the mandate is an AEAD, the wrong key fails closed — authentication fails and yields no plaintext — so trial decryption reveals no plaintext and keeps the token opaque. It is not, however, constant-time: with candidates tried in order, accepting under the i-th key takes time growing with i, so an adversary who submits a known-valid token and measures the response can learn the matching key's position — the very "which key sealed this token" signal obsigil otherwise withholds. A verifier that wants to preserve that opacity SHOULD try all candidate keys before responding, so its total work is independent of which key, or how many, authenticate. Obsigil deliberately carries no plaintext key identifier: the algorithm code names a *cipher*, not a key, and a key selector in the clear would reveal which key sealed a token and add a non-opaque field to an otherwise opaque credential. (`iss`, where present, is for display and audit, not key selection — see §8.)

16.6 Failures are uniform and opaque

A verifier MUST treat every rejection — malformed token, wrong or unrecognized separator, unrecognized algorithm code, authentication failure, wrong key, absent or empty

mandate, missing or malformed `tid`, expired `exp`, `aud` mismatch, missing required clause — as a single, indistinguishable failure, and MUST NOT signal to the bearer *why* a token was rejected (by error code, message, or, where feasible, timing). Distinguishable failures turn the verifier into an oracle: they leak whether a key matched, whether a token merely expired, or whether an audience was wrong. Internal logging or telemetry, not visible to the bearer, MAY distinguish causes.

16.7 The manifest is non-authoritative

The manifest is sealed keyless, so anyone can forge one (§5.2). Clients MUST NOT make any security decision from manifest claims; they are display and UX hints only. Because nothing binds the two halves, an attacker can pair any manifest with any mandate and deliver it over an untrusted channel (a phishing link, an open redirect). That splice is harmless *only* while clients honor the MUST NOT above; the moment a client trusts the manifest, it becomes an attack on that client.

Concretely, a client MUST NOT present manifest claims in a way that implies server verification. A UI that needs an authoritative subject, role, action, or issuer MUST obtain it from the backend after the mandate is verified, never from the manifest — a forged manifest can otherwise show `admin`, a trusted issuer, or a distant expiry while the backend enforces something entirely different.

The same non-authority fixes how a front end handles a *malformed* manifest. Every decode and validation rule this specification states for a half's authenticated CBOR content — canonical encoding and key types (§7, §16.10) and the reserved-field rules (§8) — is a requirement on the verifier's authoritative read of the *mandate*. A manifest that opens but whose content violates any of them is malformed, and a front end MUST ignore all of its claims and MUST NOT block or fail enforcement on it — §8.6 states this for a missing `iss`, and it holds for any such defect. Because the backend never reads a manifest (§9), no manifest defect is ever a uniform token failure (§16.6); the structural rules of §4, such as a bad separator or an unimplemented algorithm code, precede any decode and are a separate matter of token parsing.

16.8 Empty halves authorize nothing

A manifest-only token (`manifest0.`) carries no mandate, hence no enforceable content: forwarded to a backend it presents an empty mandate, which the backend rejects under §16.6. A mandate-only token (`.0mandate`) drops only advisory display. Enforcement rests entirely on the mandate, so neither degenerate shape weakens the model.

16.9 The mandate is a bearer credential

Possession is use: whoever captures a mandate can replay it until its `exp`. Deployments MUST transport tokens over a confidential, authenticated channel (e.g. TLS) and SHOULD keep `exp` short. Because every mandate carries a unique `tid` (§8), a deployment MAY detect or revoke a replayed mandate by recording spent `tids`; a deployment that does so MUST retain each entry until at least the mandate's `exp` (plus any configured leeway), since evicting one earlier reopens the replay window it was meant to close. A mandate that more

than one service or key domain can open SHOULD be scoped to its intended audience via an `aud` clause (§8).

16.10 Limits and robustness

An implementation SHOULD enforce a configurable maximum decoded size for a token and for each half, and MUST reject an oversize token under the uniform failure of §16.6 *before* any trial decryption (§16.5), so the size check is not itself an oracle. Without such a bound, an attacker can force repeated full-token AEAD work by submitting large tokens against a set of candidate keys.

Each half is a canonical CBOR map (§7), and a verifier MUST reject a half whose plaintext is not well-formed, canonical CBOR (RFC 8949 §4.2, §7) — including, but not limited to, an indefinite-length item, a non-shortest integer, length, float, or simple value, a `NaN`, a text string that is not valid UTF-8, map keys out of sorted order, a *duplicate map key*, or any byte after the single top-level CBOR map. Rejecting duplicates at the decoder forecloses the last-wins / first-wins divergence by which two verifiers could read different values from one byte string; obsigil needs no separate per-name duplicate rule. An unrecognized negative key MUST likewise be rejected, an unrecognized non-negative or text-string key MUST be ignored, and a map key that is neither a CBOR integer nor a text string MUST be rejected (§7, §8.1).

A reserved field present with a value of the wrong type — `tid` (key `-1`) not a 16-byte UUIDv7 (§8.2), `exp` (`-2`) not a NumericDate integer, `aud` (`-3`) not a non-empty array of text strings (§8.4), `sub` (`-4`) not a text string, or `iss` (`-5`) not a text string — MUST cause uniform rejection (§16.6).

Any clock-skew leeway a verifier allows for `exp` (§8.3) SHOULD NOT exceed a small fixed bound (for example, 60 seconds) and MUST be bounded by a configured maximum; an unbounded leeway silently extends every token past its expiry.

References

[1] Carsten Bormann and Paul E. Hoffman. Concise binary object representation (CBOR). RFC 8949, RFC Editor, December 2020. STD 94. https://www.rfc-editor.org/rfc/rfc8949.html.

[2] Scott Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, RFC Editor, March 1997. https://www.rfc-editor.org/rfc/rfc2119.html.

[3] David Crocker and Paul Overell. Augmented BNF for syntax specifications: ABNF. RFC 5234, RFC Editor, January 2008. https://www.rfc-editor.org/rfc/rfc5234.html.

[4] Kyzer Davis, Brad Peabody, and Paul Leach. Universally unique IDentifiers (UUIDs). RFC 9562, RFC Editor, May 2024. https://www.rfc-editor.org/rfc/rfc9562.html.

[5] Ned Freed, John Klensin, and Tony Hansen. Media type specifications and registration procedures. RFC 6838, RFC Editor, January 2013. BCP 13. https://www.rfc-editor.org/rfc/rfc6838.html.

[6] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce misuse-resistant authenticated encryption. RFC 8452, RFC Editor, April 2019. https://www.rfc-editor.org/rfc/rfc8452.html.

[7] Dan Harkins. Synthetic initialization vector (SIV) authenticated encryption using the advanced encryption standard (AES). RFC 5297, RFC Editor, October 2008. https://www.rfc-editor.org/rfc/rfc5297.html.

[8] Michael B. Jones, John Bradley, and Nat Sakimura. JSON web token (JWT). RFC 7519, RFC Editor, May 2015. https://www.rfc-editor.org/rfc/rfc7519.html.

[9] Michael B. Jones, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. CBOR web token (CWT). RFC 8392, RFC Editor, May 2018. https://www.rfc-editor.org/rfc/rfc8392.html.

[10] Simon Josefsson. The base16, base32, and base64 data encodings. RFC 4648, RFC Editor, October 2006. https://www.rfc-editor.org/rfc/rfc4648.html.

[11] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand key derivation function (HKDF). RFC 5869, RFC Editor, May 2010. https://www.rfc-editor.org/rfc/rfc5869.html.

[12] Barry Leiba. Ambiguity of uppercase vs lowercase in RFC 2119 key words. RFC 8174, RFC Editor, May 2017. BCP 14. https://www.rfc-editor.org/rfc/rfc8174.html.

[13] Jim Schaad. CBOR object signing and encryption (COSE): Structures and process. RFC 9052, RFC Editor, August 2022. STD 96. https://www.rfc-editor.org/rfc/rfc9052.html.

Index