The VaultSort Vault

Cryptographic architecture, threat model and known limits

An encrypted store for files and notes on macOS and Windows, keyed by hardware security keys, Touch ID, Windows Hello, a password or a printed recovery code. This document states exactly what it protects, and exactly what it does not.

DOCUMENT
Version 1.0

APPLIES TO
VaultSort 5.3.0+ · vault format v1

DATE
September 2026

About this document

This paper describes the cryptographic design of the VaultSort Vault: how a vault is keyed, how files and notes are encrypted inside it, what is authenticated, what survives a crash, and what the known limitations are. It is written for technically literate users who want to verify VaultSort's claims before trusting it with sensitive files, and for anyone reviewing the format.

It is a companion to the *VaultSort V4 Encryption — Security Design Document*, which covers the separate single-file encryption format. The two share primitives and design instincts, but they are different formats with different threat models. V4 protects a file wherever it happens to sit. The Vault protects a **place**, and therefore has to answer questions V4 never faced: what an observer of the store learns, what an interrupted write leaves behind, and what happens when the set of people who can open it changes.

Everything described here is implemented in `src-electron/modules/vault/`. Constants named in this document are the constants in the source, not paraphrases of them. Section 12 lists the files to read to check each claim for yourself.

DOCUMENT VERSION	1.0	DATE	September 2026
APPLIES TO	VaultSort 5.3.0 and later	FORMAT	Vault format version 1
PLATFORMS	macOS and Windows	SUPERSEDES	Nothing — first publication

> THE STANDARD THIS DOCUMENT HOLDS ITSELF TO
>
> A specification that lists only strengths is not useful. Section 10 states the limits of format version 1 as plainly as the rest of the paper states its properties, including the ones that would be easier to leave out. Where a claimed property is not covered by an automated test, this document says so rather than leaving the gap implicit. **If something here does not match the source, the source is the truth and this document is the bug.**

HOW TO READ IT

Sections 1 to 3 set out what the Vault is for, what it defends against, and the three constraints that shaped the format. Sections 4 to 7 are the format itself — keys, disk layout, objects, manifest. Section 8 covers crash safety, which in a long-lived store is not a detail. Section 9 covers what happens while a vault is open. Sections 10 to 12 are the limits, the rationale, and how to verify the whole thing independently.

Contents

SECTION 01

A vault protects a place

01

A VaultSort Vault is an encrypted store on the local disk that holds files and notes. It is opened with a hardware security key, Touch ID, Windows Hello, a password, or a printed recovery code — any one of which is sufficient, and any of which can be added or removed without re-encrypting the contents. While it is locked, the directory it occupies yields ciphertext and nothing else: no names, no folder structure, no note titles, no indication of what kind of thing any object is.

Why a place is harder than a file

Single-file encryption has a clean problem statement. One file goes in, one blob comes out, and the blob either decrypts or it does not. A vault has to survive being a durable, long-lived thing that people write to repeatedly, on hardware that fails at inconvenient moments, with a membership list that changes over years.

That difference produces three questions a single-file format never has to answer.

- **What does an observer of the store learn?** A directory an attacker can watch over time leaks more than any single blob does — object counts, sizes, write times. Section 2 states exactly what is visible, and section 10 states plainly that it is not padded away.
- **What does an interrupted write leave behind?** Over a vault's life it will be interrupted. The format is organised so that every interruption lands in a state that can be reasoned about and repaired, rather than one that has to be guessed at. See section 8.
- **What happens when the set of people who can open it changes?** Adding an unlock method to a vault holding a hundred gigabytes must not re-encrypt a hundred gigabytes. That single requirement is what forces the key hierarchy in section 4.

> THE ONE-SENTENCE VERSION
> Every object in the store is encrypted under a key reachable only through the vault master key; the master key is reachable only by unwrapping it with a registered slot; and there is no software path to it from anything on disk.

SECTION 02

Threat model

02

A threat model is only useful if it is symmetrical — as specific about the attacks the design does not stop as about the ones it does. Both halves are below, in summary first and then in detail.

WHAT THE VAULT PROTECTS AGAINST	WHAT IT DOES NOT
✓ The vault directory, copied wholesale	✗ Someone at your unlocked, running machine
✓ The app's WebAuthn credential registry	✗ Malware, kernel compromise, an attached debugger
✓ An object swapped, renamed or re-homed	✗ Compulsion of a biometric, or a key plus its PIN
✓ A manifest lifted from another vault	✗ A recovery code stored in a plaintext note
✓ An edited header — slots, salt, KDF params	✗ Object counts, ciphertext sizes and write times
✓ A removed slot restored from a backup	✗ A vault directory copied to a machine with no floor
✓ A crash or kill in the middle of a write	✗ Plaintext after it leaves — exports, note text
✓ A second copy of the app writing at once	✗ Durable revocation: format v1 cannot re-key

Figure 1 — The Vault's threat model at a glance. The right-hand column is not a list of future work; it is the honest boundary of what encryption at rest can do.

2.1 What the Vault protects against

An attacker holds the vault directory, but no key, password or recovery code

Copying `vault.json`, `manifest.enc` and the entire `objects/` tree yields ciphertext, object count, ciphertext sizes and file timestamps, and nothing else. There is no software path to the master key.

An attacker holds the vault directory and the app's credential store

The WebAuthn credential registry holds credential IDs, public keys and salts. None of those unwrap a slot. A PRF slot's wrap key is derived from an HMAC secret held inside the authenticator, which never leaves it. Without the physical device, the registry is inert.

An object is swapped, renamed, or copied in from another vault

Each object's data-encryption key is wrapped under a key derived *for that object id*. An object file moved to another id fails to unwrap. A decrypt can also be told which id it expects and refuses a mismatch. Two objects never share a wrap key, so exchanging two files on disk is detected rather than quietly decrypting the wrong

content under a valid tag.

The manifest is copied from another vault

The manifest envelope's additional authenticated data binds the vault id, so a manifest lifted from one vault fails to open in another even if both were somehow sealed with the same key.

The header is edited

`vault.json` carries an HMAC-SHA-256 over the canonical JSON of the validated header, keyed from the master key. Any edit — adding a slot, changing KDF parameters, changing the salt — is detected on the next unlock and the user is asked before anything proceeds. The MAC is verified only *after* a successful unwrap, because the key to check it does not exist before then.

A removed slot is restored from a backup

The header carries a monotonically increasing generation counter, inside the MAC. The app remembers, outside the vault directory, the highest generation it has seen for that vault id. A header whose generation has gone backwards is refused until the user confirms — which is what stops a restored backup silently re-enabling an unlock method the user removed.

The application is interrupted mid-write

Every multi-step operation writes a sealed intent record before it acts. Replay on the next unlock brings the store to a consistent state. A file is either fully in the vault or not in it; there is no half-written object that reads as valid.

A second copy of the app writes to the same vault

A running task holds a filesystem lease on the vault directory, with a heartbeat and stale-pruning. A second task is refused rather than allowed to interleave writes.

2.2 What the Vault does not protect against

An attacker with your unlocked machine while the vault is open. While a vault is unlocked its keys are in the process's memory and its contents are readable by that process. VaultSort is not a defence against someone sitting at your unlocked, running session. Auto-lock narrows that window; it does not close it.

Malicious software with filesystem or process access. The Vault encrypts at rest. It does not defend against a compromised OS, kernel malware, a debugger attached to the process, or anything that can read the app's memory while a vault is open.

Compulsion, or an attacker holding both your device and your biometric. If someone can compel Touch ID or Windows Hello, or has your key and its PIN, they can open the vault. This is the same limitation as any hardware-backed scheme.

A recovery code stored carelessly. The recovery code is a full unlock method. Written into a plaintext note or a synced document, it defeats everything above. Treat it as you would a wallet seed phrase.

Traffic analysis of the object store. An attacker with repeated access to the directory learns how many objects exist, their ciphertext sizes — plaintext plus sixteen bytes — and when each was last written. Names, kinds, folder structure and titles are not there; they are in the encrypted manifest. But sizes and timing are visible and are not padded.

A vault directory copied to another machine, with respect to rollback. The generation floor is per-machine. A vault directory copied elsewhere starts with no floor, so the rollback check does not apply there. Durable revocation requires re-keying the master key, which format version 1 does not implement; see section 10.

Anything about the plaintext once it leaves. Exported files are ordinary files. A note opened in the editor exists as a string in the renderer's memory, and JavaScript strings cannot be reliably wiped.

SECTION 03

Design goals

03

Three constraints shaped the format. Everything in sections 4 to 8 follows from one of them.

01

A vault is a place, so membership must be cheap to change

Adding an unlock method to a vault holding a hundred gigabytes must not re-encrypt a hundred gigabytes. That forces a key hierarchy rather than a single key, and it is the reason for the two-tier design in section 4.

02

A vault is long-lived, so interruption is normal

Over a vault's life it will be interrupted — a lid closing, a battery dying, a force-quit. Crash safety is therefore not a nicety bolted on at the end but the thing the write paths are organised around. See section 8.

03

A vault must never become unopenable through ordinary misfortune

Losing one security key is ordinary misfortune. The format therefore supports several independent unlock methods, refuses to remove the last one, and puts that refusal in the format layer rather than in the interface.

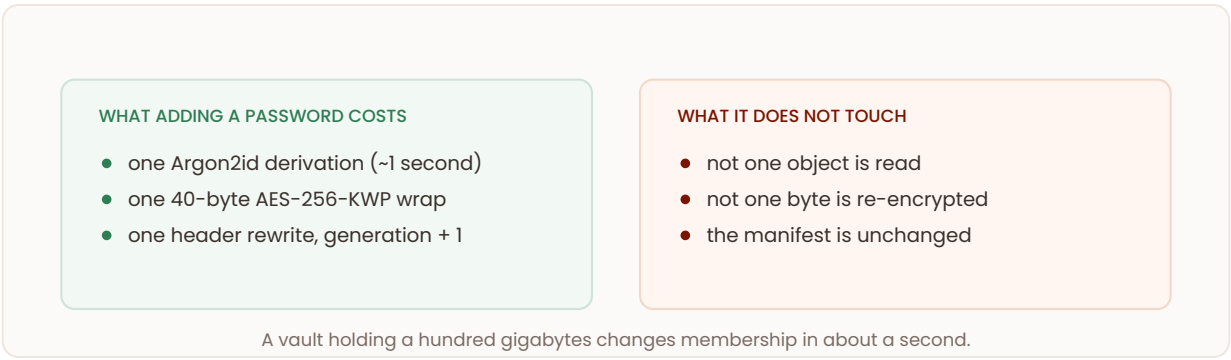


Figure 2 — The first goal, made concrete. Adding an unlock method touches the header and nothing else; section 4.6 explains why.

The third goal is worth a note on where it lives. A rule enforced in the interface is a rule that a future refactor, a scripted path or an automated job can walk around. Refusing to remove the last slot is enforced in the format layer, where every caller has to pass through it — the same reasoning that keeps the licence check out of every path that opens, reads or exports a vault (section 11).

SECTION 04

Key hierarchy

04

One root secret, three working keys derived from it, and one key per object derived from one of those. Nothing on disk is encrypted directly under anything a user holds.

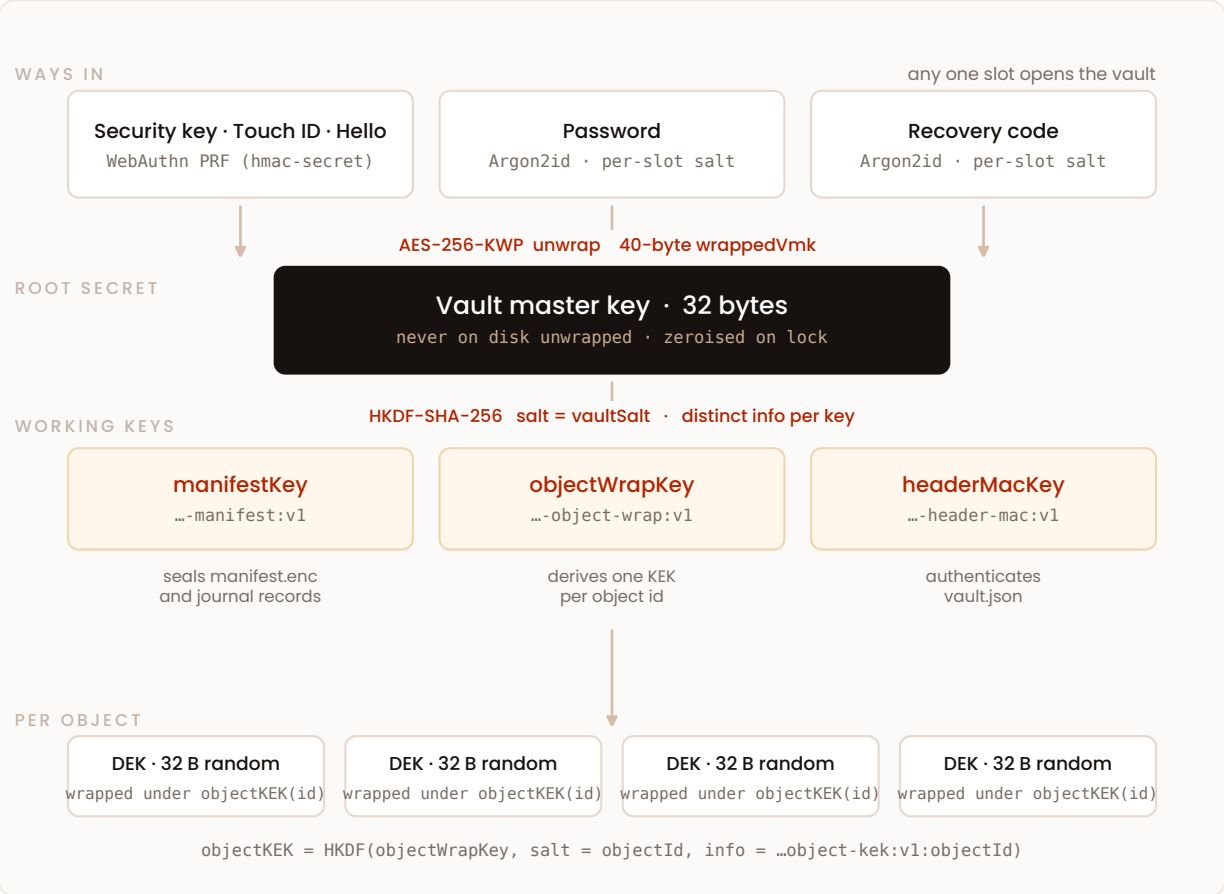


Figure 3 — The full hierarchy. Slots wrap the master key; objects are encrypted under keys derived from it. That separation is what makes adding or removing an unlock method a header-sized operation rather than a whole-vault one.

4.1 The vault master key

On creation the app generates a 32-byte vault master key (VMK) from the system CSPRNG, and a 32-byte random `vaultSalt` which is stored in the header and used as the HKDF salt for every derivation below. The VMK is never written to disk in any form other than wrapped inside a slot. It exists in process memory only while the vault is unlocked.

4.2 Derived working keys

Three working keys are derived from the VMK, each with a distinct `info` string so that no two derivations can collide:

THE THREE WORKING KEYS

```
HKDF-SHA-256(ikm = VMK, salt = vaultSalt, info = <info>)

    manifestKey     info = "vaultsort-vault-manifest:v1"
    objectWrapKey   info = "vaultsort-vault-object-wrap:v1"
    headerMacKey    info = "vaultsort-vault-header-mac:v1"
```

All three are 32 bytes. They are derived on unlock and zeroised on lock, together with the VMK copy the session holds.

4.3 Slots: one seal per way in

A *slot* is one wrapped copy of the VMK, plus whatever metadata is needed to derive the key that wrapped it. The header holds an array of them, and any single slot opens the vault. Every slot, regardless of kind, carries these fields:

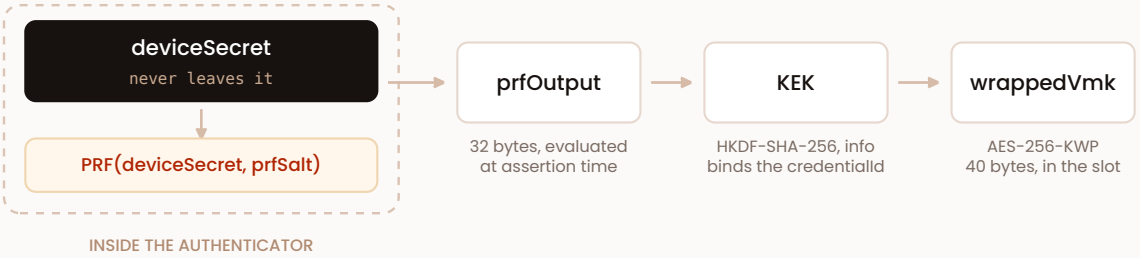
FIELD	MEANING
`kind`	`webauthn-prf`, `password` or `recovery-code`
`slotId`	UUID, stable across header rewrites
`addedAt`	ISO 8601 timestamp
`wrapAlgorithm`	`aes-256-kwp`
`wrappedVmk`	base64, always 40 bytes decoded

AES-256-KWP is RFC 5649 key wrapping. It carries its own integrity check, so a wrapped blob copied from one slot into another fails at unwrap rather than producing a wrong key that is then used. The number of slots a header may carry is capped on the read side — derived from the maximum number of registered credentials plus a password, a recovery code, and headroom for archived keys. A header exceeding that cap is refused rather than truncated.

4.4 PRF slots: security keys, Touch ID, Windows Hello

A `webauthn-prf` slot is opened by a WebAuthn assertion using the PRF extension (`hmac-secret` at the CTAP layer). Each credential is issued a random 32-byte `prfSalt` at registration, kept in the app's credential registry, and passed as `prf.eval.first` on every assertion. The authenticator evaluates it against a secret that never leaves the device and returns a 32-byte output:

```
prfOutput  = authenticator PRF(deviceSecret, prfSalt)      // inside the device
KEK        = HKDF-SHA-256(ikm  = prfOutput,
                          salt = vaultSalt,
                          info = "vaultsort-vault-wrap:v1:" + credentialId)
wrappedVmk = AES-256-KWP(KEK, VMK)
```



The prfSalt is public — 32 bytes from the credential registry. Holding it is worth nothing without the device that evaluates it.

Figure 4 — The PRF path. The salt is public and the credential id is public; the only secret is the one that never leaves the authenticator.

The `prfSalt` is not secret — it lives in the registry in the clear — and holding it is worth nothing without the device that evaluates it. Rotating it would orphan everything the credential protects, so it is fixed for the credential's life. Binding the credential id into `info` means two credentials on the same vault derive different wrap keys even if an authenticator were ever to return the same PRF output for both.

The slot stores `credentialId`, a user-visible `label`, `authenticatorKind` (`cross-platform` for a removable key, `platform` for Touch ID or Windows Hello), and the HKDF hash and info it was wrapped under — recorded rather than merely recomputed. Validation checks that recorded info against the one the slot's own credential id implies, so a slot whose credential id was edited to point at a different key is rejected as an invalid header instead of silently deriving something else. The label is plaintext on disk, and the interface says so.

Three properties worth stating explicitly

- **Credentials are registered non-discoverable** (`requireResidentKey: false`), so a vault slot does not consume one of the authenticator's limited discoverable credential slots. The trade is that the application names the credential to use rather than the authenticator volunteering it.
- **FIDO2 with PRF support is required.** A device that does not report `prf.enabled` at registration is refused there, with a specific error, and its credential is never written to disk — so a key that could not open a vault can never end up looking like one that can. PRF also requires user verification, so registration forces the PIN or biometric prompt rather than accepting a presence-only tap.
- **The PRF output returned at registration is not trusted.** Some authenticators return one there and some do not; only `prf.enabled` is read at that point, and the output is evaluated for real at assertion time. An output of the wrong length, or one that is all zeroes, is discarded rather than used as key material.

Credential ids are length-checked before use: Node's HKDF refuses an `info` over 1024 bytes, and a WebAuthn credential id may be up to 1023 raw bytes. An over-long id is refused with a stable error code instead of surfacing a `RangeError`.

4.5 Argon2id slots: passwords and recovery codes

A `password` or `recovery-code` slot derives its wrap key with Argon2id over a 16-byte random per-slot salt:

```
KEK        = Argon2id(secret, slotSalt, { m, t, p })
wrappedVmk = AES-256-KWP(KEK, VMK)
```

The parameters are stored in the slot, so a vault created on a fast machine still opens on a slow one. They are calibrated on the machine that creates the slot, targeting roughly one second. Because they are *read from the header*, they are attacker-controlled input, and they are bounded in both directions — at wrap and at unwrap alike:

	m (KiB)	t	p	WHAT THE BOUND STOPS
Floor	65536 (64 MiB)	3	1	A tampered slot requesting a trivial derivation that would make the password cheap to attack.
Cap	524288 (512 MiB)	12	4	A tampered slot requesting a derivation that would exhaust the machine. A header is not allowed to turn an unlock attempt into a denial of service.

Passwords are NFKC-normalised before derivation. Two visually identical strings composed differently — a precomposed **é** versus **e** plus a combining acute — would otherwise derive different keys and lock the user out on a different keyboard. Recovery codes go through the same path with the shared recovery-code normaliser, so formatting and case differences in transcription do not matter.

4.6 Why adding a slot re-encrypts nothing

This is the practical consequence of the hierarchy, and the reason it exists. A slot wraps the VMK. Objects are encrypted under keys derived from the VMK, not from any slot. So adding a password to a vault normally opened with a security key computes one Argon2id derivation and one 40-byte wrap, and writes a header. Removing a slot is the same in reverse (Figure 2). It is also why losing a security key costs nothing: the other slots were never derived from it.

SECTION 05

On-disk layout

05

5.1 The vault directory

VAULT DIRECTORY

```
<vault root>/
  .vaultsort-vault        marker file; every destructive feature refuses at or below it
  vault.json              the header
  vault.json.bak          last good header, for recovery
  vault.json.lock         cross-process advisory lock on the header
  manifest.enc            sealed manifest
  manifest.enc.prev       previous generation
  objects/
    ab/
      ab12….vso           one object per file or note
    _orphaned/            objects replay could not attribute (kept, never deleted)
  journal/
    <recordId>.jrn        intent records for operations in flight
    _corrupt/             records that failed to open
    .unclean              left by a close that gave up draining
```

Object ids are 32 lowercase hex characters, sharded one level by the first two. The `.vaultsort-vault` marker is not a security control against an attacker; it is how the rest of the application recognises a vault. Every feature that deletes, shreds or overwrites refuses to act at or below a directory holding one, and also refuses to act on a directory that *contains* one.

5.2 The header

vault.json · FORMAT VERSION 1

```
{
  "formatVersion": 1,
  "vaultId": "<uuid>",
  "createdAt": "<ISO 8601>",
  "vaultSalt": "<base64, 32 bytes>",
  "headerGeneration": 7,
  "slots": [ /* see 4.3 */ ],
  "headerMac": "<base64 HMAC-SHA-256>"
}
```

Validation is strict: unknown keys are refused rather than ignored, base64 fields must decode to exactly their declared length with a canonical alphabet and padding, and a header with zero slots is refused on read.

5.3 Header authentication

```
headerMac = HMAC-SHA-256(headerMacKey, canonicalJson(validate(header without headerMac)))
```

Three details matter.

Validate, then MAC. The MAC covers the canonical JSON of the *validated* structure, not the bytes as they arrived. A header that would be rejected by validation can never produce a MAC that verifies.

Canonical JSON. Object keys are sorted, so re-serialisation is stable and the MAC does not depend on incidental formatting.

Verified after unwrap, compared in constant time. `headerMacKey` is derived from the VMK, so the MAC cannot be checked until a slot has been successfully unwrapped. Comparison is `timingSafeEqual`. A missing MAC and a mismatched MAC are distinct error codes, and both stop the open to ask the user rather than proceeding silently.

5.4 Rollback protection

The header MAC authenticates content, not time. A valid *older* header — from a backup, or from an attacker who kept a copy — would verify perfectly while re-enabling a slot the user had removed.

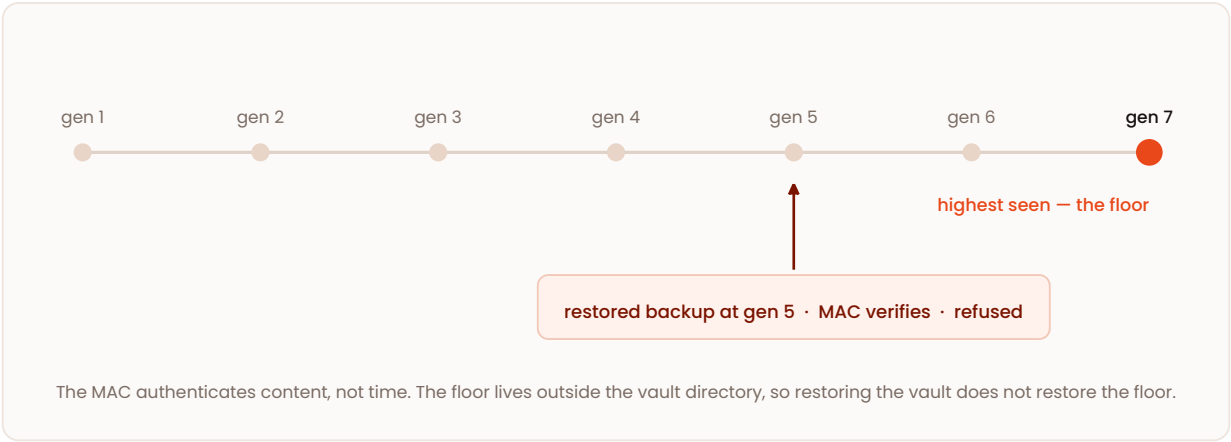


Figure 5 — `headerGeneration` is monotonic, incremented by every slot change, and covered by the MAC.

The app keeps a floor of the highest generation seen per vault id in `<userData>/vault-header-floor.json`, outside the vault directory so that restoring the vault directory does not restore the floor with it. A header below the floor stops the unlock and asks.

> A DELIBERATE FAILURE MODE
> The floor file is conservative when damaged. A file that exists but cannot be parsed is not treated as empty and is not moved aside: reads and writes both fail until it is repaired or removed. A corrupt floor therefore degrades to *"the check was skipped, and you were told"* rather than silently dropping every vault's floor.

SECTION 06

Objects

06

One object per file or note, under its own key, carrying no metadata about it.

6.1 Object file format

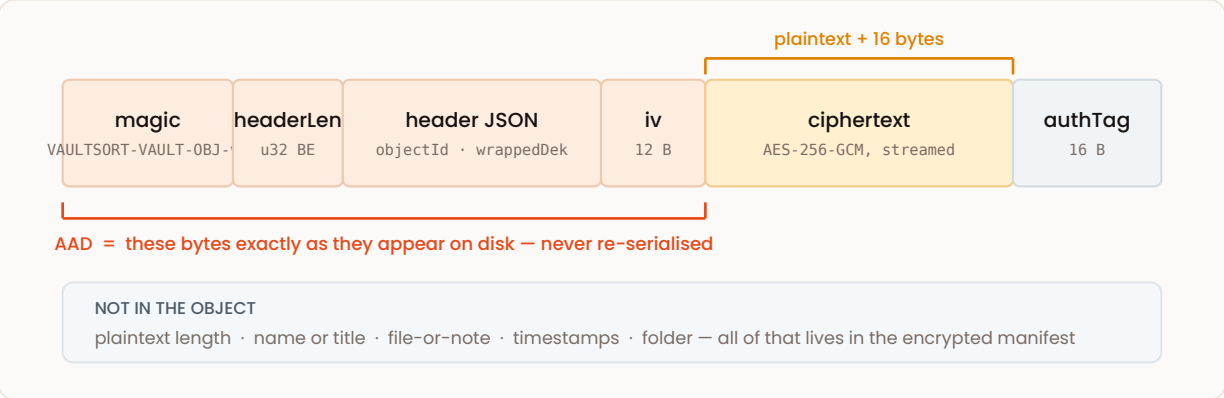


Figure 6 — The `.vso` object layout. The GCM additional authenticated data is everything before the ciphertext, byte for byte, as it appears on disk.

A reader must use the original on-disk prefix and must not re-serialise the parsed header to reconstruct it — the same rule as the V4 format, and for the same reason: re-serialisation could produce different bytes and silently break authentication.

6.2 Per-object key derivation

Each object has its own random 32-byte data-encryption key, wrapped under a key derived for that object alone:

```
objectKEK  = HKDF-SHA-256(ikm  = objectWrapKey,
                          salt = objectId bytes,
                          info = "vaultsort-vault-object-kek:v1:" + objectId)
wrappedDek = AES-256-KWP(objectKEK, dek)
```

Both the DEK and the IV come from the CSPRNG through a single seam. There is deliberately no API that accepts a caller-supplied DEK or IV: GCM under a reused key and nonce loses confidentiality and integrity at once, and an interface that allowed both would make that reachable by accident.

6.3 What deleting means

The wrapped DEK lives in the object's own header, not in the manifest. Removing a manifest entry makes an object *unreferenced*; it is unlinking the `.vso` file that makes the content unrecoverable. The store therefore treats the unlink as load-bearing, and quarantines rather than deletes any object it cannot attribute.

SECTION 07

The manifest

07

Everything the object store deliberately omits — names, kinds, structure, sizes, timestamps — lives here, encrypted.

7.1 Envelope

The manifest and the journal records share one sealed envelope, distinguished by magic so that a record can never be mistaken for a manifest:

```
magic | iv (12 B) | AES-256-GCM ciphertext of UTF-8 JSON | tag (16 B)
AAD = magic || vaultId (UTF-8)

manifest magic   VAULTSORT-VAULT-MANIFEST-v1
journal magic    VAULTSORT-VAULT-JRN-v1
```

Binding the vault id into the AAD is what makes a manifest non-portable between vaults. The manifest is sealed under `manifestKey` and written atomically, temp-then-rename, with the previous generation retained as `manifest.enc.prev`. Its generation is inside the ciphertext and therefore authenticated; the store compares it against `.prev` and against the journal, so an older manifest restored over a newer one is noticed.

Bounds are enforced on both read and seal, so a vault can never write a manifest it would refuse to read back: **64 MiB** and **150,000 entries**, with entry names capped at 255 characters.

7.2 Contents, and the content hash

The manifest holds the folder tree and one entry per item: id, parent, kind (`folder`, `file` or `note`), name or title, size, timestamps, the object id, and the **SHA-256 of the plaintext**.

The hash is what makes “the manifest says X is at this object id” checkable after the fact. An object swapped on disk fails to unwrap; one that somehow unwrapped would still fail the hash.

> **THE COST OF THE HASH, NAMED**
> The content hash is a confirmation oracle to anyone holding the manifest key: it lets them prove a *known* file is in the vault without reading the object. That party already holds every name and size in the vault, so the additional disclosure is accepted deliberately rather than overlooked.

Note titles are not path components. A title may contain anything but NUL, and a single sanctioned helper derives a filesystem-safe name from it for export.

SECTION 08

Crash safety: the intent journal

08

A vault that corrupts itself when a laptop lid closes is not a security product. Crash safety is the axis the write paths are organised around, not a repair bolted on afterwards.

8.1 The durability rule

Every durable write in the vault goes **temp → fsync → rename → parent fsync**, through one implementation, so there is one thing to audit rather than five. Files are staged at `<path>.<pid>.<ts>.<hex>.tmp`, a shape a sweep can recognise and remove without knowing which write produced it.

> **WHAT FSYNC ACTUALLY BUYS, STATED PRECISELY**
> Node exposes `fsync(2)`, which on APFS pushes data to the device but does not force the drive's own cache to flush — that is `F_FULLFSYNC`, which Node does not expose and which would make a bulk import an order of magnitude slower. So the guarantee the format is built on is: **a process crash or kill at any point leaves a state the journal can recover.** Power loss and kernel panics get best effort, the same as every other durable write in the app.

8.2 Records

Every import, export, delete and note write seals a record to `journal/<recordId>.jrn` **before** it acts, updates it as each durable stage lands, and deletes it on completion. Records use the envelope of section 7.1 under the manifest key, so a locked vault reveals nothing about what was being imported from where.

The ordering rule that makes replay decidable: a `manifest-committed` record is written *before* the commit that produces the generation it names. Replay can therefore tell which side of a commit an interruption fell on by comparing the manifest's generation against the record's.

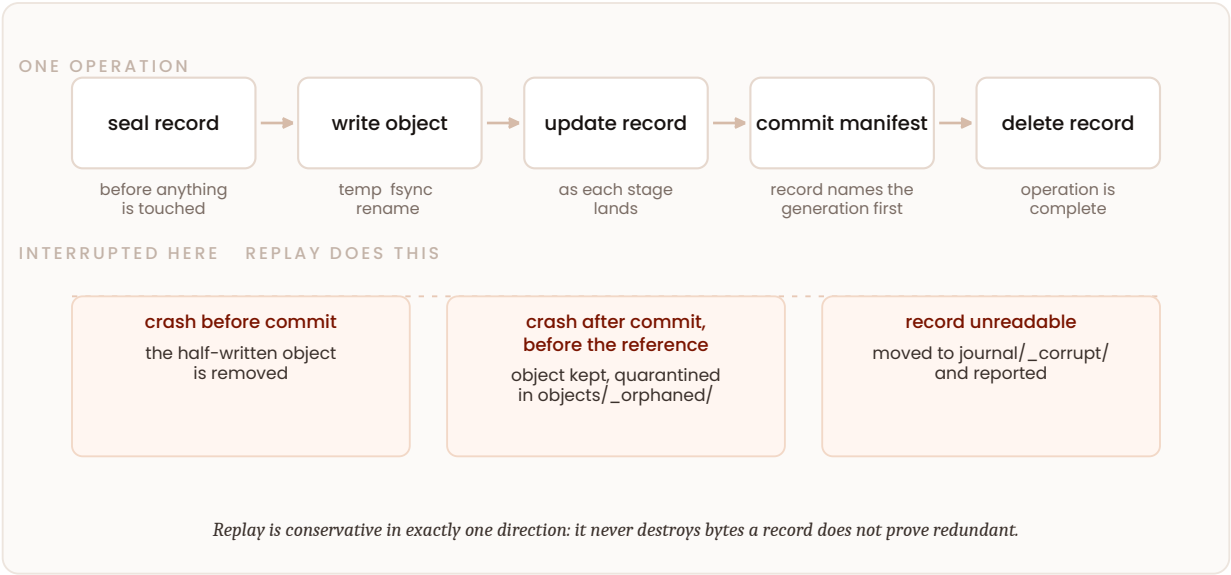


Figure 7 — The record lifecycle, and what replay does with each interruption point.

8.3 Replay

On unlock, replay examines surviving records and brings the store to a consistent state.

- Half-written objects are removed.
- Committed-but-unreferenced objects are moved to `objects/_orphaned/` and kept. A quarantined object is still decryptable; deleting it would be the one irreversible choice available, so replay does not make it.
- An import that moved a file and did not finish removing the original is surfaced to the user as a pending action, not resolved by deleting anything.
- A record that cannot be opened is moved to `journal/_corrupt/` and reported. The object it referred to is unknown by definition, and is caught by the orphan sweep.

8.4 The unclean marker and the sweep

A close that gives up waiting for in-flight work leaves `journal/.unclean`, so the next open sweeps even if the journal is empty. The marker survives **two** opens, not one: work that a close abandoned may still be running while the next open sweeps, and an open that races it can consume the journal record before that work leaves its object behind — after which nothing would ever look again.

The sweep is O(objects) and runs only when replay had something to do. Every path that can leave an object unreferenced writes a record before touching the object store, so an empty journal genuinely means no new orphan since the last sweep.

SECTION 09

While the vault is open

09

Everything above concerns a vault at rest. These are the properties that hold during the window in which it is not.

9.1 Key lifetime

Keys exist as Buffers for the duration of an unlock. On lock, the VMK copy and all three derived keys are overwritten in place rather than dropped for the garbage collector.

Overwriting in place has a consequence the implementation must respect: an operation still in flight holds references to those exact Buffers. So `close()` refuses new operations immediately, announces the locked state so nothing else can start, and zeroises only after the in-flight operations have finished — an import half-way through must land under real keys or not at all, never under a buffer zeroed beneath it. Each sealing step also re-reads the keys within its own synchronous run rather than capturing them across an `await`, and a key that has already been zeroised is detected and refused. A race therefore produces an error, not a file encrypted under zeros.

9.2 Auto-lock

The vault locks on idle timeout, on system sleep, and on screen lock. Sleep and screen lock take no flush grace: the machine is going away now, and waiting for a renderer that may never answer is worse than losing an unsaved edit.

9.3 Single-writer lease

A vault directory is claimed by a filesystem lease with a heartbeat. A second VaultSort task — another window, a scheduled job — is refused rather than allowed to interleave writes. A lease whose heartbeat has lapsed can be re-acquired, so a crashed process does not lock the vault out permanently.

9.4 Unlock rate limiting

Argon2id makes each password guess slow, but a caller free to invoke `unlock()` could still run guesses in parallel, or hammer the PRF path with a stolen credential id. The fifth consecutive failure therefore arms the first delay, and every attempt after that waits:

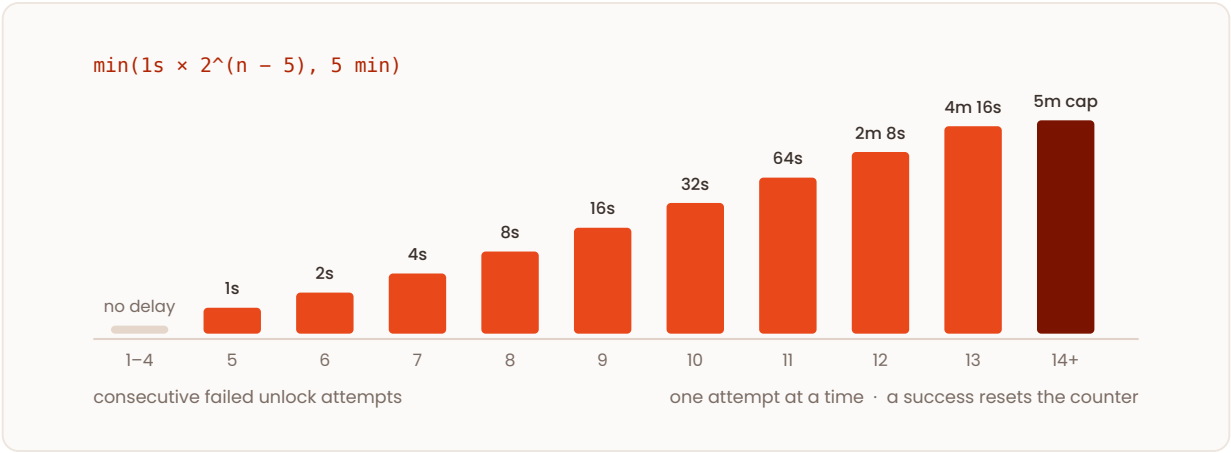


Figure 8 — Exponential backoff: the fifth consecutive failure arms a one-second wait, doubling to a five-minute cap at the fourteenth. Only one attempt runs at a time; parallel guesses would defeat the backoff entirely.

> AN HONEST SCOPE FOR THE COUNTER
> The failure counter is in-memory and per process. An on-disk counter would have to live in a file the attacker can also reset, so it would buy the appearance of a defence rather than a defence. What actually costs an offline attacker is the Argon2id floor in section 4.5.

One related ordering rule: a lock that lands while an unlock is still running wins. The store that unlock produces is closed rather than adopted, and the caller is told the attempt was cancelled — so a sleep during a multi-second derivation cannot end with an unlocked vault on a machine that has gone away.

9.5 What is logged

The Vault logs event names, stable error codes and counts — not file names, note titles or paths. Three specific properties are pinned by tests: a root the location policy refuses is logged by code and never by path; the errors the store raises carry no path; and an unexpected error's message never crosses to the renderer, which logs by channel and code instead.

> A GAP IN THE TEST COVERAGE, NAMED
> There is **no blanket test that scans the whole log** for names, titles and paths. The property is held by the call sites and by those three tests, not by a global assertion. A new log line that named a file would not be caught automatically.

Telemetry, where enabled, carries ten event names — the ten in `VAULT_EVENT_RULES` — and nothing but bucketed counts, booleans and fixed words the source itself wrote. An event name outside that union is a compile error; a property outside the event's rules is dropped at runtime rather than sent, so a caller cannot widen the payload by passing more. Counts leave bucketed to their order of magnitude: 12 becomes 10, 1247 becomes 1000.

SECTION 10

Algorithm selection

10

Nothing here is novel, and that is the point. Each choice is a standard primitive used for the job it was designed for.

CHOICE	WHY
AES-256-GCM content and manifest	Authenticated encryption with associated data, hardware-accelerated on every supported platform. Tampering aborts rather than returning garbage.
AES-256-KWP RFC 5649, key wrapping	Purpose-built for wrapping keys, carries its own integrity check, and handles the 32-byte input without padding ambiguity. A wrapped blob moved between slots fails cleanly.
HKDF-SHA-256 every derivation	Standard extract-and-expand. Distinct `info` strings give domain separation between the manifest key, the object wrap key, the header MAC key and each slot's wrap key, from one root secret.
HMAC-SHA-256 the header	The header is JSON, not a stream; a MAC over canonical bytes is the simplest thing that is correct, and it needs no nonce management.
Argon2id passwords, recovery codes	Memory-hard, resistant to both GPU and side-channel attack, and the current recommendation for password-based derivation. Parameters are stored per slot, calibrated to the machine, and bounded above and below (4.5).
Per-object keys	Make membership changes cheap (4.6), and make object substitution detectable (6.2).
A monotonic generation with an external floor	The only defence against a valid older header that a content MAC cannot give.

SECTION 11

Limits and known gaps

11

Stated plainly, because a specification that only lists strengths is not useful.

No re-keying

Format version 1 cannot rotate the vault master key. Removing a slot removes that way in, and the generation floor stops a rollback on the machine that removed it — but an attacker who captured the vault directory *and* a slot's secret before removal can still open that old copy forever. Durable revocation needs a VMK re-key and a full object re-wrap, which is deliberately not in this version.

The rollback floor is per-machine

A vault copied to a machine that has never seen it starts with no floor.

Object sizes and timings are not hidden

No padding, no cover traffic. An observer of the directory over time learns how many objects there are, roughly how large each is, and when each changed.

The content hash is a confirmation oracle

To a holder of the manifest key, as described in 7.2.

Decrypted note text cannot be wiped

A note in the editor is a JavaScript string in the renderer. Buffers are zeroised; strings cannot be. Every lock except one unmounts the editor, which bounds the exposure to heap residency until garbage collection — but a suspend that freezes the renderer before the locked state arrives can carry decrypted note text across the sleep.

Durability stops at fsync

As 8.1 says, a process crash is recoverable by design; power loss and kernel panics are best effort, because forcing the drive's own cache on every write would cost more than it buys here.

No sync, by design in this version

Each machine keeps its own vault. There is no multi-device key agreement to review, because there is no multi-device.

SECTION 12

Scope and independent verification

12

12.1 What this document does not cover

- The **V4 single-file encryption format**, which has its own document.
- The WebAuthn registration and credential-management flows, beyond what a vault slot needs from them.
- The secure-deletion engine used when an import in *move* mode removes the original. It reports the guarantee it was able to give on that particular disk; see `docs/SECURE_DELETE.md`.
- Application-level authorisation. The licence check that gates *adding* to a vault is a product decision, not a security control, and is deliberately absent from every path that opens, reads or exports — so it can never become one.

12.2 Where to check each claim

Every constant and derivation in this document appears in `src-electron/modules/vault/`, principally:

FILE	WHAT TO CHECK
`vault-format.ts`	Key hierarchy, HKDF `info` strings, slot wrap and unwrap, header validation, header MAC, generation floor check
`vault-object-format.ts`	Object layout, AAD rule, per-object KEK derivation, streaming discipline
`vault-manifest-format.ts`	Envelope, magics, AAD, manifest bounds, entry shapes
`vault-journal.ts`	Record lifecycle, replay rules, unclean marker, orphan sweep
`vault-header-floor.ts`	Rollback floor behaviour, including its behaviour when damaged
`vault-store.ts`	Atomic writes, the commit sequence, lease handling
`vault-fs.ts`	The one durable-write implementation, and what `fsync` promises here
`vault-session.ts`	Key zeroisation on lock, auto-lock triggers, the flush grace, unlock rate limiting
`vault-telemetry.ts`	The event allowlist and the bucketing

The properties this document claims are pinned by tests, indexed with the specific test names in `docs/tests/vault-security-matrix.md`. Where a property is *not* covered by an automated test, that matrix says so rather than leaving the gap implicit.

CORRECTIONS AND CHALLENGES ARE WELCOME

If something in this document does not match the source, the source is the truth and this document is the bug.

Reports of either kind are wanted — including the ones that make the list of limits in section 11 longer. Every constant and derivation named here appears in the source tree listed in section 12, and the properties claimed are indexed against the tests that pin them in docs/tests/vault-security-matrix.md.