Security Baseline

csiPass is security-sensitive firmware. The current development build is not an account-protection device.

Before real credentials are stored:

- Enable ESP32-S3 Secure Boot v2 and unique-per-device flash encryption in the release manufacturing flow. Keep development and release eFuse flows separate.
- Generate credential private keys on-device. Never log keys, PINs, hmac-secret values, decrypted resident credentials, credential IDs, OATH secrets, HOTP codes, or CTAP payloads.
- Use authenticated encryption for vault records and power-loss-safe atomic updates. An encrypted partition flag alone is not a complete vault design.
- Require a fresh physical user-presence gesture for every protected operation; ignore a button already held when the request arrives.
- Treat all USB and host fields as hostile: bound lengths, reject malformed framing and non-canonical CBOR, type-check known MakeCredential/GetAssertion members, enforce timeouts, and fuzz packet and CBOR parsers.
- HOTP secrets enter once over the management enrol command and are never returned. Codes are shown only on the trusted display after a fresh Confirm; management list/delete responses carry summaries without secrets or codes.

Verifying the on-flash format

The native vault tests use real AES-256-GCM and HMAC-SHA-256 from a test-only reference implementation, not a stand-in. That implementation is proven first against published vectors – FIPS 180-4 for SHA-256, RFC 4231 for HMAC-SHA-256, FIPS-197 for the AES-256 block, and the GCM specification for AES-256-GCM – because an oracle compared only to itself proves nothing. The same reference implementation also backs a host build of the shipped key derivation, so HKDF-SHA-256 and the PIN/UV auth protocol key schedules are checked as the production code rather than as a reimplementation of it: HKDF against the RFC 5869 vectors, AES-256-CBC against SP 800-38A, and the protocol two keys against the construction the CTAP specification describes. What that does not prove is agreement with another implementation, which only interoperability can settle.

Only the device root and the random source are simulated, since an eFuse secret cannot exist on a host. Everything downstream of them is genuine, so for a given key the host produces the bytes a device produces. Tests therefore pin the sector header, the start of the ciphertext, and the authentication tag at its documented offset. A moved field, a changed additional-authenticated-data span, or a tag written to the wrong place all surface as a mismatch rather than as a round trip that still happens to succeed.

The tag is pinned separately for a reason worth recording. It originally was not, and that gap hid a defect in the host oracle's GHASH: it ran an extra Galois field multiplication over the zero padding of a short final block, which is wrong for any payload whose length is not a whole number of blocks – every record and every PIN payload. Encryption and decryption on the host agreed with each other, so every

round trip passed; the ciphertext is counter mode and was correct either way; and the pinned bytes stopped short of the tag. The published GCM vectors in use at the time used an empty and a single-block plaintext, so they did not reach it either.

What found it was the first comparison against hardware. A lab board wrote its own sectors, the host tried to read them, and only the index superblock – alone in being an exact multiple of the block size – decrypted. The firmware was never wrong: it uses the platform's GCM. The oracle was. That is the argument for checking a real device rather than trusting a self-consistent host: a wrong implementation compared only to itself agrees with itself perfectly.

With that fixed, a board carrying one credential now reads back clean: its record, its PIN state, and its index superblock all authenticate and decode under the host format.

Parser fuzzing

Every parser that reads bytes an attacker controls is fuzzed natively: the canonical CBOR validator, the MakeCredential, GetAssertion, and credential management request parsers, and the vault sector header, record, and index codecs. The vault codecs belong on that list because a removed flash chip can be written with anything before it is put back.

The host toolchain has no AddressSanitizer, no UndefinedBehaviorSanitizer, and no libFuzzer, so the property those tools would report is checked directly instead. Each input is parsed twice, in two buffers holding identical bytes followed by different padding. A parser that reads past its declared length sees different trailing bytes and returns a different result, which makes an out-of-bounds read observable without instrumentation. Two further invariants are asserted: results must be deterministic, and every view a parser hands back must lie wholly inside the input it was given, which is how a bounded parser would otherwise leak memory to its caller.

The harness checks itself before it checks anything else. It runs its detector against a parser that deliberately reads one byte too far and against a view that deliberately points outside its input, and refuses to run if either goes unnoticed. A fuzzer that reports nothing is otherwise indistinguishable from a fuzzer that cannot see.

Seeds are deterministic and printed on failure, so any finding is reproducible. The suite runs a short campaign; the binary takes an iteration count and a seed for a longer one. What is not yet fuzzed: the CTAPHID framing layer, which is entangled with TinyUSB, and the Client PIN parser, which needs the crypto backend.

- Sign update images, enforce anti-rollback policy, and require device confirmation before enabling any management update endpoint.
- Do not claim FIDO certification, YubiKey compatibility, or production security before interoperability, side-channel, fault-injection, rollback, and recovery reviews.

The prototype confirmation input is a dedicated normally-open button between GPIO2 and GND. The firmware ignores a button held before a protected request and accepts only a fresh debounced press processed after the request enters its pending state. Input is polled before queued USB reports, so an ambiguous press at the request boundary fails closed rather than approving the new request. Pending CTAPHID requests emit user-presence keepalives, expire after 30 seconds, honor cancellation only on the active channel, and keep other channels busy. The on-board BOOT button is on strapping pin

GPIO0 and is reserved for navigation/development recovery in production builds; it must not approve protected operations there. The explicitly non-production `-lab` build duplicates confirmation on a fresh BOOT press so bench work does not require the external button. A production enclosure should protect or internally route the confirmation wiring and add a separate cancellation/navigation control if needed.

Current cryptographic implementation boundary

The development runtime now uses ESP-IDF PSA Crypto for SHA-256, HMAC-SHA-256, AES-256-CBC, ECDH P-256, on-device P-256 key generation, and signing, with fixed-buffer WebAuthn/COSE/DER formatting. A successful physical confirmation can create and use an ES256 credential. The credential collection supports at most 64 records, and holds only credentials that have to be remembered – see below. If an HMAC_UP key has already been provisioned, an explicit stable record encoding is encrypted with AES-256-GCM and written to redundant copy-on-write sectors in `secure_store`; otherwise the firmware performs no secret provisioning and stays RAM-only. This is still suitable only for interoperability testing because the release Secure Boot, flash- encryption, recovery, and provisioning profiles are not complete.

Before flash encryption is enabled, vault ciphertext is written through the raw partition API because the partition is already marked for future XTS encryption. After flash encryption is active, firmware automatically uses the encrypted partition API and the inner AES-GCM format remains unchanged.

Client PIN uses PIN/UV auth protocol 1. Only the first 16 bytes of the SHA-256 PIN digest are retained, with eight retries and a three-consecutive-failure power-cycle block. In vault mode, the PIN digest and total retry count share the authenticated encrypted storage. The PIN/UV token, ECDH private scalar, permission binding, and consecutive-failure power-cycle block remain only in internal RAM. Decrypted PIN buffers and derived secrets are explicitly zeroized. In RAM mode the entire PIN state remains volatile. PIN verification never satisfies user presence.

PSA exposes the generated private scalar only to bounded internal credential RAM and the authenticated vault encoder; it is never returned over USB or logged. Temporary key-pair records, vault keys, plaintext encodings, and raw signatures are explicitly zeroized. Replaced records are wiped before reuse.

Every mandatory per-operation buffer is statically allocated rather than placed on a task stack: the deepest startup path would otherwise hold a record payload and a decoded record at once on a 3.5 KB stack, and a stack overflow in the credential path is a security failure, not just a crash. Because those buffers now persist in static memory between operations, wiping them is part of the contract: the shared working record and the record payload buffer are zeroized on every exit path of registration, startup restore, and index rebuild, so a plaintext private scalar never outlives the operation that produced it. No credential path allocates from the heap. The release eFuse security profile must still be active before real accounts are protected.

Direct-dump protection design

The production vault uses defense in depth rather than relying on one encryption flag.

1. **Secure Boot v2** verifies the second-stage bootloader and every application image with RSA-PSS. Only project-signed code may invoke the vault derivation path.
2. **Flash Encryption in release mode** uses a unique per-device AES-XTS key held in read-protected eFuse. Application, OTA state, key partitions, and every vault data partition are encrypted at rest. Development mode is never a release configuration.
3. **An independent HMAC root key** is randomly provisioned into the vault-reserved eFuse `KEY5` block with purpose `HMAC_UP`, then read- and write-protected. The raw key is never readable by software, and firmware never substitutes another HMAC_UP block automatically.
4. **Vault key derivation** asks the HMAC peripheral for a domain-separated device root, then uses HKDF-SHA-256 with a public per-device salt and versioned `info` labels to derive record-wrapping keys. Derived material exists only briefly in internal SRAM and is explicitly zeroized. Secret material must not enter external PSRAM, logs, crash dumps, or USB buffers. A batch of operations such as the startup restore may hold the extracted root in a scope that asks the peripheral once instead of once per sector; the scope zeroizes the cached material when it ends, so the lifetime stays bounded rather than becoming permanent for the session.
5. **Per-record AEAD** uses AES-256-GCM with a fresh 96-bit random nonce. Each stored passkey has an independently protected record, so ordinary changes do not rewrite or decrypt the full collection. The whole 48-byte plaintext sector header is the additional authenticated data, binding schema version, record type, payload format, record identifier, generation, vault epoch, length, and nonce. The vault epoch is a named header field that stays zero until the eFuse-backed epoch exists; changing it makes every prior sector fail authentication, which is exactly the intended factory-reset semantics. Private keys and sensitive metadata are ciphertext; plaintext indexes contain no account or RP data.
6. **Atomic storage** gives every logical object two erase-sector copies. The new generation is encrypted, written, read back, authenticated, and compared before it becomes active; the prior generation remains recoverable. The encrypted secondary index uses its own authenticated superblock in the same two-copy form.

Index superblock

The index is one additional logical slot holding a 1552-byte encrypted payload: a 16-byte prefix followed by one 24-byte entry per credential slot. Each entry carries occupancy and discoverability flags, a reserved tombstone flag for the future authenticated delete path, the low 32 bits of the record generation, and two truncated lookup tags covering the RP ID hash and the credential ID.

The lookup tags are keyed, not raw hash prefixes. An RP ID hash is a hash of low-entropy public data, so a truncated raw hash would be a readable RP identifier anywhere index plaintext is ever exposed. Each tag is therefore an HMAC-SHA-256 under a separately derived index key, domain separated by a leading byte, and truncated to 8 bytes. A tag narrows a search but is never proof of a match: a hit must still be confirmed against the decrypted record.

The index is an optimisation and a torn-write detector, never the authority on credential content. Startup authenticates it once, compares it against the plaintext sector headers of every credential slot, and rebuilds it from the records whenever anything disagrees: an index that is absent, unreadable, or invalid; a record the index does not list; a generation mismatch; or a tombstoned entry. A slot whose sectors exist but cannot be authenticated is recorded as unreadable in the index rather than erased, so the damage is reported once and does not force a rebuild on every subsequent boot. Firmware

never erases data it cannot authenticate: the sectors stay untouched for offline analysis or a future authenticated recovery path. Exactly one state fails closed – an index entry that vouches for a record whose sectors do not exist at all, which no ordinary update or torn write can produce. A failed index write leaves records intact and degrades startup to the previous full scan rather than refusing to boot. Record writes always complete and verify before the index is updated, so a lost index update is only a stale optimisation and never a credential the host believes was rejected.

Storage schema versions are explicit. Older sectors stay readable and are rewritten in the current schema the next time their slot is updated, so a mixed-schema vault is a valid steady state and no bulk migration pass exists. A newer schema is classified before any other header field and is never read, erased, or superseded, because overwriting it would be a silent downgrade that destroys data the running image cannot represent.

Schema 3 is the first change that actually grew a stored payload: a credential record gained its `credProtect` policy and the PIN payload gained the minimum length and the policy flags. Both fields were appended rather than inserted, so the migration is a decode that reads an absent trailing field as its documented default. What had to change is the reader. The payload length declared in the header is now reported rather than compared against one compiled-in constant, because an older record is genuinely shorter than a newer one and both have to load. The length remains bounded by the workspace and remains covered by the authenticated header, so tampering with it fails the tag rather than reading past a buffer.

The credential protection policy is enforced by hiding rather than by refusing. Without user verification, a level-three credential is invisible to every assertion and a level-two credential is invisible to a discoverable assertion while remaining reachable through an allow list. Refusing would disclose that the credential exists, which is the very thing the policy is meant to prevent, so the count of matching accounts also reflects only what is visible and a continuation sequence keeps the visibility of the request that opened it.

Public capacity statistics – used, free, and unreadable slot counts, storage mode, schema and index format identities, index generation, and vault epoch – are computed on-device and may be displayed or logged. RP IDs, account names, user handles, credential IDs, and key material remain encrypted at rest and are never logged.

Flash encryption prevents a removed flash chip from revealing useful plaintext. The inner AEAD layer supplies record integrity and compartmentalization even if a storage API is misconfigured. Secure Boot prevents a simple replacement firmware from asking the HMAC peripheral to unwrap the vault.

Credentials that carry themselves

A credential the relying party did not ask to be discoverable is one the host hands back on every assertion, so the device never has to remember it. Those now travel inside their own credential ID: a 32-byte nonce, one byte of policy, and a 32-byte keyed tag over both together with the RP ID hash. Nothing is written and the count is unlimited. The vault’s 64 slots hold discoverable credentials only, which is what credential management has to enumerate.

The private key is derived on use, by rejection sampling an HMAC over the same inputs against the P-256 group order, so a derivation that landed outside the group is refused rather than reduced into it.

The tag is a keyed MAC, and both properties that matter follow from that. An ID opens only under the relying party it was made for: presenting one site's ID under another changes the MAC input, so it is refused as unknown rather than answered with a different key. And an ID opens only on the device that made it, because the keys come from the vault root – with the epoch as the HKDF salt, so a factory reset invalidates every one of them at once, including the ones the device never stored and could not enumerate.

The policy travels in the clear and is covered by the tag. `credProtect` decides whether the credential may be mentioned at all without user verification, so a device that forgot it would silently downgrade every protected credential it ever issued; rewriting the byte to claim a weaker level invalidates the tag, so the credential stops being ours rather than becoming a weaker one. Whether `hmac-secret` was requested at creation is carried for a sharper reason: without it, a relying party that never asked for the extension could obtain its secret later merely by sending salts. A flag bit this firmware never sets is refused rather than ignored.

A U2F key handle and a credential ID are the same construction under different labels, and neither opens as the other. The two protocols make different promises about presence and verification, and a value that crossed between them would arrive carrying the wrong one.

What this costs, stated rather than left to be found. A credential the device does not store is one it cannot revoke one-by-one: the only cryptographic revocation is the epoch, which revokes all of them. The private key and the credential ID still never live on flash for non-discoverable registrations.

Agreed and implemented: a site-held registration journal. Visibility without pretending these are vault credentials. See `docs/TODO.md`. In short: a separate encrypted journal (not a second credential vault) records RP ID and account for non-discoverable registrations; capacity 256; upsert by relying party; ring when full; no manual delete; cleared on erase / epoch with the derived keys. Device browse keeps resident keys and journal entries in **separate chains** (journal entered from Storage). Companion Storage must show journal occupancy; the device Ready line does not. The browse and companion surfaces may show journal rows without a PIN – the journal is closed at rest, not behind an extra gate at read time. UI must label them `site`, refuse Delete / `HOLD DEL`, and say they are held by the site.

The U2F attestation identity

U2F's registration response requires an X.509 certificate and a signature over it. There is no way to decline, as CTAP2's `none` attestation allows.

The certificate is **derived, never stored**. Its key is HKDF from the vault seed under its own label and the certificate is rebuilt on demand, so there is no first-boot ceremony, no slot, and nothing a reflash can lose – the root is in eFuse, which flashing does not touch. Deterministic ECDSA makes the rebuild byte-identical. Every field is a constant except the serial, the key, and the signature; the serial is the start of the hash of the public key so that it, too, is reproducible. The certificate is X.509 v3 with `basicConstraints CA:FALSE` marked critical, and deliberately without `keyCertSign`, which RFC 5280 section 4.2.1.9 forbids alongside a false CA boolean. Its exact bytes are pinned as a golden vector in the host tests, generated once and verified with OpenSSL.

The epoch is deliberately absent from that derivation, and this is the one place it should be. A key handle is a credential and dies with a factory reset; a certificate identifies the hardware, which a reset

does not change.

The privacy cost is real. A per-device attestation key is a cross-site tracking vector: every relying party that sees a registration sees the same unique key. FIDO's answer is a batch certificate shared by at least a hundred thousand devices, which needs a manufacturing process this project does not have. Until it does, the choice is a per-device certificate or no U2F, and the per-device one is what is shipped.

The U2F signature counter is leased. A block of thirty-two values is reserved in NVS before any of them is signed with, and a boot resumes from the reserved ceiling rather than the last value used, so power loss skips values instead of repeating them – a repeat is what a relying party reads as a cloned device, and a gap is what it is required to tolerate. A reservation that cannot be written refuses to sign. The value is public, since every signature carries it, so it is not in the vault. CTAP2 assertions still report zero, which CTAP2 permits and U2F does not.

The U2F approval screen cannot name the caller. U2F sends only the SHA-256 of the origin. The screen says a site is asking using the older protocol rather than showing a hash nobody can read or a name the device would be inventing. This is a structural weakness of the older protocol against this project's central claim, and it is stated rather than papered over. A device required to verify its user refuses every U2F command, the version query included.

Rollback boundary

AES-XTS and AEAD do not detect restoration of an older, internally valid full flash image. ESP32-S3 application anti-rollback protects firmware versions, not arbitrary data generations.

A credential can also be deleted on the device itself, without a host. That path is deliberately split across both physical inputs: BOOT selects and arms, and only the dedicated confirmation button completes the destruction, so erasing a credential needs the same trusted gesture as approving one and the navigation button can never destroy anything on its own. The armed state names the relying party, is cancelled by any short press, is cancelled by an arriving host request so a confirmation cannot be redirected away from the operation on screen, and expires on its own.

Deleting a credential and resetting the authenticator both write in a fixed order: the index tombstone reaches flash first, then the record sectors are erased, then the entry is released. Every interruption therefore lands on a state startup can reconcile. The one state the vault refuses to boot from is an index that vouches for a record whose sectors are gone, and that ordering is exactly what makes it unreachable. A reset erases storage before clearing the in-memory collection, so an interruption can never resurrect credentials the device already reported as erased.

- Use a small one-way eFuse-backed vault epoch only for rare ownership transitions such as factory reset or recovery-key rotation. csiPass reserves the low 32 bits of the user data eFuse block as a unary counter and binds its value into the authenticated header of every sector, so advancing it makes the entire stored collection permanently undecryptable. The first boot afterwards erases the superseded sectors, so the previous owner's ciphertext does not linger. Flash claiming a *later* epoch than the fuses report cannot have been produced by a one-way counter on this device, so the vault refuses to start rather than reclaiming it.
- **The device supports exactly one epoch advance.** ESP32-S3 protects every eFuse block except BLOCK0 with Reed-Solomon coding: the block and its parity are burned as a single unit, and hardware rejects any later write. BLOCK0 has no field free for project use, and `SECURE_VERSION`

there belongs to firmware anti-rollback and must not be repurposed. This is a silicon constraint, not a policy choice: a device can be securely re-homed once, and after that a factory reset can no longer be proven in hardware.
- Use an authenticated append-only generation chain and redundant superblocks for ordinary updates. The index superblock records the generation of every record slot, so restoring a single record sector to an older generation is detected and reconciled. An exact rollback of the entire flash, index included, remains undetectable.
- The authenticator reports a zero WebAuthn signature counter rather than claiming clone detection that rewritable flash cannot make rollback-safe.
- Document that strong per-write rollback resistance requires an external secure element, a trusted remote witness, or other monotonic hardware.

One gate for user verification

CTAP treats Client PIN and a built-in modality as interchangeable sources of the same fact, so the authenticator asks one question – is the user verified for this operation – rather than tracking any one method's flag. Every method reports to a single gate, which also records *which* modality answered. That distinction is not cosmetic: consuming the PIN/UV token belongs to the PIN path alone, and a built-in verifier has no token to spend.

The gate exists before any sensor does, deliberately. A fingerprint module implements a non-blocking verifier interface – begin a capture, poll, cancel – because a capture takes hundreds of milliseconds over a peripheral bus and nothing here may stall a USB callback waiting for it. Adding a modality then means implementing that interface and attaching it, not revisiting the authenticator or GetInfo: the `uv` option already follows whether a built-in method is available and enrolled.

A fingerprint touch is both a fresh physical gesture and a verification, which the current dedicated-button rule does not anticipate. Whether one touch may satisfy user presence and user verification at once is a policy decision that has to be made explicitly when the sensor arrives, not inherited by accident.

Authenticator configuration

`toggleAlwaysUv` and `setMinPINLength` are device-wide policy, authenticated by a PIN/UV token carrying a permission of their own and persisted beside the PIN verifier in the same authenticated slot.

The minimum PIN length only ever rises. A request below the current floor is refused, so an administrator's policy cannot be undone by a later, weaker request. The authenticated message is prefixed with a padding block and the command byte, so a token issued for one configuration command cannot be replayed onto another.

The policy is enforced rather than only recorded, and the enforcement is spread across the three places a policy can be evaded. A PIN below the floor is refused where it is set, not merely reported afterwards. While a change is required, no token is issued at all – the PIN is still proved first, so the refusal reveals nothing to someone who does not know it. And a change that reinstates the same PIN is refused, because it would satisfy the policy on paper and alter nothing in fact; the comparison is against the stored verifier, which is all the authenticator keeps.

To decide whether a raised floor actually invalidates the existing PIN, the authenticator records that PIN's length in code points. That is a length, not a fragment of the secret, and it is what lets a raise leave a long-enough PIN alone instead of demanding a pointless change. State stored before lengths were recorded reports zero, which is read as compliant with the default floor and non-compliant with any raised one – the safe direction.

The list of relying parties allowed to read the floor is stored as hashes of their IDs. Membership is the only question ever asked of it, and a hash answers that question while telling a reader of the stored bytes nothing. The floor itself is disclosed only to a relying party on that list: it is a property of the device's owner, not public information.

`alwaysUv` refuses the operation rather than downgrading it. A MakeCredential or a presence-taking GetAssertion that arrives without user verification is answered with a request for a token when the device has a verification method, and with a plain refusal when it has none – the difference matters to a platform deciding whether retrying is worth anything. A silent assertion is exempt because it never reaches the gesture the policy is about.

Per-credential HMAC secrets

The `hmac-secret` extension hands a relying party a symmetric secret derived inside the authenticator. Each credential carries its own 32-byte seed, generated on-device at registration and stored in the encrypted vault beside the credential. The seed is stored rather than derived from the signing key: reusing one secret for two purposes is the kind of economy that is fine until it is not.

Two secrets come from that seed under separate labels, one for verified assertions and one for unverified ones. Which one answers is decided by the UV bit the response actually carries, so an unverified assertion cannot reproduce the secret a verified one would produce. The platform's salts arrive encrypted and authenticated under a freshly agreed shared secret; a bad tag is refused before anything is decrypted.

Across a `GetNextAssertion` sequence the salts and the agreed secret are held so that each credential answers with its own secret rather than with none. They are platform data and a session key, never the authenticator's stored secret, and both are wiped when the sequence ends or is cancelled.

The lab image's log interface

The lab image enumerates as a composite device: the FIDO authenticator plus a serial port carrying its own log. That is a second interface a host can see and open, which is exactly why the production image declares none. A host can only open what the configuration descriptor offers, and production offers one thing.

The claim is absence, not unreachability. The CDC class driver is not compiled into the production image at all: the two environments have separate ESP-IDF configurations, composed at build time from `sdkconfig.base.defaults` plus, for the lab image only, `sdkconfig.lab.defaults`. This matters because unreachable code is still code in the image and still something a reviewer has to account for; the production image is 3,252 bytes smaller for it.

Interface and endpoint numbering is arranged so the authenticator is identical either way: FIDO holds interface zero and the first endpoints in both images, and the log takes what follows.

The simulated root, and why it is loud

The lab build derives the vault root from a constant compiled into the image instead of asking the HMAC peripheral. Everything downstream is genuine: the same key hierarchy, the same AES-256-GCM, the same sector layout, the same epoch binding. Only the root is not a secret.

This exists because the storage path is otherwise untestable on real hardware without an irreversible eFuse burn, and a burn is a poor thing to debug against. It is a deliberate, known-insecure key path shipped inside firmware that otherwise behaves exactly like production – which is precisely why it is announced rather than merely documented.

Three things make it impossible to mistake. The security page reports the root as `FAKE` and the level as `FAKE ROOT`. The version suffix `-lab` is drawn in the error colour on every screen, pinned by a compile-time check so the coloured span cannot drift onto part of the build number. And the posture reports a simulated root **before** it reports anything the eFuse says, because a board can hold a perfectly provisioned eFuse and still be running a build that ignores it: what protects the vault is what the code uses, not what the hardware holds.

The constant itself is a readable English sentence rather than random-looking bytes, so anyone who dumps the flash sees at once that this key was never meant to protect anything, instead of mistaking it for a real key that leaked.

But the phrase is not in this repository. It lives in `firmware/waveshare/include/lab_vault_root.hpp`, which is untracked; what is tracked is `lab_vault_root.hpp.example`, carrying the same explanation and a placeholder in place of the value. A lab build without the file does not fall back to anything – it fails to compile, and says which file to copy.

The reason is worth stating, because it is easy to read the paragraph above and conclude the value does not matter. It does. A lab board is a real board with real passkeys on it, and its flash leaves the building on occasion: a dump goes into a bug report, a board goes to a colleague, a photograph catches a screen. Anyone holding the phrase can decrypt any lab board built from it, which is exactly what `deploy/verify-vault.ps1` relies on and exactly why the phrase must not also be readable by anyone who can read a public git history. Keeping it per-developer does not make a lab build secure – nothing does – it removes the one attacker who needs no access to anything at all.

Two consequences follow. A lab image built on one machine cannot be inspected from another, because the roots differ. And rotating the phrase is deliberate destruction: a board flashed afterwards cannot read what the old phrase wrote, and factory resets on first boot.

The production build has no such path and no runtime switch into it. It does not include the file, cannot use it, and derives from hardware or refuses to start.

Confirmation on a touch panel

The baseline asks for a fresh physical gesture on an input a host cannot drive. On the boards in this family that have no second button, that input becomes the touch panel, and the properties that matter are not the ones a first reading suggests.

A dedicated region of the screen, reserved and never used for anything else, was considered and rejected. It buys nothing: a permanently drawn confirm target is habituated exactly as a permanently present button is, and it costs usable screen on a panel that has little to spare. What actually protects the gesture is unchanged from the button, and it is this: **the device draws what is being approved, and the person reads it before acting.**

Three properties carry that, and a touch build must hold all three.

The target exists only while a confirmation is pending. A stray touch on an idle screen can never approve anything, because there is nothing there to approve with. This is stronger than a button, which is always pressable.

The gesture must begin after the prompt is on screen. A finger already down when the request arrives does not count, exactly as a button already held does not count today. What the device waits for is a fresh contact, not a contact.

What is read and what is touched are the same object. The target belongs to the prompt that describes the operation, not to a fixed corner the eye has learned to skip.

Destructive operations keep their asymmetry: leaving is the easy gesture and proceeding is the deliberate one. On a button that is a short press to cancel against a press on the other button to confirm; on touch it is a dismissal against something that cannot be produced by brushing the glass.

Vault root provisioning and audit

Firmware never writes an eFuse and never provisions a secret. It reads the eFuse state, classifies it, and shows the result on the trusted display. Provisioning is a deliberate host-side operation performed by `deploy/provision-device.ps1` against the one enrolled device.

The vault root is a random 32-byte key in eFuse `BLOCK_KEY5` with purpose `HMAC_UP`. `espefuse` classifies that purpose as requiring read protection, so `burn_key` read-protects and write-protects the block unless the operator explicitly overrides it. That override must never be used here:

- A **read-protected** root can only be exercised through the HMAC peripheral. Software, including a replacement firmware, cannot extract it.
- A **software-readable** root protects nothing. Anyone able to run code on the device can read the block and decrypt the whole vault offline, while the device would still claim to be in `VAULT` mode.

Firmware therefore refuses to open a vault rooted in a readable block and shows `Insecure root` instead of degrading quietly. This is recoverable: read protection can still be applied to the existing block, so the refusal costs a provisioning step rather than the device. An absent root is a different case and stays a supported development state: the device runs RAM-only and says so.

The device classifies itself into exactly one level, shown at startup and as the first entry of the local browser:

Level	Meaning
`INSECURE`	An `HMAC_UP` root exists but software can read it. The vault refuses to open.
`DEV MODE`	No root. Credentials live in RAM and are lost on reset.
`VAULT ONLY`	A protected root, but the image and flash contents are not protected.
`PRODUCTION`	Protected root, Secure Boot, release-mode flash encryption, and a locked root eFuse.

A readable root outranks every other signal, including an otherwise fully hardened boot configuration, because it is the one state in which the device would actively misrepresent its own protection.

The provisioning tool audits before it writes, refuses to re-burn an occupied key block, resolves the enrolled hardware identity again immediately before the irreversible step, keeps espefuse's own typed confirmation as the final gate, and re-audits afterwards to prove the block ended up read-protected. It supports a rehearsal against a virtual eFuse device so the exact command and its resulting protections can be reviewed before any hardware is committed. It never enables Secure Boot or flash encryption: those belong to the final production step, after a signed rescue image and an encrypted-image flashing workflow exist.

The random key reaches `espefuse` through a file because that is the only input it accepts. The file is CSPRNG-generated under `temp/`, permission-restricted, overwritten, and deleted, and the key is never printed. A journaling or wear-levelled filesystem may still retain the overwritten block, so provisioning must be done from a trusted machine.

Production provisioning profile

- Provision unique flash-encryption and HMAC keys per device; never keep a fleet- wide decryption key.
- Burn and protect Secure Boot key digests before any user credential exists.
- Disable JTAG and USB-JTAG. Permanently disable UART ROM download mode when field recovery is provided by a signed rescue image; otherwise use Secure Download Mode with its reduced command set.
- Keep a minimal signed rescue/factory image so a broken main application can accept only authenticated recovery firmware. Secure Boot disables ROM USB-OTG update paths, but application USB remains available after a verified boot.

Recovery image

`firmware/rescue` is a separate PlatformIO project, not a build variant of the application. Flash it with `.\deploy\flash-device.ps1 -Env waveshare_esp32s3_lcd_147-rescue` after the same enrolled-identity checks as the application image; a rescue upload does not advance the application firmware build counter. It shares only presentation code – the display, the text layout, and the status LED – and links no credential, vault, eFuse, CTAP, or USB source at all. That boundary is enforced by an explicit source list rather than by convention, and it is checked by inspecting the linked symbols of the

built image rather than asserted. What the recovery image does not carry cannot be attacked in it, and what it cannot reach it cannot leak.

Two pieces of the recovery story are deliberately not built yet, and the image says so on screen rather than implying a capability it lacks:

- **Signature enforcement** requires Secure Boot v2, which is the final production-hardening step. Until then the recovery image is unsigned, so it provides isolation but no authenticity.
- **The authenticated update transport** belongs with the versioned management protocol. Until it exists, recovery means reflashing over USB, which is why the ROM download interfaces must stay enabled.

Automatic rollback is wired now so it cannot be forgotten later: the application marks itself valid only after the display, inputs, security posture, and USB transport have all initialised. An image that fails any of those never reaches that point, so the bootloader reverts to the previously working image instead of leaving an unusable authenticator installed. This path becomes live with the first over-the-air update; a directly flashed image is not pending verification and simply reports that there was nothing to validate.

- Verify eFuse state and security configuration on the device display before the vault can leave development mode.

Official ESP32-S3 references:

- https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/security/secure-boot-v2.html
- https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/security/flash-encryption.html
- https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/peripherals/hmac.html
- https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/security/security.html