DOI 10.5281/zenodo.22860069

KeyNub — Security Architecture

Document version 1.1 · Applies to firmware 1.1.x, protocol version 1, SDK version 1.1.x

Scope and product boundary, assets, threat model, environment assumptions, security objectives, the mechanisms that meet them, a cryptographic inventory, key management, per-mechanism verification status, and the limitations — stated next to the mechanisms they limit.

CONTENTS

What a Dongle Proves

A genuine KeyNub dongle with this serial is attached to this machine right now, and the session bytes came from it. Precise, and not the same as "this software is licensed".

Where the Boundary Lies

No peripheral can stop an attacker patching the application that queries it, because that happens in the application's own process. Section 9 states the limit; section 10 is the integration pattern that narrows it.

What Can Be Checked Directly

The device attests its own containment from the hardware's answer rather than its build configuration, and everything on the host side of the wire is published source: see section 12.

Document status Released. Rev. 1.1, September 2026. · DOI 10.5281/zenodo.22860069

Enquiries Report a vulnerability via the contact form at keynub.com/#contact · **www.keynub.com**

0. ABOUT THIS DOCUMENT

This describes what a KeyNub dongle protects, what it protects it against, how, and — in the same detail — what it does not protect. The intended audience is the security architect, procurement reviewer or audit function assessing a hardware licensing component.

It is structured like a Common Criteria Security Target — scope, assets, threats, assumptions, objectives, mechanisms, coverage — because that structure is familiar and complete. The claims in this document are the vendor's own.

Several of the central claims can be measured directly; section 12 sets out which, and how.

Two conventions govern the document:

1. **Every limitation is stated next to the mechanism it limits**, not collected in a footnote. If a mechanism does not help against a particular attack, that is said where the mechanism is described.
2. **Every mechanism carries its evidence.** section 11 states, per mechanism, how its status was established, so the strength of each claim is on the page.

The One-Paragraph Summary

A KeyNub dongle holds an ECC P-256 private key in a secure element that cannot export it, proves possession of that key on a fresh challenge, and carries an X.509 certificate issued by the KeyNub root CA whose public certificate is compiled into the released SDK. Sensitive operations run inside an authenticated, encrypted, replay-proof session. License records are encrypted at rest under a key derived inside the secure element. The firmware's externally reachable code — the USB stack and the protocol parser — runs in a hardware-isolated world that cannot address any key, and the device attests that isolation to the host from the hardware's own answer rather than from its build configuration. The device boots only firmware signed by KeyNub, and its debug port is permanently disabled before shipping.

The most important qualification: none of that prevents an attacker from patching the integrating application so that it stops asking. No peripheral can close that gap; section 9 and section 10 carry as much weight as the mechanisms.

1. SCOPE — WHAT IS AND IS NOT PART OF THE PRODUCT

Assessing a component requires knowing where it ends. The boundary here is unusually important, because the most likely attack falls *outside* it.

Inside the Boundary

Element	Description
The dongle	A USB-C device: an Arm Cortex-M33 microcontroller, a secure element holding the device key, and flash for license storage
Device firmware	The protocol implementation, session cryptography, license store, and update mechanism
Device identity	The per-device ECC P-256 key pair and its X.509 certificate
Wire protocol	Version 1, frozen: framing, message layer, session envelope
The host library	One core C library (`keynub_licdongle`) with a stable C ABI, plus per-language bindings
The trust anchor	The KeyNub root CA public certificate, compiled into the released library

Outside the Boundary

Element	Why it matters
The integrating application	The library is loaded into the application's own process, on the attacker's machine. Its return values are not a trust boundary.
The host operating system	Assumed hostile. See section 4.
License issuance	Which dongle is entitled to which product is the integrator's record-keeping.

Element	Why it matters
The developer master key	Held by the integrator, and rotated to their own on receipt. It authorises writes to the dongles they have issued.
The physical USB port and cable	Traffic is assumed observable and modifiable.

The first row is the consequential one. A licensing dongle is a peripheral; it cannot reach into the process that queries it. Every guarantee below is of the form *"a genuine dongle with this serial is present and these bytes came from it"* — never *"this software is licensed"*. Converting the former into the latter is an integration problem, and section 10 addresses it.

2. ASSETS

What is being protected, in rough order of consequence:

ID	Asset	Property required	If lost
A1	Device private key	Confidentiality, non-exportability	A dongle can be cloned; the whole identity guarantee fails for that unit
A2	KeyNub root CA private key	Confidentiality	Anyone can mint a dongle the shipped SDK trusts. Systemic, all units
A3	The developer master key	Confidentiality	An attacker can write licenses to any of that integrator's dongles
A4	Firmware signing key	Confidentiality	Hostile firmware boots on fielded devices
A5	License records at rest	Confidentiality, integrity, authenticity	Entitlements forged or read
A6	Session keys	Confidentiality, forward isolation	Traffic readable and forgeable for that session
A7	Application wrapping keys	Confidentiality	Data bound to the dongle becomes decryptable without it
A8	Monotonic counters	Integrity, monotonicity	Trial or activation limits reset
A9	Secure-messaging key between MCU and secure element	Confidentiality	The bus between MCU and secure element becomes readable and forgeable
A10	Device authenticity claim	Freshness	A replayed proof passes without the dongle present
A11	Derivation master	Confidentiality	Device element keys could be derived without the device
A12	Factory write-auth key	**None — published**	Nothing: it is in the SDK. A dongle is rotated to the integrator's own key on receipt, before it is written to or passed on, so no unit reaches an end customer answering to the factory key; `licd_info.writeauth_rotated` makes that checkable across a delivery

Three of these are held by KeyNub and protect every unit: the root CA key, the firmware signing key and the derivation master from which each device's element keys are derived. Section 8 describes how each is generated and kept.

3. THREAT MODEL

Threat Agents

Agent	Access	Motivation	Capability assumed
T-USER	Full physical and administrative control of the host	Use software without a license	Debugger, admin/root, disassembler, unlimited time, can replace any library
T-DEV	The above, plus hardware skills	Extract a key to clone dongles	Logic analyser, bus probing, soldering, firmware extraction attempts

Agent	Access	Motivation	Capability assumed
T-LAB	The above, plus a funded laboratory	Break the platform, not one seat	Decapsulation, fault injection, side-channel analysis, focused ion beam
T-NET	Observes or modifies USB traffic	Replay or forge licensing exchanges	Bus capture and injection
T-INSIDER	Access to KeyNub or customer signing processes	Systemic compromise	Legitimate credentials

T-USER is the realistic adversary, and is explicitly the *legitimate purchaser* of the software — not a stranger. They own the machine. The design assumes this from the start.

Against T-LAB, the device key is held in a secure element evaluated against high attack potential and certified accordingly (section 8.5). Around that, the link between microcontroller and secure element runs inside an AES-256 authenticated session with replay protection, so probing the bus yields no key material; hardware glitch detectors are armed at maximum sensitivity against clock and voltage fault injection; and both debug ports are permanently removed before a dongle leaves the factory.

Threats

ID	Threat	Agent
TH-CLONE	Duplicate a dongle by extracting its private key	T-DEV, T-LAB
TH-FORGE-CERT	Present a self-made device certificate the SDK accepts	T-USER
TH-REPLAY	Replay a recorded genuineness proof or session	T-NET
TH-MITM	Read or alter session traffic on the wire	T-NET
TH-EMULATE	Emulate a dongle in software or on other hardware	T-USER
TH-FAKE-LIB	Replace the host library with one that always says yes	T-USER
TH-PATCH	Patch the application so it stops asking	T-USER
TH-READ-STORE	Read license records out of the device's flash	T-DEV
TH-FORGE-STORE	Write or alter license records without authority	T-USER, T-DEV
TH-ROLLBACK-CTR	Restore a disk image or otherwise reset a counter	T-USER
TH-FW-BUG	Exploit a memory-safety defect in the firmware's externally reachable code	T-USER, T-DEV
TH-FW-HOSTILE	Load modified firmware onto a dongle	T-USER, T-DEV
TH-FW-DOWNGRADE	Install an older, known-weak firmware version	T-USER
TH-DEBUG	Read memory or keys through the debug port	T-DEV
TH-BUS	Probe the MCU-to-secure-element bus	T-DEV
TH-ROOT-KEY	Obtain the KeyNub root CA private key	T-INSIDER, T-LAB
TH-DEV-KEY	Obtain a customer's developer master key	T-USER, T-INSIDER

4. ASSUMPTIONS ABOUT THE OPERATING ENVIRONMENT

These are the conditions under which the claims in this document hold. Check each against the specific deployment: a false assumption invalidates whatever rests on it.

ID	Assumption
AS-HOSTILE-HOST	The host OS, its administrator and its user may all be hostile. Nothing in the host is trusted. This is an assumption in the sense of "assumed adversarial", not "assumed safe".

ID	Assumption
AS-NO-SECRETS-HOST	The integrating application ships no secret whose disclosure breaks licensing. In particular the integrator's *own* developer master key — the one they rotate to — is never shipped in an application. The **factory** write-auth key a dongle arrives with is a different object and is published; it is a starting position, not a secret.
AS-INTEGRATION	The integration follows section 10. Where a licence check reduces to one boolean, most of this document stops being relevant to that product's actual security.
AS-PHYSICAL	Attackers may possess dongles and disassemble them. Resistance to laboratory attack on the device key rests on the secure element's evaluated resistance to high attack potential (section 8.5).
AS-KEY-CUSTODY	The integrator's developer master key is protected with care proportional to its blast radius: it authorises writes to every dongle they have issued, and cannot be recovered from one.
AS-ROTATED	Dongles are rotated to the integrator's own write-auth key on receipt, before they are written to or passed on. Until that happens the unit answers to the published factory key, and `licd_info.writeauth_rotated` makes the difference checkable across a delivery.
AS-RECORDS	The integrator keeps records of which serial is entitled to what. The dongle proves identity; entitlement is known only to the issuer.
AS-SUPPLY	Dongles reach the integrator through a channel where substitution would be noticed. A delivered dongle is verifiable against the root CA on arrival — see section 12.

5. SECURITY OBJECTIVES

For the Device

ID	Objective
O-KEY-CONF	The device private key is generated on-chip and never leaves the secure element
O-IDENTITY	The device proves its identity in a way that cannot be replayed or emulated
O-CHANNEL	Sensitive traffic is confidential, integrity-protected and replay-proof
O-STORE	License records are confidential and authenticated at rest
O-ROLE	Write operations require a separate, cryptographically proven authority
O-COUNTER	Counters increase only
O-BIND	An application can bind data to one dongle, or to all of that integrator's dongles
O-CONTAIN	A defect in externally reachable firmware cannot reach a key
O-ATTEST	The device reports its own containment state from hardware evidence
O-FW-AUTH	Only KeyNub-signed firmware executes
O-FW-SAFE	A failed update cannot leave the device unusable
O-NO-DEBUG	The debug interface is permanently unavailable

For the Operating Environment

These are objectives the product **cannot** meet on the integrator's behalf. They are listed as objectives because the security of the whole depends on them as much as on the device objectives above.

ID	Objective for the integrator
OE-CRITICAL-PATH	Make dongle-derived data necessary for the application to function
OE-NO-CACHE	Do not persist decrypted material to disk
OE-MASTER-KEY	Keep the developer master key out of shipped software
OE-RECORDS	Maintain entitlement records

ID	Objective for the integrator
OE-ABSENCE	Treat dongle absence as a normal state, not an assertion

6. SECURITY MECHANISMS

6.1 Device Identity and Non-Exportability

Each dongle's ECC P-256 key pair is **generated inside the secure element** during provisioning and the private half has no read path — not through the protocol, not through the firmware, not through the debug port, and not by KeyNub. The firmware asks the secure element to perform operations with it and receives results.

The device carries an X.509 certificate issued by the KeyNub root CA, binding that public key to the device serial. The serial also appears in the USB iSerial string and in the certificate's subject `serialNumber` attribute, so the three can be cross-checked.

Addresses: O-KEY-CONF, O-IDENTITY · TH-CLONE, TH-FORGE-CERT

6.2 Genuineness Proof — Live Challenge-Response

The host sends 32 random bytes. The device returns an ECDSA P-256 signature over

```
SHA-256( "KNUB-AUTH-v1" ‖ device_serial ‖ host_challenge )
```

The host verifies the certificate against the embedded root CA, checks that the certificate's serial matches the device's reported serial, recomputes the digest, and verifies the signature against the certificate's public key.

Freshness comes from the host's challenge, so a recorded exchange is worthless. The domain-separation prefix prevents a signature produced for one purpose being accepted for another.

Addresses: O-IDENTITY · TH-REPLAY, TH-EMULATE, TH-FORGE-CERT

> **Limitation.** This proves a genuine dongle is present. It does not prove the application received a truthful answer about it — that depends on the integrity of the application's own process, which is TH-PATCH and TH-FAKE-LIB.

6.3 The Trust Anchor Requires No Configuration

The KeyNub root CA's public certificate is **compiled into the released library**. Verification works out of the box; the integrating application supplies, manages and ships no trust root. An override exists for vendor tooling that must verify dongles issued under a different CA, and a library built with no root configured fails closed (`LICD_E_CERT_INVALID`, "no trust root configured").

Addresses: TH-FORGE-CERT

6.4 Authenticated Encrypted Session

Sensitive commands run inside a session established by an authenticated key exchange: both sides contribute an **ephemeral** key, and the device signs the transcript with its *certified* key. The signature makes the handshake an identity proof; the ephemeral agreement gives it forward secrecy.

Step	Construction
Agreement	ECDH P-256, host ephemeral key against a **device ephemeral key generated inside the secure element for this session**
Transcript	`th = SHA-256("KNUB-SESSION-v1" ‖ serial ‖ host_eph_pub ‖ device_eph_pub ‖ host_nonce ‖ device_nonce)`
Identity proof	`ECDSA-P256(device certified key, th)` — signed inside the secure element
Derivation	`HKDF-SHA256(salt = host_nonce ‖ device_nonce, ikm = Z, info = "KNUB-KEYS-v1" ‖ th, L = 96)` → three 32-byte keys: host→device, device→host, confirmation
Mutual confirmation	`HMAC-SHA256(k_confirm, 0x01 ‖ th)` from the device, `0x02 ‖ th` from the host

Step	Construction
Record protection	AES-256-GCM, separate key **and** separate counter per direction
Nonce	Deterministic from direction and counter, so a nonce is never reused under a key

The host verifies that signature with the public key **taken from the verified certificate**, which proves the device holds the certified private key: genuineness and channel establishment remain the same act, and the session stays cryptographically bound to that identity.

The session keys have forward secrecy. No long-term key takes part in the key agreement, so an attacker who later obtains the device's identity key — by any means, including a successful attack on the secure element — still cannot decrypt traffic recorded earlier.

Replay and reordering are structurally excluded. Each direction's counter must match exactly; any counter or authentication-tag mismatch **destroys the session** rather than returning an error, so the host must re-handshake. There is no window in which a rejected message can be retried.

Addresses: O-CHANNEL, O-IDENTITY · TH-MITM, TH-REPLAY, TH-EMULATE

6.5 Two Roles, and the Write Role Must Be Proven

A session begins with the **read role**. Elevation to the **write role** requires an ECDSA signature by the developer master key over `SHA-256("KNUB-WRITEAUTH-v1" ‖ th)`, verified inside the secure element against the public key the device holds in manufacturing flash. A dongle ships holding the published factory key; the integrator replaces it with their own on receipt (section 8.4).

Binding the signature to the session transcript `th` makes an elevation **unreplayable to any other session** — a captured elevation cannot be reused.

Read role: read records, list, read counters, wrap/unwrap application data. Write role: write and erase records, increment counters, reboot. So a deployed application never needs the master key, and an attacker who captures everything a deployed application does still cannot write a license.

Addresses: O-ROLE · TH-FORGE-STORE

> **Limitation.** This defends the *device*. Shipping the master key inside the application bypasses the mechanism entirely, and the compromise then reaches every dongle that key authorises rather than one seat. Hence OE-MASTER-KEY.

6.6 License Records, Encrypted at Rest

Records are named blobs in a 1 MB partition, each encrypted with AES-256-GCM under a key **derived inside the secure element** from a storage master secret that never leaves it. Reading the flash directly — by desoldering it, or through a firmware defect that reaches it — yields ciphertext whose key is not in the MCU.

Writes are atomic: a record is replaced via a temporary object and a rename, so a torn write or a power failure mid-write leaves the previous version intact. The authentication tag means a modified record fails to decrypt rather than decrypting to something attacker-chosen.

Addresses: O-STORE · TH-READ-STORE, TH-FORGE-STORE

6.7 Monotonic Counters

Two hardware monotonic counters in the secure element. They increase and cannot be decremented, reset, or rolled back by restoring a disk image, reinstalling, or manipulating the host clock — because they are not on the host at all. Increment requires the write role.

Addresses: O-COUNTER · TH-ROLLBACK-CTR

6.8 Application Data Binding

This mechanism changes what an attacker must *produce*, not what they must bypass.

The SDK generates a random AES-256 key, encrypts the payload **on the host** (so payload size is unconstrained by USB), and has the dongle wrap the key. Unwrapping requires the dongle. The wrapping key is derived inside the secure element from

either:

Scope	Wrapping key from	Decryptable by
`DEVICE`	The device-unique secret	Only that one physical dongle
`DEVELOPER`	The integrator-shared secret	Any of that integrator's dongles

Both require only the read role, so deployed applications use them freely with no master key present.

Why this matters more than the genuineness check. A fake library can return "yes" to a boolean. It cannot return **plaintext it does not have**. Where the data an application needs — coefficients, calibration tables, rules, model parameters, content, a parsed entitlement payload — is wrapped to the dongle, its absence leaves nothing to run rather than a check to bypass.

Addresses: O-BIND · TH-FAKE-LIB and TH-PATCH — **partially, and only this mechanism does**

6.9 Firmware Containment — Three Hardware-Enforced Layers

Everything above concerns attacks on cryptography or integration. The remaining question is what happens if the dongle's own firmware has a defect.

Its precise form is this: every byte sent to the device arrives as a USB packet and is reassembled into a protocol frame. A USB stack and a parser are where a memory-safety defect is most likely to live, and they are the **only** part of the firmware an attacker can reach. So that is the code that is fenced off.

Runs isolated	Stays protected
USB stack, HID transport, frame reassembly, the main loop	device key, session keys, at-rest key, secure element and its bus, license storage, firmware update, watchdog, OTP, DMA

Three layers, each enforced by hardware rather than convention:

Layer 1 — TrustZone world separation. The exposed code runs in the Non-secure world. Separation is enforced by the Security Attribution Unit, and the configuration is **fail-closed by construction**: any address not explicitly granted is Secure, so the configuration enumerates what is *given away*. Peripherals are divided by a separate access-control block, which denies the Non-secure world the secure element's bus, flash programming, OTP and DMA. What the isolated world needs from the protected side it requests through a small, fixed set of gateway calls — seven, declared in one place so the boundary can be read in a few minutes. The two that accept a pointer check it **twice**: once by asking the hardware whether the Non-secure world could have reached that range itself, and again arithmetically against the permitted buffer. A single mistake in either check is not sufficient to hand out protected memory.

Layer 2 — the USB stack exists only in the isolated world. The Secure image contains **no USB stack at all** — not a disabled one, not an unreferenced one. This is verifiable by symbol inspection of the two binaries.

Layer 3 — unprivileged, with W^X. Within the Non-secure world the transport runs unprivileged: its RAM is execute-never, its code read-only, and the memory protection unit governing it is itself unreachable from that privilege level. So a memory-safety defect in frame reassembly cannot be turned into injected shellcode.

The bound on a defect in the parsing code is therefore: **code execution in a world that can talk to USB and cannot read a key.**

Addresses: O-CONTAIN · TH-FW-BUG

> **Limitations.**
>
> - This does **nothing** for TH-PATCH or TH-FAKE-LIB. Those happen on the host, in the application's own process, where the dongle has no reach. Device containment is about the dongle keeping its own promise — *this key never leaves the secure element* — even if its firmware is defective.
> - An attacker with code execution in the isolated world has everything that world has: the USB interface. Layer 3 raises the cost of *getting* code execution; it does not change what is reachable once obtained.
> - The isolated world runs the main loop, so it can **starve its own transport** — a denial of service it inherently possesses by doing nothing. It buys no read access to anything. A dongle that stops responding is a support incident, not a licensing bypass.

6.10 The Device Attests Its Own Containment

Section 6.9 does not have to be taken on trust. `licd_get_info` reports `licd_info.isolated` (`Isolated`, `IsIsolated`, depending on the binding).

The firmware does **not** derive that flag from its build configuration. At boot it uses the processor's address-attribution instruction to ask the hardware how an access *from the isolated world* would be attributed — for the session keys, the at-rest key, the protected stack, the manufacturing region holding the secure-element I/O protection key, and the license store — and confirms every one comes back **refused**, while the isolated world's own working memory and code come back **granted**. Both halves are required: a configuration that refused the isolated world everything would pass the first five checks and be a device that cannot work.

Two properties that instruction cannot express — that the isolated world's memory is execute-never, and that it truly runs unprivileged — are established instead by deliberate-fault probes during development, since the only proof that an operation faults is to attempt it.

A build whose containment is not what it claims **does not set the flag**. Anything that is not a dongle reports `false`, because the flag is only ever set by asking real hardware: a check gated on it cannot be satisfied by something standing in for a device.

Addresses: O-ATTEST

6.11 Verified Boot

The microcontroller's immutable boot ROM verifies an ECDSA signature over the firmware image before executing it. The hash of the signing public key is held in one-time programmable memory, which is **one-way per bit** — it can be written once at manufacture and never altered or erased afterwards, by us or anyone else. With signature enforcement enabled, an unsigned or modified image **does not boot at all**.

The partition table is a standalone structure rather than part of any image, so its signature is enforced by a separate one-time flag, which is set at manufacture. Without it, an attacker able to write flash could not execute their own code — image signatures still gate that — but could relocate the storage regions or force a downgrade to an older signed image.

Two boot keys are programmed at manufacture: an **active** key and an **escrow** key, both marked valid before lockdown. Programming a second key later would require physical access that the lockdown sequence removes, so manufacture is the only time it can be done; the escrow key is what keeps fielded units updatable if the active key is ever retired.

Addresses: O-FW-AUTH · TH-FW-HOSTILE

> **Limitation.** Image hash verification is enforced independently of signature enforcement, so a corrupted image is refused either way. Signature enforcement establishes *who* built an image, not *which version* it is: installing an older KeyNub-signed image is refused to anyone who cannot reach the write role, which is the same gate that installing a current one has to pass (section 6.12).

6.12 Firmware Update, and Why a Failed Update Cannot Disable a Dongle

Installing firmware is the most privileged thing a host can ask for, so **every update command requires an authenticated session with the write role** (section 6.5) — as does the only other route into the bootloader. On a unit whose write-auth key has been rotated to the integrator's own, that key is the sole means of installing any image, current or older.

Updates use an A/B slot pair with a trial-then-commit sequence:

1. Only the **inactive** slot is written; the running image is never touched.
2. The boot ROM verifies the new slot's signature and hash before it runs.
3. The device reboots into the new image **on trial**, under a watchdog.
4. The new image commits itself only after proving it works — secure element reachable, storage mounted, a session achievable — not only that it started.
5. If it does not commit, the next boot **reverts to the previous image with no host involvement**.

Step 4 is stricter than the platform's default and step 5 is the reason the design is safe: on a sealed dongle with no recovery interface, an update that leaves the active slot invalid would be unrecoverable. A trial image that fails its self-test starves its watchdog so the revert arrives in seconds.

Addresses: O-FW-SAFE

> **Limitation.** A successful commit severs the partner slot: A/B protects against **a bad update**, not against later corruption of a good one.

6.13 Debug Lockdown, Bus Protection and Physical Hardening

Mechanism	Effect
Debug disable (one-time)	The SWD debug port is **permanently** removed. This is the step that protects the MCU's copy of the secure-messaging key.
Secure debug disable (one-time)	Closes the separate authenticated-debug path
Authenticated secure-element bus	The MCU↔secure-element link runs inside an AES-256 secure-messaging session carrying a transaction identifier and a command counter, so probing the bus yields no plaintext key material or command stream, and injected or replayed commands are rejected
Glitch detectors (one-time)	Hardware detection of clock and voltage fault-injection attempts, armed at maximum sensitivity

These are programmed **last** in the manufacturing sequence: debug disable removes the diagnostic path, so it follows validation. All are one-way, so a shipped dongle cannot be returned to a debuggable state.

Addresses: O-NO-DEBUG · TH-DEBUG, TH-BUS

> **Limitation.** These protect the microcontroller and the bus between it and the secure element. Resistance to laboratory attack on the device key itself is a property of the secure element (section 8.5).

6.14 Attack Surface Reduction

Minor measures, recorded for completeness:

- **The HID descriptor contains no keyboard, mouse or consumer usages.** A licensing dongle that could type is a licensing dongle that could be repurposed.
- **Factory provisioning commands are rejected once the unit is personalised.** Two mechanisms, of which only one is absolute. The gate the firmware applies is a marker written to manufacturing flash — a flag the firmware honours, not a fuse. What is absolute lies underneath it: the secure element has no delete, format or factory-reset command at all, and the provisioning credential is destroyed at the end of the factory flow, so the privileged element operations those commands would reach require an authentication that no longer exists anywhere.
- **The raw random-number diagnostic is refused on provisioned units.** It is unauthenticated attack surface with no purpose on a finished product.
- **Reboot commands require the write role**, so they are no easier to reach than a license write.
- **Framing errors produce no response at all** — malformed input gets silence, not an error oracle.
- **The message size limit is fixed and enforced** before reassembly allocates anything.

7. CRYPTOGRAPHIC INVENTORY

The complete algorithm set, in the form CRA and SBOM processes require.

Purpose	Algorithm	Parameters	Where executed
Device identity	ECDSA	P-256 (secp256r1), SHA-256	Secure element
Certificate chain	ECDSA	P-256, SHA-256	Host (verification)
Session key agreement	ECDH	P-256, ephemeral both sides	Secure element (device side)
Key derivation	HKDF	SHA-256, 96-byte output	MCU
Session confirmation	HMAC	SHA-256	MCU

Purpose	Algorithm	Parameters	Where executed
Session encryption	AES-GCM	256-bit, 96-bit deterministic nonce, 128-bit tag	MCU
Records at rest	AES-GCM	256-bit	MCU, key derived in secure element
Application wrapping	AES-GCM	256-bit, random nonce, scope as additional data	MCU, key derived in secure element
Wrapping key derivation	HKDF-Expand	SHA-256, domain-separated	Secure element
Random numbers	Hardware TRNG	—	Secure element
Firmware signature	ECDSA	secp256k1, SHA-256 (boot ROM requirement)	Boot ROM (verification)
Secure-element bus	AES-256 authenticated secure messaging (CMAC + CBC), replay-protected	—	Secure element / MCU

Every derivation is **domain-separated by a versioned ASCII label** (`KNUB-AUTH-v1`, `KNUB-SESSION-v1`, `KNUB-KEYS-v1`, `KNUB-WRITEAUTH-v1`, `KNUB-APPWRAP-KEY-v1`, `KNUB-APPWRAP-v1`), so a value produced for one purpose cannot be accepted for another, and a future version can change a construction without colliding with the old one.

Host-side cryptography is Mbed TLS, statically linked, at a pinned version.

8. KEY MANAGEMENT AND LIFECYCLE

8.1 The Device Key

Generated **inside the element during personalisation**, never exported, and no protocol command, firmware path or debug path can read it. The key entry is created unreplaceable, and the element has no delete, format or factory reset, so the identity is fixed for the life of the device. It cannot be re-provisioned, so a stolen dongle cannot be re-identified either.

8.2 The Root CA — The Systemic Asset

The KeyNub root CA signs every device certificate. Its handling:

Property	Value
Algorithm	ECDSA P-256
Generation	Inside a hardware security module, non-exportable, verified by a failed read attempt
Hierarchy	Single tier — devices are issued directly, and the root forbids intermediates by constraint
Authorisation	M-of-N quorum, minimum 2-of-3; no single custodian can sign alone
Backup	M-of-N shares, separately stored, restore-tested
Ceremony	Witnessed, logged, signed; the record is available to reviewers under NDA
Validity	No well-defined expiry

Why the certificates do not expire. Certificate lifetimes are a revocation tool in a PKI that can reissue. This one cannot: provisioning is one-way, so an expiry could not enable reissue — only a failure. A dated certificate would mean a dongle with a stop date, causing verification to begin failing in the field on hardware that is otherwise perfectly good, with nothing the customer could do. For a product whose entire premise is working without anyone's infrastructure staying up, that is the wrong failure mode.

Why there is no device-side revocation, for the same underlying reasons. Three independent ones, none of which a certificate format or a wire protocol can address: the device has **no clock**, so validity windows and short-lived credentials cannot be evaluated on it; the product is designed to work with **no reachable infrastructure**, which is what a revocation list or responder would require; and the verifier runs **inside the integrating application on the attacker's machine**, so a check

performed there can be skipped — the same reason section 9 says patching that application is not prevented. A refusal only holds where the attacker cannot reach it, which is the integrator's own entitlement records. That is where a compromised serial is refused.

If the root key were exposed, anyone could mint a certificate the shipped SDK trusts, and for the reasons just given the certificates already issued could not be withdrawn. Recovery would require a new root, an SDK release and new dongles. This is the single worst outcome in the threat model, and the custody arrangements above follow from it.

8.3 The Firmware Signing Key

Held by KeyNub, with an escrow key programmed into every unit at manufacture (section 6.11). Two of four key slots are spent this way, leaving room for one rotation with an escrow pair.

8.3a The Derivation Master

One symmetric secret, held by KeyNub, from which every device's element keys are derived: `HKDF(master, "KNUB:<purpose>:<device UID>")` yields the provisioning master, the runtime AppKey, the storage key and the application wrapping keys. Derivation lets an interrupted personalisation be resumed — the provisioning master can be re-derived from the master and the UID — and that recoverability ends at the crypto-shred.

Integrators are separated inside the derivation: the integrator's identifier is part of every derivation's context. So one integrator's keys — including the `DEVELOPER` application-wrapping key (section 6.8), the one derivation not bound to the UID and therefore shared across that integrator's own dongles — are unique to them. An integrator only ever receives their own derived keys and never the master, so they cannot derive another's.

It is held with the same custody as the root CA key.

8.4 The Developer Master Key

Generated and held by the integrator. It authorises license writes to their own dongles. KeyNub never holds it and cannot recover it. It belongs in the license-issuing tool and must not ship inside applications (OE-MASTER-KEY, AS-NO-SECRETS-HOST).

8.5 On the Secure Element's Certification

The secure element holding the device key holds a **Common Criteria certificate**:

> **EAL6 augmented (ASE_TSS.2), CC 3.1 Revision 5**, issued under a European national scheme by an accredited evaluation laboratory.

The details of that certificate:

- **EAL6 includes AVA_VAN.5**, the highest vulnerability-analysis level, meaning the part was assessed against attackers with high attack potential. The evaluation records 19 evaluator-weeks of penetration testing, 40 % of it perturbation attacks and 55 % side-channel, with no exploitable vulnerabilities found.
- **The Target of Evaluation is the whole product** — the hardware together with its complete fixed software: boot code, firmware, crypto library and operating system. This matters more than the assurance level. Many secure elements are certified only up to their operating system, leaving the application layer that implements the callable API outside the certified boundary. Here there is no such layer and no such gap: the evaluated security functionality explicitly includes the crypto API providing AES, ECDSA, ECDH, SHA, HMAC and HKDF — the same primitives section 7 lists.
- **The Security Target claims strict conformance to BSI-CC-PP-0084-2014**, the standard security-IC protection profile.

The certificate and its evaluation report are public documents in the Common Criteria portal. The certificate reference is available on request.

9. LIMITATIONS AND NON-GOALS

What KeyNub Does Not Do

Attack	Status
Patching the integrating application	**Not prevented.** Mitigated only by section 6.8.
Substituting a fake host library	**Not prevented.** Mitigated only by section 6.8.
Debugging or instrumenting the application process	Not prevented; out of scope
Reverse-engineering the application	Not prevented; the SDK is not an obfuscator, packer or anti-debug layer
Sharing one dongle across machines sequentially	Not prevented by the device; entitlement records govern this
Revoking a compromised dongle	Not provided device-side. Entitlement records govern this — see section 8.2 for why a device-side mechanism would not work here
Denial of service by unplugging the dongle	Not prevented, and not treated as an attack

The first two rows are why this product ships with a normative integration guide. They are properties of the integration, not of the dongle, and no device-side mechanism can change that.

What KeyNub Is Not

- **Not a licence-management system.** It is an authentication and secure-storage device. An entitlement server, a floating-licence broker or usage reporting can be built on the SDK; KeyNub develops such components as contract work.
- **Not an obfuscator or anti-tamper layer for application code.**
- **Not a medical, automotive or functional-safety certified component.** It can be used within such systems as SOUP (software of unknown provenance, in the sense of IEC 62304), subject to the integrator's own qualification process.

10. INTEGRATION OBLIGATIONS

A dongle can be flawless and the product around it still trivially bypassed. The full treatment is the integration security guide , which is normative.

The anti-pattern, in any language:

```
if (dongle.VerifyGenuine().IsGenuine)   // <-- one branch, one byte to flip
    Application.Run();
```

Two attacks defeat it and neither touches the cryptography: invert the branch, or replace the library with one that always agrees. Both work because the *value* of the check is one bit.

The pattern: make the dongle hold something the application needs. Wrap data the application cannot compute — configuration, calibration, rules, model parameters, content, a parsed entitlement payload — and decrypt it at the point of use. A fake library must then produce plaintext it does not have.

In priority order:

1. **Put dongle-derived data on the critical path.** If the program still runs correctly with every SDK call stubbed out, nothing else on this list matters.
2. **Decrypt late, repeatedly, and where the work happens.** One decrypt at startup is one place to attack.
3. **Never cache plaintext to disk.** A cache is a licence-free copy.
4. **Bind counters to entitlements that are enforced.**
5. **Verify serials against issuance records.**
6. **Treat absence as a normal state.**

7. **Keep the developer master key out of shipped software.**

11. VERIFICATION STATUS

Each mechanism carries the strongest evidence currently available for it.

Legend — *HW*: demonstrated on physical hardware. *Auto*: covered by automated tests.

On the three containment layers specifically. A dongle reports the result rather than the intent: `licd_info.isolated` (section 6.10) is set only where the firmware confirmed the isolation against the hardware at boot, so that flag — not this table — is the authoritative answer for any unit in hand.

Mechanism	Status	Evidence
Device key non-exportability	HW	Property of the secure element; no read path exists in protocol or firmware
Genuineness proof	HW, Auto	Verified against real device certificates and an embedded root; independent reference verifier
Certificate chain verification	Auto	The full chain verification is exercised against a real device certificate and the embedded root
Session handshake and encryption	HW, Auto	Verified on hardware; golden test vectors shared between firmware, host SDK and an independent reference implementation
Replay rejection	Auto	Counter and tag mismatch paths tested, including session destruction
Write-role elevation	HW	Verified on hardware, including refusal without elevation
Records at rest	HW, Auto	Round-tripped on hardware; fuzzed on the host
Monotonic counters	HW	Read and increment verified on hardware
Application data binding	HW, Auto	Both scopes verified on hardware, including cross-device refusal
Containment layer 1 (TrustZone split)	HW	Attribution check confirms refusal for keys, storage and protected stack; gateway refuses Secure pointers under negative probing; peripheral denial confirmed by deliberate fault
Containment layer 2 (USB isolated)	HW	Full protocol operation with the transport isolated; Secure image contains zero USB-stack symbols
Containment layer 3 (unprivileged, W^X)	HW	Attribution check plus deliberate-fault probes for execute-never memory and privilege
Containment attestation flag	HW	Reported by the device from the hardware's own attribution answer
Verified boot	HW	Signed image boots; **unsigned image refused and does not boot at all**; corrupt image refused and the good slot chosen
Partition-table signature enforcement	HW	Enforced on production units, which boot with the table signature required; the negative case (an unsigned table accepted without the flag) was established separately
A/B update, commit and automatic revert	HW	Switch, commit and unaided revert all verified, in both directions, including under verified boot
Debug disable and glitch detectors	HW	Both written on production units and read back from every redundant copy; each unit then completes the full functional suite with the debug port gone and the detectors at maximum sensitivity
Isolated-world containment on sealed units	HW	Verified on sealed production units, including a multi-thousand-operation soak

Summary of That Table

The cryptographic and containment mechanisms are demonstrated on hardware, and so are the manufacturing lockdown steps. Production units go through the full production sequence — provisioned, then sealed with the partition-table signature enforced, secure boot on, the glitch detectors at maximum sensitivity, the debug port removed and the USB recovery route closed — and afterwards complete the same functional suite they passed beforehand, unchanged. Every

one-way write is read back from each of its redundant copies before the sequence continues.

Each unit's own values are recorded in a per-device manufacturing record, kept for every device and available on request. That record is the authoritative statement for a unit in hand, because a sealed dongle has no route left by which its one-time memory could be read again.

How That Evidence Is Obtained

Activity	Coverage
Automated test coverage	Both sides of the protocol, with the host SDK exercised on Windows, Linux and macOS
Fuzzing	Coverage-guided, on both sides — framing, reassembly, message parsing and record storage on the device; response parsing, hostile device responses and application decryption on the host
Cross-implementation vectors	Firmware, host SDK and an independent reference implementation are held to shared golden vectors, so the three cannot diverge unnoticed
Hardware-in-the-loop	A physical suite against real dongles, plus a long-running soak comparing every response and every stored record
Negative testing	Boundary refusals are established by attempting them from the attacker's side, so a refusal that stopped working would be detected
Production verification	Every unit is functionally validated before the irreversible lockdown steps and again afterwards, and its one-time values are read back from each redundant copy as they are written
Vulnerability disclosure	`SECURITY.md`, published with the SDK

A substitute for the host library can return whatever answer an application asks for. That is the point section 6.8 makes: a library that answers the API convincingly is not difficult to build, so an integration that a substitute can satisfy is an integration an attacker can satisfy. A dongle cannot be substituted the same way: without the certified key there is no valid genuineness proof and no session.

12. INDEPENDENTLY VERIFIABLE CLAIMS

Several properties can be confirmed without relying on this document. In descending order of value:

1. **Ask the dongle whether it is contained.** `licd_info.isolated` is derived from the hardware's own attribution answer, not from a build flag (section 6.10).
2. **Read the source of everything on the host side of the wire.** The bindings, samples and C ABI header are published under Apache-2.0 at github.com/AB-KeyNub/KeyNub-SDK. The licence covers the published source; the prebuilt native libraries are under separate binary terms.
3. **Verify a shipped dongle** against the root CA compiled into the library, before trusting the supply chain (AS-SUPPLY).
4. **Watch the USB traffic.** The protocol is designed to be observed: the handshake, the counters and the authenticated ciphertext are all visible on the wire, and no key material is.
5. **Confirm the isolated world really is isolated.** Send malformed and oversized frames, then observe that identity and storage operations continue to work correctly.
6. **Stub every SDK call and see whether the product still works.** This tests the property most likely to be weak, and it concerns the integration, not the device; section 10 addresses it.
7. **Request the artefacts held by the vendor**: the trust-root ceremony record and the per-unit manufacturing record. The secure element's assurance status needs no request; it is stated in full in section 8.5.

13. REFERENCES AND DOCUMENT CONTROL

Document	Contents
Integration security guide	**Normative** integration guidance — the expansion of section 10; also in the SDK repository as `docs/integration-security.md`
`SECURITY.md`	Vulnerability disclosure, published with the SDK
`licdongle.h`	The C ABI, including `licd_info.isolated`
Datasheet	One-page specification summary

Available on request: the trust-root ceremony record, per-unit manufacturing records, and the firmware's internal design documentation.

Contact: `SECURITY.md` *for security correspondence, or keynub.com/#contact for commercial and review enquiries.*