ZestSSH

SECURITY WHITEPAPER

Security architecture for an SSH client that can't read your data.

A technical overview of how ZestSSH encrypts, synchronizes, and protects your SSH credentials — so that even a full compromise of our infrastructure leaves your data cryptographically unreadable.

DOCUMENT	ZestSSH Security Whitepaper v1.0
PUBLISHED	April 2026
APPLIES TO	ZestSSH v1.4.x and later
PUBLISHER	Affluent Labs
CONTACT	security@zestssh.com

Contents

16 SECTIONS · APPROXIMATELY 25 MINUTES

01

Executive summary

ZestSSH is a cross-platform SSH client for Android, iOS, Windows, macOS, and Linux. It lets users connect to remote servers, manage SSH keys, run port forwarding, and synchronize their entire configuration across devices through an encrypted feature called **Strongbox**.

This whitepaper documents how ZestSSH handles cryptographic operations: how credentials are encrypted before they leave a user's device, how synchronization occurs without the server ever seeing plaintext, how local data is protected on each platform, and how the Automation Engine safely exposes an execution path to external apps without giving them unrestricted access.

The document is intended for security reviewers, compliance teams, and technically-oriented users who want to verify the claims made on zestssh.com/security. Nothing here is proprietary — every primitive is a standard, well-audited algorithm. The design is publicly documented because a secret security design is not a secure one.

Core claims

CLAIM 01

Zero-knowledge sync

The Strongbox sync server stores only opaque ciphertext. It never receives your password, your keys, or your plaintext data — not in aggregate, not under any circumstance.

CLAIM 02

No custom cryptography

Every cryptographic primitive is an industry standard: AES-256-GCM, Argon2id, HKDF-SHA256, ChaCha20-Poly1305. No proprietary algorithms. No shortcuts.

CLAIM 03

Zero telemetry

No analytics. No crash reports. No usage tracking. ZestSSH has no idea how you use the app, which servers you connect to, or what commands you run.

Summary of guarantees

The following properties are mathematical consequences of the architecture, not policy statements. An attacker who fully compromised ZestSSH infrastructure would still face the cost of breaking AES-256-GCM and Argon2id per user.

- ✓ Your sync password is never transmitted.
- ✓ Server sees only encrypted blobs, wrapped keys, and salts.
- ✓ Changing your password does not require re-encrypting data.
- ✓ Tampering with server-stored ciphertext is detectable on decryption.
- ✓ Local database and OS-native secure storage are both required to read on-device data.
- ✗ We cannot recover your data if you lose both your password and your recovery key.
- ✗ We cannot assist an attacker who steals your session token in deleting your data.

02

Design principles

ZestSSH's security design follows a handful of rules. They are not novel. They are the basic rules any serious modern E2EE system follows, and they are listed here so reviewers can check the rest of the document against them.

Principle 1 — Encrypt on the device, always

Any data that leaves a user's device — to our sync server, to a backup file, to a cloud provider — is encrypted on-device first, before any network call. There is no mode in which ZestSSH transmits plaintext credentials, connection details, or private keys.

Principle 2 — The server is untrusted

We treat our own server as hostile infrastructure. Its compromise must not produce a cryptographically-meaningful leak. The server's role is reduced to three functions: store opaque ciphertext, hand it back on authenticated request, and resolve push/pull conflicts via monotonic versioning. It holds no secret material capable of decrypting anything.

Principle 3 — Use boring cryptography

Every cryptographic primitive used in ZestSSH is a published, peer-reviewed standard with production implementations in audited libraries. We do not roll our own crypto. We do not combine primitives in unusual ways. We do not skip authentication tags to save bytes. The design is boring, which is the single most important property a security design can have.

Principle 4 — Separate keys by domain

Keys that serve different purposes are derived with domain separation. A verification hash used to prove knowledge of a password to the server is derived with HKDF using a different domain string than the key used to encrypt data. Intercepting one does not help an attacker reconstruct the other.

Principle 5 — Defense in depth on the local device

Local data is protected by two independent layers: the ZestSSH database is encrypted at rest with SQLCipher, and the database's master key is itself held in the operating system's native secure storage (Keychain, Keystore, DPAPI, libsecret). Both layers must be defeated for local data to be readable.

Principle 6 — No telemetry means no telemetry

ZestSSH contains zero analytics, crash reporting, or usage tracking. Not opt-in. Not anonymized. Not "privacy-preserving." None. This constrains what we can ever know about our users, which is the point.

03

Threat model

A security system is only as meaningful as the threats it claims to defeat. This section lists the adversaries ZestSSH is designed to withstand, and — importantly — the ones it is not.

Adversaries in scope

A. Compromised sync server

An attacker obtains full read/write access to the Strongbox sync server: the database, file storage, and operator credentials. They can view every encrypted blob, every salt, every verification hash, and every wrapped DEK.

Outcome: the attacker holds opaque ciphertext. To recover any user's data, they must break AES-256-GCM (infeasible) or brute-force the Argon2id-derived key for a specific user (infeasible at the configured parameters). Per-user effort; no amortization across users.

B. Network adversary (active or passive)

An attacker sits between the user's device and our servers. They can observe, modify, drop, or replay traffic.

Outcome: the attacker sees TLS-protected traffic. If TLS is defeated (e.g., a rogue certificate authority), they observe already-encrypted blobs. Certificate pinning prevents rogue-CA attacks against our sync endpoints. They cannot decrypt blobs; they cannot forge verification hashes without the MEK.

C. Lost or stolen device (locked)

An attacker steals a device that has ZestSSH installed. The device is powered on but locked behind the OS's biometric or PIN protection.

Outcome: the attacker cannot access the app's data without first defeating OS lock. ZestSSH's own biometric/PIN lock adds a second gate. The database remains encrypted at rest, and the OS-native secure storage holding its key remains inaccessible while the device is locked.

D. Malicious third-party app (on the user's device)

Another app on the user's device tries to read ZestSSH data, trigger automations, or exfiltrate SSH keys.

Outcome: ZestSSH stores secrets in OS-native secure storage with app-scoped access. On Android, private keys sit behind Keystore + app sandbox. On iOS, Keychain entries are scoped to the app bundle identifier. Automations require an API key which itself sits in secure storage and cannot be read by other apps.

E. SSH server with changed host key

A user connects to a server whose host key has changed — possibly a legitimate re-provisioning, possibly an active man-in-the-middle attack.

Outcome: ZestSSH uses Trust-On-First-Use. A changed host key produces an explicit warning, not silent acceptance. Automations fail closed on host key mismatch.

Adversaries out of scope

Compromised user device (unlocked, attacker has physical access)

If an attacker has unlocked access to a user's device, no user-space application can provide meaningful protection. The attacker can read clipboard, screen, and memory of running apps. This is a general limitation of running on consumer operating systems.

Compromised user password AND compromised recovery key

If both the user's password and their recovery key are known to an attacker, the architecture provides no defense — they are, by design, the two valid decryption paths. Password strength and recovery key storage are user responsibilities.

Supply-chain attacks on dependencies

ZestSSH relies on underlying libraries — Flutter, dartssh2/zest_ssh_core, SQLCipher, OS secure storage APIs. A malicious modification to these libraries is outside ZestSSH's direct control, although we pin dependency versions and review updates.

Quantum adversaries

ZestSSH does not currently use post-quantum-resistant primitives. Harvest-now-decrypt-later attacks are a concern the industry is tracking. We will transition to post-quantum algorithms as they mature and are standardized.

04

Cryptographic primitives

Every cryptographic operation in ZestSSH uses an algorithm from this list. No custom constructions, no "ZestSSH-flavored" variants.

Primitive	Algorithm	Purpose and properties
Symmetric AEAD	`AES-256-GCM`	Authenticated encryption for all at-rest data: sync blobs, backups, DEK wrapping. Provides confidentiality and integrity with built-in MAC.
Password-based KDF	`Argon2id`	Memory-hard key derivation. Resists GPU and ASIC brute-force. OWASP-recommended for password hashing and KDF.
Key derivation	`HKDF-SHA256`	Domain-separated derivation of verification keys from the MEK. Ensures distinct contexts produce cryptographically independent keys.
Hash	`SHA-256`	Verification hash computation, API key hashing for storage comparison.
Local DB encryption	`SQLCipher`	Transparent database encryption using AES-256-CBC with HMAC-SHA512 page authentication. Industry-standard for local encrypted SQLite.
Transport	`TLS 1.2+`	Certificate-pinned TLS for all sync server communication. Protects metadata and transport, not data confidentiality (that's handled separately by AES-256-GCM).
SSH AEAD	`ChaCha20-Poly1305`	Recommended cipher for SSH transport layer. Fast in software, no timing-channel concerns, built-in authentication.
SSH key exchange	`curve25519-sha256`	Modern elliptic-curve Diffie-Hellman for SSH session key establishment. Fast and sidechannel-resistant.
SSH host/client keys	`Ed25519`	Modern signature scheme. Short keys, fast verification, no weak-curve concerns.

WHY NO CUSTOM CRYPTOGRAPHY

Every widely-exploited cryptographic vulnerability of the last two decades has come from one of: implementing a standard primitive incorrectly, combining primitives in an unreviewed way, or rolling a new primitive without peer review. We avoid all three by using well-audited library implementations of well-specified standard algorithms.

05

Key hierarchy and derivation

Strongbox uses a three-layer key hierarchy with indirection through a randomly-generated Data Encryption Key (DEK). This section describes each key, how it is derived, and what it protects.

The three keys

Key	Full name	Role
DEK	Data Encryption Key	Random 256-bit key, generated once on-device during sync setup. Encrypts all actual user data. Never changes over the lifetime of a sync account. Never transmitted in plaintext.
MEK	Master Encryption Key	Derived from the user's sync password via Argon2id. Used to wrap (encrypt) the DEK. Also used to derive the verification hash that authenticates sync requests.
KEK	Key Encryption Key	Derived from the user's recovery key via Argon2id with a distinct salt. Used to wrap the DEK independently of the password, providing an alternative decryption path if the password is lost.

Argon2id parameters

Both the MEK and KEK are derived with Argon2id using the following parameters. These are conservative defaults that exceed OWASP's minimum recommendations and are forward-compatible via version bumping.

MEMORY COST

`65,536 KB` (64 MB)

ITERATIONS

`3` passes

PARALLELISM

`4` lanes

OUTPUT LENGTH

`32` bytes (256 bits)

SALT

`32` random bytes per user, via `Random.secure()`

VERSION BYTE

`argon2Version = 1` (future-compatible)

The 64 MB memory cost forces an attacker to allocate 64 MB per brute-force attempt. On a modern GPU, this reduces parallelism per card by several orders of magnitude compared to memory-light hash functions. The iteration and parallelism parameters add further linear-time cost.

PARAMETER VERSIONING

Argon2id parameters are stored alongside the encrypted blob on the server as a version number. If we raise the memory cost in a future release, existing users can still derive their keys using the parameters active when their blob was created. Re-derivation with new parameters happens on the next sync push, invisibly.

The hierarchy, visually

Why indirection through a DEK

Encrypting user data directly with a password-derived key would force a full re-encryption of every byte whenever the user changes their password. For users with large sync profiles this would be slow and risky — a failure mid-migration leaves data in an inconsistent state.

By wrapping a random DEK with the password-derived MEK, password changes become a single operation: re-derive a new MEK from the new password, re-wrap the same DEK. The actual data blob is untouched. Fast, atomic, low-risk.

The recovery key provides a second, independent wrap of the same DEK. Using the recovery key is not a "bypass" — it is a fully cryptographic decryption path that never depended on the password in the first place.

06

Cloud sync architecture

Strongbox is ZestSSH's cross-device synchronization feature. This section describes what happens when a user presses "sync" — from the plaintext object graph in the app's memory to the encrypted blob at rest on our servers.

What gets synced

- ✓ SSH connection profiles (hostname, port, user, auth preferences)
- ✓ SSH identities including private keys
- ✓ Stored passwords associated with identities
- ✓ Known-hosts entries (host key fingerprints)
- ✓ Connection groups and tags
- ✓ Command snippets
- ✓ Port-forwarding rules

Application settings (theme, font size, UI preferences) are not synced. They are device-local because they often reflect per-device ergonomics rather than shared state.

The encryption flow

Pushing data to the server happens in six steps:

1. **Serialize** — the application's sync object graph is serialized to JSON.
2. **Compress** — the JSON is zlib-compressed. This reduces blob size (which is useful for mobile sync) and also flattens entropy characteristics of the plaintext prior to encryption.
3. **Generate IV** — a cryptographically-secure 12-byte random IV is generated via `Random.secure()`. IVs are never reused across encryptions with the same key.

4. **Encrypt** — AES-256-GCM encrypts the compressed plaintext using the DEK. The output includes a 16-byte authentication tag.
5. **Package** — the output is concatenated: `salt(32) + IV(12) + tag(16) + ciphertext`.
6. **Transmit** — the package is POSTed to the sync server over certificate-pinned TLS along with the user's verification hash (see below).

The decryption flow

Pulling and decrypting is the reverse:

1. **Pull** — the app requests the current blob. The server responds with the package plus metadata (current version, Argon2id parameters, wrapped DEK).
2. **Parse** — the package is split into its four components (salt, IV, tag, ciphertext).
3. **Derive MEK** — the user's password plus the salt plus the Argon2id parameters are fed into Argon2id to reproduce the MEK.
4. **Unwrap DEK** — the server-stored `wrappedDekPassword` is decrypted using the MEK, yielding the DEK.
5. **Decrypt** — AES-256-GCM decrypts the ciphertext using the DEK and verifies the authentication tag. Mismatched tag produces `SyncDecryptionException`, which the app handles as "wrong password or corrupted blob."
6. **Decompress** — zlib-decompress.
7. **Deserialize** — parse JSON back into the application's object graph.

Verification hash (authentication without revealing the key)

How does the server verify that a user requesting their data actually holds the correct password, without ever receiving that password? Via a domain-separated verification hash.

1. HKDF-SHA256 is applied to the MEK using the domain string `zestssh-sync-verification`.
2. The HKDF output is SHA-256 hashed.
3. The resulting hash is sent to the server as an auth token.

The server stores this hash at account creation. On subsequent requests, the client recomputes it from the current session's MEK and the server compares. An attacker who intercepts the verification hash cannot derive the MEK — HKDF is a one-way function, and domain separation ensures that even a successful HKDF attack on the verification context would not help compromise any other key derived from the same MEK.

Recovery-key verification uses a parallel domain: `zestssh-recovery-verification`. The two hashes are cryptographically independent.

DEK wrapping format

The wrapped DEK, stored server-side in two forms (password-wrapped and recovery-wrapped), is a fixed 60-byte structure:

```
wrappedDEK = nonce(12) + gcm_tag(16) + ciphertext(32)
           = 60 bytes total
```

Wrapping uses AES-256-GCM with the MEK (or KEK) as the wrapping key and a fresh 12-byte nonce per wrap. Rewrapping on password change generates a new nonce.

Conflict resolution

The server tracks a monotonically-increasing version counter per user. Every push includes an `expected_version` matching the last version the client pulled. If another device has pushed in the interim, the server returns `SyncConflictException` with the current server version. The client must pull, merge locally, and retry with the new version.

This is deliberately simple. Last-writer-wins semantics with explicit conflict detection is well-understood and avoids the complexity class of operational-transform or CRDT systems. For the scale of data ZestSSH syncs, merging is fast and predictable.

What the server sees and stores

Data	Stored?	Notes
Encrypted blob	Yes	Opaque ciphertext. No structure visible.
Salt (password)	Yes	32 random bytes. Needed to re-derive MEK.
Salt (recovery)	Yes	32 random bytes. Needed to re-derive KEK.
Argon2id parameters	Yes	Memory, iterations, parallelism, version.
Wrapped DEK (password)	Yes	Useless without MEK.
Wrapped DEK (recovery)	Yes	Useless without KEK.
Verification hash	Yes	Auth only; HKDF-derived, not the MEK.
Version counter	Yes	Monotonic. For conflict resolution.
User password	**Never**	Not in any form.
Recovery key	**Never**	Generated on device, shown once.
MEK / KEK / DEK	**Never**	All live on-device only.
Plaintext sync data	**Never**	Encrypted before transmission.

07

Encrypted backups (.zest)

ZestSSH's manual backup format, `.zest`, uses the same cryptographic primitives as Strongbox sync. Users can export their configuration to a local file, store it wherever they like — including third-party cloud services — and restore it on another device.

File format

The `.zest` format is a binary container with a fixed header and a variable payload:

Offset	Size	Content
`0`	4 bytes	Magic bytes: `ZEST` (ASCII)
`4`	1 byte	Version byte (currently `1`)
`5`	variable	Encrypted payload

The encrypted payload for version 1 follows the same structure as Strongbox sync blobs:

```
payload = salt(32) + IV(12) + gcm_tag(16) + ciphertext
```

There is no plaintext export option. Every backup file is encrypted. A user-accessible plaintext dump of SSH credentials and private keys would be a catastrophic security failure by design — if the device is shared, scanned by another app, or uploaded to a cloud service, the data would be immediately compromised.

Password requirements

Backup passwords must be at least 12 characters. This minimum is enforced at the input layer — short passwords are rejected before any encryption occurs. The 12-character floor reflects the reality that backup files land in user-accessible storage (Downloads folder, Files app), where an offline attacker who obtains the file has unlimited time to attack.

We recommend using a high-entropy passphrase. Short numeric PINs, dictionary words, and reused passwords are all vulnerable to offline brute force, regardless of Argon2id's cost parameters, because password entropy is the limiting factor.

What a backup contains

Data category	Included
SSH connection profiles	Yes
SSH identities and private key material (encrypted)	Yes
Stored passwords	Yes
Known-hosts database	Yes
Connection groups and tags	Yes
Snippets	Yes
Port-forwarding rules	Yes
Session transcripts	No (too large, device-local)
Automation audit logs	No (device-local)
App settings and theme preferences	No (device-ergonomic)

Import behavior

Imports are **additive merges** — existing records are updated in place, new records are inserted, and nothing local is deleted unless it conflicts with an imported record of the same identifier. This prevents accidental data loss from importing an older backup.

Identity secrets (private keys, stored passwords) are restored to OS-native secure storage only after the database transaction commits successfully. This prevents orphaned secrets in secure storage if the import fails mid-operation.

Import safety limits

50 MB IMPORT CEILING

Backup files larger than 50 MB are rejected before any content is loaded into memory. Malicious or corrupted `.zest` files claiming enormous sizes cannot crash the app with an out-of-memory error. For context, a typical ZestSSH configuration backup is under 100 KB even with hundreds of connections.

Auto-backups

ZestSSH creates automatic backups when the app backgrounds after data modification. These are saved to local app storage only — they are never uploaded, transmitted, or shared. Retention is capped at three auto-backup files; the oldest is deleted automatically as new ones are created. This provides a short recovery window for accidental changes while bounding disk usage.

Auto-backups use the same encryption as manual backups. Their password is derived from the user's sync password (if sync is enabled) or a device-scoped key (if not).

08

SSH protocol support

This section documents the SSH protocol primitives ZestSSH supports on the wire for connecting to remote servers. These are separate from — and used alongside — the Strongbox encryption described earlier.

Key exchange algorithms

ZestSSH offers the following KEX algorithms during SSH session establishment, in order of preference:

Algorithm	Notes
`curve25519-sha256`	Modern elliptic curve. Recommended default.
`[email protected]`	Identical to above, alternate identifier.
`ecdh-sha2-nistp521`	NIST P-521 curve.
`ecdh-sha2-nistp384`	NIST P-384 curve.
`ecdh-sha2-nistp256`	NIST P-256 curve.
`diffie-hellman-group-exchange-sha256`	DH group exchange with SHA-256.
`diffie-hellman-group16-sha512`	4096-bit DH group with SHA-512.
`diffie-hellman-group14-sha256`	2048-bit DH group with SHA-256.
`diffie-hellman-group14-sha1`	Legacy fallback for older servers only.

Ciphers (transport encryption)

Cipher	Notes
`[email protected]`	Recommended. AEAD, fast, side-channel resistant.
`[email protected]`	AES-256 in GCM mode. AEAD.
`[email protected]`	AES-128 in GCM mode. AEAD.
`aes256-ctr`	AES-256 CTR. Requires separate MAC.
`aes192-ctr`	AES-192 CTR.
`aes128-ctr`	AES-128 CTR.

MAC algorithms

MACs are used with non-AEAD ciphers (CTR mode). AEAD ciphers — GCM and ChaCha20-Poly1305 — carry their own integrity tag and do not use a separate MAC.

Algorithm	Notes
`[email protected]`	Encrypt-then-MAC with SHA-256. Recommended.
`[email protected]`	Encrypt-then-MAC with SHA-512.
`hmac-sha2-256`	Standard HMAC-SHA-256.
`hmac-sha2-512`	Standard HMAC-SHA-512.
`hmac-sha1`	Legacy. For older server compatibility only.

Host key types

ZestSSH verifies server host keys against a Trust-On-First-Use (TOFU) known-hosts database. On first connection to a server, the host key is recorded; subsequent connections verify the stored fingerprint against the presented key. A mismatch produces an explicit warning and requires user confirmation before proceeding.

Type	Notes
`ssh-ed25519`	Modern, fast, secure. Recommended.
`ecdsa-sha2-nistp256`	NIST P-256.
`ecdsa-sha2-nistp384`	NIST P-384.
`ecdsa-sha2-nistp521`	NIST P-521.
`rsa-sha2-512`	RSA with SHA-512 signature.
`rsa-sha2-256`	RSA with SHA-256 signature.
`ssh-rsa`	Legacy RSA with SHA-1. Older servers only.

Client key types

Keys that ZestSSH can generate on-device or import from external formats:

Type	Can generate	Can import
`Ed25519`	Yes	OpenSSH PEM
`ECDSA` (P-256/384/521)	Yes	OpenSSH PEM
`RSA 4096`	Yes	OpenSSH PEM, PuTTY `.ppk`
`RSA 2048`	Yes	OpenSSH PEM, PuTTY `.ppk`
`FIDO2 hardware keys`	Yes (per device)	Device-bound

09

Local data protection

On-device data protection in ZestSSH uses two independent encryption layers. Both must be defeated for an attacker with file-system access to read any credential, key, or connection detail.

Layer 1 — OS-native secure storage

SSH private keys, stored passwords, Strongbox sync keys, and API keys for the Automation Engine all live in the operating system's secure storage facility. The specific backend per platform:

Platform	Backend
iOS, iPadOS	Keychain (hardware-backed when Secure Enclave is available)
macOS	Keychain
Android	Android Keystore (hardware-backed on devices with TEE/StrongBox)
Windows	DPAPI (Data Protection API, user-scoped)
Linux	libsecret (via Secret Service API, typically GNOME Keyring or KWallet)

These systems are scoped to ZestSSH's app identifier. Another app on the same device cannot read ZestSSH's entries. Unlock requires the device's user authentication (biometric or PIN/password), which gates both OS-level access and ZestSSH's entries.

Layer 2 — Local database (SQLCipher)

ZestSSH's local database — which contains connection profiles, known hosts, snippets, port forwarding rules, and metadata about keys — is encrypted at rest using SQLCipher. SQLCipher provides transparent AES-256-CBC encryption with per-page HMAC-SHA512 authentication, so tampering with the database file is detectable.

SQLCipher's master key is held in OS-native secure storage (layer 1). It is never written to disk in plaintext, never printed to logs, and never transmitted. Opening the database requires layer 1 to be unlocked first.

BOTH LAYERS MUST FAIL

An attacker with only raw file-system access sees an encrypted SQLCipher file and no way to open it. An attacker who extracts the SQLCipher key but not the file has a key to nothing. Both layers must be compromised for local data to be exposed.

App-level lock (PIN and biometric)

Above the OS and database layers, ZestSSH offers an optional app-level lock. When enabled, a PIN or biometric check is required on every app launch and after the app has been backgrounded for a configurable interval. This defends against the scenario where the device is unlocked but the user walks away from it.

The app-level lock does not replace OS-level protection — it supplements it. A device that is not locked at the OS level still has its data at rest encrypted, because SQLCipher and secure storage are always on.

10

Automation security

ZestSSH's Automation Engine lets external apps — Tasker, MacroDroid, iOS Shortcuts, Siri, or any app capable of URL invocation — trigger pre-configured SSH commands on saved connections. Because this feature exposes a remote-execution surface, it is built with multiple layers of defense to keep power scoped to what the user has explicitly allowed.

Eight layers of defense

1. Default OFF

The Automation Engine must be explicitly enabled in Settings. Enabling it shows a security warning describing what automation can and cannot do. No automation runs while the engine is disabled, and disabling it while a trigger is pending cancels the trigger.

2. API key authentication

Every automation request requires a valid API key. Keys are managed GitHub-style: each key has a human-readable label, creation date, last-used timestamp, and optional expiration (30, 60, 90 days, or custom). Keys can be revoked individually without affecting other keys.

3. Encrypted API key storage

API keys are stored in OS-native secure storage alongside SSH credentials. They are never written to the app database in plaintext. They are never logged.

4. Constant-time comparison

API key validation uses constant-time equality over the SHA-256 hashes of the submitted key and stored key. This defeats timing side channels that could otherwise let an attacker learn key prefixes by observing response time differences.

5. Rate limiting

Failed API key attempts trigger exponential lockout. An attacker cannot brute-force enumerate keys in practice — lockout cascades quickly render repeated guesses infeasible.

6. Credential isolation

The triggering app never gains direct access to SSH credentials. An external app can invoke a pre-configured automation — say, "run the `deploy` snippet on my `prod-web-01` host" — but it cannot read, export, or exfiltrate the SSH key or password that automation uses. The credentials remain inside ZestSSH's secure storage; only the output of the SSH command is returned.

7. Host key verification

Automations use the same TOFU known-hosts verification as manual connections. A changed host key fails the automation closed. There is no "trust on use" flag that bypasses verification for automations.

8. No shell injection

Commands are passed directly to the SSH protocol channel — not interpolated into a shell string on the client side. Automation inputs cannot escape into unintended commands through metacharacter injection at the client. The remote shell's own parsing rules still apply to the command string as sent, but the client does not construct command strings by concatenation.

Audit trail

Every automation run is logged locally: timestamp, API key label used, command, connection, exit code, and output. These logs never leave the device and are viewable in Settings → Audit Logs. There is no remote audit destination; the logs exist for the user's own forensic review after a suspected credential compromise.

What automation cannot do

- ✗ Execute arbitrary shell commands not pre-configured in the automation library.
- ✗ Bypass host key verification on unknown or changed servers.
- ✗ Access credentials for servers not in the user's saved connections.

✗ Run while the Automation Engine is disabled.

✗ Be enabled silently — every activation requires explicit user action in Settings.

11

Transport and network security

Transport security protects data in motion between the user's device and ZestSSH's sync server. It is distinct from — and layered underneath — the Strongbox end-to-end encryption already described.

TLS 1.2+ with certificate pinning

All sync API calls use TLS 1.2 or later with certificate validation. The sync client uses a pinned HTTP client (`PinnedHttpClient`) that compares the server's certificate against a known set of acceptable certificates or their public-key hashes.

Pinning prevents a hostile certificate authority from issuing a rogue certificate that would let a network-level adversary perform a man-in-the-middle attack. Even if the device's certificate store is compromised, ZestSSH does not trust any certificate that does not match the pinned set.

Why transport security matters less than you'd think

An intuitive question: if data is already encrypted with AES-256-GCM using a key the server never sees, why bother with TLS at all?

Two reasons. First, TLS protects metadata — request patterns, access frequency, approximate payload sizes. Unencrypted HTTP would reveal these to any network observer. Second, TLS protects the verification hash used to authenticate sync requests. An attacker who captures the verification hash cannot decrypt data, but could potentially replay sync requests — TLS prevents that at the transport layer.

The important property is that compromising TLS is not catastrophic. An attacker who defeats TLS on a specific connection observes already-encrypted blobs. Data confidentiality does not depend on transport confidentiality.

Metadata minimization

The sync protocol is designed to reveal as little metadata as possible. Blob sizes are compressed (hiding exact plaintext sizes), and all sync users' blobs appear identically structured. The server cannot distinguish between a user with 5 connections and a user with 500 based on blob structure alone, though aggregate size is necessarily visible.

12

Account recovery

ZestSSH's recovery model is deliberately simple and deliberately unforgiving. It consists of a single mechanism — the recovery key — generated at sync setup and shown exactly once.

Recovery key format

During sync setup, ZestSSH generates a recovery key in the following format:

```
ZEST-XXXX-XXXX-XXXX-XXXX-XXXX
```

Each block is a random alphanumeric string. The full key represents 128+ bits of entropy, sufficient to make brute-force infeasible even if an attacker obtains the recovery-wrapped DEK.

The recovery key is generated on-device using `Random.secure()`. It is displayed to the user exactly once — at sync setup. It is never transmitted to ZestSSH's servers. The salt used to derive the KEK is stored server-side (it has to be, to let the client re-derive the KEK on recovery), but the recovery key itself is not.

How recovery works

1. The user has forgotten their sync password. They sign in to ZestSSH on a fresh device and select "I forgot my password."
2. ZestSSH fetches the user's salt_recovery and Argon2id parameters from the server.
3. The user enters their recovery key.
4. Argon2id derives the KEK from recovery key + salt.
5. The server-stored wrappedDEK_recovery is decrypted using the KEK, yielding the DEK.
6. The DEK decrypts the sync blob — data access restored.
7. The user is prompted to set a new sync password. A new MEK is derived from it, and wrappedDEK_password is rewritten with the new wrap.

Recovery does not re-encrypt data. Only the wrapping of the DEK under the MEK changes. The blob itself is untouched.

What happens if both are lost

UNRECOVERABLE BY DESIGN

If a user loses both their sync password and their recovery key, their data is permanently unrecoverable. There is no backdoor, no master key, no customer-support override. Our infrastructure holds only ciphertext and salts — without a user's password or their recovery key, we have no material capable of decrypting anything.

This is not a gap in the design — it is the design. The same property that prevents an attacker from decrypting your data after a server breach also prevents us from decrypting it at your request.

The user-facing implication: save the recovery key somewhere you can access but an attacker can't. A password manager, a printed copy in a physical safe, a trusted encrypted note — wherever you keep things you cannot afford to lose.

Account deletion protection

Destructive operations — deleting sync data, deleting the account — require proof that the requester holds the sync password, not merely a valid session token. Specifically, the verification hash (derived from the current session's MEK) must match the server's stored value.

This defeats a specific attack: an adversary who steals a user's session token (e.g., through device compromise or a network flaw in a third-party library) cannot maliciously wipe the user's data to cause harm or extort them. They would still need the sync password, which is never in transit and is held only in-memory on the user's unlocked device.

13

Privacy and telemetry

ZestSSH contains no analytics. No crash reporting. No usage telemetry. No opt-in tracking. No anonymized metrics. The absence of telemetry is not a privacy feature — it is the baseline.

What we do not collect

- ✗ How often you open the app.
- ✗ What features you use.
- ✗ Which servers you connect to.
- ✗ What commands you run.
- ✗ How long your sessions last.
- ✗ Your IP address (beyond transient TLS termination).
- ✗ Your device model, OS version, or locale.
- ✗ Crash stack traces.
- ✗ Performance metrics.
- ✗ A/B test bucket assignments.

Third-party services in the app

The only third-party service that ever receives data from ZestSSH is **RevenueCat**, used for purchase receipt verification. The data sent to RevenueCat consists solely of:

- An anonymous app-user ID (opaque UUID, not tied to any real identity)

- The purchase receipt token from the respective platform store (Google Play, Apple App Store)

No other third-party SDK is embedded. No Firebase. No Crashlytics. No Sentry. No Mixpanel. No Amplitude. No Google Analytics. No Meta Pixel. None.

Why this matters

An SSH client has access to your servers. The bar for trusting one should be high. We think the appropriate bar is: you install ZestSSH, and we have no more information about you or your infrastructure than we did before you installed it. Anything less than that is a compromise we have chosen not to make.

14

What ZestSSH cannot do

This list exists to make our constraints explicit. These are properties of the system, not choices we can reverse with a policy change. They are why the architecture is designed the way it is.

We cannot read your encrypted sync data

Our server holds an AES-256-GCM ciphertext. The key is derived from your password, which we never see. There is no copy of the key, plaintext, or derivation anywhere in our infrastructure.

We cannot decrypt your local backups

Backups are encrypted with a password you choose. We do not receive the password or the backup file. Even if we did receive a backup, we would hold opaque ciphertext.

We cannot recover your password or your recovery key

Neither secret is stored on our servers in any form. Password is not even in transit. Recovery key is generated on-device and shown once.

We cannot trigger automations on your behalf

API keys live in your device's secure storage. Our servers do not hold them, cannot derive them, and have no authenticated path into your Automation Engine.

We cannot selectively modify your sync data

AES-256-GCM's authentication tag detects any modification. A partial or targeted edit by us would produce a blob that fails decryption on your device — you would see it immediately, not silently accept tampered data.

We cannot delete your data without proof you hold the password

Destructive operations require the verification hash, which requires the MEK, which requires the password. A stolen session token is insufficient.

> **MATHEMATICS, NOT POLICY**
>
> These are not promises. They are mathematical consequences of how the system is built. A policy change by Affluent Labs cannot grant us capabilities we have deliberately architected away from. If that ever changes, it will require a new major version with a different security model and explicit user consent — not a silent update.

15

Responsible disclosure

ZestSSH is built by one person. Security reports are taken seriously and responded to personally, not routed through a ticketing system or a bug bounty platform.

How to report

Email security@zestssh.com. Include:

- A description of the vulnerability
- Reproduction steps, ideally with a proof-of-concept
- Affected versions, platforms, and any relevant environment details
- Your preferred contact method for follow-up

If the issue is particularly sensitive, request a PGP key and we will exchange one for encrypted communication.

What happens next

1. **Acknowledgment** within 72 hours of receipt.
2. **Triage** within 7 days — assessment of severity and exploitability.
3. **Fix development** proportional to severity. Critical issues take priority over any other work.
4. **Coordinated disclosure** — we agree on a disclosure date before a fix ships, typically 14-90 days depending on severity and user exposure.
5. **Credit** in release notes and on this whitepaper's acknowledgments (if you want it — many researchers prefer to stay anonymous).

What's in scope

✓ The ZestSSH mobile and desktop applications

- ✓ The Strongbox sync server and its API
- ✓ The `zestssh.com` and `docs.zestssh.com` websites
- ✓ The Automation API
- ✓ Cryptographic design issues in this document

What's out of scope

- ✗ Third-party dependencies (report upstream; notify us so we can pin around the fix)
- ✗ Social engineering attacks against the developer
- ✗ Physical attacks against the developer's hardware
- ✗ Issues requiring a compromised user device as a precondition
- ✗ Self-XSS and other issues requiring a user to paste attacker-supplied input into privileged contexts

Safe harbor

Good-faith security research targeting ZestSSH is welcome. We will not pursue legal action against researchers who:

- Report findings promptly and give us a reasonable time to remediate before disclosure
- Avoid accessing, modifying, or destroying data that is not theirs
- Do not attack users, production infrastructure at scale, or adjacent services
- Operate within applicable law

Rewards

ZestSSH is a one-person project and cannot currently offer cash rewards. Validated security findings receive:

- A free ZestSSH Pro license or Strongbox subscription (your choice)
- Public acknowledgment in the release notes (if you want it)
- Direct communication with the developer during remediation

As the project grows, a formal bug bounty program is on the roadmap.

16

Appendix: algorithm references

For reviewers who want to verify the specific standards ZestSSH implements, the following references point to the original specifications and implementation guides.

Algorithm	Reference
`AES-256-GCM`	NIST SP 800-38D (Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode)
`Argon2id`	RFC 9106 — Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications
`HKDF-SHA256`	RFC 5869 — HMAC-based Extract-and-Expand Key Derivation Function
`SHA-256`	FIPS 180-4 — Secure Hash Standard
`ChaCha20-Poly1305`	RFC 8439 — ChaCha20 and Poly1305 for IETF Protocols
`Ed25519`	RFC 8032 — Edwards-Curve Digital Signature Algorithm (EdDSA)
`curve25519`	RFC 7748 — Elliptic Curves for Security
`TLS 1.2`	RFC 5246 — The Transport Layer Security Protocol Version 1.2
`TLS 1.3`	RFC 8446 — The Transport Layer Security Protocol Version 1.3
`SSH-2`	RFC 4251–4254 — Secure Shell Protocol Architecture and Sub-Protocols
`SQLCipher`	Zetetic LLC — SQLCipher Design Specification (sqlcipher.net/design)

Library implementations

ZestSSH relies on the following well-audited library implementations for cryptographic operations:

- **OpenSSL** / **BoringSSL** — AES-GCM, SHA-256, random generation via platform bindings
- **libsodium** / **cryptography package equivalents** — Argon2id, HKDF

- **SQLCipher** — database-level encryption
- **zest_ssh_core** (proprietary fork of dartssh2) — SSH protocol implementation
- **flutter_secure_storage** — platform-native secure storage access (Keychain, Keystore, DPAPI, libsecret)

Document version history

Version	Date	Changes
1.0	April 2026	Initial publication. Covers ZestSSH v1.4.x.

Questions, concerns, or findings?

This whitepaper describes the design. If anything in it is unclear, appears mistaken, or contradicts observed behavior, we want to know. Security is an ongoing conversation — not a fixed claim.

security@zestssh.com · zestssh.com/security

ZestSSH Security Whitepaper v1.0

Published April 2026 by Affluent Labs. This document describes the security architecture of ZestSSH as of version 1.4.x. Future versions may introduce changes; this whitepaper will be updated alongside any architectural modifications.