Necron Vault Manager Security Model

Public security whitepaper
April 2026

At Necron, we build Vault Manager to give users private file storage without requiring them to trust ordinary storage locations with readable data. We make it possible to use familiar folders — local folders, external drives, and cloud-synced folders — while keeping file contents and sensitive metadata encrypted before they ever reach those locations.

The core idea is simple: we encrypt your files on your own computer before writing them to storage, and we keep the secret material needed to unlock them separately on a Necron key drive. The storage locations hold encrypted vault data. The key drive holds cryptographic key material. Neither is useful on its own.

This document explains the Necron Vault Manager security model. It is somewhat technical, but we have written it for a general technical audience. It focuses on what we protect, what we do not protect, how files and metadata are encrypted, and how our key drive and mirror-location model work together.

Overview

Most cloud storage systems are built around convenience. They make it easy to sync files across devices, but they often require users to trust the storage provider, the account provider, or a password-derived key stored near the encrypted data.

We take a different approach. We treat storage as untrusted. A vault can be stored in one or more locations, including ordinary folders that sync through services such as Google Drive, Dropbox, or OneDrive. We do not rely on those providers for confidentiality. We use them only to store and synchronize encrypted data.

Our security model is based on four principles:

1. **Local encryption before storage**
   We encrypt file contents and sensitive metadata locally before writing them to any storage location.
2. **Physical separation of data and key material**
   We store encrypted data in user-selected folders. We store the key material required to decrypt it separately on a key drive.
3. **Authenticated encryption, not just encryption**
   We use authenticated encryption so that unauthorized changes to encrypted files or metadata are detected.
4. **Redundant encrypted mirrors**
   We allow a vault to be mirrored across multiple storage locations. If one location is missing data or contains a damaged copy, we can use another verified copy to restore it.

The result is a vault system where cloud folders provide synchronization and backup convenience while remaining outside the trust boundary for confidentiality.

What Necron is designed to protect

We design Necron Vault Manager to protect sensitive files against attackers who obtain access to encrypted storage locations but do not have the user's key drive.

This includes situations such as:

- a cloud account compromise;
- a stolen laptop or external drive containing encrypted vault data;
- malware or a remote attacker copying encrypted vault files while the key drive is not present;
- accidental corruption or deletion in one mirror location;
- malicious modification of encrypted files in a storage location;
- a storage provider, network observer, or backup service seeing only encrypted vault data.

We protect both file contents and user-facing names. A storage provider does not need to know whether an encrypted object represents a tax return, a photograph, a contract, or a folder of business records.

We also avoid making the account server a decryption authority. Subscription status, licensing, or account management may involve a server, but vault key material is not uploaded to that server as part of normal vault operation.

What Necron is not designed to protect against

No local encryption product protects against every compromise. Our security model depends on the user's computer and the Necron application being trusted while a vault is unlocked.

We do not claim to protect against:

- a compromised operating system that can read files as the user opens them;
- malware running as the same user while the vault is unlocked;
- screen capture, clipboard capture, keylogging, or memory inspection on a compromised endpoint;
- malicious or compromised third-party applications used to open decrypted exports or working copies;
- loss of every copy of the key drive needed to decrypt the vault;
- a stolen key drive combined with a compromised or weak unlock factor;
- social engineering that tricks the user into exporting plaintext or revealing credentials;

- malicious software updates or supply-chain compromise;
- physical coercion or legal compulsion.

When files are opened in external applications, those applications may create their own caches, temporary files, thumbnails, autosave records, or recent-file entries. We control our own working copies, but we do not fully control the behavior of every external editor.

For the strongest protection, users need to treat the key drive like a safe key, keep recovery keys in a secure place, use strong unlock factors, and unlock vaults only on trusted machines.

Data model

A Necron vault is a private file space. Inside the application, the user sees familiar file and folder names. Outside the application, the storage locations contain encrypted objects and sealed metadata.

A vault has four main concepts:

Vault

A vault is the user’s encrypted file collection. It contains a logical folder tree, encrypted file objects, and configuration describing which storage locations belong to the vault.

Node

A node represents a user-visible file or folder in the vault tree. Nodes contain encrypted metadata such as the display name, parent folder relationship, and the pointer to the encrypted file content.

We render the vault tree from node metadata without decrypting file contents. This lets the application show the user’s folder structure without loading every file body.

Encrypted object

An encrypted object stores the protected content of a file. We encrypt file content independently from the metadata that places it in the folder tree.

This separation keeps browsing fast and avoids scanning bulk file data simply to display the vault.

Mirror location

A mirror location is a user-selected storage location that holds an encrypted copy of the vault data. A vault may use one mirror location or several. We treat mirror locations as untrusted storage.

Multiple mirror locations improve availability and integrity. If a valid encrypted object exists in one location but is missing or damaged in another, we can use the valid copy to repair the incomplete location.

Security boundary

We keep the security boundary intentionally narrow.

The trusted side contains:

- the Necron application while it is running on a trusted computer;
- the key drive while it is inserted and unlocked;
- the user’s chosen unlock factors, such as a time-based one-time code where enabled.

The untrusted side contains:

- cloud storage providers;
- local folders holding encrypted vault data;
- network connections used for account or subscription checks;
- backup systems that copy encrypted vault folders;
- attackers who obtain encrypted vault files but not the key drive.

The important design choice is that storage is outside the trust boundary. A storage provider can store, sync, delete, corrupt, duplicate, or withhold encrypted data, but it cannot decrypt valid vault contents without the key drive.

Main encryption model

We use modern symmetric authenticated encryption for vault data. We encrypt file contents using XChaCha20-Poly1305, an authenticated encryption algorithm designed to provide both confidentiality and integrity.

Authenticated encryption means that decryption does not merely produce bytes. It also verifies that the encrypted data has not been modified under the wrong key, with the wrong metadata, or in the wrong context. If verification fails, we reject the data.

For each file, we derive a file-specific encryption key from key material stored on the key drive. We use HKDF-SHA-256, a standard key-derivation function, to separate keys used for different purposes.

The important security properties are:

- each file is encrypted under its own derived key;
- the derived key depends on material that is not stored with the encrypted vault;
- encrypted content is bound to its vault and file identity;
- corrupted or tampered ciphertext is detected before plaintext is accepted;
- encrypted file content can be handled without exposing the user’s logical filenames to storage providers.

The key drive is therefore not just a convenience token. It is part of the cryptographic root of trust for the vault.

Metadata protection

Protecting file contents is not enough. Filenames, folder names, and folder structure can reveal sensitive information even when file bodies are encrypted.

We protect user-visible metadata by sealing it before writing it to storage. We store file and folder names inside encrypted metadata objects rather than as readable filenames in the storage provider's folder.

For deterministic metadata protection, we use AES-128-SIV. This is useful for small structured metadata because it provides authentication and avoids the risks of nonce reuse in situations where the same name or metadata field may need a stable encrypted representation.

The storage provider sees opaque encrypted objects. It does not see the user's real filenames, folder names, or meaningful folder structure.

Some metadata cannot be hidden completely from storage. For example, a storage provider may still observe approximate encrypted object sizes, modification timing, and the fact that data exists. Our goal is to protect content and sensitive user-facing metadata, not to provide a complete traffic-analysis-resistant storage system.

Key drive model

The key drive stores cryptographic key material used to unlock vaults and derive the keys that protect vault data. It does not store the user's documents, photos, videos, or other vault contents.

This separation gives Necron a different security posture from a system where encrypted files and key material are stored together. An attacker who obtains only the encrypted storage folder lacks the key material required to decrypt it. An attacker who obtains only the key drive does not automatically have the encrypted vault contents.

The key drive also supports a practical recovery model. A user may create one or more backup key drives, each a full clone of the primary key. Backup keys carry the same key identity and capabilities as the original, so either key can be used to unlock and manage the vault.

A key drive can be protected by an additional unlock factor such as a time-based one-time code. This turns access into a two-factor model: something the user has, plus something the user knows or can generate.

Users need a deliberate recovery plan. If all compatible key drives are lost, the vault may become permanently unrecoverable.

How file saving works

When a user saves a file into a vault, we perform the security-sensitive work locally.

At a high level:

1. The application reads the plaintext file on the trusted endpoint.
2. We derive the file-specific encryption key from hardware-held key material.
3. We encrypt the file content using authenticated encryption.
4. We seal the metadata needed to display and manage the file inside the vault.
5. We write encrypted objects and sealed metadata to the selected mirror locations.
6. We verify and track the new vault state without storing the plaintext file in the encrypted storage locations.

For updates, we use an object-oriented pattern: new encrypted content is written first, and the file's metadata is then updated to point to the new encrypted content. This reduces the risk that a partial write or interrupted sync leaves the vault in an ambiguous state.

How opening a file works

When a user opens or exports a file from a vault, we reverse the process locally.

At a high level:

1. The user selects a file from the vault view.
2. We read the sealed metadata for that file.
3. The application uses the key drive to derive the key needed for the file's encrypted content.
4. The encrypted object is authenticated and decrypted locally.
5. If authentication succeeds, plaintext is returned to the user or to an application-managed working copy.
6. If authentication fails, the object is rejected and may be repaired from another verified mirror if one is available.

The application does not need to decrypt every file body to display the vault tree. We decrypt file content only when the user requests that content.

Integrity and tamper detection

A vault needs to protect against both disclosure and silent modification. Encryption alone is not sufficient if an attacker can modify ciphertext and cause corrupted or maliciously swapped data to be accepted.

We use authenticated encryption and authenticated metadata so that tampering is detected. Encrypted content is cryptographically bound to its vault context and file identity. Metadata is sealed so that unauthorized changes are rejected.

This protects against attacks such as:

- modifying bytes inside an encrypted file object;
- replacing one encrypted object with another from a different context;
- changing sealed metadata without the key drive;
- corrupting a mirror location and hoping the application accepts the damaged copy;
- attempting to make the vault tree point to the wrong encrypted content.

If verification fails, we treat the copy as untrusted. We do not decrypt and accept unauthenticated plaintext.

Mirror locations and self-healing

A traditional encrypted folder often has a single storage location. That provides confidentiality, but it does not solve availability. If the folder is deleted, corrupted, or partially synced, the user may lose data.

We allow a vault to be stored in multiple mirror locations. Each mirror location contains encrypted vault data. No mirror location needs to be trusted with plaintext.

When we detect that one mirror is missing data or contains a damaged encrypted object, we compare the available copies and use a verified copy to repair the incomplete mirror.

This model protects against:

- accidental deletion in one location;
- cloud sync errors;
- partial uploads or interrupted writes;
- storage corruption;
- tampering that affects one mirror but not all mirrors.

Self-healing is only possible when at least one valid copy remains. If every mirror contains the same missing or damaged object, we cannot reconstruct the lost plaintext from nothing.

Mirror locations are also not a substitute for key backups. A perfect encrypted mirror is still useless if the user loses every compatible key drive.

Account server role

Necron Vault Manager may use an account server for services such as account creation, licensing, subscription status, or product updates. The server is not part of the vault decryption path.

The account server does not receive the key drive's vault-decryption material. It cannot decrypt vault contents, filenames, or sealed vault metadata.

This separation is important. A compromise of an account system does not, by itself, allow an attacker to decrypt the user's vault data. The attacker still needs the relevant key drive material and any required unlock factors.

This does not mean account security is irrelevant. An attacker who controls an account, payment record, or update channel may still cause operational harm. But the confidentiality of vault data depends on local encryption and hardware-held key material, not on trusting the account server with plaintext access.

Software key mode

Our strongest security model uses a physical key drive. We also offer software key mode for trial, demo, or low-friction onboarding use. This mode is useful for evaluation, but it has a different threat profile.

When key material is stored on the same computer as the application, the separation between encrypted data and key material is weaker. Platform protection can make offline copying harder, but software-only protection does not provide the same assurance as a separate physical key.

We treat software key mode as a convenience and evaluation mode, not as the recommended configuration for highly sensitive or long-term secrets.

Quantum-resilience considerations

Necron's vault encryption is based on symmetric cryptography. Symmetric encryption is not affected by the same class of quantum attacks that threaten widely used public-key systems such as RSA and elliptic-curve cryptography.

A sufficiently large quantum computer running Grover's algorithm can reduce the effective security margin of symmetric keys. This is why we use high-strength symmetric primitives and key sizes intended to preserve a substantial security margin even under conservative long-term assumptions.

This does not mean that any system can promise unlimited future security. Cryptographic engineering depends on correct implementation, secure endpoints, careful key handling, and the continued strength of chosen primitives. Our design goal is to avoid reliance on public-key encryption for vault confidentiality and to use well-studied symmetric primitives with strong security margins.

Cryptographic primitives

We use standard, widely studied cryptographic building blocks.

Purpose	Primitive
File content encryption and authentication	XChaCha20-Poly1305
Deterministic sealing of small metadata	AES-128-SIV
Directory integrity and file authentication	HMAC-SHA-256
Key derivation and domain separation	HKDF-SHA-256
Password hardening where applicable	Argon2id
Time-based second factor where enabled	TOTP
Transport security for account communication	TLS

The security of the system depends not only on the primitives, but also on how we compose them, how keys are generated and stored, how authentication failures are handled, and how the application behaves around plaintext.

Standard primitives reduce cryptographic risk. They do not remove the need for implementation review, security testing, and independent audit.

Plaintext handling

We design Necron so that vault storage locations hold encrypted data, not plaintext files. Browsing the vault tree does not require decrypting every file body.

However, plaintext must exist somewhere when the user opens a file. It may exist in application memory, in an export chosen by the user, or in an application-managed working copy if the user opens a file in an external editor.

We minimize plaintext exposure by:

- avoiding persistent plaintext in vault storage locations;
- using controlled working locations for temporary files;
- cleaning up working copies where practical;
- warning users where external applications may create their own caches;
- requiring the key drive for decrypting or modifying protected content.

Secure deletion cannot be guaranteed on all filesystems and storage devices, especially modern SSDs. Users handling very sensitive files need to consider the security of the endpoint and external applications, not only the vault encryption.

Operational recommendations

The security of an encrypted vault depends on both cryptography and user operations. We recommend that Necron users:

- keep at least one recovery key drive in a secure place;
- enable an additional unlock factor for sensitive vaults;
- avoid unlocking vaults on untrusted computers;
- maintain more than one mirror location for important data;
- periodically verify that recovery keys work;
- keep the Necron application up to date from trusted sources;
- treat exported plaintext files as sensitive;
- remember that cloud sync is not the same as a full backup strategy.

Organizations using Necron define access policies, key custody procedures, recovery procedures, and endpoint-security requirements before relying on Necron for regulated or mission-critical data.

Security status and review

We build Necron Vault Manager from standard cryptographic primitives and a conservative local-encryption model. We intend the design to be reviewable and understandable.

At the same time, the complete security of any encryption product depends on implementation quality. Important areas for review include:

- key generation and hardware-key provisioning;
- authenticated encryption and metadata sealing;
- unlock-factor handling;
- plaintext lifecycle management;
- update security;
- mirror repair logic;
- error handling after authentication failure;
- memory handling for sensitive material;
- platform-specific storage and permission behavior.

Use of standard algorithms is not a substitute for an independent security audit. We welcome review from cryptographers, security engineers, and technically sophisticated users.

Conclusion

We build Necron Vault Manager to give users private, hardware-key-protected file storage without requiring them to trust cloud providers with plaintext data.

Our design combines local authenticated encryption, hardware-held key material, encrypted metadata, and multi-location encrypted mirrors. This allows ordinary folders and cloud sync services to act as storage and redundancy layers while remaining outside the confidentiality boundary.

The most important security property is separation: encrypted vault data can live in convenient storage locations, while the key material required to unlock it remains under the user’s physical control.

Necron does not remove the need for trusted endpoints, careful key custody, and good operational security. But for the threat we are designed to address — protecting files stored in untrusted or semi-trusted locations — we provide a clear and practical model: encrypt locally, keep keys separate, verify integrity, and repair from trusted encrypted mirrors when possible.