ExoServe

ExoServe: Architecture and Security

July 2026

Alex Benlolo, Founder

Ciphranova LLC

abenlo1@ciphranova.com

Ciphranova LLC

Table of Contents

1. Executive Summary

ExoServe is a self-hosted, end-to-end encrypted (E2EE) file storage solution designed for deployment on local hardware. The system enforces a strict separation of concerns between the client and the server. All cryptographic operations, including key management, file encryption, and directory topology obfuscation, are performed exclusively on the client side.

The server functions as an opaque blob store. It possesses no knowledge of file contents, file names, or directory structures. This architecture ensures that even in the event of a server compromise or physical seizure of storage media, the data remains unintelligible without client-side master keys.

This document outlines the cryptographic architecture, authentication mechanisms, and threat model of ExoServe. It serves as a technical reference for security engineers and system administrators evaluating the platform for deployment.

2. Introduction

Centralized cloud storage solutions typically require trust in the service provider to maintain data confidentiality. While E2EE implementations exist within the cloud ecosystem, they often rely on provider-side infrastructure, which introduces uncontrollable attack surfaces or metadata leakage.

ExoServe addresses this by shifting the trust boundary entirely to local hardware. By hosting the infrastructure on local hardware, the user retains physical and logical control over the data. The system is designed to eliminate the risk of provider-side inspection, automated scanning, or unauthorized access to plaintext data.

The goal of this document is to define the security provided by ExoServe and explicitly state the limitations inherent to the design.

3. System Architecture Overview

The ExoServe architecture relies on a strict client-server model where the server is cryptographically blind to the data it stores.

3.1. Component Roles

• **Client (Web UI):** Built with vanilla JavaScript. Responsible for all cryptographic operations, key generation, file encryption, and decryption. The client never transmits plaintext file data or secrets to the server.

• **Server (Backend):** A lightweight storage engine built with Python Flask. Responsible for storing encrypted blobs, handling authentication metadata, and managing session tokens. The server does not possess decryption keys.

• **Network:** Communication is secured via TLS 1.3. While this protects data in transit from passive observers, an adversary monitoring live traffic on the server can still ascertain limited metadata.

3.2. Deployment Model

ExoServe is designed to run behind a reverse proxy (e.g., Nginx). This allows for standard TLS termination and security hardening (headers, rate limiting) at the network perimeter, while the application logic remains isolated within the local environment.

4. Cryptographic Design and Data Integrity

ExoServe utilizes standard cryptographic primitives to ensure confidentiality and integrity. The algorithms selected (AES-256, SHA-256, RSA-2048) are industry standards and are compatible with FIPS-compliant operating environments where applicable.

4.1. File Content Encryption

All files are encrypted before leaving the client device.

• **Algorithm:** AES-256-GCM.

• **Key Management:** Each file is encrypted with a unique, cryptographically random key.

• **Integrity:** The GCM mode provides an authentication tag. If an adversary modifies an encrypted blob, the client will detect the tampering upon download as the authentication tag will fail validation.

4.2. Directory Topology Encryption

Standard E2EE systems often encrypt file contents but leave directory structures visible. ExoServe encrypts the directory topology using a Merkle Tree structure.

• **File Nodes:** Contain the encrypted file content. Referenced by the SHA-256 hash of the encrypted content.

• **Folder Nodes:** Contain a dictionary linking child names to their respective hashes and decryption keys. The folder node itself is also encrypted.

• **Root Node:** The master folder containing the top-level directory structure, encrypted with the user's master key.

Implication: An attacker with access to the server cannot reconstruct the folder hierarchy. They cannot determine if a specific blob belongs to a folder named "Secrets" or "Photos." Every modification to the directory structure (add, delete, modify) requires re-hashing the affected nodes up to the root.

4.3. Hash Sharding

To prevent directory depth limits and obscure file relationships, files are not stored in traditional paths. Instead, they are stored using hash sharding.

• **Mechanism:** A blob with hash `a8f5f167…` is stored at `/sandbox/{uuid}/a8/f5/a8f5f167…`

• **Security Benefit:** This prevents an adversary with physical disk access from inferring file relationships based on directory nesting. It also ensures high performance for large datasets by distributing files across the file system.

5. Authentication and Access Control

Authentication is designed to ensure that the server never handles the user's password or plaintext private key.

5.1. Registration and Key Generation

• **Username Mutation:** The username is hashed to derive a UUID, discarding the original username.

• **Key Derivation:** The password and a random salt are used to derive a master key (PBKDF2).

• **Key Pair:** A random RSA-2048 key pair is generated client-side.

• **Storage:** The private key is encrypted with the master key. The UUID, salt, IV, public key, and encrypted private key are stored on the server.

5.2. Challenge-Response Authentication

Login does not involve transmitting a password hash.

• **Challenge:** The server requests a signature using an encrypted private key for a provided nonce.

• **Response:** The client decrypts the private key locally, signs the nonce, and submits the signature.

• **Verification:** The server validates the signature using the stored public key.

Security Benefit: Unlike password hashes, which are primary targets for offline cracking, encrypted private keys require successful password derivation before they can be utilized. Even if the encrypted private key is stolen, it remains useless without the master key.

5.3. User Enumeration Protection

To prevent attackers from guessing valid usernames, the system handles does not reject invalid UUIDs. If a UUID does not exist, the server returns dummy data of the same size and entropy as valid data. Crucially, this dummy data is deterministically derived from the UUID. This ensures that consecutive attempts with the same invalid UUID return identical responses, preventing attackers from comparing responses to distinguish between dummy data and valid user data.

6. Client-Side Operations and Performance

Maintaining security often degrades usability. ExoServe implements specific architectural choices to mitigate this without compromising the threat model.

6.1. Ephemeral Decryption

Previewing media files typically requires downloading and decrypting the entire file to disk. ExoServe utilizes a Service Worker to intercept network requests.

• **Process:** Encrypted chunks are streamed from the server, decrypted in ephemeral memory by the Service Worker, and piped directly to the browser renderer.

• **Benefit:** Decrypted content is never explicitly written to persistent storage by the client. While browser caching behavior depends on reverse proxy configuration, the client architecture does not mandate writing plaintext data to disk

6.2. Search Efficiency

Searching an encrypted Merkle tree is computationally expensive. ExoServe utilizes JavaScript Promises to perform parallel depth-first searches. The number of parallel workers is configurable in the settings, allowing users to balance performance against resource usage. This allows the UI to remain responsive while traversing the encrypted directory structure.

6.3. Concurrency Control

To support multi-device synchronization, the system uses atomic compare-and-swap logic.

• **Uploads:** Staged in parallel.

• **Tree Mutations:** Serialized to ensure integrity.

• **Transient Locks:** Used to prevent race conditions during simultaneous read and write operations.

6.4. Implementation Stack

The system utilizes a lightweight, dependency-minimized stack to reduce attack surfaces.

• **Frontend:** Built with vanilla JavaScript and the native Web Crypto API.

  ◦ **Performance:** Native API calls provide faster cryptographic operations compared to third-party JavaScript libraries.

  ◦ **Security:** Avoids heavy frameworks, reducing supply-chain vulnerabilities.

  ◦ **Compliance:** The Web Crypto API interfaces directly with OS-level cryptographic modules, allowing compatibility with FIPS-compliant browser environments.

• **Backend:** Built with Python Flask and the “cryptography“ library.

  ◦ **Transparency:** The use of standard, audited libraries ensures cryptographic operations rely on industry-standard implementations (OpenSSL).

  ◦ **Compliance:** The backend leverages system-level cryptographic providers, supporting deployment in FIPS-compliant operating environments.

7. Threat Model

Security is a trade-off. ExoServe prioritizes data confidentiality and integrity over obscuring metadata. This section defines the boundaries of the security features.

7.1. Protected Assets

• **File Contents:** Encrypted at rest and in transit.

• **File Names:** Encrypted within the Merkle tree structure.

• **Directory Topology:** Encrypted and obfuscated via hash sharding.

• **Authentication Secrets:** Passwords and private keys never touch the server in plaintext.

7.2. Exposed Metadata

While the content is hidden, limited metadata leakage occurs during network transit and at rest:

• **File Sizes:** The size of the encrypted blob is visible on the wire and at rest on the server.

• **Access Patterns:** The timing of requests reveals when nodes are accessed.

• **Traffic Volume:** The total amount of data transferred can be correlated with user activity.

• **Node Counts:** While specific file counts are not explicitly stored, the number of nodes is visible.

Planned Mitigation Strategy: To obfuscate file sizes and node counts, it is planned to chunk and pad nodes to a constant size while in storage. This would effectively split a large file into many nodes, obfuscating file size and making node count a less insightful piece of metadata.

Deferred Mitigation Strategy: To mitigate information gained from traffic observation, traffic padding and throttling to a constant transmission rate could be used, but would introduce significant latency and bandwidth overhead.

7.3. Endpoint Security

This architecture assumes a trusted client device. If the client device is compromised (e.g., via keylogger, screen capture, or malware), the encryption provides no protection on that client. The security boundary ends at the user's device and the integrity of the client code delivered by the server. However, in a multi-user environment, a compromise of one user's credentials does not expose the private data of other users., as keys are isolated per account.

7.4. Server Compromise

If the server is compromised:

• **Data:** Remains readable only in ciphertext form.

• **Metadata** User UUIDs, node counts, and node sizes are visible.

• **Session Integrity:** A compromise allows modification of the client-side JavaScript delivered to the browser. This could expose credentials or plaintext data in future login sessions. This is a known limitation of web-based E2EE architectures compared to native applications.

• **Integrity:** Tampering with blobs is detectable by the client via authentication tags.

8. Data Sovereignty and Licensing

ExoServe is distributed under a Business Source License (BSL 1.1). This model is chosen to support long-term data sovereignty while ensuring the code can be audited

8.1. Ownership

The user owns the hardware and the software license. There are no subscription dependencies that could result in loss of access due to service discontinuation or pricing changes.

8.2. Portability

Because the server is merely a blob store and the encryption keys are client-side, data is not vendor-locked. Users can migrate their storage backend to new hardware without losing access to their files, provided they retain their master key.

8.3. License Model

The Business Source License allows users to audit the source code freely, ensuring transparency regarding the cryptographic implementation. However, it restricts the use of the code for competitive commercial services, protecting the development ecosystem while allowing full technical inspection.

8.4. Key Management

A forgotten password results in total, irrecoverable data loss for the affected user.

• The server does not store password hints or recovery keys.

• The server cannot reset the password without invalidating the existing encryption.

This limitation is a feature of the sovereignty model. It ensures that there are no backdoors and no administrative override of user privacy. Even a rogue system administrator with full access to the server infrastructure cannot decrypt user files without the specific user's master key.

9. Conclusion

ExoServe provides a verifiable architecture for private file storage. By enforcing client-side encryption and opaque server storage, it eliminates the risk of server-side data inspection. While the system does not hide all metadata, it provides strong guarantees regarding content confidentiality and integrity.

The design prioritizes a transparent, auditable codebase and a deployment model that ensures user ownership of data. For organizations and individuals requiring strict control over their storage infrastructure, ExoServe offers a technically sound alternative to centralized cloud solutions.