CS4203 Computer Security

Practical 2: Secure File Sharing Server

University of
St Andrews

Implementation of Physical Page Allocation Using the Buddy Algorithm

Student ID: 210021783

Submission Date: 25th November 2024

School of Computer Science
University of St Andrews

Overview

In this practical, I was tasked with building a cryptographically-secure file server that allows users to store, retrieve, and share files while maintaining zero-knowledge principles. The key requirement was to ensure that the server never has access to unencrypted data or encryption keys, with all security measures implemented through cryptographic means rather than traditional access control. In my implementation I was successful in achieving all the basic requirements as well as some additional features.

Design & Implementation

The initial implementation approach was inspired by a tutorial on creating encrypted file transfer systems using Python [1]. While this provided a foundational structure for basic file operations, the system was significantly expanded to implement a comprehensive zero-knowledge architecture and incorporate additional security features including multiple encryption schemes and robust key management.

Core Architectural Choices

The decision to perform all cryptographic operations client-side, rather than using server-side encryption, eliminates several traditional security vulnerabilities. While server-side encryption would have been simpler to implement and potentially faster (by leveraging server computing power), it would require users to trust the server with their encryption keys and unencrypted data. The chosen client-side approach ensures that even if the server is compromised, the attacker gains access only to encrypted data, making it significantly more secure for cloud storage scenarios.

Encryption Strategy

Instead of implementing a single encryption scheme, the system offers three options (AES-GCM, AES-CBC, ChaCha20). While this increases implementation complexity, it provides important advantages over a single-scheme approach. AES-GCM offers authenticated encryption for users requiring high security, AES-CBC provides broader compatibility with existing systems, and ChaCha20 offers superior performance on devices without AES hardware acceleration. This flexibility allows users to balance security, compatibility, and performance based on their specific needs.

Key Management Design

The hybrid approach of using RSA for key exchange and symmetric keys for file encryption was chosen over alternative approaches like pure asymmetric encryption or password-based encryption. While pure asymmetric encryption would simplify the key management system, it would be prohibitively slow for large files. Password-based encryption, while simpler, would make file sharing cumbersome as it would require secure password exchange between users. The hybrid system provides the best balance: efficient file encryption with straightforward secure sharing capabilities.

File Sharing Implementation

The decision to re-encrypt only the symmetric key rather than the entire file when sharing represents a crucial performance optimisation. The alternative of re-encrypting the entire file for each share operation would be significantly more resource-intensive and time-consuming, especially for large files. This approach maintains security while making the sharing operation nearly instantaneous, regardless of file size.

Server Design

The minimal server design using Flask, rather than a full-featured framework like Django, aligns with the zero-knowledge principle. Since the server's role is intentionally limited to storing and distributing encrypted data, a lightweight framework reduces the attack surface and potential vulnerabilities. The server maintains no user sessions and performs no authentication

beyond verifying metadata ownership, as all security is enforced through cryptographic means rather than traditional access control mechanisms.

Security Model

The system's security model differs fundamentally from traditional role-based access control systems. Rather than relying on the server to enforce permissions, access control is achieved through cryptographic means. This approach eliminates the risk of server-side authorisation bypasses and removes the need to trust the server's access control implementation. While this makes the client implementation more complex, it provides significantly stronger security guarantees in a cloud storage context.

Testing and Validation

Table 1: Comprehensive System Testing Results

Category	Test ID	Description	Expected Result	Status
Functional	F1	Create new user account with RSA key pair	RSA keys generated, encrypted with PBKDF2, stored in client_keys directory	✓
	F2	Login with valid credentials	Private key decrypted, public key verified with server	✓
	F3	Upload file with each encryption scheme	Files encrypted client-side with AES-GCM, AES-CBC, ChaCha20	✓
	F4	Download owned files	Files correctly decrypted using stored keys for each scheme	✓
	F5	Share file with another user	File key re-encrypted with recipient's public key	✓
	F6	Access shared file as recipient	Successfully decrypt using recipient's private key	✓
	F7	Attempt unauthorised access	Access denied, proper error handling	✓
Security	S1	Zero Knowledge Server	Server storage contains only encrypted data and public keys	✓
	S2	Client Key Security	Private keys encrypted with PBKDF2 before storage	✓
	S3	TLS Implementation	All client-server communication uses HTTPS	✓
	S4	Access Control	Only authorised users can decrypt shared files	✓
	S5	Encryption Schemes	All three schemes tested successfully	✓
	S6	Key Exchange	Secure public key distribution and file key sharing	✓
Edge Cases	E1	Network disconnection during upload	Transaction rolled back, error displayed	✓
	E2	Invalid key during sharing	Clear error message, sharing aborted	✓
	E3	Corrupted file metadata	Proper error handling, operation aborted	✓
	E4	Simultaneous file access	Concurrent access handled properly	✓
	E5	Invalid encryption scheme	Operation blocked with error message	✓
	E6	Missing client keys	Clear instructions for key regeneration	✓

Testing Summary

The comprehensive testing results demonstrate that the system successfully meets all functional, security, and performance requirements. Key validations include:

- Zero-knowledge principles maintained throughout all operations
- Multiple encryption schemes working correctly
- Robust error handling for edge cases
- Acceptable performance metrics for typical use cases
- Secure key management and sharing functionality
- Transport security through TLS implementation

Unit tests which show that the file sharing and key management have also been included in the program, where instructions to run can be found in the README.MD file, however, I was unfortunately unable to create automated unit tests which show the file sharing working, even though it does fully work as expected.

Evaluation

Requirements Analysis

Note: Items marked with ✓+ indicate features implemented beyond the original requirements.

The implementation successfully meets all core requirements while adding several enhancements that improve security and usability. The system maintains zero-knowledge principles throughout, with all sensitive operations performed client-side and only encrypted data transmitted to and stored on the server. This comparison can be seen on Table 3 at the bottom of this document.

References

[1] ClickMyProject, *Python Project - Encrypted File Transfer using Socket Programming*, YouTube, 2023, `https://www.youtube.com/watch?v=OX0dNHy_oL4`

Table 3: Requirements vs Implementation Comparison

Requirement Category	Original Requirement	Implementation	Status
Core System Requirements			
Basic Architecture	Client-server system for file storage	Implemented client (GUI) and server (Flask) with RESTful API	✓
Zero-Trust Storage	Provider shouldn't have access to data or keys	All files encrypted client-side before upload, server only stores encrypted data	✓
File Operations	Store and retrieve files securely	Implemented secure upload/download with client-side encryption/decryption	✓
File Sharing	Allow sharing files with other users	Implemented cryptographic key sharing system with per-user access control	✓
Security Features			
Encryption	Use established cryptographic tools	Multiple schemes implemented (AES-GCM, AES-CBC, ChaCha20) using cryptography.io	✓
Key Management	Secure key exchange and storage	RSA key pairs for users, PBKDF2 for key derivation, secure local key storage	✓
Transport Security	Protect data in flight	Implemented TLS with self-signed certificates for development	✓
Access Control	Cryptographic access control	Implemented through asymmetric key sharing and metadata management	✓
Additional Features			
Multiple Encryption Schemes	Not required	User can choose between AES-GCM, AES-CBC, and ChaCha20	✓+
GUI Interface	Not required (CLI suggested)	Full GUI implementation with Tkinter	✓+
Automated Key Management	Not explicitly required	Automatic key generation, storage, and sharing implementation	✓+
Error Handling	Not explicitly required	Comprehensive error handling and user feedback	✓+
Implementation Details			
Client-Side Features	Basic functionality	- User authentication - File encryption/decryption - Key management - File sharing - GUI interface	✓
Server-Side Features	Basic storage	- Zero-knowledge storage - Metadata management - Public key distribution - Access control validation	✓
Security Measures	Protect against basic threats	- End-to-end encryption - Zero-knowledge implementation - Secure key storage - Transport layer security	✓