Secure File Transfer System

RSA + AES Hybrid Encryption Protocol

Systems and Network Security Project

University Project

October 22, 2025

Outline

1. Problem Description
2. Encryption Scheme
3. Communication Protocol
4. Features
5. Security Features
6. Demo
7. Conclusion

The Challenge: Secure File Storage

Problem Statement

How do we encrypt files when saving them on a remote server?

- Files need to be stored securely on the server
- The encryption key is **not known in advance**
- Each client session should have a unique encryption key
- Must prevent unauthorized access to stored files

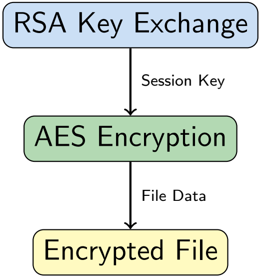
Security Requirements

- **Confidentiality**: Only authorized parties can read files
- **Integrity**: Detect any tampering with encrypted data
- **Key Exchange**: Securely share symmetric keys without prior setup

Solution: Hybrid Encryption (RSA + AES)

Why Hybrid?

- 🔑 RSA (Asymmetric)
  - Secure key exchange
  - No pre-shared secrets
  - 2048-bit keys
- 🔒 AES (Symmetric)
  - Fast data encryption
  - 256-bit keys
  - GCM mode (authenticated)



RSA Key Exchange

Session Key

AES Encryption

File Data

Encrypted File

Encryption Details

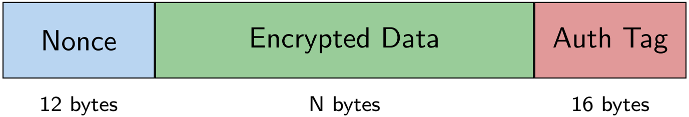
RSA-OAEP (Key Exchange Only)

- **Algorithm**: RSA with OAEP padding
- **Key Size**: 2048 bits
- **Hash Function**: SHA-512
- **Purpose**: Encrypt and transmit AES session key

AES-256-GCM (Data Encryption)

- **Algorithm**: AES in Galois/Counter Mode
- **Key Size**: 256 bits (32 bytes)
- **Nonce Size**: 12 bytes (unique per encryption)
- **Authentication Tag**: 128 bits (16 bytes)
- **Benefits**: Provides both confidentiality and integrity

AES-GCM Message Format



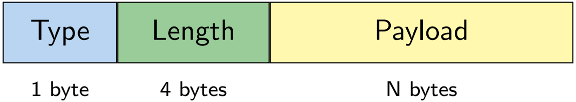
Nonce	Encrypted Data	Auth Tag
12 bytes	N bytes	16 bytes

- **Nonce**: Random value prepended to ciphertext
- **Encrypted Data**: AES-encrypted file content
- **Authentication Tag**: Verifies data integrity

Protocol Overview

Binary Protocol over TCP

- Custom binary format for efficiency
- TCP ensures reliable, ordered delivery
- Message-based communication

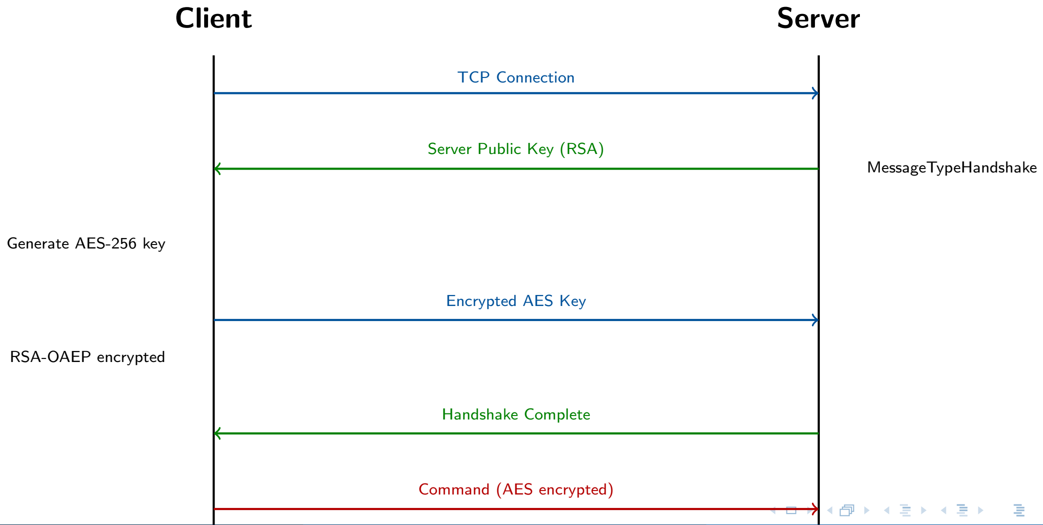


Type	Length	Payload
1 byte	4 bytes	N bytes

Message Types:

- 0x01: Handshake (RSA key exchange)
- 0x02: Command (file operations)
- 0x03: Data (chunked file transfer)
- 0x04: Response (server replies)

Connection Flow



Client

Server

TCP Connection

Server Public Key (RSA)

MessageTypeHandshake

Generate AES-256 key

Encrypted AES Key

RSA-OAEP encrypted

Handshake Complete

Command (AES encrypted)

Handshake Protocol

Step 1: Server → Client

```
Message Type: 0x01 (Handshake)
Payload: RSA Public Key (PEM format, 2048-bit)
```

Step 2: Client → Server

```
Message Type: 0x01 (Handshake)
Payload: AES-256 Key (encrypted with server's public key)
Encryption: RSA-OAEP with SHA-512
```

Result

Both client and server now share a unique **AES-256 session key** that will be used to encrypt all subsequent communication.

Available Commands

File Operations

- Upload (0x01) Upload a file to the server (encrypted with AES-256-GCM)
- Download (0x02) Download a file from the server (chunked transfer)
- List (0x03) List all files available in client's directory
- Delete (0x04) Delete a file from the server

All Data is Encrypted

Every file operation encrypts data using the session-specific AES-256-GCM key established during handshake.

Chunked File Transfer

For efficient large file transfers:

- Files are split into chunks (64 KB - 256 KB)
- Each chunk includes progress information:
  - Chunk index (current chunk number)
  - Total chunks (how many total)
  - Chunk size (bytes in this chunk)
  - Total file size
- Adaptive chunk sizing based on file size:
  - Small files ($<$ 256 KB): 64 KB chunks
  - Medium files ($<$ 5 MB): 128 KB chunks
  - Large files ($\geq$ 5 MB): 256 KB chunks

Benefits

✓ Memory efficient ✓ Progress tracking ✓ Network optimization

Command Message Format

Upload Command Example:

Command	Name Len	Filename	Encrypted Data
1 byte	2 bytes	N bytes	M bytes

Response Message Format:

Success	Msg Len	Message	Data
1 byte	2 bytes	N bytes	M bytes

Path Traversal Protection

Vulnerability: Path Traversal Attack

Malicious clients might try to access files outside their directory:

- `../../../etc/passwd`
- `/etc/shadow`
- `../../config/private.pem`

Our Protection Mechanism

1. **Reject absolute paths**: Only relative paths allowed
2. **Clean and resolve paths**: Use `filepath.Clean()` and `filepath.Abs()`
3. **Verify containment**: Ensure resolved path starts with root directory
4. **Reject empty filenames**: Prevent directory access

`validatePath()` function ensures all file operations stay within bounds!

Path Validation Code

```go
func (h *CommandHandler) validatePath(filename string) (string, error) {
    // Reject empty filenames
    if filename == "" {
        return "", fmt.Errorf("filename cannot be empty")
    }

    // Reject absolute paths
    if filepath.IsAbs(filename) {
        return "", fmt.Errorf("absolute paths not allowed")
    }

    // Get client’s root directory
    rootDir, _ := h.getClientDir()
    absRoot, _ := filepath.Abs(rootDir)

    // Build and clean full path
    fullPath := filepath.Join(absRoot, filename)
    cleanPath := filepath.Clean(fullPath)
    absPath, _ := filepath.Abs(cleanPath)

    // Ensure path is within root directory
    if !strings.HasPrefix(absPath, absRoot+string(filepath.Separator)) {
        return "", fmt.Errorf("path traversal detected")
    }

    return absPath, nil
}
```

Per-Client Session Storage

Isolation Strategy

Each client gets a unique directory based on their session key:

1. Compute SHA-256 hash of the AES session key
2. Use first 16 hex characters as directory name
3. Create directory: `data/<client_hash>/`
4. All file operations restricted to this directory

Benefits:

- ✓ Client isolation
- ✓ No shared storage
- ✓ Session-based access
- ✓ Automatic organization

Example

```
data/
  1e8130cada7a548b/
  a30155fdb2c96dab/
  c45ff82a901b43ef/
```

Additional Security Features

1. **Authenticated Encryption**
   - AES-GCM provides both encryption and authentication
   - Any tampering with ciphertext is detected
   - 128-bit authentication tag prevents forgery
2. **Unique Nonces**
   - Each encryption operation uses a fresh 12-byte nonce
   - Prevents replay attacks
   - Cryptographically secure random generation
3. **Session-Based Keys**
   - New AES key for each connection
   - Keys never stored on disk
   - Limits impact of key compromise

Security Analysis

Implemented Protections

- ✓ RSA-2048 for secure key exchange
- ✓ AES-256-GCM for authenticated encryption
- ✓ Path traversal prevention
- ✓ Per-client storage isolation
- ✓ Unique nonces for each encryption
- ✓ Session-based encryption keys

Potential Improvements

- ✗ No mutual authentication (vulnerable to MITM)
- ✗ No perfect forward secrecy (same RSA key reused)
- ✗ No replay protection (no sequence numbers)
- ✗ No user authentication system

Demo: Starting the Server

Build and run the server:

```
# Build the server
make server
# OR
go build -o bin/server cmd/server/main.go

# Run server with default settings (localhost:8080)
./bin/server

# Run with custom port
./bin/server -port 9000

# Run with custom host and data directory
./bin/server -host 0.0.0.0 -port 8080 -root-dir data
```

Server automatically:

- Generates RSA keys (if not exist)
- Creates data directories

Demo: Client Usage

Build and connect to server:

```
# Build the client
make client
# OR
go build -o bin/client cmd/client/main.go

# Connect to server
./bin/client -host localhost -port 8080
```

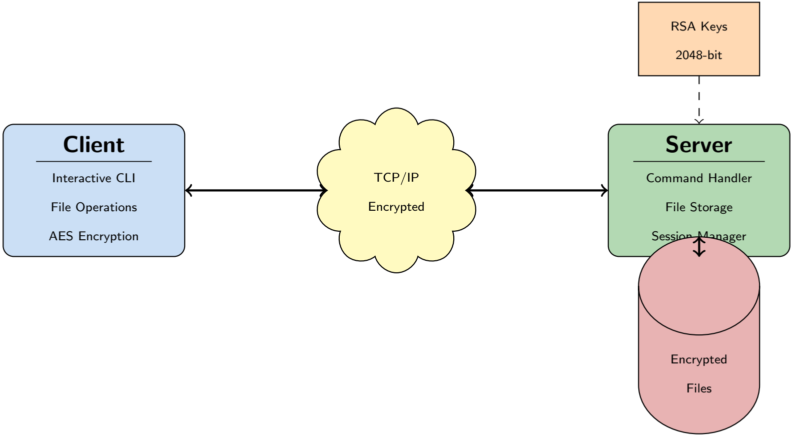
Available commands:

```
> upload myfile.txt        # Upload a file
> download myfile.txt      # Download a file
> list                     # List all files
> delete myfile.txt        # Delete a file
> help                     # Show help
> exit                     # Disconnect
```

Demo: Interactive Session

```
$ ./bin/client -host localhost -port 8080

Connected to server successfully!
Handshake completed. Session secured with AES-256-GCM.

Available commands: upload, download, list, delete, help, exit

> upload test.txt
File ’test.txt’ uploaded successfully

> list
Files on server:
test.txt

> download test.txt output.txt
Downloading file in chunks...
[====================================] 100%
File downloaded to ’output.txt’ (1024 bytes)

> delete test.txt
Are you sure you want to delete ’test.txt’? (y/n): y
File ’test.txt’ deleted successfully

> exit
Goodbye!
```

System Architecture Diagram



Summary

Project Achievements

- **Secure file transfer system** with hybrid encryption
- **RSA-2048 + AES-256-GCM** for key exchange and data encryption
- **Custom binary protocol** over TCP for efficiency
- **Complete command set**: upload, download, list, delete
- **Security features**: path traversal protection, per-client isolation
- **Optimized transfers**: chunked download with progress tracking

Key Takeaways

- Hybrid encryption combines best of symmetric and asymmetric crypto
- Proper input validation is critical for server security
- Session-based isolation prevents unauthorized access
- Authenticated encryption (GCM) provides confidentiality and integrity

Future Enhancements

1. **Mutual Authentication**
   - Certificate-based authentication
   - Prevent man-in-the-middle attacks
2. **Perfect Forward Secrecy**
   - Implement ephemeral Diffie-Hellman key exchange
   - Protect past sessions if keys compromised
3. **User Management**
   - Multi-user support with authentication
   - Role-based access control
4. **Performance Optimizations**
   - Parallel chunk transfers
   - Compression before encryption
   - Resume interrupted transfers

Technologies Used

Programming Language:

- Go (Golang) 1.21+

Cryptography:

- `crypto/rsa`: RSA key generation
- `crypto/aes`: AES encryption
- `crypto/cipher`: GCM mode
- `crypto/sha256`: Hashing
- `crypto/sha512`: RSA-OAEP

Networking:

- `net`: TCP socket programming
- `bufio`: Buffered I/O

Utilities:

- `go.uber.org/zap`: Structured logging
- `encoding/binary`: Binary protocol
- `filepath`: Path manipulation

Repository Structure: Well-organized Go project with `cmd/`, `pkg/`, and clear separation of concerns

Thank You!

Questions?

Secure File Transfer System
RSA + AES Hybrid Encryption

github.com/lcensies/ssnproj