SecTrans project report

Documenting the development and security assessment of the SecTrans client-server application within the SSDLC framework.

Akhyate Brahim

Benziane Swann

Summary

A - Specification phase

1. System architecture

System description

The SecTrans application's architecture is designed to facilitate secure file operations between clients and a central server. At its core, the application leverages two primary libraries provided by Macrohard: libclient.so for the client side and libserver.so for the server side.

File Upload Process:

When an employee initiates a file upload with the command

sectrans -up <file>

the client application first establishes a connection to the server using the server's designated port by sending the command's name. The file is then retrieved in the client then divided into 1024-byte segments. Starting with the first segment, the sndmsg function is called to send the data to the server. As each segment is transmitted, the server uses getmsg to receive the data and write it to a temporary file, ensuring the segments are appended in the correct order. Once the last segment is received and the file is fully transferred, the server moves the temporary file to a permanent storage location.

File Listing Process:

To list files, an employee issues the *sectrans -list* command, which triggers the client to send a predefined list request command to the server using sndmsg. Upon receiving the request, the server reads the file index and compiles a list of files that the employee is authorized to access. This list is sent back to the client in 1024-byte segments, if necessary, using the sndmsg function. The client receives these segments and reconstructs the complete list for the user to view.

File Download Process:

For downloading a file, the command *sectrans -down <file>* is used. The client sends a request specifying the file name to the server using sndmsg. The server locates the requested file in the permanent storage, reads it into memory, and segments it into 1024-byte chunks. These chunks are sent back to the client via sndmsg, with the client using getmsg to receive each one. The client application reconstructs the file from the received segments, ensuring data integrity by verifying segment order and completeness before writing the file to the client's local storage.

2. Security architecture

a. Threat model

Given the basic functionality of the SecTrans application and the lack of inherent security in the Macrohard libraries, the threat model will focus on the following areas:

Data Interception and Modification: One of the primary concerns for the SecTrans application is the risk of unauthorized interception and potential modification of files during their upload or download processes. This threat encompasses the possibility that an attacker could intercept sensitive data in transit, either modifying it to compromise data integrity or stealing it for malicious purposes. The risk is particularly acute considering the utilization of the Macrohard libraries, which may lack robust encryption mechanisms to secure data during transmission.

Unauthorized Access : Another critical threat involves unauthorized access to the server. Attackers gaining unauthorized access can upload, download, or list files without proper authorization, leading to data breaches or the dissemination of harmful files. This scenario could arise from various weaknesses, such as inadequate authentication mechanisms, poor session management, or exploitation of software vulnerabilities.

Data Integrity : The integrity of file data during transfer is also at risk. Files might get corrupted due to network issues or deliberate tampering by attackers. Such corruption can lead to the loss of crucial data or the injection of malicious content into files, thereby compromising the security of the users who access these files.

Man-in-the-Middle (MitM) Attacks Given that the SecTrans application involves data transmission, it is vulnerable to MitM attacks. In such attacks, an adversary could position themselves between the client and the server to intercept, modify, or reroute communications. Without strong encryption and proper certificate validation, the application could be susceptible to such eavesdropping and data manipulation.

Insider Threats Insider threats must also be considered. These can come from employees or associates who have legitimate access to the system but might misuse their access rights for unauthorized activities. Such threats are often hard to detect and can lead to substantial data leaks or system compromises.

Software Vulnerabilities The application could be vulnerable to exploits targeting underlying software dependencies, including the operating system, third-party libraries, and the Macrohard libraries. Regular patching and vigilant monitoring for new vulnerabilities are essential to mitigate this risk.

Physical Security Breaches Lastly, physical security breaches at data centers or locations where servers and backup systems are hosted can lead to unauthorized access to sensitive hardware and data. This aspect is often overlooked but can be just as damaging as cyber threats.

b. Security solutions

Given the threats identified in the threat model, the following solutions are proposed to enhance the security of the SecTrans application. These solutions are tailored to address specific vulnerabilities and risks.

1. Robust Authentication Mechanism

- Implementing an authentication mechanism is vital to prevent unauthorized access. User credentials will be stored in a securely encrypted database. The encryption of the database adds a layer of protection against data breaches, ensuring that even if unauthorized access is gained, the data remains unreadable without the encryption key.
- Storing the encryption key securely is crucial. Best practices include storing the key separately from the data it encrypts, using dedicated hardware security modules (HSMs), or relying on secure key management systems.

2. Hybrid Encryption for Data Communication

- A combination of asymmetric and symmetric encryption will be used for secure communication between the client and the server. Asymmetric encryption, due to its computational intensity, will be used initially to exchange a symmetric encryption key securely.
- The subsequent communication will utilize the symmetric key, which offers better performance for continuous data exchange. This approach balances security with efficiency, ensuring that data in transit is protected against interception and modification.

3. Encryption of Stored Data

- Files uploaded to the server will be encrypted, adding an extra layer of security. This ensures that even if the server is compromised, the files remain protected.
- The decryption of files for authorized users ensures that data integrity and confidentiality are maintained throughout the data lifecycle. The

encryption keys used for this purpose will be securely managed and stored, similar to the database encryption keys.

4. Input Validation and Sanitization

- Rigorous input validation mechanisms will be implemented to prevent malicious or corrupted command-line arguments. This includes whitelisting acceptable input and using prepared statements in database operations to prevent SQL injection attacks.
- Path traversal attacks will be mitigated by sanitizing user inputs, ensuring that file operations are restricted to intended directories and scopes.

5. Integrity Assurance of Data

- To guarantee the integrity of transmitted data, the AES Galois counter mode of operation will be used, which provides both encryption and integrity checks. This ensures that any tampering with the data during transmission or storage can be detected and mitigated.

B - Development phase

1. Encryption and cryptographic measures

In this project, we have employed a hybrid encryption model, utilizing both asymmetric and symmetric encryption techniques to **achieve a balance between security and performance**. This dual approach is particularly crucial given the project's primary functionality involves transmitting files, some of which may be large in size.

Asymmetric Encryption for Key Exchange

Initially, the prospect of using asymmetric encryption, specifically RSA, for the entire file transmission process was considered. However, RSA encryption typically imposes significant performance limitations when dealing with large files, primarily due to its computational intensity and the inherent size constraints of RSA-encrypted data.

To overcome these limitations, RSA is strategically used for a different purpose: the secure transmission of the symmetric encryption key. This process begins as soon as an authenticated employee initiates a session. The employee's client application automatically generates an AES-128 key, leveraging OpenSSL's secure random generator. This key is then encrypted using the server's public RSA key and transmitted securely to the server.

Symmetric Encryption for Data Exchange

Once the AES key is securely transmitted and decrypted by the server using its private RSA key, it becomes the cornerstone for encrypting all subsequent exchanges between the client and the server. This approach leverages the speed and efficiency of AES for bulk data encryption while maintaining robust security.

Each session between a server and a client is safeguarded with a unique AES key. The uniqueness of each session's key is pivotal in fortifying the security, ensuring that the compromise of one session key does not impact the security of others. Moreover, the Initialization Vector (IV) for AES encryption is also generated randomly for each session and is included in the ciphertext, enhancing the encryption's resilience against certain types of cryptographic attacks.

File Storage Security on Server

Regarding the files stored on the server, AES encryption is again utilized to secure them. This encryption not only protects the files during transmission but also ensures their confidentiality while at rest on the server.

It's important to note that in the current implementation, the AES key for file encryption is stored within the source code directory. We acknowledge that this practice, while convenient for development and demonstration purposes, is not recommended for production environments due to potential security risks. In a production setting, it's imperative that encryption keys are managed and stored using more secure methods. This can include **key obfuscation, key splitting, or utilizing dedicated key management services or hardware security modules (HSMs).** These practices help mitigate the risk of key exposure and ensure the overall integrity and confidentiality of the encrypted data.

AES in GCM Mode ensuring data integrity

In this project, AES encryption is implemented in GCM (Galois/Counter Mode). AES-GCM offers two key advantages:

1. **Efficient Encryption**: Utilizing counter mode, AES-GCM encrypts data in a high-speed, parallelizable manner, making it suitable for large files.
2. **Integrated Authentication**: The Galois mode provides message authentication via an authentication tag. This tag ensures data integrity and authenticity, verifying that the data has not been tampered with and confirming its source.

The implementation of AES-GCM is critical, especially in ensuring the uniqueness of the nonce for each encryption operation. Our approach ensures robust encryption and authentication while maintaining high performance.

2. Application and data security

Sanitization of User Input

Only pre-approved characters and patterns are accepted in user inputs, effectively blocking harmful data that could lead to SQL injections or path traversal attacks. Prepared statements are used for all database interactions, they keep user input strictly as data, not executable code, thereby preventing SQL injection.

Encryption of Stored Data

File and Employee Data Encryption: All files and sensitive employee data stored on the server are encrypted, safeguarding them even in case of unauthorized access.

Employee Data is stored on a SQLite database that is encrypted using the SQLCipher extension.

3. User authentication

User authentication is managed through a simple and secure name-password mechanism. Credentials are transmitted securely and verified against an encrypted database, with stringent constraints on usernames and passwords to enhance safety. Usernames are limited to safe characters, while passwords must meet complexity standards. Crucially, passwords are stored as hashes, not plaintext, bolstering security against potential database breaches and ensuring user data protection.

C – Security analysis phase

1. Code review

Positive Aspects:

- **Handling of Various Data Types**: The project is designed to handle any type of data due to its method of transmitting binary data that is Base64 encoded, allowing for a broad range of data compatibility.
- **Secure and Efficient Communication**: The use of AES-128 encryption in the project ensures the security of data during transmission while maintaining efficient data flow.

- **Encrypted Data Storage**: Data encryption prior to storage is a key feature of the project, enhancing the security of stored data against unauthorized access.
- **User Authentication**: The inclusion of user authentication in the project adds an important security layer, which is effective when users select strong passwords.

Areas for Improvement:

- **File Storage Directory**: Storing files in the same directory as the source code, though functional for execution on any machine, could lead to security vulnerabilities. A more secure approach would be to store files in a separate directory with appropriate access controls, especially for final deployment.
- **Key Storage Method**: The current approach to storing encryption keys could be improved. Using hardware security modules (HSM) or key management services would offer enhanced security for cryptographic keys.
- **Adaptability to Different Networks**:
    - The project is currently limited to single-machine operation, a constraint mainly due to its reliance on Macrohard libraries. To extend its network functionality, one approach is to implement network relay services for inter-machine communication.
    - A superior long-term strategy is to redevelop the middleware from the ground up, utilizing direct socket management. This allows for effective handling of IP addresses and network connections. Adopting TLS, supported by OpenSSL for both socket management and security, would significantly enhance the project's network capabilities and security across different environments.

2. Fuzzing SecTrans

We recently tested the SecTrans application for how well it handles unexpected file names and contents. This process was done using the tool **zzuf** seen in the fuzzing lab.

To do this, we wrote a simple script, **fuzz_sectrans.sh**. This script used **zzuf** in two ways. First, it changed the contents of a file we wanted to upload. We then

uploaded this changed file to see how SecTrans would react. Second, we used **zzuf** to change the names of files we wanted to download. By doing this, we checked if SecTrans could handle strange or incorrect file names.

From this testing, we learned some important things. SecTrans did not always work well when dealing with empty files or certain unusual file names. For example, if we tried to download a file that didn't exist, it could cause problems, like making the server crash. This showed us that we need to make SecTrans better at handling these kinds of situations. It's important for the app to deal with unusual or wrong inputs without crashing.