Secure Software Development Project: SecTrans Detailed Report

1. Executive Summary

This report presents the implementation and security analysis of the SecTrans application, a secure file transfer system operating in a client-server model. The application fulfills the functionality requirements specified, including uploading, listing, and downloading files. However, certain constraints and limitations in the provided libraries, as well as the architectural decisions, resulted in compromises in the implementation of security measures, particularly in encryption and authentication.

---

2. System Architecture

The SecTrans application was implemented using the Macrohard-provided libraries (`libclient.so` and `libserver.so`). Due to the limitations of these libraries, the architecture incorporates a workaround where a server is started on the client side to handle responses from the main server. This non-standard design introduces potential security concerns and deviates from typical secure communication protocols. It also makes it impossible to use in the real world as the clients would have to forward a port to receive responses from the server.

Functional Design

The application supports the following operations:

- **Upload a file**: Transfer a file from the client to the server.
- **List files**: Query the server for files stored by the client.
- **Download a file**: Retrieve a file from the server to the client.

The communications were implemented using the following protocol: the library imposes a max packet size of 1024 bytes. The first 7 bytes are used as a header where the first byte is a code to give information about the request or the response type. The different request and response codes are defined in the header file *codes.h*. Then the 6 other bytes are used to write the length of the body sent in hexadecimal characters. The length is in hex and not directly using the bytes values so that the value 0 is not interpreted as '\0' by the functions from the standard library used in the getmsg function provided. The rest of the 1017 bytes are then used to hold the body of the message.

A directory named *dist* should be created next to the server's executable, this is where all the files are going to be uploaded.

Security Design

The following security measures were intended but not fully realized due to constraints:

- **Encryption**: The communication between the client and server lacks robust encryption. Implementing TLS was infeasible with the provided libraries, and hardcoding encryption keys was deemed insecure.
- **Authentication**: No mechanism exists to authenticate clients or servers, making the system vulnerable to impersonation and unauthorized access.

But buffer overflows were a major concern so we paid special attention to this during the development of the application, and they should not occur.

---

3. Threat Model

The threat model identifies key vulnerabilities arising from the design and implementation constraints:

Identified Threats

- **Eavesdropping**: Without encryption, data transmitted between the client and server is susceptible to interception.
- **Impersonation**: The lack of authentication allows adversaries to impersonate clients or the server.
- **Man-in-the-Middle Attacks**: An attacker could intercept and manipulate communication.

Risk Assessment

The absence of encryption and authentication mechanisms significantly elevates the risk of data breaches and unauthorized access.

---

4. Security Analysis

The implemented application was analyzed for vulnerabilities and shortcomings:

Fuzzing

Fuzz testing using Google’s afl-fuzz revealed potential weaknesses in input handling but did not expose critical vulnerabilities. On the server side, which is more vulnerable as it the one hosting the files, the input from the clients are sanitized so that users cannot access files outside of the dedicated directory.

Architectural Concerns

The workaround requiring a server on the client side introduces a non-standard architecture that complicates securing the communication channel.

---

5. Recommendations

Based on the analysis, the following recommendations are proposed:

Encryption

Implement a secure communication protocol, such as TLS, to protect data in transit. While this may require integrating additional libraries, it is critical for ensuring data confidentiality and integrity.

Authentication

Introduce mutual authentication between the client and server to prevent impersonation. Using public/private key pairs or an external authentication mechanism could be considered.

Language Considerations

Given the challenges faced with C and the provided libraries, parts of the application, particularly the communication module, could benefit from being rewritten in Rust. Rust's strong memory safety guarantees and libraries for secure communication make it a suitable choice. For example, we could use Rust’s safer Vec<u8> instead of the current C buffers to make it even less prone to buffer overflows.

---

6. Conclusion

The SecTrans application successfully implements the required functionality but falls short of achieving robust security due to constraints imposed by the provided libraries. The lack of encryption and authentication exposes the system to significant risks. Addressing these issues through architectural redesign ,adopting secure programming practices and handling sockets our-sleves to avoid workarounds is critical for the application's viability in a real-world scenario.