Secure File Transfer Application

By Stella Trout, Connor Haugh, Sagar Panjabi, and Omari Matthews

Overview

The application is a secure file transfer application. Authenticated users interact with an authenticated server program to carry out operations like creating a folder on the server, downloading or uploading a file, etc. After each operation, the server will provide the user with some result, such as a message confirming success and/or a file/folder location. The server can support the data of multiple users, though only one user may be logged into the server at a time. The main functional requirements of the application are that it can ensure: secure communication between the user (through the client) and the server as well as secure storage of user data on the server.

Attacker Model[1]

There are multiple attacker models, each with potentially different goals and capabilities.

- **Outsiders**
  - Goals
    - Impersonate user to server
    - Impersonate server to user
    - Access user data (without impersonation)
    - Obtain user password(s)
    - Obtain server's public key
    - Obtain server's private key
    - Observe user's activities
  - Capabilities
    - Eavesdrop on server-user communication
    - Communicate with user or server
    - Intercept server-user communications
    - Modify messages sent between server and user
- **User**
  - Goals
    - Impersonate different user
    - Access a different user's data
    - Obtain a different user's password
    - Obtain server's private key
    - Impersonate server to different user
  - Capabilities
    - Eavesdrop on communication with server to other users
    - Communicate with server (as themselves)
    - Communicate with other users
    - Intercept other server-user communications
    - Modify messages sent between server and other user
- **Server**
  - Goals
    - Read contents of user files/folders
  - Capabilities
    - All standard server capabilities (communicate with users, etc.)

[1] The first and most important assumption is that our desired server's public key is securely transmitted to the user, and that public key is certified by some reputable Certificate Authority (CA). We assume the user does not knowingly share their password or server public key with others. We trust the server to appropriately carry out user requests and protocols. We assume that the attacker cannot break cryptographic primitives

Security Requirements

Confidentiality

-User data should be hidden from everyone but the user (including the server)

Integrity

-Communication between server and user should be protected against any modification

Authentication

-User must be authenticated before making requests to the server

-Server must be authenticated to the user

-User requests to the server should be verifiable as coming from a specific user

Replay protection

-Replayed messages must be detected and discarded by receivers

Message freshness

Perfect forward secrecy

External Functions

HASH: H(m) - hashing using SHA-256

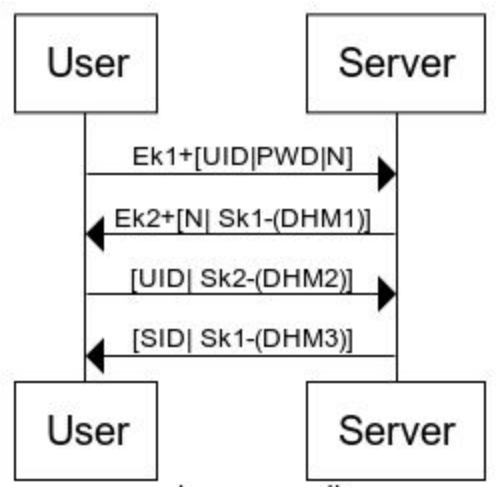
MAC: HMAC(k, m)
Where the hash is SHA-256, ISO-7816 padding is used to make the input blocks the correct size, k is the key, and m is the message. We use HMAC as it is an efficient method, and provides security against collisions.

Symmetric Key Encryption with Key length:
Encrypting payload with AES in CBC mode using a random IV with ISO-7816 padding with a 256 bit block size.

Public Key Encryption:
It is assumed that the server has a long-term secure public key encryption scheme using El-Gamal-256 with a and a verified public key, and upon registration, the user also provides a pointer to their public key and its verification. We label messages encrypted by the server keys Ek1+[M], and the user's Ek2+[M].

Digital Signature:
By using the DSA signature protocol, using the same key sets as the public key encryption, the user and server can sign messages using their long-term public keys and others can decrypt using the public key of the sender. We label messages M signed by the server with Sk1-[M], and the user's Sk2-[M]. We assume that the long-term public keys are made available

Authentication and Session Key Exchange



This protocol is based on the Diffie-Hellman Key Agreement protocol to establish two keys, the session MAC key and Message Key. The main difference is the first message, in which the user encrypts its ID, Password, and a random nonce N to the server encrypted with the server's public key. The server then computes a hash of the password and compares it to his dictionary of id-hashed passwords-public keys. If this succeeds, the server generates the parameters of DH Message One:

DHM1: $[g_1^{X1} \bmod(p_1), g_2^{x2} \bmod(p_2), g_1, g_2, p_1, p_2]$[2], with:

- Two Large Primes p1, p2
- Generators g1, g2 of a cyclic subgroup of their respective Zp * = {1, 2, …, p-1}
- Random x1,x2

The DH message is then signed using the private key of the server. The DHM1 is appended to the end of the nonce N, and all encrypted using the user's public key before being sent to the user. The user then decrypts the message using its private key, confirms the nonce N is the one it sent, proving that the server verified its password, and then begins determining the next steps in the DH protocol for DHM2.

DHM2: $[g_1^{y1} \bmod(p_1), g_2^{y2} \bmod(p_2)]$ with:

- Random y_1, y_2

DHM2 is then signed, appended to the UID, encrypted using the server's public key, and sent to the server. DHM2 is decrypted, unsigned, and used to calculate the session key and message key. The server calculates:

$K_{Mac} = (g^{y1})^{x1} \bmod p_1$

$K_{Message} = (g^{y2})^{x2} \bmod p_2$

While the user calculates:

$K_{Mac} = (g^{x1})^{y1} \bmod p_1$

$K_{Message} = (g^{x2})^{y2} \bmod p_2$

Because both the server and user provide the necessary components to compute the keys with less than computational difficulty, integrity is preserved. Authentication is also preserved, assuming the sanctity of the two public keys. The nonce prevents replay protection, as the user must always begin the key exchange process.

[2] Note that the generators and primes are sent in the encrypted message-- this is not necessary, as they can be agreed upon publicly, but this is simple.

In-App Commands

After the shared secret has been established via the authentication and key exchange protocols, we can now begin sending encrypted commands to the server using our symmetric encryption scheme with message keys ($K_{Message}$) and MAC keys (K_{MAC}) generated from our session message key ($SK_{Message}$) and our session MAC key (SK_{MAC}) respectively. Note that this process happens for each command sent to the server.

1. The client generates a fresh random number (N) via a pseudorandom number generator and sends it along with a label to distinguish what the generated key will be used for (either a message key or a MAC key) in cleartext to the server.
2. Both the client and server use either $K_{Message}$ and K_{MAC} and N to generate a 256 bit vector in the following ways:
    a. If the label appended to the received random number is “K_MESSAGE” we generate $K_{Message}$ like so: HMAC($SK_{Message}$, N)
    b. If the label appended to the received random number is “K_MAC” we generate K_{MAC} like so: HMAC(SK_{MAC}, N)

Now that both the client and the server have the necessary tools to encrypt and decrypt messages, the client can now send encrypted commands to the server and the server can send encrypted response messages to the client. The following format will be used for each message:

<table>
<tr><td>Ver</td><td>T</td><td>Len</td><td>SQN</td><td>Random IV</td></tr>
<tr><td colspan="3">random IV (cont)</td><td colspan="2">Payload</td></tr>
<tr><td colspan="5">Payload (cont.)</td></tr>
<tr><td colspan="4">Padding</td><td>MAC</td></tr>
<tr><td colspan="5">MAC (cont.)</td></tr>
</table>

- **Ver**: protocol version encoded on 2 bytes (major and minor version numbers)
- **T**: the message type encoded on 1 byte
- **Len**: the (full) message length encoded on 2 bytes
- **SQN**: the message sequence number encoded on 4 bytes, IV is a random number used to encrypt the payload
- **Payload**: the command encrypted with AES in CBC mode using the random IV
- **Padding**: ISO-7816 padding
- **MAC**: is a MAC value computed with HMAC where the key is K_{MAC} and the message is the header fields (Ver, T, Len, SQN), the IV, and the payload.

Security Guarantees

Confidentiality - By encrypting the data with a key only known to the client and the server, we can ensure that the commands are not readable by a malicious third party.
Integrity - By using a MAC, we ensure message integrity as modifying the message would cause the MAC to be invalidated.

Authentication - Because the user is already authenticated before beginning this process, we can make the assumption that these messages are only valid when being sent to and from the desired parties.

Replay protection - By using sequence numbers, we ensure that the same message cannot be received twice; thus protecting against replay attacks

Message freshness - A fresh, non-predictable nonce is generated before sending each message, thus ensuring message freshness

Perfect forward secrecy - Because we have chosen our hash function to be a secure, one-way function, having a compromised message or MAC key reveals no information about the session message and MAC keys.

Termination

If the server receives an exit command or a timeout of 4 minutes has been reached, it will execute a scrubbing of the session key, remove the userID from the list of current users, and then wait for the user to attempt to login again. If the user attempts to send a message after this window with rotten session keys, the server will tell them that they are not logged in, and nothing more, as revealing that they have a rotten key could expose several attacks.