**Network Security**

**Project Design Report**

**Arun Kumar – axb160631**

**Shivdev – sxt160530**

The objective of this project is to design and implement a secure Internet file transfer application/protocol.

**FUNCTIONAL CAPABILITIES**

An overview of the design and the functionalities of the application is given in this section. There are only two entities that are involved in this application.

* **Server**:

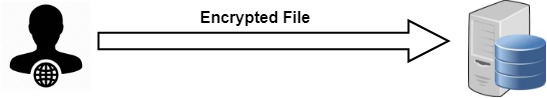
It is a central application for hosting the files.

* **Clients:**

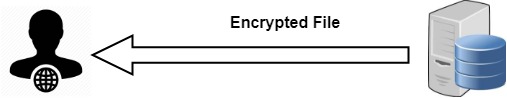
Clients can upload and download a file.

**Use cases:**

* **Upload**

****

* **Download**

****

**Security Aspects of the application**

In this section, the main threats and adversaries considered while designing this system are discussed.

1. **Confidentiality Attack (Eavesdropping):**

If the file is being sent as it is (i.e., as plain text), then the attacker will be able to sniff the traffic and use the contents of the file. This violates the privacy of the users. Thus, all the files must be encrypted before being sent to the server. We have used SHA-3(512) for encryption.

1. **Replay Attack and Impersonation:**

The malicious user can replay the same message to the user and impersonate as the legitimate user. To avoid this kind of attack, we are using different keys for every session.

**Generation of session key:** Before the start of any file transfer, the client (after authenticating the client) will choose a random number R, encrypt it with Server’s public key, send {R}Server to the server, and R could be the secret session key KAB.

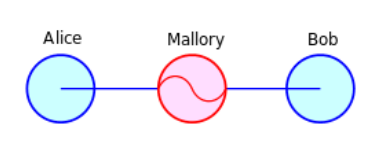
**How are we preventing replay messages with in a session**?

To prevent replay attack within a session, initially we planned to tag every message of the session with a sequence number. But, later we found out that with the encryption method that we use, where we use different session keys and IVs for each session and calculate the random values (B­­­­­­is) independently on both the server and client sides, replay attack is not feasible within a session.

1. **Reflection Attack:**

Reflection Attack is one of the main drawback of challenge-response authentication approaches using Secret key cryptography, where an attacker can open two parallel sessions and successfully impersonate one of the users. We mitigate this attack by doing authentication based on Certificate Authority (CA) signature verification. In this case no one will be able to impersonate the CA as the attacker will not be able to generate CA’s signature without CA’s private key.

1. **Man-In-The-Middle Attack (MITM):**



In the above example, assume Alice and Bob are using public key cryptography. Mallory the attacker can intercept both the messages involving the transfer of Alice and Bob’s public keys and send his own key to both. Now Mallory can decrypt any messages sent by both Alice and Bob (since the messages are encrypted by the public key of Mallory) and then he encrypts the message again with the receiver’s public key (either Alice’s or Bob’s public key) before sending to them.

But in our project since we are authenticating the server’s public key using CA signature and no one can impersonate CA or generate CA’s signature, the attacker cannot perform the MITM attack between our client and the server.

1. **Offline Guessing attack on the key:**

Since we are using SHA-3 for encryption, there are no clever existing attacks to break it other than the brute force attack which is computationally not feasible.

1. **Using Different Keys Integrity and Confidentiality**

We are using different session keys for integrity and confidentiality so that an attacker can't use one protocol to help break another. We create the integrity session key by adding 1 to the confidentiality session key.

**SECURITY FEATURES**

1. **Confidentiality**

* Encryption is performed using hashing (SHA-3(512)). In the below text, MD(m) always refers to the message digest obtained after hashing the message m using SHA-3(512) algorithm.
* This approach is like OFB mode where we generate a random number and XOR it with the message. Here, we generate that random number (Bi) using hashing as shown below.
* Sender must send the IV (Initialization vector) to receiver before sending any messages. After sending the IV, both the server and the client can generate the random numbers bi.
* We are mixing in the plain text for the generation of random numbers so that the attacker cannot generate all the Bi’s by XORing a guessed plain text with Cipher text.
* **Encryption**:

B1 = MD (KAB | IV) C1 = P1 XOR B1

. .

. .

Bi = MD (KAB | Ci-1) Ci = Pi XOR Bi

* **Decryption**:

B1 = MD (KAB | IV) P1 = C1 XOR B1

. .

. .

Bi = MD (KAB | Bi-1) Pi = Ci XOR Bi

1. **Authentication**

CA issues digital certificate to the server that contains the public key and the identity of the server. The matching private key is not made available publicly, but kept secret within the server who generated the key pair.

Authentication of the Server is carried out by using CA Signature verification. Client knows CA’s public key and every time client connects to the server, server sends the CA certificate. The client can verify the CA's signature by using CA’s public key, and thus can confirm that the public key indeed belongs to the server.

The public key of the CA could be used to determine a temporary session secret encryption key (a random quantity); these keys in such a key exchange protocol can be enciphered with the public key of the server provided by CA in such a way that only the server has the private key to read them. The rest of the communication then proceeds using the new symmetric session keys.

1. **Integrity**

MD(m) won’t work because anyone can modify the message m to m’, find MD(m’) and then append to the message. Therefore, we planned to use Keyed Hash where we concatenate the **secret integrity session key KAB** with the message m, and use MD (KAB |m) as the MAC. Since KAB is known only to the server and the client, the attacker cannot tamper the message as he/she cannot create the corresponding MAC without knowing the secret key.

But there is a subtle flaw in this kind of approach which would allow an attacker to be able to compute a MAC of a longer message beginning with m, given message m and the correct MAC for m. In general, how the hashing algorithms such as SHA-512 work is that the message is padded to a multiple of 512 bits with a pad that includes the message length. The padded message is then digested from left to right in 512-bit chunks. To compute the message digest through chunk n, all that you need to know is the message digest through chunk n-1, plus the value of chunk n of the padded message. Let's assume that an attacker would like to send a different message to server, and have it look like it came from the client. Since attacker can see both of those quantities he concatenates the padding and then whatever he likes to the end of m, and initializes the message digest computation with MD (KAB |m). This is called **as length extension attack**.

This flaw can be avoided by adding the secret key to the end of the message instead at the beginning before taking the hash. Thus, the MAC will be of the form MD (m | KAB). This alternative suffers from a minor problem that an attacker who can find collision in the (unkeyed) hash function has a collision in the MAC (as two messages m1 and m2 yielding the same hash will provide the same start condition to the hash function before the appended key is hashed, hence the final hash will be the same). Using MAC = H (key | m | key) is better, but various security papers have suggested vulnerabilities with this approach, even when two different keys are used.

No known extensions attacks have been found against the current HMAC specification which is defined as H (key | H (key | m)) because the outer application of the hash function masks the intermediate result of the internal hash. But since we are hashing the message twice, its performance may not be very good.

The Keccak hash function, that was selected by NIST as the SHA-3 competition winner, doesn't need this nested approach and can be used to generate a MAC by simply prepending/appending the key to the message, as it is not susceptible to length-extension-attacks. Hence, we are using SHA-3(m | KAB) as MAC. To reduce the bandwidth consumption, we send only first 128 bits of the MAC for verification.

**Contribution by Each Member:**

**Arun Kumar:**

* Design of Security framework.
* Design draft preparation.
* Implementation of the security aspects of the application.

**Shivdev:**

* Worked on the sockets framework.
* Implementation of the security aspects of the application.
* Documentation of the project.