CS 6740: Network Security

Domien Schepers
schepers.d@husky.neu.edu

Amogh Pradeep
pradeep.am@husky.neu.edu

December 3, 2018

1 Introduction

In this document we propose an architecture and design for a secure instant messaging system.

The assignment for Problem Set 4 is described as follows:

> The architecture can have a server (but not necessarily, it is up to you to make the decision). If a server is used the messages between the users should not go through the server (with the exception of the first discovery messages).
>
> Your protocols should satisfy the following constraints:
>
> - The users only needs to remember a single password. The client application should not remember the users passwords.
> - If you are using public/private keys, the client application cannot remember the private/public key of the users. You can however assume that the client application knows the public key of the server (if you are using public/private keys).
> - Your system should provide mutual authentication (user ¡-¿ server), message integrity and confidentiality.
>
> Describe in detail the security protocols that you are proposing for the whole system. Remember that you are not allowed to use a TLS/SSL library or a complete existing protocol. You are allowed to use cryptographic libraries that provide encryption, hashing, etc.

> You must provide authentication, integrity, and confidentiality of user and server messages (if a server is present). Users must be authenticated using only a username and password.
>
> In particular, if you support resistance to denial of service attacks, end-points hiding, perfect forward secrecy, protection for weak passwords, and/or message secrecy against a malicious server, you will gain more points.

> You will get more points for core security, addressing weak passwords, resiliency to denial of service attacks. Bonus points can be obtained for additional services such as identity hiding.

2 Design Assumptions

The following assumptions are made by the system we propose.

2.1 Server

We assume the following with respect to the server.

1. *Identity authenticity*: Users that connect to the server assume that the identities supplied by the server are legitimate i.e, have been authenticated by the server.
2. *Secure long living key*: The long term private key associated to a given server is assumed to be safe and only known by the server.

2.2 Client

We assume the following with respect to the client.

1. *Trusted hardware*: The hardware used to run the client is assumed to be safe and trusted.
2. *Server public key*: The clients are assumed to have knowledge of a given servers public key.
3. *Amnesic*: The client stores no information in the long term.

3 Protocol Specification

The instant messaging architecture we define has a number of different operations that lead to secure communications between two users that belong to the network.

3.1 Register on a server

When a new user wants to start using our instant messaging application, they first need to register themselves on the server. The registration operation has the following steps.

1. *Talk layer security*: When a new user wants to register themselves to the server, they first establishes a secure session with the server using our talk layer security protocol4. All further communications are encrypted using the session keys.
2. *Registration request*: They then send a registration request to the server with a username and password (encrypted with the session key obtained after TLS).
3. *Server response*: The server responds with either a success message or a failure message for the registration request.

3.2 Login to a server

Each time a client wishes to connect to the server with the credentials of a pre-registered user, he performs a user login operation. The login operation has the following steps.

1. *Talk layer security*: The client first establishes a shared secret with the server using our talk layer security protocol4.
2. *Login request*: The user sends all necessary information required to authenticate themselves and enough cryptographic material to be contacted by another user connected to that server.
3. *Server response*: The server responds with either a success message or a failure message for the login request.

The login request has the following information.

1. Username.
2. Password.
3. A user's identity key for the current logged in session.
4. N one time pre keys.
5. Listen information (IP address and port).

Once successfully logged in the client periodically performs the login operation to signal to the server that it is still alive and to update prekeys.

3.3 List registered users

The server supports a list feature which lists the usernames that are currently online on that server. the following steps are followed to achieve the same.

1. *Talk layer security*: The client first establishes a shared secret with the server using our talk layer security protocol4.
2. *List request*: The user sends a list request to the server.
3. *Server response*: The server responds with either a success message and the list of users connected to the server, or a failure message for the list request.

3.4 Fetch user keys bundle from a server

When a user wants to contact another user for the first time in a given session(having logged in to the server), they ask the server for a prekey bundle. The prekey fetch operation has the following steps.

1. *Talk layer security*: The client first establishes a shared secret with the server using the talk layer security protocol 4.
2. *Prekey request*: The user sends all necessary information required to fetch prekeys for a given username.

3. *Server response*: The server responds with either a valid prekey bundle or a failure.

The Prekey request has the following information in it.

1. Username of the user to be contacted.

It is worth noting here that the Prekey request does not contain any identifying information as to who is sending the request. This is a measure to prevent the server from recognizing which user is trying to contact which other user in the network.
Having received a Prekey request, the server responds with either an error or a user prekey bundle that consist of the following.

1. The user's identity key for its present logged in session.
2. One one time pre key.
3. The listen information for the user (IP address and port).

Errors the server would throw would be one of the following:

- *User not online*: If the username does not exist or the registered user is not online.
- *Invalid request*: If the request is malformed.

3.5 Send client message

Having performed a prekey fetch 3.4 ; a client is ready to contact the user it wishes to communicate with.
The client performs an XDDH5 computation and use the output secret from that algorithm to initiate two KDF chains6. One each for sending and receiving.
Having initiated these chains to send a message the following steps are performed:

1. *Generate message key*: This involves using the chain key state to produce a message secret. The message secret is further divided into a Message key, Mac key and an IV to be used to encrypt the message.
2. *Update KDF chain*: Move the state of the KDF sending chain to the next step and delete the information regarding the previous state.
3. *Encrypt and MAC*: Using the Message key and IV obtained in the first step, the message to be sent is encrypted using AES-256 in CBC mode with PKCS#7 padding. The Mac key is used in the HMAC-SHA-256.

AES-256 in CBC used with a secure key gives us the property of confidentiality. We use a 256 bit key because we believe that modern computers have enough processing power to deal with these while making it much harder to be broken than the 128 bit mode. HMAC-SHA256 provides us with integrity and a fixed length output which is idea to us.
A special case of sending a client message exists which is the one right after the XDDH protocol is performed for the first time by Client 1 with intentions to send Client 2 a message. This message must include additional information such as which of Client 2's prekey was used by Client 1 and the ephemeral public key used by Client 1 in the XDDH step.

3.6 Receive client message

When a client receives a message, the following steps are performed:

1. *Derive message key*: The message key is derived from the sending chain, making sure to store intermediate message keys(if any are skipped), to deal with out of order messages.
2. *Update KDF chain*: Move the state of the KDF receiving chain to the step specified in the message and store intermediate keys.
3. *Verify and Decrypt*: The hmac attached to the message is verified and if it passes, the ciphertext is decrypted.
4. *Delete used keys*: Keys that were used for decryption can now be deleted safely.

A special case of receiving a client message exists which is the case when a Client (say 2) receives a message from another (say 1) for the first time. This message has information which mentions which of Client 2's prekeys were used by Client 1 and the ephemeral public key it used. These are required to successfully perform the XDDH protocol5 with Client 1 to initiate the state of the sending and receiving KDF chains6

3.7 Typical protocol flow

Figure 3.7 contains a typical flow of the protocol steps required for Client 1 to send the first message to Client 2 and for Client 2 to decrypt the same.

4 Talk Layer Security

In this section, we describe in detail the steps involved in the talk layer security protocol(TLS). TLS is used to produce a shared secret between an entity and the server. The initiator of the connection needs to have knowledge of the server's public key. Our goal does not involve authenticating the user at this stage in the protocol stack, this is used as a bootstrapping for a secure channel above which ot

4.1 TLS Roles

This protocol involves two roles; roles and associated assumptions are listed below.

- *Server*: A server that has a long living private identity key.
- *Client*: A client that wishes to talk to a server, has knowledge of said servers identity key.

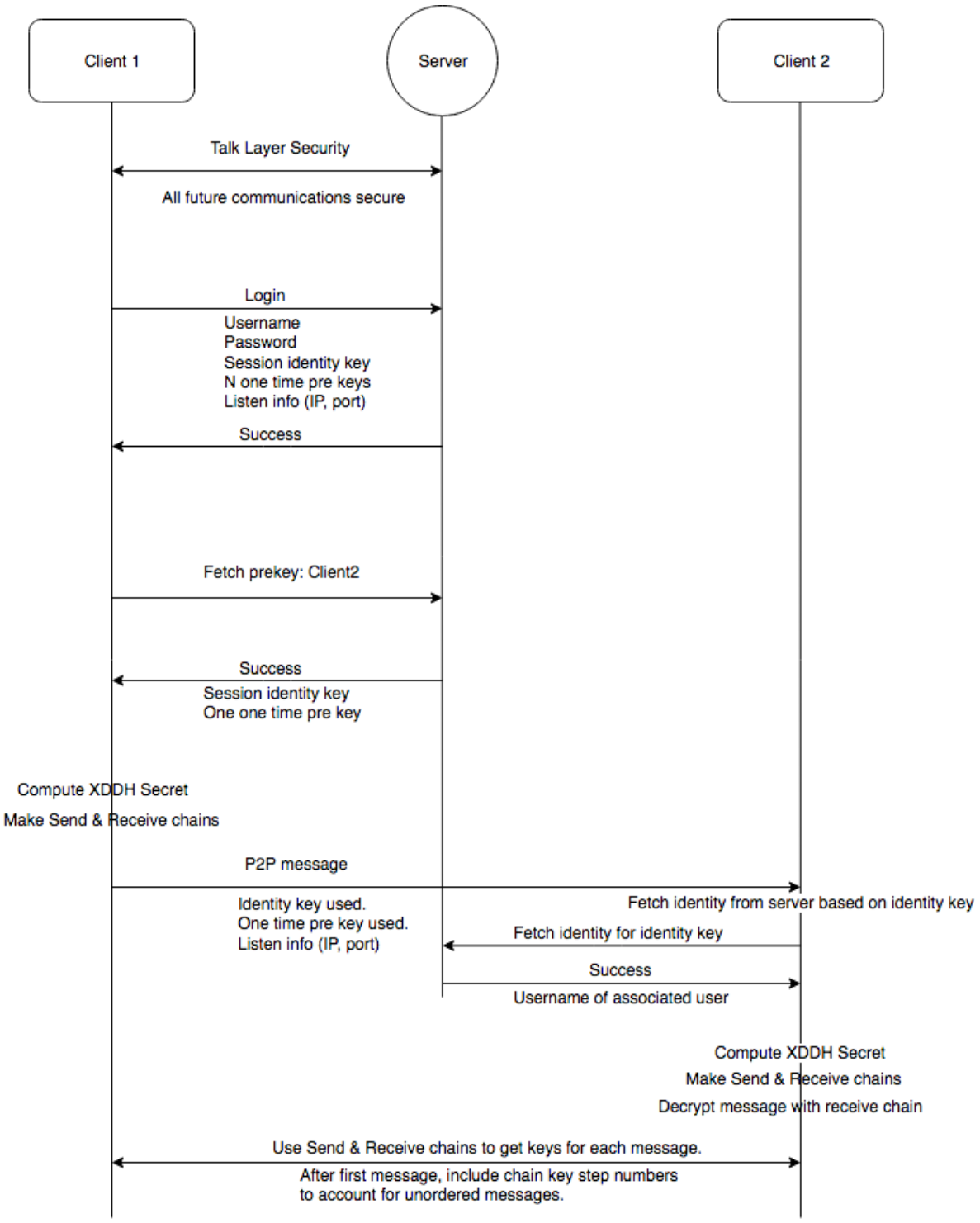

Figure 1: Client logging in and sending a message.

4.2 TLS Protocol

Given a `Client` and `Server` as defined above, the following steps are performed:

1. *Client hello*: $Client \rightarrow Server : g^{c} \bmod n || K_{cs[0]}\{ nonce_1 \} || HMAC\left(K_{cs[1]}, msg\right)$

   Where $msg = g^{c} \bmod n || K_{cs[0]}\{nonce_1\}$.

   The client first computes a diffie-hellman half $g^{c} mod\, n$ and mixes it with the servers public key to get a random. Using this random they derive two keys.

   $K_{cs[0,1]} = KDF\left((g^{c} mod\, n)^{s} mod\, n\right)$ Where g^{s} is the server's long living public key.

   On receiving this, the server computes $K_{cs[0,1]} = KDF\left((g^{c} mod\, n)^{s} mod\, n\right)$ Where g^{s} is the server's public key, using the corresponding s which is it's private key.

   It is worth mentioning that given that the server is the only one with knowledge of its private key, it is the only one who can come up with the message defined in the next step.

2. *Server challenge*: $Server \rightarrow Client : g^{s} mod\, n \,||\, K_{cs[0]}\{ nonce_1 \,||\, nonce_2 \} || HMAC\left(K_{cs[1]}, msg\right)$

   Where $msg = g^{s} mod\, n \,||\, K_{cs[0]}\{ nonce_1 \,||\, nonce_2 \}$.

   At this point, the client has enough information to compute $K_{cs[0,1]} = KDF\left((g^{s} \bmod n)^{c} mod\, n\right)$.

   Once that is computed, the client verifies that the $\texttt{nonce}_1$ it attached is valid and produces the final message.

3. *Client challenge response*: $Client \rightarrow Server : K_{cs[0]}\{nonce_2\} || HMAC\left(K_{cs[1]}, msg\right)$

   Where $msg = K_{cs[0]}\{nonce_2\}$.

   The server then checks if nonce_2 is valid. If it is, the shared secret can be used for whatever future communications between the client and server.

4.3 Convention

The convention used above have the following meaning.

- nonce_{n} stands for a 128-bit nonce picked randomly and used only once.
- $[]_{\text{Ks}}$ Stands for asymmetric encryption with `Ks`'s public key.
- $K_{cs}\{\}$ Stands for symmetric encryption with secret key $K_{cs} = (g^{c} mod\, n)^{s} mod\, n$

We discuss in detail the different ways adversaries could attack this protocol in section 8

5 XDDH Key Agreement Protocol

This section describes the "XDDH" (or "Extended double Diffie-Hellman") key agreement protocol. XDDH established a shared secret between two parties that mutually authenticate each other based on identity keys supplied to them by a trusted intermediary.
The trusted intermediary is only used to verify identities and doesn't have access to the shared secret established at the end of the protocol. The security considerations of this are discussed in **??**.

5.1 XDDH Roles

This protocol involves three roles; roles and associated assumptions are listed below.

- *Server*: A server that stores Identity keys and pre keys.
- *Client 1*: Wants to establish a secret with Client 2.
- *Client 2*: Wants to allow parties like Client 1 to contact it and established a shared secret with it.

5.2 XDDH Keys

XDDH uses the following keys elliptic curve public keys.

- IK_{C1}: Client 1's identity key.
- IK_{C2}: Client 2's identity key.
- EK_{C1}: Client 1's ephemeral key.
- OPK_{C2}: Client 2's one time pre key.

All public keys have corresponding private keys stored on the respective clients.

5.3 XDDH Protocol

Given a server that has the following stored on it:

1. Identity keys for Client 1 and Client 2.
2. N one time pre-keys for Client 1 and Client 2.

When Client 1 wants to establish a secret with Client 2, it does the following.

1. Fetch a prekey bundle from the server.
2. Perform the following calculations:

   $DH1 = DH\left(IK_{\mathrm{C1}}, IK_{\mathrm{C2}}\right)$

   Mix its own private key with the public identity key for Client 2 in a diffie-hellman key exchange.

   $DH2 = DH\left(EK_{\mathrm{C1}}, OPK_{\mathrm{C2}}\right)$

   Mix an ephemeral private key with one of Client 2's one time prekey.

   $Secret = KDF\left(DH1||DH2\right)$

   Once the secret has been calculated, Client 1 deletes the ephemeral key in order to obtain forward secrecy.
3. Send a message to Client 2 with associated data allow it to calculate the secret.

   Here, the associated data would be $\mathrm{OPK_{C2}}$, $\mathrm{IK_{C1}}$ and public key $\mathrm{EK_{C1}}$.
4. Client 2 first looks up the identity of the user associated to the public key $\mathrm{IK_{C1}}$.

5. Client 2 then calculates the following:

   $DH1 = DH\,(IK_{\mathrm{C2}},\, IK_{\mathrm{C1}})$

   Mix it's own private key with the public identity key for Client 1 that it just received in a diffie-hellman key exchange.

   $DH2 = DH\,(OPK_{\mathrm{C2}},\, EK_{\mathrm{C1}})$

   Match the private key to the one time prekey it received and mix that private key with the ephemeral public key it received from Client 1.

   $Secret = KDF\,(DH1||DH2)$

Thus after Client 1 initiates a connection and sends the first message, both Client 1 and Client 2 share a common secret and can communicate using it. For the client sending the first message, it is necessary to attach extra information so the receiving client knows which prekey was used with the ephemeral key in the `DH2` calculation. It is also necessary to include the ephemeral public key used.

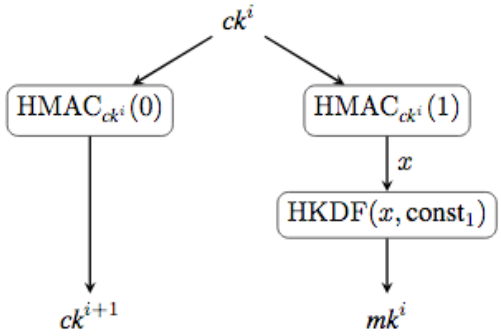

Figure 2: KDF Chain step operation.

6 KDF Chains

A `KDF` is a cryptographic function that takes in a secret random `KDF key` and some input data and returns output data. The output data is indistinguishable from random to any entity that has no knowledge of what the key is. Thus, a `KDF` satisfies the requirements of a cryptographic pseudorandom function. If the key is not random and secret, the `KDF` should still provide a secure hash of its key and input data. Using HMAC and HKDF constructions with a secure hash algorithm meet the requirements of a `KDF`.
A `KDF chain` is constructed using such a `KDF` function by splitting the output of the function parts. One of which is used as the next rounds `KDF key` and the other as an output key. Figure 5.3 shows how a `KDF chain` is initiated using HMAC and HKDF.

7 Protocol Guarantees

Using the above presented constructs, we ensure that our protocol guarantees the following properties:

- *Server authenticity*: Communications initiated by a client to the server are guaranteed to be only read by that server using TLS 4 sessions.
- *Replay resistance*: Communications with the server are relay resistant thanks to TLS 4. Communications with other clients are replay resistant thanks to the use of send and receive KDF chains 6.
- *Client-Client mutual authentication*: Communications between two clients are ensured to be mutually authenticated given a legitimate server serving keys. This property arises as a usage of the XDDH protocol 5.
- *Client-Client end-to-end authentication*: Two clients that communicate in our system use end to end authentication and are the only ones that possess keys to encrypt/decrypt their communications.
- *Client-Client perfect forward secrecy*: Using the KDF chains 6 provide two clients that communicate with each other unique keys for each message they send each other. Thus, once a message has been sent, the keys are deleted ensuring perfect forward secrecy.
- *Client-Server perfect forward secrecy*: Establishment of a new TLS 4 session for each operation a Client performs with a Server, allows us to discard keys instantly. Thus ensuring perfect forward secrecy for Client-Server communications.
- *Endpoint hiding*: Messages sent over the network in the clear do not contain enough information for an adversary to infer identities using just the traffic. Thus, we say that our endpoints are hidden and resistant to passive identification attacks.
- *DoS protection*: DoS resistance is obtained by implementing strict rules to the number of requests that can be made to the server and clients.

8 Security

We discuss the various security threats we consider that could attack our messaging infrastructure and list out how we mitigate such attacks through various aspects of our protocol.

8.1 User spoofing

If a user registers with an unsafe password, the client login step 3.2 could be performed by an adversary that has knowledge of the password. This is an online attack, as it isn't possible to mount the attack without getting a response from the server to check whether a password is valid or not. Thus, it could be prevented by the server by setting a limit on the number of login attempts possible for a given registered username on it's system.
This is the only possible attack vector for a user with a weak password, as the password itself is used in just this step of the protocol. All other steps are implemented using safe to use cryptography which do not depend on the password in any way.

8.2 Denial of service attacks

There are two roles in our system, `Server` and `Client`. The server's address is known publicly and could be attacked by sending a number of invalid requests. Such an attack could be thwarted by using denyhosts or fail2ban.

9 Discussion

Does your system protect against the use of weak passwords? Discuss both online and offline dictionary attacks.

Refer 8.1. Our system is resistant to offline attacks. For an online attack, it is a similar model to websites that let you login with a username and password. An attacker could possibly try to guess the password by trying to login multiple times.
To prevent such online attacks, we suggest that the implementation of the server track IPs making request and limit the number of logins it accepts from a particular IP by number of requests every minute or some other suitable limiting mechanism. The scope of efficiency of limiting mechanisms is out of the limits of this document.

Is your design resistant to denial of service attacks?

A one server approach with this design could end up resulting in vulnerability to DoS attacks. Having said that, we'd like to emphasize that our protocol does not enforce the fact that the system must contain just one server. Thus, it is possible for a multi-server scenario that would be resistant to such attacks. Such a group of servers could follow one of two methods to store user keys. One would be to be synchronize keys between each other. The other would be to include details of which server to register onto to get prekeys for a given user.
Both methods would be equally effective resisting DoS attacks but a detailed analysis of their pros and cons is outside the scope of this document.

To what level does your system provide end-points hiding, or perfect forward secrecy?

Our system provides both end point hiding and forward secrecy. Refer 7 for our specific definition of these terms with respect to our protocol. Perfect forward secrecy arises from the fact that keys for server communications are short lived (used for one operation of a few messages); for the client client communications, `KDF chains` provide us with new keys for each message being sent which can be disposed instantly after encryption thus ensuring forward secrecy.
End point hiding arises from the fact that client identities are never sent in plain text. A network adversary cannot passively infer identities using traffic fingerprinting.

If the users do not trust the server can you devise a scheme that prevents the server from decrypting the communication between the users without requiring the users to remember more than a password? Discuss the cases when the user trusts (vs. does not trust) the application running on his workstation.

In our model, the server is used to produce identity authenticity. The servers themselves can not decrypt any of the communications between two clients given that our protocol is end-to-end encrypted.