The Mitnick Protocol

By Nathan Goodman and Nam Luu

Assumptions

- The server is always up and running. (We do handle the case when the client is ran before the server starts, but we DON’T handle the case when the server crashes while the clients are still running, crash the server while the clients are running will lead to the clients crash as well)
- Each client has an username and a password. The usernames are public, but each client only knows his/her password.
- The server knows all the usernames and passwords
- The server has an RSA key pair
- Both clients know the server’s public key
- (Implementation assumptions will be mentioned in implementation part)

Services

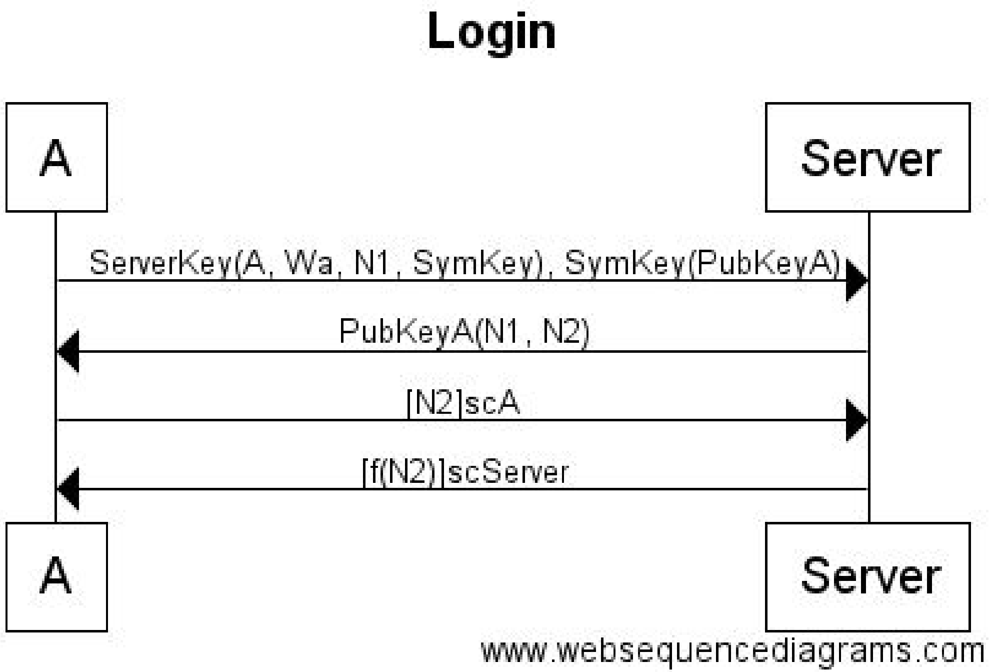
- Connects clients together so they can exchange with each other
- Authenticates clients from a trusted server
  - Client A can be sure that they’re talking to client B (and vice versa)
- Resilient against man in the middle attacks by making use of RSA verification
- Perfect forward secrecy as a result of Diffie Hellman Implementation

Protocol Notation

T(M): message M is encrypted with key T

[M]scT: message M is signed with T’s secret key

Authentication Protocol


Login
A
Server
ServerKey(A, Wa, N1, SymKey), SymKey(PubKeyA)
PubKeyA(N1, N2)
[N2]scA
[f(N2)]scServer
A
Server
www.websequencediagrams.com

Authentication Protocol Explanation

A sends its identity (including username, IP address, port number), its password, a random nonce, and its public key (generated each time a user logged in) to the server.

The server decrypts the message, gets A’s public key, use this key to encrypt N1 and a new nonce N2, and send it back to A. A decrypts, get N2, and send this nonce signed with its private key to the server. Once the server receives this message, it sends ACK message back to A so A can know that it is properly logged in.

*Note:

- In our original design, the first message is {A, Wa, N1, PubKeyA}serverKey. However, during implementation, we realize that PubKeyA is too long to encrypt with serverKey, so we put instead a symmetric key to replace PubKeyA, and use this symmetric key to encrypt PubKeyA

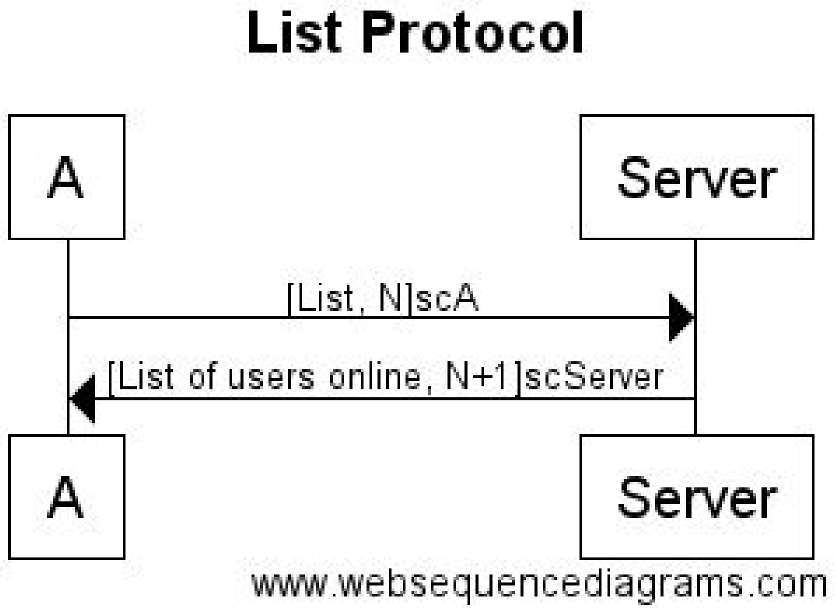
Threat Discussion

N1 is used to make sure that the attacker cannot impersonate the server, since he cannot decrypt N1. N2 is used to make sure that the attacker cannot use replay attack to login as A.

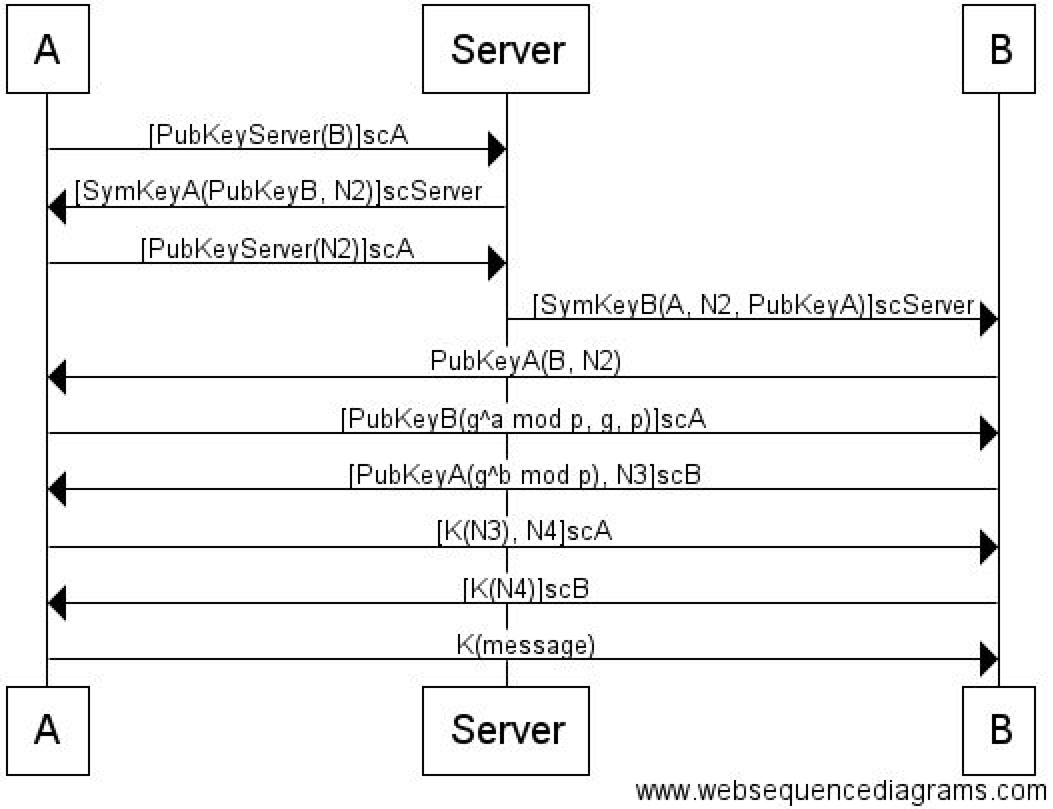
The attacker can’t impersonate A either because he doesn’t know Wa

Weak password: If A has weak password, the attacker can’t brute force it, because in order to do so, he also needs to bruteforce N1 and A’s public key.

List Protocol


List Protocol
A
Server
[List, N]scA
[List of users online, N+1]scServer
A
Server
www.websequencediagrams.com

Key Establishment Protocol


A
Server
B
[PubKeyServer(B)]scA
[SymKeyA(PubKeyB, N2)]scServer
[PubKeyServer(N2)]scA
[SymKeyB(A, N2, PubKeyA)]scServer
PubKeyA(B, N2)
[PubKeyB(g^a mod p, g, p)]scA
[PubKeyA(g^b mod p), N3]scB
[K(N3), N4]scA
[K(N4)]scB
K(message)
A
Server
B
www.websequencediagrams.com

Key Establishment Protocol Explanation

The objective of this protocol is to let both A and B knows each other’s public key and the shared secret key.

First A tells the server he wants to talk to B. The server sends back B’s public key, and nonce N2 to A. A sends N2 encrypted by the server public key so the server can know that A has already received B’s public key. Once this happens, the server sends information of A, nonce N2 (same nonce as before), and A’s public key to B. B will use this key to encrypt the message sent to A, this message include B information (IP address, port), and nonce N2.

After A and B know each other’s public key, they can start establish a shared secret key using Diffie Hellman protocol and use this key to encrypt their messages.

Note: We also make a small modification to the original design by using a symmetric key to encrypt A and B’s public key. The reason is the same as in the authentication protocol. Here SymKeyA and SymKeyB are the server and client’s shared keys that are established during authentication

Threat Discussion

Replay attack: An attacker can try to replay the first message. However, he will not get far in the protocol: The server will send [PubKeyA(PubKeyB, N2)]scServer back to A and since the attacker doesn’t know what A’s secret key and N2 (N2 is different every time) are, he cannot proceed. The attacker can’t impersonate the server to replay the second message to A either, because this message is supposed to be signed with the server’s secret key, and the attacker can’t do that.

Man in the Middle: The attacker cannot man-in-the-middle at any time during the protocol since every message is signed with the sender’s private key.

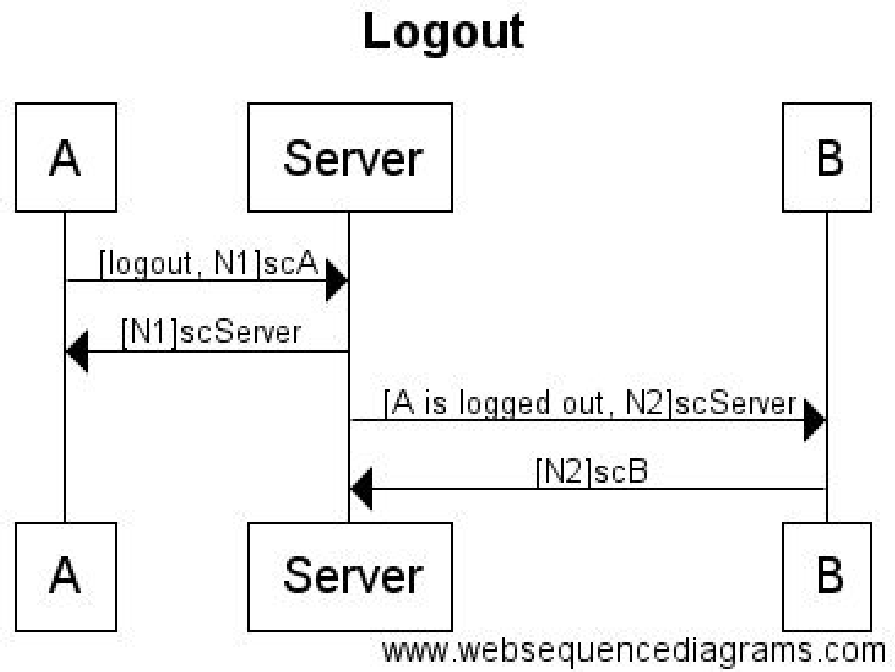
Reflection Attack: Using same argument as above, since all messages are signed with the sender’s private key, the attacker can’t impersonate any party to decrypt any of the nonces.

DDos Attack: The attacker can’t try to blast the server with massive junk messages since all messages are supposed to be signed by a valid secret key hence if the server sees so many messages coming from a source that are not signed properly the server can just block that source.

Threat Discussion (cont)

Because A and B establish a shared secret key on their own and not using the server, even in the case that the server is not trusted, the server cannot decrypt the message between A and B because it doesn’t have Kab.

Logout Protocol



Logout Protocol Explanation

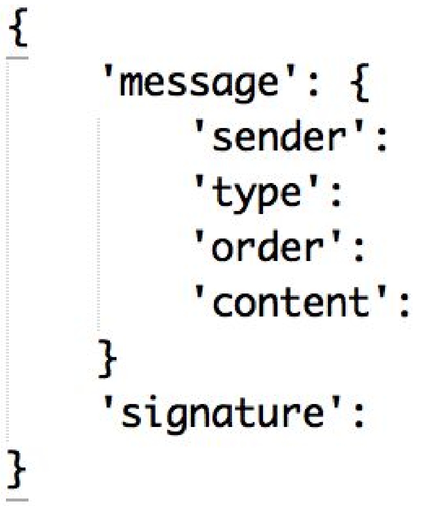
If A wants to logout, it needs to tell the server so that the server can delete A’s current public key. The nonce that A sends with the message has to be the same as the nonce that A uses to login. This is important, because otherwise an attacker can just launch a replay attack and log A out.

After that the server can tell B that A is already logged out.

Threat Discussion

Replay attack: The attacker can try to replay the first message to log the user out, however, since the secret key is different every time, if the attacker sends this message the server will ignore since it cannot decrypt this message.

Implementation


```
{
    'message': {
        'sender':
        'type':
        'order':
        'content':
    }
    'signature':
}
```

Each message has the following format:

Sender is the username of the sender. (Message from server to client doesn’t have this field)

Type is either ‘key establishment, ‘login’ or ‘list’

Signature is the signature of the message, which is message signed with the private key of the sender. For messages that doesn’t need to sign (based on the protocol), the field’s value is an empty string.

Implementation

Due to time constraint, we are not able to implement logout protocol, but user can logout by hitting Ctrl-C, and the server will handle that event (i.e. crossing that user off the active user list).

All other protocols are fully implemented.

Implementation (cont)

In src folder there are following files: client.py, server.py, util.py, server_public_key, server_private_key, users.json.

We assume that the client can ONLY use the following file: server_public_key, util.py

The server can read util.py, server_public_key, server_private_key, users.json

Right now, users.json store raw password of users, which is never a good idea in real life. However, it is good enough for the scope of this assignment. We can always improve the security here a little bit by storing the hashed password in users.json and then server will have to hash password before comparing it with the value in the file. However, doing so will slow down the processing of adding/removing users while testing so we decide not to do it.

This software is protected under the MIT++ License Agreement