Secure Messenger Design

Aishwarya Vinod, Mallory Gilligan

Setup

Server-Client Model Architecture:

Our design is a server-client model where the server will be used for initial discovery, key exchange, and user authentication but messages will be transmitted between the clients (peer-to-peer). The server and clients will store the shared server-client keys for the duration that the client is logged in and the server is running.

Assumptions

Type of Keys/Who knows what:

- User only knows their password
- Client software does not store anything regarding users
- Client software is configured to connect to the server's address and port at runtime
- Client software is trustworthy
- Client will know the g and p used by the server to store gWmodp at run time
    - Will be shared securely in the real world
- Server knows which users are registered and their gWmodp's
- Clients and Server stores the generated server-client key only for the duration of the logged in session
- Clients temporarily store client-client generated key for the duration of the session.

Protocols

Authentication Protocol:

We have decided to use password-derived properties in the client, specifically Secure Remote Protocol (SRP) because it allows for mutual authentication of the server and client, protects against any eavesdropping, and eliminates the need to transmit passwords over the network. SRP provides us with a shared key between the client and the server, also allowing for future secure communication between the client and server. A new shared session key will be created every time a user logs out and logs back in. This also provides PFS because the session key is forgotten after that session and cannot be used to decrypt past communication

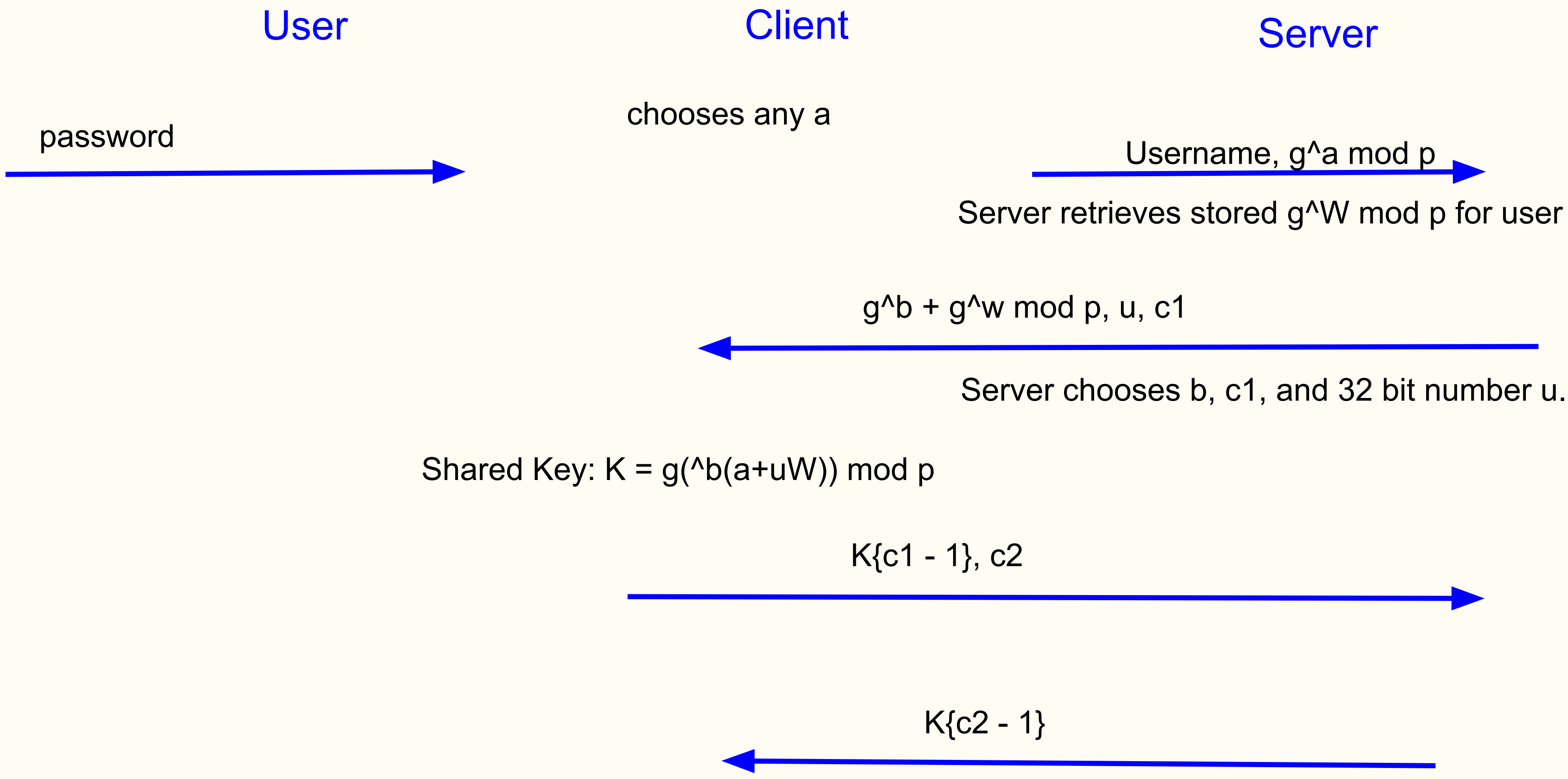
Registration:

Registration is dealt with through a preconfigured file.

Login:

- User provides username and password to client application (assume that the user is already registered).
- Client generates a random private key and computes g^a modp and sends this along with the username to the server.
- Server sends back g^b+g^W mod p, u, c _1
- Client and server compute shared key g^(b(a+uW))mod p
- Client sends c_1 - 1 encrypted using the new shared secret key to prove to the server that it is the user and also sends c_2
- Server decrypts c_1 - 1 using the shared key and verifies that this is the right user and also sends back c_2 - 1 encrypted using the shared key.
- Once the client verifies this, mutual authentication is achieved
- AESGCM is used for encryption

Login:


User
Client
Server
chooses any a
password
Username, g^a mod p
Server retrieves stored g^W mod p for user
g^b + g^w mod p, u, c1
Server chooses b, c1, and 32 bit number u.
Shared Key: K = g(^b(a+uW)) mod p
K{c1 - 1}, c2
K{c2 - 1}

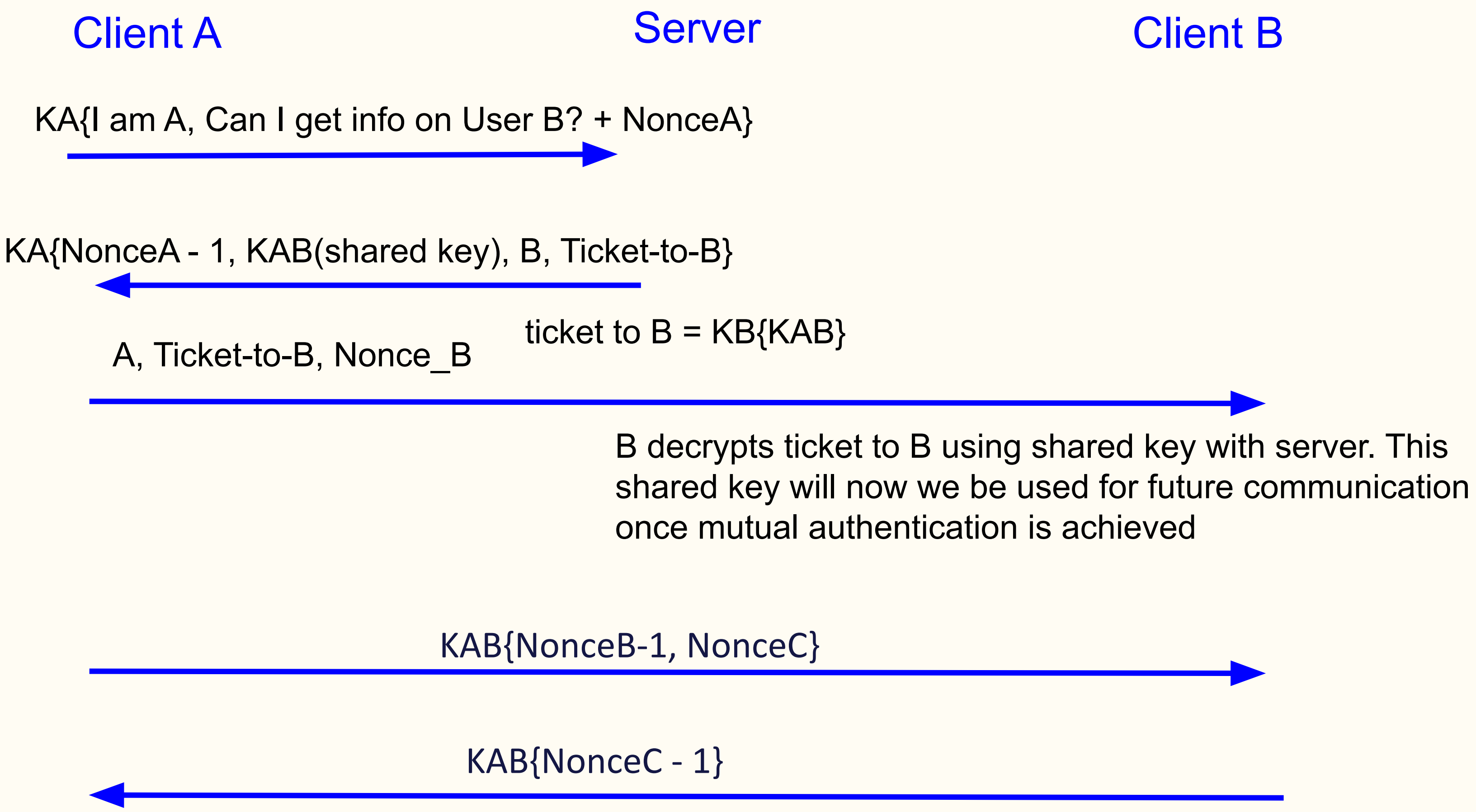
Protocols (Cont.)

Message protocol:

- Client sends a message to the server encrypted using the shared key K with the server which was created during login requesting info about another user.
- The server will respond back with a message encrypted with their shared key which contains another nonce, a shared key that the two clients can use for initial communication, and the IP address of client B so that client A can message them, This message will also contain another message encrypted using the shared key between the server and client B containing the client to client shared key - ticket to B
- Client A decrypts the part of the message that is encrypted with its shared key with the server and retrieves the shared key with the user it wants to talk to and their address
- Client A will send the message encrypted with the shared key between client B and the server to client B (ticket to B) which contains the shared key between the two (client A cannot decrypt this message). The client will also send its identity and a Nonce.
- Client B will decrypt the client-client shared key using the key it has shared with the server.
- Using the retrieved shared key, client B will decrypt the Nonce and send back the Nonce - 1 along with its nonce
- Client A verifies that it can decrypt the message and that the Nonce sent is the nonce that it had sent (- 1)
- Client A sends back Client B's nonce - 1
- Once Client B verifies that the nonce sent by Client A is the nonce that it had sent - 1, Mutual authentication is achieved.
- Future messages are encrypted using this shared key until the session ends

Logout Protocol:

- When the user decides to log out, the client securely deletes/forgets the session key and any other information stored during the session. So, the client application is left with storing no information relating to the user.
- The client message will also send a logout message to the server.
- The server will then forget the user's shared key, IP address, and other any information other than gWmodp.


Client A
Server
Client B
KA{I am A, Can I get info on User B? + NonceA}
KA{NonceA - 1, KAB(shared key), B, Ticket-to-B}
ticket to B = KB{KAB}
A, Ticket-to-B, Nonce_B
B decrypts ticket to B using shared key with server. This shared key will now we be used for future communication once mutual authentication is achieved
KAB{NonceB-1, NonceC}
KAB{NonceC - 1}

Discussion

Services:

Mutual authentication: Mutual authentication is ensured through the use of Secure Remote Protocol (SRP) and Needham Schroeder.

Offline Dictionary Attacks Our choice to use SRP protects against this. When a user registers with the server, instead of storing the password directly, the server stores the gWmodp of a user where W is the password. So, even if the server database was compromised, it is computationally very difficult to crack a password.

Confidentiality: Confidentiality is achieved by establishing shared symmetric keys between clients during communication and between the client and server during login. Messages are encrypted with the key using AESGCM, making them unreadable to unauthorized parties.

DOS protection: DoS protection is achieved through rate limiting on connection requests from clients (a user can only be logged in once at a time), preventing excessive requests from overwhelming the system and ensuring that legitimate users can access resources without interruption. There will also be a temporary account lockout after 4 failed password attempts. This also protects against online dictionary attacks.

PFS: Perfect Forward Secrecy is provided because past communications cannot be decrypted even if future keys are compromised. The shared key between client to client and client and server are all forgotten after a session ends (user logs out or server shuts down)

ATTACK PHASE!

An analysis on the design of 5 different teams

(and an attack demo!)

P.S - Please access the vulnerability report for a more detailed write up

James Landry

- **Priority of Security Risk**: Ranked as having the most significant security vulnerability due to the transmission of private keys in plain text over a network.
- **Primary Vulnerability Exploitation**: Anyone on the network can potentially capture these private keys using simple packet-sniffing tools such as Wireshark, as evidenced by our capture tests.
- **Potential Risks of Private Key Exposure:**
  - Decrypting encrypted data intended for the key's owner.
  - Impersonating the key's owner.
  - Signing malicious data.
  - Engaging in other harmful activities.
- **Implications of Private Key Exposure:**
  - All past and future communications using this key are compromised until the key is changed.
  - An attacker could access sensitive information not only currently but also any past and future messages if the keys are not rotated frequently.
- Messages sent within the "send command" are not encrypted; they are transmitted as plaintext.

Randall Ventrone & Jonathan Chery

- **Unsecured Transmission of Symmetric Key**: Symmetric keys are transmitted in plaintext, compromising the security of encrypted communications.
- **Insecure IV Generation**: Initialization Vectors (IVs) are not randomly generated, making the encryption method predictable and vulnerable.

```
# If Authentication is a success, obtain the new port nummber and symmetric key from the server and create a cipher
if 'Authentication Successful' in status:
    _, _, assigned_port, symmetric_key = status.split(',')
    PORT = int(assigned_port)
    extract_symmetric_key = re.findall(r"'(.*?)'", symmetric_key, re.DOTALL)
    symmetric_key = extract_symmetric_key[0].encode('utf-8')
    cipher = Cipher(algorithms.AES(symmetric_key[:16]), modes.CBC(symmetric_key[16:32]), backend=default_backend())
```

- **Improper Session Management**:
  - No session timeouts or checks to prevent the reuse of old session tokens or symmetric keys.
  - Multiple logins on different terminals using the same credentials can cause DOS.
- **Password Hash and Salt Exposure**: Both password hashes and salts are transmitted in plaintext, allowing for replay attacks and vulnerability to dictionary attacks using common tools like hashcat or John the Ripper.
- No Rate Limiting: Lack of rate limiting on authentication attempts, allowing attackers to brute force passwords.
- No Perfect Forward Secrecy: Old session tokens or symmetric keys can be reused to impersonate a user, compromising future sessions.
- Improper error handling and some operational issues

Time to display attacks!

CY4740-final-code (names not provided)

- **Lack of Real Authentication:** No effective user or server authentication, leaving the system open to unauthorized access.
- **Insecure Password Storage:** Passwords stored unencrypted in the server-side user dictionary, susceptible to memory dump attacks and data breaches.

Operational and Design Flaws:

- **Excessive Error Information:** Error messages provide excessive details, which could be exploited to gain insights into the system architecture or to execute more targeted attacks.

```
aishuvinod@Aishwaryas-MacBook-Air-2 cy4740-fi
nal-code % python3 clie
nt.py
Enter your username: aishu
Enter your password: password
Enter command: list
Failed to get user list: Ciphertext length mu
st be equal to key size.
Error
```

- **Vulnerability to DOS Attacks:** The server starts a new thread for each client connection without rate limiting, making it vulnerable to resource exhaustion from numerous simultaneous requests.
- **Insecure Key Exchange:** Key exchange processes, though non-functional, would be vulnerable to man-in-the-middle (MiTM) attacks if implemented as currently designed due to the unencrypted transmission over insecure channels.

Ali Bobi and Eitan Leinwand

Public Key Verification Vulnerabilities:

- **Lack of Public Key Authentication**: Currently, there is no mechanism in place to verify the authenticity of public keys being exchanged, leaving the system vulnerable to impersonation attacks.
- **Use of Certificates**: Implementing digital certificates could provide a method for authenticating public keys, ensuring that the keys belong to their claimed owners and are not tampered with during transmission.

Replay Attack Susceptibility:

- **Absence of Nonces or Timestamps:** The system does not utilize nonces (numbers used once) or timestamps in its communications, which are essential for preventing replay attacks

Ella Isgar and Nate Krauss

Operational Issues:

- **Send Command Not Functional:** The "send" command is non-operational, which disrupts basic messaging functionality, impacting user experience and utility.
- **Flawed Logout Mechanism:** The logout process is problematic, potentially leaving user sessions improperly terminated and susceptible to unauthorized access.

Private Key Compromise: The security of the key exchange hinges on the secrecy of the private key used to encrypt the AES key. If this private key is compromised, the entire encryption scheme is at risk.

Security Measures and Suggestions:

- Implement a more robust encryption protocol for key exchanges, such as using ephemeral keys or integrating a key agreement protocol like Diffie-Hellman.
- Revamp session management to ensure that logout procedures fully terminate the session and clear all associated data, reducing the risk of session hijacking.