Nathan Ong, Cameron Buck, & David Stein
CS 1653
March 29 2013



# Team Project Phase 4

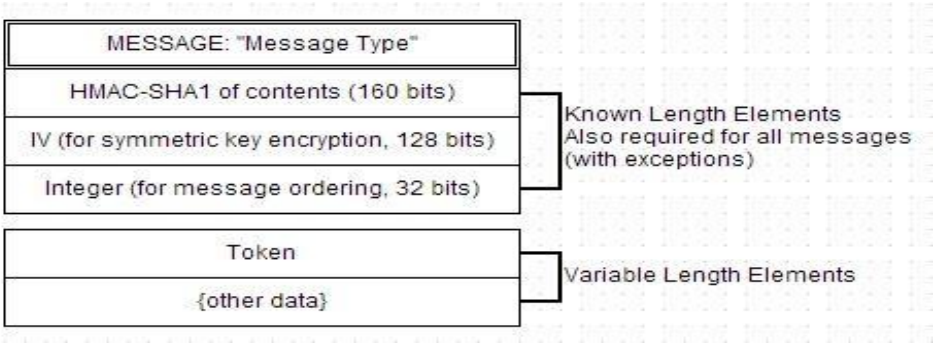
**Introduction**

The threats addressed in this phase deal mostly with an active attacker. Threat 5 deals with reordering, replaying, and modifying messages in transit by a malicious router. Threat 6 deals with file leakage by an untrusted file server. Threat 7 deals with token theft, described as the file server attempting to steal a user's token. Generally speaking, the ideas presented here are not necessarily new cryptographic techniques. Most of the mitigation comes with merely adding more information to the messages that are sent across the network or to the token that proves the authentication of a user.

For our 128-bit AES keys, 1024-bit RSA key pairs, and 160-bit SHA-1 hashes, we use Java's SecureRandom class to introduce cryptographically-strong randomness. We use the lightweight Bouncy Castle API as our security provider because of its maturity. Our key sizes are accepted by the cryptographic community as being secure through tests over times, but this is always changing. SHA-1 is probably the weakest link out of our cryptographic functions, and the largest known RSA attack has been 768-bit. Bouncy Castle takes care of the AES implementation and rounds for us. We chose counter (CTR) because of it being accepted as one of the better modes for AES. Even if computers become more powerful, AES can easily be strengthened with larger key sizes and adding additional rounds. RSA, too, can be strengthened against future attacks with larger key sizes.

**Threat 5: Message Reorder, Replay, or Modification**

In order to mitigate this threat, some changes need to be made to the messages that are being sent across the network. By assigning an order to the fields of messages, checking messages for validity will be simple. The message will be in the following order:



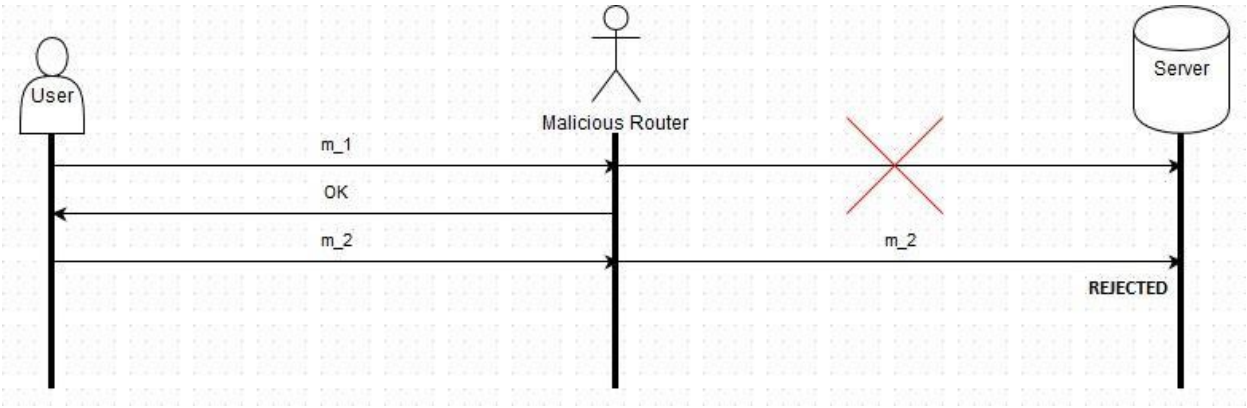

The IV was determined from the previous phase of the project.

The messages for setting up a secure connection is different, since the message requires a symmetric key encrypted with the server's public key to be sent to the server. The specifics for these kinds of messages can be found in the write-up for phase 3. Note that the token is also required for all messages, excluding setting up the secure connection.

REORDERING

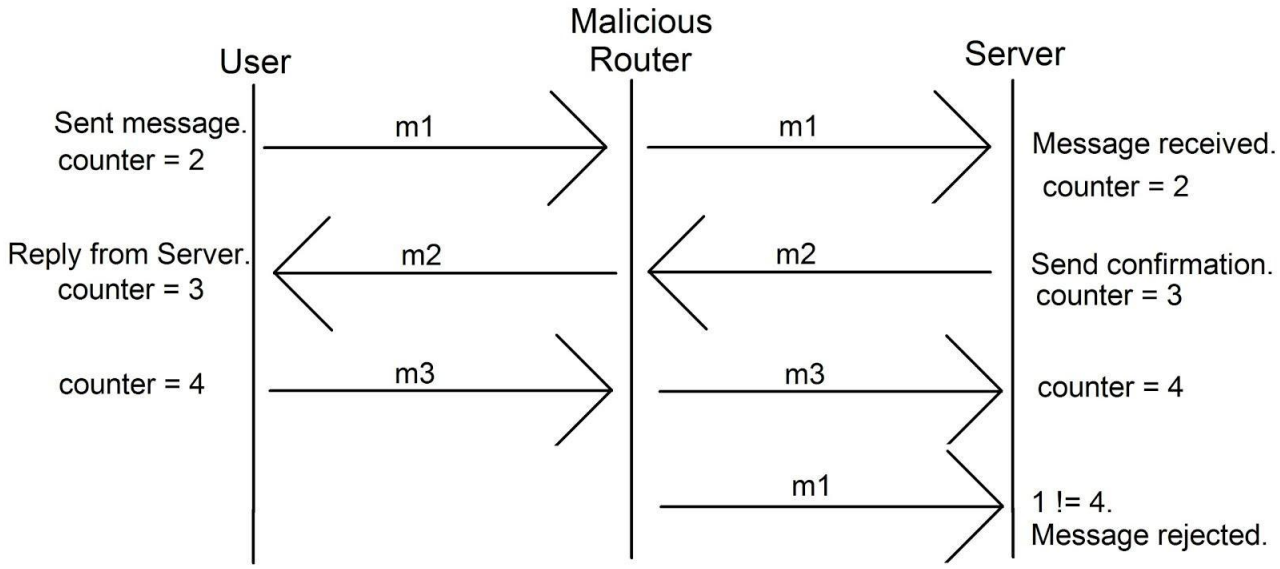
Reordering messages primarily does not make a huge impact. Reordering messages creates a denial of service to the user, generally unpreventable because there is also a possibility that the network dropped the packets, rather than the possibility that a malicious router forcibly re-ordered the messages. However, the biggest issue may come from uploading a file. If a user wants to upload a file, the file server requires that chunks of the file be sent in order. If they are sent out of order, then the file may become corrupted or unusable.

To mitigate reordering, we will number all envelopes starting from one and add one to each envelope. The server and the user should check the envelope number. If the number does not follow the numerical order based on the session, then it should reject the message. For example, if user sends a message with integer n, then the server will reply with integer n+1, and then the user will send a new message with integer n+2, etc.
Due to strange issues with the group server about not sending objects correctly without disconnecting and reconnecting, this numbering will be ignored for the group server, as a new session is always created per message sent.



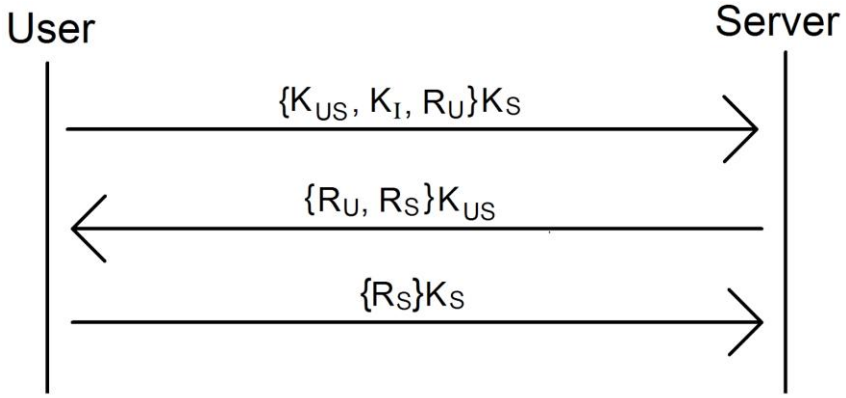

The diagram shows an example of a numbered message being stolen by a malicious router, and the second message instead being sent, where it is rejected by the server because it was expecting 1, not 2.

REPLAYING
There are two types of replay situations that could occur. In the first situation, a malicious router could replay messages within a session. In the second situation, a malicious router could replay an entire session from start to finish. The message ordering / numbering to prevent reordering already prevents an **inter-session replay attack**. If message $m_1$ is sent to the server by the user, the malicious router could send $m_1$ to the server again. The server will reject message $m_1$ because it was expecting message $m_2$. A simple counter prevents inter-session replay attacks.




Replay attacks where a malicious router records an **entire session** and plays it back to a server is also a problem. The server can prevent session replays by using a challenge system.

1) The user initiates a session with a server by encrypting a message with 1024-bit RSA with the server's public key. The server's public key is obtained from the certificate authority. The message will contain a 128-bit AES key, a 160-bit integrity to be used for the HMAC function, and a 128-bit user challenge for the server to solve.
2) The server replies with a message encrypted with the 128-bit symmetric key the user sent. The message contains the 128-bit user challenge, authenticating the server with the user, and a new 128-bit challenge for the user.
3) The user replies in cleartext with the server challenge after decrypting with the symmetric key. The user authenticates with the server by decrypting the challenge.




MODIFICATION

To prevent modification, we generate a cryptographically strong MAC (message authentication code) via a hash function to form an HMAC. The HMAC will contain a concatenation of the

contents of the envelope (message, IV, number of message, token and data). A symmetric key, separate from the key used to encrypt the message but of the same size (128 bits), will be used to account for the integrity of the message. This second symmetric key will also be sent during the secure-connection-setup phase. HMAC-SHA-1 (160 bit) will be used as follows:

**HMAC(k, m)** = H( (k $\oplus$ opad) || H((k $\oplus$ ipad) || **m** ) )
where **m** = message || IV || number of message || token || data

To verify that the request/response has not been tampered with the recipient can compute the HMAC to verify its validity. Using the HMAC is considered secure because in order to modify a message the adversary will need to recomputed the HMAC to make a modification. If any piece of the message is tamped with, it is reasonable to assume that utilizing an HMAC will account for the modification and will provide sound evidence that the message should be ignored.

**Threat 6: File leakage**

The threat of files being served to unauthorized or malicious users is constant. In the event that an adversary gains access to a file server and requests a file, mechanisms must be in place to prevent this user from being able to view the contents of the file. If the adversary is able to retrieve a file from the server, it is important that its content be encrypted as a layer of security. An encrypted file that must be decrypted with the proper group credentials is essential to keep this user from gaining access to its content.

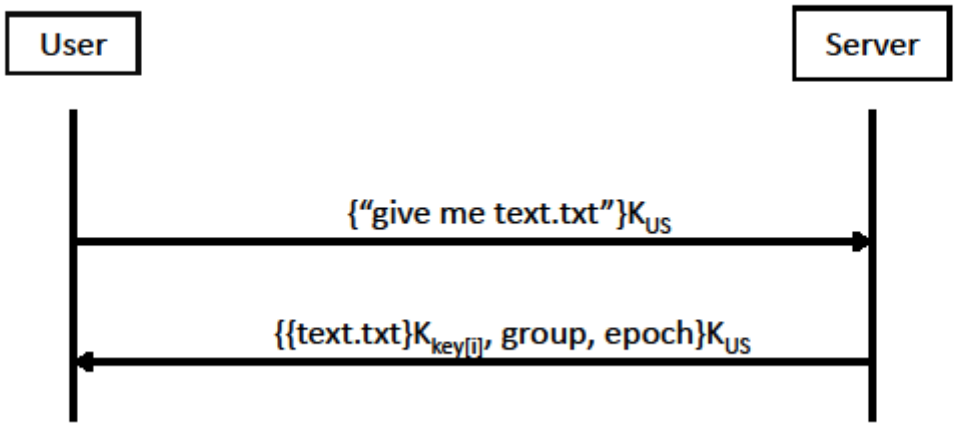
We want to avoid the problem of files being leaked by the file server, and users removed from groups should not be able to see new files. Our policy for leaked files will be backward-security compliant. If user u is part of group g at time t, then u is kicked out of g at time t+1, a user could technically still download and unencrypt all of the files from a poorly kept file server, since it is assumed that the user could have stored all of the symmetric keys for the group and the files for that group. If u is removed from g at time t, then the user will not be able to decrypt any files added from time t+1 and any time afterward without the key.

When a new user is added to a group, they can decrypt all files associated with that group. A user is able to download all files from the group, so only backward security makes sense for our policy. To enforce this policy, we propose the following mechanism:

For downloads:

1) A user requests a file from file server. The request is encrypted with the 128-bit AES key.
2) The file server sends the encrypted file to the user. The encrypted file is sent with the group name and epoch number.

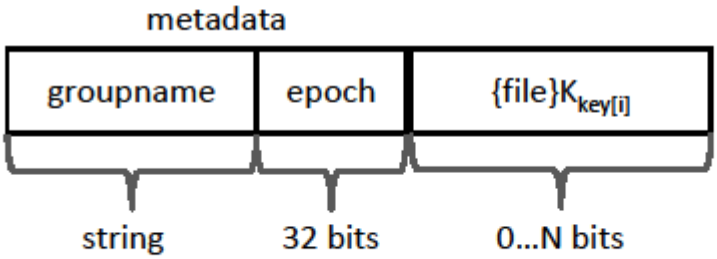
3) The User unencrypts the file based on the group name and epoch found in the meta data of the file. The epoch indicates which 128-bit AES key must be used to unencrypt the file.



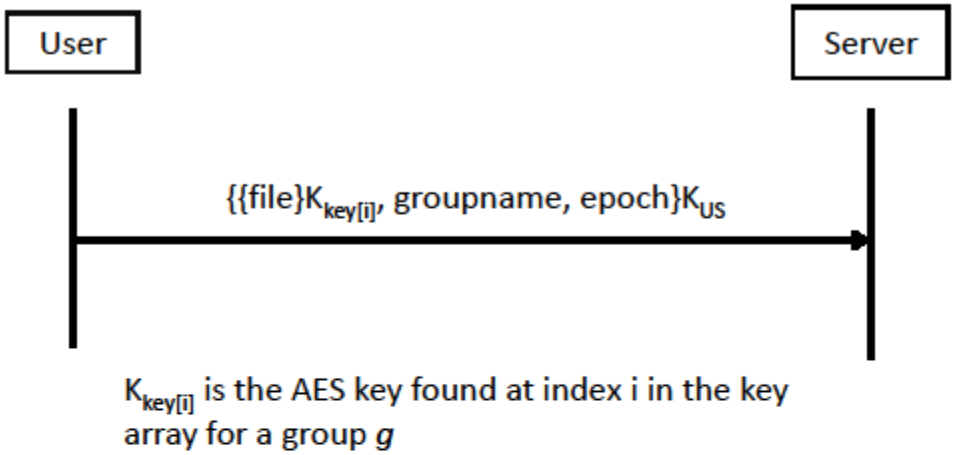

After downloading, the user will retrieve the encrypted file. The user requested a list of 128-bit AES keys for each of his or her groups at the start of each session. The user will recover the epoch number and group name from the metadata of the downloaded file. By referencing that index in the key array for that group, the user is able to decrypt files downloaded from the file server. Malicious users modifying the epoch has the same effect of deleting a file: legitimate users are unable to access the file.

For uploads:

1) A user encrypts a file with the latest group key. A group key is a 128-bit AES key generated by the group server.
2) The user sends the group name and epoch of the file as part of the metadata. The epoch is a counter and references the array index in the key array for that group. The epoch will be a 32-bit integer because two billion (signed) should be plenty of user deletions for the scope of this project.



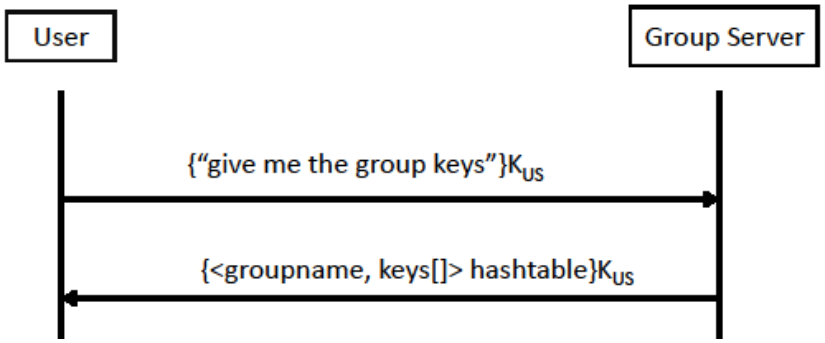

3) The user sends the file to the file server, and this message is encrypted with a 128-bit AES key.




To see this is secure, we examine all steps and attempt to break them. While uploading, the user must encrypt the file with the latest key. The user is then expected to encrypt the file with the 128-bit AES key and send the group name and epoch with the file. This encrypted file is then sent to the file server. Should the file server leak the file, the key is still needed to decrypt it.

For each session:

1) A user requests all keys from the group server. These are 128-bit AES keys.
2) The group server will return all keys associated with groups the user is in. The keys are encrypted with the 128-bit AES key that was established at the beginning of the session. The group server will return a hash table where a group name will map to an array of keys. The index of a key in the array is the epoch. Note that a user possibly will get a subset of the hash table because the user is not in all of the groups.
3) Every time a group is updated, the updated key list is sent back.




**Threat 7: Token Theft**

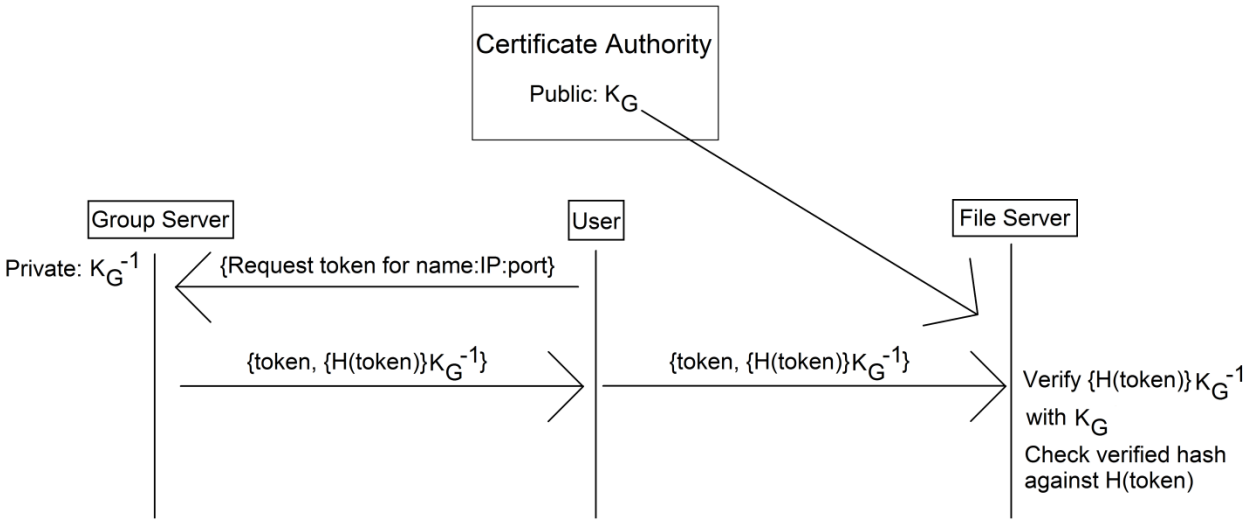
When a user wants to connect to a file server, the user requests a token from the group server. For this phase, more security is needed since file servers are untrusted. It is possible for a file server to try a user token on another server. Possibly, file server $f_1$ is a benign server, but file

server $f_2$ is malicious. $f_1$ will not try to use other user tokens if they did not come from the file server because it is benign. However, $f_2$ will try to steal the user's files from other servers by giving stolen tokens to other users.

To prevent a malicious file server from fooling other file servers or users by using a stolen user token, other file servers need to confirm or deny authorship of requests. Proving authorship can be achieved by adding information in the user token specifying the server for which the token was intended. If a user requests a user token for file server $f_1$, the token will can only be used on file server $f_1$. The user will tell the group server the name, IP address, and port number of a file server with which it wants to interact. The group server will make a new token as in our Threat 2 (diagram below) with the file server's **name**, **IP**, and **port** will be added to the token. The group server will sign the token as usual, so no legitimate file server will accept an unauthorized token.

The new token has the following categories:

- Issuer: the Group Server which is the author of the token
- Subject: the user's username
- Groups: the user's groups concatenated in alphabetical order
- File Server (FS): name, IP, and port number to prevent token forgery




Token = {issuer, username, groups*, FS Name:FS IP:FS port}

* The groups of a user are concatenated in alphabetical order.

A 160-bit SHA-1 hash is used to hash the user token. The group server signs the hash of the user token with its private key, allowing other users to verify the group server was the author of the token. This satisfies **correctness**. By correctness, everyone can verify the group server signed a user token. Adding the file server's **name**, **IP**, and **port** to the token satisfies **security**. By security, nobody can steal a user token because user tokens can only be used for a

specified file server. If a user requests to interact with file server $f_1$, the token will only be accepted if the group server signed it. In the case of a malicious file server $f_2$, $f_2$ will give files to unauthorized users; anyone who asks for a file will be granted a file. Threat 6 deals with file leakage through encryption.

**Proof of validity of previous threat mitigation**

The modified protocols implemented in this phase of the project still address threats T1 through T4. Users are still issued tokens which can be verified by the servers. The tokens cannot be tampered with since they are hashed and signed. Trusted certificate authorities will also still issue public keys to the user for the requested servers. Users must also still be authenticated with a username and password. The manner in which we handle File Leakage in this phase by requiring files to be decrypted with the group's symmetric key issued by the group server will also help to mitigate the threat of unauthorized file servers covered in phase 3. Files should not be leaked to users who are not properly authenticated with a correct username and password pair. The protocols established in this phase of the project will not inhibit the mechanisms established in the previous phase of the project. In actuality they will add features that will further secure the system and mitigate the threats described in both phase 3 and phase 4.

**Conclusion**

Overall, the mitigation of the threats are mostly isolated cases. Adding a nonce or a number for portions of threat 5 can stand as-is. However, these additions will alter the total message hash, which is used to prevent modification. As for dependencies, all of the mechanisms described here require the infrastructure created in the previous phase. For example, we require that the symmetric key structure is in place (encrypting a symmetric key with the server's public key).

We struggled the most in figuring out how to eliminate file leakage. We knew that the file server had to store the files encrypted. However, it was debated as to who should do the encryption/decryption, either the user or the group server. We reasoned that sending the file to the group server is unfeasible, especially for larger files, so instead, the group will send the user the key to decrypt/encrypt if the user has the credentials for the group.