SE425: Communication and Information Security	Assignment 3
Semester 2 5779	Due: 30 June 2019 at 11:55pm

SCAFTIA: SCAFTI with Authentication

This assignment is an extension of the Secure Chat and File Transfer with Integrity (SCAFTI) tool you developed for assignment 2. You will add a new feature to the tool - key distribution and authentication based on a version of the Needham-Schroeder key distribution protocol. You can read more about the protocol at: `https://en.wikipedia.org/wiki/Needham%E2%80%93Schroeder_protocol`.

1 Introduction: Adding Authentication and Key Distribution

The tool built for the second assignment protects the secrecy and integrity of chat messages and files sent using AES-CTR and HMAC-SHA256. Now we're going to add authentication and fresh key distribution protection to the files sent (we'll leave chat messages alone).

The Needham-Schroeder protocol changes the structure of the tool a bit. Since Needham-Schroeder is a centralized protocol, we will need to add a server to help with the key distribution. To minimize the change to the communication protocol, we'll just apply the key distribution and authentication to part of the tool's tasks - file sending. We certainly could expand it to work on chat messages too if we wanted.

We'll still use HMAC-SHA256 to protect all messages (with the exception of on message below) and files sent using an Encrypt-then-MAC scheme, regardless of whether they are sent using the shared key or a session key. As in assignment 2, our scheme will be that all messages or files are encrypted first using AES-CTR. Afterwards, an HMAC digest is computed over the ciphertext.

Best practices dictate that when encrypting, authenticating, and computing a MAC, separate keys should be used (one key = one task). Therefore, we'll add a third key to the tool (the server authentication key), meaning that the tool's GUI must be configurable with the following parameters:

1. A user name. A user name is a string consisting of printable UTF8 characters which will fit in a Java string. The user name **may not** contain whitespace characters or the '@' character.
2. The list of neighbors in the group - each neighbor has an IP address and port.
3. A (shared) text encryption password. The password is converted to an AES key by:

   (1) Convert the password to bytes using the UTF8 encoding

   (2) Hashing the resulting bytes with SHA2-256 hash algorithm

   (3) Use the 256 output bits as the key for the AES cipher

   This is the same encryption password used in the previous assignment.
4. A (shared) text MAC password. The password is converted to an HMAC secret by:

   (1) Convert the password to bytes using the UTF8 encoding

   (2) Hashing the resulting bytes with SHA2-256 hash algorithm

   (3) Use the 256 output bits as the secret for the HMAC algorithm

   This is the same MAC password used in the previous assignment.
5. The IP address and port of the authentication/key distribution server.
6. A (non-shared) personal authentication password. The password is converted to an AES key by:

   (1) Convert the password to bytes using the UTF8 encoding

   (2) Hashing the resulting bytes with SHA2-256 hash algorithm

(3) Use the 256 output bits as the key for the AES cipher

This is a new password which was not present in the previous assignments. It is known to just the individual user and the authentication server.

Note: Items 1–4 are unchanged from assignment 2.

2 Main User Story

Alice, Bob, Claire, and Dan have been using SCAFTI for a while now and are quite happy with it. They can send encrypted messages between themselves with protection from changes by attackers. Things began to get a bit complicated, however, when Bob wanted to send Claire some highly secret cat photos. Somehow Alice and Dan had a strong aversion to cats (and photos of cats), so it was critical that neither Alice or Dan see the files. Ideally, they'd be able to send files between them using their shared encryption key, but an incident in the past got Bob and Claire worried. One time (after a particularly wild key signing party), Alice logged into SCAFTI using Bob's user name. She sent and received messages using his user name for a few hours before she noticed what had happened. The incident reminded Bob that despite their having secrecy and integrity, there was no internal authentication build into the system. Bob and Claire met and decide it is time to upgrade SCAFTI to include user authentication - SCAFTIA.

While it's possible in theory to do distributed authentication without a central server, Bob and Claire realize that centralizing authentication makes a lot of sense despite the introduction of a single point of failure. The *Needham-Schroeder shared key protocol* is fairly light-weight and only requires one user to contact the server. It also doesn't require the server to proxy messages, so it's a fairly low price to pay for authentication.

The implications of adding authentication and a server are that Alice, Bob, Claire, and Dan need to establish personal long term keys with the server. They set up a secure server host by a third party (Crazy Eddie's Discount House of Hosting, if you're interested) and get it an internet connection. They then each take turns entering a private key into the server to get it up and running.

From then on, their new SCAFTIA tools work almost identically to the old SCAFTI ones. They each configure their SCAFTIA tools with the server's IP address and port and the personal key (in addition to the existing encryption and integrity key).

The next time Bob wants to send Claire a highly secret cat photo, his SCAFTIA sends a request to the server, receives a session key for the file from the server, and sends it to Claire for validation. Claire's SCAFTIA validates the message and makes sure that it's Bob on the other end. Once the validation setup is done and the session key is established, they use the session key to transfer the file. This way, neither Alice or Dan can understand the file sent, even if they were to somehow record the sent traffic.

To test the resilience of the new protocol, Bob tries to get a key for Claire using Alice's user name (he sends a message to the server as if he was Alice). The server responds appropriately, sending Bob a token for Claire that he can't open or use. The resulting failure convinces Bob that their file transfers are now secure and can't be seen by others.

Epilogue One night after dozing off reading a Bruce Schneier book on security, the ghost of Roger Needham appears to Dan in a dream and warns him about a key reuse attack proposed by Denning and Sacco. Dan turns over and mumbles, "I never lose my keys."

Coda Claire goes for a walk one night, looks at the stars, and thinks wistfully of how nice it'd be to extend SCAFTIA to use public key authentication and digital certificates instead of all of the symmetric keys she and the others need to manage now (there's 6 keys now and they'll expire at some point). She knows, however, that her development time is limited right now. Maybe next year.

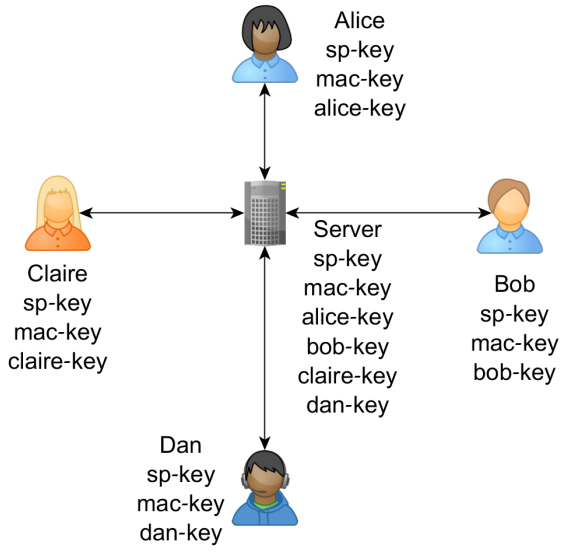

Figure 1: Users, Server, and Keys

3 Adding an Key Distribution Server

The Needham-Schroeder symmetric key distribution protocol works by using an server which performs user authentication in a bit of a roundabout way. The server doesn't check the user's password or identity directly, instead in responds to users with messages that only the correct user will be able to understand. Consider the tool setup shown in Figure 1. In it, each user (and the server) have a copy of the shared encryption key (*sp-key*) and the shared MAC key (*mac-key*). Each user has a personal key (*alice-key*, *bob-key*, etc.) which are known only to the individuals and the server. Table 1 shows which keys are found at which computers in the network.

Table 1: Table of keys. A ✓ indicates that the user has the key indicated.

	sp-key	mac-key	alice-key	bob-key	claire-key	dan-key
Alice	✓	✓	✓			
Bob	✓	✓		✓		
Claire	✓	✓			✓	
Dan	✓	✓				✓
Server	✓	✓	✓	✓	✓	✓

The keys are used for different purposes. The shared encryption key (*sp-key*) is used for encrypting chat messages. The MAC key (*mac-key*) is used for protecting the completeness of chat messages and files. Sending files is treated differently than chat messages since it requires the sender to consult the server first and receive a session key. The client contacts the server, identifies himself, and requests a key for a specific user. The server responds to the request by encrypting the response with a key that only the client would know (assuming it's not lying about its identity). Therefore, if a client connects to the server and lies about its identity (*ex.* Alice contacts the server and claims to Bob), the server will respond to Alice, but the (lying) client won't be able to do anything useful with the response; it's encrypted with the key that Alice doesn't have.

The steps in the protocol we'll use are summarized in Figure 2. The steps include the classical Needham-Schroeder protocol (without the fix of Denning and Sacco). The steps are as follows:

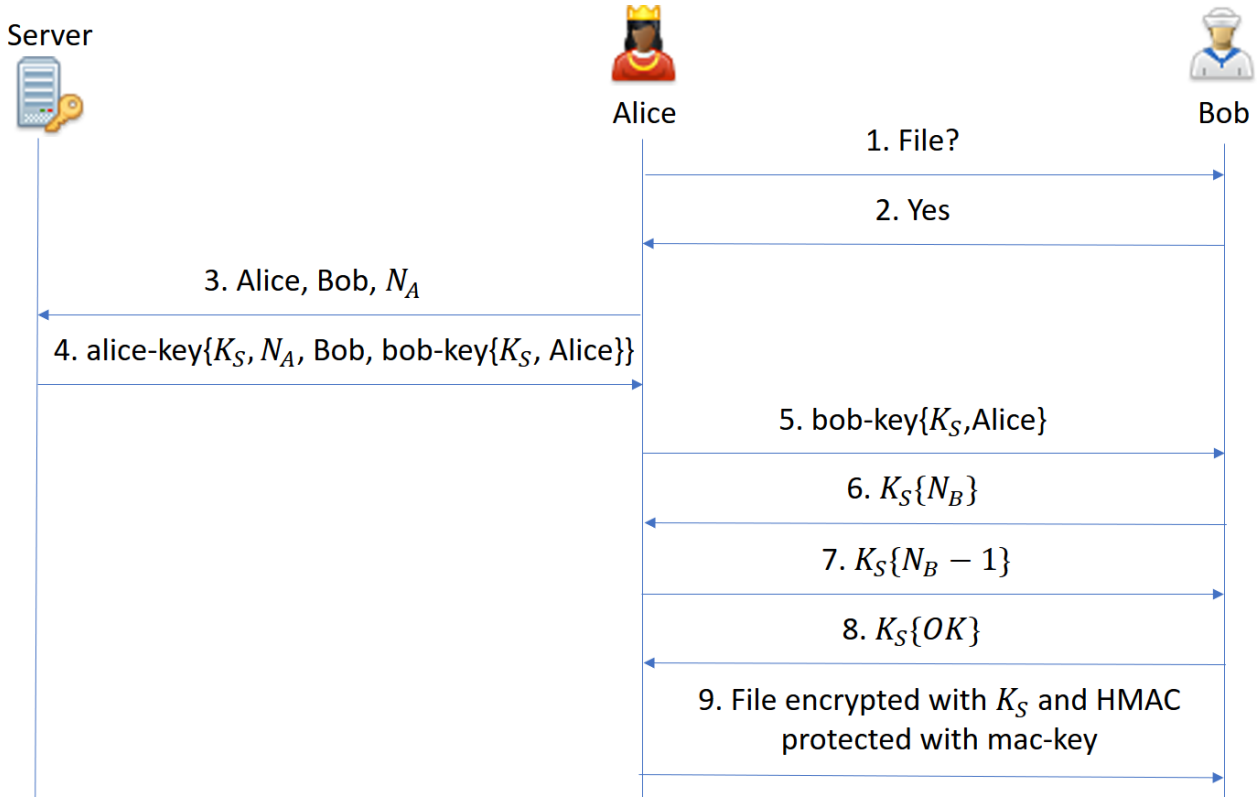

Figure 2: Steps in Needham-Schroeder protocol when sending a file

1. Alice offers Bob a file. The offer is encrypted using AES-CTR and protected using HMAC-SHA256 (same as in the previous assignment).

2. Bob agrees to accept the file. He sends Alice an Ok message and the port number to send all subsequent file related message on. The message is encrypted with AES-CTR and protected using HMAC-SHA256 (the same as in previous assignment).

3. Alice sends a request to the server to receive a session key for Bob. The message includes the name of the sender, the name of the recipient, and a random *nonce* N_A. The message does not need to be encrypted, but it must be protected with an HMAC digest using *mac-key*.

   Note: This is the only message in the protocol which does not require encryption. You may encrypt it anyway if you wish using AES-CTR and *alice-key*.

4. The server finds Alice's (the sender's) key *alice-key* and Bob's (the recipient's) key *bob-key* in its internal database and prepares a random new 256-bit session key (K_S). The server then encrypts K_S and "Alice" (the sender's name) with *bob-key* using AES-CTR (the result is called *the token*). The server calculates the HMAC digest on the token using *mac-key*. The server then concatenates the token, the HMAC digest on the token, the IV of the token, K_S, and N_A and encrypts them at once with Alice's key *alice-key* using AES-CTR (the result is called *the outer message*). The server calculates the HMAC digest on the outer message using *mac-key*. The server sends Alice the encrypted outer message, the outer message's IV, and the outer message's HMAC digest.

Note: I recommend using Base64 encoding to encode the encrypted token, IV, and HMAC digest. Base64 strings can be easily converted to and from **byte**[].

5. Alice checks the outer message's HMAC digest and retrieves the key K_S. She checks that the nonce that she sent N_A appears in the plaintext of the outer message. If all is ok, she sends Bob the token (*bob-key*{K_S,Alice}), its IV, and its HMAC digest.

   Note: As noted above, this message and all subsequent ones in the story are sent and received using a dedicated port that Bob sent Alice in step 2 and using a separate thread.

6. Bob checks the HMAC digest and the token (opening it with *bob-key*). If it came from Alice, Bob challenges Alice to encrypt a random nonce N_B with the session key K_S. The nonce N_B is sent encrypted with K_S using AES-CTR (the result is called *the challenge message*). The challenge message's HMAC is computed using *mac-key*. Bob then send Alice the IV, the challenge message, and the HMAC digest.

7. Alice checks the HMAC digest on the challenge message. If it's ok, Alice opens the challenge message and extracts the nonce N_B. She subtracts 1 from the nonce ($N_B - 1$) encrypts the result with K_S using AES-CTR (the result is called *the response message*). She computes the HMAC digest on the response message and sends Bob the IV, the response message, and the HMAC digest.

8. Bob checks the response message. If it's ok, Bob sends Alice an OK message encrypted with K_S using AES-CTR. Bob computes the HMAC digest using *mac-key*. Bob sends Alice the IV, the encrypted OK message, and the HMAC digest.

9. If Alice gets a valid OK message (HMAC digest and decryption are ok), she begins to send Bob the file encrypted with K_S using AES-CTR. The HMAC digest is computed using *mac-key*.

4 What to do: SCAFTIA Tool

You will modify the SCAFTI tool you prepared for assignment 2. Aside from implementing all elements of the user story above, your SCAFTIA tool must meet the following requirements:

1. The tool must ensure that the server port and IP address are not missing.
2. The tool must ensure that the personal key is not empty or missing.
3. All encrypted messages (no matter which key is used) must be encrypted using AES-CTR.
4. All messages must be protected with an HMAC digest using HMAC-SHA256.
5. All messages must be protected using the HMAC-SHA256 algorithm in an Encrypt-then-MAC scheme. This means first the message (or file) is encrypted using AES-CTR and then the digest is computed over the ciphertext message (or file) and its IV together. The only exception is message 3 above which **does not** need to be encrypted.
6. When sending a file, the sender must first get approval from the receiver to accept the file. After the recipient has approved, the sender must contact the server to get a session key.
7. The sender must get a new session key for each file sent. Therefore, if Alice sends Bob two files, each will be encrypted using a different session key retrieved from the server.
8. If the recipient refuses to accept the file, the sender must not contact the server to ask for a session key.
9. After the sender and receiver have completed the transfer of the file and the file has been successfully verified and decrypted, the recipient must send the sender a file approval message encrypted with the session key (message after message 9 above).

10. After the receiver sends the file approval message, it must forget the session key.
11. After the sender receives the file approval message, it must forget the session key.
12. If the sender receives an invalid response from the server when asking for a session key, the sender tool must show a short error message about the problem and a detailed entry in the log (see below). The sender must only ask for another session key if the **user requests** to try again (user intervention required).
13. If the receiver receives an invalid token or fails to validate the sender, the recipient tool must show a short error message about the problem and a detailed entry in the log (see below). The receiver must close communication on the dedicated port and not respond to or display any more messages related to the offer.
14. After receiving an invalid token or failed validation, the receiver **will** accept other offers from the sender, either for the failed file or for other files.
15. When sending a file, the file must be encrypted completely CTR using a single IV. The HMAC digest must be calculated over the entire file at once. This means you can't break the file into chunks and encrypt-then-MAC the individual chunks.
16. The log file must include the following information about invalid authentication messages or tokens: The date and time of the event, the message's apparent sender (IP address, port, user name), the content of the received message or token, the IV, the HMAC value received, and a textual description of the error.
17. The SCAFTIA tool must enable the user to intentionally send the following types of invalid messages. The message numbers below refer to the steps in Figure 2:
    (a) Invalid Server key request (Msg 3) - With an invalid requestor name
    (b) Invalid Server key request (Msg 3) - With an invalid receiver name
    (c) Invalid token message (Msg 5) - Send the token to the wrong user
    (d) Invalid token message (Msg 5) - Create and send a random token.
    (e) Invalid challenge message (Msg 6) - Send a nonce encrypted with the wrong key
    (f) Invalid response message (Msg 7) - Send a response which is encrypted with the wrong key (not K_S)
    (g) Invalid response message (Msg 7) - Send a response which is encrypted with the wrong numerical response (correct K_S, but some value other than $N_B - 1$ encrypted).
    (h) Invalid OK message (Msg 8) - Send a message other than the proper OK encrypted with the session key K_S
    (i) Invalid OK message (Msg 8) - Send an OK message encrypted with a different key.
    (j) Send the file (Msg 9) encrypted with the wrong key (not K_S)

5 What to do: Server

You will need to create a server in addition to what you wrote for assignment 2. Aside from implementing all elements of the user story above, your server must meet the following requirements:

1. The server must be written in Java
2. The server must have a JavaFX graphical user interface (GUI).

3. The server must be configured with the following information: listening IP, listening port, list of known users and passwords.
4. All of the server's configuration information must be configurable using the GUI
5. The server's configuration information must be stored in a configuration file that is loaded by default when the server is turned on and is save whenever the information is changed in the GUI.
6. The server's user list must have the following fields: user name, password. You may add additional fields as necessary.
7. All encrypted messages (no matter which key is used) must be encrypted using AES-CTR.
8. All messages must be protected with an HMAC digest using HMAC-SHA256.
9. All messages must be protected using the HMAC-SHA256 algorithm in an Encrypt-then-MAC scheme. This means first the message (or token) is encrypted using AES-CTR and then the digest is computed over the ciphertext message (or token) and its IV together.
10. The server must be multithreaded (be able to handle multiple clients at once)
11. It must be possible to start and stop the server listening.
12. If the server is stopped, it must be possible to resume listening again without the need to restart the server.
13. When the server successfully completes a session key request, it must show information on the GUI about the requestor and intended recipient. It must also write a detailed log message (see below).
14. When the server fails to process a session key request, it must show information on the GUI about the requestor and what failed when processing the request. It must also write a detailed log message (see below).
15. The server must choose a new random session key when requested using a secure random number generator (*ex.* SecureRandom in Java).
16. All session keys must be 256 bits.
17. The server must enable the user to intentionally send the following types of invalid messages. The message numbers below refer to the steps in Figure 2:
    (a) Invalid Server responses (Msg 4):
        i. Response with an invalid nonce (not the one the user sent)
        ii. Response with an invalid target name (not the one the user sent)
        iii. Response that is encrypted with the wrong requestor key (using a random key)
        iv. Response encrypted for the wrong recipient (not the one requested)
18. The server's log must have the following information about every message received:
    (a) The date and time
    (b) The sender's IP and port
    (c) The sender's declared name
    (d) The intended recipient's name
    (e) The nonce (N_A) sent
    (f) Whether the request message was encrypted (encrypted/not encrypted)

(g) Whether the message's HMAC was valid (valid/invalid)

(h) A textual description of any error which occurred in processing (*ex.* unknown sender, unknown recipient, etc.)

(i) Whether a response was sent back

(j) Whether the response sent back was intentionally incorrect.

6 On Debugging

You will find that debugging the SCAFTIA tool will be about as difficult as debugging SCAFTI, perhaps even a bit easier since you have already written the communication logic in the previous assignments. To test the ability of recipients to detect bad tokens (the hardest part here), you'll need to use the "intentionally invalid" features mentioned above. Alternatively, you can do packet copying and injection on your home network if you have the ability.

7 What to turn in by 30 June 2019 at 11:55pm

You may work in groups of up to two (2) students. The assignment may be submitted either via Moodle or via GitHub.

7.1 Moodle Submission

If submitting via Moodle, each **member of the group** must turn in a single ZIP file that contains the following items. Students who do not submit a copy of the work will not be given a grade for the assignment, even if their names appear in the submission of another student. The submission must contain the following items:

- Full source code for the submission, including any additional libraries necessary for the code to be compiled.
- An executable JAR of the SCAFTIA program runnable on any Windows 10 computer, along with any required configuration files. The JAR must be runnable by just clicking on it.
- A README file (*e.g.* README.txt, README.pdf) with the following sections:
  1. A list of the students in the group, including names and ID numbers
  2. A list of how many hours were spent on each part of the assignment
  3. Instructions for compiling the source code for SCAFTIA
  4. Instructions for running the SCAFTIA programs
  5. Documentation of the configuration files for the SCAFTIA client and server
  6. Documentation of how the SCAFTIA tool implements the Needham-Schroeder protocol and uses session keys. If you modified the communication protocol from your assignment 2 submission, you must fully document it here as you did for assignment 2.
  7. Documentation of how the server works, stores user information, and generates session keys.
  8. Documentation of the location and format of the log files for SCAFTIA and the server.

7.2 GitHub Submission

If submitting via GitHub, you must use the private repository opened by the instructor for the assignment. The repository must contain the following items in the master branch:

- A final commit with the words "Submitted for grading" in the commit comment.
- Full source code for the submission, including any additional libraries necessary for the code to be compiled.
- An executable JAR of the SCAFTIA program runnable on any Windows 10 computer, along with any required configuration files. The JAR must be runnable by just clicking on it.
- A README file (*e.g.* README.md, README.pdf) with the following sections:
  1. A list of the students in the group, including names and ID numbers
  2. A list of how many hours were spent on each part of the assignment
  3. Instructions for compiling the source code for SCAFTIA
  4. Instructions for running the SCAFTIA programs
  5. Documentation of the configuration files for the SCAFTIA client and server
  6. Documentation of how the SCAFTIA tool implements the Needham-Schroeder protocol and uses session keys. If you modified the communication protocol from your assignment 2 submission, you must fully document it here as you did for assignment 2.
  7. Documentation of how the server works, stores user information, and generates session keys.
  8. Documentation of the location and format of the log files for SCAFTIA and the server.

8 Grading

The weights for the grading will be assigned as follows:

- Server operations: 30%
- Client operations (session key request, token processing): 35%
- Robustness: 10%
- Log file correctness and completeness: 10%
- Readme documentation 15%

8.1 Penalties

The following penalties will be assessed for the errors or omissions below:

- Hard coding file names, encryption keys, MAC keys, IVs, IP addresses, or port numbers into the code. **(-15) points**
- Not handling quit correctly. **(-15) points**
- Client crashes when the authentication server isn't up. **(-3) points**
- No support for invalid or incorrect messages as the testing regimen specifies.**(-20) points**
- Not handling bad tokens or requests correctly. **(-15) points**

- No README file. **(-15) points**
- README file missing key element **(-5) points per key element**
- Unhandled exception. **(-3) points per exception**
- Need to restart the server to restart listening. **(-5) points**
- File transfer doesn't send files with the correct size or contents. **(-25) points**
- Only sends text files, not binary ones. **(-10) points**
- Session keys are not random. **(-15) points**
- Server doesn't respond to requests from clients. **(-25) points**
- Server doesn't behave properly when given an incorrect message. **(-10) points**
- Doesn't send the file on a different port - uses the chat one. **(-15) points**
- Sending succeeds even when the private password with the server is wrong. **(-15) points**
- Log is missing important element. **(-3) points per element**
- Shows gibberish messages if a message doesn't decrypt correctly (wrong key). **(-5) points**
- No user names, just IP addresses. **(-5) points**