Networks and Systems Security - Winter 2019

Sambuddho Chakravarty

October 24, 2023

1 Assignment 2 (total points:105)

Deadline: Nov. 11. 2023 @ 2359 hrs.

1.1 Secure file copy (subtotal points:45)

This exercise requires you to write a secure file copy program, which works similar to `scp`. The program must use `netcat` as a client and a server, where the payload is fed to (or from) a separate program that encrypts (or decrypts) the data being transmitted. The program emulates the `scp` program, *i.e.* the program to copy files encrypted and encapsulated inside a `ssh` connection/tunnel.

The parent program must fork two child processes or threads. In case of the client program, one of these child process/threads should read the contents of a file and encrypt it and send it (using any known IPC mechanism) to the other sibling process/thread. The other thread's job is to encapsulate the encrypted data into TCP packet payload and send it to the peer (*i.e.* the server).

Correspondingly, the server program must also create two child processes or threads. One of these listens on a particular IP/Port number (to which the client connects). This is also to be implemented using the `netcat` program. This thread/process reads the data sent by the client and sends it to the other process/thread (of the server), using some IPC mechanism. This other process/thread decrypts it and saves it in a local file.

The encrypted ciphertext that the client shares with the server, must also sent together with an IV/nonce and a HMAC signature, derived from the key used to encrypt the data. The server must not only decrypt the data, but also validate the HMAC, *before* saving it to a local file.

Important assumptions:

1. The two programs use a pre-shared key to encrypt (or decrypt) the data before (or after) sending (or receiving) it. One way to derive the key is by reading the requisite bytes from the file `/dev/urandom`.
2. Both programs could run on the same machine, whereby both use the local host prefixes and subprefixes (127.0.0.0/8) and ephemeral ports. Alternatively, the could be on separate VMs as well.
3. The file encryption/decryption and generation/validation of the HMAC signatures MUST involve the use of `openssl libcrypto` library. You could use the envelope functions (starting names like `EVP_`) for doing the same.
4. Every file is accompanied with a single HMAC, which is sent along with the packets bearing the encrypted bytes corresponding to the file being transmitted. This must be validated before the decrypted data is stored in a local file.

5. The ciphertext is also accompanied with the IV/nonce used to encrypt the data. Both the IV/nonce and the key can be derived from the reading bytes from the pseudo-device file `/dev/urandom`.

The following figure can help you get a better picture of how this is to be done.

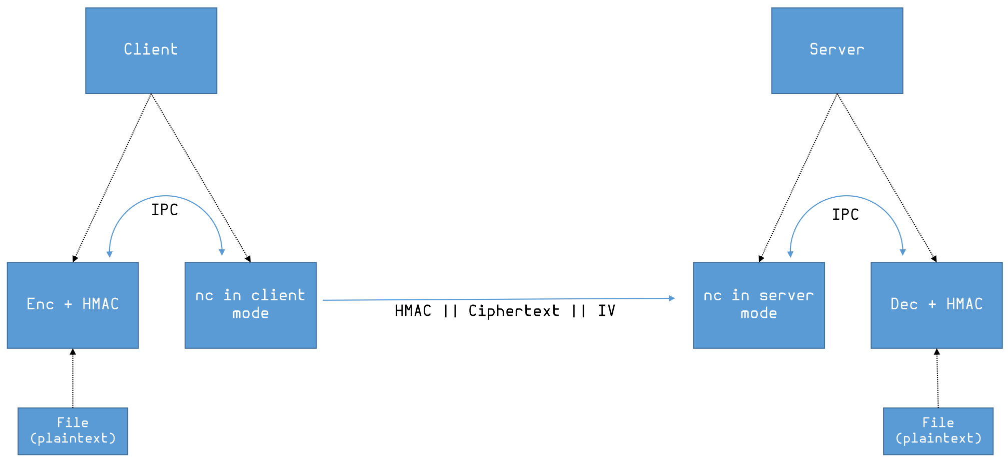


Figure 1: The secure file copy client and server programs.

Some useful links with examples on how to use envelope functions:
`https://www.openssl.org/docs/man1.1.1/man7/evp.html`
`https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption`
`https://home.uncg.edu/cmp/faculty/srtate/580.s13/digest_ex.php`

What you submit/Rubric:

1. Correctly compiled programs (both client and server) (via a Makefile) – 10 points.
2. Correct functioning of all the commands of the program, designed using `sockets` API, `libcryto` – 20 points.
3. HMAC correct validation: To test HMAC validation, you could you `netfilter` kernel module wherein you flip certain bits of the encrypted payload of packets (generated using `netcat`). The HMAC validation should fail in such cases. The file MUST not be written to the server's disk/filesystem – 10 points.
4. Documentation describing the system design and the assumptions made – 5 points.

1.2 Multi-user IRC chat client and server (subtotal points:60)

You would build upon whatever you designed and implemented in the previous assignments. You are required design a multi-threaded server to which the users now connect remotely (the same (real) users which you considered previously).

The users are to be authenticated via a Needham Schrodher (NS) authentication scheme. The server should listen on two ports – one for the KDC function and the other for chat server operations. Before you launch the server you need to create long-term symmetric keys for every user. One way to do this is by using the Password Based Key Derivation Function (PBKDF) built into `openssl (libcrypto)` library. These long term keys are used during the NS authentication scheme to derived a symmetric key. In our system, the user communicates to the KDC to derive a shared secret with the server.

A user begins by connecting (via `sockets` API) to the KDC server port. Thereafter, the *ticket* is presented to the chat server port (different from the KDC port) which finally authenticates the user.

Once authenticated the server presents a IRC like interface to the user where the user can see all other users. He/she can also view his/her own files, as well as that of others. He/she can communicate with all users by broadcasting.

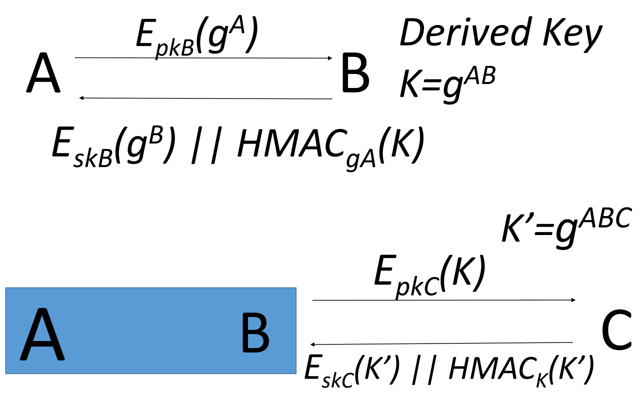
Further, the users can privately communicate with a few chosen set of users. For this, a user first creates a group and sends an invitation to (his/her) chosen users to join the group. When these users receive it they may reply back with their public keys and upon receiving a multi-party Diffie Hellman Key Exchange would ensue.

You must support the following commands:

- "/who": Who all are logged in to the chat server, along with a user IDs.
- "/write_all": Write message which gets broadcasted to all users.
- "/create_group": Create a group to which users may be added. A group ID and name is returned.
- "/group_invite": Send an invite to individual users IDs.
- "/group_invite_accept": Accept the group invite.
- "/request_public_key": Send request for public key to a specific users.
- "/send_public_key": Send back public key back as a response to the above request.
- "/init_group_dhxchg": This process initiates a DH exchange first with any two users and then adds more users to the set. The following steps describes this in more detail:

    1. User U_1 initiates a DH exchange with another user, say U_2. To do so U_1 sends his DH exponent g^{U_1} to user U_2, encrypted with the public key of user U_2.
    2. User U_2 responds with his own value of DH exponent g^{U_2} encrypted with his/her own private key and concatenated with a HMAC signature of the established shared secret $g^{U_1 U_2}$, created using g^{U_1}. Let the shared key be denoted by K.
    3. Next, when user U_3 is to be added to the group of existing users (*e.g.* the user of users U_1, U_2, the group initiator U_1, sends the existing shared secret of the group K to user U_3, encrypted with the public key of U_3.
    4. The user U_3 responds with his own computed DH exponent $g^{U_3 U_2 U_3}$ encrypted with his own private key and concatenated with the HMAC signature of the now established secret $g^{K U_3}$, created using K. The updated shared key may be denoted as $K^{'}$.

    Once a group is created and the DH exchange is done, the key $K^{'}$ (see above) is the shared key of all users of the group. Thereafter, messages are encrypted and decrypted using the key and broadcasted to all users of the group. Other users, who are not a part of the group cannot see these messages. The key establishment is shown in figure 2.



A
$E_{pkB}(g^A)$
B
Derived Key
$K=g^{AB}$
$E_{skB}(g^B) \mid\mid HMAC_{gA}(K)$
$K'=g^{ABC}$
$E_{pkC}(K)$
A
B
C
$E_{skC}(K') \mid\mid HMAC_K(K')$

Figure 2: Adding multiple users to a group and deriving a shared secret.

- "/write_group": Write messages to a group specifying its group ID.
- "/list_user_files": Show files (only with appropriate read permissions) of any user.
- "/request_file": Request a file and also specify the port number to connect to and send. Once a request_file command is sent, the sender waits for the file by listening on a particular IP address port number. The receiver thereafter connects back to the IP:port pair and sends the file. This connection between the users MUST be encrypted using TLS handshake (`openssl libcrypto` library) where the requester (acting as server) uses a self-signed certificate.

2 How you would be graded:

1. Correctly compiled programs (both client and server) (via a Makefile) – 5 points.
2. Correct functioning of all the commands of the program, designed using `sockets` API, `libcryto` and `libssl` – 35 points.
3. Description of any three vulnerabilities that the system is designed to be resilient against and demonstrating this (via scripts/test cases) – 15 points.
4. Documentation describing the system design and the assumptions made – 5 points.