CSC4012Z - Network and Internetwork Security Practical

Shakeel Mohamed - MHMSHA056
Tony Guo - GXXHEN001
Ryan Scott - SCTRYA002
Moegamat Ra-eez Stenekamp - STNMOE001

Implementation

Implemented this practical in Java.

Assumptions

The following were assumptions taken from the practical outline:

- Public keys are “available” to the Client and Server, so no exchange of certificates is required to obtain the public keys.
- Some kind of local ‘key ring’ can be assumed.

Decisions

From the practical outline, we decided to create a client and server application that would run on separate computers and communicate via a communication system based on TCP. The server application would run on one computer, and the client application would be able to run on one or more computers.

The following implementation decisions were chosen based on the assumptions.

- Since a local key ring can be assumed, we decided to locally save the private keys in a textfile on the computer for which the server or client application is running on. This textfile would act as the local key ring on either the server or client. Computers running the client application would find this textfile to be named EncrpytionPrivate.txt, and computers running the server application would find this textfile to be named ServerPrivate.txt.
- Another textfile named GlobalPublic.txt was created to act as a place where public keys of both client and server were available. Since it was assumed that public keys can be available with no exchanges between entities to obtain them, we copy and pasted the public key with its key id generated from the generation of an RSA key pair for either a client or server to this textfile. Any new public keys would have to be copy and pasted to this textfile and be updated on every copy of this file on the machines running either the client or server application. This will ensure that public keys are available to the client and server.

Applications

The 3 Java applications were created: RSAKeyGenerator, TcpClient and TcpServer. The machines running the client application would have the RSAKeyGenerator and TcpClient applications, and a machine running the server application would have the RSAKeyGenerator and TcpServer applications.

RSAKeyGenerator

Generates an RSA Key Pair that consists of a public and private key with a key size of 2048-bit.

Runs with 2 inputs – key id of private and public key generated, and the prefix of the textfile to save the keys in e.g. given prefix input Encryption, the private key and public key will be saved in EncryptionPrivate.txt and EncryptionPublic.txt respectively along with their key id associated with it.

TcpClient
Run with 2 arguments – IP address and port number of which the server is listening on.

Loads its own client's private key from EncryptionPrivate.txt.

Loads all public keys from GlobalPublic.txt into a hashmap with the public keys' key id as the hashmap's key and the public keys' encoded string form as the hashmap's value.

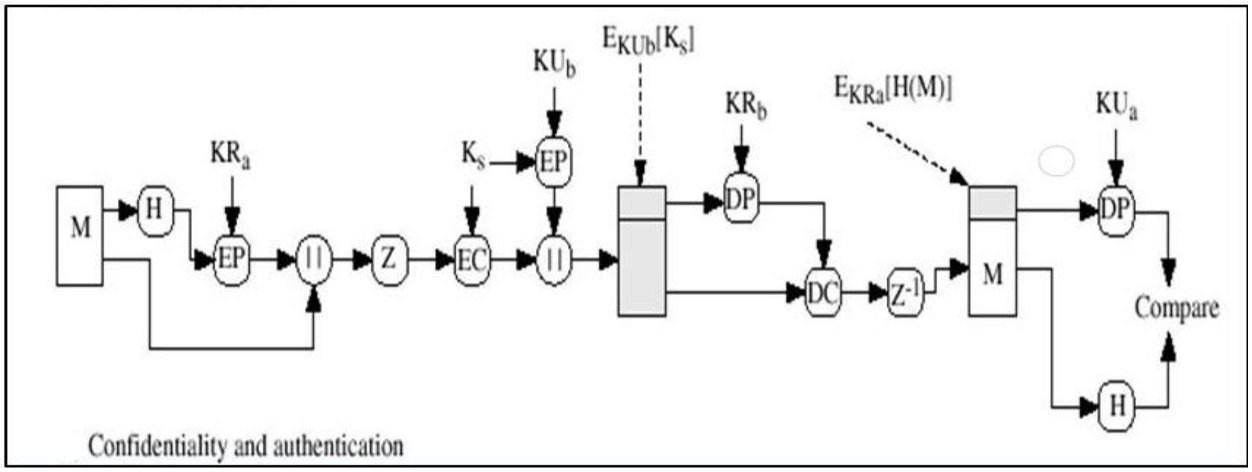
Tries to connect to server with given arguments.

Once the connection is established between the TCP server and TCP client, the client can type messages to send to the server. Once the message is typed and the client hits the "Enter" button, the message is secured by the following process for transmission:

- A hash of the message is generated, we did this with the Message Digest object in Java. The algorithm used to generate the hash is "SHA-256".
- This hash gets signed by the private RSA key of the client. We create a Signature object in Java with the algorithm "SHA256withRSA" to sign this hash. This adds authenticity.
- The signed hash encoded using radix-64, the original message and the key id of the client for it's public key gets packaged into a string. This byte representation of this is taken and compressed by the ZIP algorithm.
- A one-time shared key is generated with the AES algorithm.
- The compressed package is encrypted with this shared key using the AES algorithm in CBC mode with PKCS5 padding. This adds confidentiality.
- The public key of the server is retrieved from the hashmap.
- The generated shared key gets encrypted with the server's public key using the RSA algorithm in ECB mode with PKCS1 padding.
- The encrypted shared key, encrypted compressed package and the initialization vector for the shared key is encoded to string using the radix-64 conversion. This is then packaged together as one string to be transmitted to the server.

Once the packaged message is ready for transmission, the message gets transmitted to the server.

The user is now able to type another message to send to the server or type "!!exit" to drop the connection and close the client.



TcpServer
Loads its own server's private key from ServerPrivate.txt.

Loads all public keys from GlobalPublic.txt into a hashmap with the public keys' key id as the hashmap's key and the public keys' encoded string form as the hashmap's value.

Creates a TCP server on its own IP address and port number 12061.

After creation, it listens for any connections from clients.

When a client tries to establish a connection, a thread gets spawned to accept the connection and to handle the communication with the client. This way multiple connections with clients can be handled.

The spawned thread listens for any input from the client. Once it receives the secured message, it does the following to decrypt and check the message:

- The message gets broken down into the encoded forms of the encrypted shared key, the initialization vector for the generation of the shared key and the encrypted compressed package.
- The encoded forms get decoded into its binary form – bytes with a radix-64 decoder.
- The encrypted shared key gets decrypted with the server's private key using RSA algorithm in ECB mode with PKCS1 padding.
- A shared key is created using the binary form of the shared key and its initialization vector.
- This shared key is used to decrypt the encrypted compress package.
- The decrypted compressed package gets decompressed using the ZIP algorithm.
- This decompressed message is converted into string and broken down into the key id of the client, the original message and the signed hash.
- From the original message, an unsigned hash is generated to compare with the signed hash.
- A Signature object is created with the client's public key found with the key id of the client. The original hash is gotten by decrypting the signed hash using the public key of the client.
- This Signature object is used to verify whether the generated unsigned hash is the same as the original hash.
- The verification and the message get outputted to console to show what the server decrypted and checked.

This whole process is outputted to console to show what is done on the server side.

Choice of Language
We chose Java as our development language due to it's useful arrangement of security and networking libraries in the SDK, such as:

- **java.security** - recreating keys, hash of messages, and creating signatures to sign with.
- **java.util.zip** - compressing and decompressing messages in byte form.
- **java.crypto** - generating keys, encrypting, decrypting and signing with keys.
- **java.net** - creating a communication system with a TCP client and TCP server.

All of these security and networking libraries were used. In addition, we all have considerable programming experience in java, and are well adjusted to using it's syntax.

Choice of Crypto Algorithms

Generation of Keys

Public/private key pairs were created for use with RSA and a key size of 2048-bit. 1024-bit RSA keys became crackable some time between 2006 and 2010. 2048-bit keys are recommended for the era now as they are sufficient and unlikely to be crackable until 2013. The National Institute of Standard and Technology recommends 2048-bit keys for RSA. RSA was chosen to create key pairs due to the practical outline stating so.

Shared keys were generated using Advanced Encryption Standard (AES) algorithm. We chose to use AES over Data Encryption Standard (DES) since it supersedes. It is adopted by the US government and used worldwide. We chose to use a key length of 256 bits since it is the most secure in terms of being cracked. It is even difficult for quantum computers to crack.

Generating Hash

To generate a hash of a message, the “SHA-256” algorithm was used. This cryptographic hash function is part of the SHA-2 set which is a US federal standard. SHA stands for Secure Hash Algorithm. We chose this algorithm for its security and a hash size of 256-bit due to its wide adoption and secureness.

Signing Hash

The hash was signed with a Signature instance in Java with the “SHA256withRSA” algorithm since the hash was generated using the SHA256 algorithm and the private key used to sign the hash was generated using RSA.

Encoding and Decoding Strings

Our objective was to simulate aspects of PGP and so the binary bytes were encoded and decode using radix-64 conversion. This was equivalent using the Base64 class in Java which we implemented.

Encryption and Decryption

For asymmetric encryption with RSA public/private keys, we created a Cipher instance and used the RSA algorithm in ECB mode with PKCS1 padding since it was encouraged in the practical outline and seems to be the standard to generate unique cyphertext.

For symmetric encryption with AES shared key, we created a Cipher instance and used the AES algorithm in CBC mode with PKCS5 padding. This was encouraged and seems to be the standard for AES algorithms.

Key Management

RSA key pairs were generated using the RSAKeyGenerator Java application. Since a local key ring could be assumed, the private keys of the client or server were saved in a local textfile. Since we could assume the public keys to be available, all the public keys were stored in a global textfile.

When a client or server application started, it would look for its own RSA key in its local key ring file which is a local textfile in this case. This gets loaded and a private key instance is created for either the client or server. This assumes that all clients already have its own private key, and that this private key is the primary use for signing hashes for this practical. All public keys of clients and server also gets loaded in so that the message gets sent to the correct recipient and that encrypted messages can be decrypted. In this case it was assumed that the message only ever reaches the server and ends there. It is also assumed that all public keys of clients and server is in the global textfile. All public keys gets loaded into a hashmap for fast access and retrieval.

If a new client generates their own RSA keypair, their public key generated will need to be copied and pasted into the global textfile and updating all copies of the global textfile on different client machines.

Communication Connectivity Model

A TCP server runs on its own IP address and port number 12061 and listens for a TCP client to make a connection on its IP address and port number. Once a TCP client runs and connects with a Socket instance using the server's IP address and port number, a connection gets established and the client is now able to transmit a message to the server. When the server accepts the connection, a thread is spawned to handle any inputs from the clients that it will receive. This way a communication system is setup.

Format of the messages sent by the client to the server consists of the following (no next lines, presented this way for easy comprehension):

|KEY|<encoded shared key(radix64)>
|IV|<encoded initialization vector(radix64)>
|MSG|<encoded encrypted compressed message package(radix64)>

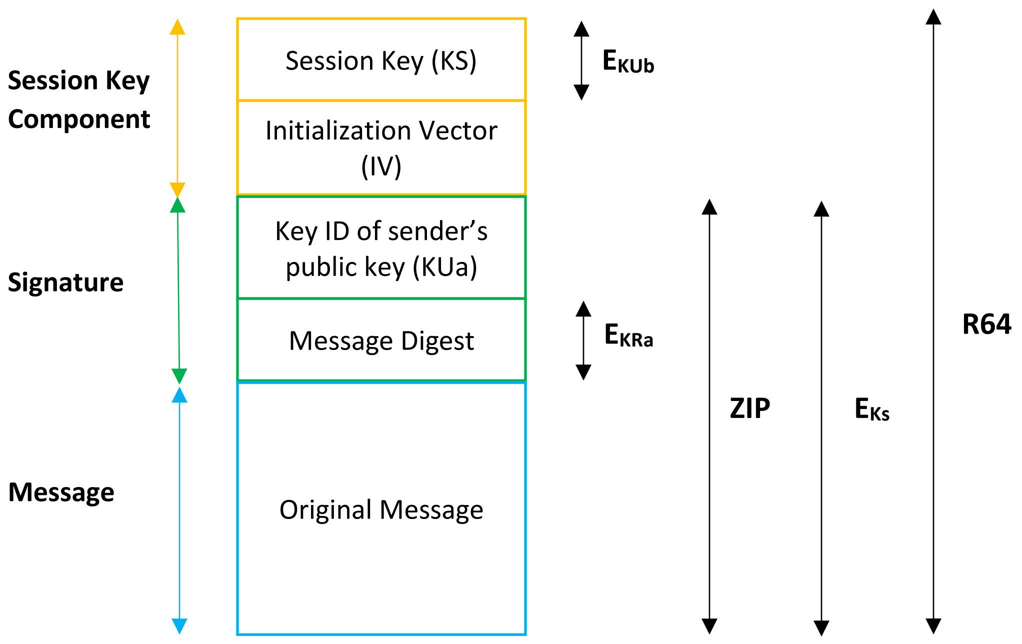
→ Decoded and Decrypted

<compressed message package>

→ Decompressed

<message package> consists of the following (no next lines, presented this way for easy comprehension):

|KEYID|<key id of client>
|SIGNATURE|<encoded signed hash(radix64)>
|MESSAGE|<original message>


Session Key Component
Signature
Message
Session Key (KS)
Initialization Vector (IV)
Key ID of sender's public key (KUa)
Message Digest
Original Message
E_{KUb}
E_{KRa}
ZIP
E_{Ks}
R64