Network and Internetwork Security Practical

CSC4012Z 2019

By:

Duncan Campbell (CMPDUN001), Claire Denny (DNNCLA004), Jarryd Dunn (DNNJAR001), Matthew Poulter (PLTMAT001)

Development

We used a collaborative Test-Driven Development approach to developing the application for this assignment. The collaboration was made possible by a well-used GitHub repository (including for Issue Tracking) which you can access using the link below:

https://github.com/mdpoulter/nis-prac

Implementation explanation

Pretty Good Privacy (PGP) provides both confidentiality and authentication. We have thus used Figure 5.1 from slide 103 of the slide set for this course as the basis of our implementation. Classes *Client* and *Server* together comprise the communications implementation. Classes *AES*, *GZIP*, *Hashing*, and *RSA* provide the methods for the security implementation.

Communications implementation

The communication between client and server was created using a simple Java socket interface. The server is started and listens for a connection on port 12345. The client can then be started and the server address entered to initiate a connection. In its current form, this only supports one-to-one connections.

Once connected, the client may send messages to the server including the “exit” command which will kill both the client and the server. All debug messages can be seen on both the server and client to view the full security implementation.

Security implementation

To create new private and public keys

The private and public keys for both the server and the client are stored in a “keys” folder. These are generated by running the *main* method of the *RSA* class or by running “make keys”.

Client

The client, Alice, wishes to send a message to another client, Bob. The plaintext undergoes two separate processes: one yields a signature and the other yields the encrypted message. The signature, encrypted message and an encrypted private session key are transmitted via the server to Bob.

To produce the signature

The method *hash* in class *hashing* accepts as input the plaintext message the client wishes to send and returns a 128-bit hex string known as a message digest. The digest serves as both something by which data integrity can be checked and as something that can be used to provide authentication. Alice’s private key is obtained by method *getKeyFromFile* in class *RSA* and used to sign the message digest using the method encrypt in class RSA.

To produce the encrypted message

The method *compress* in class *GZIP* compresses the plaintext message using GZIP compression, reducing the patterns in the plaintext which cryptanalysis would seek to exploit so as to crack the cipher. The compressed message is then encrypted with the private session key (the SecureRandom library is used to get 16 random bytes to generate the key) using the advanced encryption standard (AES) encryption algorithm with cipher block chaining (CBC) in class SymmetricEncryption. These algorithms are implemented using BouncyCastle as a provide for the Java Cryptography Extension.

To produce the encrypted session key

The private session key is encrypted with the public key of Bob (obtained by method *getKeyFromFile* in class *RSA*) by method *encrypt* in class *RSA*.

Server

Bob receives Alice's transmission. Bob decrypts the encrypted private session key and uses it to recover the plaintext message. The signed message digest verifies who sent the message and can be decrypted such that the message digest can be used to check that the signature applies to the message received.

To recover the private session key

The client, Bob, decrypts the received encrypted private session key using his private key (obtained by *getKeyFromFile* in class *RSA*) by method *decrypt* in class *RSA*.

To recover the plaintext message

The key of the *AES* object is set to the recovered private session key using its *setKey* method. Next the *decrypt* method in class *AES* decrypts the encrypted message using the recovered private session key.

To use the signature

Method *decompress* in class *GZIP* decompresses the message contents before it is authenticated by method *decrypt* of Class *RSA* which decrypts the message digest using the public key of Alice (obtained by method *getKeyFromFile* in class *RSA*). Method *hash* in class *Hashing* verifies the integrity of the data in the message by comparing the hash of the recovered plaintext against the hash sent by the client of the original plaintext.

Choice of language

We chose to use Java for our implementation. Our choice was primarily influenced by the following five factors:

1. Java was recommended in the assignment brief;
2. encryption libraries, such as Bouncy Castle, are offered for Java;
3. security libraries are included in the SDK, with documentation available;
4. all group members have experience with coding in it;
5. using Java allows us to use JUnit, a framework that facilitates our test-driven development approach.

Choice of crypto algorithms

Hashing

SHA-512 is used to hash the plaintext message. This was chosen because it tends to run faster on 64-bit processors than SHA-256, and the assumption was made that a communications application would value speed.

Compression

GZIP was used as the compression library for several reasons. Firstly, Java provides a simple library to use for GZIP compression. Secondly, GZIP produces small compressions in shorter times to alternative compression libraries. It also decompresses text faster. Finally, GZIP in Java allows for compression of strings, unlike ZIP which works better for files.

Symmetric Key Encryption

The Advanced Encryption Standard (AES) algorithm with Cipher Block Chaining (CBC) is used for symmetric encryption. The AES algorithm is used over its predecessors Data Encryption Standard (DES) and Triple-DES since it is harder to decrypt by force and is not vulnerable to meet in the middle attacks; although it is more computationally intensive. AES encrypts data in 128 bit blocks, CBC take these blocks and XORs each block with its predecessor before they are encrypted. Using CBC means that repeating patterns in the plain text will not result in repeating patterns in the ciphertext.

Asymmetric Key Encryption

The RSA (Rivest–Shamir–Adleman) algorithm with Electronic Code Block (ECB) was used for asymmetric encryption. RSA is the most widely used public key encryption because it has been successful for many years. ECB mode encrypts each plaintext block independently and allows easy parallelization to yield higher performance. Public key encryption is computationally intensive when compared with symmetric encryption such as AES. Therefore, public-key encryption is often used with other encryption techniques to ensure a senders/receivers correct identity and the security of other encryption keys.

Key management

A new private session key is generated for each method. This key is stored in the *AES* object and can be obtained using the *getKey* method.

The private and public keys for both the server and the client are stored in a “keys” folder. These are generated by running the *main* method of the *RSA* class or by running “make keys”. The private key of the server is not explicitly available to the client and vice versa, although the files are obviously stored in the same folder.

Communication connectivity model

The communication connectivity model is a one-way one-to-one client-server model. The server waits for a single client socket connection and then accepts messages from the client sent in the form of a Java Object Stream.

Running the application

All the make commands from the bash terminal:

```
make              # Compile the application
make keys         # Generate new public and private keys
make test         # Run the tests
make server       # Run the server
make client       # Run the client
make clean        # Clean the folder
make docs         # Make the javadocs
```

To get the application running, use:

```
make
make server
make client       # In a separate terminal
```

When running the client, enter the server address as “localhost” when running on the same system as the server.

To exit the client and server, type *exit* in the client.

Note: Occasionally, you will need to run *make* twice if an error occurs.