CO3099/7099 Programming Assignment

Released Feb 12, 2018 Deadline Mar 2, 2018 11:59 pm

Instructions to Students

- This assignment consists of four tasks. Each of Tasks 1–3 build upon earlier ones, so (for example) if you submit Task 3 you do not need to separately submit Tasks 1 and 2. Task 4 is a separate task. Students on CO7099 please see the footnote at the end.
- The completed work should be submitted by the Electronic Handin System following the instructions on the module web page. Your handin will consist of up to three files `Client.java`, `Server.java` and `Crack.java`.
- Programs will be marked by *execution testing* and *code inspection*. You should make sure your programs are compilable and executable on the departmental linux system. If your programs cannot be compiled or run, for whatever reason, you will lose a substantial portion of the marks. Your programs should also be readable. The 5% "missing" mark is for readability of your programs.
- This document d ise for further clarifica

You are part of an underground resistance organisation ag f your new recruits, Winston, have come up with a scheme for secur nt it by completing the following tasks.

Userids and keys. Each member of this organisation has a userid, which is a simple string like `alice`, `bob` etc. There is also a user with userid `server` that will be needed in Tasks 2 and 3. Each user has a pair of public and private keys of the form `userid.pub` and `userid.prv`, where `userid` is their userid. Those are 2048-bit RSA keys, generated by the program `RSAKeyGen.java` that you can find on the module webpage. (More details are in the comments of that program.)

For all the programs to be written, it is assumed that they already have access to the appropriate keys (and only those keys) prior to the execution of the programs, and that the keys (in the form of `.pub` and `.prv` files) are in the current working directories of the respective programs.

Task 1: Produce the ciphertext file (25%)

Commands to members of the group are distributed in an encrypted file. A sample of such a file can be found in `ciphertext.txt` on the module webpage. Each line in the file contains an "encoded userid", followed by a space character, and then an encrypted message intended for that user. In more detail:

- The membership of this organisation is top secret, and even members of the group do not know the identities of most other members. (This way you can never betray more than a few people when you eventually, and inevitably, get arrested.) To protect the identites of the members, userids are encoded by computing the MD5 digest of the userid, converting it to hexadecimal, and take the first 8 hex characters. For example, the encoded userid of `winston` is `a699e397`.
- Each message is to be encrypted with `RSA/ECB/PKCS1Padding` with the relevant RSA key of the recipient, then encoded into a Base64 string.

Write a program `Server.java` that produces such an encrypted file. The program reads two files: `userid.txt` which contains one (unencoded) userid per line, and `plaintext.txt` containing a message in each line. The i-th line in `plaintext.txt` is the message for the i-th user in `userid.txt`. If there are more lines in the plaintext file than the userid file, or vice versa, just ignore those extra lines. The program writes the output to a file named `ciphertext.txt`. All these txt files are in the same working directory as the Java file.

Task 2: Full client-server communication (25%)

After a while, it was decided that instead of publishing the whole encrypted file, the users will communicate with a client-server system. Write client and server programs as described below.

General client-server architecture. The programs must be called `Client.java` and `Server.java` respectively, and must

```
java Client host port us
java Server port
```


Each member of the resistance group uses the client program
by someone with a special userid, server.

Server side encryption. On startup (before accepting any client connections) the server reads the `userid.txt` and `plaintext.txt` files and produces a `ciphertext.txt` file, as in Task 1. Then it listens for incoming connections at the port specified in the command line argument.

When a client is connected, the server handles the request, then waits for the next request (i.e., the server never terminates). For simplicity, you can assume that only one client will connect to the server at any one time.

Upon accepting a new client connection, the server receives an encoded userid, finds the encrypted message for that user, and sends that encrypted message to the client.

Client side decryption. The client program connects to the server at the host and port specified in the command line arguments. It then sends its encoded userid to the server. Then it receives an encrypted message, decrypts it, and displays it on screen.

Task 3: Authentication (20%)

Add server authentication to the programs in Task 2. Specifically:

- Use the `SHA1withRSA` signature algorithm, with the same set of public and private keys of the users. (RSA keys can also be used for signatures.)

- Upon connection of a new user, the server first generates the signature (using the appropriate key) of a message consisting of the string `GOLDSTEIN` and a timestamp. The timestamp and the signature (but not the `GOLDSTEIN` string!) are sent to the client. Then it proceeds as in Task 2.
- The client (after sending its encoded userid) receives and verifies the signature. If it does not verify, it disconnects immediately. Otherwise it proceeds as in Task 2, displaying both the received timestamp and the message.

Task 4: Breaking the key (25%)

NOTE: This task is "optional" for CO7099 students (see the footnote[1]).

Unfortunately for Winston, this whole resistance does not really exist, and you are in fact part of a "false flag" operation run by the Ministry of Love. The RSA public key of `winston` (which you can find in the module webpage in the form of a `.pub` file) was actually deliberately made weak, in that one of the two primes p and q forming the modulus n is small, and thus n is easily factorisable. Write a program `Crack.java` that performs the attack. It should

- Extract the modulus n and the exponent e from the public key;
- Factorise n into p and q by going through the factors one by one;
- Compute a matching private key exponent d;
- Display the resul

[1]For students taking CO7099: to account for the credit difference between CO3099 and CO7099, Task 4 of this assignment is "optional" in the following sense: the higher of the two marks, (Tasks 1+2+3)/75 or (Tasks 1+2+3+4)/100, will be used as your mark.