Individual Project Assignment

CE/CZ 4064, CPE413, CSC409: Cryptography & Network Security

Note:

1. The accompanying files for this project have been created by Mr. Ertem Esiner (ERTEM001@e.ntu.edu.sg), and he will also be grading the project. For any clarifications of the provided files, or on the deliverables, etc. contact him directly. Feel free to put the professors in the loop (cc).

2. Students taking CPE413 and CSC 409 need to carry out an extra task. If you are such a student, please get in touch with the course coordinating professor Anwitaman Datta (Anwitaman@ntu.edu.sg) by 1st March 2016 to discuss this additional project.

Submission:

Submission via NTULearn.

Deadline: 11pm, 9th April 2016

This project is to be done and submitted individually. Delay in submission by calendar day will entail 20% penalty per day. So, for example, if a student submits the solutions on anytime on 11th April, then s/he will receive only 40% of whatever would be the score s/he would have received for the technical merit, had it been submitted within the deadline.

Objective:

The objective of this assignment is for the students to learn the applications of cryptographic primitives in network protocols, in particular, to learn how to analyze the security of network protocols and see how to exploit the vulnerabilities of the protocols by implementing an attack.

Problem Description:

Consider a scenario where the users use a proxy client (the proxy) to communicate with a central server (the server). All users send their messages through the proxy. The communication between the users and the proxy is beyond the purview of this project. The proxy and the server share a secret symmetric key, which is initially known only to them and no one else. They use this secret key to generate a stream cipher and encrypt the messages with it.

The Protocol

The communication between the proxy and the server is encrypted using Rivest Cipher 4 (RC4, a.k.a. ARC4 or ARCFOUR). We assume the following setup:

- The proxy and the server share a secret key sk known only to them and on one else. This key is by both the proxy and the server to generate a stream cipher S that is used to encrypt their messages. We write S_i to denote the i-th byte in the stream S.
- The proxy maintains two counters, cp_e (used for encryption) and cp_d (used for decryption), and similarly, the server maintains two counters cs_e (used for encryption) and cs_d (used for decryption).
- At every step the proxy and the server exchange an encrypted message of size 128 bytes. Given a key sk and its corresponding RC4 stream cipher S, a counter c, and a plaintext M of size 128 bytes, the encryption of M, denoted by $Encrypt(sk, c, M)$ below, is done as follows:
  Let $M = M_1 M_2 \ldots M_{128}$ where each M_j is a byte, for $j = 1, .., 128$. Then

  $$Encrypt(sk, c, M) = (S_c \oplus M_1) \,||\, (S_{c+1} \oplus M_2) \,||\, \ldots \,||\, (S_{c+127} \oplus M_{128})$$

  where $\oplus$ denotes the XOR operator and || denotes the concatenation operator for bit strings.
- The decryption function is defined in the same way, i.e., by XOR-ing the stream cipher with the encrypted message. That is, given a key sk, a counter c and a cipher text C of size 128 bytes, the decryption of C is defined as follows: Let $C = C_1 C_2 \ldots C_{128}$. Then

  $$Decrypt(sk, c, C) = (S_c \oplus C_1) \,||\, (S_{c+1} \oplus C_2) \,||\, \ldots \,||\, (S_{c+127} \oplus C_{128}).$$

Initial setup:

- The proxy initiates cp_e and cp_d to 0.
- The server initiates cs_e and cs_d to 0.
- Both the proxy and the server generate a stream cipher S using RC4 with key $\boldsymbol{sk}$. We denote with S_i the i-th byte of the stream S.

Protocol steps:

After the initial setup, the protocol consists of iterations of the following two steps:

Step 1. Proxy → Server : $Encrypt(sk, cp_e, M)$

Step 2. Server → Proxy : $Encrypt(sk, cs_e, R)$

The protocol actions at each step are as follow:

- In Step 1, M is the plain text to be encrypted. The proxy sends the encrypted message $C_1 = Encrypt(sk, cp_e, M)$ to the server, and updates the cp_e counter: $cp_e = cp_e + 128$. The server, upon receiving the message, decrypts C_1 to obtain $M = Decrypt(sk, cs_d, C_1)$ and computes a response R depending on the content of M (see below for the message-response pairs of the protocol). The server then updates the cs_d counter as follows $cs_d = cs_d + 128$.
- In Step 2, the server sends the encrypted response $C_2 = Encrypt(sk, cs_e, R)$ and increments its cs_e counter as follows: $cs_e = cs_e + 128$. The proxy receives the encrypted message and decrypts it to obtain $R = Decrypt(sk, cp_d, C_2)$. The proxy then updates its counter cp_d as follows: $cp_d = cp_d + 128$.

The message M from the proxy can take the following forms:

- LOGIN *username password*
  This message initiates a login request to the server.
- MESSAGE *username1 username2 text*

Here the words in italics represent parameters whose values can vary depending on which users sends the message.

The response R by the server depends on the message M, and can take any of the following forms:

- WELCOME *username*
  This message is sent in response to the "LOGIN username password" message, if the username and the password supplied by the proxy matches an entry in the user database at the server.
- INCORRECT USERNAME
  This message is sent in response to the "LOGIN *username password*" message, if the supplied username is not in the user database at the server.
- PASSWORD MISMATCH
  This message is sent in response to the "LOGIN username password" message, if the supplied password does not match the password for the username stored in the database at the server.
- REPLY MESSAGE *username1 username2 text*
  This message is sent in response to the "MESSAGE *username1 username2* text" message.

Recall that each message, either from the proxy or from the server, must be exactly 128 byte long. If the actual message is less than 128 bytes, it is padded with as many space characters as neccessary to form a 128 byte long message.

The Protocol Simulator

For the purpose of this project, the above functionalities are implemented using a simulator. The simulator runs a client, a proxy and a server and returns the messages transmitted between the proxy and the server in two log files (one for the messages from the proxy and the other for the messages from the server) where the encrypted messages from each party to the other are appended in respective log files.

The size of each message between the proxy and the server is **128** bytes (**128** characters). The attacker has access to the logs containing the encrypted messages. The attacker may read all bytes from the file and split them into 128 byte chunks to process individual message separately.

You may use the provided jar file to generate a different set of log files. Note that each time the messages will be different. A text file that has a username and a password per line is provided with the jar file. Each time you generate a new set of log files, this is used as the user database. Make sure you have the latest JRE installed on your computer and double-click on the jar file to generate new set of log files. You can update the user database ("Users.txt") as you wish for your trials (different combinations and lengths) to make sure your code works for all cases. Note that the simulation jar file only works if it is in the same directory with the "Users.txt". The "Users.txt" file is provided to be used with the simulation only.

Problem 1:

Find out which users are actually logged in to the server, using only the two log files of encrypted messages between the proxy and the server.

Deliverable notes:

Implement your attack in a class named "Problem1". Your program must take two files **only** as input called "ClientLogEnc.dat" and "ServerLogEnc.dat", that represent the encrypted message logs from proxy to server and vice versa. Create an executable, where putting the log files in the same directory is enough to run your solution. Write your output in a file named "Problem1.txt". Put a newline between each username. Write the usernames in the login order.

Example output file:
ALICE
BOB
FRANK

Problem 2:

Reveal the passwords used per login trial, using only the two log files of encrypted messages between the proxy and the server.

Deliverable notes:

Implement your attack in a class named "Problem2". Write your output in a file named "Problem2.txt". State "[CORRECT]" if the server accepts the username and password pair, state "[WRONG]" otherwise. ". Put a newline between each correctness, username, password tuple separated by one space character (" "). Write the tuples in the transmission order. Your program must take two files **only** as input called "ClientLogEnc.dat" and "ServerLogEnc.dat", that represent the encrypted message logs from proxy to server and vice versa. Create an executable, where putting the log files in the same directory is enough to run your solution. Proper output indentation will be rewarded.

Example output file:

```
[CORRECT]    ALICE    asdasfgsdg
[WRONG]      BOB      bhnbfngf
[CORRECT]    FRANK    fsdgdjrdtj
```

Deliverables:

You must only submit the following files in a **zipped** folder (named: **"surname_givenName_matriculationNumber.zip"**.

1) The source code (readable & runnable) – Good commenting and indentation will be rewarded. (your source code file names: "Problem1" and "Problem2")
2) A PDF file "Report_studentName_MatriculationNumber.pdf" that has 3 sections (1. Describe the vulnerability discovered, 2. Describe your attack, 3. Conclusion (the take home message))
3) The output files generated by your code for the .dat files provided (.dat files provided: "ClientLogEnc.dat" and "ServerLogEnc.dat", your output file names: "Problem1.txt", "Problem2.txt").