COMP90043 Cryptography and Security
Project Part A2 Specification

Department of Computing and Information Systems
University of Melbourne
Semester 2, 2015

1 Introduction

The aim of this part of the project is for you to gain some experience in writing stream ciphers and to understand how the encryption process works. As an opportunity for you to gain extra credits, you are also encouraged to implement a block cipher known as Data Encryption Standard, which was once upon a time the standard for data encryption, but now replaced by Advanced Encryption Standard (AES) or, additional, attempt the implementation of a proxy server to show a possible Man-in-the-Middle attack. You should attempt all required sections of the assignment prior to your attempt at DES or MIM.

1.1 Stream Cipher

A stream cipher is a symmetric key cipher where plaintext digits are combined with a pseudorandom cipher digit stream (keystream). In a stream cipher each plaintext digit is encrypted one at a time with the corresponding digit of the keystream, to give a digit of the ciphertext stream. An alternative name is a state cipher, as the encryption of each digit is dependent on the current state. In practice, a digit is typically a bit and the combining operation an exclusive-or (XOR). The pseudorandom keystream is typically generated serially from a random seed value using digital shift registers. The seed value serves as the cryptographic key for decrypting the ciphertext stream.

Stream ciphers represent a different approach to symmetric encryption from block ciphers. Block ciphers operate on large blocks of digits with a fixed, unvarying transformation. This distinction is not always clear-cut: in some modes of operation, a block cipher primitive is used in such a way that it acts effectively as a stream cipher. Stream ciphers typically execute at a higher speed than block ciphers and have lower hardware complexity. However, stream ciphers can be susceptible to serious security problems if used incorrectly (see stream cipher attacks); in particular, the same starting state (seed) must never be used twice.

1.2 Man in the Middle Attack

A man-in-the-middle attack (often abbreviated to MITM, MitM, MIM, MiM or MITMA) is an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other. One example is active eavesdropping, in which the attacker makes independent connections with the victims and relays messages between them to make them believe they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker. The attacker must be able to intercept all relevant messages passing between the two victims and inject new ones.

For the purposes of this subject, you will be 'reverse-engineering' the communication protocol between a client and a server, then modifying the content of messages to demonstrate that you have performed a simulated Man in the Middle attack.

1.3 Data Encryption Standard Block Cipher

The Data Encryption Standard (DES) was once a predominant symmetric-key algorithm for the encryption of electronic data. It was highly influential in the advancement of modern cryptography in the academic world. Developed in the early 1970s at IBM and based on an earlier design by Horst Feistel, the algorithm was submitted to the National Bureau of Standards (NBS) following the agency's invitation to propose a candidate for the protection of sensitive, unclassified electronic government data. In 1976, after consultation with the National Security Agency (NSA), the NBS eventually selected a slightly modified version (strengthened against differential cryptanalysis, but weakened against brute force attacks), which was published as an official Federal Information Processing Standard (FIPS) for the United States in 1977. The publication of an NSA-approved encryption standard simultaneously resulted in its quick international adoption and widespread academic scrutiny. Controversies arose out of classified design elements, a relatively short key length of the symmetric-key block cipher design, and the involvement of the NSA, nourishing suspicions about a backdoor. The intense academic scrutiny the algorithm received over time led to the modern understanding of block ciphers and their cryptanalysis.

The link to the DES specification is as follow:

`http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`

As for the mode of operation of the cipher, you will only have to implement the Electronic Cookbook (ECB) mode.

2 Implementation Details

This section contains all information pertaining to implementation details. It is imperative you read closely what follows as the details for each functions we expect from you must adhere to the specification strictly.

2.1 Provided Packages

Similar to the previous project, you will be provided with skeletons to work with in both Python and Java. On top of that, you should be aware that your code from the previous project will be necessary in this project as this part is a continuation of the previous part.

- Java users should only use Java 1.8
- Python users should only use Python 2.6.6
- DO NOT modify skeleton function names, definitions and etc. You may declare extra definitions or create additional helper functions, but make sure to import them correctly and include them appropriately. If you modify skeleton functions names AND/OR declarations, unit testing will FAIL.

You can download the source code for any language from */home/subjects/comp90043/partB*.

2.2 Auxiliary Functions

Prior to enciphering, you will need to perform auxiliary functions on the DH Key you have obtained earlier in part A1 to derive additional keys for the stream cipher. The following are the auxiliary functions you will need to implement and descriptions included:

`parityWordChecksum(dh_key)`, takes the DH key from part A1, performs a longitudinal parity check, which breaks the input into words with a fixed number of 64 bits, and then computes the exclusive OR of all those words. The output of this function/method should be an long integer of 64 bits in length.

`deriveSupplementaryKey(dh_key,` p_n`)`, takes the DH key from part A1, performs the modulus operation of it with respect to p_n. The value of p_n will be a 64 bits prime number generated by the server and pass to you as a part of a message in the protocol. Please note that p_1 and p_2 correspond to the two derived keys used by the stream cipher, called a and b respectively.

After completing these auxiliary functions, you should submit at least once to test if you have these auxiliary functions implemented correctly before you proceed further as any errors in the Auxiliary functions may propagate further as you perform encipherment.

2.3 Stream Cipher

The stream cipher you will be implementing is a simple stream cipher. Whilst implementing the stream cipher, you should actively consider the security of the stream cipher and consider its usage viability in a production environment. You do not have to do a formal research into this and write a lot to demonstrate your findings, you simply have to comment on the security and the viability of the stream cipher summarily. You should do this in no more than 100 words in a .txt document saved in the `candidate-readme` directory.

To get the initial shift register variable value, you will need to apply `parityWordChecksum` on the DH key negotiated. The initial shift register variable is denoted r_0. Subsequent shift registers will be denoted r_i where $i \in \mathbb{N}$. To obtain subsequent shift register values, the shifting operation is given as $r_i = (ar_{i-1} + b) \mod p$ and $r_0 = parityWordChecksum(k_{AB})$, where k_{AB} is the Diffie Hellman Shared Secret key negotiated in the previous part of the project.

Prior to encipherment, you will have to derive two additional keys for use in the stream cipher, these two keys are denoted (a, b). To derive a, you will be using `deriveSupplementaryKey` on the Diffie Hellman Shared Secret key negotiated in the previous part of the project. Similarly, to derive b, you will be using the same function.

To perform the enciphering, the encipherment function is $E(b_i) = b_i \oplus MSB[(ar_{i-1} + b) \mod p]$, where MSB is the Most Signifcant Byte, b_i is the ith byte in the plain text message. On the other hand, the decipherment function is $D(b_i) = b_i \oplus MSB[(ar_{i-1} + b) \mod p]$, but in this case b_i is the ith byte in the cipher text message. While performing both encipherment and decipherment, it is important that you update the shift registers for each byte you encrypt or decrypt.

The following are methods you will have to implement for the Stream Cipher:

`updateShiftRegister()`, updates the shift register from r_{i-1} to r_i.

`msb()`, gets the most signifcant byte (most significant 8 bits) from r_i. If r_i was an integer less than $2^8 - 1$, then it is okay to simply return r_i directly

`_crypt(msg)`, takes a message as an input either encrypted or decrypted, then output the inverse of it. For example, if the input message is the cipher text, the output should be the plain text.

`reset()`, resets the shift register to the initial state, ie. r_0.
After encrypting an input message, the output message should be encoded in base64 encoding for it to be serializable in JSON format. Similarly, prior to decryption, it is important to apply base64 decoding to obtain the information in raw bits, then apply the decryption algorithm to get the Plain Text.

2.4 Man in the Middle Attack Report

All candidates must write a small report (up to 1000 words) explaining how to perform a Man-in-the-Middle Attack using the provided code and protocol. Use *pseudo-code* and *diagrams* to detail all the steps needed to build the MIM server.

2.5 BONUS: Simulated Man in the Middle Attack

To perform the simulated man in the middle attack, you must reverse engineer the protocol and create your own proxy server using the language of your choice. As the submission daemon will not be performing unit testing on your self-engineered Man in the Middle client, it is your responsibility for ensuring that your self-engineered client outputs all intercepted messages to Standard Output. Prior to output, you should apply the necessary changes in message content to indicate that you have successfully performed a Man in the Middle attack.

To reverse engineer the protocol to implement your modified malicious client, you should read about the Client-Server protocol in the Appendix section of the specification.

To indicate that you have successfully mounted a Man in the Middle attack, you should include the phrase "HAHA! YOU HAVE BEEN ATTACKED!" to the end of every message sent and received during the Communication Phase.

You will also have to output the Diffie Hellman Shared Secret Key between the Client and your Modified Malicious Client PLUS the Diffie Hellman Shared Secret Key between your Modified Malicious Client and the Server. You will have to output these to Standard Output. We won't be able to guess which of the keys will be which, so you will have to print what each key is prior to outputing them to Standard Output.

To aid us in marking your submissions, you should write a few words (no more than 200 words) about how you have implemented your malicious man in the middle server and save it to the `candidate-readme` directory.

2.6 BONUS: Data Encryption Standard

Please read the NIST specification. Remember to include all the constant variables required such as the permutation blocks, transposition tables and etc. There are only 3 main functions/methods you will need to implement and the input and output of those functions/methods are clearly defined in the NIST specification. Partial credit will be awarded for each correct implementation.

To obtain the 64 bit key to instantiate the DES Cipher, simply take the 64 bits starting from the Most Significant Bits (including the Most Significant Bit). The implementation of this cipher must be design to work between the client and the Man in the Middle server only. The main server will not support DES encryption/decryption.
Please, note that the DES implementation should work using your own code. You will have to encrypt and decrypt a message locally. Please, explain in the README file how you developed this and how is possible to observe it in action.

3 Points To Note

- By now, you should be familiar with how to use the University Servers and be familiar with copying and uploading archive files to/from the University Servers. If you don't, learn it immediately.
- Please start working on this part of the project WELL AHEAD of time as the submission system may be overloaded close to the due date of the project.
- Candidates should note that you are still required to work on part A1 of the project prior to working on this project as you are expected to use YOUR OWN CODE from part A for this part of the project. If you have yet completed part A1, you MUST complete that part of the project prior to working on this project.
- It may be a good idea to write generic bit manipulation functions/methods that perform routine tasks such as extracting a bit at a specific bit position, rotate left, rotate rights and etc.
- Similar to the previous part, to protect yourself against suspicion of plagiarism, it may be a good idea to log the progress of your work with a Version Control System such as git, svn or bazaar.
- Students struggling with bitwise operations in Python or Java should contact tutors for help or advice AS SOON AS POSSIBLE.
- If you are struggling to implement a complete version of any cipher, partial credits will be awarded if you attempt to implement the cipher. For example, if you got the register value correct, you will be awarded partial credits for correctness. Partial credit will be awarded according to your methods/functions passing test cases.
- To test your code, use the server available at **dimefox.eng.unimelb.edu.au:8001**. Remember that, for security reasons, the server can't accept more than **10 connections concurrently**. We can't change this so please consider this before suffer some testing problems near deadline.

4 How to get help!?

Candidates are advised that the first point of contact is the discussion forum in the LMS. Please try not to send individual emails to tutors or the lecturer as what you ask will most likely be asked again in future by other students. In the interest of our time and yours, please use the LMS discussion forum for all enquiries pertaining the project. If there is a private matter you wish to discuss, please email Renlord at renlordy[at]unimelb.edu.au or Pablo at serranop[at]unimelb.edu.au.

5 Administration

5.1 Credit Awards

Criteria	**Max Cr.**
Auxiliary Functions	2
Stream Cipher Implementation	4
Man in the Middle Attack Report	2
Quality of Code and Documentation	3
Data Encryption Standard Block Cipher or Man In The Middle Attack	3 (BONUS)

Full credits for this part of the project is 11 marks. However, it is possible to obtain further extra credits if you complete the Data Encryption Standard Block Cipher or the Man in the Middle Attack and the maximum credit awards will be increased to 14 marks.

6 Submission Details

Only include script(s) of your preferred programming language in your archive file. For example, if you have chosen to implement in Python, only include .py file(s) in your archive file and exclude all other scripts we have provided to you.

Submission Instructions:

1. Zip the entire package with your work.
2. Login to [nutmeg, dimefox].eng.unimelb.edu.au
3. run `submit 90043 A2 [whatever].zip` or run `submit 90043 A2` and submit each file one at a time.
4. run `verify 90043 A2` to verify your submission. It should also tell you if it passes all test cases.

Submission Opens: Mid Week 9

Submission Closes: Sunday, 4th of October 2015, 2359 (AEST)

Extensions and Late Submissions

Late submissions will incur a penalty of 10% of your project awarded credits per day (or part thereof) late.

Late Submission Instructions:

1. Zip the entire package with your work.
2. Login to [nutmeg, dimefox].eng.unimelb.edu.au
3. run `submit 90043 A2.late [whatever].zip` or run `submit 90043 A2.late` and submit each file one at a time.
4. run `verify 90043 A2.late` to verify your submission. It should also tell you if it passes all test cases.

Late Submission Open: Monday, 5th October 2015, 0000 (AEST)

Late Submission Closes: Friday, 9th October 2015, 2359 (AEST)

Extensions may be granted on the provision of a valid reason. You must contact either Renlord (renlordy[at]unimelb.edu.au) or Pablo (serranop[at]unimelb.edu.au) at the earliest opportunity, which in most instances should be WELL before the submission deadline. Requests for extensions are not automatic and are considered on a case by case basis. You may be required to supply supporting evidence such as a doctor certificate.

7 Academic Honesty

The university has a policy on academic honesty and plagarism and it is currently being enforced across all computing and information systems subjects. If in doubt of violation of this policy, please visit `https://academichonesty.unimelb.edu.au/`

8 Appendix

Client - Server Protocol

The following is a brief description of the messaging protocol between the Client and the Server to aid you in your endeavors to "reverse-engineer" a proper server to produce a malicious Man in the Middle server to intercept messages between the server and your client. To know the in-depth implementation details, you should read the source code provided to you.

Contact Phase

In the greeting phase, the Client will initiate contact with the server by sending a `CLIENT_HELLO` message containing your student ID. In response, the server will respond with a `SERVER_HELLO` message or your client may be put to a queue if there is an excess number of connections. On receipt of the `SERVER_HELLO` message on the client, the client will proceed to the Exchange Phase. On dispatch of the `SERVER_HELLO` message, the server will proceed to Exchange Phase.

Exchange Phase

Server Side - After the dispatch of the `SERVER_HELLO` message, the server will wait for the Client to respond with a `CLIENT_DHEX_START` message. On receipt, the server will respond with a `SERVER_DHEX` message. This message contains all the diffie hellman key exchange parameters. On dispatch of this message, the server will wait for the Client to respond with a `CLIENT_DHEX` message. Once this message is received, then the Diffie Hellman Key Exchange is complete and server responds with `SERVER_DHEX_DONE`. On receipt of the Client's response, `CLIENT_DHEX_DONE`, the Server will formally proceed to the Specification Phase.

Client Side - Client starts the exchange phase by sending `CLIENT_DHEX_START` message. In response, the server will respond with a `SERVER_DHEX` message. Once this message is received, the Client responds with a `CLIENT_DHEX` message containing the Client's public integer to be shared with the Server. In response, the server will respond with `SERVER_DHEX_DONE`. The Exchange Phase for the Client ends immediately after the `CLIENT_DHEX_DONE` message is sent off.

Specification Phase

Server Side - The server will randomly generate a list of lines of texts it wants from the corpus of the Client and a list of lines of texts it will send to the Client from its own corpus. These details will be included in a `SERVER_SPEC` message. On receipt of a `CLIENT_SPEC_DONE` message, the server will proceed to the Communication Phase.

Client Side - On receipt of the `SERVER_SPEC` message, the Client responds with a `CLIENT_SPEC_DONE` message. On dispatch of this message, the Client proceeds to the Communication Phase.

Communication Phase

In the Communication Phase, the Server will be the first to send messages to Client, followed by messages being sent by the Client to the Server. Prior to each message being sent, the Server must notify the Client of the message length by sending a `SERVER_NEXT_LENGTH` message containing the message length of the following message (a line of text from the corpus). Similarly, the Client will also have to send a similar message to the Server called `CLIENT_NEXT_LENGTH`.

After these messages are dispatched, the Client must respond with `CLIENT_NEXT_LENGTH_RECV`, thus indicating that the Client is now ready to accept a message of a specific length (Server will also responds properly to indicates that is ready to receive the ciphertext). Following the exchange of these messages, the Server (or Client) will send a `TEXT` message containing the line of text from its corpus. Following the transmission of the `TEXT` message, either the client or the server will respond with a `SERVER_TEXT_RECV` or `CLIENT_TEXT_RECV`. This phase is complete once both sides complete the exchange of messages.

The communication phase will finish when server sends a `SERVER_COMM_DONE` message. Client will respond properly using `CLIENT_COMM_DONE`.

Encryption is only applied to the *body* field in the `TEXT` message.

Example of message

Message Structure of Communication Phase `TEXT` Messages:

- type, indicates the type of the message
- id, indicating the number of message in the process
- body, contains the line of text from the corpus. This is where you should apply modification to demonstrate your Man in the Middle attack. ENCRYPTED FIELD (Base64 Encoding).
- n, message counter.