CS255 - Programming Assignment 1 Writeup
==========

Students
==========
Madhukar Krishnarao
Jack Wu

Contents
========

1. Introduction
==========
The writeup explains the various encryption/decryption schemes used in the project. It also explains how keys are generated for message encryption/decryption. The writeup also includes summary of design for MAC integrity checks being done along with MAC key generation. Each section also covers why the key generation is secure and why the design that is implemented can mitigate possible attacks as covered in class.

2. Key Generation
==========
The project requires 3 independent keys for authenticated encryption of messages and 3 independent keys for authenticated encryption of the Group-Key table.

2.1. Key Generation for Authenticated Encryption of Messages

The design requires 3 independent keys. One key for encryption in CBC mode and two for MAC authentication (CBC-MAC).

A root key is first derived using a 128 bit random value and a 128 bit random seed. The random value and seed are fed to the pbkdf2 to generate a ROOT_KEY. 3 different 128 bit salts are then generated and using the ROOT_KEY plus the salts, 3 new 256 bit independent keys are generated. This ensures that the keys used for encryption and MAC tagging are independent of each other.

```
ROOT_KEY = pbkdf2(128_bit_Rand, 128_bit_Salt)
  |
  |----> Key for CBC encryption
  |
  |----> Key1 for CBC-MAC
  |
  |----> Key2 for CBC-MAC
```

The salts used for above key generation are not stored since the keys themselves are stored securely on localStorage.

2.2. Key Generation for Authenticated Encryption of Group-Key table

The design requires 3 independent keys, one key for encryption of the Group-Key table in CBC mode and two keys for authentication in CBC-MAC mode.

In this case, the user's password acts as the base and the three keys are derived from it. 128 bit salts are created for key generation

```
User_Password
  |
  |---+ salt_a +----> Group-Key table key for CBC encryption
  |
  |---+ salt_b +----> Key1 for CBC-MAC tagging of Group-Key table
  |
  |---+ salt_c +----> Key2 for CBC-MAC tagging of Group-Key table
```

The salts used for the above key generation are stored on localStorage, for recomputing the keys during Verification.

2.3. Key security

* All keys used are 256 bit long
* Probability of guessing the key using a brute force attack is neglibible as it is O(2^128)
* The user password is the weakest link in the design. Ensuring the user enters a safe password is not covered in the project scope.

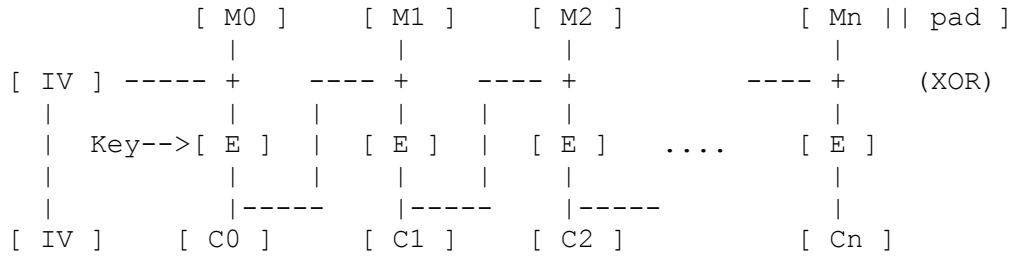
3. Encryption and Decryption using CBC

Encryption and Decryption of message posts as well as the Group->Key table is done using Cipher-Block-Chaining mode.
Detailed design and security constraints are described below.

3.1. Encryption

Encryption is done in Cipher-Block-Chaining mode. The design handles messages of variable length. Encryption follows the design below:

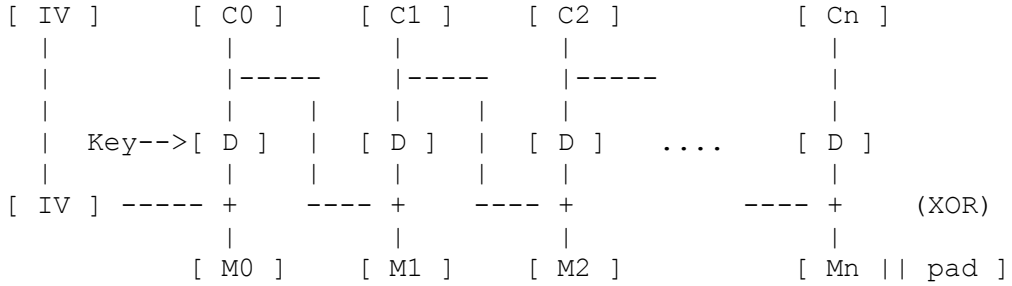
- The AES primitives used in the design require the message block to be 128 bits.
- The message is first padded to block length multiple. In case the message is already a multiple of block length, then a dummy padding block is appended.
- An random 128 bit block is chosen as the IV. The IV will be the first block of data on the final cipherText.
- The IV is then XORed with the first block of the message. The resulting value is encrypted using AES primitive.
- The resulting cipherText block is now used as the IV for the next message block.
- The cipherText block then concatinated with the final cipherMessage.
- The above steps are done in a loop till all message blocks are processed.



3.2. Decryption

Since encryption is done using CBC mode, a corresponding Decryption design is used. The steps are detailed below:

- Strip the first block from the cipherText message as the IV
- The next block of cipherText is decrypted using the AES decrypt primitive.
- The resulting block is then XORed with the IV to get the first block of the message.
- The first block of cipherText is also used as the IV for the next round.
- Once all cipherText blocks are processed, the padding is removed to get the plain text message.



3.3 Security in Encryption/Decryption

- * The underlying AES primitive is assumed to be a Secure PRP. Hence the block cipher is secure.
- * AES block size is 128 bits; This means that 2^64 blocks of data can be encrypted before having to change the key.
- * Although that limit can be reached theoretically, the scope of the project does not cover changing keys.

4. MAC Signing and Verification

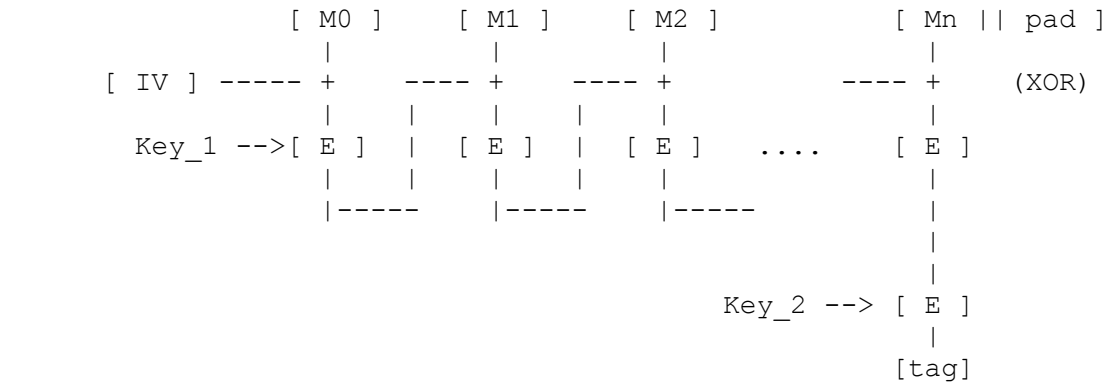
The project uses ECBC-MAC for message integrity. MAC is calculated using the two sets of keys generated which are independent of the Encryption/ Decryption keys.
Detailed design and security constraints are described below.

4.1. MAC Signing

The design uses CBC-MAC tagging. The last block is then encrypted using the AES primitive to get the tag. Two independent 256-keys are used. One for the rawCBC mode and one to Encrypt in the end. Variable length messages are padded to become a block length multiple. The overall design is as follows:

- ECBC uses 128 bit block size and 256 bit keys
- The message is first padded so that it becomes a multiple of the block length.
- A static IV is used as the first XorBlock
- The first message block is XORed with the IV
- The resulting data is then encrypted using the first key. This is done

using the AES primitive.
- The resulting cipher block is used as the IV for the next message block.
- The final message block is then encrypted one last time using a different key using the AES primitive.
- The final resulting 128 bit block is used as the message tag.



4.2. MAC Verification

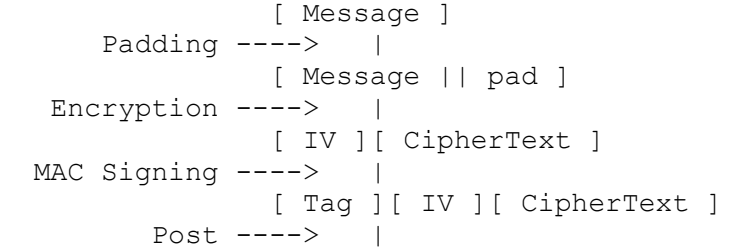
The verification is done by recalculating the MAC tag of the message and comparing it with the given tag. Note that direct bit-wise comparison is done which could lead to timing attacks. The solution for this is not covered as part of the project.

4.3. MAC Security

* The MAC keys are independent of the Encryption key. Without this, the attacker can change every bit of the message except the last block and still have the same tag.
* Using independent keys ensures that the probability of the keys being the same and thus ending up with existential forgery is negligible.

5. Authenticated Encryption of the Message

When the user first enters the message, the keys are fetched from memory. It is then Encrypted and then MAC signed. Below is the flow:



To decrypt, the process if followed in the reverse order to retrieve the

original message.

The keys required for Encryption and MAC Signing are available in the Group->Key table. In case the required keys are not found, suitable messages are posted and no encryption/decryption/tagging are attempted.

All users in the same group need to have the three keys. Transferring the keys between the users is not covered as part of the project. It is assumed that they have established a secure channel for transfer.

6. Authenticated Encryption of Group-Key table

When the user logs out, the keys have to be securely stored on the disk. The project users localStorage for this. To securely store the keys, the table has to be stored with authenticated encryption. This is achieved by first encrypting the Group-Key table and then MAC signing it. As with messages, different keys are used for encryption and signing.

The Group-Key table is first converted to a big object (string in our case) and then encryption and MAC signing are done. This is to hide the group names and number of group-key pairs that are there and any other information that could leak if just the encrypted keys are stored.

Encryption + Signing is done on the Group-Key table every time the keys are saved on the disk.

Verification + Decryption is done each time the Group-Key table is loaded from disk to memory.

During key generation or during message decryption, if the user password is not available (because its a new session or because its not set yet), the user is prompted for the user. In case of decryption, the user is expected to enter the correct password.

An attempt to decrypt the Group-Key table is done using the entered password. In case the password is wrong, the Verification and Decryption fails. At this time the user is prompted to enter the password again. No check is done on the number of attempts the user gets.

7. Issues with cryptography in a browser

- Securely delivering javascript to a browser is a chicken-egg problem.
- Perfect security could be slow (esp for page refresh/responsiveness)
- Unreliable runtime environment.
- SessionStorage is insecure across tabs. One could potentially be logged out of the site, but still have access to sessionStorage on a different one.

8. Side channel attacks

- Once attacker controls web requests, all security is lost since attacker can inject <script> tags to fetch any unencrypted data.
- Get data stored on session storage
- Get data stored in browser memory
- All encryption is based on password. Although attempt is made in the code to leak information about the keys, an attacker can always guess the password using social engineering attacks or dupe the user into entering the password using pishing attacks.
- The keys for encryption/decryption and MAC calculations are fixed. The code does not provide a way to change the keys after their limits are reached. This could potentially lead to compromised security.

9. Assumptions in the code

- Random number generator is truly (Pseudo) random
- No bugs in AES primitives.
- No bugs in pbkdf2 implementation