Team: Hung Tran, Megan Kanne

1. **Maintain a secure database for the keys (one for each group):**

I use function sjcl.misc.pbkdf2 to convert the group name into a key, using a salt. The 128-bit salt is cryptographically-secure and uniquely-generated-amongst-different-users by using function GetRandomValue(). The different groups will have unique keys because the keys are generated using GetRandomValue() and the random salt for the input. The keys are stored securely by function SaveKey() in which I use CBC with random IV to encrypt the key database. The key to encrypt the database is the generated by user's password using the key expansion PRF pbkdf2. This is CPA secure because, the IV is truly random generated by GetRandomValue (Assume that the GetRandomValue() is cryptographically-secure then an attaker cannot predict the IV). Secondly, I use function sjcl.cipher.aes() which is a secure PRP. Therefore, according to CBC theorem, if sjcl.cipher.aes() is a secure PRP then the CBC with random IV is semantically secure under CPA.

Loadkeys is secure because to get the keys database out decrypted, the user must have the correct generated key from the password. If the user enters a wrong password or the key is stored in sessionStorage is incorrectly, the key database will not be decrypted or decrypted incorrectly. This is ensured by the decryption CBC algorithm.

2. **Ensure that the keys are stored securely**

I allow the user to use a key database password to securely store/load the keys. I save the encrypted key database in localStorage. I don't store any decrypted keys in local or session storage. I ask first time users to create a key database password. If they already have a database, I ask them for the password once per session.

I use the generated key from the user password to encrypt and decrypt database key. The generated key is created using random 128 bit salt from cryptographically secure GetRandomValue and the PRF pbkdf2.

I use sessionStorage to store the generated key. Therefore, after I close my browser properly, the generatedKey will be deleted from sessionStorage. An attacker who sits down on my computer after that will not be able to get the generated key and thus, can't decrypt the key database and can't decrypt the message.

3. **Provide a function to generate new keys (in GenerateKey).**

As discussed in part 1, I use pseudorandom function sjcl.misc.pbkdf2 to expand a random value from GetRandomValue() into a key, using a salt. The salt is a cryptographically-secure and uniquely-generated-amongst-different-users 128-bit salt by using function GetRandomValue(). This ensures that the group key is indistinguishable from random (I didn't use Math.random). Key generation is not deterministic.

4. **Build encryption and decryption functions that provide CPA security for Facebook group messages**

All my encryption and decryption algorithms in this assignment (key database encrypt/decrypt, message encrypt/decrypt) use CBC with a random IV. The IV is cryptographically random generated by using GetRandomValue(). Attacker cannot predict this IV (assume the function is implemented correctly). I use padding to ensure the aes block will be multiple of 16 bytes. So according to CBC theorem, as long as sjcl.cipher.aes() is a secure PRP then my CBC algorith m with random IV is semantically secure under CPA. No matter what the message the attacker sends to the encryptor, he will get back a random encrypted message since IV is generated randomly. Even the same plaintext submitted twice will generate different random ciphertext. Therefore, CPA attack does not work (given that the (pL)^2/|X| is negligible)

5. **Build a MAC system based on the AES implementation given to you.**

For this portion of the project we implemented CBC-MAC as our MAC system. This meant changing the size of the key associated with a group to be 128*3 = 384 bits long. In our encryption and decryption algorithms, we slice these 384 bits into three 128-bit keys the first being the plain text encryption key, the second being MAC key 1, and the third being MAC key 2 (for the tag re-encryption). We wrote functions to generate a MAC tag (generatemac) and to validate cipher text (validatemac). They are described in the next section.

6. **Use your MAC system to authenticate group messages.**

When encrypting messages, we first encrypt the plain text using CBC and its 128-bit key. We then generate the MAC tag in a function (generatemac) that takes in the cipher text and the two MAC keys. It does CBC on the cipher text using MAC key 1 with an AES cipher but without an IV (since it was mentioned in class that it is not necessary). Once done, we take the last block to be encrypted, and re-encrypt is again using a new AES cipher and MAC key 2. This last encryption results in the MAC tag. This last encryption is necessary in order to protect the MAC system from existential forgery attacks. We then prepend this tag to the original ciphertext and return that as the encrypted text.

We don't use padding in the MAC system because it will always be taking a cipher text as its argument that is of the proper length. We pad the plain text at encryption time to ensure it is always a multiple of 128 so that the cipher text is always a multiple of 128. We do check that the text argument of validatemac is a multiple of 128 (that it is a valid cipher text) and fail if it isn't.

Validation of the cipher text proceeds in a similar manner (validatemac). We run the MAC tag generation function on the cipher text from the FB wall using the three keys recovered from the database. This results in a MAC tag specific to our cipher text. We then compare this new tag to the tag that was prepended to the cipher text using sjcl.bitArray.equal. We use this

function rather than an equality comparator because we need the comparison to run in a predictable amount of time (the comments on this library function guarantee that it is a predictable time comparator). If we used an equality comparator that went bit-by-bit-, our MAC system would be vulnerable to timing attacks.

7. **Use your MAC system to make sure keys in the key store haven't been changed between program runs.**

The use of our system to sign and verify the key database proceeds in a manner similar to that above. We have changed our password handling to generate a 384-bit key hash from the password input by the user (using the PRF sjcl.misc.pbkdf2). As above, we break this into three keys. We use the last two as MAC keys to run the generatemac function described above on the key database after it has been encrypted and prepend the resulting tag to the cipher text. We save the result in local storage.

Similar to above, we validate the key database by removing the tag from the beginning and expanding the password input by the user to 384-bits using pbkdf2. Then, we run validatemac on the key database retrieved from storage using the two generated MAC keys before decrypting the database. Because we use the same MAC system and functions for database integrity as for message integrity, as described above, our entire MAC system is secure against existential forgery and timing attacks.

8. **What are some of the biggest issues with doing cryptography in the browser? Why might we want to do it anyhow?**

The biggest issue here is that if I don't close the browser properly after each session or I leave my session on, an attacker can sit in my computer and steal my generated key from sessionStorage. With that, the attacker can encrypt and decrypt every messages. Another danger is when people in the same group share group key with each other without a secure channel. There is no guarantee that an attacker cannot get his hand on the key during the key transfer process. We want to do it because it is still more secure than storing plaintext, limiting the number of people who can see the text. And as long the users follow the proper protocol, the chance of breaking the cryptography is low. Now that we have included MAC to weed out corrupted or malicious cipher texts of keys or messages, the chances of successful attack are even lower.

9. **How could somebody go about circumventing the security of your implementation, if they really wanted to? (e.g. side-channel attacks)**

As discussed in the previous answer, the attacker can:

+ Use man-in-middle attack during the key transferring process to get the group key

+ Try to get my generated key from sessionStorage if my browsing session does not close properly

+ Side-channel attack (timing attack, power monitoring attack, acoustic cryptanalysis, etc..)

+ Social engineering to get the users' key/password