Facebook Encryption

(Milestone 1)

Team Members:

Hai Xiao (SUNetID: haixiao) Amit Chattopadhyay (SUNetID: amitch)

Design Aspects

- **Facebook Login**

- *First Time User*

When a user connects to Facebook we use the following process to secure the connection for the encrypted groups:

1. At login (first time after project JS extension installed) user creates a plaintext password that becomes the master secret for validating the user reenter later on, and it is also used to derive the encryption/decryption key for the user's copy of the group-keys (here is the facebook messaging key per facebook group) database.

2. Since the password is the master secret for the user authentication and it is an ultimate for retrieving user's secrecy, it is never been stored anywhere in the system (except for local variable in memory), neither in plaintext nor ciphertext! So to validate a returning user and grant his/her access to saved/encrypted databse information, We designed secure hash (using sjcl.cipher.aes) to generate digest from user's input password alone with a truly random salt:
    a. Salt: A 128-bit cryptographic secure random salt value generated using GetRandomValues() that comes with good enough entropy.
    b. Salted Password Hash: It is basically a Hash Digest generated on the base64 codec concatenation string of the plain text password and the salt.
    c. Final hash digest string is concatenated with salt (base64 codec string), and saved as persistent localStorage object with user ID encoded in its name.
    d. As the concatenated digest||salt is saved in localStorage, so whenever user returns (re-login) code can always re-compute the digest and compare for password validation!

    By storing the random salted password hash, we intend to prevent rainbow table attacks based on low entropy plaintext password.

    There are other ways to verify a returning user's password, but still need the secure aes128_hash that we implemented (use Merkle-Damgard Construction with Davies-Meyer compression).

    E.g. instead of aes128_hash(password||salt), we can aes128_hash(user's database key||salt), as user's key-database key is derived from user's input password by PBKDF2. This way user's identity can be verified as well.

    Both approaches have provided enough security from following aspects:

    - Both case password is salted, and the salt in use is truly random.
    - aes128_hash is secure given our design & sjcl.cipher.aes is secure
    - aes128_hash() generates 128bit digest, it takes O(2^64) evaluations of AES (E,D) to find a hash collision (the birthday paradox). Which is believed to be secure for this application
    - PBKDF2 is believed to be secure.

    In coding, we just choose the first approach as it is simple and secure enough.

3. As mentioned earlier the plaintext password is never used directly nor stored anywhere. Code use it to verify user and derive a 128-bit key to the user's copy of group-(messaging)keys database, with a truly random salt. This is done using the PBKDF2 (Password Based Derivation Function 2), a key derivation function that is part of the RSA Lab's Public-Key Cryptography Standards.

    PBKDF2 applies a pseudorandom function to the plaintext password along with a salt value and repeats the process many times (1000 in code) to produce a derived database key. The process of key stretching allows more security as brute force attacks (exhaust search) on the 128-bit key will consume lot of computational power & time.

    Related to this, we have this PBKDF2 using salt saved to persistent localStorage, as we can't save the PBKDF2 derived database E/D key to localStorage. As a result we can save the user's encrypted group-keys database in localStorage. This is secure, as nothing from localStorage can be decode w/o a valid user password from input.

Facebook Encryption

(Milestone 1)

Team Members:

Hai Xiao (SUNetID: haixiao) Amit Chattopadhyay (SUNetID: amitch)

4. To enable the user the convenience of not having to reenter the password each time loading or changing a page during the session, we store the PBKDF2 derived database E/D key in the sessionStorage. This is believed to be secure as sessionStorage is only kept during the open session period, it is automatically erased at session close. It is also thought to be safer than storing any plaintext (or decrypted) information.

   Again, we can't avoid active attack such as malicious code injection by simply doing this, if hacker can monitor or modify memory content. But that is beyond the scope here.

5. With the PBKDF2 derived database E/D key, code can using aes128_enc() to encrypt the groups-keys(messaging) database and aes128_dec() to decrypt it back to plaintext per user context or session. Today we have not added a lot extra security design to this database encryption scheme, e.g. add security padding to hide length from attacking. But we believe extra can be done over time.
   The encryption scheme we use is just using the derived database key (per user) to encrypt the key database. In design we use nonce counter mode from AES-128 block cipher. We consider no need to re-generate database encryption key even with the many-time key security concern, the reason being we will never have an encrypted database object close to 2^64 blocks long (16byte/block), neither will have up to 2^32 encrypted database objects. So the one time derived (by PBKDF2) database key per user is used for good for its database E/D with AES-128!

- *Returning User*

1. At user login or a new browser session/tab reopen, if the user's hashed password-salt object is already present in the localStorage, we assume this is a returning user.

2. Then we request the user to reenter the password so that we can verify the user with matched (re-compute password-salt hash and compare to saved one) password, so guarantee the user an access to his/her database without an semantic error(e.g Ops w. mismatch DB keys). If user input password verification doesn't pass, code continues to prompt user password (a better design would be bail out after certain times). If this passes, code will regenerate the user database E/D key (use PBDF2 to derive alone with the key derivation salt saved in local Storage) again & cache/set it in sessionStorage.

3. Password validation: specifically we recomputed the aes128_hash(input_password||saved_salt) , the salt for a user's password-salted hash was stored in localStorage at user's first login with password creation. Then compare it against the stored hash digest that was initially generated.

- **Group messaging Keys Generation**

We design to use AES-128 nonce counter mode to encrypt and decrypt the facebook messages exchange within a group. GenerateKey() function is designed to generate cryptographic secure 128-bit random key for AES-128 E/D, it uses provided GetRandomValues() function.

The key (per group) generated is converted to base64 codec string before assigning to group➔key keys map. This is necessary as it will show up on facebook UI extension (www.facebook.com/settings) as copy-paste able string (i.e. no none displayable CHAR) so that user can expose it to other users in the same group easily.

Later user from the same group would have to 'copy-paste' the same key string to their UI, then use 'AddKey' to roll-in the same message key to the matching group.

- **LoadKeys/SaveKeys for Group Keys**

We chose to encrypt the user's group➔key keys map to be the encrypted database. The encryption uses AES-128 nonce counter mode still (to keep code space small, even we do have CBC mode implementation, we don't include it here). This database encryption is also done at per user level, different user will have their own copy of encrypted group➔key keys database stored in the localStorage.

SaveKeys() function is to encrypt the global keys map and save to localStorage (disk), whenever there is a change made to the keys map.

Facebook Encryption

(Milestone 1)

Team Members:

Hai Xiao (SUNetID: haixiao) Amit Chattopadhyay (SUNetID: amitch)

LoadKeys() is primarily the reverse, except that it deals with some extra logic regarding user login password creation or confirmation, and localStorage/sessionStorage objects management (mainly the user login and revisiting state machine), etc.

As earlier mentioned, we believe using nonce CTR mode for database E/D is appropriate due to the fact that 1) the database size won't be big enough (<< 2^64 blocks); 2) the total encrypted database copies (from different users) in a system/browser would also be << 2^32 (birthday paradox).

➢ Facebook Group Messaging

To secure the messages published on the Facebook group we preferably used the nonce CTR Mode versus CBC mode for AES-128 E/D for various reasons (more than following):

- *Provably secure* – The CBC mode can be susceptible to hamming distance attacks as two adjacent blocks are related in some way. Instead CTR mode using a random nonce with a counter allows for a secure and block independent implementation.
- *Software efficiency* – The nature of CTR mode allows for parallelization allowing for faster encryption/decryption in certain hardware. In CBC mode, a previous block needs to be encrypted/decrypted before one can proceed to the next block, leading to sequential processing and making inefficient use of vector architectures.
- *Random access* – Since the integrity of a block is not related to a previous block, in CTR mode block errors can be constrained.
- *Better security bound (error term)* – $\frac{2q^2L}{|X|}$ vs. $\frac{2q^2L^2}{|X|}$
- *No dummy padding necessary*

Once the key of a group is acquired by user, he/she can encrypt plaintext message first then post it. On the other hand, if user rolled in a correct matching group message key, the 'secret' post made by other group members will instantly be decrypted. While non-member of a group won't have access to the group's messaging key, so he/she can't see the decrypted message within that group.

There is no need to re-generate group messaging key, as 1) message count in a group would never be close to 2^32! 2) block counts per message would be far less than 2^64, in fact it is limited to ~10,000/16 by facebook!

Our nonce CTR Mode encryption/decryption works as follows:

➢ Encrypt/Decrypt

Encrypt() will call aes128_enc() if plaintext length isn't 0 and it doesn't start with special 'aes128:' delimiter (to the corner case that plaintext really starts with 'aes128:', we haven't taken care of it!).

aes128_enc() works as follows (nonce counter mode is a good choice in this multiple parties messaging scenario vs. det. counter mode):

- Use 64-bit random nonce (GetRandomValues), it is renewed per every message.
- Use 64-bit counter for block IDs in a message.
- Derive 128-bit IV where the first 64 bits are the random nonce & the last 64 bits are the counter.
- Loop through the message in 128-bit block sizes:
    - Encrypt the IV Block using SJCL AES primitive with the Group Key.
    - XOR the encrypted IV Block with the corresponding Plaintext Block to form an encrypted Ciphertext Block.
    - Concatenate (at tail) this encrypted Ciphertext Block with previous concatenated ciphertext blocks so far.
    - Increment the IV counter field as moving to next block
    - In the end, append the initial IV at the beginning of the ciphertext (for decryptor use)
- In case we have a block which is < 128-bits
    - We apply the above operation only to the valid smaller bitArray segment.

Decrypt() will call aes128_dec() only if ciphertext starts with 'aes128:' delimiter.

The aes128_dec() works in reverse manner, with two main different points:

- First it needs to recover the initial IV from the ciphertext header.
- Then instead of XORing the encrypted IV Block with the corresponding Plaintext Block, we XOR the encrypted IV Block with the corresponding Ciphertext Block to get a Plaintext Block

Both Encrypt() and Decrypt() take care of exceptional cases, when keys[group] is missing.

Facebook Encryption

(Milestone 1)

Team Members:

Hai Xiao (SUNetID: haixiao) Amit Chattopadhyay (SUNetID: amitch)

- **Cryptographic Hash**

We implemented aes128_hash() as secure hash for the required password verification functionality.

For aes128_hash(), we use Merkle-Damgard Construction with Davies-Meyer compression with a fixed IV (128Bit). We take that fixed IV from MD5 implementation.

128Bit digest doesn't seem to be very long but it's enough to this facebook application criteria: assuming underlying sjcl.cipher.aes is an ideal block cipher, then with 128bit block/digest size, it takes O(2^64) evaluations of AES (E,D) to find a hash collision (due to the birthday paradox).

÷

Security Aspects

- **Key Storage**
  1. Database Key Derivation Salt (to PBKDF2) is generated randomly using GetRandomValues() with cryptographic good entropy.
  2. User Password Secure Hash Digest is generated using a combination of the plaintext password and cryptographic random salt, making it harder to do rainbow table attack.
  3. User's Database Derived Key is generated using PBKDF2.
  4. As master secret of a user, we never store the user's database password anywhere neither plaintext nor ciphertext. Instead we only store the Password Hash along with the random Salt in LocalStorage.
  5. All users' Group-keys databases are encrypted using AES128 block cipher, stored in localStorage
  6. User's Group-keys database E/D key is never stored in localStorage, but sessionStorage only.
  7. There is no plaintext Group-keys database stored neither in localStorage nor sessionStorage.

  There is no information (except for database length potentially, as it is yet security padded) leaked about key lengths or aspects of the plaintext key/password/group name to an attacker looking at localStorage.

  The sessionStorage is used to securely contain the database E/D key (derived key from password and salt) which makes it hard for performing brute force attack and the attack to sessionStorage is less likely as it is session life span limited.

- **Group Key Generation**

GetRandomValues() gives us a 128-bit random value with high entropy, so its unlikely a brute force attack is feasible.

- **Nonce CTR mode Encryption (Dec) CPA analysis**

1. We use AES-128 bit key length which is considered to be secure choice at this time.
2. The construction of the IV for CTR should be secure enough so that the probability of its overlapping with a same K in F(K, *) is really minimized. In our choice nonce space $is\ \{0,1\}^{64}$, counter space is $is\ \{0,1\}^{64}$.
3. CPA analysis. Consider the hybrid nature of nonce+ctr mode, its security bound must be better an assumed pure random CTR mode with $X \in \{0,1\}^{64}$, but worse than assumed random CTR mode with $X \in \{0,1\}^{128}$(if message length L is not long enough!).
   Now we try to derive the secure error bound for out nonce + ctr implementation to this facebook secure messaging project:
   Method I. (as description above)

$$\frac{2q^2L}{2^{128}} \leq \text{Nonce-CTR-ErrTerm} \leq \frac{2q^2L}{2^{64}}$$

   Method II. (using probability theory and approx.)
   Condition: We know that given chosen nonce, there won't be overlapping of the IV from message being too long (longer than 2^64 blocks), this is for sure, as facebook message length upper limit is 8,000~10,000 bytes today, very small. So the only IV overlapping criteria would be a duplicated nonce being chosen. Given **q** independent queries, the total probability of having any overlapping is **q** times of the probability for any two queries that a same nonce was randomly chosen in $\{0,1\}^{64}$, that is:

$$\text{Nonce-CTR-ErrTerm} \cong \frac{q^2}{2^{64}}$$

   Method III (most accurate birthday paradox method)

$$\boldsymbol{Pr}[\exists \boldsymbol{q_1} \neq \boldsymbol{q_2}, \boldsymbol{nonce1} = \boldsymbol{nonce2}] \geq \mathbf{1} - \boldsymbol{e}^{-\frac{\boldsymbol{q^2}}{\boldsymbol{2|X|}}}$$

   q is number of message queries from Adv A
   $X \in \{0,1\}^{64}\ is\ the\ nonce\ space.\ |X| = 2^{64}$

   Now problem changes to make sure derived ErrTerm to be smaller than assumed $ADV_{CPA}\ (A, E_{CTR})$ upper

Facebook Encryption

(Milestone 1)

Team Members:

Hai Xiao (SUNetID: haixiao) Amit Chattopadhyay (SUNetID: amitch)

bound. E.g. **if we require** $\boldsymbol{ADV_{CPA}\ (A, E_{CTR}) \leq 1/2^{32}}$ Then we have ➔:

$$\boldsymbol{1 - e^{-\frac{q^2}{2|X|}} \leq 1/2^{32}}$$

➔

$$\boldsymbol{q \leq 92,680}$$

This proves that for the cryptography scheme to become insecure a facebook group users would need to post a total of ~100,000 messages (at a strong $ADV_{CPA} \leq 1/2^{32}$). After that we should re-generate a new group messaging key. But given the project criteria, we consider this secure for the time-being ☺!

This NONCE + COUNTER scheme is also considered to be acceptable for users' key database encryption initially (please see page 1-2 for earlier approx. analysis, but those may not be as accurate as here).

If a stronger sem. Sec. under CPA is required, we can move onto random CTR mode. Also we can move onto AES-256 from AES-128. Please let us know.

÷

Issues with Cryptography in Browser

There are several issues with cryptography in browser:

- Different implementations of browser may have insecure primitives in the runtime. For example GetRandomValues() a key primitive in cryptography may have poor entropy leading to easy attack.
- Javascript based cryptography is malleable as functions be overwritten through injection from DOM or during cross site scripting. It is not possible to guarantee a safe clean sandbox for the crypto code to run.
- Browsers may not be cleaning up memory securely after ending a session (Javascript is GC based), allowing memory scan attacks.
- Partial uses of transmission of data like TLS for some transactions but insecure transmission of Javascript code makes it inherently insecure.
- Poor Crypto Library implementation could lead to incorrect usage and defects.
- Could require computational power on the client making it hard to implement on low end devices like phone.

÷

Side Channel Attacks

Although our usage of cryptographic techniques and patterns is secure, there are certain ways to circumvent the security:

- Attacker can perform script injection into the chrome extension. If one takes control of the extension, the attacker can add code to expose sessionStorage and other communication.
- We assume the attacker cannot take over session storage and look at the master db key as it can then be used to decrypt all of the localStorage data.
- The attacker doesn't design a Facebook equivalent page and gets the user to enter the password. This type of attack is called Phishing.
- An attacker can monitor the timing of encryption/decryption of messages to guess the key length and other characteristics.
- An attacker can have a keylogger or other malicious software installed through other means circumventing any security on the system.
- Brute force attacking the system. Currently there isn't a user account lock out after unsuccessful attempts.

÷