CS 255 – Project 1 (Part 1)

Sammy El Ghazzal
(SUNetID: selghazz)

Sébastien Robaszkiewicz
(SUNetID: robinio)

1 Implementation

1.1 Maintaining a secure database

In the rest of the document, the *database* will refer to a JSON object called `keys`. In this JSON object, the key-values pairs are as follows:

- Key: it is the name of a Facebook group a user belongs to;
- Value: it is the (cryptographic) key used to encrypt the messages in this group.

1.1.1 Initialization

When a user connects to Facebook *for the first time*, we use the following process.

Password The first time a user connects to Facebook, we ask him / her to set a password to encrypt the database thanks to the function `prompt()`.

Salt Once the password has been set, we generate the salt (a `bitArray` of length 128) with the function `GetRandomValues`. We convert the salt to the `base64` string format, and store this string in plaintext in the `localStorage`.

Key We generate a key k (another `bitArray`) thanks to the function `sjcl.misc.pbkdf2`: the key is derived from the password and the salt mentioned above. This key will then be used to encrypt the database through *Randomized Counter Mode* on top of AES (see below). We convert the key to the `base64` string format, and we store it in `sessionStorage`.

1.1.2 Regular use

When a user comes *back* to Facebook (*i.e.* when the plaintext salt already exists in `localStorage`), we ask for the password in order to decrypt the database. Note that as the key is stored in the `sessionStorage`, the user is not asked to provide his / her password everytime the script is loaded, but only when he / she logs in on Facebook.

Checking the password When the user enters his password, we generate a key k' like previously, that is, we use the function `sjcl.misc.pbkdf2` with the password, and the salt stored in `localStorage`. In order to be able to check that the password is correct, we added a dummy entry[1] in the database, `''CXJTucBBuM/eQQV6 ... Ur5uvcLL0=''` `=> ''0000''`. To see if the password is correct, we try to decrypt the database with the generated key k': if we can read the entry `''CXJTucBBuM/eQQV6 ... Ur5uvcLL0=''` and its associated value `''0000''` (*i.e.* if $k' = k$), then the entered password is the right password. If not, we ask for the password again. If the user doesn't want to enter a password, we terminate the script.

[1]Note that the choice we made for the dummy entry supposes that the user will not create a group named "CXJTucBBuM/eQQV6 . . . Ur5uvcLL0=", which seems reasonable: this is a 1024-bit string generated at random thanks to the function `GetRandomValues`. You can see the whole string in the `README.txt` file.

Encrypting / decrypting the database when the password is correct We chose to encrypt the entire `keys` JSON object by:

- Converting it to a string (simply using `JSON.stringify`;
- Encrypting the whole string with the key k. We use the *Randomized Counter Mode* construction on top of AES, as explained in section 1.3.

Then, we store this encrypted stringified JSON object in `localStorage`.

1.2 Generating new keys

The function `GenerateKeys` is pretty simple as it relies heavily on the `GetRandomValues` function. We chose to generate 256-bit keys (even though as we saw in class, the 256-bit AES implementation may be attacked in 2^{99} under certain conditions). For convenience, we store a `base64` encoding of the key in the database `keys`.

1.3 Encryption and decryption functions that provide CPA security for Facebook group messages

Choice of the block cipher We chose to implement the *Randomized Counter Mode* construction on top of AES because of the advantages it presents compared to other constructions (CBC in particular):

- It is parallelizable (although in general Facebook messages will not be long enough to justify the need for a parallel system);
- In case a block is corrupted, only one block of ciphertext will be corrupted[2];
- Its security bound is better (see Section 2 for more details).

[2]In our particular case, the fact that one block is corrupted is sufficient to throw an error in the JS script and therefore stop the decryption / encryption taking place, but this remark goes beyond the scope of this assignment.

For the IV, we use a 128-bit nonce (which corresponds to exactly one block) chosen at random thanks to the function `GetRandomValues`. This nonce is concatenated (in clear) at the beginning of the ciphertext.

Padding Although padding is not necessary when using *Randomized Counter Mode*, our encryption and decryption schemes use padding whenever the encrypted message length is not a multiple of the block size and a dummy block otherwise (for convenience). More precisely:

- If the plaintext has a length that is a multiple of the block size (namely 128 bits *i.e.* 16 characters), we add a dummy block of 16 characters `f` at the end of the plaintext For instance "Hello nice world" would be transformed into "Hello nice worldffffffffffffffff" before being encrypted;
- If the plaintext has a length that has $1 \leq r \leq 15$ characters after the end of the last full block, then we add $16 - r$ times the encoding of $15 - r$ in hexadecimal[3] at the end of the plaintext before the encryption. For instance, the sentence "Hello world" which has 11 characters, would be padded as follows: "Hello world4444".

When decrypting, we consider the string as if it had a length that is a multiple of the block length, and then we remove the number of characters indicated by the last character.

Encodings We experienced some trouble with the encodings of strings. As a result, the encoding of the plaintext and ciphertext are different: the plaintext is considered to be a `utf8String` and the ciphertext is encoded in `base64` string format. When decrypting, instead of splitting the ciphertext into 16-character chunks, we split it into 24-character chunks (size of a block for a `base64` encoding.

[3]The fact that we chose to encode the residual length in hexadecimal is that it allows us to have only one character for double digit lengths.

Remark *For now*, in order to indicate to the script which messages are truly encrypted (and thus need decryption), and which messages were posted in plaintext to the Facebook group, we kept the "rot13:" prefix for all encrypted messages. In the second part of the project, we will be able to get rid of this trick using message integrity checks.

2 Security

2.1 Security of the key storage

Each step of the process is secure:

- The salt is generated at random based on Chrome's `window.crypto.getRandomValues` function, which provides cryptographically secure random numbers;

- Key generated from that salt along with the password using PBKDF2, as recommended by RSA's PKCS #5 standard;

- We encrypt / decrypt the database using the same algorithms as for encrypting / decrypting message, please refer to section 2.3 for security details about those algorithms.

That way, the only information that an attacker has about the database is its total size.

In summary, the salt and the securely encrypted database are stored in `localStorage` (which is the only thing the attacker can access), whereas the database key (which is generated securely) is stored in `sessionStorage` and the attacker is not supposed to see it (but this assumption may not always be verified so this is one point that we will mention in the section "Difficulties of doing cryptography in a web browser"). Note that during the whole process, we never store the password anywhere.

2.2 Security of the key generation

The key generation step relies on the `GetRandomValues` function. The design of this function ensures that information encoded by such a key is indistinguishable from random (unlike a function from the javascript `Math` library for instance).

2.3 Security of the encryption / decryption steps

There are two main steps to "prove" the security[4] of our encryption and decryption schemes:

- The underlying primitive we used for block cipher is *AES-256* which is supposed to be a secure PRP;

- The construction we used on top of *AES-256* is *Randomized Counter Mode*, which is secure under CPA as long as the nonce space is large enough and the nonce is chosen at random (we took a 128-bit random nonce so it is the case here). More precisely, if q is the number of queries the adversary A is allowed to do, we know that there exists a PRF adversary B, such that:

$$\begin{aligned}&\mathrm{Adv}_{CPA}[A, E_{CTR}] \leq \\ &\qquad 2\mathrm{Adv}_{PRF}[B, \mathrm{AES}] + \frac{2q^2L}{|X|}.\end{aligned} \tag{1}$$

  where in our case $|X| = 2^{128}$ and therefore, if A is able to break E_{CTR} under the assumption that $\frac{2q^2L}{|X|}$ is negligible[5], then it would mean that $\mathrm{Adv}_{PRF}[B, \mathrm{AES}]$ is non-negligible, *i.e.* that adversary B would be able to break AES, which we suppose to be impossible.

[4] In the remainder of this write-up, we use "is secure" instead of "supposed to be secure".

[5] This assumption basically means that the user of the script should change his key when the term $\frac{2q^2L}{|X|}$ is non-negligible, *i.e.* that the user uses the library correctly.

Important note To argue for the security of our encryption and decryption schemes, we make the important assumption that $\frac{2q^2 L}{|X|}$ is negligible, *i.e.* that the user changes the key whenever $q^2 L$ is "too big". In our script, we *do not check* that this is indeed the case[6]. Just to have an idea of how many messages can be sent using one key, let us do a quick calculation: Facebook requires the message length to be less than 10,000 characters long, that is $L = 625$ blocks. Then we compute that the number of such messages sent q should be such that (we say that the advantage is negligible if it is less than $\epsilon = 2^{-80}$):

$$q \leq \sqrt{\frac{\epsilon |X|}{2L}} = q_{max} \Leftrightarrow q_{max} \approx 470,000.$$

That is, for the encryption / decryption scheme to become insecure, the user would need to send around 470,000 messages of 10,000 characters each. Saying that this is too much for a single group is probably reasonable, even if to be more secure, it would be better to implement a system that keeps track of the number of messages being sent through the group.

[6] Jeffrey Chen told us that it was ok not to make this checking.