CS 255 – Project 1 (Part 1)

Sammy El Ghazzal
(SUNetID: selghazz)

Sébastien Robaszkiewicz
(SUNetID: robinio)

1 Implementation details

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 to set a password to encrypt the database thanks to the function `prompt()`. After this step, a salt is automatically generated.

Salt We generate the salt (a `bitArray` of length 128) with the function `GetRandomValues`, which is optimized to produce enough entropy. We convert the salt to the `base64` string format, and store this string in plaintext in the `localStorage`. toto why is this secure (to be discussed in next section)

Key We generate the key (another `bitArray`) thanks to the function `sjcl.misc.pbkdf2`: the key is derived from the password and the salt mentioned above. We convert the key to the `base64` string format, and we store it in `sessionStorage`. (Note that consequently, we dont store the password anywhere.)

Encrypting the database We chose to encrypt the entire `keys` JSON object by converting it to a string, and encrypting the whole string with the key mentioned above[1]. We store this encrypted JSON in `localStorage`.

1.1.2 Regular use

When a user comes *back* to Facebook (*i.e.* when the database already exists), we ask them for their password in order to decrypt the database.

Is the password correct? When the user enters his password, we generate a key just like previously: we use the function `sjcl.misc.pbkdf2` with this password, and the salt which is already stored in `localStorage`. In order to be able to check that the password is correct, we added a dummy entry in the database, `00000000 => 0000`. Then, we try to decrypt the database with the generated key: if we can read the entry `00000000` and its associated value

[1] That way, the only information that an attacker could get if he doesnt have the right password is the size of the encrypted database. This would not have been the case if, for example, we had chosen to individually encrypt each key and each value from the database `keys`.

0000, 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.

1.2 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 is buggy and can be attacked in 2^{112} toto verif this number. 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)

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 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. 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, namely 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.

2 Security

2.1 Security of the key storage

Never store the password anywhere.

Salt generated at random (enough entropy)

Key generated from that salt along with the password using PBKDF2

[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.

[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.

Only information about the database which is stored on the disk is the encrypted database (cf. encryption/decryption algorithms below, we use the same to encrypt the messages and the db) =¿ the only information we practically have about the db is its size.

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 toto 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 have that there exists a PRF adversary B, such that:

  $$\mathrm{Adv}_{CPA}[A, E_{CTR}] \leq 2\mathrm{Adv}_{PRF}[B, \mathrm{AES}] + \frac{2q^2L}{|X|}$$

  where in our case $|X| = 2^{128}$.

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