CS255: Introduction to Cryptography **Winter 2013**

Project 1: Encrypted Facebook groups

Botao Hu (botaohu@stanford.edu), Borui Wang (borui@stanford.edu) *February 13, 2013*

1 Response to Milestone 1 Feedback

In response to the milestone 1 feedback, here are several items we want to specially emphasize to make sure the TA understands our design (also commented in code).

- Notice that we only store the encrypted password in sessionStore and the salt for the password in localStorage. We NEVER store the clear text of password in any location.
- Notice that we use a random nonce and XOR the nonce with the counter to generate the IV. The counter assumes that we're not encrypting over 2^{32} bits of data.
- Notice that we first encrypt the message, then MAC the cipher text, which is suggested to be more secure in class.
- Notice that the encrypted db password and the encrypted group keys are all CBC-MAC-ed, and no same MAC key is used for multiple purposes.
- As we are using callback functions for UI and it's quite complex to change our code back to prompt again. We'll leave it there. We're sorry to create the extra work for you to test through it manually. We made sure we passed all tests suggested in https://piazza.com/class#winter2013/cs255/137

2 Design and Implementation

2.1 Assumptions

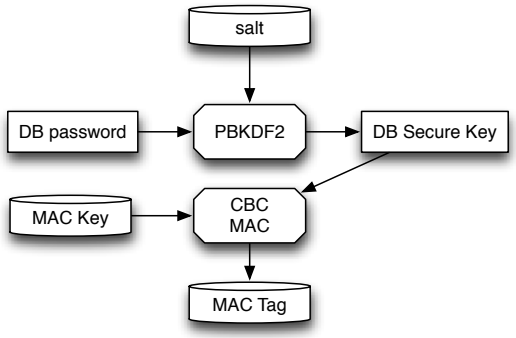
The attacker can see all data in localStorage and the code of our extension in Chrome and can not see any data in sessionStorage in Chrome (because the user closes the browser before attacking as the problem setting states). The usernames are exposed to the attacker because the usernames are leaked by Facebook's cookie no matter what you do. The message posted in the facebook will not exceed 2^{32} blocks.

2.2 Creating and storing database keys

When the user firsts visit facebook.com and logs in, we ask her to setup a database password. We used the database key plain text provided by the user together with a randomly generated 128 bit salt to generate a encrypted database key using PBKDF2, shown in figure 1. The purpose of salt here is to add entropy to the password chosen by the user and prevent attacker to precompute rainbow tables to lookup encrypted keys.

The encrypted database key is then used to encrypt and decrypt the group keys in the database, and is only stored in the user's sessionStorage, which means that the encrypted database key is never written to a persistent data store. The attacker will not know any information directly about the database key from the local storage. We stored the salt on the user's localStorage on disk to recalculate the encrypted database key later. If the salt is missing, we ask the user to re-setup their database.

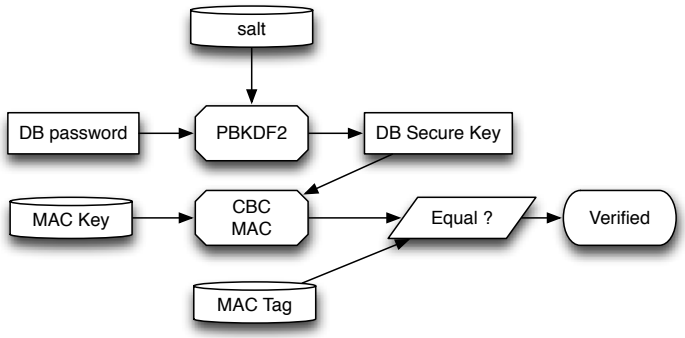
Figure 1: Setup the database password. The salt, MAC key, and MAC Tag is stored in localStorage.



2.3 Verifying if the user enters the right password

To verify if the user enters their password correctly, we generate a MAC tag (using CBCMAC) for the encrypted database key and store it in the localStorage on disk, which is illustrated in figure 2. Later we re-evaluate the MAC tag when user enters her password and verify if it equals to the MAC tag on localStorage. We used a randomly generated MAC key to generate the tag, where the MAC key is also stored on localStorage. Note that using MAC here is only for verification purpose and it does not prevent forgery (the attacker can use the MAC key to generate a new tag, then overwrite the tag on localStorage, but the attacker wouldn't decrypt the database correctly unless he know the database key. It can not be used to access other user's database key. Furthermore, even though the attacker may use the MAC key to find the collision of the password having the same MAC tag, but because MAC key and the salt for the database key generation are different, the attacker could not use the collision password (but different to the database password) to decrypt the database correctly.

Figure 2: Verify the database password

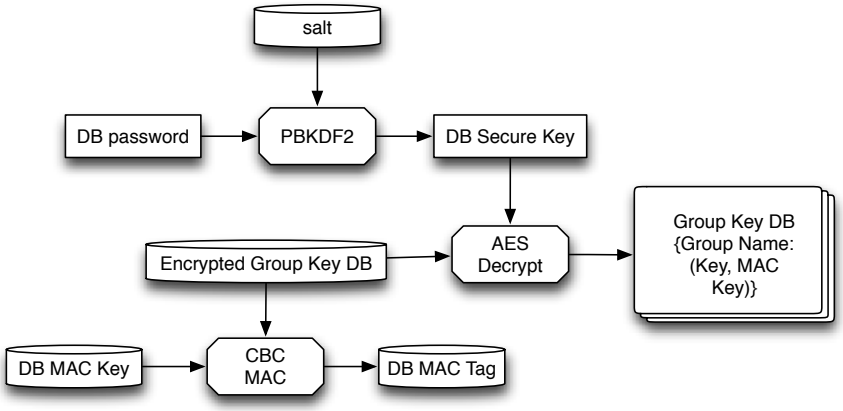


2.4 Maintaining a secure database for the group keys

We used the encrypted database keys to encrypt and decrypt groups keys on the fly without storing the encrypted database key in localStorage, which is shown in figure 3. We encrypt and decrypt the group keys in nonce-based counter mode with AES. In this way the attacker has no way to find out what the encrypted database key is given the cipher text if the block cipher under counter mode

is secure. In addition, to prevent to leak the group name information, we encrypt and decrypt the whole JSON string of the hash table of all group keys. For integrity, we use the same MAC technology in section 2.3 with a seperate key to ensure the integrity of the data for making sure that the store haven't been changed between program runs.

Figure 3: Maintain the database of group keys



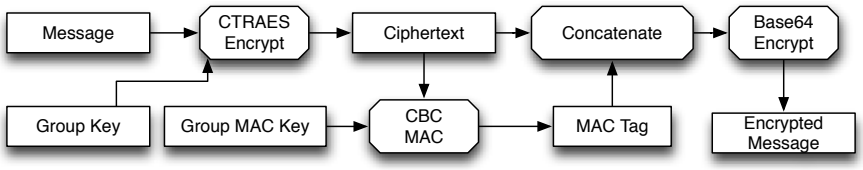
2.5 The block cipher for group message

We implemented nonce-based counter mode (CTR) using AES to encrypt and decrypt group messages using the group keys. For encryption, we first generate a random nonce IV, then we derive the ciphertext of i-th block by $c_i = AES(k, IV \oplus i) \oplus m_i$ where m_i is the i-th block of plaintext, and finally we append the nonce in the end of the ciphertext. For decryption, we first pick the nonce out from the end of the message, then we derive the plaintext of i-th block by $m_i = AES^{-1}(k, IV \oplus i) \oplus c_i$.

For integrity, we use the MAC with a seperate key to ensure the integrity of the data. We send the MAC tag with the ciphertext together. The workflow to verify the messages by MAC is shown in figure 4 and 5.

By Counter-mode Theorem, since AES is a secure PRP over (K, X, X), CTR has semantic security under q queries of CPA over (K, X^L, X^{L+1}) as long as $q^2 L \ll |X|$. So such mode is proved to be secure.

Figure 4: Encryption of group message



2.6 Block padding

Our padding strategy is that we append a bit of 1 immediately after the text and then pads bits of 0 until the length of the text is a multiple of the length of a block. The figure 6 shows how we implement the block padding.

Figure 5: Decryption of group message

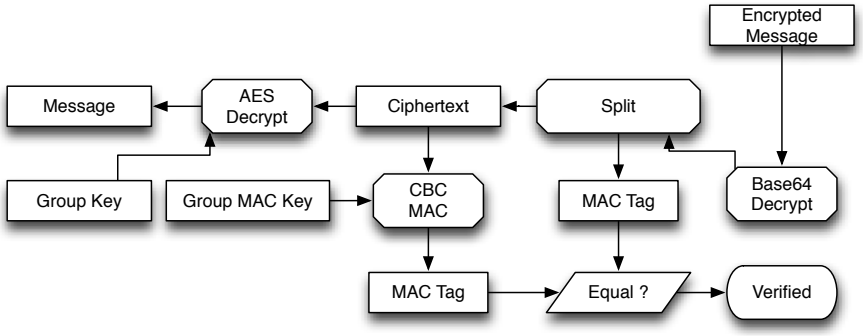
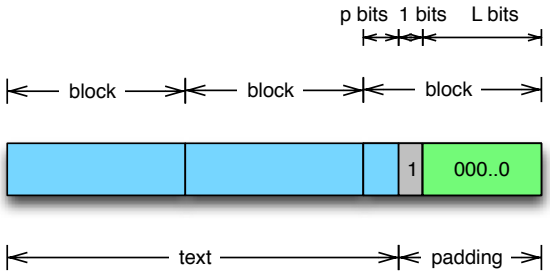


Figure 6: Padding a message



2.7 MAC system

We implemented CBC-MAC system using AES. We first let $c = 0$. In the order of blocks in the message, we iterate and generate $c \leftarrow AES(k_1, m_i \oplus c)$ where m_i is i-th block of the message.

To prevent the attacker to append new blocks to the ciphertext, CBC-MAC requires another 256-bit key k_2 to encrypt the last block. Namely, finally, we output $c' \leftarrow AES(k_2, c)$. Thus, we need to store two 256-bit keys as an 512-bit key in the storage. By CBC-MAC Theorem, CBC-MAC under q queries of CPA is secure as long as $q^2 \ll |X|$.

We use Encrypt-then-MAC to append the MAC tag to our ciphertext, i.e., we append the MAC tag, which calculated from the encrypted ciphertext, to the ciphertext. The MAC does not provide any information on the plaintext. It also provides integrity of ciphertext.

2.8 Generating keys

The `Math.random()` function from javascript is not random. So we use the `crypto` library from Chrome to generate random keys. In `GenerateKey`, we generate a random key of 256+512 bits. The first 256 bits are used as the key for the group message encryption, and the last 512 bits are used as the two 256-bit keys for MAC of the message encryption.

3 Security

3.1 What would happen if the attacker has access to localStorage and other locally stored materials?

The assumption here is that the attacker has access to all locally stored materials after the user closes the session (browser tab). Namely, the locally stored information includes the encrypted database of group keys, the salt of the database password, the MAC key, and the MAC tag for the encrypted database password. The attacker can not get significant information about the group keys because the encrypted database key is not stored locally, and he can not recover the encrypted database key from a secure MAC function without exhaustive search. He's not able to understand the encrypted database without knowing the encrypted database key since the nonce-based counter mode is secure under CPA attack.

3.2 How could somebody go about circumventing the security of your implementation, if they really wanted to?

The attacker could use side channel attacks on the computer system to monitor the timing and energy flow on chip of decryption or encryption of the messages to find out the keys. The attackers can also potentially implant viruses or use web browser security holes to steal sessionStorage information from the user or even listen to key strokes on the system level. The attacker can also forge a insecure web browser with their own implementation of web standards and javascript runtime for the user to download and use. Insecurely executed javascript on the browser or injected scripts from untrusted third party would also lead to trivial attacks. Finally, the attacker can deploy generic attack so that he can use the cipher to process huge amount of plaintext until the key is no longer secure (need renew key at this point).

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

The biggest issues might be that the designer of the `crypto` program and the web system using the `crypto` library on the client has no control on the user behavior and other programs running on the client. A mistakenly implemented web browser might leak session storage; A malicious piece of software can monitor system memory to sniff data loads and store; A website can forge the exact same look and feel to trick the user to use other systems unintentionally; A browser plugin can inject unsecure javascript to the browser, etc.

We might still want to do it anyhow because `crypo` on the client saves the computational resources on the server side of the system. It also brings the benefit of separating modules from larger systems to maintain less layer of server logic and better isolation of code logic. In addition, when the server end is compromised, the service does not lose important user information if sensitive encrypted keys are stored on the client locally.

4 Instruction to using the system

4.1 Running Program

We created our own UI imitating the original facebook style. We designed the UI to follow the exact requirement of this assignment. Namely, when user logs in, she has to setup a database password if she hasn't done so. Once the password is setup, she has to enter the password to proceed to

view the page on each new browser tab. If the password is wrong, she won't be able to continue and the UI reflects the wrong password. If no key is found in a group, we show a message box to indicate the user to create one. We didn't change the workflow of group key creation. So the user, say A, could generate a new group key in account setting page. If the other user B want to see the encrypted message sent by A, she has to add the same group key of A in B's account setting page.

Because we can not modify the UI logic of the starter code in the submission, there is a small issue of user experience, i.e., if you enter the group page directly by url, we will prompt you to enter your password; and after that, you will not immediately see the decrypted messages. You have to refresh the page for activating the decryption of the message with your just entered password.

4.2 Code

We didn't change `menifest.json` except modifying the javascript file name and adding our names to the description.

We didn't make any change to the code below `GetRandomValues()` and we implemented the required functions (`Encrypt`, `Decrypt`, `GenerateKey`, `SaveKeys`, `LoadKeys`) in the assignment. We added several helper functions under `LoadKeys()`, their main purposes are building UI (`BuildUIBox`), initiating and querying database password (`GetDBSecurePassword`), calculate padding (`Unpadding`, `Padding`), calculating MAC (`CBCMAC`) and implement the nounce-based counter mode (`CTRAESDecript`, `CTRAESEncript`). It's easy to see the documentation in the code to learn what they do.

4.3 Comment

Because localStorage is not shared over two protocols http and https even under the same domain "facebook.com". The starter code implements a localStorage that actually stores the data into the shared persistent chrome.storage with caching in the localStorage of the browsing page. But it does not implement the data synchronization between caches of different protocols. Thus, after modifying the group key in the account setting page (https), we cannot immediately get the group key in the group page (http), unless refreshing the tab. So, in our implementaion, we force to redirect user to the https page.