Report: Project 1

Riwaaz Ranabhat, Elias El Bouzidi
University of Antwerp

November 3, 2024

1 Symetric Encryption

1.1 AES-256-CBC Implementation

1.1.1 Encryption implementation

We began by defining the Initialization Vector (IV). Since AES-CBC mode requires a 16-byte IV, we got this by slicing the provided nonce to use only the first 16 bytes.
Next was the key cause it needed to be properly formatted, so we used a hash function to generate a consistent 256-bit (32-byte) key from our provided hashkey function generated by SHA256. With the hashed key and IV initialized, we then created an AES object using the library's functionality to encrypt the plaintext, resulting in the desired ciphertext.

1.1.2 Decryption implementation

The decryption process follows similar steps to encryption where we initialize the AES object with the same hashed key and truncated nonce. We then decrypt the ciphertext and remove the padding added during encryption to retrieve the original plaintext.

1.2 Salsa20

1.2.1 Encrypt Implementation

This method was likely the most challenging to implement.
At first, we relied on the theoretical concepts presented in the lecture slides, but some parts weren't fully explained, so we faced difficulties in certain areas.
We searched online for additional test cases and found an official Salsa20 documentation, which provided detailed specifications and test cases relevant to our setup.
Initially, we encountered an issue where we mistakenly padded the key instead of hashing it, as we had done with AES implementation. After correcting this by applying the SHA512 hash function, the remaining steps involved setting up the initialization matrix. This required truncating the necessary bytes for each part: the key, nonce, position, and constants, and then executing the doubleround function to produce the pre-final matrix. All of these operations are done in smaller functions. Herefore we recommend you guys to see the salsa.py file.
Once we obtained the resulting matrix, we proceeded with an XOR operation against the original matrix to generate the ciphertext. This process continued iteratively until the entire content was encrypted.

1.2.2 Decrypt Implementation

The lecture slides made us clear that decryption could be simplified by using word-by-word summation with the initial matrix due to the invertibility of the round function f. This meant that decryption essentially mirrored the encryption process.
We simply called the encrypt function again, this time with the ciphertext, key, and nonce as inputs. Since the function's operations are symmetric, re-running the encryption process with the same parameters reverses the encryption, allowing us to retrieve the original plaintext.

1.3 Encrypting HTTP traffic

1.3.1 Implementation details

The main goal here was to securely encrypt the content exchanged between the client and the Flaskr server. Initially, figuring out how to approach this encryption was challenging, as it required the knowledge of how the headers had to be handled cause nothing was clear.

1.3.2 How we apply our algorithms

On the client side, we first read the client configuration to extract essential information. Between the client and the server there is a preshared-key that is being used. We will only encrypt if and only if there is content to be encrypted. For this, we generate random-nonce for uniqueness.This will be given to our encryption function (aes or salsa20) to generate the ciphertext. As last, we make our encryption header with all the necessary information that is needed at the server side for decryption. There we take the encryption header if it is present and then take the information out and then decrypt using the correct method. Initially, we implemented this only for AES, as we were uncertain about multi-encryption support and thought supporting Salsa20 might imply supporting multiple encryption methods. After clarification in lab sessions, we extended encryption to include Salsa20.
This setup directly respects the confidentiality aspect of the CIA-Trad. In the event of a man-in-the-middle attack, any intercepted data remains unreadable to an attacker, as the content between the client and the Flaskr server is securely encrypted.
Both the client and Flaskr server are the only entities that can access this data. This is due to the decryption method, which runs on both the client's response side and the server's request side, ensuring secure, end-to-end encrypted communication.

2 Cryptographic Hash Functions and Message Integrity

2.1 SHA-1 implementation

2.1.1 Implementation Details

Implementing SHA-1 was relatively easy because we could use the provided mac.py file, where the Hash class with the SHA-1 setup was already available.
The first step was to pad the message. We began by converting the message to binary format and padding it until its length was exactly 64 bits less than a multiple of 512.

Next, we appended the original message length as a 64-bit binary number. This step adds complexity and helps secure the integrity of the message.
Finally, we split the message into 512-bit blocks. Each block was processed to calculate intermediate hash values (h0 to h4).

2.2 HMAC

2.2.1 Implementation Details

In order to understand the HMAC-algorithm, we referred to a detailed video tutorial, which we've linked in the references cause we were a bit confused by how it was written on our theory slides. Our implementation follows the approach outlined there.
The process begins with initializing the key to match the block size of the hash function. This requires the key to be exactly b bits as mentioned in the video. If the key is too long, we hash it to reduce its length, and if it's too short, we pad it with zeros on the right. A similar operation is performed on the nonce, though we only pad it as needed without hashing.
Next, we generate the inner and outer padding values, referred to as ipad and opad. These paddings are defined as specific byte values (0x36 for ipad and 0x5c for opad) and are XORed with the key to create modified keys (k+ $\oplus$ ipad and k+ $\oplus$ opad).
Finally, we compute the HMAC by first hashing the concatenation of ipad and the message, which produces an intermediate hash (s1). Then, we concatenate opad with s1 and hash this result to obtain the final HMAC value.

2.3 Authorizing HTTP traffic

2.3.1 Implementation Details

To achieve the integrity of the traffic between the client and the flaskr server we use the algorithms sha1 or hmac to authenticate the message, to make sure nobody tampered with the message. But unlike with the encryption and decryption the headers and their content will not be ignored. So when we send a request message from the client to the server and we use for example sha1, we authenticate first and then encrypt. So we take all the request headers and also the content if there is content. We then put them in a string and we use this string to create a mac using sha1. Now we put this mac inside the Authorize header so that when we get the message on the server side we first have to decrypt the message first and then we authorize. Now we can take the mac out of the authorize header and we then do the same thing as before, we take all the headers in the request and content too if there is and we put it in a string. We then use this string to create a mac using the same algorithm, nonce and key as before. We then compare the two macs to see if they are the same. If they are not the same, this means the message has been changed. This could be in the headers or in the content so we then reject the authorization. This helps against a Man-in-the-Middle attack that intercepts and possibly alters the message as it travels between the sender and the receiver.

2.4 Testing of Algorithms

To keep the implementation folder organized, we created a tests directory in the implementation folder where we use Python unit tests to verify the functionality of each algorithm.

References

[1] HMAC-tutorial
`https://www.youtube.com/watchv=sxWxCtJ3ogO&list=PLLOxZwkBK52ChOy2lLtfepy4Lt_SVkwo3&index=13`

[2] Salsa20 specs document
`https://cr.yp.to/snuffle/spec.pdf`