2024

LZSCC.363 Security and Risk Coursework 1 Code Report

38553767
TOLGA ZIYA KAVUNCU

TABLE OF CONTENTS

EXERCISE 1: HILL CIPHER IMPLEMENTATION

This task required me to create an implementation of the Hill Cipher. I used a 2x2 key matrix against modulo 26 in or to encrypt any given plaintext ASCII input. The input can be a mix of uppercase and lowercase ASCII letters, numbers, and punctuation characters. The only characters that get encrypted are the uppercase ASCII letters.

Any input that consists solely of uppercase ASCII letters, will be encrypted, and decrypted perfectly. Since the key matrix is 2x2 if the given input has an odd number of letters in it the encryption would fail. To overcome this issue my implementation is using padding for words that are odd number of letters long. And the padding letters are being removed before the deciphered text is returned to the console.

Most inputs with a mix of uppercase, lowercase, numbers, and punctuation characters can be encrypted and decrypted without any issues. The implementation can handle these inputs as well.

However, there seems to be a bug with some mixed input that get divided with one uppercase and any other character in the same block. In this case 2x2 so an input like “aweJ” which gets divided into blocks “aw” and “eJ” will not give the decryption as it should.

The encrypt() function works by padding the input with characters if they are in need of lengthening to be a multiple of the key matrix size. Then the input is broken into blocks and each block gets encrypted using matrix multiplication with the key matrix. Which is how the ciphertext is created.

The decrypt() function works by taking in the ciphertext the key matrix and the number of padding characters used. It decrypts the ciphertext by calculating the determinant of the key matrix and finding the inverse modulus, after which it multiplies the inverse matrix with the vector from the ciphertext.

EXERCISE 2: THE DIFFIE HELLMAN MITM ATTACK

This task required me to create an implementation of the Diffie Hellman key exchange protocol. I also implemented two versions of the Man-in-the-Middle attack for which I provided two different test case python files. The power of the attacker is implemented in the normal version of the code itself as well. However, I didn’t think it was clear enough to display the attacks power using just one script. That is the reason I provided 3 different python files for this exercise.

The Diffie_Hellman() function creates and prints the keys in PEM format, and also calculates the derived keys and prints those as well.

In the first test case in which the attacker is eavesdropping on the conversation between the two other users. The attacker is in between the User1 and User2. Using the position they have gained between the users. The attacker can get the encrypted message from User1 and decrypt it using the key that they got from them. This allows the attacker to just read the encrypted message as plaintext since they have access to the keys of both users that are supposed to be using the encryption. After reading and having access to the transmitted message the attacker encrypts the message again without doing any modifications to the message and send it off to User2. This allows the attacker to be unseen by the Users and they can keep eavesdropping the conversation.

The second test case in which the attacker is modifying the message works in a very similar way. However, the main difference being instead of just encrypting the same message and sending that to User2. The attacker manipulates the message in a way that the new modified message is made up by the attacker. This allows the attacker to completely take over the conversation between User1 and User2.

EXERCISE 3: ELLIPTIC CURVE SIGNATURE ALGORITHM

This task required me to create a ECDSA key pair and sign a message given by the user using the private key, and then verify the signature using the accompanying public key.

The generate_key_pair() function just creates the private-public key pairs using the elliptic curve. And returns the keys.

The hash_message() function hashes the inputted message using SHA-256 algorithm and returns the digest.

The sign_message() function signs the message using the private key generated using a the hashed version of the message and returns the signature of the message.

The verify_signature() function is where the message is verified using the corresponding public key for the private key signed the message in the first place. If the keys match the function returns true, else it returns false.

There is not much to talk about regarding this exercise as the reading provided is more than enough to do the required task.

README.MD

The README.md file I provided is where the instructions on how to run the code and get the dependencies for the whole project is explained in detail.

The project is also on my GitHub; however, I will keep the repository private until the deadline passes. If you would like access to the repository, please let me know.