LIGHT WEIGHT CRYPTOGRAPHIC ALGORITHM

Key Generation Algorithm:

Here's the key generation algorithm broken down into following steps:

Step 1: Initialize Variables

- Create a variable to store the user-provided password.
- Create a variable to store the 16-byte random salt.
- Create a variable to store the number of iterations for PBKDF2.
- Create a variable to store the desired derived key length (16 bytes for a 128-bit key).
- Create an empty list to store the generated sub-keys.

Step 2: Input Password

- Prompt the user to enter a password.
- Store the entered password in the designated password variable.

Step 3: Generate Random Salt

- Use a cryptographically secure random number generator (e.g., os.urandom(16)) to generate a 16-byte random salt.
- Store the generated salt in the designated salt variable.

Step 4: Derive Master Key with PBKDF2

- Call the PBKDF2-HMAC-SHA256 key derivation function with the following arguments:
- Password: The user-provided password.
- Salt: The randomly generated salt.
- Iterations: The predefined number of iterations.
- dkLen: The desired derived key length (16 bytes for a 128-bit key).
- Store the output of PBKDF2 (the derived key) in the master key variable.

Step 5: Loop for Sub-key Generation (8 iterations)

- For each round i from 0 to 7, perform the following steps:
- Convert Round Number to String: Convert the integer i to its string representation.
- Hash Round Number: Hash the string representation of i using SHA-256.
- Extract Round Constant: Take the first 16 bytes of the SHA-256 hash output as the round constant for this round.
- Create Temporary Key Variable: Create a new variable to store intermediate results during sub-key generation.
- XOR with Round Constant: Perform byte-wise XOR operation between the master key and the round constant. Store the result in the temporary key variable.
- Rotate Left: Rotate the bytes in the temporary key variable left by 8 positions.

Substitute Bytes:

- Create a variable to store the S-box values.
- For each byte in the temporary key variable:
- Calculate the index for the S-box by taking the byte value modulo 16.
- Substitute the byte with the corresponding value from the S-box at the calculated index.

- Store the substituted bytes back in the temporary key variable.
- Append to Sub-key List: Add the final 16-byte sequence in the temporary key variable to the list of sub-keys.

Step 6: Output Sub-keys

- Return the list containing the 5 generated 128-bit sub-keys.

Encryption Algorithm:

Here's the algorithm for the encryption process, including all the details and steps:

Step-1: Input

- data: The data to be encrypted (as bytes).
- key_filename: The name of the file containing the master key and sub-keys.

Step-2: Initialization:

- Define the block size (e.g., 16 bytes for a 128-bit block size).
- Create an empty byte string variable to accumulate the ciphertext.

Step-3: Padding:

- Calculate the number of padding bytes required to make the data length a multiple of the block size.
- Append the padding bytes (each with the value equal to the padding length) to the end of the data.

Step-4: Import Sub-keys:

- Call the import_sub_keys function with the key_filename as input. This function reads the sub-keys from the key file and returns them as a list of bytes objects.
- Store the returned list of sub-keys in a variable.

Step-5: Process Data Blocks (Loop):

- Iterate through the data, processing one block (16 bytes) at a time.
- For each block:
    - Split Block: Divide the block into two halves, left and right, each 8 bytes long.
    - Encryption Rounds (Loop 8 times):
    - For each round i from 0 to 7:
    - Round Function:
    - Pass the right half of the block and the i+1th sub-key to the round_function.
    - The round_function performs the following:
    - Substitution: Substitute each byte using a predefined S-box.
    - Rotation: Rotate the bytes left by 2 positions.
    - XOR with Sub-key: Perform byte-wise XOR with the sub-key.
    - Store the output of the round_function in a temporary variable.
    - XOR with Left Half: Perform byte-wise XOR between the left half of the block and the output from the round_function. Store the result in a temporary variable.
    - Swap Halves: Update the left and right halves for the next round: left becomes the previous right, and right becomes the result of the XOR operation.
- Combine Halves: Concatenate the final left and right halves to form the encrypted block.
- Append to Ciphertext: Append the encrypted block to the ciphertext variable.

Step-6: Master Key XOR:

- Extract the first sub-key from the list of sub-keys (assuming it's the master key).
- Perform byte-wise XOR between the ciphertext and the master key for additional security.
- Update the ciphertext variable with the result.
- Generate Ciphertext Filename:
- Extract the timestamp from the key_filename.

- Create a filename for the ciphertext file using the format: "ciphertext_" + timestamp + "_" + key_filename.

Step-7: Encode Ciphertext with Base64:

- Encode the ciphertext using Base64 encoding.
- Decode the Base64 encoded bytes into a string for easier storage and printing.

Step-8: Save Ciphertext to File:

- Open the ciphertext file in write mode.
- Write the Base64 encoded ciphertext string to the file.
- Close the file.

Step-9: Output:

- Print a message indicating the filename where the ciphertext was saved.
- Print the Base64 encoded ciphertext string to the console.

Decryption Algorithm:

Here's the decryption algorithm broken down into following steps:

Step-1: Read and Decode Ciphertext:

- Open the ciphertext file in read mode.
- Read the Base64 encoded ciphertext from the file.
- Decode the Base64 ciphertext to obtain the original binary ciphertext.

Step-2: Import Keys:

- Open the key file in read mode.
- Read the master key (first line) and sub-keys (remaining lines) from the file.
- Extract the hexadecimal key values and convert them to bytes objects.

Step-3: Prepare Sub-keys for Decryption:

- Reverse the order of the sub-keys, excluding the master key. This is necessary because the decryption process reverses the encryption steps.
- Decrypt Ciphertext (Block-by-Block):
- Iterate through the ciphertext, processing it block by block (16 bytes per block).
- For each block:
- Split Block: Divide the 16-byte block into two 8-byte halves.
- Reverse Master Key XOR: XOR the current block with the master key to reverse the initial XOR operation done during encryption.

Step-4: Eight Rounds of Decryption:

- Iterate 8 times, performing the following steps in each round:
- Inverse Round Function: Apply the inverse round function to the first 8-byte half-block using the corresponding sub-key (starting from the last sub-key and moving towards the first).
- XOR with Second Half: XOR the output of the inverse round function with the second 8-byte half-block.
- Swap Halves: Swap the two half-blocks to prepare for the next round.
- Combine Halves: Concatenate the final two half-blocks to form the decrypted block.
- Append to Plaintext: Append the decrypted block to the growing plaintext byte string.

Step-5: Remove Padding:

- Extract the last byte of the plaintext. This byte represents the padding length added during encryption.
- Validate that the padding length is within a valid range (1 to block size) and that the padding bytes have the expected value (the padding length itself).
- If the padding is valid, remove the padding bytes from the end of the plaintext. Otherwise, indicate an error.

Step-6: Decode Plaintext:

- Decode the plaintext byte string using the appropriate character encoding (e.g., UTF-8) to obtain the original plaintext as a string.

Step-7: Output:

- Print the decrypted plaintext to the console.