```
# 2024 Â© Idan Hazay encrypting.py
# Import libraries

from modules.config import *
import os, rsa, struct

# Key exchange
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
from base64 import b64encode, b64decode



class Encryption:
    def __init__(self, network):
        self.network = network
        self.block_size = AES.block_size

    def encrypt(self, plain_text, key):
        """
        Encryption function
        Adds necessary padding to match block size
        """
        key = hashlib.sha256(key).digest()
        plain_text = self.pad(plain_text)
        iv = Random.new().read(self.block_size)
        cipher = AES.new(key, AES.MODE_CBC, iv)
        encrypted_text = cipher.encrypt(plain_text)
        return b64encode(iv + encrypted_text)

    def decrypt(self, encrypted_text, key):
        """
        Decryption function
        Remove added padding to match block size
        """
        key = hashlib.sha256(key).digest()
        encrypted_text = b64decode(encrypted_text)
        iv = encrypted_text[:self.block_size]
        cipher = AES.new(key, AES.MODE_CBC, iv)
        plain_text = cipher.decrypt(encrypted_text[self.block_size:])
        return self.unpad(plain_text)

    def pad(self, plain_text):
        """
        Adds padding to test to match AES block size
        """
        number_of_bytes_to_pad = self.block_size - len(plain_text) % self.block_size
        ascii_string = chr(number_of_bytes_to_pad)
        padding_str = number_of_bytes_to_pad * ascii_string
        padded_plain_text = plain_text + padding_str.encode()
        return padded_plain_text


    def unpad(self, plain_text):
        """
        Removes padding to test to match AES block size
        """
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]


    def rsa_exchange(self):
        try:
            self.network.send_data_wrap(b"RSAR", False)
            s_public_key = self.recv_rsa_key()
            shared_secret = self.send_shared_secret(s_public_key)
            return shared_secret
        except:
            print(traceback.format_exc())


    def recv_rsa_key(self):
        """
        RSA key recieve from server
        Gets the length of the key in binary
        Gets the useable key and saves it as global var for future use
        """
        key_len_b = b""
        while (len(key_len_b) < LEN_FIELD):   # Recieve the length of the key
            key_len_b += self.network.sock.recv(LEN_FIELD - len(key_len_b))
        key_len = int(struct.unpack("!l", key_len_b)[0])

        key_binary = b""
        while (len(key_binary) < key_len):   # Recieve the key according to its length
            key_binary += self.network.sock.recv(key_len - len(key_binary))
    s_public_key = rsa.PublicKey.load_pkcs1(key_binary)   # Save the key
    return s_public_key


def send_shared_secret(self, s_public_key):
    """
    Create and send the shared secret
    to server via secure rsa connection
    """
    shared_secret = os.urandom(16)
    key_to_send = rsa.encrypt(shared_secret, s_public_key)
    key_len = struct.pack("!l", len(key_to_send))
    to_send = key_len + key_to_send
    self.network.sock.send(to_send)
    return shared_secret
```