```
# 2024 Â© Idan Hazay

# Import required libraries
import hashlib, os, rsa, struct
from modules.config import *
from Crypto import Random
from Crypto.Cipher import AES
from base64 import b64encode, b64decode

class Encryption:
    """
    Provides encryption and decryption methods using AES and RSA.
    """
    def __init__(self):
        self.block_size = AES.block_size  # Block size for AES encryption

    def encrypt(self, plain_text, key):
        """
        Encrypts a plaintext string using AES encryption.
        Pads the plaintext to match the block size before encryption.
        """
        key = hashlib.sha256(key).digest()  # Derive a fixed-length key using SHA-256
        plain_text = self.pad(plain_text)  # Pad the plaintext
        iv = Random.new().read(self.block_size)  # Generate a random IV
        cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES cipher in CBC mode
        encrypted_text = cipher.encrypt(plain_text)  # Encrypt the plaintext
        return b64encode(iv + encrypted_text)  # Return encoded ciphertext with IV prepended

    def decrypt(self, encrypted_text, key):
        """
        Decrypts a ciphertext string using AES decryption.
        Removes padding after decryption.
        """
        key = hashlib.sha256(key).digest()  # Derive a fixed-length key using SHA-256
        encrypted_text = b64decode(encrypted_text)  # Decode the base64 ciphertext
        iv = encrypted_text[:self.block_size]  # Extract the IV
        cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES cipher in CBC mode
        plain_text = cipher.decrypt(encrypted_text[self.block_size:])  # Decrypt the ciphertext
        return self.unpad(plain_text)  # Remove padding

    def pad(self, plain_text):
        """
        Pads the plaintext to make its length a multiple of the 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
        return plain_text + padding_str.encode()  # Append padding

    def unpad(self, plain_text):
        """
        Removes padding from the plaintext.
        """
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]  # Remove padding

    def create_keys(self):
        """
        Generate RSA public and private keys.
        Save the keys to files for reuse.
        """
        self.public_key, self.private_key = rsa.newkeys(1024)  # Generate RSA keys
        if not os.path.isfile(f"{os.getcwd()}/keys/public.pem"):
            with open(f"{os.getcwd()}/keys/public.pem", "wb") as f:
                f.write(self.public_key.save_pkcs1("PEM"))  # Save public key
        if not os.path.isfile(f"{os.getcwd()}/keys/private.pem"):
            with open(f"{os.getcwd()}/keys/private.pem", "wb") as f:
                f.write(self.private_key.save_pkcs1("PEM"))  # Save private key

    def load_keys(self):
        """
        Load RSA public and private keys from files.
        """
        with open(f"{os.getcwd()}/keys/public.pem", "rb") as f:
            self.public_key = rsa.PublicKey.load_pkcs1(f.read())
        with open(f"{os.getcwd()}/keys/private.pem", "rb") as f:
            self.private_key = rsa.PrivateKey.load_pkcs1(f.read())

    def send_rsa_key(self, sock, tid):
        """
        Send the RSA public key to a client.
        """
        key_to_send = self.public_key.save_pkcs1()  # Serialize public key
        key_len = struct.pack("!l", len(key_to_send))  # Pack key length
        sock.send(key_len + key_to_send)  # Send key length and serialized key

    def recv_shared_secret(self, sock, tid):
```

```
    """
    Receive and decrypt a shared secret from a client.
    """
    key_len_b = b""
    while len(key_len_b) < LEN_FIELD:  # Receive key length
        key_len_b += 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:  # Receive the key
        key_binary += sock.recv(key_len - len(key_binary))
    return rsa.decrypt(key_binary, self.private_key)  # Decrypt the shared secret

def rsa_exchange(self, sock, tid):
    """
    Perform an RSA key exchange by sending the public key and receiving a shared secret.
    """
    self.send_rsa_key(sock, tid)  # Send the public key
    return self.recv_shared_secret(sock, tid)  # Receive and return the shared secret
```