To implement the SSL handshake and key establishment using either Diffie-Hellman (DHE) or RSA, you'll need to follow these steps:

1. **Negotiate Cipher Suites and Key Exchange Algorithm**: During the handshake, the client and server negotiate the cipher suite and key exchange algorithm they will use for the secure connection. This includes agreeing on whether to use DHE or RSA for key exchange.
2. **Perform Key Exchange**: Depending on the negotiated parameters during the handshake, perform the key exchange using either DHE or RSA.
3. **Generate Shared Key**: After the key exchange, generate or derive a shared secret key based on the exchanged keys.
4. **Save Shared Key**: Save the shared key and its length within the SSL context for later use in encryption and decryption.
5. **Return Success or Failure**: Return success (0) if the handshake and key establishment are successful, or return an error code (-1) otherwise.

Here's a guideline for implementing these steps in **SslClient::connect()** and **SslServer::accept()**:

**In SslClient::connect(ip, port, mode)**:

1. Establish TCP connection with the server.
2. Perform handshake initialization (negotiate cipher suites, protocols, etc.).
3. Depending on the mode (DHE or RSA), perform the key exchange.
4. Generate or derive a shared secret key based on the exchanged keys.
5. Save the shared key and its length within the SSL context.
6. Return success (0) if everything is successful, or an error code (-1) otherwise.

**In SslServer::accept()**:

1. Wait for the client's TCP connection.
2. Perform handshake initialization (negotiate cipher suites, protocols, etc.).
3. Depending on the negotiated parameters, perform the key exchange.
4. Generate or derive a shared secret key based on the exchanged keys.
5. Save the shared key and its length within the SSL context.
6. Return a new SSL instance (or SSL context) on success, or NULL otherwise.

You'll need to implement the specific details of the handshake and key exchange for DHE and RSA based on the provided framework and cryptographic primitives (such as those provided in the **cryptopp** library). Additionally, ensure that the shared key and its length are properly managed and accessible for encryption and decryption operations in both client and server implementations.

**SKELETON:**

#include "ssl\_client.h"

#include "stdlib.h"

#include "string.h"

#include <iostream>

#include "dh.h"

#include "integer.h"

#include "osrng.h"

#include "tcp.h"

#include "crypto\_adaptor.h"

#include "logger.h"

#include "utils.h"

using namespace std;

SslClient::SslClient() {

string datetime;

if ( get\_datetime(&datetime, "%Y%m%d-%H%M%S") != 0 ) {

exit(1);

}

this->logger\_ = new Logger(("ssl\_client\_"+datetime+".log"));

this->tcp\_->set\_logger(this->logger\_);

get\_datetime(&datetime, "%Y/%m/%d %H:%M:%S");

this->logger\_->log("Client Log at " + datetime);

}

SslClient::~SslClient() {

if ( this->logger\_ ) {

delete this->logger\_;

this->logger\_ = NULL;

this->tcp\_->set\_logger(NULL);

}

}

int SslClient::connect(const std::string &ip, int port, uint16\_t cxntype) {

// Step 1: Establish TCP connection with the server

if ( this->tcp\_->socket\_connect(ip, port) != 0 ) {

cerr << "Couldn't connect" << endl;

return -1;

}

// Step 2: Perform handshake initialization (negotiate cipher suites, protocols, etc.)

if (!perform\_handshake\_initialization(cxntype)) {

cerr << "Handshake initialization failed" << endl;

return -1;

}

// Step 3: Depending on the mode (DHE or RSA), perform the key exchange

if (cxntype == DHE\_CONNECTION) {

if (!perform\_dhe\_key\_exchange()) {

cerr << "DHE key exchange failed" << endl;

return -1;

}

} else if (cxntype == RSA\_CONNECTION) {

if (!perform\_rsa\_key\_exchange()) {

cerr << "RSA key exchange failed" << endl;

return -1;

}

} else {

cerr << "Invalid connection mode" << endl;

return -1;

}

// Step 4: Generate or derive a shared secret key based on the exchanged keys

generate\_shared\_secret();

// Step 5: Save the shared key and its length within the SSL context

save\_shared\_key();

// Step 6: Return success (0) if everything is successful

return 0;

}

int SslClient::close() {

int ret\_code;

ret\_code = this->tcp\_->socket\_close();

return ret\_code;

}

bool SslClient::perform\_handshake\_initialization(uint16\_t cxntype) {

// Placeholder for handshake initialization

// For DHE and AES-CBC, we might specify specific cipher suites and protocols

// For example, in TLS, you might negotiate the TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA cipher suite

// For simplicity, let's assume we're using TLS and specify the cipher suite and protocol

// Note: This is a simplified example and might not cover all negotiation scenarios

// Check if the desired connection type is DHE

if (cxntype == DHE\_CONNECTION) {

// Negotiate cipher suites and protocols for DHE

// For example, specify the TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA cipher suite

// and negotiate the TLS protocol version

// Set the negotiated cipher suite and protocol

this->cipher\_suite\_ = "TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA";

this->protocol\_version\_ = "TLSv1.2";

// Log the negotiated parameters

cout << "Negotiated Cipher Suite: " << this->cipher\_suite\_ << endl;

cout << "Negotiated Protocol Version: " << this->protocol\_version\_ << endl;

// Return success

return true;

}

// If cxntype is not DHE or other supported connection types, return false

cerr << "Unsupported connection type" << endl;

return false;

}

bool SslClient::perform\_dhe\_key\_exchange() {

// Placeholder for DHE key exchange

return true; // Placeholder for success

}

bool SslClient::perform\_rsa\_key\_exchange() {

// Placeholder for RSA key exchange

return true; // Placeholder for success

}

void SslClient::generate\_shared\_secret() {

// Placeholder for generating shared secret

}

void SslClient::save\_shared\_key() {

// Placeholder for saving shared key

}