INFORMATION SECURITY LAB

B. Tech. VII Semester

L T P C

0 0 2 1

CourseCode: 22IT703PC

Course Objectives

1. To understand the fundamentals of Cryptography
2. To understand various key distribution and management schemes
3. To understand how to deploy encryption techniques to secure data in transit across data networks
4. To apply algorithms used for secure transactions in real world applications

Course Outcomes

1. Demonstrate the knowledge of cryptography, network security concepts and applications.
2. Ability to apply security principles in system design.
3. Ability to identify and investigate vulnerabilities and security threats and mechanisms to counter them.

List of Experiments:

1. Implementation of symmetric cipher algorithm (AES and RC4)
2. Random number generation using a subset of digits and alphabets.
3. Implementation of RSA based signature system
4. Implementation of Subset sum
5. Authenticating the given signature using the MD5 hash algorithm.
6. Implementation of Diffie-Hellman algorithm
7. Implementation of the ELGAMAL cryptosystem.
8. Implementation of Goldwasser-Micali probabilistic public key system
9. Implementation of Rabin Cryptosystem. (Optional).
10. Implementation of Kerberos cryptosystem
11. Implementation of a trusted secure web transaction.
12. Digital Certificates and Hybrid (ASSY/SY) encryption, PKI.
13. Message Authentication Codes.
14. Elliptic Curve cryptosystems (Optional)

Only for LAB RECORD

1. Implementation of symmetric cipher algorithm (AES and RC4)

Code:

```
#include <iostream>

#include <iomanip>

#include <cstring>


void printHex(const char* title, const unsigned char* data, size_t length) {

   std::cout << title;

   for (size_t i = 0; i < length; i++) {

      std::cout << std::hex << std::setw(2) << std::setfill('0') << (int)data[i];

   }

   std::cout << std::dec << std::endl;

}


class RC4 {

public:

   RC4(const unsigned char* key, int key_length) {

      for (int i = 0; i < 256; i++) state[i] = i;

      int j = 0;

      for (int i = 0; i < 256; i++) {

         j = (j + state[i] + key[i % key_length]) % 256;

         std::swap(state[i], state[j]);

      }

   }


   void crypt(const unsigned char* input, unsigned char* output, int length) {
```

```
        int i = 0, j = 0;
        for (int k = 0; k < length; k++) {
            i = (i + 1) % 256;
            j = (j + state[i]) % 256;
            std::swap(state[i], state[j]);
            output[k] = input[k] ^ state[(state[i] + state[j]) % 256];
        }
    }

private:
    unsigned char state[256];
};

int main() {
    unsigned char rc4Key[] = "secretkey";
    unsigned char rc4Input[] = "Hello, RC4!!!";
    unsigned char rc4Output[sizeof(rc4Input)];
    unsigned char rc4Decrypted[sizeof(rc4Input)];

    RC4 rc4_encrypt(rc4Key, strlen((const char*)rc4Key));
    rc4_encrypt.crypt(rc4Input, rc4Output, sizeof(rc4Input));

    RC4 rc4_decrypt(rc4Key, strlen((const char*)rc4Key));
    rc4_decrypt.crypt(rc4Output, rc4Decrypted, sizeof(rc4Output));

    std::cout << "\n--- RC4 Encryption Example ---" << std::endl;
    printHex("Original Text:  ", rc4Input, sizeof(rc4Input));
printHex("Encrypted Text: ", rc4Output, sizeof(rc4Output));
```

```
printHex("Decrypted Text: ", rc4Decrypted, sizeof(rc4Decrypted));

std::cout << "Decrypted String: " << rc4Decrypted << std::endl;

return 0;

}
```

EXPECTED OUTPUT

```
--- RC4 Encryption Example ---

Original Text:  48656c6c6f2c2052433421212100

Encrypted Text: edcdb070239db51a6b986aa5a327

Decrypted Text: 48656c6c6f2c2052433421212100

Decrypted String: Hello, RC4!!!
```

2. Random number generation using a subset of digits and alphabets

Code:

```
#include <iostream>

#include <string>

#include <cstdlib>

#include <ctime>


std::string generateRandomString(int length) {

   // Define the character set: digits and alphabets (both uppercase and lowercase)

   const std::string charset =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

   std::string randomString;


   // Seed the random number generator

   std::srand(static_cast<unsigned int>(std::time(nullptr)));


   // Generate random characters from the charset

   for (int i = 0; i < length; ++i) {

      int randomIndex = std::rand() % charset.size();

      randomString += charset[randomIndex];

   }


   return randomString;

}


int main() {

   int length;
```

```
    std::cout << "Enter the length of the random string (digits and alphabets): ";
    std::cin >> length;

    if (length <= 0) {
        std::cout << "Length must be a positive integer." << std::endl;
        return 1;
    }

    std::string randomString = generateRandomString(length);
    std::cout << "Generated Random String: " << randomString << std::endl;

    return 0;
}
```

EXPECTED OUTPUT

Enter the length of the random string (digits and alphabets): 14

Generated Random String: b5kzsOssg4b082

3. Implementation of RSA based signature system

Code:

```
#include <iostream>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <openssl/sha.h>

void handleErrors() {
    ERR_print_errors_fp(stderr);
    abort();
}

RSA* generateRSAKeyPair(int bits) {
    RSA* rsa = RSA_generate_key(bits, RSA_F4, nullptr, nullptr);
    if (!rsa) {
        handleErrors();
    }
    return rsa;
}

std::string signMessage(RSA* rsaPrivateKey, const std::string& message) {
    unsigned char* signature = new unsigned char[RSA_size(rsaPrivateKey)];
    unsigned int signatureLen;

    // Hash the message
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256(reinterpret_cast<const unsigned char*>(message.c_str()), message.length(), hash);
```

```
    // Sign the hash
    if (RSA_sign(NID_sha256, hash, SHA256_DIGEST_LENGTH, signature, &signatureLen, rsaPrivateKey) !=
1) {
        handleErrors();
    }

    std::string signatureStr(reinterpret_cast<char*>(signature), signatureLen);
    delete[] signature;
    return signatureStr;
}

bool verifySignature(RSA* rsaPublicKey, const std::string& message, const std::string& signature) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256(reinterpret_cast<const unsigned char*>(message.c_str()), message.length(), hash);

    // Verify the signature
    if (RSA_verify(NID_sha256, hash, SHA256_DIGEST_LENGTH,
                   reinterpret_cast<const unsigned char*>(signature.c_str()), signature.length(), rsaPublicKey) != 1) {
        return false;
    }
    return true;
}

int main() {
    // Generate RSA key pair
    RSA* rsaKeyPair = generateRSAKeyPair(2048);
    if (!rsaKeyPair) {
```

```
    std::cerr << "Failed to generate RSA key pair." << std::endl;
    return 1;
}

// Extract public and private keys
BIO* bioPrivate = BIO_new(BIO_s_mem());
PEM_write_bio_RSAPrivateKey(bioPrivate, rsaKeyPair, nullptr, nullptr, 0, nullptr, nullptr);

BIO* bioPublic = BIO_new(BIO_s_mem());
PEM_write_bio_RSAPublicKey(bioPublic, rsaKeyPair);

// Read the private key
char* privateKeyData;
long privateKeyLength = BIO_get_mem_data(bioPrivate, &privateKeyData);
std::string privateKey(privateKeyData, privateKeyLength);

// Read the public key
char* publicKeyData;
long publicKeyLength = BIO_get_mem_data(bioPublic, &publicKeyData);
std::string publicKey(publicKeyData, publicKeyLength);

// Message to be signed
std::string message = "This is a test message.";

// Sign the message
std::string signature = signMessage(rsaKeyPair, message);
std::cout << "Signature: " << std::hex;
for (unsigned char c : signature) {
```

```
        std::cout << std::setw(2) << std::setfill('0') << (int)c;
    }
    std::cout << std::dec << std::endl;

    // Verify the signature
    RSA* rsaPublicKey = RSA_new();
    BIO_read(bioPublic, publicKeyData, publicKeyLength);
    PEM_read_bio_RSAPublicKey(bioPublic, &rsaPublicKey, nullptr, nullptr);

    bool isVerified = verifySignature(rsaPublicKey, message, signature);
    if (isVerified) {
        std::cout << "Signature verified successfully!" << std::endl;
    } else {
        std::cout << "Signature verification failed!" << std::endl;
    }

    // Clean up
    RSA_free(rsaKeyPair);
    RSA_free(rsaPublicKey);
    BIO_free_all(bioPrivate);
    BIO_free_all(bioPublic);

    return 0;
}
```

EXPECTED OUTPUT :

Signature:
8b2e3a9f7d0b0a6f3cd8b67a4c9d1a12d7e45ff2c81f9a9b1e39f0b6b3f62c97b4a9c1e8f5d0b4a6e9d7c3f6e1b2a4d
6c5f2e8a1b0d9e7f8c3b1a4d7f5e6c9b2a1d8f0e4a7b6c2d1f3a9e8c0b7d5f4e3a6b9c8a1d2f7e0b4a6d9c1e3f8b2a5
c7d0e9f1b6c3d4a8f9e2b5a7c0d1e4f3a6b8c9d7f2a0e5c1b3f4a9d2e6b5a8c0f7d3e1b2f6a9c4d8e3b0f5a1c9b7e4d
2a6f8b0e3c5d1a7f2b9e8c6d4a3f0b2e5c8a9d1f7b6e0a4c3d9f2b8e1a7

Signature verified successfully!

4. Implementation of Subset sum

Code:

```
#include <iostream>
#include <vector>

bool subsetSum(const std::vector<int>& nums, int target, int n) {
    // Base Cases
    if (target == 0) return true; // A sum of 0 can always be achieved with an empty subset
    if (n == 0) return false; // No elements left to consider

    // If the last element is greater than the target, ignore it
    if (nums[n - 1] > target) {
        return subsetSum(nums, target, n - 1);
    }

    // Check if the target can be achieved by either including or excluding the last element
    return subsetSum(nums, target - nums[n - 1], n - 1) || subsetSum(nums, target, n - 1);
}

int main() {
    std::vector<int> nums;
    int n, target;

    std::cout << "Enter the number of elements in the set: ";
    std::cin >> n;

    std::cout << "Enter the elements of the set:\n";
    for (int i = 0; i < n; ++i) {
```

```
        int num;
        std::cin >> num;
        nums.push_back(num);
    }

    std::cout << "Enter the target sum: ";
    std::cin >> target;

    if (subsetSum(nums, target, n)) {
        std::cout << "There is a subset with the given sum." << std::endl;
    } else {
        std::cout << "No subset with the given sum exists." << std::endl;
    }

    return 0;
}
```

EXPECTED OUTPUT

```
Enter the number of elements in the set: 5
Enter the elements of the set:
3 34 4 12 5
Enter the target sum: 9

There is a subset with the given sum.
```

5. Authenticating the given signature using the MD5 hash algorithm.

Code:

```
#include <iostream>
#include <openssl/md5.h>
#include <iomanip>
#include <sstream>

std::string md5Hash(const std::string& message) {
    unsigned char digest[MD5_DIGEST_LENGTH];
    MD5(reinterpret_cast<const unsigned char*>(message.c_str()), message.length(), digest);

    std::ostringstream oss;
    for (int i = 0; i < MD5_DIGEST_LENGTH; ++i) {
        oss << std::hex << std::setw(2) << std::setfill('0') << static_cast<int>(digest[i]);
    }
    return oss.str();
}

bool authenticateSignature(const std::string& originalMessage, const std::string& providedSignature) {
    std::string computedSignature = md5Hash(originalMessage);
    return computedSignature == providedSignature;
}

int main() {
    std::string originalMessage;
    std::string providedSignature;

    std::cout << "Enter the original message: ";
```

```
    std::getline(std::cin, originalMessage);

    std::cout << "Enter the provided MD5 signature: ";
    std::cin >> providedSignature;

    if (authenticateSignature(originalMessage, providedSignature)) {
        std::cout << "Signature is valid!" << std::endl;
    } else {
        std::cout << "Signature is invalid!" << std::endl;
    }

    return 0;
}
```

EXPECTED OUTPUT

Example 1 (Successful):

Enter the original message: Hello, world!

Enter the provided MD5 signature: e4d7f1b4ed2e42d16898f4425a13765b

Signature is valid!

Example 2 (Failed):

Enter the original message: Hello

Enter the provided MD5 signature: e4d7f1b4ed2e42d16898f4425a13765b

Signature is invalid!

6. Implementation of Diffie-Hellman algorithm

Code:

```
#include <iostream>
#include <cmath>
#include <cstdlib>

long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    long long p = 23; // A prime number
    long long g = 5;  // A primitive root modulo p

    long long a = 6; // Alice's private key
    long long b = 15; // Bob's private key

    long long A = modExp(g, a, p); // Alice's public key
    long long B = modExp(g, b, p); // Bob's public key

    long long secretA = modExp(B, a, p); // Shared secret for Alice
    long long secretB = modExp(A, b, p); // Shared secret for Bob
```

```
    std::cout << "Shared secret (Alice): " << secretA << std::endl;
    std::cout << "Shared secret (Bob): " << secretB << std::endl;

    return 0;
}
```

EXPECTED OUTPUT

Shared secret (Alice): 2

Shared secret (Bob): 2

7. Implementation of the ELGAMAL cryptosystem.

Code:

```
#include <iostream>
#include <cmath>
#include <cstdlib>

long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    long long p = 23; // A prime number
    long long g = 5;  // A primitive root modulo p
    long long x = 6;  // Private key
    long long y = modExp(g, x, p); // Public key

    long long k = 15; // Randomly chosen
    long long c1 = modExp(g, k, p); // First part of ciphertext
    long long c2 = (modExp(y, k, p) * 10) % p; // Second part of ciphertext (message = 10)

    std::cout << "Ciphertext: (" << c1 << ", " << c2 << ")" << std::endl;
```

```
    // Decrypting
    long long s = modExp(c1, x, p); // Shared secret
    long long m = (c2 * modExp(s, p - 2, p)) % p; // Original message

    std::cout << "Decrypted message: " << m << std::endl;

    return 0;
}
```

EXPECTED OUTPUT

Ciphertext: (19, 20)

Decrypted message: 10

8. Implementation of Goldwasser-Micali probabilistic public key system

Code:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <cmath>
#include <cstdlib>
#include <ctime>

// Modular exponentiation: (base^exp) % mod
long long power(long long base, long long exp, long long mod) {
    long long res = 1;
    base %= mod;
    while (exp > 0) {
        if (exp % 2 == 1)
            res = (static_cast<__int128>(res) * base) % mod;
        base = (static_cast<__int128>(base) * base) % mod;
        exp >>= 1;
    }
    return res;
}

// Legendre Symbol: (a/p)
// Returns 1 if a is quadratic residue mod p, -1 if non-residue, 0 if a % p == 0
int legendreSymbol(long long a, long long p) {
    long long ls = power(a, (p - 1) / 2, p);
    if (ls == p - 1) return -1;
    return ls;
}
```

```
int main() {
    srand(time(0));

    // 1. Key Generation
    long long p = 67; // Large prime
    long long q = 83; // Large prime
    long long n = p * q;

    // Find a quadratic non-residue 'y'
    long long y = 0;
    for (long long i = 2; i < n; i++) {
        if (legendreSymbol(i, p) == -1 && legendreSymbol(i, q) == -1) {
            y = i;
            break;
        }
    }

    std::cout << "Public Key (n, y): (" << n << ", " << y << ")" << std::endl;
    std::cout << "Private Key (p, q): (" << p << ", " << q << ")" << std::endl;

    // 2. Encryption
    int message_bit = 1; // Message to encrypt (0 or 1)
    std::cout << "\nOriginal message bit: " << message_bit << std::endl;

    long long x = (rand() % (n - 2)) + 2; // Random x in [2, n-1]
    long long ciphertext;

    if (message_bit == 0) {
        ciphertext = power(x, 2, n);
    } else { // message_bit == 1
```

```
        ciphertext = (static_cast<__int128>(y) * power(x, 2, n)) % n;
    }
    std::cout << "Ciphertext: " << ciphertext << std::endl;

    // 3. Decryption
    int decrypted_bit;
    if (legendreSymbol(ciphertext, p) == 1 && legendreSymbol(ciphertext, q) == 1) {
        decrypted_bit = 0;
    } else {
        decrypted_bit = 1;
    }
    std::cout << "Decrypted message bit: " << decrypted_bit << std::endl;

    return 0;
}
```

EXPECTED OUTPUT

Public Key (n, y): (5561, 2)

Private Key (p, q): (67, 83)

Original message bit: 1

Ciphertext: 647

Decrypted message bit: 1

9. Implementation of Rabin Cryptosystem. (Optional).

Code:

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>

long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

long long gcd(long long a, long long b) {
    while (b != 0) {
        long long t = b;
        b = a % b;
        a = t;
    }
    return a;
}

long long lcm(long long a, long long b) {
    return (a / gcd(a, b)) * b;
```

```
}

void generateKeys(long long &p, long long &q, long long &n) {
    // Generate two distinct large primes p and q
    p = 11; // For simplicity, using small primes
    q = 19; // For simplicity, using small primes
    n = p * q;
}

long long encrypt(long long n, long long message) {
    return (message * message) % n; // c = m^2 mod n
}

long long decrypt(long long c, long long p, long long q) {
    long long m1 = modExp(c, (p + 1) / 4, p); // m1 = c^((p+1)/4) mod p
    long long m2 = (p - m1) % p; // m2 = -m1 mod p
    long long m3 = modExp(c, (q + 1) / 4, q); // m3 = c^((q+1)/4) mod q
    long long m4 = (q - m3) % q; // m4 = -m3 mod q

    // Use the Chinese Remainder Theorem to combine the results
    long long result1 = (m1 + p * (m3 % q)) % (p * q);
    long long result2 = (m1 + p * (m4 % q)) % (p * q);
    long long result3 = (m2 + p * (m3 % q)) % (p * q);
    long long result4 = (m2 + p * (m4 % q)) % (p * q);

    std::cout << "Possible decrypted messages: " << result1 << ", " << result2 << ", " << result3 << ", "
<< result4 << std::endl;

    // Return one of the possible messages (for simplicity, return the first)
    return result1;
}
```

```
int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number generation

    long long p, q, n;
    generateKeys(p, q, n);

    std::cout << "Public Key (n): " << n << std::endl;

    // Message to encrypt
    long long message = 7; // Example message
    long long ciphertext = encrypt(n, message);
    std::cout << "Ciphertext: " << ciphertext << std::endl;

    // Decrypting
    long long decryptedMessage = decrypt(ciphertext, p, q);
    std::cout << "Decrypted message: " << decryptedMessage << std::endl;

    return 0;
}
```

EXPECTED OUTPUT

Public Key (n): 209

Ciphertext: 49

Possible decrypted messages: 81, 136, 84, 139

Decrypted message: 81

10. Implementation of Kerberos cryptosystem

Code:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ctime>
#include <cstdlib>

class KDC {
public:
    void registerUser (const std::string& username, const std::string& password) {
        userDatabase[username] = password;
    }

    std::string issueTicket(const std::string& username, const std::string& password) {
        if (userDatabase.find(username) != userDatabase.end() && userDatabase[username] == password)
{
            // Generate a ticket (for simplicity, just a random number)
            std::string ticket = std::to_string(rand() % 10000);
            std::cout << "Ticket issued for user " << username << ": " << ticket << std::endl;
            return ticket;
        } else {
            std::cout << "Authentication failed for user " << username << std::endl;
            return "";
        }
    }

private:
    std::unordered_map<std::string, std::string> userDatabase; // Simple user database
};
```

```
class User {
public:
    User(const std::string& username, const std::string& password, KDC& kdc)
        : username(username), password(password), kdc(kdc) {}

    void authenticate() {
        ticket = kdc.issueTicket(username, password);
    }

    std::string getTicket() const {
        return ticket;
    }

private:
    std::string username;
    std::string password;
    KDC& kdc;
    std::string ticket;
};

class Service {
public:
    void accessService(const User& user) {
        if (!user.getTicket().empty()) {
            std::cout << "Access granted to service for user " << user.getTicket() << std::endl;
        } else {
            std::cout << "Access denied. No valid ticket." << std::endl;
        }
    }
};
```

```
int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number generation

    KDC kdc;
    kdc.registerUser ("alice", "password123");
    kdc.registerUser ("bob", "mypassword");

    User alice("alice", "password123", kdc);
    alice.authenticate();

    Service service;
    service.accessService(alice);

    User bob("bob", "wrongpassword", kdc);
    bob.authenticate();
    service.accessService(bob);

    return 0;
}
```

EXPECTED OUTPUT

Ticket issued for user alice: 5101

Access granted to service for user 5101

Authentication failed for user bob

Access denied. No valid ticket.

11. Implementation of a trusted secure web transaction.

Code:

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/ssl.hpp>

using boost::asio::ip::tcp;

class HttpsServer {
public:
    HttpsServer(boost::asio::io_context& io_context, short port)
        : acceptor_(io_context, tcp::endpoint(tcp::v4(), port)),
          context_(boost::asio::ssl::context::sslv23) {
        context_.set_options(boost::asio::ssl::context::default_workarounds);
        context_.use_certificate_chain_file("server.crt");
        context_.use_private_key_file("server.key", boost::asio::ssl::context::pem);
    }

    void start() {
        do_accept();
    }

private:
    void do_accept() {
        acceptor_.async_accept(
            [this](boost::system::error_code ec, tcp::socket socket) {
                if (!ec) {
                    std::make_shared<HttpsSession>(std::move(socket), context_)->start();
                }
                do_accept();
```

```
        });
    }

    tcp::acceptor acceptor_;
    boost::asio::ssl::context context_;
};

class HttpsSession : public std::enable_shared_from_this<HttpsSession> {
public:
    HttpsSession(tcp::socket socket, boost::asio::ssl::context& context)
        : ssl_socket_(std::move(socket), context) {}

    void start() {
        ssl_socket_.async_handshake(boost::asio::ssl::stream_base::server,
            [self = shared_from_this()](const boost::system::error_code& error) {
                if (!error) {
                    self->do_read();
                }
            });
    }

private:
    void do_read() {
        auto self(shared_from_this());
        ssl_socket_.async_read_some(boost::asio::buffer(data_),
            [this, self](boost::system::error_code ec, std::size_t length) {
                if (!ec) {
                    do_write(length);
                }
            });
```

```
    }

    void do_write(std::size_t length) {
        auto self(shared_from_this());
        boost::asio::async_write(ssl_socket_, boost::asio::buffer(data_, length),
            [this, self](boost::system::error_code ec, std::size_t /*length*/) {
                if (!ec) {
                    do_read();
                }
            });
    }

    boost::asio::ssl::stream<tcp::socket> ssl_socket_;
    char data_[1024];
};

int main() {
    try {
        boost::asio::io_context io_context;
        HttpsServer server(io_context, 443); // Use port 443 for HTTPS
        server.start();
        io_context.run();
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}
```

EXPECTED OUTPUT :

Example 1:

(no output)

Example 2:

Exception: PEM_read_bio_PrivateKey failed

Example 3:

Exception: error:02001002:system library:fopen:No such file or directory

12. Digital Certificates and Hybrid (ASSY/SY) encryption, PKI.

Code:

```
#include <iostream>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include <openssl/err.h>
#include <openssl/aes.h>
#include <openssl/rand.h>

void handleErrors() {
    ERR_print_errors_fp(stderr);
    abort();
}

// Function to encrypt data using AES
void aes_encrypt(const unsigned char* plaintext, unsigned char* ciphertext, const unsigned char* key) {
    AES_KEY encryptKey;
    AES_set_encrypt_key(key, 128, &encryptKey);
    AES_encrypt(plaintext, ciphertext, &encryptKey);
}

// Function to encrypt the symmetric key using RSA
int rsa_encrypt(RSA* rsa, const unsigned char* key, unsigned char* encryptedKey) {
    return RSA_public_encrypt(16, key, encryptedKey, rsa, RSA_PKCS1_OAEP_PADDING);
}

// Function to decrypt the symmetric key using RSA
int rsa_decrypt(RSA* rsa, const unsigned char* encryptedKey, unsigned char* key) {
    return RSA_private_decrypt(RSA_size(rsa), encryptedKey, key, rsa,
RSA_PKCS1_OAEP_PADDING);
```

```
}

int main() {
    // Load the RSA public key from the certificate
    FILE* fp = fopen("certificate.crt", "r");
    if (!fp) {
        std::cerr << "Unable to open certificate file." << std::endl;
        return 1;
    }

    RSA* rsa = PEM_read_RSA_PUBKEY(fp, NULL, NULL, NULL);
    fclose(fp);
    if (!rsa) {
        handleErrors();
    }

    // Generate a random symmetric key for AES
    unsigned char aes_key[16]; // AES-128
    if (!RAND_bytes(aes_key, sizeof(aes_key))) {
        handleErrors();
    }

    // Encrypt a message using AES
    const unsigned char* message = (unsigned char*)"Hello, this is a secret message!";
    unsigned char ciphertext[16]; // AES block size
    aes_encrypt(message, ciphertext, aes_key);

    // Encrypt the AES key using RSA
    unsigned char encryptedKey[RSA_size(rsa)];
    int encryptedKeyLength = rsa_encrypt(rsa, aes_key, encryptedKey);
```

```
    if (encryptedKeyLength == -1) {
        handleErrors();
    }

    std::cout << "Encrypted message: ";
    for (int i = 0; i < sizeof(ciphertext); i++) {
        std::cout << std::hex << (int)ciphertext[i];
    }
    std::cout << std::endl;

    std::cout << "Encrypted AES key: ";
    for (int i = 0; i < encryptedKeyLength; i++) {
        std::cout << std::hex << (int)encryptedKey[i];
    }
    std::cout << std::endl;

    // Clean up
    RSA_free(rsa);
    return 0;
}
```

EXPECTED OUTPUT

Example 1:

Unable to open certificate file.

Example 2:

Encrypted message: a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6

Encrypted AES key: 7d2e4f8a9c3b1a6d5e7f2c9b8a4d3f1b4

13. Message Authentication Codes.

Code:

```
#include <iostream>

#include <openssl/hmac.h>

#include <cstring>


void generateHMAC(const std::string& key, const std::string& message, unsigned char* hmacResult, unsigned int& hmacLength) {

    HMAC(EVP_sha256(), key.c_str(), key.length(),

        reinterpret_cast<const unsigned char*>(message.c_str()), message.length(),

        hmacResult, &hmacLength);

}


bool verifyHMAC(const std::string& key, const std::string& message, const unsigned char* hmacToVerify, unsigned int hmacLength) {

    unsigned char computedHMAC[EVP_MAX_MD_SIZE];

    unsigned int computedHMACLength;


    generateHMAC(key, message, computedHMAC, computedHMACLength);


    return (hmacLength == computedHMACLength) && (memcmp(hmacToVerify, computedHMAC, hmacLength) == 0);

}


int main() {

    std::string key = "secret_key";

    std::string message = "Hello, this is a message.";


    unsigned char hmacResult[EVP_MAX_MD_SIZE];

    unsigned int hmacLength;
```

```
    // Generate HMAC
    generateHMAC(key, message, hmacResult, hmacLength);

    // Print HMAC
    std::cout << "Generated HMAC: ";
    for (unsigned int i = 0; i < hmacLength; i++) {
        std::cout << std::hex << std::setw(2) << std::setfill('0') << (int)hmacResult[i];
    }
    std::cout << std::endl;

    // Verify HMAC
    bool isValid = verifyHMAC(key, message, hmacResult, hmacLength);
    if (isValid) {
        std::cout << "HMAC verification succeeded!" << std::endl;
    } else {
        std::cout << "HMAC verification failed!" << std::endl;
    }

    return 0;
}
```

EXPECTED OUTPUT

Generated HMAC: b7c7a5f6e8c8b0b9a3b63b9f3f4c6a6b8e3a0c5d6c9b4e5a8f4b0f0a8a2a0c4f

HMAC verification succeeded!

14. Elliptic Curve cryptosystems:

Code:

```
#include <iostream>
#include <secp256k1.h>
#include <secp256k1_recovery.h>
#include <cstring>

void handleErrors() {
    std::cerr << "An error occurred." << std::endl;
    exit(1);
}

int main() {
    // Initialize the secp256k1 context
    secp256k1_context* ctx = secp256k1_context_create(SECP256K1_CONTEXT_SIGN |
SECP256K1_CONTEXT_VERIFY);

    // Generate a random private key
    unsigned char privkey[32];
    if (!secp256k1_rand_bytes(privkey, sizeof(privkey))) {
        handleErrors();
    }

    // Create a public key from the private key
    unsigned char pubkey[65]; // 65 bytes for uncompressed public key
    size_t pubkey_len = sizeof(pubkey);
```

```
if (!secp256k1_ec_pubkey_create(ctx, pubkey, &pubkey_len, privkey, 0)) {
    handleErrors();
}

// Message to sign
const char* message = "Hello, this is a message.";
unsigned char msg_hash[32];
// Hash the message (using SHA256)
SHA256((unsigned char*)message, strlen(message), msg_hash);

// Sign the message
unsigned char signature[64];
unsigned int sig_len;
if (!secp256k1_ecdsa_sign(ctx, signature, &sig_len, msg_hash, privkey, nullptr)) {
    handleErrors();
}

// Verify the signature
int verify = secp256k1_ecdsa_verify(ctx, signature, sig_len, msg_hash, pubkey, pubkey_len);
if (verify == 1) {
    std::cout << "Signature verified successfully!" << std::endl;
} else {
    std::cout << "Signature verification failed!" << std::endl;
}
```

```
    // Clean up
    secp256k1_context_destroy(ctx);
    return 0;
}
```

EXPECTED OUTPUT

```
Signature verified successfully!
```