COMP 448: Computer Security

Fall 2019

Project 1: Securing Message Communication using Cryptography

Date Released: 9/30/2019

**Assignment:** In this project, we will implement DH key exchange with RSA signatures. Alice and Bob would like to communicate without having Mallory or Eve read their communication. Thus, they decide to use symmetric key encryption. To derive the secret key, they use Diffie Hellman key exchange. To prevent MITM attacks, they sign their DH communication using RSA. Once they derive a secret key, Alice and Bob can encrypt and decrypt their messages with it. Here are the steps that they follow:

* Alice sends DH parameters (q, α) and Alice's public key ya to Bob. To prevent MITM attacks, Alice signs the DH parameters and public key with her RSA private key.
* Bob receives the DH parameters/public key and uses Alice’s RSA public key to verify their authenticity.
* Bob generates his DH key pair using the same DH parameters and sends his public key yb to Alice. To prevent MITM attacks, Bob signs the public key with his RSA private key.
* Alice uses Bob’s RSA public key to verify the authenticity of his DH public key received.
* Alice and Bob use Diffie Hellman key exchange to derive a secret key K and initialization vector IV.
* Alice and Bob use symmetric encryption with K and IV to encrypt further communication.

To help you, you are provided with four files:

* Makefile: you can use this file to build your project. It compiles your code and generates two executables: alice and bob. To use this file, simply run: make
* ab.h: this file contains the declarations of the functions that you need to implement. Each function is fully commented, and the parameters are fully listed. You are not allowed to make any changes to this file.
* main.c: this file contains the main method for the alice and bob executable files. It reads the command line arguments that the user specifies and calls the proper function in ab.c. You are not allowed to make any changes to this file.
* ab.c: this file is the bulk of your project. Fill in all of the empty functions. Make sure that you close any open files, allocate memory correctly, and deallocate all the memory locations used. For debugging purposes, a hexdump function is provided for you. This function implements the same functionality as running hexdump -C in the Linux command line.

You will use the EVP APIs in the OpenSSL C library (see tutorial in Appendix) to complete this project. You need to choose **secure cryptographic algorithms and parameters**, as discussed in class, for the implementation.

**Note about deriving symmetric key and IV:** the length of the secret key material derived using DH key exchange will be equal to the chosen DH key size, let’s say N bits. Depending on the encryption algorithm and key size you choose, you will need, let’s say, a K-bit key and an M-bit IV. You will have more shared key material than needed: N > K + M. For this project, you must take first K bits as the key, the next M bits as the IV, and the remaining N-K-M bits must remain unused.

Derived secret key material: N bits

|  |  |  |
| --- | --- | --- |
| K-bit key | M-bit IV | N-K-M bits unused |

A real-world application would mix the N bits of shared key material further to get the K+M bits needed for the key and IV. This additional mixing is outside the scope of this project.

Here is the sequence of commands used in a sample run of the project:

* make

# Alice generates DH params and sends them to Bob

* ./alice generate-dhparams a/dh\_params.pem
* cp a/dh\_params.pem b/dh\_params.pem

# Alice generates her DH and RSA key pairs, and signs her DH public key

* ./alice generate-keys a/dh\_params.pem a/rsa\_pair.pem a/rsa\_pub.pem a/dh\_pair.pem a/dh\_pub.pem a/dh\_pub.sig

# Alice sends her RSA public key, DH public key, and DH public key signature to Bob

* cp a/rsa\_pub.pem b/alice\_rsa\_pub.pem
* cp a/dh\_pub.pem b/alice\_dh\_pub.pem
* cp a/dh\_pub.sig b/alice\_dh\_pub.sig

# Bob generates his DH and RSA key pairs, and signs his DH public key

* ./bob generate-keys b/dh\_params.pem b/rsa\_pair.pem b/rsa\_pub.pem b/dh\_pair.pem b/dh\_pub.pem b/dh\_pub.sig

# Bob sends his RSA public key, DH public key, and DH public key signature to Alice

* cp b/rsa\_pub.pem a/bob\_rsa\_pub.pem
* cp b/dh\_pub.pem a/bob\_dh\_pub.pem
* cp b/dh\_pub.sig a/bob\_dh\_pub.sig

# Alice verifies the signature on Bob’s DH public key and derives symmetric key and IV

* ./alice derive a/bob\_rsa\_pub.pem a/dh\_pair.pem a/bob\_dh\_pub.pem a/bob\_dh\_pub.sig a/key.dat a/iv.dat

# Bob verifies the signature on Alice’s DH public key and derives symmetric key and IV

* ./bob derive b/alice\_rsa\_pub.pem b/dh\_pair.pem b/alice\_dh\_pub.pem b/alice\_dh\_pub.sig b/key.dat b/iv.dat

# No output from ‘diff’ means that Alice and Bob derived the same symmetric key and IV

* diff a/key.dat b/key.dat
* diff a/iv.dat b/iv.dat

# Alice encrypts a plaintext message with her key/IV and sends the ciphertext to Bob

* echo "This is a message" > a/plaintext.dat
* ﻿./alice encrypt a/key.dat a/iv.dat a/plaintext.dat a/ciphertext.dat
* cp a/ciphertext.dat b/alice\_ciphertext.dat

# Bob decrypts the ciphertext with his key/IV

* ./bob decrypt b/key.dat b/iv.dat b/alice\_ciphertext.dat b/plaintext.dat
* cat b/plaintext.dat # should output “This is a message”

**Rubric:** To receive credit, submit a .zip file via mygcc containing the all of your code.

Points will be awarded based on the following:

|  |  |
| --- | --- |
|  | Points |
| Task 1: The project allows the user to correctly generate DH Parameters. | 10 |
| Task 2: DH and RSA key pairs are generated correctly. DH public key is properly signed. | 25 |
| Task 3: The signature on a DH public key is verified correctly and the symmetric key and IV are properly derived. | 25 |
| Task 4: The project can successfully encrypt using a symmetric key and IV. | 15 |
| Task 5: The project can successfully decrypt using a symmetric key and IV. | 15 |
| The project uses secure cryptographic algorithms with proper key sizes. | 10 |

**A program that does not compile or link will not be graded.** You are not allowed to make any changes to ab.h or main.c files. Your code should compile with the provided Makefile.

**Your code must compile, link, and execute in the Linux environment using the class VMs: no compile equals no credit. Also, if your code crashes, no partial credit will be given**

The following criteria will be used to grade your submission:

* Does the code compile?
* Does the code function according to the problem specification?
* Is the code readable and well-formatted? Is it well-documented and clear?

**Extensions will not be granted for technology-related issues.** Leave yourself enough time to complete the assignment, submit the assignment using mygcc, and contact the instructor if you run into problems.

**Project Policies:**

* Assignments must be submitted electronically via my.gcc. Be sure to upload your files correctly the first time.
* This project is a group project. Every student needs to work as a part of a 2-person team. If you cannot find a team to work with, contact Dr. Al Moakar and she will assign you a team. Any student that did not declare a team, will be assigned to a team by the professor.
* 20% of the grade will be weighed with the peer evaluation. Students are expected to turn in the peer evaluation form posted on mygcc at the end of the semester. If a student works on his or her own without a team, then 20% of the grade will be deducted from the project grade.
* Students are expected to keep the same team for the whole semester.

**Academic Integrity Policy:**

* Each team is expected to work on its own. Members of each team can work together, discuss ideas, look at each other’s code, and share files among each other.
* Students belonging to different teams should not discuss, share code that directly bears on this project, or look at each other’s code. Any instances of this will be considered a violation of the academic integrity policy of this course and will be reported to the SFRC committee.
* Use or possession of past solutions and similar solutions from online resources is strictly prohibited and is considered a violation of the academic integrity of this course.
* You may use online resources to look up how to use a function or a system call, but you may not copy code from online resources. Any copied code (whether cited or not) is considered a violation of the academic integrity policy of this course.

**Appendix: OpenSSL Library C API**

OpenSSL provides two C libraries for developers to use the cryptographic algorithms and protocols provided:

* libcrypto - provides implementations of major cryptographic algorithms discussed in this class. You will use the libcrypto library for this project.
* libssl - provides implementation of the SSL and TLS network security protocols. This library is outside the scope of this project.

The OpenSSL libcrypto library provides high-level and low-level APIs for using cryptographic algorithms:

* The EVP (short for Envelope) APIs are high-level APIs that hide the complexity of the different cryptographic algorithms. They allow applications to switch between cryptographic algorithms of the same type (e.g. switch symmetric encryption algorithm from 3DES to AES) without substantial code changes. You will use the EVP APIs for this project.
* The low-level APIs allow developers to work with each algorithm separately and provide a greater degree of control. Switching to a different algorithm requires changing the code to switch from one API to another. These APIs are outside the scope of this project.

OpenSSL's librcrypto is a C library - each API consists of a set of data structures and a set of functions that operate (get, set, or transform) on them.

More information: <https://wiki.openssl.org/index.php/Libcrypto_API>

Most OpenSSL API functions expect pre-allocated/pre-initialized data structures as arguments. Many OpenSSL API functions return 1 on success and 0 on error, but some functions follow a different convention (0 on success or data structure pointer on success). Applications using the OpenSSL API are responsible for:

* Initializing and cleaning up the overall OpenSSL internal state
* Allocating, initializing, and deallocating each interface's data structures
* Checking and handling error conditions

The OpenSSL project maintains manual (man) pages for all functions in the OpenSSL API. Refer to each function's man page to learn more about its input/output arguments, behavior, and return codes:

<https://www.openssl.org/docs/man1.0.2/crypto/>

**OpenSSL Initialization and Cleanup**

OpenSSL maintains global internal state that the calling applications need to initialize and clean up.

* openSSL\_add\_all\_algorithms() allocates and initializes a global internal table of cryptographic algorithms. EVP\_cleanup() deallocates it.
* ERR\_load\_crypto\_strings(), ERR\_free\_strings() - allocate and deallocate global error strings. See "Error Handling" section below for more information.

Typically, programs using OpenSSL call the initialization functions at startup and the clean up function right before exiting.

**Error Handling**

Most OpenSSL API functions can encounter error conditions and return a return value that indicates failure (see each function's man page for specific return values). The calling application can check the return values and handle any error conditions.

OpenSSL maintains an error queue of all errors recorded so far. When an OpenSSL function fails, the calling application can print all errors in the error queue and empty the queue:

* ERR\_print\_errors\_fp(stderr) prints the errors in the error queue to stderr and empties the queue.

For more developer-friendly debugging, OpenSSL provides error strings for all error codes. To make the error strings available for error handling functions like ERR\_print\_errors\_fp(), the error strings must be loaded before any other OpenSSL functions are called (usually when a program starts) and cleaned up when the program is done using OpenSSL (usually right before a program exits).

* ERR\_load\_crypto\_strings() loads the libcrypto error strings
* ERR\_free\_strings() cleans up all error strings

**Envelope (EVP) API**

* The EVP (short for Envelope) API is the OpenSSL high-level interface. It provides functions for all major cryptographic algorithms discussed in this course.
* More information: <https://wiki.openssl.org/index.php/EVP>
* **Symmetric encryption/decryption**
  + Refer to each function’s OpenSSL man page for more information.
  + EVP\_EncryptInit()
  + EVP\_EncryptUpdate()
  + EVP\_EncryptFinal()
  + EVP\_Decrypt\_Init()
  + EVP\_Decrypt\_Update()
  + EVP\_Decrypt\_Final()
  + The functions above use an EVP\_CIPHER\_CTX structure to save intermediate state - see EVP\_CIPHER\_CTX\_init() and EVP\_CIPHER\_CTX\_cleanup().
  + The encryption algorithm, key size, and mode of operation for the functions above are specified using an EVP\_CIPHER structure. OpenSSL maintains internally EVP\_CIPHER structures for all supported algorithm/size/mode combinations. The structures can be looked up using EVP\_get\_cipherbyname() or obtained directly with EVP\_des\_cbc(), EVP\_aes\_256\_ctr(), etc.
  + <https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption>
* **Asymmetric key pair and parameter generation**
  + Refer to each function’s OpenSSL man page for more information.
  + DH parameter generation
    - EVP\_PKEY\_paramgen()
    - EVP\_PKEY\_paramgen works on an EVP\_PKEY\_CTX structure: see EVP\_PKEY\_CTX\_new\_id(), EVP\_PKEY\_paramgen\_init(), EVP\_PKEY\_CTX\_set\_dhparamgen\_prime\_len(), EVP\_PKEY\_CTX\_free()
  + Key pair generation
    - EVP\_PKEY\_keygen()
    - EVP\_KPEY\_keygen() works on an EVP\_PKEY\_CTX structure: see EVP\_PKEY\_CTX\_new\_id() for a new key and EVP\_PKEY\_CTX\_new() from existing parameters or key, EVP\_PKEY\_keygen\_init(), EVP\_PKEY\_CTX\_rsa\_keygen\_bits(), EVP\_PKEY\_CTX\_free()
  + <https://wiki.openssl.org/index.php/EVP_Key_and_Parameter_Generation>
* **Digital signature computation and verification**
  + Refer to each function’s OpenSSL man page for more information.
  + EVP\_DigestSignInit()
  + EVP\_DigsstSignUpdate()
  + EVP\_DigestSignFinal()
  + EVP\_DigestVerifyInit()
  + EVP\_DigestVerifyUpdate()
  + EVP\_DigestVerifyFinal()
  + The functions above use a public/private key pair stored in an EVP\_PKEY structure - see asymmetric key generation and PEM read/write functions.
  + The functions above use an EVP\_MD\_CTX structure to save intermediate state - see EVP\_MD\_CTX\_create() and EVP\_MD\_CTX\_destroy().
  + The functions above compute and verify a signature over a message digest (hash). The secure hash function is specified using an EVP\_MD structure. OpenSSL maintains EVP\_MD structures internally for all supported message digests. The structures can be looked up using EVP\_get\_digestbyname(), or obtained directly using EVP\_sha1(), EVP\_sha256(), EVP\_sha512(), etc.
  + <https://wiki.openssl.org/index.php/EVP_Signing_and_Verifying>
* **Key exchange/derivation**
  + Refer to each function’s OpenSSL man page for more information.
  + EVP\_PKEY\_derive()
  + EVP\_PKEY\_derive() works on a EVP\_PKEY\_CTX data structure: see EVP\_PKEY\_CTX\_new(), EVP\_PKEY\_derive\_init(), EVP\_PKEY\_derive\_set\_peer(), and EVP\_PKEY\_CTX\_free().
  + EVP\_PKEY\_derive() uses a public/private key pair and a peer public key in EVP\_PKEY structures - see asymmetric key generation and PEM read/write functions.
  + <https://wiki.openssl.org/index.php/EVP_Key_Agreement>

**PEM Encoding/Decoding API**

* Privacy-Enhaced Mail (PEM) is a format for storing and sending cryptographic data, such as public/private key pairs, and DH parameters. While the initial PEM specification was developed for e-mail applications, PEM has become a de-facto standard for storing cryptographic data in a range of applications.
* Notable functions in OpenSSL PEM API:
  + PEM\_read\_bio\_Parameters, ﻿PEM\_write\_bio\_Parameters
    - Decode/encode PEM parameters (such as DH parameters) from/to a BIO structure to/from an EVP\_PKEY structure.
  + ﻿PEM\_read\_bio\_PrivateKey﻿, PEM\_write\_bio\_PrivateKey
    - Decode/encode PEM public/private key pair from/to a BIO structure to/from an EVP\_PKEY structure.
  + ﻿PEM\_read\_bio\_PUBKEY, ﻿PEM\_write\_bio\_PUBKEY
    - Decode/encode PEM public key from/to a BIO structure to/from an EVP\_PKEY structure
  + <https://www.openssl.org/docs/man1.0.2/crypto/PEM.html>
  + The man page above does not include the description and explanation for PEM\_read\_bio\_Parameters, ﻿PEM\_write\_bio\_Parameters so here are some code excerpts to help you understand how to use them.
    - Assuming that you have the DH parameters stored in a data structure called dh\_params and the bio file is already created, to store the dh parameters in the bio file,

PEM\_write\_bio\_Parameters(bio, dh\_params);

* + - To read the dh parameters from a bio file assuming that bio file was already opened:

EVP\_PKEY \*dh\_params = PEM\_read\_bio\_Parameters(bio, NULL);

* + For more information on BIO structures, see "Buffered Input/Ouput (BIO) API" below.
  + For more information on EVP\_PKEY structures, see "Envelope (EVP) API" above.

**Buffered Input/Ouput (BIO) API**

* The OpenSSL BIO API present an I/O stream abstraction. There are different BIO types:
  + source/sink BIOs: read data from a source or write data to a sink
  + filter BIOs: transform data in some way
* The BIO API works with BIO structures that are allocated and freed.
* Notable BIO types:
  + Refer to each function’s OpenSSL man page for more information.
  + BIO\_s\_mem: memory buffer allocated on the heap, source/sink BIO
    - BIO\_new()
    - BIO\_s\_mem()
    - BIO\_get\_mem\_data()
    - BIO\_free()
  + BIO\_s\_file: file, source/sink BIO
    - BIO\_new\_file()
    - BIO\_free()
* <https://wiki.openssl.org/index.php/BIO>