Project 1: Implementing a Secure Channel

This project is due on **Thursday, October 7** at **5 p.m.** and counts for 9% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 12 hours until they're received. The professor may grant individual extensions, but only under truly extraordinary circumstances. As always, *we recommend that you begin early.*

This is a group project; you will work in **groups of two** and submit one project per group. Please find a partner as soon as possible. If you have trouble forming a group, first try posting to the class forum on CTools, and then get in touch with the GSI.

The code your group submits must be entirely your own work. You are free to discuss the project with other members of the class, but you may not look at any part of someone else's solution. You may consult published resources, provide that you appropriately cite them (with program comments), as you would in an academic paper.

We will answer questions about the project in the Friday discussion section and the CTools forum. As per the collaboration policy, do not post any code from your project to the forum. Please check the forum regularly for updates or clarifications.

Update 9/17: Corrected key derivation strings in Section 2.2

1 Introduction

In this project, you will use C++ and the OpenSSL library to implement a secure channel between a client and a server. The channel will use HMAC-SHA256 for integrity and AES-128 encryption in CTR (counter) mode for confidentiality. At first, you will use a pre-shared secret key, and in later parts of the project you will implement Diffie-Hellman key exchange and RSA digital signatures in order to establish and authenticate session keys. Because networking is not a prerequisite for this course, we will provide networking functions. We will also provide a basic framework for your code to make sure that everyone starts with a reasonable design.

Objectives:

- Familiarize yourself with basic cryptographic primitives
- Be able to use implementations of cryptographic primitives correctly
- Develop an understanding of encrypted communication protocols

2 Requirements

Your program will behave like a very simple encrypted chat client. For either the server or client, it will send each line entered on standard input to the remote host, which will output the line on standard output. The code you add should not print any other messages to standard output. Whenever the program terminates because of a protocol error, it should print a diagnostic message to standard error and exit with nonzero exit status.

Complete Parts 0–3 below by implementing subclasses of `SecureChannelPeer`. You may add other new classes as necessary.

2.1 Part 0: Exchange unsecured messages

The first part of the project is to familiarize yourself with the code we provide and make sure you can exchange messages between the client and server. In this part and all others, both the client and the server should print each received message to the standard output, followed by a newline.

2.2 Part 1: Create a secure channel using a pre-shared key

The next part of the project is to implement the secure channel using a fixed, pre-shared key.

Be careful: For interoperability, all integers that are used as opaque inputs to cryptographic functions or that are sent over the network should be in network byte order (big-endian). See the manpage for the `htonl` function for how to convert integers to and from network byte order.

The protocol follows:

1. To intialize the channel (on either end):

   (a) Derive separate keys for integrity and confidentiality for messages originating on each side of the connection. Use the pre-shared key (`const char PRE_SHARED_KEY[16]`) to compute MAC values of four strings: "`int key for client`", "`conf key for client`", "`int key for server`", and "`conf key for server`". In all cases, *do not include the terminating null character at the end of the string when computing the MAC*. Specifically, compute HMAC-SHA256(*pre-shared-key*, *string*) for each of the four strings and drop the second half of the 256-bit result to get a 128-bit integrity or confidentiality key for the client or the server.

   (b) Initialize two unsigned 32-bit message counters (sent and received) to zero. All messages will be numbered to ensure that messages are received in order and are not replayed.

2. To send a message with plaintext *p*:

   (a) If the message counter would wrap if incremented, terminate with an error. Otherwise, increment the sent message counter by 1, and let *i* be the value of the counter.

(b) Encrypt the body of the message with AES-128 in CTR mode. CTR mode uses a block cipher to construct a stream cipher. Implement CTR mode as follows. For each message, maintain an unsigned 32-bit stream counter, j. To derive the next 16 bytes of the stream, compute AES-128$_{\textit{sender's-conf-key}}(i \parallel j \parallel 0s)$ (i.e., i concatenated with j and then padded to the right with null characters and increment j by 1. Generate enough stream bytes to cover the length of p, and discard any extras. The ciphertext is then $c = p \oplus \textit{stream}$.

(c) Compute $v =$ HMAC-SHA256$_{\textit{sender's-int-key}}(i \parallel c)$.

(d) Transmit $i \parallel c \parallel v$ to the receiver. Note that i and v are not encrypted.

3. To receive a message:

   (a) If the message is too short (less than 4 bytes + 32 bytes long), terminate with an error.

   (b) Split the received message into $i' \parallel c' \parallel v'$ based on the fixed lengths of i and v.

   (c) If $i' \leq$ the count of received messages, terminate with an error.

   (d) If $v' \neq$ HMAC-SHA256$_{\textit{sender's-int-key}}(i' \parallel c')$, terminate with an error.

   (e) Decrypt c' with AES-128 in CTR mode using the sender's confidentiality key.

   (f) Set the count of received messages to *i'* and return the message plaintext.

To implement this protocol, you must use the OpenSSL implementations of AES, HMAC, and SHA-256. Do not use any other cryptographic libraries (including OpenSSL's SSL implementation).

2.3 Part 2: Negotiate a session key with Diffie-Hellman

The problem with your solution in Part 1 is the fixed, pre-shared key. In practice, you want to use different keys for each session, and you don't want to have to share a different secret with every potential communication partner. In this part of the project, we'll use a protocol based on Diffie-Hellman key exchange to negotiate a session key.

The three messages in the protocol are as follows:

1. Upon connecting to the server, the client randomly chooses d, the 0-based index of the Diffie-Hellman parameter set (p, q, g) to use, and sends d to the server as an unsigned 32-bit integer. The server must verify that d is a valid index. (We have provided several sets of Diffie-Hellman parameters; see `part2.cpp` and `part2.h`.)

2. The server randomly chooses x, where $1 \leq x \leq q-1$. It computes $X = g^x \bmod p$ and sends X to the client.

3. The client verifies that $X \bmod p \neq 1$ and $X^q \bmod p = 1$ (necessary for underlying mathematical reasons). It then chooses y where $1 \leq y \leq q-1$, computes $Y = g^y \bmod p$, and sends Y to the server. The server verifies that $Y \bmod p \neq 1$ and $Y^q \bmod p = 1$. If verification fails on either the client or the server, the program should terminate with an error.

Without sending any further messages, the client and the server each compute the shared key k. First, the client computes $l = X^y \bmod p$ and the server computes $l = Y^x \bmod p$; these values are the same because $(g^x)^y = (g^y)^x$. Next, each side computes k by taking the first 128 bits of HMAC-SHA256$_0(l)$ (i.e., HMAC with a key of 0). Use k in place of the `PRE_SHARED_KEY` you used in Part 1.

You may use the OpenSSL `BIGNUM` library to complete this part of the project. To send a `BIGNUM` object over the network, use the `BN_bn2mpi` function to convert it to a bitstring. Use the `BN_mpi2bn` function to convert bitstrings received from the network back to `BIGNUM` objects. The bitstrings generated by these functions conveniently begin with a 32-bit unsigned length field, n; the entire bitstring occupies $n+4$ bytes.

2.4 Part 3: Authenticate the other party with RSA

The problem with the solution in Part 2 is that the protocol does not provide any way to authenticate the sender and is thus susceptible to man-in-the-middle attacks. Assuming that the two parties know each other's RSA public key, we can use digital signatures to authenticate them. For purposes of this part, we provide both the RSA public and private keys for the client and the server in `client.{pub,private}` and `server.{pub,private}`. However, your implementation must not use the private key for the role it is not currently playing (client or server), as it would not know that private key in a real implementation.

The changes to the protocol from Part 2 are as follows:

1. In the first message (from the client to the server), add a random 32-bit integer (a *nonce*) N as the last 32 bits of the message. Use the OpenSSL `RAND_bytes` function to generate this random number.

2. To the second message (from the server to the client), append an RSA signature for the message received from the client concatenated with the entirety of the second message except the signature (i.e., $d \parallel N \parallel X$). Before the RSA signature, append its length as an unsigned 32-bit integer. The whole protocol message will look like $X \parallel \textit{sig-len} \parallel \textit{sig}$. When the client receives this second message, it must verify its signature and terminate with an error if the signature is incorrect.

3. To the third message (from the client to the server), append an RSA signature for: (1) the original message sent to the server, (2) the entire message just received from the server, and (3) the message about to be sent to the server. In other words, the signature will be over $d \parallel N \parallel X \parallel \textit{sig-len} \parallel \textit{sig} \parallel Y$, and the message to be sent is $Y \parallel \textit{sig2-len} \parallel \textit{sig2}$. When the server receives this third message, it must verify its signature and terminate with an error if the signature is incorrect.

You should use the OpenSSL library's implementation of RSA to sign and verify the signatures in this part. Specifically, use the high-level EVP interfaces to signing and verification (see the `EVP_SignInit(3ssl)` manpage). Load private keys using `PEM_read_PrivateKey` and load public keys using `PEM_read_PUBKEY` (see the `PEM(3ssl)` manpage for more information).

3 Implementation

We have provided `network.h`, which contains functions for sending and receiving messages. We have also provided `driver.cpp`, which parses command-line arguments and decides which of your protocol implementations to use based on the part of the project being tested. The files `part2.cpp` and `part2.h` contain necessary constants for Part 2 of the project. In addition, there is a header file, `SecureChannelPeer.h`, specifying the interface for the classes you are to write in order to complete the project. The file `peer_template.cpp` is a starting point for your implementation; you can compile with it in order to verify that our driver compiles correctly in your development environment. Do not modify any of the provided code except as indicated by "TODO" comments.

3.1 Building and running

To compile the project with g++, you need to specify the `-lssl` and `-lcrypto` flags in order to link with the OpenSSL libraries. For grading, we will compile your project with:

```
$ g++ -lssl -lcrypto -o driver *.cpp
```

You can develop your program in whatever environment you wish, but when you hand it in, it must compile and work correctly on the CAEN `login.engin.umich.edu` Linux machines.

You will be running both the client and the server on the same machine for purposes of this project. Run your program as follows:

```
$ driver -P <N> [-p <server port>] [-L <log file>]
```

Here, `driver` is the path to your program, and `N` is the number of the part you are working on (0–3). The optional flag `-p` will start your program as the client and connect to the server using the given port; if this flag is omitted, the driver will start your program as the server and print the port number on which it is running. The `-L` flag, also optional, causes all network traffic to be appended to a specified log file, which should help you debug your code. Specify different log files for the client and server in order to avoid confusion.

3.2 Submitting

This project is due **Thursday, October 7** at **5 p.m.**. For submission, place your files in a directory named "`proj1`" and archive it as "`proj1.`*member1uniqname*`.`*member2uniqname*`.tar.gz`". (To help us process submissions easily, please put your uniqnames in alphabetical order). Attach the archive file to an email to `eecs398@umich.edu` with "`Proj1 Submission`" as the subject. You should receive a confirmation email within 15 minutes.

To create the archive under Linux, change to the `proj1/` directory and run:

```
$ cd ..; tar -zcf proj1.member1uniqname.member2uniqname.tar.gz proj1/
```

3.3 Grading

Your program is expected to run correctly and provide the security properties guaranteed by the specific protocol in each part. Security bugs will be penalized. □