EECS 398 Project 1: Implement a Secure Channel

29 September 2009

Part 1 Released: September 25, 2009
Parts 2 and 3 Released: September 29, 2009
All parts due: 5 PM, October 12, 2009.

Revision History

- 3.0 (October 1, 2009) – Change the cipher mode in use to OFB because OpenSSL's CTR implementation is obscure at best. Correct previous advice on how to set IVs; they should *not* be zero. Extend the due date to October 12, 2009 (but project 2 may still be released on October 8). Require four *specific* strings for generating encryption and authentication keys instead of leaving the exact strings up to students. Clarify that the server's choice of x in the second key-negotiation message in Parts 2 and 3 should be random. Include clarification about printing received messages from the forum.
- 2.1 (September 29, 2009) – remove references to part3.cpp and part3.h, which don't exist. Add part2.cpp and part2.h to the provided code. They were mistakenly left out in the previous revision.
- 2.0 (September 29, 2009) – add revision history. Clarify that || means concatentation. Describe parts 2 and 3 in detail. Clarify that integers must be sent over the network in network byte order. In the code, change the name of SecureChannelPeer::SendMessage to SecureChannelPeer::SendEncryptedMessage. Minor formatting/presentation fixes.
- 1.0 (September 28, 2009) – describe chat-like functionality of the program at the end of Section 6. Minor formatting/presentation fixes.
- 0.1 (Septmber 25, 2009) – printed handout in discussion. Original version.

1 Introduction

In this project, you will implement a secure channel between a client and a server to familiarize yourself with the basic cryptographic primitives provided by the OpenSSL library. The channel will use AES encryption in OFB (output feedback) mode with a pre-shared key. In the later parts of the project, you will implement Diffie-Hellmann and RSA in order to exchange and authenticate session keys. Because networking is not a prerequisite for this course, we will provide networking functions. We will also provide a framework for your code to make sure that everyone starts with a reasonable design.

2 Goals

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

3 Groups

This is a group project; you will work in teams of 2 and submit one project per team.

4 Requirements

4.1 Part 0: Exchange cleartext

The first part of the project is to exchange cleartext between the client and the server in order to familiarize yourself with the code we provide and make sure you can see the packets sent on the network. 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.

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

The second part of the project is to implement the secure channel using a fixed, pre-shared key. The details of the protocol will be explained in discussion on Friday, September 25. In all the protocol descriptions to follow, || represents concatenation. An overview follows:

- Compute the encryption and authentication keys for the client and the server by hashing the pre-shared key with four hard-coded strings. Specifically, use the strings "Auth key for client", "Auth key for server", "Enc key for client", "Enc key for server" to generate the respective keys. In all cases, do not include the terminating NUL character at the end of the string when hashing. In particular, compute HMAC-SHA-256(pre-shared-key, string) for each of the 4 keys and drop the second half of the (256-bit) result to get a 128-bit key for AES-128.
- Initialize 4-byte message counters (sent and received) to zero. All messages will be numbered to ensure that messages are received in the correct order and not replayed or lost.
- To send a message:
  - If the message counter would wrap if incremented, terminate with an error.
  - Increment the sent message counter by 1.
  - Let i be the value of the sent message counter.
  - Encrypt the body of the message with AES-128 in OFB mode, using i (padded with zeros on the right) as the initialization vector. (Choosing the IV like this instead of randomly introduces a weakness in our cryptosystem. What is it? This illustrates the difficulties of implementing secure cryptosystems.)

  - Compute MAC = HMAC-SHA-256(myauthkey, *i* || encrypted message)
  - Send *i* || encrypted message || MAC to the other machine. Note that *i* and the MAC are not encrypted.

- To receive a message (*i* || encrypted message || MAC):
  - If the message is too short (less than 4 bytes + 32 bytes long), reject it.
  - If *i* is <= the count of received messages, reject the message.
  - Compute MAC = HMAC-SHA-256(sendersauthkey, encrypted message except last 32 bytes). If the MAC does not match the MAC in the incoming message, reject the message.
  - Decrypt the encrypted message, using *i* as the initialization vector.
  - Set the count of received messages to *i* and return the message.

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

4.3 Part 2: Negotiate a session key with Diffie-Hellmann

The problem with your solution in part 1 is the pre-shared key. In practice, you need to change the key used for encryption regularly. We'll use a protocol based on Diffie-Hellmann (recall that an overview of the Diffie-Hellmann protocol was given in the reading assigned for September 15 – `http://www.cl.cam.ac.uk/~rja14/Papers/SE-05.pdf`) to negotiate a session key instead of reusing a pre-shared key repeatedly.

The three (3) messages in the protocol are as follows:

- Upon connecting to the server, the client randomly chooses d, the 0-based index of the Diffie-Hellmann parameter set (p,q,g) to use, and sends d to the server as a 32-bit number. The server must verify that d is a valid index.

- The server randomly chooses x where $1 \leq x \leq q-1$ and replies to the client with $X = g^x \bmod p$.

- 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 $1 \leq y \leq q-1$ and sends $Y = g^y (\bmod p)$ 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 that found the error should exit with nonzero exit status and print an error message to standard error.

- Without sending any further messages, the client and the server each compute the shared key k. The client computes k as the first 128 bits of SHA-256(SHA-256($X^y \bmod p$) and the server computes k as the first 128 bits of SHA-256(SHA-256($Y^x \bmod p$)). k is used as the AES key in the protocol just as in part 1.

We have provided several sets of Diffie-Hellmann parameters (p,q,g) for use in the protocol in part2.cpp and part2.h.

You may use the OpenSSL BIGNUM library to complete this part of the project. To send an OpenSSL BIGNUM over the network, use the `BN_bn2mpi` function, and use the `BN_mpi2bn` function to convert bitstrings received from the network back to BIGNUMs. The bitstrings generated by these functions conveniently start with the length of the number.

4.4 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 each know the other's RSA public key lets us authenticate the key with digital signatures. For purposes of this part, we provide both the RSA public and private keys for the client and the server in client,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:

- In the first message (from the client to the server), add a random 32-bit number (a.k.a. a "nonce") N as the last 32 bits of the message. Use the OpenSSL `RAND_bytes` function to generate this random number. (There is no point in digitally signing this message. Why?)
- 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., $d||N||X$). Before the RSA signature, append its length as a 32-bit integer. The whole packet will look like $X||siglen||sig$. When the client receives this second message, it must verify this signature.
- 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 $d||N||X||siglen||sig||Y$, but the message to be sent is just $Y||siglen2||sig2$. When the server receives this third message, it must verify its signature.

We have provided public and private keys for in client,server.pub,private. 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).

5 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. 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. 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. part2.cpp and part2.h contain necessary constants for part 2 of the project. You should not modify any of the provided code except as indicated.

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

```
$ g++ -o ./proj1 *.cpp -lssl -lcrypto
```

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. We will answer questions about the project on the forum on the course CTools site, but please do not include any of your project code as per the collaboration policy. You should check the forum on the course CTools site and the course webpage daily for clarifications, as we may have to make substantial changes to the project if there are errors in this specification. As always, we recommend that you begin early.

All integers 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.

6 Running your program

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 partN [serverport]
```

driver is the path to your program. N is the number of the part you are working on (0 to 3). serverport is optional; if it is supplied, the driver will start your program as the client and connect to the server using the given port. If serverport is omitted, the driver will start your program as the server and print the port number on which it is running.

For either the server or client, the program will send whatever you type to the other end (i.e., it's a very simple encrypted chat client). The output of the program should be the same (cleartext) no matter what part you're running.

7 Debugging / Wireshark

If you want to debug your program by checking what messages it's sending, we recommend WireShark (http://www.wireshark.org/). Unfortunately, WireShark requires administrator privileges, so it can't be used on the University Linux computers. If you must use the University machines, we suggest that modify our implementations of SendMessage and ReceiveMessage to log messages to files.

We'll briefly discuss how to use WireShark in discussion on Friday, September 25th. There is also plenty of documentation available on the Web.

8 Submitting

Submit your code via email to swolchok@eecs.umich.edu by 5 PM on October 8, 2009. Please put it in a gzipped tar archive named proj1.member1uniqname.member2uniqname.tar.gz with all the files in one directory called proj1. If your working directory is called proj1, the commands to do this starting from your working directory on Linux are:

```
$ cd ..
$ tar czf proj1.member1uniqname.member2uniqname.tar.gz proj1/
```

So that we can process submissions easily, please put your uniqnames in alphabetical order.

9 Grading

Your programs are expected to run correctly and provide the security properties guaranteed by the specific protocol. Security bugs will be penalized.