>

Project 2: Security

Due on Thursday, March 20th at 11:59 pm

Quick Links

Starter Code
Using the Local Autograder
Using the Local Autograder (...for Project 2)
libtransport
libsecurity
TLV Helpers

Overview

In Project 1, you implemented a **reliable**, bidirectional pipe between two processes over the network. Congrats! Your program is able to send data from any two locations over the internet in a reliable manner.

When sending packets over the internet, tens of routers forward them to the correct destination. For example, see the output from the `traceroute` program that detects the forwarding path for `google.com`:

```
$ traceroute -a -q 1 google.com
traceroute to google.com (142.250.189.14), 64 hops max, 52 byte packets
1 [AS52] 164.67.62.83 (164.67.62.83) 21.406 ms
2 [AS52] 169.232.8.34 (169.232.8.34) 14.888 ms
3 [AS52] cr00f2.csb1--bd10f1.anderson.ucla.net (169.232.4.5) 18.035 ms
4 [AS2152] losa4-agg-01--ucla--50g--01.cenic.net (137.164.24.134) 16.358 ms
5 [AS15169] 142.250.173.78 (142.250.173.78) 30.159 ms
6 *
7 [AS15169] lax31s16-in-f14.1e100.net (142.250.189.14) 16.062 ms
```

Not only do your packets to Google go through multiple routers, but also multiple Autonomous Systems (which are all under different management). How can we trust them to securely transfer our data over? Imagine we're making an online purchase and giving Google our credit card number. How can we ensure that:

1. we're actually giving Google our credit card and not some other entity—these routers could route our packets anywhere they want,
2. only Google sees our credit card—these routers could read the data in our packets and buy things on our behalf, and
3. the data we're sending is accurate—these routers could change our data such that we're buying thousands of dollars of some other product.

> In this project, you'll be implementing a security layer on top of the reliability layer created in Project 1. Your security layer will ensure:

1. **identity**: both the client and the server can verify each other's identity,
2. **privacy**: all data is encrypted to avoid snooping, and
3. **integrity**: all messages were not tampered with.

Notices

1. If you're using GitHub or a similar tool to track your work, please make sure that your repository is **set to private**.
2. As per the syllabus, the use of AI assisted tools is not allowed. Again, we can't prove that such tools were used, but we do reserve the right to ask a couple questions about your solution and/or add project related questions to the exam.
3. In the provided autograding Docker container, there are reference binaries. Please do not reverse engineer the binaries and submit that code—which would be obvious and clear academic dishonesty. Remember, we do manually inspect your code.
4. This project must be done in groups of up to **3 people**. (Can be different from Project 1.)
5. **Make sure to select your highest score as your active submission before the deadline.**
6. **If you're in a group, use Gradescope's group feature to add your group members. Only one member must submit.**
7. Use `./helper zip` to create your submission if you plan on submitting by ZIP file.
8. A lot of material in this specification may seem foreign to you—that's okay! Make sure to review the week 7 discussion slides which go over the intuition behind these new cryptographic protocols.

Setup

Clone the **Starter Code**. The `helper` tools work exactly the same as Projects 0 and 1—see **Using the Local Autograder** and **Using the Local Autograder (...for Project 2)** for more info.

Specification

Create two programs (a server and a client) written in C/C++ (up to version 17). Both programs will use BSD sockets using IPv4 and UDP to listen/connect. For cryptographic primitives, use `libcrypto` which is part of OpenSSL (version 3); the provided `libsecurity` does most of the heavy lifting for you. The expected behavior is the exact same as Project 1, but with different inner logic. See the Project 0 specification and Project 1 Specification for more information.

Libraries

> Like Project 0, we'll be using our own libraries. `libtransport` is essentially the solution to Project 1. You're free to use `libtransport` or to use your own solution to Project 1. See **libtransport** for more info. `libsecurity` has helper functions that help you use cryptographic primitives. See **libsecurity** for more info.

Note: `libtransport` and `libsecurity` are only available in the Docker container. No other third party libraries are allowed.

Quick Info

- **Algorithms and Formats**
  - Message encoding uses Type-Length-Value
  - Key and signature format is in ASN.1 DER
  - Keys use the `NID_X9_62_prime256v1` elliptic curve
  - Signatures use ECDSA
  - Derive shared secret using Diffie-Hellman
  - Encryption uses AES-256-CBC with PKCS #7 padding
    - $L_{\text{plaintext}} = \left\lfloor \frac{L_{\text{payload}} - 60}{16} \right\rfloor \cdot 16 - 1$
  - Key derivation uses HKDF SHA-256. Use info ENC for the encryption key and info MAC for the decryption key. The salt is the `Client-Hello` with the `Server-Hello` appended right after.
  - Message authentication uses HMAC SHA-256—authenticate over the IV and ciphertext concatenated (in that order with the TLV header)
- **Lengths**
  - Public key length: **~91 bytes**
  - Nonce length: **32 bytes**
  - Signature length: **70-72 bytes**
  - IV length: **16 bytes**
  - MAC length: **32 bytes**
- **Files**
  - CA signed server certificate: `server_cert.bin`
  - Server private key: `server_key.bin`
  - CA public key: `ca_public_key.bin`
- **Exit Statuses**
  - Bad certificate exit status: **1**
  - Bad DNS name exit status: **2**
  - Bad signature exit status: **3**
  - Bad transcript exit status: **4**
  - Bad MAC exit status: **5**
  - Unexpected message exit status: **6**

Type-Length-Value (TLV)

In Project 1, packets were defined according to strict boundaries; the SEQ was the first 2 bytes, the ACK was the next 2 bytes, and so on. For this Project 2, this is still true, but the security layer will define its messages differently.

> The security layer **will carry its messages in our reliability layer's payload**. Here is how the security layer's messages look like:

```
                    Bits
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+-+-+-+-+...
|      Type     | Length      | Value
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+-+-+-+-+...
```

The first byte of the message is the type. For example, the type of the Client Hello message is 0x10. The next bytes of the message represent the length of the value. The number of bytes needed to encode the length is variable:

- If the length is less than or equal to 252 (0xFC), the length is encoded in one byte.
- If the length is greater than or equal to 253 (0xFD), the length is prepended with (0xFD) and then encoded in the following two bytes (in network order). For example, the number 255 (0xFF) is written as (0xFD, 0x00, 0xFF).

The last bytes of the message represent its value.

More formally:

```
TLV-TYPE = OCTET

VAR-NUMBER-1 = 0x00-0xFC
VAR-NUMBER-3 = 0xFD 2OCTET
TLV-LENGTH = VAR-NUMBER-1 / VAR-NUMBER-3

<message> = TLV-TYPE TLV-LENGTH *OCTET
```

Note: `OCTET` is the same thing as a byte.

We can also nest TLV messages recursively. For example, take a (modified) Client Hello definition:

```
NONCE-TYPE = 0x01
Nonce = NONCE-TYPE TLV-LENGTH 32OCTET

CLIENT-HELLO-TYPE = 0x10
Client-Hello = CLIENT-HELLO-TYPE TLV-LENGTH Nonce
```

The total size of this (modified) Client Hello message is 36 bytes. Why is that? The size of `Nonce` is 34 bytes: 32 bytes for its data (given by `32OCTET`), 1 byte for the `NONCE-TYPE`, and 1 byte for its `TLV-LENGTH` (since it's less than 252). Add another 2 bytes for `Client-Hello`'s type and length to get 36.

General Operation

Before we send over any data from standard input, we must start with a handshake procedure. Please note that this is different from Project 1's handshake. Remember, this is a

completely separate layer from reliability—the security layer is a drop in replacement for standard input and standard output from the perspective of the reliability layer.

If you receive a message you're not expecting (e.g. a Client Hello after you've sent a Server Hello), exit with **status code 6**.

Client Hello

To start the handshake procedure, the client will send a Client Hello message. This message contains a random value (called a nonce). Generate a public/private key pair and send the public key. Make sure to cache the entire Client Hello message-you'll need it for another step.

```
NONCE-TYPE = 0x01
Nonce = NONCE-TYPE TLV-LENGTH 32OCTET

PUBLIC-KEY-TYPE = 0x02
Public-Key = PUBLIC-KEY-TYPE TLV-LENGTH *OCTET

CLIENT-HELLO-TYPE = 0x10
Client-Hello = CLIENT-HELLO-TYPE TLV-LENGTH Nonce Public-Key
```

Note that the length of the public key is not deterministic. Typically, the public key is 91 bytes.

Server Hello

After receiving the Client Hello, the server will construct and send its own Server Hello message. This message contains its own nonce (independent from the client's nonce), its certificate, a public key, and a signature over the rest of this message.

The certificate contains three values: the associated DNS name (e.g. example.com), a public key, and a signature of the `DNS-Name` and the `Public-Key` appended after each other by the certificate authority. You'll be able to access this certificate by reading the file `server_cert.bin` in the current working directory.

```
DNS-NAME-TYPE = 0xA1
DNS-Name = DNS-NAME-TYPE TLV-LENGTH *OCTET

SIGNATURE-TYPE = 0xA2
Signature = SIGNATURE-TYPE TLV-LENGTH *OCTET

CERTIFICATE-TYPE = 0xA0
Certificate = CERTIFICATE-TYPE TLV-LENGTH DNS-Name Public-Key Signature
```

Note that the length of the certificate is not deterministic. Typically, the public key is 91 bytes, the signature is around 72 bytes, but it could be 70, 71, or 72 bytes long[1]. The DNS name could be of any length.

```
HANDSHAKE-SIGNATURE-TYPE = 0x21
Handshake-Signature = HANDSHAKE-SIGNATURE-TYPE TLV-LENGTH *OCTET

SERVER-HELLO-TYPE = 0x20
Server-Hello = SERVER-HELLO-TYPE TLV-LENGTH Nonce Certificate Public-Key Handshake-Signature
```

To generate the Server Hello:

1. Generate a nonce.
2. Copy the certificate into the `Server-Hello`.
3. Generate a public key. Please note that the `Public-Key` in `Server-Hello` is different from the `Public-Key` in your certificate. This public key must be generated yourself. We will refer to this public key as the server's **ephemeral public key**.
4. Create a handshake signature over the received `Client-Hello`, `Nonce`, `Certificate`, and `Public-Key` appended to after each other using the given private key. The given private key is available by reading the file `server_key.bin` in the current working directory.

We now have the client's public key and our own ephemeral private key. This is enough to derive a symmetric shared secret using Diffie-Hellman. Using HKDF, generate two other keys from the shared secret: an **ENC** key and a **MAC** key. The salt is the `Client-Hello` with the `Server-Hello` appended right after.

Finished

On receiving the Server Hello, the client will verify that:

1. The certificate was indeed signed by the certificate authority. This is done by verifying the signature in the certificate with the public key of the certificate authority. The CA's public key is available by reading the file `ca_public_key.bin` in the current working directory. If the signature fails to verify, exit with **status 1**.
2. The certificate has a valid DNS name. Check the DNS name that was passed into the client's arguments. If it does not match, exit with **status 2**.
3. The Server Hello was indeed signed by the server. This is done by verifying the signature in the Server Hello message with the public key in the server's certificate. If the signature fails to verify, exit with **status 3**.

We now have the server's ephemeral public key and our own private key. This is enough to derive a symmetric shared secret using Diffie-Hellman. Using HKDF, generate two other keys from the shared secret: an **ENC** key and a **MAC** key. These keys have their info strings as ENC and MAC respectively. The salt is the `Client-Hello` with the `Server-Hello` appended right after.

The client may now generate its finished message. The finished message contains a Transcript, which is the HMAC digest of the `Client-Hello` with the `Server-Hello` appended right after.

```
TRANSCRIPT-TYPE = 0x04
Transcript = TRANSCRIPT-TYPE TLV-LENGTH 32OCTET

FINISHED-TYPE = 0x30
Finished = FINISHED-TYPE TLV-LENGTH Transcript
```

When the server receives the Finished message, it should verify that its transcript matches its own (by calculating its own HMAC digest of the `Client-Hello` with the `Server-Hello` appended right after). If it doesn't match, the server should exit with **status 4**.

Data

Both sides may now send (encrypted) data over to each other. Read in from standard input the appropriate amount that the reliability layer says is available. (Remember, you might only be allowed to read a certain amount if the window is close to full.)

```
IV-TYPE = 0x51
IV = IV-TYPE TLV-LENGTH 16OCTET

CIPHERTEXT-TYPE = 0x52
Ciphertext = CIPHERTEXT-TYPE TLV-LENGTH *OCTET

MAC-TYPE = 0x53
MAC = MAC-TYPE TLV-LENGTH 32OCTET

DATA-TYPE = 0x50
Data = DATA-TYPE TLV-LENGTH IV Ciphertext MAC
```

You'll have to do some math to determine how much plaintext to encrypt. See the discussion slides on CBC and PKCS #7 padding. For example, to fill up a packet's payload as much as possible:

- The max payload size is 1012.
- This leaves 1012 - 3 (Data message header) - 18 (IV message) - 34 (MAC message) - 3 (Ciphertext message header) = 950 bytes for the ciphertext.
- Each block is 16 bytes. The nearest multiple of 16 to 950 is 944.
- To prevent another 16 byte block being added, go one less than 944.
- The max plaintext size is 943 bytes.
- $L_{\text{plaintext}} = \left\lfloor \frac{L_{\text{payload}} - 60}{16} \right\rfloor \cdot 16 - 1$

Generate an initialization vector (random bytes, just like a nonce) and encrypt the appropriate amount of plaintext using the **ENC** key. Create an HMAC digest of your `IV` + `Ciphertext` using the **MAC** key.

When receiving data, firstly verify its integrity by calculating the HMAC digest of the received `IV` + `Ciphertext`. If the digest matches the MAC code in the received Data message (exact memory comparison), proceed. If not, exit with **status 5**.

Decrypt the ciphertext using the IV and output it to standard output.

Uhhh, what did I just read?

Admittedly, this is a ton of new information. What's important to note is that **libsecurity** takes care of all the heavy lifting. Read the function definitions in `libsecurity.h`. For example, the entire HKDF stuff is taken care of with the `derive_keys` function. The encrypt/decrypt functions and the HMAC function already use the appropriate keys. The point of this project is to help you learn about network security, but to also expose you to certain cryptographic technologies. You're not expected to know how to use them from scratch, but simply be aware of their existence.

We also have **TLV Helpers** that will help with most of the serialization and deserialization steps.

Common Problems (list will be frequently updated)

- Make sure that **lengths** are in network order/big endian.
- Make sure your hmac is computed over the TLV-serialized representation of you cryptographic primitives.

Autograder Test Cases

Note: Gradescope is the ultimate source of authority for your score. While the local autograder tries its best to replicate Gradescope's environment, it doesn't always match up (computer performance, kernel configuration, etc).

0. Compilation

This test case passes if:

1. Your code compiles,
2. yields two executables named `server` and `client`,
3. has a `README.md`,
4. and has no files that aren't source code/text.

1. Security

1. **Handshake: Client Hello:** `test_client_hello` **(10 points)**
   This test case runs your client executable against our reference server. Your client must send a valid Client Hello message to pass.

   Malformed message: **5 points**

2. **Handshake: Server Hello:** `test_server_hello` **(25 points)**
   Same as 1, but using your server executable to assess valid Server Hello messages.

   Malformed message: **15 points**

3. **Handshake: Finished:** `test_finished` **(25 points)**
   Same as 1, but assessing valid Finished messages and the client's ability to verify information in the Server Hello. Log messages that test your client with bad CA public keys are appended with **`badcert`**. Log messages that test your client with bad server private keys (an impostor) are appended with **`badpriv`**. Log messages that test your client with a mismatched DNS name are appended with **`badname`**.

   Malformed message: **5 points**
   Correct message, but does not perform certificate verification: **10 points**
   Correct message, but does not perform DNS name verification: **15 points**
   Correct message, but does not perform signature verification: **20 points**

4. **Encrypt and MAC: Client:** `test_encrypt_and_mac_client` **(15 points)**
   This test case runs your client executable against our reference server. Your client must encrypt and decrypt data according to the spec. It must also determine when messages with bad authentication codes are sent.

   Either encryption or decryption is faulty: **5 points**
   Encryption/decryption are valid, but does not perform message authentication: **10 points**

5. **Encrypt and MAC: Server:** `test_encrypt_and_mac_server` **(15 points)**
   Same as 5, but using your server executable against our reference client.

Description of Work (10 points)

We ask you to include a file called `README.md` that contains a quick description of:

1. the design choices you made,
2. any problems you encountered while creating your solution, and
3. your solutions to those problems.

This is not meant to be comprehensive; less than 300 words will suffice.

Submission Requirements

Submit a ZIP file, GitHub repo, or Bitbucket repo to Gradescope by the posted deadline. You have unlimited submissions; we'll accept your best scoring one (please do select it as your active submission). If you're in a group, use Gradescope's group feature to select your group members (only one member must submit). As per the syllabus, remember that all code is subject to manual inspection.

In your ZIP file, include all your source code (including `libtransport.h`/`libsecurity.h`) + `README.md` + a `Makefile` that generates two executables named `server` and `client`. **Do not in-**

> **clude any other files; the autograder will reject submissions with those.** If you're using the starter repository, the `helper` tool has a zip method that automatically creates a compatible ZIP file.

Footnotes

1. This is due to how the ECDSA algorithm is nondeterministic. See https://learnmeabitcoin.com/technical/cryptography/elliptic-curve/ecdsa/ if you're curious. ↩