AES-DH

This repository contains a working implementation of both Diffie-Hellman key exchange, and AES encryption, with an emphasis on heavy documentation in order to better understand both algorithms, and how they may be used together in a fully functional, network based application. It serves as an educational aid for those wanting to better understand these two algorithms, how they work underneath the hood, and how they might be implemented in code.

> ⚠ **Warning**
>
> The implementation of AES and Diffie-Hellman within this repository is for **educational purposes only!** They should not be considered cryptographically secure, and go out of their way to choose inefficient methods in order to better understand the underlying concepts.

Where Do I Start?

There's three routes to approach this repository:

- If you want to see AES and DH in action, run the pre-compiled (Or, if you don't trust random programs from the internet, compile it yourself! (See BUILDING.md)) program, and run it from the command line! There's two programs to choose from:
  - `main` / `main_pc` uses AES and DH to allow two instances of the program to securely connect and exchange messages over the network by utilizing sockets!
  - `aes` / `aes_pc` is a command-line utility that allows you to encrypt/decrypt strings and files using the implementation of AES!
- If you want to dive into details and inner workings, simply open one of the source code files and start reading! This implementation is written in C++, but the codebase strives to be understandable even from those who may not be familiar with C++ or programming at all. Source files are heavily documented to explain what each step does, and why this step is necessary:
  - Interested in how to develop a command line utility in C++? `aes.cpp` contains the source code for the `aes` application, and shows interfacing with the AES implementation and reading in files and user input.
  - Interested in getting into the weeds of AES? `aes.h` is the fully functional implementation of the AES algorithm, including ECB, CTR, and GCM modes, and supporting key sizes of 128, 192, and 256 bits. Its size may be daunting, so make sure to check out the Codebase Walkthrough to go over the various parts of each file, and functions of interest!
  - Interested in learning about Diffie-Hellman? `exchange.h` is our Diffie-Hellman implementation.
  - Interested in creating an application that links to OpenSSL? `hmac.h` uses the OpenSSL implementation of `HMAC-SHA256`!
  - Interested in Network programming, and using Linux Sockets? `network.h` holds all the functionality that the program uses to talk over the network, including sending and receiving arbitrary data.
  - Want to better understand the inner working of the programs themselves? `main.cpp` contains the code for the `main` application, `prime.h` contains the functions related to prime-number generation and other mathematical functions, and `util.h` contains helper functions for the `main` application, including the functionality for sending encrypted messages between the peers.

> **Note**
>
> While the primary motivation of this repository is a well-documented AES and DH implementation, the entire code-base has received equally thorough documentation!

- Finally, if you want a higher-level understanding of the code base, but may not want to read raw C++ code, this repository has been documented with Doxygen, and as thus you have an interactive, HTML version of the codebase available in `docs/html`. Simply open the `index.html` with your favorite web browser, and navigate between the various namespaces and functions!

Codebase Walkthrough

> **Tip**
>
> Reading source code be intimidating if you use a normal text editor like Notepad. There are many applications designed for developing/reading code, with features like folding and syntax highlighting. Notepad++ and Kate are excellent options!

This repository is written in `C++`. If you're not familiar with the language, or Programming as a whole, the syntax may seem nebulous, but it's been written to try and make the logic easy to follow. That said, here are some general points:

Operations:

1. Standard mathematical operations include:
    1. Addition: `x + y`
    2. Subtraction: `x - y`
    3. Multiplication: `x * y`
    4. Division: `x / y` Note that because all values in this repository are integers—as opposed to floating point with decimals—division rounds down to the nearest whole number.
    5. Modulus `x % y`, This returns the remainder of `x / y`
2. Bitwise/Logical operations include:
    1. AND: `x & y`
    2. OR: `x | y`
    3. XOR: `x ^ y`
    4. NOT: `!x`

Fixed-Width Types

Fixed width types: The `uint_t` class of numbers are *fixed-width*, defined by the number of bits:

1. `uint8_t`: Is 8 bits, or a byte.
2. `uint32_t`: Is 32 bits, 4 bytes, or a word.

Array Indexing

Array Indexing: AES uses a block of 16 bytes, typically organized as a 4x4 grid. While most implementations forgo this to simply have 16 bytes in a row, this implementation uses the 4x4 scheme to make it easier to follow. Indexing is done with the `[]` operator, so `array[x][y]` would return the value located at column x, row y.

Loops

C++ has two main types of loops, `for` loops, and `while` loops:

For Loops

`for (X; Y; Z)` will initialize the statement X, and continue to loop until Y is no longer satisfied, calling statement Z on each iteration. This is most commonly used in reference to indexing the block:

```
for (size_t row = 0; row < 4; ++row) {
        for (size_t col = 0; col < 4; ++col) {
          buffer[col][row] = array[(col - row) % 4][row];
        }
      }
```

Where `size_t row = 0` and `size_t col = 0` initialize the `row` and `col` index as `0`, `row < 4` and `col < 4` ensure that the loop runs until these values iterate through the entire array, and `++row` and `++col` increment these values on each iteration.

You'll also see the C++ versions:

```
for (const auto& x : arrays) {
...
}
```

This iterates through each element of `arrays`, setting the current element to the value `x`.

While Loops

`while (X)` will repeatedly run until statement X becomes false. Loops (both `for` and `while`) can jump back to the beginning using `continue`, or exit immediately with `break`

The Standard Library

The standard library includes all functions and classes contained in the `std` namespace. This repository uses it extensively, such as the fixed-size collection of objects in the `std::array`, the variable sized collection of objects in the `std::vector`, and utilities like `std::rotl`.

> **Tip**

The C++ Reference: https://en.cppreference.com/w/cpp is a great source for documentation related to everything in the standard library!

Tip

Want to learn C++? *Programming Principles and Practice Using C++* was written by its creator, and is a great resource! https://www.amazon.com/dp/0138308683

AES

Note

This AES implementation was made in reference to:
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf
Alongside:
https://cs.ru.nl/~joan/papers/JDA_VRI_Rijndael_2002.pdf
Be sure to check them out if you want a more mathematically focused explanation of AES.

The GCM implementation was made in reference to:
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

The AES implementation is available within the `aes.h`; as with every file in the repository, the functionality is partitioned into a *namespace*, namely the `aes` namespace. This is why other programs will call AES functionality like `aes::gcm::Enc`, where `Enc` is a function within the `gcm` namespace, which is within the `aes` namespace.

AES operates by taking a message of arbitrary size, and breaking them down into 16 byte *Blocks*. These *Blocks* are collected into a single state. Then, a series of algorithms and transformations are applied to each *Block*, four in total: `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`. These four steps are then repeatedly applied depending on the size of the key, with 10 rounds for a 128 bit key, 12 for a 192 bit key, and 14 for a 256 bit key. This key is transformed into a *Key Schedule* where the key is turned into a unique set of words for each round, and is then applied to the *Block* during the `AddRoundKey` step.

Once we have run through all the steps, we have the ciphertext. For ECB and CTR modes, we generate an HMAC against the ciphertext and key, producing a string that will change if the key or ciphertext is modified in transit, ensuring integrity. Because the key is used for HMAC generation, an attacker cannot modify the ciphertext *and* create a corresponding HMAC! GCM creates its own integrity check that is stored as a *Block* at the end of the ciphertext.

The `aes` namespace includes the following members:

- The `gf` namespace contains functions related to Galois Field computation, used frequently in both AES and AES-GCM.
- The `key` namespace contains functions related to the `KeyExpansion` algorithm in AES, which is used by the `AddRoundKey` step in AES.
- The `state_array` class is the basic unit of AES, containing the *Block*. Every step of AES applies to the *Block*, and is thus implemented within this class as member functions. Importantly there is:

    - `state_array::AddRoundKey`: Add the round key.
    - `state_array::SubBytes`: Perform the substitution step.
    - `state_array::InvSubBytes`: Revert `SubBytes`
    - `state_array::ShiftRows`: Transpose the rows by a cyclical shift.
    - `state_array::InvShiftRows`: Revert `ShiftRows`
    - `state_array::MixColumns`: Transform each column by a matrix.
    - `state_array::InvMixColumns`: Revert `MixColumns`
    - The `state_array` is transformed in place, which means it is initially filled with plaintext, and each of these above steps are applied, changing the internal values, before the final ciphertext is unraveled out as a string.
- The `state` is little more than a collection of individual `state_arrays`. Because *Blocks* are fixed at 16 bytes, the `state` contains an entire message broken into these 16 byte segments. It is responsible for generating the Key Schedule (See `state::Schedule`), but besides that does nothing more than apply all of the steps mentioned in the `state_array` to each *Block*.
- The `Cipher` and `InvCipher` functions are a verbatim translation of the Encryption and Decryption outlined in the Reference paper. Taking a string, a key, and a round number, it encrypts the message with AES, returning the resulting cipher text. Using these functions by themselves is using AES in ECB mode.
- The `Ctr` function is a implementation of the AES-CTR mode, where rather than passing the plaintext through AES directly, we instead generate a nonce value, pass that through AES to get a *Pad*, and then perform a One-Time Pad form of encryption where the plaintext is XOR'd against this *Pad*, to which a unique pad is generated for each *Block* in the plaintext by the incrementing nonce. Because encryption is done via XOR, `Ctr` both encrypts and decrypts a message.
- The `gcm` namespace includes all the functions related to the AES-GCM mode. These functions were implemented in reference to: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf
    - The `increment` function increments the Nonce value; unlike AES-CTR, the nonce has a specific algorithm for incrementing it to the next value.
    - The `mult` function multiplies two *Blocks* together.
    - The `GHASH` function is the main aspect of AES-GCM, and generates an authenticated hash of the state, returning it in a *Block* that can be appended onto the state.
    - The `GCTR` function is almost identical to `aes::Ctr`, but rather than taking a numerical nonce, it uses a *Block* nonce/IV called `ICB`. It also uses `aes::gcm::increment` To step the `ICB` to new values, and rather than returning a string message, returns the block state instead.
    - The `Enc` function takes a message, a key, a round count, and nonce, and encrypts the message with AES-GCM.
    - The `Dec` function takes a ciphertext, a key, a round count, and a nonce, and will decrypt the message with AES-GCM if and only if the `GHASH` matches, and will refuse to decrypt if there have been changes to the key or any blocks.

> **Note**
>
> You'll find many auxiliary functions in the `state_array`, and other classes. They aren't important to the fundamental understanding of AES, so feel free to ignore them if they aren't mentioned here.

Exchange

> **Note**

This implementation was made in reference to:
https://datatracker.ietf.org/doc/html/rfc2631#section-2.1

Diffie-Hellman is particularly clever because it relies on the associativity of multiplication. The basic steps are this:

- Alice and Bob generate two secret values a and b.
- Alice and Bob agree upon two public values.
  - A prime number p that is used as modulus.
  - A value g, such that it is a *primitive root* of p. This means that raising g from every value between 1 and $p-1$ will lead to a unique mapping of values that includes every value between 1 to $p-1$. This ensures no private key a, b when raised to g, has a corresponding value a', b' that would return the same value.
- Alice takes her private key, and raises g by it and mods by p: $A = g^a \mod p$ This constitutes her *intermediary*, which is then sent to Bob.
- Bob, likewise, raises his private key: $B = g^b \mod p$ and sends it to Alice.
- Now, each peer raises this intermediary by their own private key:
  - Alice: $B^a = (g^b)^a = g^{b \times a} \mod p$
  - Bob: $A^b = (g^a)^b = g^{a \times b} = g^{b \times a} \mod p$
  - Therefore, then they reach the same *Shared Key*, which can be used for communication, without sending any private values across the network.

Diffie-Hellman relies on a the idea that, with large enough values p, a, b, trying to manually determine values a, b from only knowing $g^a \mod p, g^b \mod p$ is infeasible. For more information: see https://en.wikipedia.org/wiki/Discrete_logarithm

The Diffie-Hellman Key Exchange implementation is located in `exchange.h`, within the `exchange` namespace. Unlike `aes`, there isn't near as many members:

- The `compute_intermediary` function takes the public values p and g, alongside a private key k and computes the intermediary value that is sent to the other peer.
- The `exchange_keys` function generates the private and public keys, and establishes a shared key between another computer by communicating over a socket.

> **Note**
>
> While the primary Diffie-Hellman algorithm is implemented as `exchange_keys`, This implementation uses 64 bit keys, which is unacceptable for use within AES. Therefore, the `main` program actually exchanges 4 keys, totaling 256 bits. Take a look at `util::construct_shared_key` for the code!

Application Walkthrough

Interesting in seeing AES and DH in action? This project compiles two applications that you can use to see the implementations working: `aes` and `main`

`aes`

`aes` is the simpler of the two, simply providing a command line utility interfacing with our AES implementation. Simply run `./aes` from the project directory in your shell of choices!

> ⚠ **Warning**
>
> `aes` only supports long-style command line flags, such as `--flag1=value1 --flag2=value2`. Values must be separated by an `=`, with no white space between

The only required argument is `--mode`, which specifies which mode of AES to use. This is represented as three values separated by dashes, in the format `ENC-256-GCM` where:

- `ENC/DEC` is the first string, specifying whether this is an encryption or decryption operation.
- `128/192/256` is the second string, specifying the key size.
- `ECB/CTR/GCM` is the final string, specifying the specific AES method to use.

`./aes --help` will list all the available options, but here are the important ones:

- `--infile` specifies the source of data. If not provided, the user will be prompted to supply a message.
- `--outfile` specifies where the output data should be send. If not provided, the output will be output to the console.
- `--keyfile` specifies a file used for the key. If not provided, the user will be prompted to supply a key.

Some examples:

```
# Type in a message and key, encrypt with AES-256-GCM, and write the output to /out.bin
aes --mode=ENC-256-GCM --outfile=/out.bin

# Decrypt that file, asking the user for the key, and writing to the console.
aes --mode=DEC-256-GCM --infile=/out.bin

# Encrypt myfile.txt with key entered in the console, output the ciphertext onto the console
aes --mode=ENC-128-ECB --infile=/myfile.txt

# Encrypt a typed in message using mykey.txt as a key, writing to the console.
aes --mode=ENC-192-CTR --keyfile=/mykey.txt
```

> 🔥 **Tip**
>
> ECB/CTR modes do not have any authenticity checks when running from the `aes` program, as no HMAC is generated. However, since GCM does have integrity, try encrypting data to a file, then change part of that file before asking to decrypt. GCM will immediately report the modification was detected and refuse to decrypt! You can also try this with keys: ECB/CTR will return garbage data if an incorrect key is provided, but GCM will refuse to decrypt altogether!

`main`

`main` provides an interface for two peers to communicate over a socket, exchange a shared key using Diffie-Hellman, and then use those shared keys to securely send messages with AES encryption. No command line arguments are

required, simply run from your shell!

When first starting, `main` performs a sanity check to ensure that the AES implementation is working correctly. You should see three sentences, each with an AES mode at the end. If these sentences look incorrect (incoherent text, characters that cannot be rendered), then there's something wrong with the program. If you're using the pre-compiled version (`main_pre`), try compiling it yourself, vice versa if you're using a self-compiled version. If everything looks alright, press enter, if not: `Ctrl+C` to stop the program.

Once past that, you will be at the main interface of the program:

```
Status: IDLE
What would you like to do?
0: Request New Connection
1: Listen for New Connection
2: Quit
```

- `Status`: Specifies the state of program. If you are connected to another peer, it will be `CONNECTED`, otherwise it will be `IDLE`. This determines what options you have available.
- `0. Request New Connection` Will allow you to connect to another peer who has selected `Listen for a New Connection`. `main` uses a numerical list for users to provide input. To select this option, type `0`, and then `ENTER`.
- `1. Listen for New Connection` listens for peers to connect to.
- `2. Quit` will close the application.

The networking model of `main` is a two-way communication of a shared socket. On an initial connection, however, one peer will need to be be the *server*, selecting `Listen for a New Connection`, and the other will be the *client*, selected `Request New Connection`. When listening, you provide a port to listen on, and the program will wait 30 seconds for another peer to connect. When requesting, you will provide that same port, and the IP Address of the second computer.

> **Tip**
>
> `main` does not perform DNS lookup, so you need to provide the raw IP address of the listening peer. If you aren't sure what that is, use `ping`! Make sure your firewall allows communication to the port you've picked!

> **Tip**
>
> If you're running two instances of the program on a single computer, provide either the localhost address (`127.0.0.1`), or just type `local`!

> **Note**
>
> The *server*/*client* relationship only applies to the initial communication; once a connection has been established, both peers will be able to freely send and receive messages!

> ⚠ **Warning**
>
> The *client* will fail the connection if the *server* isn't in listening mode. Listen with the *server* first, then connect with the *client*!

> ⚠ **Warning**
>
> Some ports are reserved by the system. Some ports may already be in use by other applications. If `main` reports `Failed to connect!` try using a high port like `2000`, and incrementing by one until you find a free port to bind to!

For an example, we'll launch two instances of `main` on a single computer, and will use port `5000` to listen to. We should see the program report `Listening...` where we have 30 seconds to connect. Don't worry if you don't connect in time, you can just `Listen for New Connection` again, and it'll even remember the port!

With the second program, we'll provide port `5000`, and then `local` because these are both running on the same computer. You should immediately see the program report `Exchaging Keys...`, and then `Complete!` The two peers just used Diffie-Hellman to exchange a shared key! If you see an error, you'll be brought back to the home page, and you can try to connect again.

Now, you should be brought back to the home page, but the status should report `CONNECTED`. Now, you have some new options:

```
Status: CONNECTED
What would you like to do?
Shared Key (Mod 100): 36682272
0: Listen for Request
1: Send an Encrypted Message
2: Re-Exchange Keys
3: Terminate Connection
4: Quit
```

- The `Shared Key` provides you a truncated version of the shared key that was negotiated. Sometimes, a blip in the network communication can lead to values being dropped or changed in transit. If this happens during the key exchange, you won't be able to communicate. Therefore, look at the value, and ensure that they are identical between both peers.
- `0. Listen for Request`: The networking between peers is simplistic, which means that communication is done in a similar way to the initial handshake. One peer will Listen for Requests, which will put the program in an idle state for 30 seconds as it awaits a request from the other peer. In this time, the other peer will use one of the other options.
- `1. Send an Encrypted Message`: Use AES to send an encrypted message using the shared key to the other peer. More details on this below.
- `2. Re-Exchange Keys`: If the shared keys do not match, request that new shared keys be generated and shared.
- `3. Terminate Connection`: Terminate your connection with the peer.

> **Warning**
>
> If the shared keys do not match, then the Encryption/Decryption process will return garbage data!

So, if our shared keys don't match, and we need to re-exchange new values, peer 1 will type `0` to `Listen for Request`, and peer 2 will type `2` to `Re-Exchange Keys`. Peer 1 will be prompted to accept the exchange, and if they accept a new key-exchange will be performed.

To send an encrypted message, Peer 1 will type `0` to Listen, and peer 2 will type `1` to send an encrypted message. As with the initial connection, Listening has a 30 second timeout, but if you timeout the first peer setting up the message, you can just listen again!

For peer 2, you will firstly need to provide a message. This can be of any size, but ends with a newline, or the enter key. Next, you'll have to provide the key size, which can either be 128/192/256. Finally, select which AES mode to use, either ECB/CTR/GCM. Peer 2 will then wait for Peer 1, and once the peer has accepted, will send the ciphertext over. Peer 1 can then decrypt it with the shared key, and the program will print out the Message. Have fun!

> **Note**
>
> CTR and GCM modes will send the Nonce across the wire as well
> CTR and ECB modes will send an HMAC for integrity! GCM manages integrity itself.

> **Tip**
>
> If you run into issues, such as communications immediately failing, simply exit the program, either through the `Quit` option or with `CTRL+C`, and relaunch the program to try again!