CSI4109 Assignment #3
Due: May 2nd 1:59pm

Cryptography with OpenSSL

In this homework assignment, you will be learning how to use the OpenSSL cryptographic library to perform some of the cryptographic operations you learned in class. Note that we are assuming the OpenSSL version 3.0.2—the default version used by Ubuntu 22.04—which can be checked as follows.

```
$ openssl
OpenSSL> version
OpenSSL 3.0.2 15 Mar 2022 (Library: OpenSSL 3.0.2 15 Mar 2022)
```

Part A. Symmetric Crypto (2 pts)

Let's try to encrypt and decrypt a file using OpenSSL's *command-line* tool, `openssl`. Here's one way to encrypt a file:

```
$ openssl enc -aes256 -in plaintext.txt -out ciphertext.txt
```

Notice that we specified a cipher (in this example, AES256), but no key! Instead of passing in a key, you'll be prompted to choose a password when you enter the command. The password is used, together with a salt, to derive a symmetric key. (You can also perform encryption using a particular key, but we'll skip that for now. For more options, refer to https://wiki.openssl.org/index.php/Enc.)

Here's the corresponding decryption command:

```
$ openssl enc -aes256 -d -in ciphertext.txt -out plaintext.txt
```

The syntax is very similar, except for the `-d` option, which instructs OpenSSL to *decrypt* a given ciphertext rather than encrypt it. When decrypting, you'll be asked for the same password that you used to encrypt it. (Yes, the salt that was randomly generated during encryption has been stored in the `ciphertext.txt` file.)

What to submit. For this part, submit `part_a.ctxt` which is the ciphertext of `part_a.txt`, encrypted using `-aes256` and your student ID (no whitespace, e.g., `2016123456`) as the password.

Part B. Asymmetric Crypto

You can use the same command-line tool, `openssl`, to generate RSA key pairs as well.

- Here's one way to generate an RSA private key:

```
$ openssl genrsa -aes256 -out private_key.pem 2048
```

  In this example, we've asked OpenSSL to generate a 2048-bit RSA private key, and call it `private_key.pem`. (`.pem` is a file format commonly used for cryptographic objects, like digital certificates and keys. See here for more detail.) Like in Part A, you'll be asked for a password. That's because the `-aes256` option tells OpenSSL to encrypt the private key using AES256 before writing it to the file. Yes, it's good practice to make sure that keys "at rest" are encrypted. This helps avoid dangerous situations, like pushing an *unencrypted* private key to a public GitHub repository. This means that, technically, `private_key.pem` isn't the "real" private key, but the "real" private key can be derived from `private_key.pem` plus the password (plus the salt that was used to generate the symmetric key) you enter.

- Next, you need to generate a matching public key:

```
$ openssl rsa -in private_key.pem -out public_key.pem -pubout
```

  Here, the input is the private key you just generated, and the output is a matching public key. Because we stored the private key in an *encrypted* form, you'll need to enter your password again in order to generate its corresponding public key here. As you can expect, unlike your private key, the matching public key won't be encrypted.

What to submit. For this part, submit a 2048-bit RSA public key, named `part_b.pem`. (Hold onto the private key corresponding to the public key being submitted, `part_b.pem` —— you'll need it in Part C!)

Part C. Digital Signature Generation (2 pts)

Here's an example of a command that will digitally sign a file using OpenSSL:

```
$ openssl dgst -sha256 -sign private_key.pem -out input.sig input.txt
```

This command will hash the input (in this example, `input.txt`) using whichever algorithm we specify (in this example, SHA-256). `-sign private_key.pem` that follows indicates that we want to sign the resulting hash with the given private key.

What to submit. For this part, you'll submit `part_c.sig`, which is a signature on `part_c.txt` (attached to the homework assignment) using your private key from Part B and the hashing algorithm SHA-256. We will verify your signature using the public key that you submitted, `part_b.pem`, in Part B.

Part D. Digital Signature Verification (2 pts)

Now, you'll verify some signatures that were generated by us using the command shown in Part C. When you unzip `sigs.zip` provided as part of this homework assignment, you'll find a directory named `sigs`. In this directory, there is a unique directory for each student. To find your personal directory, compute a SHA-256 hash of your student ID (no whitespace, no newline); your directory name is the first eight hex digits of this hash. **Your homework submission will be graded assuming that you are using the files in your own personal directory.**

For example, if your student ID were `2016123456` then your directory would be `sigs/47f4d4ab`. Your directory contains three files (`file1.txt`, `file2.txt`, `file3.txt`), and 3 signatures (`file1.sig`, `file2.sig`, `file3.sig`). These files and signatures are unique to you.

You'll verify these signatures against the 2048-bit RSA public key `part_d.pem` (also included in the homework assignment). The signatures were all created using the same hashing algorithm (SHA-256) and the same private key, which corresponds to the provided public key `part_d.pem`. Use the OpenSSL command-line tools to determine whether `file1.sig` is a valid signature on `file1.txt` (and so on for the other file-signature pairs). Some of the signatures will verify, and some will not.

What to submit. For this part, you'll submit a file called `part_d.txt` with 3 lines. On each line, write either `VALID` or `INVALID`, followed by a newline character, to indicate whether the signature was valid. For example, if you found that only the second (`file2.sig`) and third signatures (`file3.sig`) were valid, you'd submit a file with the following:

```
INVALID
VALID
VALID
```

Part E. Message Authentication Codes (4 pts)

In this part, you will use OpenSSL's **application programming interface (API)** (**Note:** no longer its command-line interface) to write a C or Rust program called `cryp`, which supports **authenticated encryption/decryption** using HMAC-SHA256. `cryp` must support two modes: `enc` or `dec`. Your program should read in the input file and encrypt the file content (if the mode is `enc`) or decrypt them (if the mode is `dec`).

You can choose **any symmetric cipher with any mode of operation** (e.g., AES256 with CBC) for encryption and decryption. It is required, however, that your program performs ***authenticated* encryption using HMAC-SHA256**: **after** encryption, you need to produce a HMAC-SHA256 tag (i.e., message authentication code) for the ciphertext (called `encrypted.tag` below); **before** decryption, you would need to first verify the authenticity of the given ciphertext with the given HMAC tag.

For example, here's how we would run `cryp` to perform an authenticated encryption and decryption of the content of `original.txt` with a key stored in a file `shared.key`:

```
$ cat shared.key
This is shared key
$./cryp enc -key shared.key -in plain.txt -out cipher.txt -tag cipher.tag
```

As a further example, when provided with proper keys, ciphertexts, and tags, the following sequence of commands should write the original content of `original.txt` to the file `decrypted.txt`:

```
$ ./cryp enc -key shared.key -in original.txt -out encrypted.txt -tag encrypted.tag
$ ./cryp dec -key shared.key -in encrypted.txt -tag encrypted.tag -out decrypted.txt
```

You can overwrite all output files if they exist already. You can assume that all input files exist when `cryp` is invoked. You can choose the format and encoding (e.g., hex or base64) of `encrypted.txt` and `encrypted.tag`.

- **On success**: Your program should exit with `0` and not print anything to stdout as shown above.
- **On failure**: There are two broad classes of security issues that we will test your submission against.
  - The first failure condition pertains to attacks to confidentiality. After the `enc` command, `encrypted.txt` must be an *encrypted* version of `original.txt`.
  - The other pertains to attacks to authenticity. Upon receiving `dec` command, your program must perform an authenticity check (think about an attacker tampering either with `encrypted.txt` or with `encrypted.tag`), your program must exit with `1` and **print** `VERIFICATION FAILURE` followed by a newline character to stdout, **without** producing a decrypted output file.
  - Additionally, if there are **any** other errors (e.g., a wrong number of arguments, or any other failure condition), then the program must exit with `2` and print `ERROR` followed by a newline character to stdout.

What to submit.

- `Makefile`
- `README`
- The source code of your program

Implementation.

- **Your program must work on Ubuntu 22.04 64-bit with the default packages installed.** In addition to the default packages, the following packages (for compiling C/Rust programs, and for using the OpenSSL library) are also installed:
  - C (`gcc`)
  - Rust and Cargo (`rustc` and `cargo`). Use the following crates if you would like to use them:
    - `hex-literal = "0.4"`
    - `sha2 = "0.10"`
    - `hmac = "0.12"`
    - `openssl-sys = "0.9"`
    - `openssl = "0.10"`

  - OpenSSL library (`libssl-dev`)

  You'll probably need to set up a virtual machine to do your development. VirtualBox is a free and open-source VM system. Or, if you are using MS Windows, you may want to use WSL (WSL version 2 is recommended.) (Ubuntu 22.04 on Microsoft Store).

- **Hint:** You may want to start by understanding the library functions found in the following files.

  - If you are writing your program in C, consult the following.
    * `https://github.com/openssl/openssl/blob/openssl-3.0/include/openssl/aes.h`
    * `https://github.com/openssl/openssl/blob/openssl-3.0/include/openssl/hmac.h`
    * `https://github.com/openssl/openssl/blob/openssl-3.0/include/openssl/sha.h`
  - If you are writing your program in Rust, consult the following.
    * `https://docs.rs/openssl/0.10/openssl/symm`
    * `https://docs.rs/hmac/0.12/hmac`

- You can use publicly available source code (e.g., Rust crates) licensed under an open-source license (other than the aforementioned ones), **provided that they have been approved by the instructor or TAs beforehand.** Using them without our approval will lead to serious penalties.

Submission Instructions

Submit on LearnUs (`ys.learnus.org`) the following files.

- Part A: `part_a.ctxt`
- Part B: `part_b.pem`
- Part C: `part_c.sig`
- Part D: `part_d.txt`
- Part E: `Makefile` , `README` , and your source code files.

The `Makefile` for Part E must create your executable, called `cryp` , when the command `make` is run. Note that we may invoke `make` multiple times, and it needs to work every single time. Your `README` file must be **plain text** and should contain your name, student ID, and a brief description of the source code you wrote for Part E.

Grading Rubric

- Part A. (2 pts)
- Part B + C. (2 pts)
- Part D. (2 pts)
- Part E. (4 pts)