Exercise Sheet Hybrid Encryption

Solve the following exercises and submit them until the communicated date.

LB-HE 01. (*Code::Blocks* template *GMP and libsodium project*)

A server (sender) wants to send an encrypted message to a client (recipient). In the process, a symmetric encryption scheme, specifically AES, is to be used for performance reasons. In order to exchange the key between the two communicating parties in advance, Diffie-Hellman key exchange is used. From key generation to symmetric encryption, all steps are to be implemented.
To do so, using the *GMP* and *libsodium* (see LB-S 01.), write a program which implements the following operations and can be invoked like `<operation> [<operand 1> <operand 2>]`:

- `ServerGeneratePartialKey` to generate a number $x \in \mathbb{Z}_p^*$ and to subsequently compute the partial key $S := g^x \mod (p)$. x and S are output in decimal representation to `std::cout` in **exactly** the following format (example output):

```
x: 1559258775283944[...]
S: 1311271927357378[...]
```

- `ClientGeneratePartialKey` to generate a number $y \in \mathbb{Z}_p^*$ and to subsequently compute the partial key $C := g^y \mod (p)$. y and C are output analogously to the server (example output):

```
y: 1175596360350942[...]
C: 2826535490322092[...]
```

  Note that S and C **must** be mutually distinct, i.e., $g^x \not\equiv g^y \mod (p)$.

- `ServerGenerateSessionKey` to compute the 256-bit-long session key $k := H\left(C^x \equiv (g^y)^x \mod (p)\right)$ for the sender based on C and x, which are passed as command line arguments in this order. The key is output in hexadecimal representation to `std::cout` in **exactly** the following format (example output):

```
d319bbb924009b66[...]
```

- `ClientGenerateSessionKey` to compute the 256-bit-long session key $k := H\left(S^y \equiv (g^x)^y \mod (p)\right)$ for the recipient based on S and y, which are passed as command line arguments in this order. The key is output analogously to the server (example output):

```
d319bbb924009b66[...]
```

Note that the session key k **must** be identical for the server and the client. This can only be checked by an observer who sees the keys on both sides.

- `ServerEncrypt` to encrypt a message with the (previously and separately generated) session key. The message as text as well as the session key in hexadecimal representation are to be passed as command line arguments in exactly this order. The encrypted message is output to `std::cout` in hexadecimal representation. Note that **only** the passed message is to be encrypted, but no additional data.
- `ClientDecrypt` to decrypt a message with the (previously and separately generated) session key. The encrypted message as well as the session key, both in hexadecimal representation, are to be passed as command line arguments in exactly this order. The decrypted message is output as text to `std::cout`.

Example invocations:

- `ServerGeneratePartialKey`
- `ClientGeneratePartialKey`
- `ServerGenerateSessionKey 2826535490322092[...] 1559258775283944[...]`
- `ClientGenerateSessionKey 1311271927357378[...] 1175596360350942[...]`
- `ServerEncrypt Hallo d319bbb924009b66[...]`
- `ClientDecrypt e7e25195d4bcff8c[...] d319bbb924009b66[...]`

Hints: To generate random numbers using GMP, first call `gmp_randinit_default(prng_state);` and then `gmp_randseed_ui(prng_state, time(nullptr));` in order to initialize the pseudo-random number generator with your system's time. Subsequently, use the `mpz_urandomm` function to generate the actual (pseudo-)random number. Use the following values for g and p (adopted from `https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html`):

```
const char * const g_as_text = "\
  f7e1a085d69b3ddecbbcab5c36b857b97994afbbfa3aea82f\
  9574c0b3d0782675159578ebad4594fe67107108180b44916\
  7123e84c281613b7cf09328cc8a6e13c167a8b547c8d28e0a\
  3ae1e2bb3a675916ea37f0bfa213562f1fb627a01243bcca4\
  f1bea8519089a883dfe15ae59f06928b665e807b552564014\
  c3bfecf492a";
const mpz_class g(g_as_text, 16);

const char * const p_as_text = "\
  fd7f53811d75122952df4a9c2eece4e7f611b7523cef4400c\
  31e3f80b6512669455d402251fb593d8d58fabfc5f5ba30f6\
  cb9b556cd7813b801d346ff26660b76b9950a5a49f9fe8047\
  b1022c24fbba9d7feb7c61bf83b57e7c6a8a6150f04fb83f6\
  d3c51ec3023554135a169132f675f3ae2b61d72aeff222031\
  99dd14801c7";
const mpz_class p(p_as_text, 16);
```

In order to reduce the 1,024-bit long result $(g^x)^y \equiv (g^y)^x \mod (p)$ to 256 bits, i.e., to k, compute the SHA-256 hash of the result's ASCII representation (`mpz_class::get_str()`) as in LB-S 00. a).
Before encryption and decryption, use the `HexStringToArray` function from example LB-S 03. To encrypt and decrypt messages, respectively, use AES-256 in GCM mode from libsodium*. The documentation at `https://doc.libsodium.org/secret-key_cryptography/aead/aes-256-gcm` describes authenticated encryption which additionally performs a verification during decryption. For encryption and decryption, use `ad = nullptr`, `adlen = 0` and a zero nonce as follows:*

```
const unsigned char nonce[crypto_aead_aes256gcm_NPUBBYTES] =
    ↪ {0};
```

***Warning**: **Never** use a constant nonce in real-world applications! It is only used here in order to simplify the implementation.*

LB-HE 02. (not to be submitted)

Form groups of two as determined by the lecturers, where one person plays the role of the server and the other person correspondingly plays the role of the client. Verify the correctness and interoperability of your programs from example 01. in two separate steps. First, exchange all data required for the key exchange via e-mail. Take care that you do not exchange data which need to stay secret in the Diffie-Hellman protocol. Then, send a message from the server to the client with the agreed-upon session key. Make sure that the client can decrypt the message correctly.

LB-HE 03. (not to be submitted)

Generate a self-signed X.509 certificate for yourself which contains the public key of an RSA key pair. Simultaneously (with the same command), the corresponding secret key is to be created (**without** line breaks!):

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.crt
-days 365
```

Adjust the output paths of the two files (`key.pem` and `cert.crt`) if necessary. Specify meaningful values for the prompted data and remember the entered password (*keyphrase*).

LB-HE 04. (*Code::Blocks* template *GMP, libsodium and SSL project*)

A client wants to authenticate itself to a server. To do so, a challenge-response-based protocol is used. The necessary steps for this are to be implemented.
Using the *GMP*, *libsodium* and the *SSL*, write a program which implements the following operations and can be invoked like `<operation> <operand 1> [<operand 2> [<operand 3>]]`:

- `ServerReadPublicKey` to read the public key $pk := (e, N)$ of an RSA cryptosystem from a certificate (e.g., the one created in example 03.), whose file path is passed as an argument. e and N are output to `std::cout` in **exactly** the following format (example output):

  ```
  Public key: (65537, 7059515099399582[...])
  ```

- `ClientReadPrivateKey` to read the secret key $sk := (d, N)$ of an RSA cryptosystem from a key file (e.g., the one created in example 03.), whose file path is passed as an argument. d and N are output to `std::cout` in **exactly** the following format (example output):

  ```
  Private key: (4697366898921479[...], 7059515099399582[...])
  ```

  The password (*keyphrase*) required for this is automatically prompted from the user via `std::cin` as long as the hints below are followed.

- `ServerCreateChallenge` to create a challenge $c := E_{pk}(r)$, where r is a random number between 0 and $N - 1$ created by the program. Here, N means the modulus N of the public key pk. e and N are to be passed as command line arguments in exactly this order and in decimal representation. c and r are output to `std::cout` in **exactly** the following format (example output):

  ```
  Challenge: 4766680102085249[...]
  Random number: 6688689096349587[...]
  ```

- `ClientCreateResponse` to create a response $h := H(D_{sk}(c))$, where c denotes the challenge from the previous step and H denotes the cryptographic hash function SHA-512 which is applied to the decimal (base-10) representation of the decrypted challenge. c, d and N are to be passed as command line arguments in exactly this order and in decimal representation. h is output to `std::cout` in decimal representation.

- `ServerVerifyResponse` to compare the expected response $h' := H(r)$ to the actual response h. If h and h' match, `Authenticated successfully` is output to `std::cout`, `Authentication failed` otherwise. h and r are to be passed as command line arguments in exactly this order and in decimal representation.

Example invocations:

- `ServerReadPublicKey cert.crt`
- `ClientReadPrivateKey key.pem`
- `ServerCreateChallenge 65537 7059515099399582[...]`
- `ClientCreateResponse 4766680102085249[...] 4697366898921479[...]`
  `↪7059515099399582[...]`
- `ServerVerifyResponse 1262648800327937[...] 6688689096349587[...]`

Hints: Use as many code parts as possible from the previous exercises. Analogously to example 01, to create random numbers, use the `mpz_urandomm` function, where an upper bound can be specified.
The provided template project already contains a code file `certhelp.cpp` as well as the corresponding header file `certhelp.h` in which the two functions `ReadPublicKeyFromFile` and `ReadPrivateKeyFromFile` are contained to read keys in the first two steps of the protocol. The usage of the functions is self-explanatory.

LB-HE 05. (not to be submitted)

Form groups of two as determined by the lecturers, where one person plays the role of the server and the other person correspondingly plays the role of the client. Verify the correctness and interoperability of your programs from example 04. with the respective protocol steps. Exchange the necessary data via e-mail. In doing so, take care that no data is exchanged which would compromise the security of the challenge-response scheme.