Aufgabenblatt Hybride Verschlüsselung

Lösen Sie die nachfolgenden Aufgaben und bereiten Sie diese bis zum nächsten Lehrveranstaltungstermin vor.

LB-HV 01. (*Code::Blocks*-Template *GMP and libsodium project*)

Ein Server (Sender) möchte eine verschlüsselte Nachricht an einen Client (Empfänger) verschicken. Dabei soll aus Performancegründen ein symmetrisches Verschlüsselungsverfahren, konkret AES, verwendet werden. Um den Schlüssel zwischen den beiden Kommunikationsteilnehmern vorab auszutauschen, wird Diffie-Hellman als Schlüsseltauschverfahren verwendet. Von der Schlüsselgenerierung bis zur symmetrischen Verschlüsselung sollen alle Schritte umgesetzt werden.
Schreiben Sie dazu ein Programm unter Verwendung der *GMP* und der *libsodium* (vgl. S 01.), das die folgenden Operationen implementiert und sich nach dem Muster `<Operation> [<Operand 1> <Operand 2>]` aufrufen lässt:

- `ServerGeneratePartialKey` zum Erzeugen einer Zahl $x \in \mathbb{Z}_p^*$ und zum anschließenden Berechnen des Teilschlüssels $S := g^x \mod (p)$. Die Ausgabe von x und S erfolgt in dezimaler Darstellung über `std::cout` in **exakt** folgendem Format (Beispielausgabe):

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

- `ClientGeneratePartialKey` zum Erzeugen einer Zahl $y \in \mathbb{Z}_p^*$ und zum anschließenden Berechnen des Teilschlüssels $C := g^y \mod (p)$. Die Ausgabe von y und C erfolgt analog zu der des Servers (Beispielausgabe):

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

  Beachten Sie, dass S und C paarweise verschieden sein **müssen**, d.h. $g^x \not\equiv g^y \mod (p)$.

- `ServerGenerateSessionKey` zum Berechnen des 256 Bit langen Sitzungsschlüssels $k := (C^x \equiv (g^y)^x \mod (p)) \mod \left(2^{256}\right)$ für den Sender basierend auf C und x, die in dieser Reihenfolge als Kommandozeilenparameter angegeben sind. Die Ausgabe des Schlüssels erfolgt in hexadezimaler Darstellung über `std::cout` in **exakt** folgendem Format (Beispielausgabe):

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

- `ClientGenerateSessionKey` zum Berechnen des 256 Bit langen Sitzungsschlüssels $k := (S^y \equiv (g^x)^y \mod (p)) \mod \left(2^{256}\right)$ für den Sender basierend auf S und y, die in dieser Reihenfolge als Kommandozeilenparameter angegeben sind. Die Ausgabe des Schlüssels erfolgt analog zu der des Servers (Beispielausgabe):

`d319bbb924009b66[...]`

Beachten Sie, der Sitzungsschlüssel k bei Server und Client identisch sein **muss**.

- `ServerEncrypt` zum Verschlüsseln einer Nachricht mit dem (zuvor separat generierten) Sitzungsschlüssel. Die Nachricht als Text sowie der Sitzungsschlüssel in Hexadezimaldarstellung sind als Kommandozeilenparameter in exakt dieser Reihenfolge zu übergeben. Die verschlüsselte Nachricht wird in Hexadezimaldarstellung auf `std::cout` ausgegeben. Beachten Sie, dass **nur** die übergebene Nachricht verschlüsselt werden soll, nicht aber zusätzliche Daten.
- `ClientDecrypt` zum Entschlüsseln einer Nachricht mit dem (zuvor separat generierten) Sitzungsschlüssel. Die verschlüsselte Nachricht als Text sowie der Sitzungsschlüssel in Hexadezimaldarstellung sind als Kommandozeilenparameter in exakt dieser Reihenfolge zu übergeben. Die entschlüsselte Nachricht wird als Text auf `std::cout` ausgegeben.

Beispielaufrufe:

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

Hinweise: Um voneinander verschiedene Zufallszahlen mittels der GMP *zu erzeugen, rufen Sie nach* `gmp_randinit_default(prng_state);` *die Funktion* `gmp_randseed_ui(prng_state, time(NULL));` *auf, die den Pseudozufallszahlengenerator mit der aktuellen Systemzeit initialisiert. Verwenden Sie anschließend* `mpz_urandomm` *zum Generieren der eigentlichen Zufallszahl. Verwenden Sie für g und p die folgenden Werte (übernommen von* `https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html`*:*

```
1  const char * const g_as_text = "\
2    f7e1a085d69b3ddecbbcab5c36b857b97994afbbfa3aea82f\
3    9574c0b3d0782675159578ebad4594fe67107108180b44916\
4    7123e84c281613b7cf09328cc8a6e13c167a8b547c8d28e0a\
5    3ae1e2bb3a675916ea37f0bfa213562f1fb627a01243bcca4\
6    f1bea8519089a883dfe15ae59f06928b665e807b552564014\
7    c3bfecf492a";
8  const mpz_class g(g_as_text, 16);
9
10 const char * const p_as_text = "\
```

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

Um das 1024 Bit lange Ergebnis $(g^x)^y \equiv (g^y)^x \mod (p)$ *auf 256 Bit, d.h. k, zu reduzieren, verwenden Sie folgende Funktion oder eine Variation davon:*

```
#include <sstream>

std::string ExtractKey(const mpz_class &g_xy_mod_p)
{
  mpz_class two_256;
  mpz_ui_pow_ui(two_256.get_mpz_t(), 2, 256);
  mpz_class g_xy_short;
  mpz_mod(g_xy_short.get_mpz_t(), g_xy_mod_p.get_mpz_t(),
    ↪ two_256.get_mpz_t());
  std::stringstream s;
  s << std::hex << g_xy_short << std::endl;
  return s.str();
}
```

Verwenden Sie vor der Ver- und Entschlüsselung die Funktion `HexStringToArray` *aus Beispiel 03. Um Nachrichten zu ver- bzw. entschlüsseln, verwenden Sie AES-256 im GCM-Modus aus der* libsodium. *Die Dokumentation unter* `https://download.libsodium.org/doc/secret-key_cryptography/aes-256-gcm.html` *beschreibt authentifizierte Verschlüsselung, die zusätzlich eine Verifikation bei der Entschlüsselung durchführt. Für die Ver- und Entschlüsselung verwenden Sie dabei* `ad = NULL`, `adlen = 0` *und eine Null-Nonce wie folgt:*

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

***Achtung**: Verwenden Sie in realen Anwendungen **niemals** eine Konstante als Nonce! Hier wird sie nur verwendet, um die Implementierung zu vereinfachen.*

LB-HV 02. (nicht abzugeben)

Schließen Sie sich zu den von den Lehrveranstaltungsleitern bestimmten Zweiergruppen zusammen, wobei je eine Person die Rolle des Servers und die andere entsprechend die des Clients übernimmt. Überprüfen Sie die Korrektheit und Interoperabilität Ihrer Programme aus Beispiel 01. in zwei separaten Schritten. Tauschen Sie zuerst per Email alle für den Schlüsseltausch notwendigen Daten aus. Beachten Sie, dass Sie dabei keine Daten austauschen, die im Protokoll nach Diffie-Hellman geheim gehalten werden müssen. Senden Sie danach mit dem vereinbarten Sitzungsschlüssel eine Nachricht vom Server an den Client. Stellen Sie sicher, dass diese vom Client korrekt entschlüsselt werden kann.

LB-HV 03. (nicht abzugeben)

Generieren Sie für Ihre Person ein selbst signiertes X.509-Zertifikat, das den öffentlichen Schlüssel eines RSA-Schlüsselpaares enthält. Gleichzeitig soll der dazugehörige private Schlüssel (mit demselben Befehl) erzeugt werden (**ohne** Zeilenumbrüche!):

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

Passen Sie bei Bedarf die Ausgabepfade der beiden Dateien (`key.pem` und `cert.crt`) an. Geben Sie sinnvolle Werte für die abgefragten Daten ein und merken Sie sich das eingegebene Passwort (*keyphrase*).

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

Ein Client möchte sich gegenüber einem Server authentifizieren. Dazu wird ein zertifikatsbasiertes Challenge-Response-Protokoll verwendet. Die dafür notwendigen Schritte sollen implementiert werden.
Schreiben Sie dazu ein Programm unter Verwendung der *GMP*, der *libsodium* und der *SSL*, das die folgenden Operationen implementiert und sich nach dem Muster `<Operation> <Operand 1> [<Operand 2> [<Operand 3>]]` aufrufen lässt:

- `ServerReadPublicKey` zum Auslesen des öffentlichen Schlüssels $pk := (e, N)$ eines RSA-Verschlüsselungssystems aus einem Zertifikat (z.B. das in Beispiel 06. erzeugte), dessen Dateipfad als Argument übergeben wird. Die Ausgabe von e und N erfolgt über `std::cout` in **exakt** folgendem Format (Beispielausgabe):

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

- `ClientReadPrivateKey` zum Auslesen des privaten Schlüssels $sk := (d, N)$ eines RSA-Verschlüsselungssystems aus einer Schlüsseldatei (z.B. der in Beispiel 03. erzeugten), deren Dateipfad als Argument übergeben wird. Die Ausgabe von d und N erfolgt über `std::cout` in **exakt** folgendem Format (Beispielausgabe):

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

  Das dazu notwendige Passwort (*keyphrase*) wird automatisch vom Benutzer über `std::cin` abgefragt, sofern die Hinweise unten befolgt werden.

- `ServerCreateChallenge` zum Erzeugen einer Challenge $c := E_{pk}(r)$, wobei r eine vom Programm erzeugte Zufallszahl zwischen 0 und $N-1$ ist. N meint dabei den Modulus N des öffentlichen Schlüssels pk. e und N sind in Dezimaldarstellung als Kommandozeilenparameter in exakt dieser Reihenfolge zu übergeben. Die Ausgabe von c und r erfolgt über `std::cout` in **exakt** folgendem Format (Beispielausgabe):

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

- `ClientCreateResponse` zum Erzeugen einer Response $h := H(D_{sk}(c))$, wobei c die Challenge aus dem vorherigen Schritt ist und H die kryptografische Hashfunktion SHA-512 bezeichnet. c, d und N sind in Dezimaldarstellung als Kommandozeilenparameter in exakt dieser Reihenfolge zu übergeben. h wird in Dezimaldarstellung auf `std::cout` ausgegeben.
- `ServerVerifyChallenge` zum Vergleichen der erwarteten Response $h' := H(r)$ mit der tatsächlichen Response h. Stimmen h und h' überein, wird `Authenticated successfully` auf `std::cout` ausgegeben, ansonsten `Authentication failed`. h und r sind in Dezimaldarstellung als Kommandozeilenparameter in exakt dieser Reihenfolge zu übergeben.

Beispielaufrufe:

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

Hinweise: Verwenden Sie so viele Codeteile wie möglich aus den vorherigen Aufgaben wieder. Zum Erzeugen von Zufallszahlen verwenden Sie analog zu Beispiel 01. die Funktion ***`mpz_urandomm`****, bei der eine obere Grenze angegeben werden kann.*
Das bereitgestellte Musterprojekt enthält bereits die Codedatei ***`certhelp.cpp`*** *sowie die dazugehörige Headerdatei* ***`certhelp.h`****, in denen die beiden Funktionen* ***`ReadPublicKeyFromFile`*** *und* ***`ReadPrivateKeyFromFile`*** *zum Auslesen der Schlüssel in den ersten beiden Schritten des Protokolls enthalten sind. Die Verwendung der Funktionen ist selbsterklärend.*

LB-HV 05. (nicht abzugeben)

Schließen Sie sich zu den von den Lehrveranstaltungsleitern bestimmten Zweiergruppen zusammen, wobei je eine Person die Rolle des Servers und die andere entsprechend die des Clients übernimmt. Überprüfen Sie die Korrektheit und Interoperabilität Ihrer Programme aus Beispiel 04. mit den jeweiligen Protokollschritten. Tauschen Sie die notwendigen Daten per Email aus. Beachten Sie dabei, dass kein Datum ausgetauscht wird, das die Sicherheit des Challenge-Response-Verfahrens kompromittiert.