Aufgabenblatt #2: Public Key Cryptography (15 Punkte)

2.1 RSA (6 Punkte)

In dieser Aufgabe sollen Sie die Grundlagen von RSA anhand kleiner Zahlen verstehen und nachprogrammieren. Für größere Zahlen dürfen Ihre Programme Fehler produzieren (Stichwort Integer-Overflow). Nutzen Sie als Programmiersprache C; binden Sie keine mathematischen oder kryptographischen Bibliotheken ein (bspw. kein `<math.h>` und kein `<crypt.h>`).

- Schreiben Sie ein Programm `rsa_genkey`, das zu zwei gegeben Primzahlen den öffentlichen und den privaten Schlüssel generiert:

  ```
  ./rsa_genkey <p> <q>

  ./rsa_genkey 17 19
  Public  key (e,n) = (43,323)
  Private key (d,n) = (67,323)
  ```

  Prüfen Sie, dass die eingebenen Zahlen p und q wirklich Primzahlen sind. Berechnen Sie dann n und $\phi(n)$ und wählen Sie ein zu $\phi(n)$ teilerfremdes e, das in der Größenordnung 43 liegt. Berechnen Sie als letztes d – das Inverse zu e modulo $\phi(n)$. Der private Schlüssel entspricht (e, n), der öffentliche (d, n). (3 Punkte)

- Schreiben Sie nun ein Programm `rsa_crypt`, welches eine Nachricht m und einen öffentlichen (bzw. privaten) Schlüssel erwartet und die Nachricht verschlüsselt (bzw. entschlüsselt):

  ```
  ./rsa_crypt enc <m> <e> <n>
  ./rsa_crypt dec <m> <d> <n>

  ./rsa_crypt enc 219 43 323
  281
  ./rsa_crypt dec 281 67 323
  219
  ```

  Berechnen Sie für die Verschlüsselung m^e mod n und für die Entschlüsselung m^d mod n. Stellen Sie sicher, dass die Nachricht m kleiner dem Modulus n ist und nutzen Sie zum Exponentieren die *Square-and-Multiply* Methode. (1 Punkt)

- RSA mit so kleinen Zahlen zu verwenden ist unsicher, da der private Schlüssel mit Hilfe von Primfaktorzerlegung schnell aus dem öffentlichen gewonnen werden kann. Schreiben Sie ein Programm `rsa_crack` welches genau das tut:

  ```
  ./rsa_crack <e> <n>

  ./rsa_crack 43 323
  Private key (d,n) = (67,323)
  ```

  Wie lauten die privaten Schlüssel zu $(23, 1517)$, $(43, 302303)$ und $(41, 147807041)$? (2 Punkte)

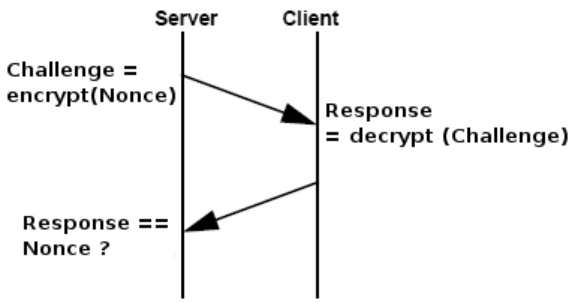
Anhang: Testläufe zu etwas größeren Zahlen:

```
./rsa_genkey 197 571
Public  key (e,n) = (43,112487)
Private key (d,n) = (18187,112487)
./rsa_crypt enc 1234 43 112487
64816
./rsa_crypt dec 64816 18187 112487
1234
./rsa_crack 43 112487
Private key (d,n) = (18187,112487)
```

```
./rsa_genkey 31321 32141
Public  key (e,n) = (43,1006688261)
Private key (d,n) = (327738307,1006688261)
./rsa_crypt enc 909090909 43 1006688261
122832246
./rsa_crypt dec 122832246 327738307 1006688261
909090909
./rsa_crack 43 1006688261
Private key (d,n) = (327738307,1006688261)
```

2.2 Challenge / Response (6 Punkte)

- Implementieren Sie eine Challenge-Response Authentifizierung, basierend auf Public/Private-Key Kryptographie, die folgendes Protokoll umsetzt (5 Punkte):



1. Server: Generiere eine zufällige Nonce und verschlüssel diese mit dem öffentlichen Schlüssel des Client. Kodiere diese *Challenge* mit Base64 und schicke sie an den Client.
2. Client: Entschlüssel die Nonce mit dem eigenen privaten Schlüssel und schicke diesen *Response* zurück an den Server (ebenfalls Base64 kodiert).
3. Server: Überprüfe ob Nonce und Response übereinstimmen.

  Client und Server müssen in Ihrer Lösung nicht Netzwerk-fähig sein; Sie können Challenge bzw. Response per Hand wie folgt eintragen:

```
server:~$ ./server pubkey.pem
Please challenge the following BASE64 string:
erq...btA=
Enter your response and press Ctrl-D when finished:
ZWg...8Ug=
Done reading input.
Challenge successful, proceeding ...

client:~$ ./client privkey.pem
Enter PEM pass phrase:
Enter the challenge, when finished, press Ctrl-D:
erq...btA=
Done reading input.
The Response is:
ZWg...8Ug=
```

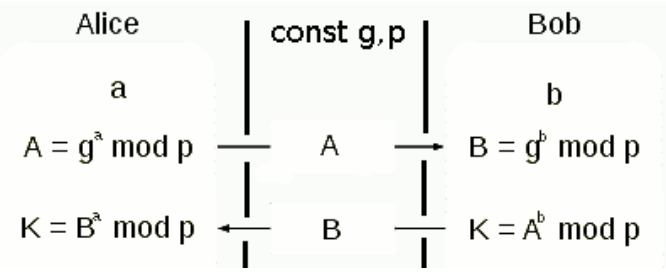
  Privater und öffentlicher Schlüssel sollen jeweils echte RSA-Keys sein (nicht wie in Aufgabe 1) und aus den Dateien `privkey.pem` und `pubkey.pem` gelesen werden, die wie folgt erstellt wurden:

```
$ openssl genrsa -out privkey.pem -aes256 1024
$ openssl rsa -in privkey.pem -pubout -out pubkey.pem
```

- Das von Ihnen implementierte Verfahren authentifiziert nur den Client gegenüber dem Server – es authentifiziert nicht (wie bspw. bei SSL) den Server gegenüber dem Client. Darüber hinaus wird die Nonce im Klartext übermittelt, so dass Client und Server kein Geheimnis teilen das als symmetrischer Schlüssel zur weiteren Kommunikation dienen könnte. Entwickeln Sie, basierend auf einem zusätzlichen Schlüsselpaar des Servers, ein Protokoll das beide Missstände behebt. Achten Sie darauf, dass Ihr Protokoll sicher gegen *Reflection Attacks*, *Replay Attacks* und *Man-in-the-Middle Attacks* ist. Sie müssen dieses Protokoll nicht implementieren; eine Beschreibung genügt.
  (1 Punkt)

2.3 Diffie Hellman (3 Punkte)

In Aufgabe 2b haben Sie ein Protokoll zum Schlüsselaustausch entwickelt, das auf RSA basiert und ein öffentlich/privates Schlüsselpaar der Parteien voraussetzt. Im Gegensatz dazu bietet das Diffie-Hellman Verfahren eine Möglichkeit zum Schlüsselaustausch ohne zuvor bekanntes Schlüsselpaar.



- Implementieren Sie zwei Programme `alice` und `bob`, die einen Diffie-Hellman Schlüsselaustausch mit kleinen Zahlen durchführen. (1 Punkt)

  Die Parameter $p = 29$ und $g = 11$ können Sie als konstant voraussetzen; die Parameter a und b sollen zufällig generiert werden. Für die Werte $a = 5$ und $b = 8$ sollen die Programme bspw. folgende Ausgabe liefern:

```
$ ./alice                    $ ./bob
A: 14                        A> 14
B> 16                        B: 16
>> Secret key: 23            >> Secret key: 23
```

  Optional soll `alice` mit den Parametern `<a> <B>` und `bob` mit den Parametern `<b> <A>` aufgerufen werden können.

- Angenommen die Programme `alice` und `bob` übertragen die Werte $A = 24$ und $B = 19$. Wie lautet in diesem Fall der geheime Schlüssel? Wieso ist es im Allgemeinen problematisch den Schlüssel nur anhand der übertragen Werte zu bestimmen? (1 Punkt)

- Diffie-Hellman alleine kann nicht zur Authentifizierung genutzt werden und ist anfällig gegenüber Man-in-the-Middle Attacken. D.h. Diffie-Hellman bietet Schutz gegenüber Abhören, nicht aber gegenüber Manipulation. Skizzieren Sie solch einen Man-in-the-Middle Angriff. (1 Punkt)

2.4 Zusatzaufgaben (5 Punkte)

- Faktorisieren Sie den RSA Modulus

```
n = 15632280012491008761379186122424208246024086583917\
    60932600826416514228125349095495238601388872204263
```

  in zwei Primfaktoren p und q. (2.5 Punkte)

- Implementieren Sie Ihr Challenge/Response Protokoll aus Aufgabe 2b. Nach der Authentifizierung sollen Client und Server verschlüsselt über AES kommunizieren können. Machen Sie Ihre Implementierung Netzwerk-fähig. (2.5 Punkte)

Deadline: 17.12.2010