# Information Security Exercises

## Chapter 3: Set three: Encryption Part II. Deadline: October 2, 11:00 AM

Document version: 1.2

---

*You can earn 7 points for this set of exercises: Exercise 16 is worth 2 points. The other exercises are worth 1 point each.*
**Notes:**

- **Don't forget to submit your source code for all of the programming exercises (both as separate attachment so we can run it, and in the report so we can give feedback).**
- Computing a SHA256 hash is possible using either the built-in cryptography capabililities of your language of choice, or, if it does not have it, by calling into the 'sha256sum' command-line program that is available on Linux systems, `shasum -a 256` on Mac. Another alternative is to call into the Foreign-Function-Interface of OpenSSL. You don't have to implement it yourself.

---

**Exercise 11.**
Purpose: design a program computing large powers modulo N

Use a recursive procedure to compute large powers modulo N using squares and simple multiplications (as explained in the (hidden) slides).

As an illustration of the program, use it to compute $43210^{23456}$ % 99987. *(Don't forget to include your program's source code in the report!)*

---

**Exercise 12.**
Purpose of this exercise: learn to use the RSA encryption method

Assume an RSA cipher uses the primes 31 and 41, and uses e = 7;

- What is the public key, what is the private key?
- If the message is 42, what is the encrypted message?
- What does the receiver have to to do retrieve 42 from the received encrypted message? Show the steps to retrieve the unencrypted message.

---

**Exercise 13.**
Purpose: Find generator(s) for a prime `p`.

The Diffie-Hellman key exchange procedure requires you to have a *generator* for a prime number `p`. Implement a function (or class) expecting a prime number `p` and returning a set of generators (`% p`). It's nice if the set contains all generators for the provided prime number, but a subset is also OK.

To find a generator $g$ % $p$ the following algorithm can be used (cf. Schneier, B. (1996) Applied Cryptography, p. 253):

- Compute the set of prime factors `{q}` of (p - 1);
- For each of the values `x in {q}` compute for each potential generator `g` in `2 .. p - 1` compute $g^{(p-1)/x}$ % $p$;
- If at least one of the previous expressions equals 1, g is *not* a generator (% p).

The 1000th prime number equals 7919. What is the largest generator for this prime using Schneier's algorithm?

What are all generators of 23?

---

**Exercise 14.**

Purpose: use ECC with DH key exchange: part I: ECC final point computation

This exercise requires an e-mail exchange with the TAs. Your e-mail to the TAs must have been sent no later than one full day before this exercise's deadline (or earlier if you want to be able to send in a second attempt). Also, be sure to react in time to a reply from the TAs so that the exercise is completed before its deadline.

Send the TAs a text file called `knapsack.key` as an e-mail attachment containing the the a, b, N, initial point's X and Y coordinates, and the final point you obtained after computing `m * (x, y)` for the provided elliptic curve, where `m` is the multiplier of your choice (so you don't send `m` to the TAs!). Each value must be submitted on a separate line, so you send a file with the values of these seven variabeles, one per line, to the TAs:

```
a
b
```

```
N
X1
Y1
Xm
Ym
```

You then receive from the TAs the Xn and Yn coordinate of his point on the elliptic curve (two values) and an encrypted text. The assistant's Xn and Yn coordinates are mentioned in the e-mail text, the encrypted message is stored in a zip, containing `message.enc` which is a binary file.

The encrypted text is not used in this exercise, but is used below, in one of the later exercises: you can ignore it if you're not completing that exercise.

For this exercise, compute the final (shared) point on the elliptic curve (i.e., `m * (Xn, Yn)`) write it like this: '`(X,Y)`' (no blanks, the quotes are not part of the representation, no leading zeros).

Submit the final shared point as answer to this exercise in your report.

---

**Exercise 15.**
Purpose: Learn to speed up multiplication by addition with ECC, using a binary representation of the multiplication factors.

In the context of ECC the parameters `a = 10, b = -21, p = 41, P1 = (3,6)` are given. Assuming that Alice uses multiplier 44 and Bob uses multiplier 57 their shared secret will be the x-coordinate of 44 * 57 * P1.

The x-coordinate of what point will they use?

What point is sent by Alice to Bob and what point is sent by Bob to Alice?

How many additions are required to compute 57 * P1?

Explain why this is the case.

---

**Exercise 16.** *This exercise is worth 2 points.*
Purpose: use ECC DH key exchange.

This exercise requires an e-mail exchange with the TAs. Your e-mail to the TAs must have been sent no later than one full day before this exercise's deadline (or earlier if you want to be able to send in a second attempt). Also, be sure to react in time to a reply from the TAs so that the exercise is completed before its deadline.

The result of one of the previous exercises (exercise 14), is a shared ECC point. This shared point on the elliptic curve should be written like this:'`(X,Y)`' (no blanks, the quotes are not part of the representation, no leading zeros). In this

form it is passed to the sha256sum program (note: do not append an end-of-line character) to obtain hexadecimal digits. E.g., for `(123,456)` you get the hash value

```
f262443bc02828d434b09edcb59ea730c800bc8f9b69bbeb676be6ad77430137
```

(64 hexadecimal digits). These 64 characters are used to determine the round-keys in the Feistel part of a CBC block cipher algorithm. Here, the least significant value is the leftmost character (i.e., f), while the rightmost character (i.e., 7) is the most significant value.

Your CBC block cipher must work like this:

- Use a block size of 16 bytes. The byte that is read first is stored at index 0, the byte that's read last is stored at index 15. When splitting a block into halves, indices 0 through 7 are the RH, indices 8 through 15 are the LH.

  So, if the frst 16 bytes of your file have the values 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, then the first 8 bytes receive the lowest 8 indices in a buffer, becoming the block's RH, and the remaining 8 bytes become the block's LH. In fact a simple `read` instruction of 16 bytes will store the bytes in the correct way, with indices 0..7 being the RH, and indices 8..15 being the LH.

- The key schedule uses a Feistel algorithm of 4 rounds, to compute the next round's RH. Note that you're supposed to *decrypt* the data using the Feistel algorithm.
  - At each round take the next 16 characters from the sha256 hash value to compute that round's key.
  - Two hexadecimal values fit into one byte, so the least significant byte (LSB) of the first key (using the above sha256sum value) equals `0xf2`, and the most significant byte (MSB) equals `0xd4`.
  - Thus, the first key of 8 bytes becomes:

    ```
    0xf2, 0x62, 0x44, 0x3b, 0xc0, 0x28, 0x28, 0xd4,
    LSB                                       MSB
    ```

    For the subsequent rounds use the subsequent sets of 16 characters.
  - At each round xor the key for that round with the current RH and the LH to produce the next round's RH: `RH2 = (RH1 ^ KEY1 ^ LH1)`
  - The next round's LH is (standard Feistel) the current round's RH (LH2 = RH1)

- Using the keys as described, blocks of 16 bytes were encrypted using the CBC block cipher mode. A random initialization vector was used, which you receive as the 1st set of 16 bytes of the encrypted message.
- Use a standard padding rule for the final block: if the last block is b bytes long, and k is the length of the key, then pad the final b - k bytes with the byte value b - k (before doing the encryption). There must always be at least one byte of padding.

Submit the decrypted text as part of your report.