Homework #3 - Size matters: small subgroups

Deadline: 29.11.2022 23:59:59 Brno time

Points: 10 points

Responsible contact: Vojtěch Suchánek <vojtechsu@mail.muni.cz>

Submit: here

Background

As you know from the lecture, the Diffie-Hellman key exchange scheme allows two parties to securely establish a shared secret. Currently, the most popular variant of this scheme is the elliptic curve Diffie-Hellman (ECDH) which is based on the group of points of an elliptic curve. Recall that an elliptic curve is the set of solutions (x, y), called points, of an equation $y^2 = x^3 + ax + b$ for $x, y \in \mathbb{F}_p$. This set forms a group with the operation $P + Q$ for any two points P, Q as defined in the lecture. For cryptographic applications, for any two points P and $k \cdot P$ it should be unfeasible to compute the scalar k (the discrete logarithm problem).

We will consider the following version of ECDH between server and a client on a group of points with a generator G:

1. Server randomly samples a private scalar $k \in \{1, \ldots, |G| - 1\}$ and computes his public key as $P = k \cdot G$. Server then sends P to the client.
2. Client randomly samples a private scalar $l \in \{1, \ldots, |G| - 1\}$ and computes his public key as $Q = l \cdot G$. Client then sends Q to the server.
3. Server computes $k \cdot Q = S$. The shared secret is then the hash SHA-256($x|y$), where $S = (x, y)$ and $|$ denotes concatenation.
4. Client computes $l \cdot P = S$. The shared secret is then, again, the hash SHA-256($x|y$) where $S = (x, y)$.

Instructions

Petr has decided to implement ECDH on his server and selected the elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$, where

$p = \texttt{0x586be5268256ae12d62631efc2784d02dcff420d262da9cd94c62d5808bee2}$
$a = \texttt{0x7e7}$,
$b = \texttt{0x0}$.

The neutral element in the group of points is represented as $(0, 1)$. The curve has the number of points equal to $N = n_1 \cdot n_2$, where n_2 is a prime and

$$n_1 = 94025829692594460866289522123566443121 0,$$
$$n_2 = 42535295865117307932921825928971027169.$$

Petr heard that groups in cryptographic applications should have number of elements equal to prime and so he decided to only use the subgroup of points of size n_2. He figured that this will be more efficient anyway. For his ECDH implementation Petr picked a point G of order n_2 and a private scalar $k \in \{1, \ldots, n_2\}$. His public key is then $P = kG$. He implemented on his server a function which, given a point Q on the curve, outputs SHA-256($x|y$), where $(x, y) = kQ$. In real life scenario, the shared secret should not be revealed at the end, but rather used for further operations like key derivation and symmetric key encryption (whose output then might be revealed). In this homework, we will settle with this simplification.

$$\downarrow$$

$$k \cdot Q = (x_k, y_k)$$

$$\downarrow$$

$$\textbf{Output: } \text{SHA-256}(x_k | y_k)$$

Task 1 (7 points)

Your goal is to determine Petr's secret scalar k given access to his function running on his server. In order to do this, you should implement the so-called small subgroup attack. An explanation of certain variants of this attack can be found, for instance, here or here. The task is divided into two subtasks:

1. Find a generator of Petr's elliptic curve (i.e., a point of order $N = n_1 \cdot n_2$). You might find Chapter 4 of the Handbook of Applied Cryptography helpful (see course webpage for a link). **(2 points)**
2. In this scenario, implement the small subgroup attack **(3 points)**. Provide a short description of your attack **(2 points)**.

You can find a copy of an implementation of elliptic curve operations and the ECDH scheme running on Petr's server in `ec.py`. You might find useful, among others, an implementation of point addition (`ec.add`), scalar multiplication (`ec.rtl`) or the compression function (`ec.compress`).

Warning: Petr wrote the provided implementation in `ec.py` for his purposes only. While it correctly implements the computation specified above, it is not correct to re-use it for more general elliptic-curve computations. We recommend you use a foolproof implmentation of elliptic curve arithmetic from computer algebra software such as Sagemath or PARI/GP.

Note that you will be using the API functions Pubkey (easy) and ECDH (easy) for this task.

check in his function from the safecurves website:

$$\begin{tikzcd}
 & \textbf{Input}: Q \arrow[d] & \\
 & h \cdot Q = (x_h, y_h) \arrow[dl, "{(x_h, y_h) = (0,1)}"'] \arrow[dr, "{(x_h, y_h) \neq (0,1)}"] & \\
\textbf{Output}: \text{Invalid point} & & k \cdot Q = (x_k, y_k) \arrow[d] \\
 & & \textbf{Output}: \text{SHA-256}(x_k | y_k)
\end{tikzcd}$$

However, this recommended countermeasure does not work in this case. Your goal is, again, to determine Petr's secret scalar k using an appropriate modification of the small subgroup attack **(2 points)**. Provide a short description of your attack **(1 point)**.

Hint: You have to use Petr's public point P.

Note that you will be using the API functions Pubkey (hard) and ECDH (hard) for this task.

API

There are several API functions available, you will be using the `easy` functions for the first task and the `hard` functions for the second.

Pubkey (easy)

Response: JSON dictionary with two keys: `"x"` and `"y"`, containing the coordinates of Petr's public key in hexadecimal.

ECDH (easy)

Path: `/hw03/easy/ecdh/<uco>/`
Method: POST
Body: JSON (`application/json`) dictionary with two keys: `"x"` and `"y"` containing the coordinates of your chosen public key in hexadecimal.
Response: JSON dictionary with one key: `"secret"` containing the shared Diffie-Hellman secret in hexadecimal (i.e. the hash SHA-256($x|y$) where $S = (x, y)$).

Pubkey (hard)

Path: `/hw03/hard/pubkey/<uco>/`
Method: GET
Response: JSON dictionary with two keys: `"x"` and `"y"`, containing the coordinates of Petr's public key in hexadecimal.

ECDH (hard)

Path: `/hw03/hard/ecdh/<uco>/`
Method: POST
Body: JSON (`application/json`) dictionary with two keys: `"x"` and `"y"` containing the coordinates of your chosen public key in hexadecimal.
Response: JSON dictionary with one key: `"secret"` containing the shared

Submission

You should submit a zip-file containing three things:

- A **`solution.txt`** file that contains three lines:
  - x-coordinate and y-coordinate of the generator you found as decimal integers separated by a comma,
  - Petr's secret scalar from Task 1 as a decimal integer,
  - Petr's secret scalar from Task 2 as a decimal integer.
- A **`description.txt`** file that contains your description of how you solved the task (i.e. what you did in order to obtain the secret message, how you came up with it, etc.). Please keep the descriptions reasonably short and clear (4-5 paragraphs should be enough).
- A **code** directory that contains any source code used in solving the task.

Grading

Not conforming to the above format of the solution leads to a -0.5 point penalty.