Advanced Networks — Laboratory 9

Juliusz Chroboczek

22 April 2025

Opportunistic encryption consists in enrypting data without authenticating the peer. Opportunistic encryption is vulnerable to man-in-the-middle (MiTM) attacks, but it effectively and cheaply prevents passive attacks.

The goal of this lab is to implement a TCP client that performs an opportunistic Diffie-Hellman exchange and then receives an encrypted message. These techniques are easy to adapt to UDP: it will be enough to manually implement reliable communication of the Diffie-Hellman key exchange.

The protocol proposed in this lab does not conform to current best practices:

- it performs a Diffie-Hellman exchange on 768-bit integers, while at least 2048 bits should be used in 2025;
- it performs a Diffie-Hellman exchange on a modular group, one would use an elliptic curve group nowadays.

Exercice 1 (Preliminary questions).

1. Opportunistic encryption is sometimes called *better than nothing cryptography (BTN)*. What are the weaknesses of opportunistic encryption? Why is it still useful?
2. When a HTTPS server's certificate cannot be validated, the browser displays a big red scary warning; it does not display a warning when connecting to an unencrypted HTTP server. Opinions? (You are exceptionally allowed to develop a conspiracy theory.)
3. What are the advantages of ECDH over DH?

Exercice 2 (Diffie-Hellman key exchange). Run a local copy of the supplied TCP server with the option `-verbose`. Write a program that connects to the server then:

- draws a ranom string of 768 bits and converts it into an integer $a < 2^{786}$ (use the functions `crypto/rand.Read` and the method `SetBytes` of the type `math/big.Int`);
- computes $A = g^a \mod p$ (the values p and g are given in the fil supplied);
- sends A to the server, as a string of 768/8 bytes;
- receives a string of 768/8 bytes from the server, which it interprets as an integer $B < 2^{768}$;
- verifies that B is not a trivial element of the group $\mathbf{Z}/p\mathbf{Z}$ (the trivial elements are 0, 1 and $p - 1$);
- computes the integer $s = B^a \mod p$.

Verify at each step that your program produces the same values as thee server (put `Printf` statements all over the place).

Exercice 3 (Encryption)**.** The value s computed by your program is shared between the client and the server and is not known to a passive observer; it can therefore be used to generate an opportunistic encryption key.

We cannot use value s directly as an input to a block cipher, for at least two reasons. First of all, block ciphers take a key of a fixed size, which is not necessarily equal to 768/8; we reduce the size of the key using a hashing function.

Second, using the same key with multiple messages would allow a passive observer to detect that two messages are identical. To avoid this, we combine the key with a random *initialization vector* (IV), which is transmitted in clear over the socket.

After the Diffie-Hellman key exchange has competed, the server sends::

- 16 random bytes that serve as an initialization vector (IV);
- the ciphertext.

It then closes the connection. (Which is bad practice: the server should be using a proper protocol based on TLVs rather than relying on a transport-layer indication to determine the end of the data. Oh, well.)

Modify your program so that, after the Diffie-Hellman key exchange, it:

- computes $h = \mathrm{SHA256}([s])$, where $[s]$ is the value of s represented as a string of bytes;
- sets k to be the first 16 bytes of h; k will be the shared key;
- reads 16 bytes, which will serve as the initialization vector (IV);
- reads the remainder of the data sent by the server; this is the cyphertext;
- decrypts the ciphertext using the AES-128 block cipher in CTR mode with the key and IV obtained above, and displays the result as a string.