CAA 2020

Lab #2

27-04-2020

1 Introduction

The goal of this lab is to implement a **proof of concept** of a **password-authenticated key exchange** (PAKE) to login on a company server.

- Please provide a **report** describing your choices. Describe also all the bonuses you implemented.
- Please provide also your **code** as well as a compilation script (if needed).
- Provide a small **guideline** on how to use your program (if needed).
- You do not need to do care about anything besides security. This means there is no need for the server to accept multiple incoming connections. Accepting one connection and closing the server when it is over is an acceptable behavior for this lab. Of course, if you want to do more, it is encouraged (see below).
- You do not need to use any complicated database system. A simple text file is enough for this lab.
- You can use any of these programming languages: C, C++, Java, Python, Sage (version ≥ 9), Rust, Go. If you want to do it in another language, please contact me. Give the relative complexity of the implementation, I recommend **Sage** (version ≥ 9).
- Since it is a proof of concept, we are going to ignore side-channel attacks (timing attacks, etc.) in this implementation.
- You are free to use any cryptographic library you want (except one doing the PAKE for you).
- All the implementations that are beyond proofs of concept will give **bonus points** on the **final grade**.

2 OPAQUE

You are working in the IT team of a small company who wants to authenticate users connecting to the server using a password. Your idea is to use a modern PAKE for this, and you naturally thought about OPAQUE. Since you care about speed and efficiency, you want to use the **HMQV** version (although patented) and apply it on **elliptic curves** (we will use ECDH for the oblivious PRF).

1. The OPAQUE protocol is, unfortunately, not very precise. You can find the details of the protocol in Figure 12, page 47 of https://eprint.iacr.org/2018/163.pdf. There are also some details in the following incomplete draft https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-03.

   The goal is to **authenticate** users arriving on the website. Knowing that you want to use the HMQV version and that you want to apply it on **elliptic curves**, describe the protocol as clearly as possible. Note that you **do not need to use** the derived symmetric key. You just need to authenticate users.

Describe, as precisely as possible, the algorithm you are going to implement. Don't forget to specify parameter sizes. Specify also clearly which primitive you choose (e.g. which MAC for the PRF). Given your specification, the implementation should be as straightforward as possible.

Hints/indications:

- This part will have a significant weight in the final grade (at least half of the points).
- The construction requires a PRF. You can use any (good) MAC for this purpose.
- At some point, you need to verify whether $\alpha \in G$, where G is the group generated by g. Since g has prime order q, it suffices to verify that α has order q. Indeed, for prime orders, all the elements of this order are in the same subgroup.
- The security parameter corresponds to the number of bits of a symmetric key. It fixes the security level of the construction. It is your job to select a good security parameter.
- For the elliptic curve choice, have a look at the secp standard http://www.secg.org/SEC2-Ver-1.0.pdf. In particular, focus on the "r" versions.
- Hash functions in the construction can either be normal hash functions (SHA2, SHA3,...) or password-hashing algorithms. Choose carefully which one you use and justify it in your report. You might need to use both.
- You will need a hash function H' that has the set generated by the generator g as an image. This is usually a complex operation.[1] For this lab, I will accept a simpler operation: $m \to g^{H(m)}$. Not that this is **not a secure implementation**. If you wish to implement the more complex operation, bonus points will be given.

2. Implement this in your favorite language. To simplify the implementation, everything can run on a single machine (localhost) on a fixed port. You do not have to take care of multiple clients. More precisely, here is the **minimal** behavior of both the client and the server:

   Server: Once launched, waits for a connection. Executes then the protocol and outputs (stdout) either OK/ERROR. The server application can stop after one execution.

   Client: Connect on the server socket. Asks password (stdin). Outputs (stdout) either OK/ERROR.

   Hints/indications:

   - You might encounter many serialization problems. I will accept dirty solutions there (for the POC). In particular, encrypting three mathematical elements with an authenticated encryption algorithm will make you think.
   - If you block on an element, ask me.
   - You are allowed to swear at your screen and even to curse me during this lab :-)

[1] See https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-06 for elliptic curves.