CAA – Lab 2

Nikolaos Garanis

May 24, 2020

Contents

1 Choice of Primitives

1.1 Elliptic Curve

I chose the `secp521r1` elliptic curve defined by secg.org. According to SECG, this elliptic curve's parameters are 521 bits in length and provide a security parameter of 256 bits (section 2.1 and table 1). According to keylength.com, this EC provides long-term security. As an attacker could store communications between the client and the server and try to decrypt it decades later, to obtain data that may still be sensible after that much time, I believe this EC is a good choice.

The elliptic curve `secp521r1` is denoted E and is defined over the finite field $\mathbb{F}_p$, where p is prime and is the order of a point G belonging to E.

In the protocol description below, when choosing a private and public key pair (p_x, P_x), it means that p_x is randomly chosen in $\mathbb{F}_n$ and that $P_x = p_x \times G$.

1.2 H and H'

The OPAQUE specification states that H must have a range of $\{0,1\}^{2t}$ where t is the security parameter. As I have chosen a security parameter of 256 bits, H must produce a hash of 512 bits. For this reason I have chosen SHA-512 as the hash function H.

H takes an arbitrary number of parameters which are concatenated before being hashed, i.e. $H(a, b, \dots) = \text{SHA-512}(a \parallel b \parallel \dots)$.

H' maps its input to a point on the EC. I defined it as $H' = H(m) \times G$ as permitted by the lab **although it does not provide a secure implementation**.

1.3 OPRF F_k

The OPRF (Oblivious Pseudorandom Function) is defined in the OPAQUE specification as $F_k(x) = H(x, (H'(x))^k)$.

1.4 PRF f_k

For the PRF I have chosen HMAC-SHA512 (HMAC was proved to be a PRF in 2006 by Mihir Bellare). Similar to H, the OPAQUE specification states that f_k must have a range of $\{0,1\}^{2t}$ where t is our security parameter, thus SHA-512 was chosen for the HMAC.

1.5 Authenticated Encryption

The OPAQUE draft proposes three possibilities for the AE. I chose AES-GCM because it requires only one key, contrary to the first two solutions. Also, as we've studied AES-GCM in the previous lab, it was interesting to put it in practice.

The draft requires the AE to be *random-key robust*. For this, for AES-GCM, 0s must be appended to the plaintext until its last block is composed only of 0s (here we're talking of 128-bit blocks as we're using AES). In order to distinguish between the plaintext and the padding, a first byte of `0x80` before the 0s.

1.6 e_u and e_s

The two variables e_u and e_s are used during the login phase and are defined as follows:

$$e_u = H(X_u, \text{server id}, ssid')$$
$$e_s = H(X_s, \text{user id}, ssid')$$

X_u and X_s are defined in the login phase. In this implementation, I chose the server's domain name as the server id and the user's username as the user id (which is unique among users).

2 Protocol description

The OPAQUE protocol is composed of two steps. In the first step, the user must register his password to the server. During the second step, the user logs in with the server to obtain a shared symmetric key between both parties.

2.1 Registration

In this step, the client registers his password to the server, without the server actually seeing the password in plaintext. This is done in the following way:

1. The **client** chooses a password pw and generates a private and public key pair (p_u, P_u). He chooses a random value $r \in \mathbb{F}_n$ and computes $\alpha = H'(pw) \times r$. He sends α to the server.

2. The **server** chooses a random key k_s for the OPRF (must be distinct for each user). He also generates a private and public key pair (p_s, P_s). He computes $\beta = k_s \times \alpha$ and sends it to the client along with P_s which the client will require.

3. The **client** computes $rw = F_{k_s}(pw)$ without knowing k_s. In order to do this, he computes $rw = H(pw, \beta^{1/r})$. The client then computes $c = \text{AuthEnc}_{rw}(p_u, P_u, P_s)$ and sends it to the server.

The client then erases pw, rw and all keys. In a database of its choice, the server stores (k_s, p_s, P_s, P_u, c) in an entry dedicated to the current client.

2.2 Login

The goal of this second step is to establish a shared symmetric key between the client and the server that will be used to encrypt all further communications between the two parties.

1. The **client** randomly chooses $r \in \mathbb{F}_n$ and computes $\alpha = H'(pw) \times r$. It also generates a private and public key pair (x_u, X_u). It sends α and X_u to the server.

2. The **server** makes sure α is on the curve or quits. It retrieves the values stored during the client's registration, namely k_s, p_s, P_s, P_u and c. It generates a private and public key pair (x_s, X_s) and computes:

   - $\beta = k_s \times \alpha$,
   - $K = H((X_u + e_u P_u) \times (x_s + e_s p_s))$,
   - $ssid' = H(sid, ssid, \alpha)$,
   - $SK = f_k(0, ssid')$ and
   - $A_s = f_k(1, ssid')$.

   It sends β, X_s, c and A_s to the client.

3. The **client** makes sure β is on the curve or quits. It computes the key $rw = H(pw, \beta^{1/r})$ and proceeds to decrypt c, i.e. $p_u, P_u, P_s = \text{AuthDec}_{rw}(c)$. The tag must be valid or else the client quits. The client computes:

   - $K = H((X_s + e_s P_s) \times (x_u + e_u p_u))$,
   - $ssid' = H(sid, ssid, \alpha)$,
   - $SK = f_k(0, ssid')$,
   - $A_s = f_k(1, ssid')$ and
   - $A_u = f_k(2, ssid')$.

   Before sending A_u to the server, the client must verify the A_s received is

the same as the A_s just computed or it must quit.

4. The **server** computes $A_u = f_k(2, ssid')$ and verifies it equals the A_u received from the client or else, the server quits.

   The client and the server now both have the same value for SK.