CAA – Lab 2

Nikolaos Garanis

May 24, 2020

Contents

1 Primitives

1.1 Elliptic Curve E

I chose the `secp521r1` elliptic curve (EC) 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, it 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 $\mathbb{F}_p$. A base point $G \in E$ of order n (which is prime) is also defined.

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 Hash functions 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 chose SHA-512. 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 $H'(x) = H(x) \times G$ as permitted by the lab **although it does not provide a secure implementation**.

1.3 Oblivious Pseudorandom Function F_k

The Oblivious Pseudorandom Function (OPRF) is defined in the OPAQUE specification as $F_k(x) = H(x, (H'(x))^k)$. The OPRF is used to compute the key rw used for the authenticated encryption described below.

1.3.1 Password-Based Key Derivation Function

In case of a server compromise where the result of the authenticated function is stolen, it's possible to increase the computing cost of a dictionary attack by applying a Password-Based Key Derivation Function (PBKDF) on the result of the OPRF, i.e. $rw = \text{PBKDF}(F_k(pw))$. I chose **Scrypt** because it's recommended as a modern PBKDF in the OPAQUE specification and it's the only PBKDF provided by the `cryptography` library used by this implementation.

1.4 Pseudorandom Function f_k

For the Pseudorandom Function (PRF) I chose 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 Authenticated Encryption (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 (128-bit blocks as AES is the underlying encryption cipher). In order to distinguish between the plaintext and the padding, a first `0x80` byte is added before the 0s.

1.6 Key-Exchange Protocol

OPAQUE uses a key-exchange (KE) protocol from which is derived a symmetric key at the end of the login phase. The KE protocol used in this implementation is HMQV. Here are two variables that are needed by the KE protocol in the login phase:

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

X_u and X_s are defined below. 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 with the server. During the second step, the user logs in with the server so that both can obtain a symmetric key. Note that all sensible variables (e.g. keys, password) should be erased from memory at some point, something that is not done in this implementation.

2.1 Registration

In this step, the client registers his password with 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.

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

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 symmetric key between the client and the server that will be used to encrypt all further communications between the two parties. When one party is said to **abort**, it means that it exits the login step and outputs (None, *sid*, *ssid*). Of course, any output of the login step is not to be made public.

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** retrieves the values stored during the client's registration, namely k_s, p_s, P_s, P_u and c. It must make sure α, X_u and P_u are on the curve or it must abort. It generates a private and public key pair (x_s, X_s) and computes:

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

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

3. The **client** must make sure β and X_s are on the curve or it must abort. It computes the key $rw = \text{PBKDF}(H(pw, \beta^{1/r}))$ and proceeds to decrypt

c, i.e. $p_u, P_u, P_s = \text{AuthDec}_{rw}(c)$. The tag must be valid and P_s must be on the curve or else the client must abort. The client computes:

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

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 must abort.

The client and the server now both have the same symmetric key SK. The step's output for both parties is set to (SK, *sid*, *ssid*).