Projet Yubino

Informatique embarquée

Vincent Ruello - 2023

Table des matières

Consignes

Ce projet est à réaliser par groupe de Travaux Pratiques.

Rendu

Chaque groupe doit envoyer une archive au format **tar** appelée `projet-<nom-du-groupe>.tar` par e-mail à l'adresse **vincent.ruello@u-paris.fr** avant le **vendredi 08/12 à 16:00 (CET).**

Cette archive doit contenir à minima :

- Un fichier `README` présentant les fonctionnalités implémentées, les éventuelles difficultés rencontrées, les tests réalisés et la procédure à utiliser afin de compiler les sources du projet et téléverser le binaire généré sur une carte Arduino.
- Un schéma électrique explicitant la façon dont les différents composants externes sont utilisées ainsi que les identifiants des broches du microcontrôleur utilisées.

- L’ensemble des fichiers sources nécessaires à la compilation du projet et aux tests.

Les fichiers sources rendus doivent pouvoir être compilés à l’aide de `avr-gcc` (et `avr-libc`). Il est attendu que le code soit testé avec les moyens mis à disposition et que l’ensemble des tests passent.

Soutenance

Une « soutenance » aura lieu le **vendredi 15/12** sur l’horaire habituel du cours (8h30/12h30). Chaque groupe se verra attribuer un créneau de dix minutes, pendant lesquelles il devra répondre à des questions portant sur le projet et son rendu (en particulier ses choix d’implémentation).

Critères d’évaluation

L’évaluation du projet tiendra compte du rendu initial et de la qualité des réponses apportées lors de la soutenance.

Une attention particulière sera portée sur la lisibilité du code (choix des noms de variables et fonctions, commentaires) et ses performances (rapidité d’exécution, utilisation des différents espaces mémoires, consommation énergétique). N’hésitez pas à justifier vos choix techniques dans les commentaires.

Tout code soumis doit être **compris** et **maîtrisé**. L’utilisation d’outils d’aide à la programmation ou le recours à une aide extérieure n’est pas interdite, à condition que l’assertion précédente soit respectée.

Contexte

Traditionnellement, la plupart des applications authentifient leurs utilisateurs à l'aide d'un couple « (nom d'utilisateur, mot de passe) ». Cette méthode d'authentification souffre pourtant de nombreux défauts, notamment :

- la faible qualité des mots de passe utilisés ;
- la multitude de mots de passe différents à retenir pour l'utilisateur (idéalement un par service) ;
- la vulnérabilité de l'utilisateur aux attaques par hameçonnage.

Face à ces limites, plusieurs organisations travaillent pour généraliser de nouvelles méthodes d'authentification, à la fois plus sûres et plus simples. Parmi ces dernières, le standard FIDO2 (*Fast IDentity Online*) vise à remplacer le mot de passe par l'utilisation d'un périphérique physique dédié. Il est proposé et défendu par le consortium FIDO Alliance regroupant de nombreux géants du numérique (comme Amazon, Apple, Google, Intel, Microsoft ou Thalès).

FIDO2

Le standard FIDO2 (Fast IDentity Online) est basé sur un périphérique dédié, appelé *Authenticator*, communiquant via USB, NFS ou Bluetooth qui, à l'aide de cryptographie à clé publique, permet à l'utilisateur de s'authentifier auprès d'un ensemble de services (appelés *Relying Parties*) à travers un *Client* (souvent un navigateur web).

Trois acteurs sont ainsi identifiés :

- le *Relying Party* (RP) : service souhaitant authentifier des utilisateurs ;
- le *Client* : souvent un navigateur web, il fait le lien entre le *Relying Party* et l'*Authenticator ;*
- l'*Authenticator* : élément physique responsable des opérations cryptographiques ainsi que de la génération et conservation des clés propres à chaque service ;

Le standard décrit deux étapes :

1. L'enregistrement de l'*Authenticator* auprès d'un *Relying Party* ;
2. L'authentification de l'utilisateur auprès d'un *Relying Party* à l'aide de son *Authenticator*.

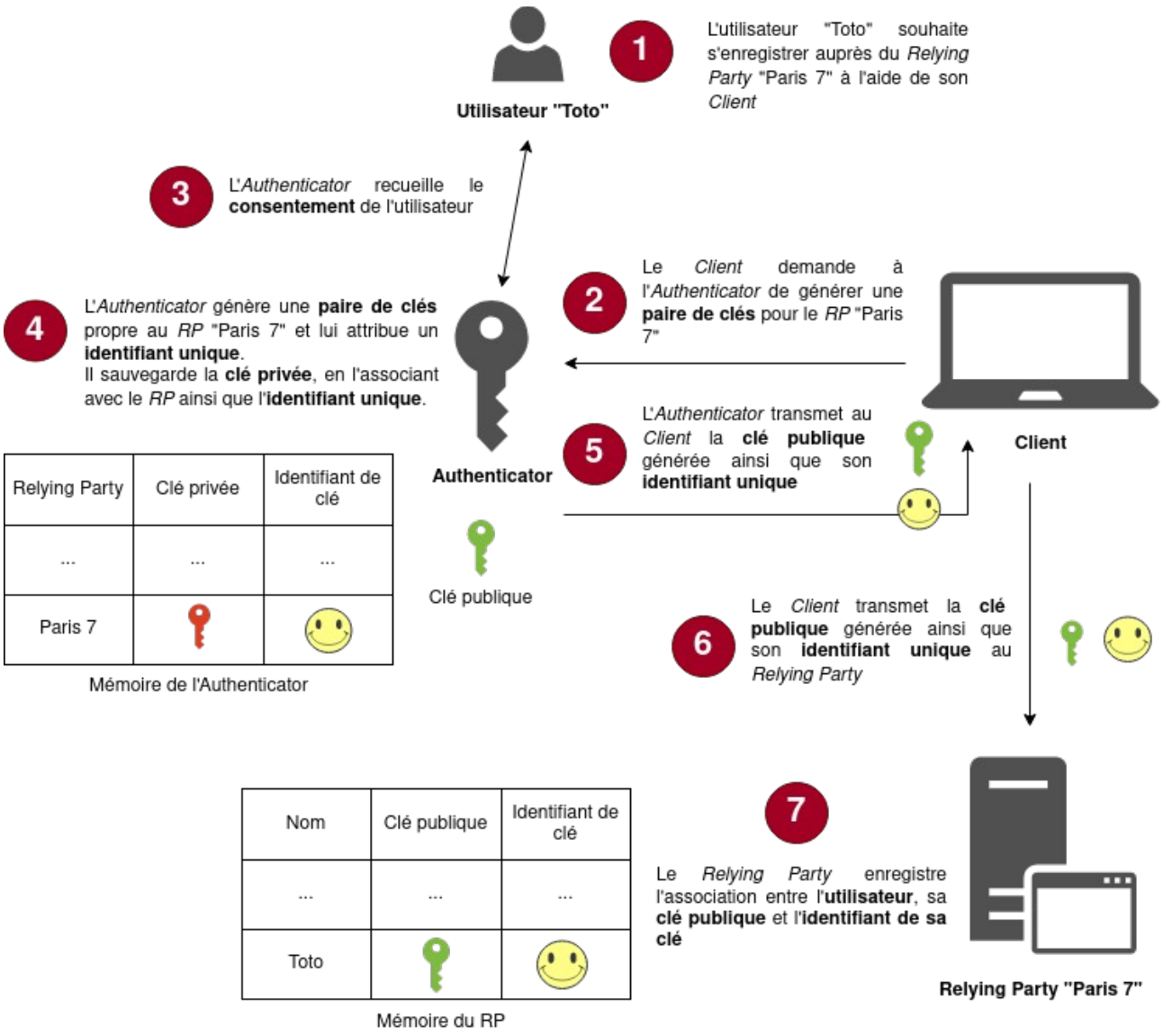
Les échanges entre le *Client* et l'*Authenticator* sont standardisés (standard CTAP2, pour *Client-To-Authenticator Protocol*).

Les opérations réalisées par le *Client* et le *Relying Party* sont également standardisées (WebAuthn).

Les deux étapes sont présentées de façon volontairement simplifiée dans la suite de ce document.

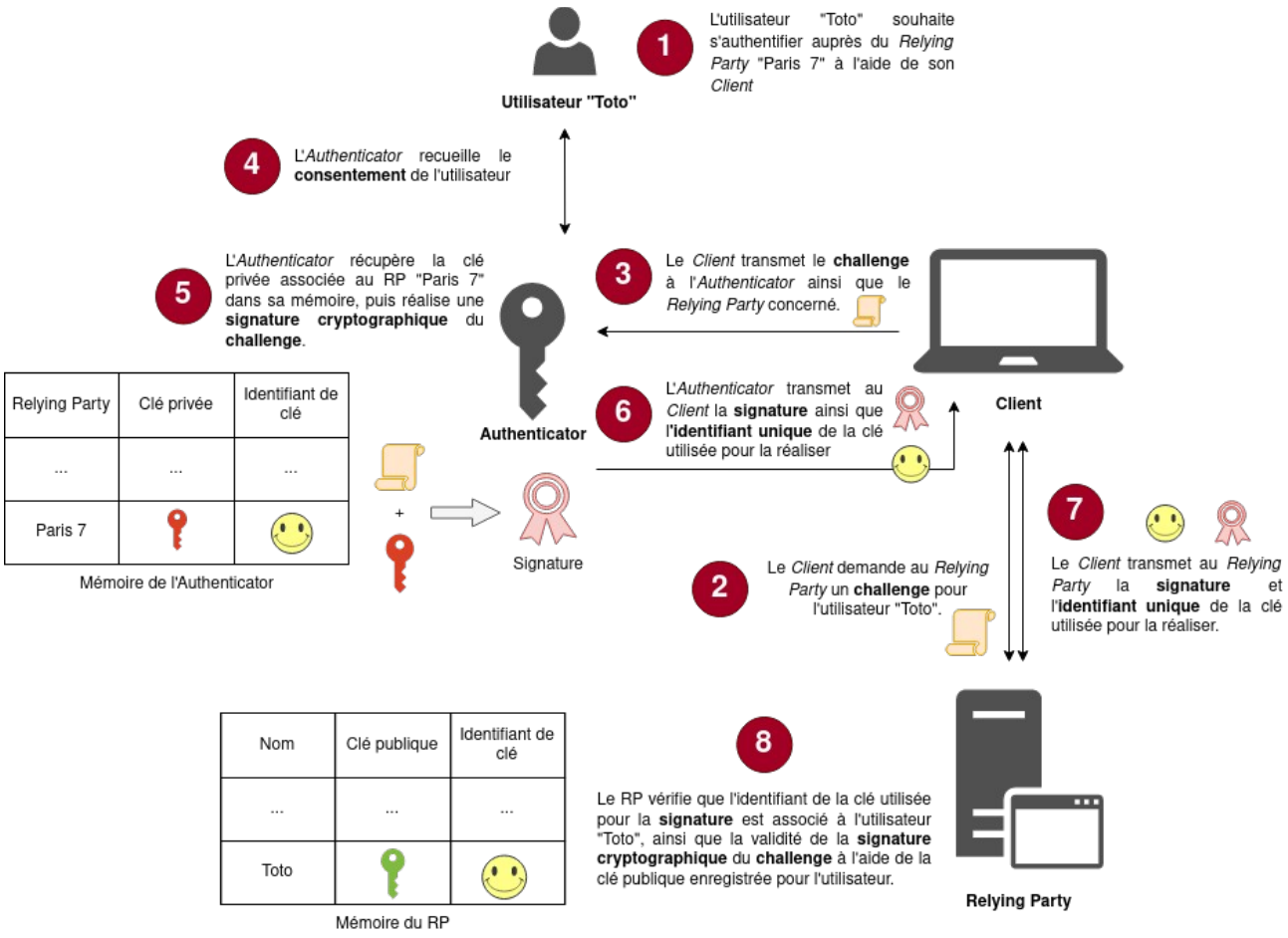
Étape 1 : Enregistrement

L'utilisateur souhaite s'enregistrer auprès d'un *Relying Party* **(1)** supportant le standard FIDO2 à travers son *Client* (par exemple un navigateur web). Pour se faire, le *Client* demande à l'*Authenticator* de générer une paire de clés propre à ce *Relying Party* **(2)**. L'*Authenticator,* après recueil du consentement de l'utilisateur **(3),** génère la paire de clé **(4)**, lui attribue un identifiant unique, sauvegarde l'association entre le *Relying Party*, la clé privée et l'identifiant unique, puis transmet au *Client* la clé publique **(5)** accompagnée de l'identifiant unique. Le *Client* transmet ces éléments au *Relying Party* **(6)**. Le *Relying Party* enregistre l'association entre l'utilisateur et le couple « (identifiant unique, clé publique) » reçu **(7)**.



Étape 2 : Authentification

Lorsque l'utilisateur souhaite s'authentifier auprès d'un *Relying Party* **(1)** (après avoir enregistré son *Authenticator*), son *Client* récupère auprès de celui-ci un **challenge** **(2)** (généré aléatoirement par le RP). Le client transmet le nom du *Relying Party* ainsi que le **challenge** à l'*Authenticator* **(3)**, qui, après validation de l'utilisateur **(4)**, calcule une signature cryptographique à l'aide de la clé privée associée au *Relying Party* **(5)**, et envoie cette signature ainsi que l'identifiant unique de la clé utilisée au *Client* **(6)**. Le *Client* transmet ces informations au *Relying Party* **(7)**. Le *Relying Party* valide que l'identifiant de la clé utilisée est bien lié à l'utilisateur, puis vérifie la signature à l'aide de la clé publique associée à cet identifiant **(8)**. Une signature valide prouve que l'utilisateur possède la clé privée générée lors de l'enregistrement.



Cahier des charges

On souhaite réaliser un « *Authenticator* » implémentant une version simplifiée du protocole CTAP à l'aide du microcontrôleur ATmega328P (présent sur une carte Arduino UNO R3).

La signature cryptographique est réalisée avec l'algorithme ECDSA en utilisant la courbe secp160r1.

Afin de montrer à l'utilisateur que son consentement est demandé, l'*Authenticator* fait clignoter une LED (à la fréquence 1Hz). Afin de donner son consentement, l'utilisateur doit appuyer sur un bouton relié à l'*Authenticator*.

*Remarque : **ce projet est uniquement à but éducatif.** En effet, le microcontrôleur ATmega328P n'est pas adapté à cet usage (absence de mémoire non volatile protégée, absence de primitives cryptographiques mises à disposition par le matériel, ...). De même, la courbe choisie n'apporte pas un degré de sécurité élevé mais bénéficie d'un temps de calcul de signature acceptable sur ATmega328P (de l'ordre de la seconde).*

Ressources

- Le protocole CTAP simplifié à implémenter est détaillé dans une partie dédiée.

- Afin de réaliser les opérations cryptographiques avec l'algorithme ECDSA (courbe secp160r1), il est proposé d'utiliser la bibliothèque micro-ecc[1] qui implémente l'algorithme ECDSA (avec cette courbe) pour différentes architectures embarquées dont AVR.
  Remarque : cette bibliothèque a besoin d'une source d'aléatoire pour fonctionner. L'ATmega328P ne possède pas de générateur d'aléatoire, mais il est possible d'obtenir un résultat « acceptable » en utilisant des périphériques comme l'ADC ou un timer.

- Une implémentation d'un *Client* en Python est fournie à l'adresse https://vps.vruello.fr/k9npX2nqL7/resources/yubino-client.tar. Elle prend en charge le protocole CTAP simplifié présenté par la suite, ainsi que la communication avec un *Relying Party* (protocole simplifié, non documenté). Une documentation succincte de l'utilisation du *Client* fourni est disponible dans son fichier `README`.
  De plus, un **ensemble de tests** du comportement de l'*Authenticator* sont fournis avec le *Client* (procédure détaillée dans le `README`).

- Un *Relying Party* conçu pour dialoguer avec le *Client* fourni est mis à disposition à l'adresse https://vps.vruello.fr/k9npX2nqL7/rp/. Il permet notamment à un utilisateur de créer un compte (avec le couple « (nom d'utilisateur, *Authenticator*) ») puis de s'authentifier afin d'obtenir une page d'accueil personnalisée.
  Remarque : une fois les fonctionnalités de l'Authenticator implémentées, il est possible de les tester en situation réelle avec le Relying Party mis à disposition. Les comptes enregistrés sur le Relying Party sont remis à zéro tous les jours à minuit. Afin d'utiliser ce Relying Party, le client yubino doit être lancé de la façon suivante :

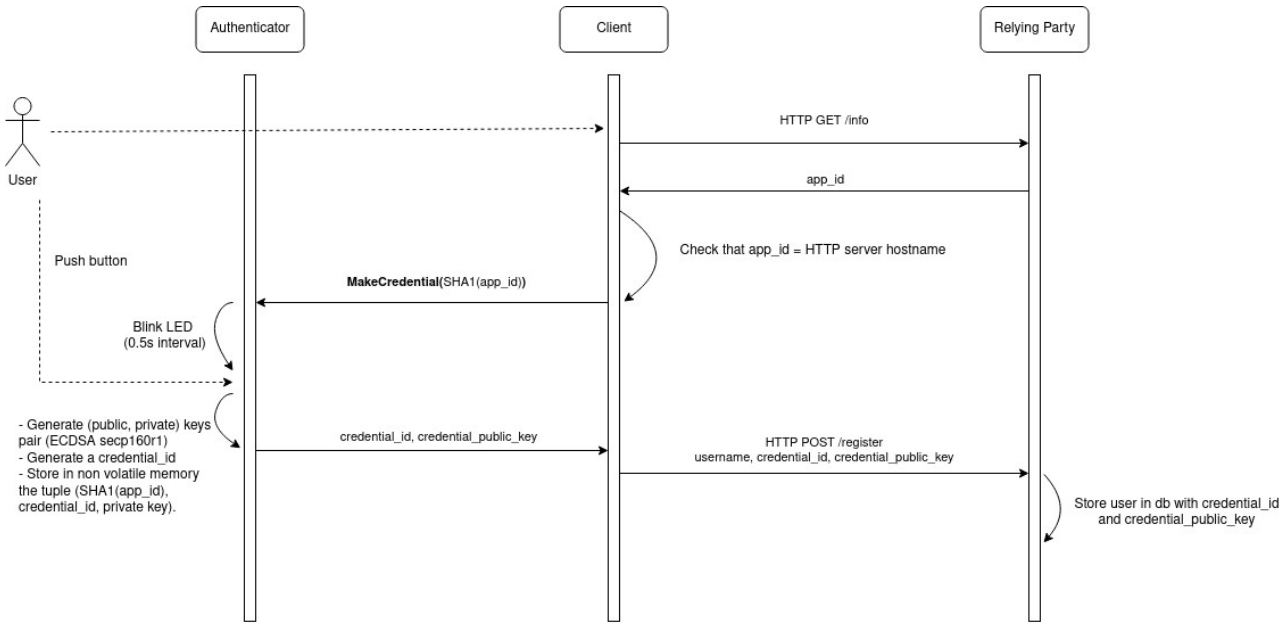
```
yubino -r https://vps.vruello.fr/k9npX2nqL7/rp/
```

1 https://github.com/kmackay/micro-ecc

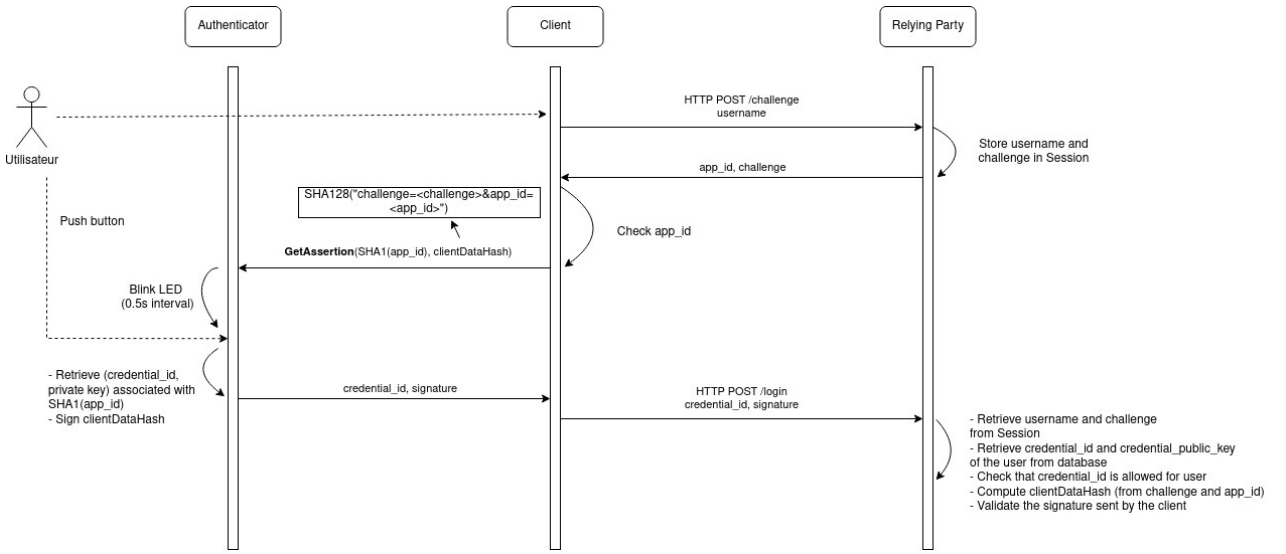
Protocole

Les interactions entre les différents acteurs (utilisateur, *Authenticator*, *Client*, *Relying Party*) sont décrites dans les diagrammes suivants :

- Enregistrement



- Authentification



Protocole CTAP simplifié

Le protocole présenté par la suite est une version simplifiée du protocole CTAP2. La communication entre le *Client* et l'*Authenticator* se fait via une liaison série (8 bits de données, pas de bit de parité, 1 bit de stop) avec un baud rate de 115 200.

On distingue quatre types de messages :

- **MakeCredential** : utilisé lors de l'enregistrement de l'*Authenticator* auprès d'un *Relying Party*
- **GetAssertion** : utilisé lors d'une authentification à l'aide de l'*Authenticator* auprès d'un *Relying Party*
- **ListCredentials** : permet de lister les *Relying Parties* enregistrés dans l'*Authenticator* (ainsi que l'identifiant unique de la clé qui leur est associé)
- **Reset** : permet de réinitialiser la mémoire non volatile de l'*Authenticator* et de supprimer toutes les clés existantes

Chaque *Relying Party* est identifié par son « nom d'hôte », supposé unique, et nommé `app_id`.

Chaque type de messages se décompose en un message de requête (envoyé par le *Client* à l'*Authenticator*) suivi d'un message de réponse (envoyé par l'*Authenticator* au *Client*). Le message de requête commence systématiquement par un octet dont la valeur permet de déterminer le type de requête envoyée. De même, chaque réponse commence par un octet contenant un statut où toute valeur différente de `STATUS_OK` (`0`) signifie qu'une erreur s'est produite (et la valeur indique le type d'erreur rencontrée).

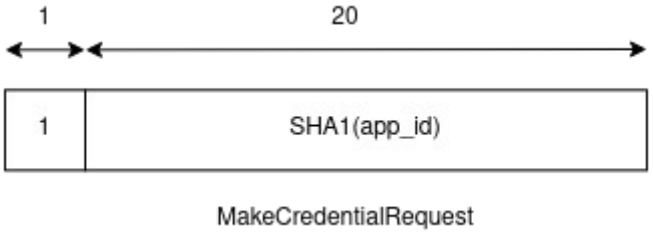
Sauf mention contraire, toutes les données échangées sont transférées en commençant par l'octet de poids faible (*little-endian*).

MakeCredential

Le message `MakeCredentialRequest` est envoyé à l'*Authenticator* afin qu'il génère une nouvelle paire de clé pour une application. Une empreinte `SHA1` de l'identifiant de l'application (`app_id`) est présente dans le message.

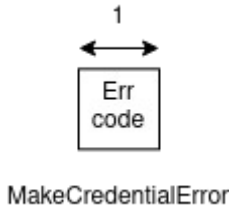
Le format du message est le suivant :

- le premier octet contient la valeur `0x01`
- les vingts octets suivant contiennent l'empreinte de l'identifiant de l'application en commençant par l'octet de poids faible.



MakeCredentialRequest

En cas d'erreur de réception de ce message, l'*Authenticator* envoie un message `MakeCredentialError` avec le code `STATUS_ERR_BAD_PARAMETER`.



A la réception de ce message, l'*Authenticator* indique à l'utilisateur qu'il doit donner son consentement en faisant clignoter une LED (changement d'état toutes les 0.5s). L'utilisateur donne son consentement en appuyant sur un bouton, ce qui arrête le clignotement de la LED.

Si l'utilisateur n'a pas donné son consentement au bout de 10 secondes, un message `MakeCredentialError` est renvoyée avec le code `STATUS_ERR_APROVAL`.

Une fois le consentement de l'utilisateur obtenu, l'*Authenticator* génère une nouvelle paire de clés (ECDSA, courbe secp160r1). La taille de la clé privée est 21 octets, la taille de la clé publique est 40 octets.

En cas d'erreur lors de la génération de la paire de clés, l'*Authenticator* envoie un message `MakeCredentialError` avec le code `STATUS_ERR_CRYPTO_FAILED`.

L'*Authenticator* génère également un identifiant unique associé à cette paire, appelé `credential_id`, de longueur 128 bits.

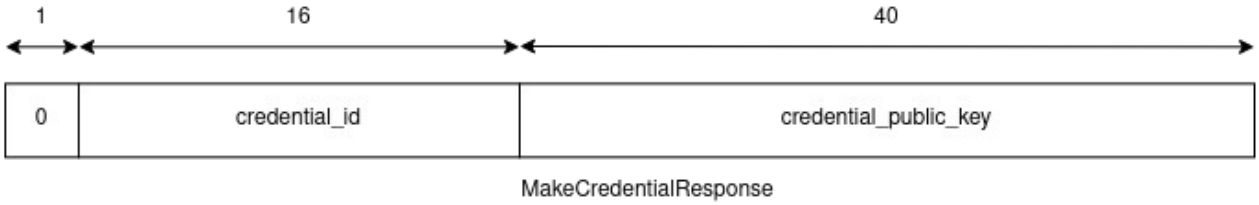
L'*Authenticator* sauvegarde de façon non volatile l'association entre l'empreinte de `app_id`, `credential_id` et la clé privée générée.

Si la mémoire non volatile est saturée et qu'il n'est pas possible d'ajouter la nouvelle entrée, l'*Authenticator* envoie un message `MakeCredentialError` avec le code `STATUS_ERR_STORAGE_FULL`.

Si une entrée existe déjà pour l'empreinte de `app_id`, ses valeurs doivent être remplacées par la nouvelle entrée.

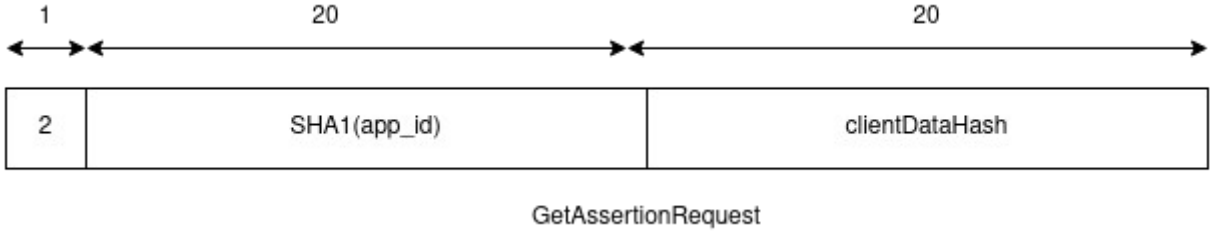
Si aucune erreur ne s'est produite, l'*Authenticator* envoie un message `MakeCredentialResponse` contenant :

- `credential_id` (16 octets)
- la clé publique générée (40 octets)



GetAssertion

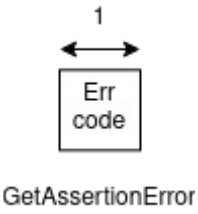
Le message `GetAssertionRequest` est envoyé à l'*Authenticator* afin d'obtenir la signature d'un *blob* `clientDataHash` (contenant notamment un challenge) avec la clé associée à un *Relying Party* identifié par l'empreinte `SHA1` de son `app_id`.



Du point de vue de l'*Authenticator*, le blob `clientDataHash` est opaque. Pour information, il est calculé par le *Client* et est le résultat de l'opération :

```
SHA1("challenge=<challenge>&app_id=<app_id>")
```

En cas d'erreur de réception de ce message, l'*Authenticator* envoie un message `GetAssertionError` avec le code `STATUS_ERR_BAD_PARAMETER`.



A la réception de ce message, l'*Authenticator* recherche les informations (`credential_id`, clé privée) correspondant à l'empreinte de `app_id` fournie en paramètres.

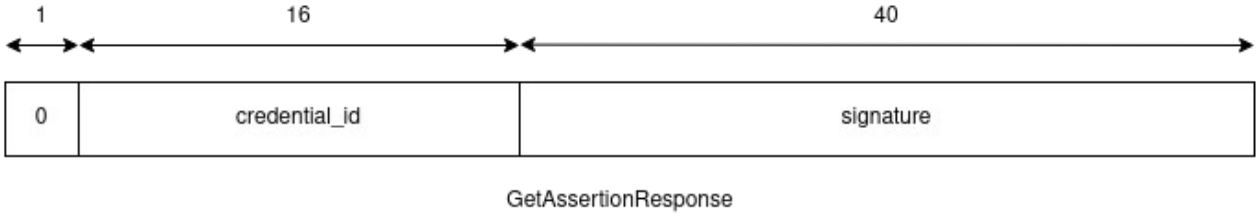
Si l'*Authenticator* ne trouve pas d'entrée correspondant à l'empreinte de `app_id`, il envoie un message `GetAssertionError` avec le code `STATUS_ERR_NOT_FOUND`.

L'*Authenticator* indique à l'utilisateur qu'il doit donner son consentement en faisant clignoter une LED (changement d'état toutes les 0.5s). L'utilisateur donne son consentement en appuyant sur un bouton ce qui arrête le clignotement de la LED.

Si l'utilisateur n'a pas donné son consentement au bout de 10 secondes, un message `GetAssertionError` est renvoyée avec le code `STATUS_ERR_APROVAL`.

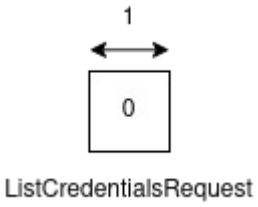
Une fois le consentement de l'utilisateur obtenu, l'*Authenticator* réalise une signature cryptographique via l'algorithme ECDSA (courbe secp160r1) du *blob* `clientDataHash`. En cas d'erreur, un message `GetAssertionError` est renvoyé avec le code `STATUS_ERR_CRYPTO_FAILED`. La taille de la signature est 40 octets.

Si aucune erreur ne s'est produite, l'*Authenticator* envoie un message `GetAssertionResponse` contenant l'identifiant de la clé utilisée pour réaliser la signature ainsi que la signature elle-même.



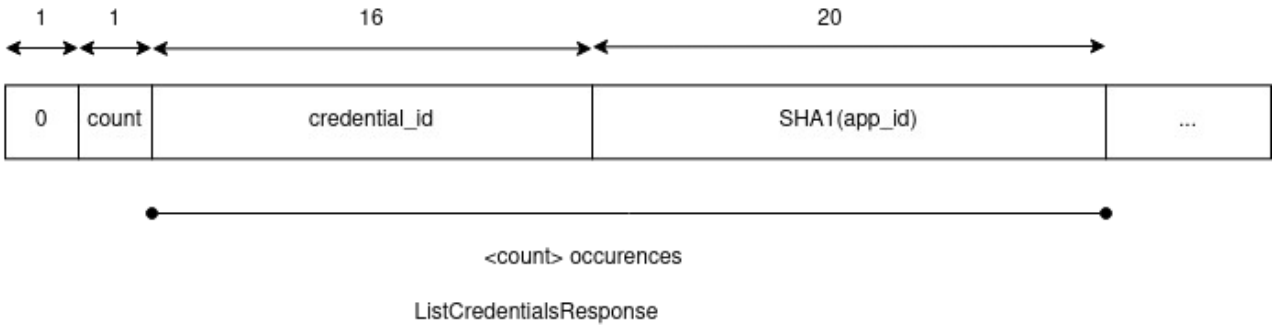
ListCredentials

Le message `ListCredentials` permet au *Client* d'obtenir la liste des informations stockées dans la mémoire non volatile de l'*Authenticator* (empreinte de l'`app_id` et identifiant de clé).



A le réception de ce message, l'*Authenticator* parcourt sa mémoire non volatile afin de récupérer les informations qui y sont stockées.

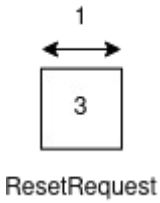
L'*Authenticator* envoie un message `ListCredentialsResponse` contenant le nombre de couples (identifiant de clé, empreinte de l'`app_id`) envoyés, puis pour chaque couple le `credential_id` suivi de l'empreinte de l'`app_id`.



Cette opération ne peut pas échouer.

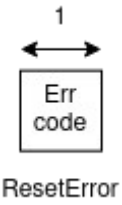
Reset

Le message `ResetRequest` permet au *Client* de demander à l'*Authenticator* de réinitialiser son stockage non volatile, supprimant ainsi toutes les clés qu'il contient.



A la réception d'un message `ResetRequest`, l'*Authenticator* indique à l'utilisateur qu'il doit donner son consentement en faisant clignoter une LED (changement d'état toutes les 0.5s). L'utilisateur donne son consentement en appuyant sur un bouton ce qui arrête le clignotement de la LED.

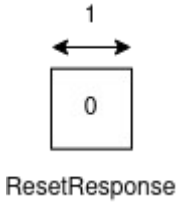
Si l'utilisateur n'a pas donné son consentement au bout de 10 secondes, un message `ResetError` est renvoyée avec le code `STATUS_ERR_APROVAL`.



Une fois le consentement de l'utilisateur obtenu, l'*Authenticator* supprime de sa mémoire non volatile l'ensemble des informations qu'il y a enregistré.

Remarque : les secrets doivent être effacés de la mémoire non volatile (par exemple remplacés par des données aléatoires ou des zéros).

Si aucune erreur ne s'est produite, l'*Authenticator* envoie un message `ResetResponse`.



Constantes

Nom	Valeur
COMMAND_LIST_CREDENTIALS	0
COMMAND_MAKE_CREDENTIAL	1
COMMAND_GET_ASSERTION	2
COMMAND_RESET	3
STATUS_OK	0
STATUS_ERR_COMMAND_UNKNOWN	1
STATUS_ERR_CRYPTO_FAILED	2
STATUS_ERR_BAD_PARAMETER	3
STATUS_ERR_NOT_FOUND	4
STATUS_ERR_STORAGE_FULL	5
STATUS_ERR_APPROVAL	6