

Modern Authentication

FIDO2 Web Authentication (WebAuthn) を学ぶ

栗原 淳

兵庫県立大学 大学院応用情報科学研究科
株式会社ゼタント

May 13, 2020

はじめに

はじめに

この講義では、

- パスワード認証に代わるモダンな認証方式「FIDO」の概要
- FIDO の認証を Web ブラウザ経由で利用する「FIDO2 WebAuthn」の利用

のさわりを学ぶ。

この講義の対象と事前準備

対象:

- 暗号・セキュリティ技術に興味がある初学者
- Web に新しい認証技術を導入したい Web 系のエンジニア

※但し、ある程度の公開鍵暗号・電子署名の知識を前提とする¹

必須ではないが触って楽しむのには必要な事前準備:

- Bash/Zsh, Git が使えるようになっていること
- Node.js, npm, yarn が使えるようになっていること
- Google Chrome ブラウザが利用可能のこと

¹どういうものか、というのを知つていれば十分。「JavaScript を使って学ぶ End-to-End セキュリティ」の資料を読んでいることを推奨
(https://github.com/junkurihara/class-e2e_security_js)。

パスワード認証から FIDO へ

認証とは

認証

「何らかの手段」で対象の正当性を確認すること。

- メッセージの正当性を確認 ⇒ メッセージ認証
- サービス利用ユーザの正当性を確認 ⇒ ユーザ認証
- etc.

※このスライドで単純に「認証」と呼んだときは、認証対象を「正規ユーザ本人」としたユーザ認証・本人認証を指すこととする。

本人認証の3つの要素

本人認証において、正当性確認のため検証されるものは大きく3要素に分類。

■ 知識

⇒ 本人しか知らない知識を持っていればOK (ex. パスワード)

■ 所有物

⇒ 本人しか持っていない物を提示できればOK (ex. HWキー)

■ 生体

⇒ 本人の体の一部を提示できればOK (ex. 指紋)



本人しか知らない



本人しか持っていない
(複製できない)



本人の体の一部

オンラインサービスでのパスワード認証

- サービスの利用者の識別子 (ID) と対応するパスワードをサービス事業者に登録、サービス利用時に利用者が自分の ID とパスワードを入力する。
- パスワードは個人の記憶にのみ存在するため、**パスワードを知っている人はそのサービスに登録してある本人と同一人物と考えることができる。**

おそらく、誰にとっても最も馴染み深い認証方式！

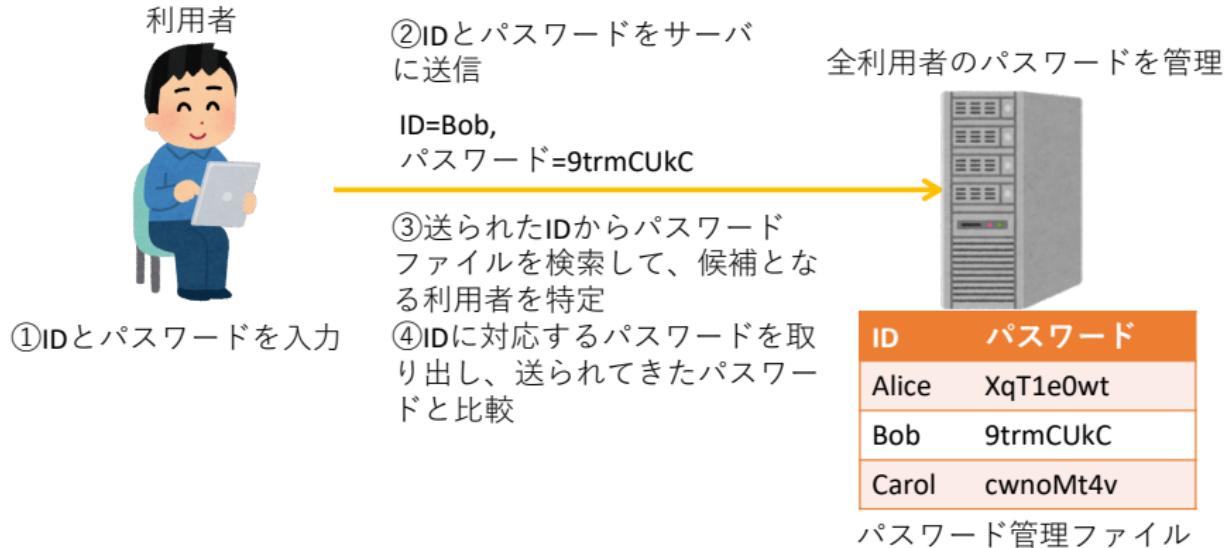


Figure: オンラインでの単純なパスワード認証

オンラインでのパスワード認証の問題

英数字・記号を組み合わせたパスワード:

- 攻撃者にとって比較的予測しやすい²
- 「強い」 パスワードを使わせるにはユーザ教育が必要³
- 覚えられない
- etc...



予測できず、誰が使っても強力で、確実に認証できる方法が必要
⇒ ハードウェアセキュリティキーを使った認証が人気に
⇒ FIDO はそのような手法の標準化された方式

²しかもオンラインだと予測→認証トライを繰り返せる

³教育なしだと覚え易く「弱い」ものを利用しがち

FIDO (Fast IDentity Online)

業界団体 FIDO Alliance⁴ の策定する、ハードウェアセキュリティキー+生体認証⁵と公開鍵暗号方式をベースとしたオンラインでの本人認証技術。

現在は FIDO2 (v2.0) が最新の規格。以降、FIDO2 の内容について触れていく。

厳密には、FIDO2 はパスワードレス認証をサポートしつつも、パスワード+デバイス・生体認証の多要素での認証もサポートする。

⁴<https://fidoalliance.org>

⁵すなわち、「所有物」と「生体」の二要素を同時に使った認証が可能。

FIDO 認証概略

FIDO 認証の特徴:

- 公開鍵暗号を利用した、オンラインでの認証方式の提供
 - 認証器によるローカルでの本人認証
 - 認証器内部に閉じた署名生成
- ⇒ 秘密鍵・パスワード等の秘密情報は外部に出ない

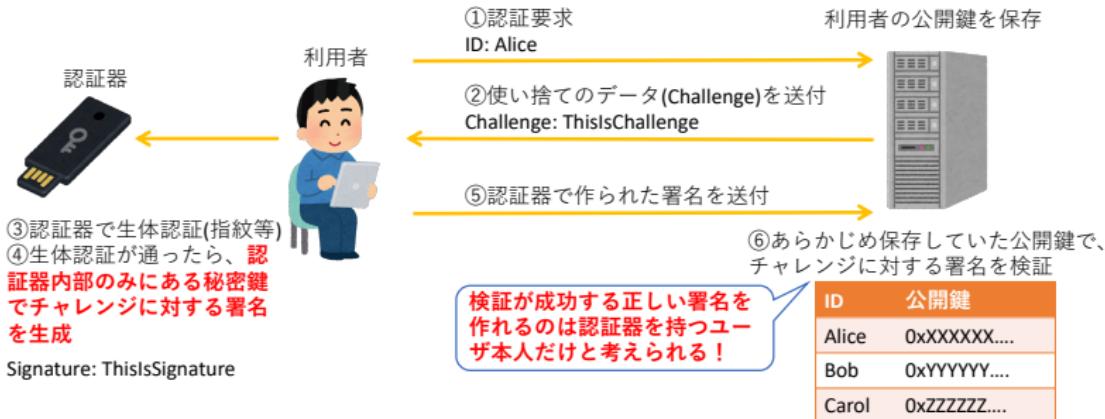
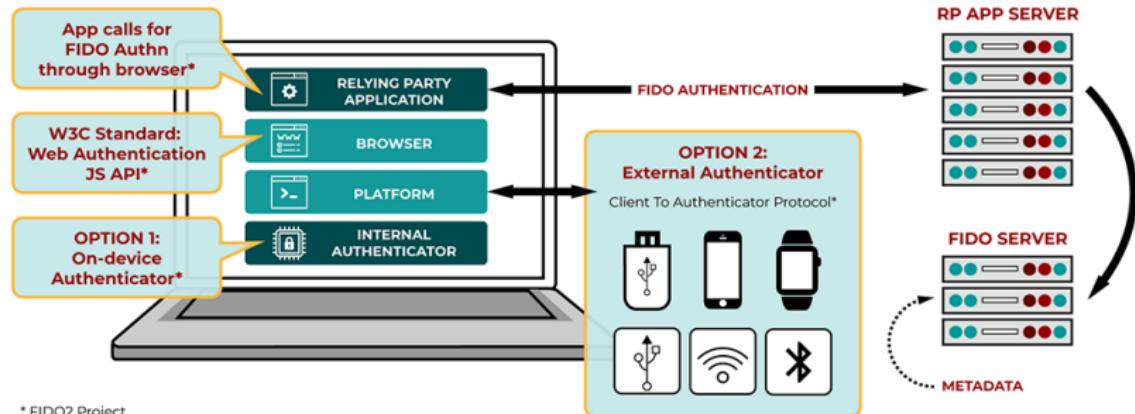


Figure: FIDO 認証概略

FIDO2 の要素

FIDO2 は、WebAuthn (Web Authentication)⁶と、CTAP
(Client-to-Authenticator Protocol)⁷の 2つの要素で構成される。



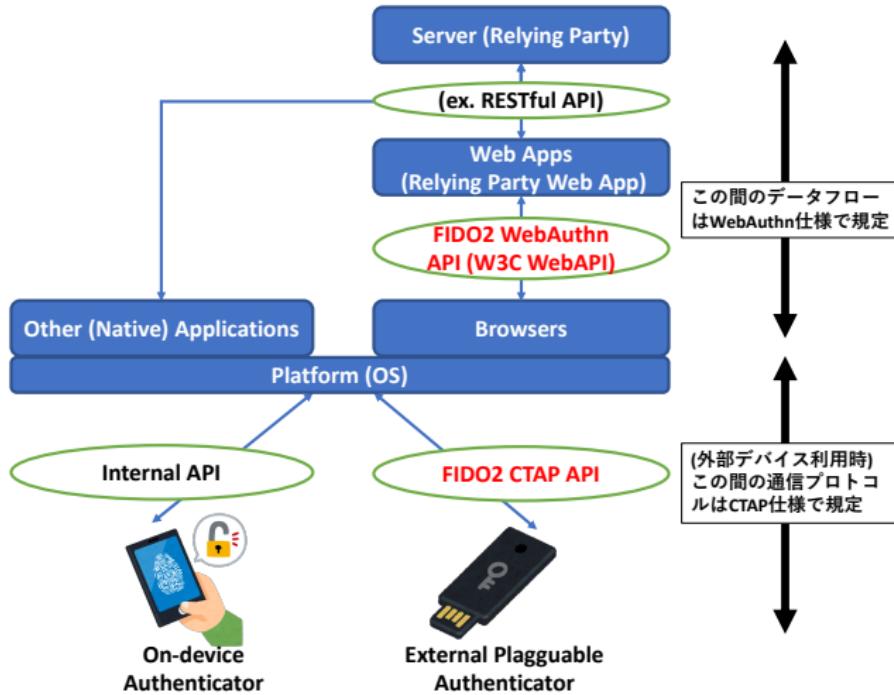
* FIDO2 Project

Figure: ©FIDO Alliance, from <https://fidoalliance.org/specifications/>

⁶Spec: <https://www.w3.org/TR/webauthn-1/>

⁷Spec: <https://fidoalliance.org/specs/fido2/fido-client-to-authenticator-protocol-v2.1-rd-20191217.html>

- **WebAuthn**: 内部/外部認証器を Call する WebAPI と、WebApp・サーバ間のデータフローを規定。
- **CTAP**: 外部認証器を Call する API と、クライアント/プラットフォームと認証器の通信プロトコルを規定。



FIDO2 CTAP⁸

USB/BLE/NFC 等で接続された HW セキュリティキーなどの外部認証器と、クライアントアプリ（ブラウザ等）およびプラットフォーム（OS）との間の通信プロトコルを規定。以下の要素で構成。

- USB/BLE/NFC など物理層の種別に応じた、通信確立のためのプロトコル
- 認証器での処理を Call する API
 - 外部認証器の情報取得
 - PIN によるローカルでのユーザ認証⁹
 - 認証器組込の秘密鍵での、ユーザ秘密鍵・証明書生成
 - ユーザ秘密鍵による署名の生成、など

ブラウザ・OS(のドライバ)は上記を実装した上で、より上位の WebAuthn のプロトコルをサポート。

⁸ <https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html>

⁹ 指紋認証やジェスチャーなどは認証器のみで完結するので API は用意されない。署名生成などのときに認証器内での認証を要求するフラグを立てる。

FIDO2 WebAuthn¹⁰

Web ブラウザをプラットフォームとし、内部/外部認証器によって生成される署名を用いた、オンラインサーバでのユーザ認証のプロトコルを規定。より具体的には、以下を規定する。

- 認証器でのユーザの公開鍵証明書の生成、サーバへの登録プロトコル
- 認証器での署名生成、サーバでの認証プロトコル
- 認証器とやりとりするためブラウザが具備すべき Web API¹¹。
大雑把に以下の 2 種類。
 - ユーザの公開鍵証明書の生成 (Credential Creation)
 - ユーザ秘密鍵による署名の生成 (Assertion Generation)

認証器とのやり取りはブラウザ/プラットフォームがサポート。
⇒ 基本的に Web App の観点からは、WebAuthn のみを意識する。

¹⁰ <https://www.w3.org/TR/webauthn-1/>

¹¹ JavaScript で Call される API。ブラウザの内部でさらに認証器の API (CTAP や内部 API) を Call する。

補足: FIDO1

FIDO1 (v1.x) は、以下の 2 つの要素で構成されている。

- UAF (Universal Authentication Framework): 生体認証機能を持つ FIDO 対応端末 (スマートフォン等) でパスワードレス認証を行う機構。USB 接続などの外部 HW セキュリティキーは利用できない。
- U2F (Universal 2nd Factor)¹²: ID・パスワード認証に加えた 2 要素認証を行うのに、外部 HW セキュリティキーを利用可能とする機構。

FIDO2 は、UAF と U2F を統合し、さらに外部 HW キーを用いてもパスワードレス認証可能な、より利便性の高い規格と見做せる。

¹²U2F は FIDO2 規格では CTAP1 と改称。FIDO2 で追加された仕様は CTAP2 と呼ばれる。

FIDO2 対応の認証器

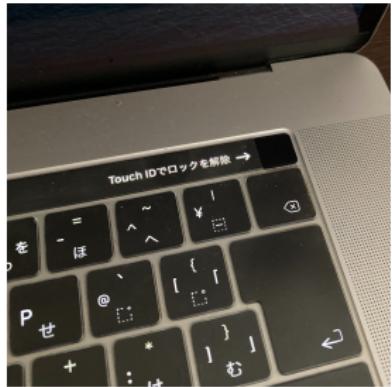
USB/NFC/BLE 等対応の外部認証器 (External Authenticator)、端末付属の認証器 (On-device/Internal Authenticator) 共々、様々な対応デバイスがリリースされつつある。



Security Key by Yubico
FIDO2 専用¹³



YubiKey 5Ci
FIDO2+OpenPGP+etc...



MacbookPro TouchID
FIDO2 認証可能¹⁴

¹³ FIDO2 CTAP1=FIDO1 U2F には対応。

¹⁴ ブラウザ等が TouchID API を Call できれば FIDO2 の On-device Authenticator として動作。Chrome 等は対応済。

FIDO2 標準化状況

FIDO は業界団体の策定した規格ではあるが、

- **FIDO2 CTAP**: ITU-T で勧告として国際標準化¹⁵
- **FIDO2 WebAuthn**: W3C で勧告として国際標準化¹⁶

と、認証器とプラットフォーム/ブラウザ間の通信プロトコル、
サーバ・ブラウザ間の認証プロトコルの両者共に国際標準として
策定済。

¹⁵<https://fidoalliance.org/>

fido-alliance-specifications-now-adopted-as-itu-international-standards/

¹⁶<https://www.w3.org/2019/03/pressrelease-webauthn-rec.html.ja>

この後、FIDO2 WebAuthn の内容に実際に触れ、最新の認証技術について理解を深めてみよう。¹⁷

¹⁷ 今回は Web 技術から学ぶセキュリティに注力するため、ローレイヤの FIDO2 CTAP については別の機会で。

実験環境の準備

準備

説明を聞きつつ手を動かすため、まず環境準備。

今回は以下の 2 つを WebAuthn の API を Call しながら実験してみる。

- 認証器を使って「ユーザ登録」

⇒ 認証器からのメッセージを解してみて実際に証明書および生の公開鍵を取り出してみる。

- 認証器を使って「ユーザ認証」

⇒ 署名を解してみて、登録時に取り出した公開鍵で署名が通ることを確認してみる。

⇒ この 2 つが FIDO2 WebAuthn のパスワードレス認証の基礎。

環境

以下の環境が前提:

- Node.js LTS (≥ 12) がインストール済で yarn が使える¹⁸
- ブラウザとして、Google Chrome (系ブラウザ)、もしくは Firefox がインストール済み
- Visual Studio Code や WebStorm などの統合開発環境がセットアップ済みだとなお良い

¹⁸インストールコマンド: `npm i -g yarn`

JavaScript プロジェクトの準備

1 プロジェクトの GitHub リポジトリを Clone

```
$ git clone  
https://github.com/junkurihara/class-fido2_webauthn.git  
$ cd sample
```

2 依存パッケージのインストール

```
$ yarn install
```

3 ライブラリのビルド

```
$ yarn build
```

認証器の準備

実験の前に認証器をセットアップしておく。例えば「Security Key by Yubico」の場合は、「YubiKey Manager」をインストールし、PINを設定。¹⁹

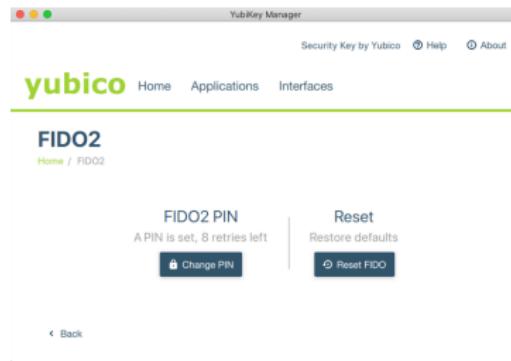


Figure: YubiKey Manager (Mac)

¹⁹ 「PIN+認証器へのタッチ」が生体認証という扱い。PIN 設定はなくても動作するが、タッチだけで生体認証したことになってしまう。PIN ではなく指紋認証を使うような認証器では、指紋登録が前もって必要。

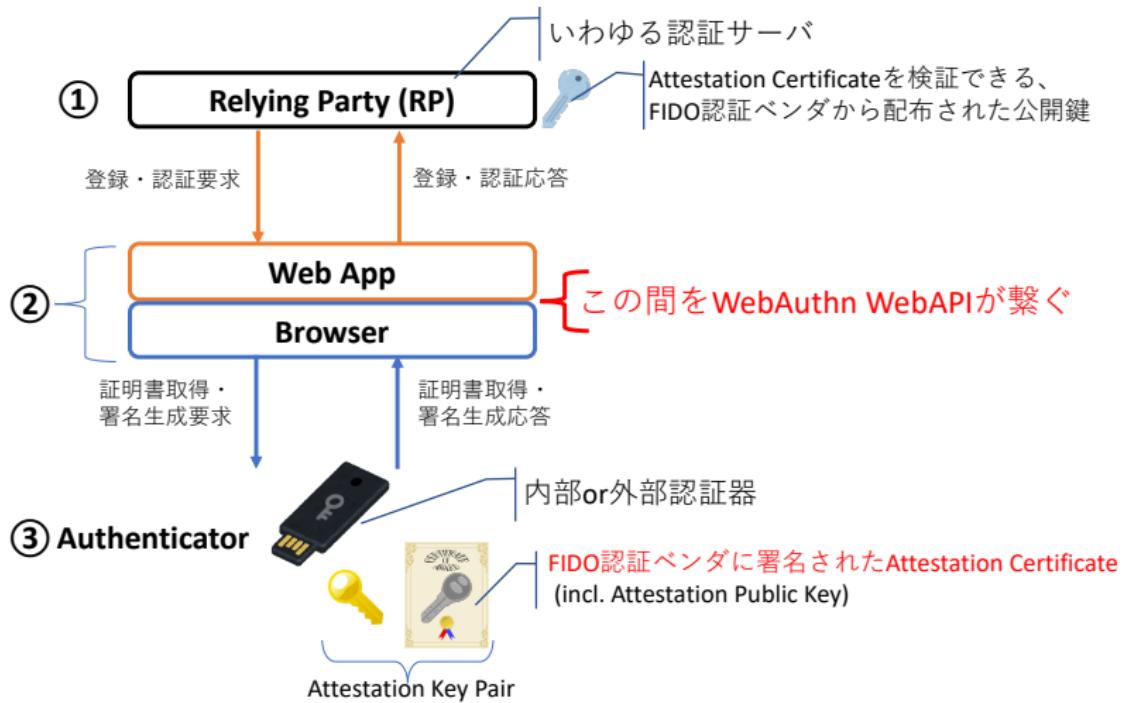
FIDO2 WebAuthn

FIDO2 WebAuthn 事始め

FIDO2 WebAuthn のフローの参加エンティティ

WebAuthn の動作フローでは、以下の 3 つの (抽象化された) 参加エンティティが存在。

- ① **Relying Party (RP)**: サービス提供者の認証サーバ
- ② ブラウザ+RP の **WebApp**: ユーザおよび認証器とやり取り
- ③ **Authenticator**: 内部/外部認証器。以下をセキュアに保持。
 - **Attestation Key Pair**: 公開鍵・秘密鍵ペア
 - **Attestation Certificate**: 上記の公開鍵に対し、FIDO2 で承認された製造元が署名した証明書



WebAPIとして用意されるWebAuthn APIは、ブラウザ経由でWebAppが認証器とやりとりする役割を担う。

FIDO2 WebAuthn の 2 つのフロー

FIDO2 WebAuthn は、2 つの動作フローを規定する。

- **ユーザ登録フロー**: 認証器を使って、Relying Party にユーザの ID や認証情報=公開鍵²⁰を登録する処理
- **ユーザ認証フロー**: 認証器を使って、Relying Party に事前に認証情報を登録したユーザ自身であることを証明する処理。

以降、この 2 つのフローの中身を見ていくが、その前に FIDO2 WebAuthn において **登録・認証の安全性を担保する Attestation** という概念について解説しよう。

²⁰Credential Public Key/Certificate のこと

FIDO2 における Attestation

FIDO2 では、「Attestation」という重要な概念が存在する。

FIDO2 における Attestation

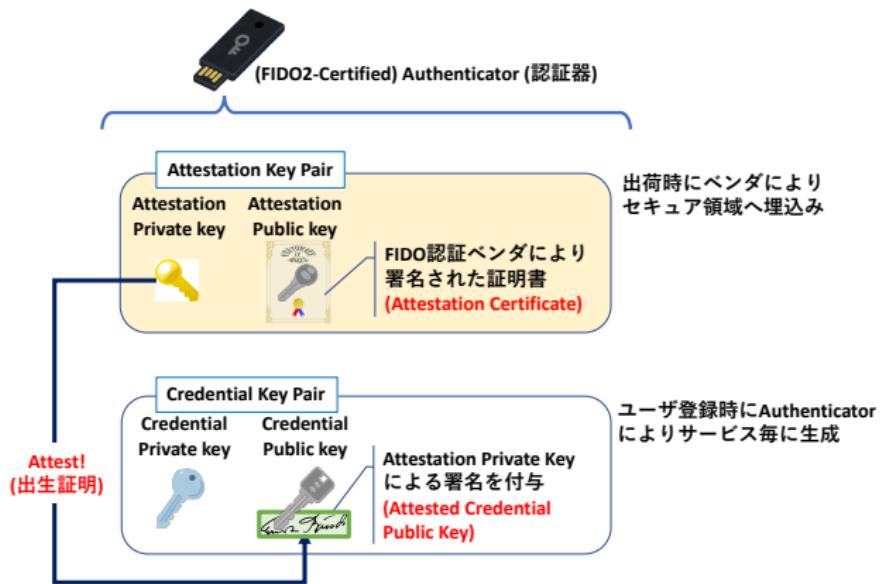
主として、「新しく生成・登録するユーザの公開鍵が、正しく FIDO2 認定を受けている認証器で生成された公開鍵であることを保証する機構」という意味。すなわち、出生証明。

Relying Party は、出生証明を確認してユーザを登録。
⇒ 偽造認証器を挿されたユーザの登録を弾ける。
⇒ 認証器に基づく FIDO2 認証の安全性は維持。



※認証器で生成するユーザの鍵ペアを Credential Key Pair、Attest されたその公開鍵を Attested Credential Public Key と呼ぶ。

Attestation の流れ。Attestation は 2 段階の証明で成立:



ユーザ登録時にユーザ公開鍵 (Credential Public Key) を出生証明して登録
⇒ Attested Credential Public Key の署名を Attestation Certificate で検証
⇒ Attestation Certificate を FIDO 認証ベンダの公開鍵²¹で検証

²¹ルート証明書

補足: Attestation の種類

- **Basic:** ベンダが認証器モデルごとに特有の Attestation Key Pair を埋込む。同じモデルの認証器では同じ鍵ペアでも良い。
- **Self:** Attestation Private Key = Credential Private Key として、自分の秘密鍵で署名して自己 Attest する。
- **AttCA²²:** Attestation Certificate を動的に生成する手法。外部に信頼できる第三者の認証局を設け、認証器が Attestation (Identity) Key Pair を生成して、認証局へその公開鍵への署名を依頼。
- **ECDAA²³:** 楕円曲線上の匿名認証 (Direct Anonymous Attestation; DAA) を利用して、認証器の情報を与えることなく出生証明を実現。
- **None:** Attestation なし。

この資料では **Basic** 前提。

Basic でも HW 構造的に秘密鍵は認証器から取出せない。AttCA では、認証器の TPM に埋め込まれた鍵を、証明書生成ではなく認証局との暗号通信用に用いる。

²² Attestation Certificate Authority

²³ Elliptic Curve based Direct Anonymous Attestation; アルゴリズム仕様は現状ドラフト。

この後やってみること

この後は、

- ユーザ登録フロー
- ユーザ認証フロー

の両者において、localhostでRPを模擬²⁴しつつ、認証器・ブラウザ間でやりとりされるデータと、認証器内部での処理をコードを見ながら確認・解説していく。

²⁴ ブラウザ・RPとのやりとりは標準がないことと、処理フロー・データフローを理解することに重点をおくため。しかし Python Flask 等で REST API でやりとりするサーバは簡単に実装可能。

以下のコマンドをとりあえず叩いてみると、ブラウザ (Google Chrome) が立ち上がって認証器の挿入を求められることが確認できる。

ユーザ登録→ユーザ認証の一連のテストコードを実行

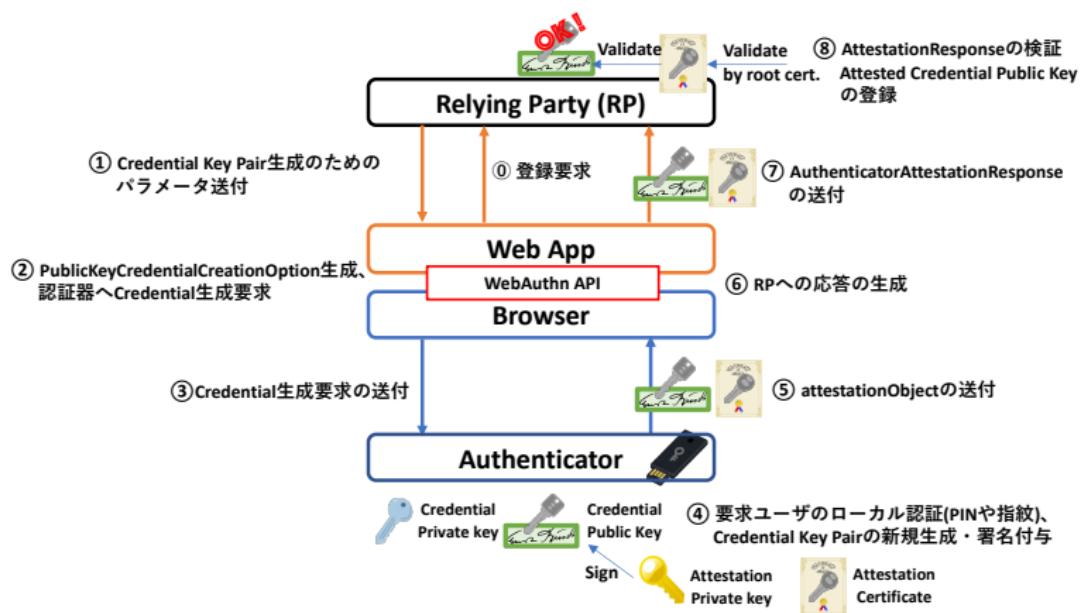
```
$ yarn test // sample ディレクトリで実行
```

この時に行われている動作を解説する。

FIDO2 WebAuthn ユーザ登録フロー

WebAuthn ユーザ登録フロー

以下のような流れで WebAuthn の認証のための登録を行う。

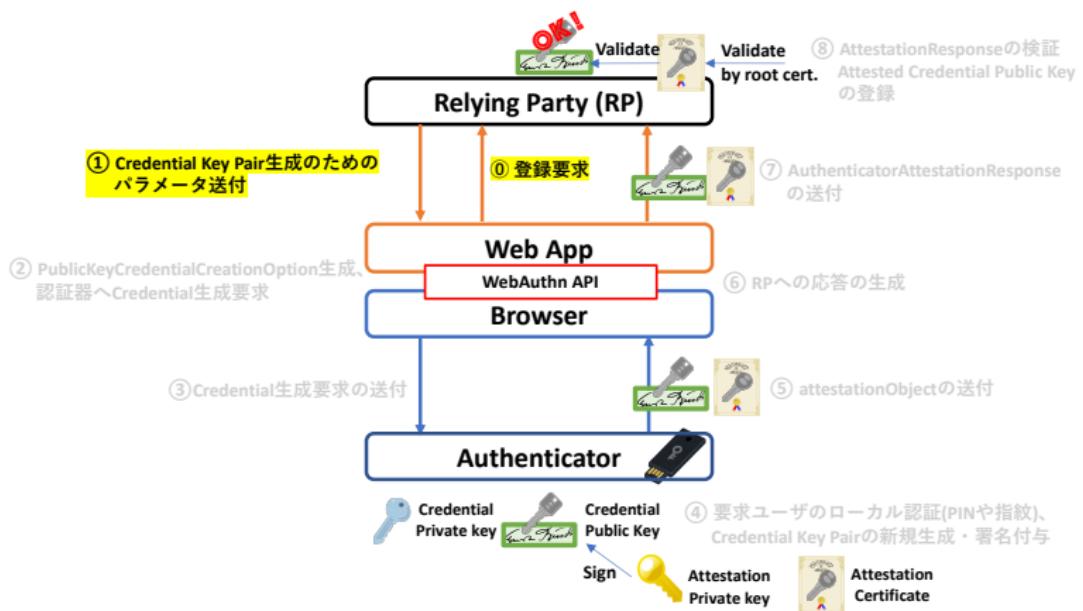


単純に言うと、認証器で Credential Public Key を生成、その出生証明を RP で確認・登録という処理。

このユーザ登録の各ステップを、実際のデータを確認しながら追っていく。

WebAuthn ユーザ登録: RPへユーザ登録要求

①, ② ユーザ登録のため Credential 生成パラメタを RP から取得。



WebApp と RP 間の要求・応答フォーマットは規定されていない。
⇒ RP 側の (REST) API は実装者に任せられている。

WebApp が RP から取得するパラメタは以下の通り。²⁵

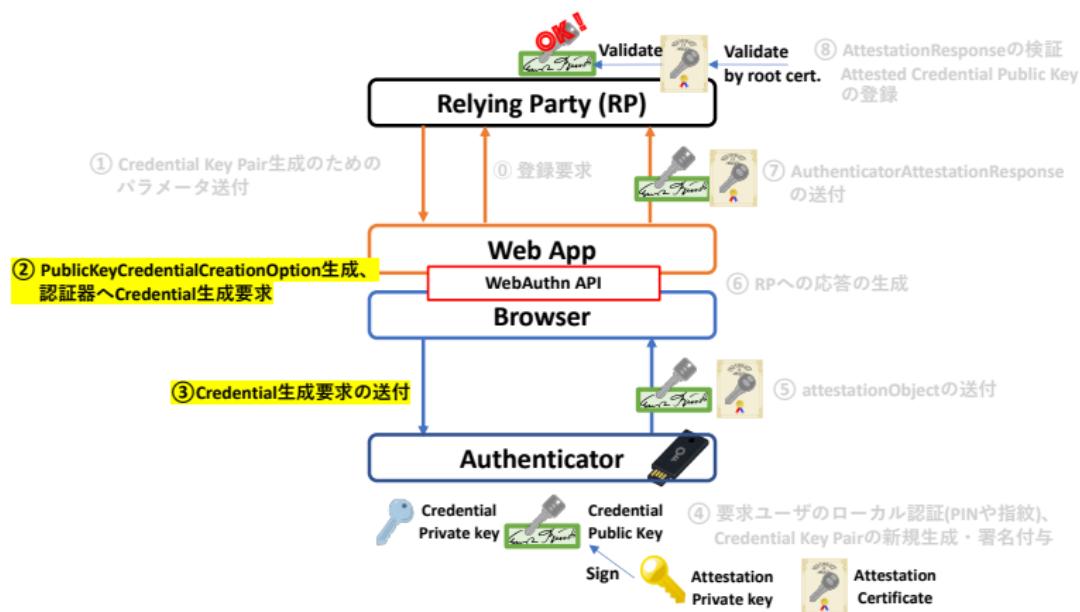
- **Challenge:** 暗号学的にランダムな使い捨ての binary string (最低 16bytes, 通常 32bytes 程度)
- **User Info:** ユーザ情報。ID、メールアドレス、名前。
- **Relying Party Info:** RP(すなわちサービス) の名前、FQDN、アイコンのアドレス。

ブラウザは、これらを PublicKeyCredentialCreationOptions Object にし、WebAuthn API 経由で認証器へ Credential 生成を要求。

²⁵ UserInfo, RP Info は WebApp すなわちユーザが自分で定めることも (一応) できる。

WebAuthn ユーザ登録: 認証器へ Credential 生成要求

②, ③ ブラウザの API を Call して認証器へ Credential 生成を要求。



PublicKeyCredentialCreationOptions Object をブラウザの
window.navigator.credentials.create() へ入力。

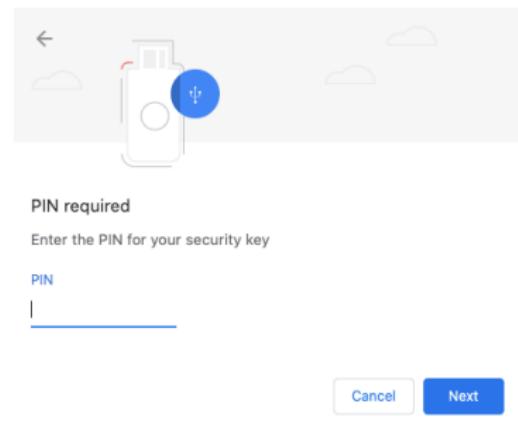
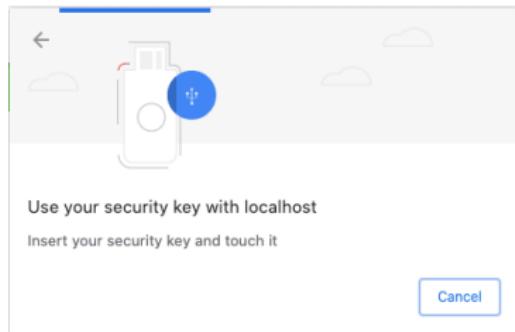
実際に JavaScript のコードを見ていく。

PublicKeyCredentialCreationOptions の構造 (./test/credential-params.ts)

```
const createCredentialDefaultArgs: CredentialCreationOptions = {
    publicKey: {
        // Challenge 本当はサーバーで生成した暗号学的に安全な乱数をセット (16bytes 以上)
        challenge: new Uint8Array([0x8C, 0x0A, 0x26, 0xFF, 0x22, ...]).buffer,
    },
    // Relying Party Info (a.k.a. - Service)
    rp: {
        id: 'localhost', // テストコードはローカルで走るため
        name: 'Example RP'
    },
    // User Info
    user: {
        id: new Uint8Array(16),
        name: 'john.p.smith@example.com',
        displayName: 'John P. Smith',
    },
    // 利用したい Public Key Credential Params のリスト (認証器は先頭から試行):
    pubKeyCredParams: [
        {
            type: 'public-key', // As of March 2019, only 'public-key' is accepted.
            alg: -7 // Signature Algorithm (ECDSA with SHA-256)
        },
    ],
    // Attestation Type (optional, default は 'none' (RP による attestation 検証なし))
    attestation: 'direct', // 'direct' は認証器の生成した Attestation を直接 RP に送るタイプ
    // Time Out (optional, in msec)
    timeout: 60000, // 認証器からの応答をブラウザはどれくらい待つか。
},
};
```

PublicKeyCredentialCreationOptions Object を使ってブラウザの WebAuthn API を以下のように Call すると、**認証器の挿入・接続要求、PIN 入力要求がブラウザ通知される。**

```
window.navigator.credentials.create() の Call (./test/test.spec.ts)  
const cred: Credential|null  
= await window.navigator.credentials.create(createCredentialDefaultArgs);
```



認証器挿入・接続要求

PIN の入力要求

この流れは、Shell からサンプルコードのディレクトリで以下を実行すると確認できる。

ユーザ登録→ユーザ認証の一連のテストコードを実行

```
$ yarn test
```

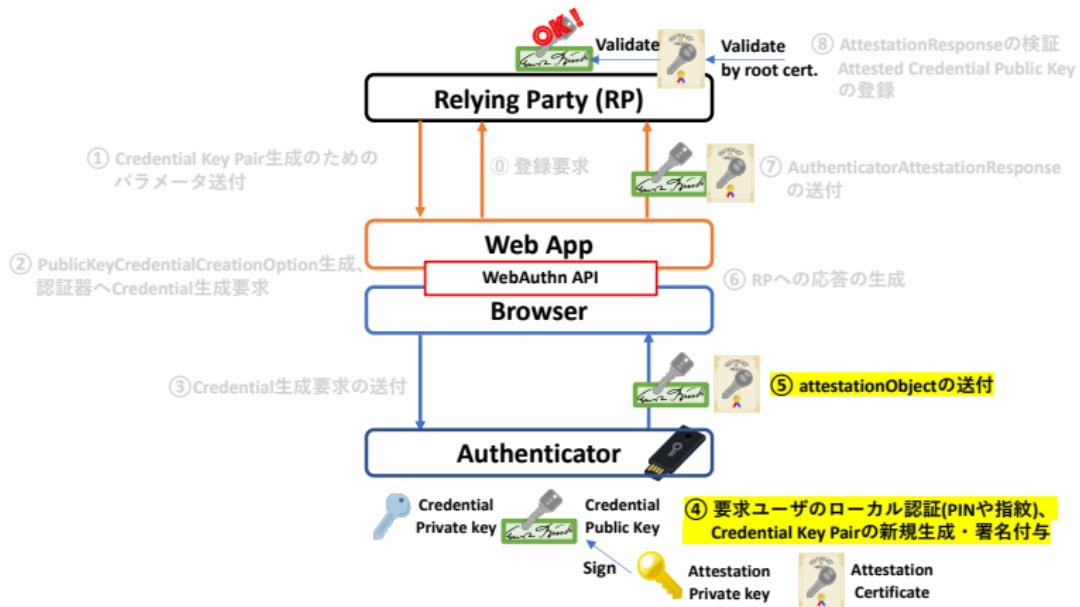
ブラウザにダイアログが出たところで認証²⁶を行えば、認証器内部で Credential が生成&出生証明される。

それでは、次の Credential 生成ステップと、生成した Credential の中身を覗いてみよう。

²⁶Security Key by Yubico の場合は PIN 入力+タッチ

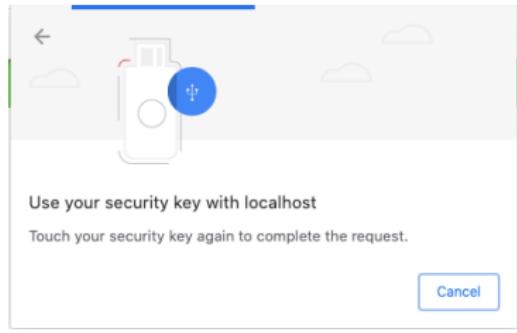
WebAuthn ユーザ登録: Credential 生成・取り出し

④, ⑤ 認証器で Credential 新規生成、Credential Public Key と署名を取得。

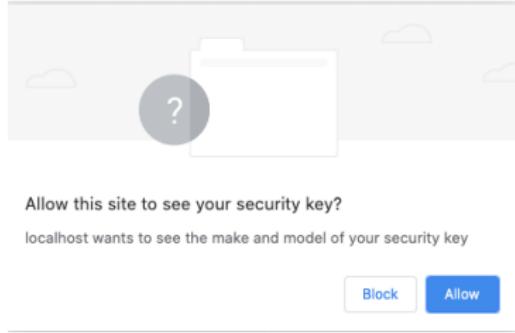


前ステップの `create()` の返り値として、Attestation Certificate や Credential Public Key (attestationObject) を格納した PublicKeyCredential Object をブラウザが取得。

前ステップの後、認証器ローカルでの生体認証²⁷、および FIDO2 WebAuthn の利用確認が求められる。



認証要求

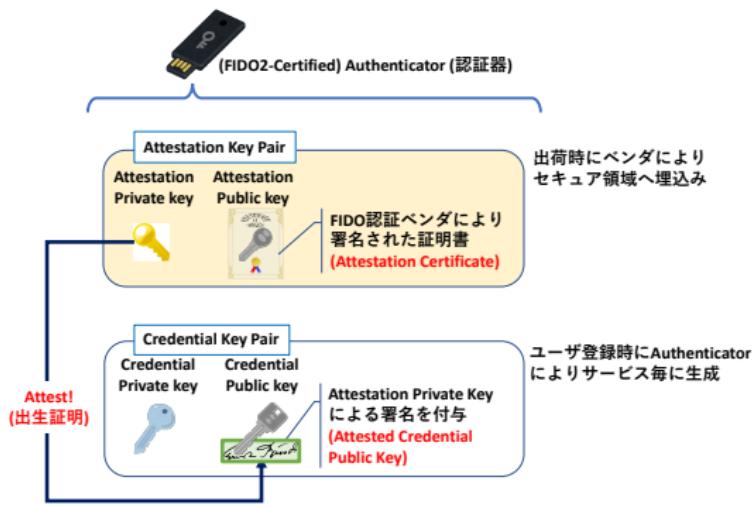


FIDO2 利用意思の確認

²⁷ Security Key by Yubico の場合、PIN 入力の後にタッチすること。

認証と利用意思確認が完了すると、認証器内部で以下の処理を実行。

- 1 口一カルでの生体認証結果の確認
- 2 `create()` で入力されたパラメタに応じて、ユーザの新しい鍵ペア 'Credential Key Pair' を生成
- 3 認証器内部の Attestation Private Key で Credential Public Key に署名
- 4 (Attested) Credential Public Key とその署名を出力²⁸



²⁸ Attestation Type: direct の場合は Attestation Certificate も出力

`create()` の返り値 `PublicKeyCredential Object` は単純に以下の 4 つの要素で構成されている。

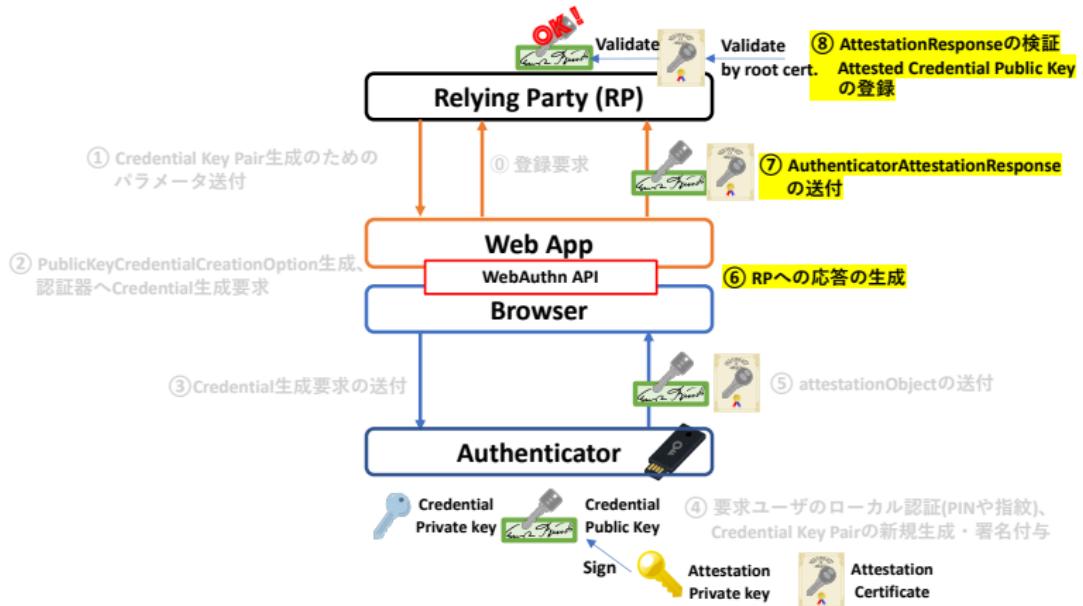
PublicKeyCredential の構造 (\$ yarn test の途中出力)

```
'----- [Response from Authenticator: PublicKeyCredential] -----'  
'> Credential ID: HfM8J_xY7mn7bfiHxF7f7MLxf...' ← 生成した公開鍵の ID  
'> Credential Raw ID: [object ArrayBuffer]' ← 生成した公開鍵の ID のバイナリ版  
'> Credential Type: public-key' ← 公開鍵証明書なので'public-key'  
'> AuthenticatorAttestationResponse.clientDataJSON: [object ArrayBuffer]' ← RP の Challenge の情報 (バイナリ)  
'> AuthenticatorAttestationResponse.attestationObject: [object ArrayBuffer]' ← ここが Credential Public Key と署名が含まれる本体 (バイナリ)
```

このうち、「`clientDataJSON`」と「`attestationObject`」からなる `AuthenticatorAttestationResponse` を RP に送って検証・登録する。次のステップでその検証について解説する。

WebAuthn ユーザ登録: Attestation の検証

⑥, ⑦, ⑧ ブラウザが AuthenticatorAttestationResponse を RP に送って、そこで Attestation の検証とユーザ登録を実行。



Attestation の検証は RP の行うバックエンドの処理なことに注意。

`clientDataJSON` はバイナリにエンコードされた JSON で、以下の要素で構成。

`clientDataJSON29` の構造。 (\$ `yarn test` の途中結果)

```
LOG: '----- [Decoding result of elements of AuthenticatorAttestationResponse] -----'
LOG: '> Decoded clientDataJSON:
{
  "challenge": "o9sKvn8ls2QAMFNyiv_g...", ← 登録処理開始の際、RP が送付した challenge (base64url)。
  "origin": "http://localhost:9876",       ← Relying party の ID。今回の例だと localhost。
  "type": "webauthn.create"              ← ユーザ登録のときは webauthn.create 固定。
}'
```

他、`tokenBindingId` という RP との通信セッションとの紐付けを行うパラメタ (Optional)。

これは、後述する `attestationObject` がどういうパラメタに対して生成されたのかを示す一覧という位置づけ。

²⁹ WebAuthn の認証の時もパラメータの異なるこのオブジェクトが生成される。

<https://developer.mozilla.org/en-US/docs/Web/API/AuthenticatorResponse/clientDataJSON>

`attestationObject` は CBOR で表現され、以下の要素で構成。

`attestationObject30` の構造。 (\$ yarn test の途中結果)

```
[  
  {  
    "fmt": "packed",  
    "attStmt": {  
      "alg": -7,  
      "sig": {  
        "type": "Buffer",  
        "data": "MEUCIBJog73SH9q+gu...", ← "x5c"の証明書で検証可能な署名（本当はバイナリ）  
      },  
      "x5c": [  
        {  
          "type": "Buffer",  
          "data": "MIICvDCCAaSgAwIB...", ← これこそが Attestation Certificate!  
        }  
      ]  
    },  
    "authData": {  
      "type": "Buffer",  
      "data": "SZYN5Yg0jGh0NBcPZHg..." ← 署名対象、つまり Credential Public Key のフィールド */  
    }  
  }  
,
```

`fmt="packed"` の場合、署名は、`clientDataJSON` のハッシュ値と、
`attestationObject.authData` を連結したデータに対して生成。

³⁰ `clientDataJSON` と異なり、登録の時のみ生成されるオブジェクト。

<https://developer.mozilla.org/en-US/docs/Web/API/AuthenticatorAttestationResponse/attestationObject>

RP は attestationObject と clientDataJSON について以下を検証。

① RP 自身が要求した Credential 生成なのか？

⇒ clientDataJSON 内部と、RP で保持していたチャレンジの比較

② RP のサービスで登録すべき Credential 生成なのか？

⇒ clientDataJSON 内部の origin のチェック

⇒ attestationObject.authData に含まれる RP ID のチェック

③ FIDO2 で利用可能な正しい認証器を使った登録か？

⇒ clientDataJSON のハッシュ値と authData について、
attStmt.sig (署名) の正しさを attStmt.x5c (Attestation
Certificate) を使って検証

⇒ ルート証明書により、attStmt.x5c の信頼性を検証

上記のチェックが全部 OK であれば authData (内部の Attested Credential
Public Key) を保存、ユーザ登録完了！

テストコードで模擬する RP の Attestation 検証動作結果を見てみよう。

RP による PublicKeyCredential の検証結果の例 (\$ yarn test の途中出力)

```
LOG: '----- [Verification result on PublicKeyCredential.AuthenticatorAttestationResponse] -----'
LOG: '> Verification result: true'           ← Attestation の検証成功 (challenge/origin/署名の検証成功)

LOG: '> Attested Credential Public Key: ← 検証が成功した Attested Credential Public Key (毎回変化; これを登録)
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEt11IqJnVr2Wi4nIip57LPhoAejRG
TH86zg3S7CUYSFibLqVOQrbQ0ADz9IpYHoKzQCbbHc13o8Zj7WgHUy1yQ==
-----END PUBLIC KEY-----'

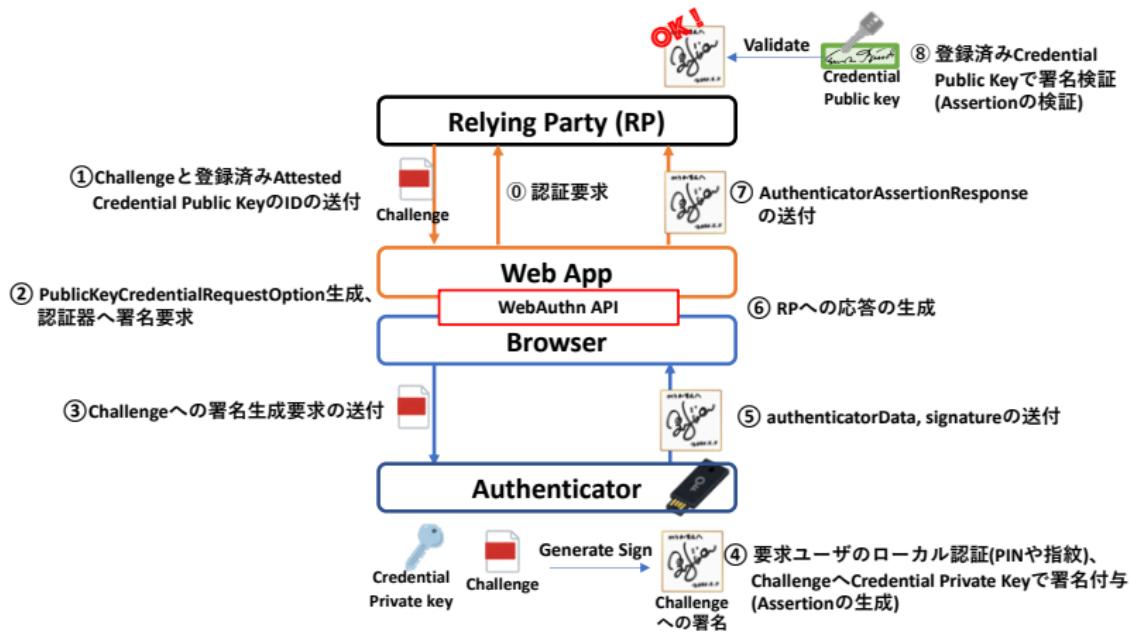
LOG: '> Attestation Certificate:           ← 認証器が送ってきた Attestation Certificate (固定)
-----BEGIN CERTIFICATE-----
MIICvDCCAAsgAwIBAgIEBMX+ /DANBgkqhkiG9w0BAQsFADuMSwwKgYDVQQDEyNZ
dWJpY2g8VTJGIFJvb3Qg0EgU2VyaWFsIDQ1NzIwMDYzMTAgFw0xNDA4MDEwMDAw
MDBaGAyMDUwMDkwNDAwMDAwMFowbTELMAkGA1UEBhMCU0UxEjAQBgNVBAoMCV11
YmljbyBBQjEiMCAGA1UECwwZQXV0aGVudGljYXRvcibBdhRlc3RhGlvbjEmMCQG
A1UEAwwdWXViawNvIFUyRiBFBSBTZXJpYWwgODAwODQ3MzIwWTATBgcqhkJOPQIB
BggqhkJOPQMwNCAAQc2Np2EaP17x+iXpuPl2A4zSFU5FYS9R/W3GcUyNcJChk
45m9tXNgkGQk1dmYu8kUwuZyTfk5T8+n3qixgEo2wwajAiBgkrBgeEAYLEcGIE
FTEuMy42LjEuNC4xLjQxDgyLjEuMTATBgsrBgeEAYL1HAIQAQEAwIFIDAhBgsr
BgEEAYL1HAEBAQSBD4oBHzjApNFYAGFxEfntx9MAwGA1UdEwEB/wQCMAAwDQYJ
KoZIhvCNAQELBQAQDggEBAhCT091LRoF8wpThdwhvbj6wGNxcLaiYqUZXPX+0Db+
AGVODSkVvEVSmj+jXmrBzNQel3FW4AupOgbgrJmmcWWEZyXSpR0tYcl2LTNU0+I
z9WbyHNN1wQJ9ybFwj608xBuoNRC0rG8wgYbMC4usyRadt3dY0VdQi0cfaksVB2V
NKnw+tQUWkoZsPHtuzFx8NlazLQBep1W2T0FCONFEG7x/l+ZcfNhT13azAbaurJ
2J0/ff6H0PXJP6h+Obne4xfz0+8ujftWDUSh9oaiVRYf+tgam/tzOKyEU38V2liV
11zMyHKWrXiK0AfYDgb58ky2HSrn/AgE5MW/oXg/CXc=
-----END CERTIFICATE-----'
```

署名検証のソースコード解説は、ただのフォーマット解説のため省略。

FIDO2 WebAuthn ユーザ認証フロー

WebAuthn ユーザ認証フロー

以下のような流れで、RP に登録されたユーザに対し、認証を行う。



RP で生成した Challenge に対して認証器内部で署名させ、その検証結果で事前に登録したユーザかどうかを判別・認証する処理。

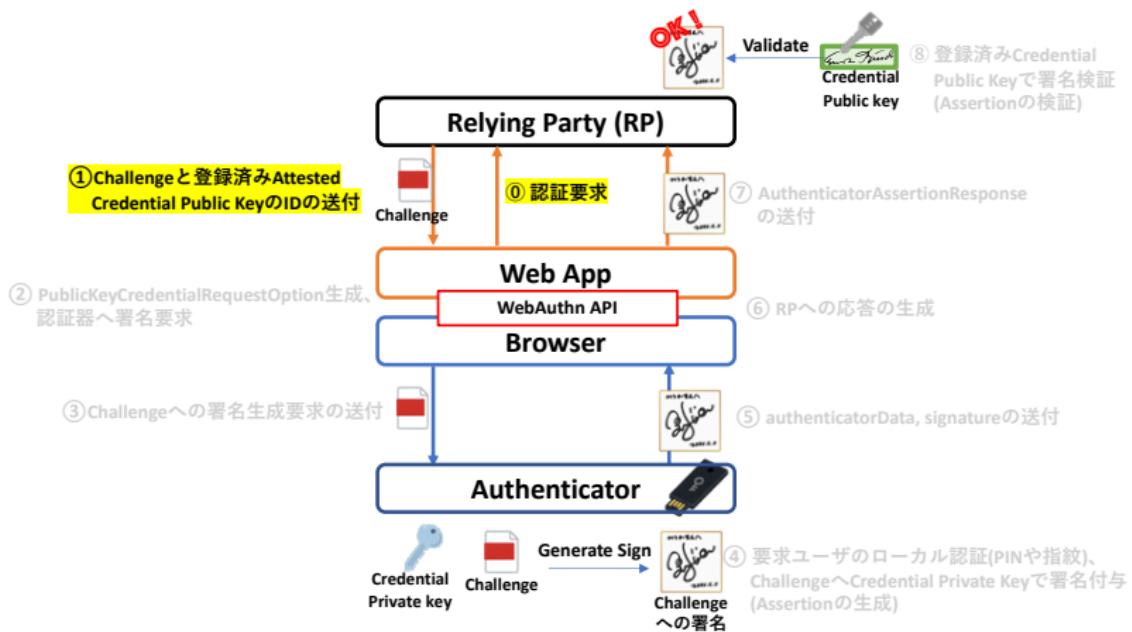
登録処理の場合同様、ユーザ認証の各ステップを実際のデータを確認しながら追っていく。

“Assertion”について

ユーザ登録の時、Credential (Key Pair) の「認証器による出生証明」を“Attestation”と呼んだ。これと同様に FIDO2 WebAuthn では、ユーザ認証において「認証器による正当性の主張」を“Assertion”と呼ぶ。すなわち、Assertion の検証を行うことで、RP は正当性の主張を受け入れ認証 OK とする。

WebAuthn ユーザ認証: RPへユーザ認証要求

① Challenge と署名させる Credential の ID を RP から取得。



登録同様、WebApp・RP 間のやりとりは規定されておらず、実装者に任せられている。

WebApp が RP から取得する必要があるパラメタは以下の通り。

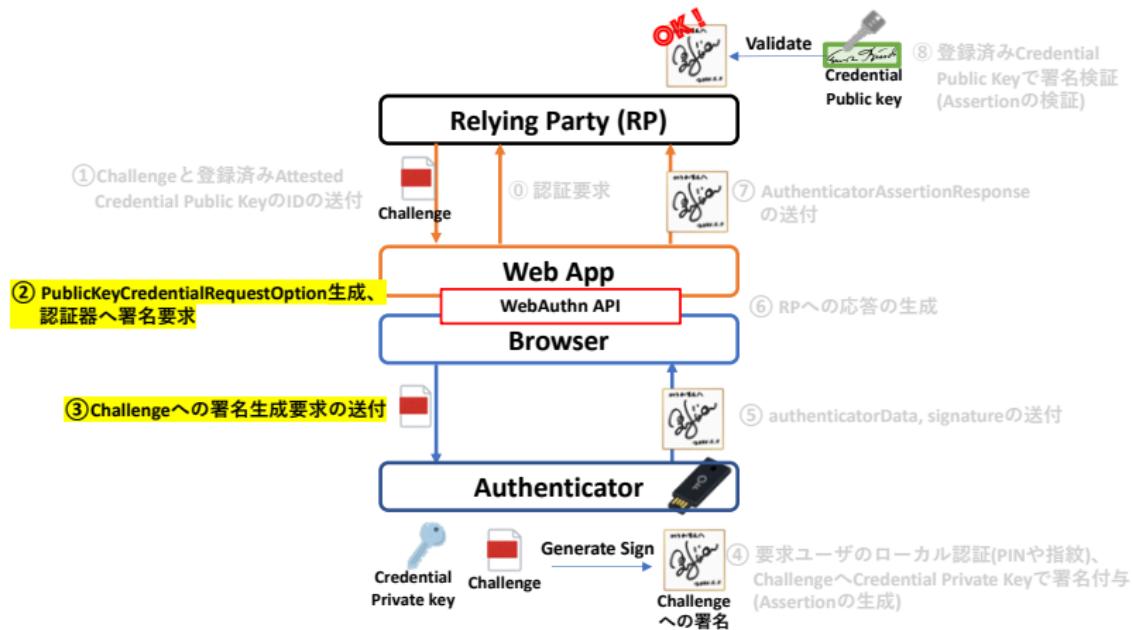
- **Challenge**: 暗号学的にランダムな使い捨ての binary string (最低 16bytes, 通常 32bytes 程度)
- **Credential Info**: ユーザの Credential Public Key の ID。認証器に対応する Credential Private Key (署名鍵) を指定するのに必要。規格上は Optional だが認証器が非対応の場合は指定が必須。³¹

ブラウザはこれらを PublicKeyCredentialRequestOptions Object にし、WebAuthn API 経由で認証器へ Assertion を要求。

³¹ Client-side Discoverable Credential (後述) に対応していることが必要。Security Key by Yubico は非対応。

WebAuthn ユーザ認証: 認証器へ Assertion 生成要求

②, ③ ブラウザの API を通して認証器へ Assertion を要求。



PublicKeyCredentialRequestOptions Object をブラウザの
`window.navigator.credentials.get()` へ入力。

実際に JavaScript のコードを見ていく。

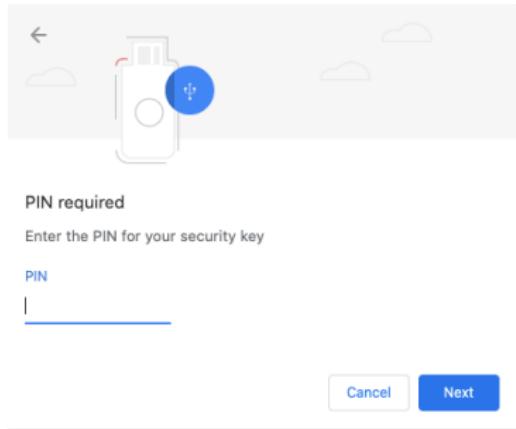
PublicKeyCredentialRequestOptions の構造 (./test/credential-params.ts)

```
export const getCredentialDefaultArgs: CredentialRequestOptions = {
  publicKey: {
    // Challenge 本当はサーバーで生成した暗号学的に安全な乱数をセット (16bytes 以上)
    challenge: new Uint8Array([
      0x79, 0x50, ...
    ]).buffer,
    // Info of credential public keys allowed to use authentication (optional)
    // 認証器次第ではここを RP が指定しなくても OK
    // (RP ID に応じてユーザが鍵を選べる, Client-side Discoverable Credential と呼ぶ)
    allowCredentials: [
      {
        id: new Uint8Array([0xA1, 0x55, ...]).buffer,
        transports: ['usb', 'nfc', 'ble'],
        type: 'public-key'
      },
      // rpId indicating Relying Party ID (optional, default = current domain)
      rpId: 'localhost', // テストコードはローカルで走るため
      // User verification (biometrics authentication, optional, default = 'preferred')
      // PIN が未指定の場合などは、'required' にすると検証不可として認証エラー
      userVerification: 'required',
      // Time out (optional, in msec)
      timeout: 60000,
    ],
  };
};
```

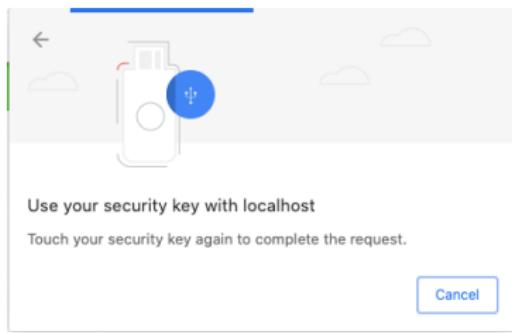
PublicKeyCredentialRequestOptions Object を使ってブラウザの WebAuthn API を以下のように Call すると、登録の時と同様に認証器の挿入・接続要求、PIN 入力要求、認証要求がブラウザ通知される。

```
window.navigator.credentials.get() の Call (./test/test.spec.ts)
```

```
const cred: Credential|null  
= await window.navigator.credentials.get(getCredentialDefaultArgs);
```



PIN 入力要求



認証要求

認証器にタッチすることで生体認証を完了させると、**認証器内部**で Assertion = Challenge への署名生成が行われる。

補足: Client-side Discoverable Credential について

PublicKeyCredentialRequestOptionsにおいて、allowCredentialsなしとする Client-side Discoverable Credential³² が利用できる。この機能は、以下の特徴を持つ。

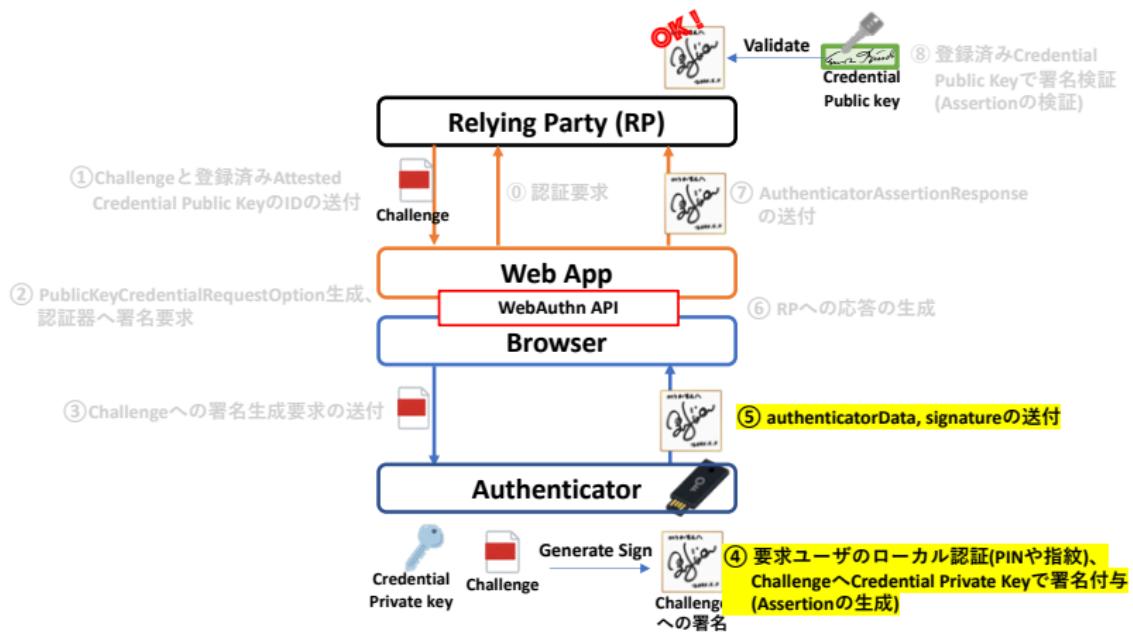
- 認証時に RP から Credential ID を取得する必要がなく、ユーザ自身が RP ID に応じて Credential を切り替え可能
- RP 側でユーザ名などと Credential ID の紐付け管理が不要

ただし、利用には**認証器がこの機能に対応している必要がある。**

³²現在の仕様 (ver. 4 Mar. 2019) 上は Resident Credential と呼ばれているが、Working Draft で名称が変わった。<https://w3c.github.io/webauthn/#client-side-discoverable-credential>

WebAuthn ユーザ認証: Assertion 生成・取り出し

④, ⑤ 認証器で Challenge へ署名し、Assertion を取得。



前ステップの `get()` の返り値として、署名 (`signature`) やメタ情報 (`authenticatorData`) を格納した `PublicKeyCredential Object` を取得。

get() の返り値 PublicKeyCredential Object は以下の要素で構成される。create() では AuthenticatorAttestationResponse が入っていたが、**AuthenticationAssertionResponse** が入ることに注意。

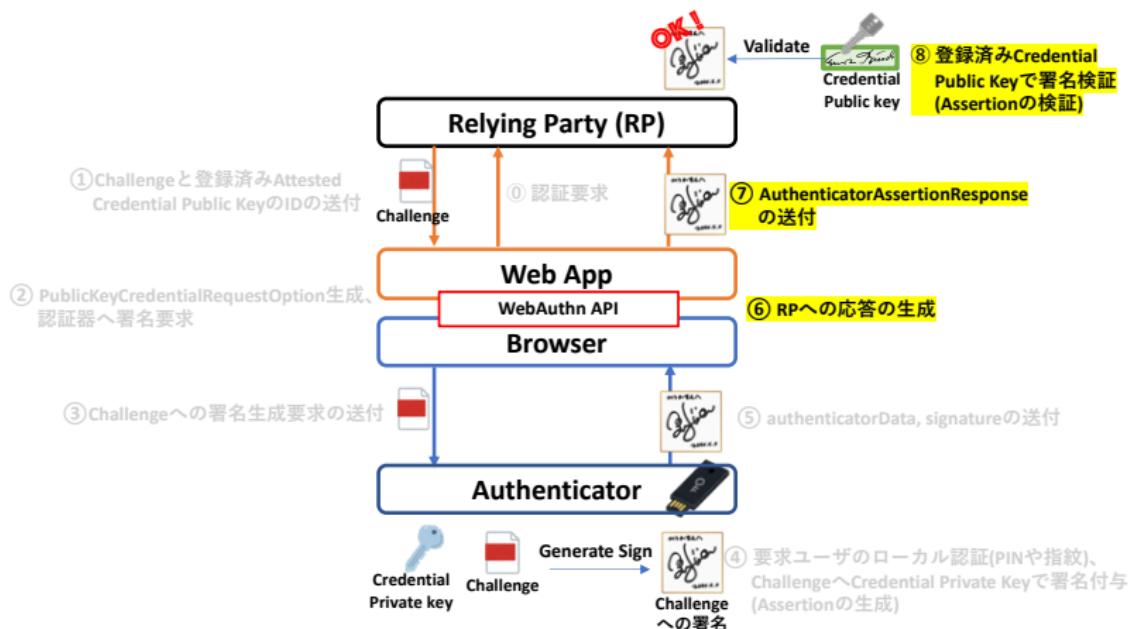
PublicKeyCredential の構造 (\$ yarn test の途中出力)

```
LOG: '----- [Response from Authenticator: PublicKeyCredential] -----'  
LOG: '> Credential ID: jsQqwn1tT5C-I01ELUVq7m...' ← 署名検証に用いる公開鍵の ID = Credential Public Key ID  
LOG: '> Credential Raw ID: [object ArrayBuffer]' ← ID のバイナリ版  
LOG: '> Credential Type: public-key' ← 署名は公開鍵で検証されるものなので'public-key'  
LOG: '> AuthenticatorAssertionResponse.clientDataJSON: [object ArrayBuffer]' ← RP の Challenge の情報  
LOG: '> AuthenticatorAssertionResponse.authenticatorData: [object ArrayBuffer]' ← 認証器の情報や RP の ID など  
LOG: '> AuthenticatorAssertionResponse.signature: [object ArrayBuffer]' ← RP の Challenge に対する応答=署名！  
LOG: '> AuthenticatorAssertionResponse.userHandle: null' ← Create 時に入力したユーザ ID  
が入ることが多い
```

このうち、AuthenticatorAssertionResponse 内部の 4 要素を RP に送って検証 (Assertion の検証)、認証可否を判断する。次ステップでその手順を解説する。

WebAuthn ユーザ認証: Assertion の検証

⑥, ⑦, ⑧ ブラウザが AuthenticatorAssertionResponse を RP へ送つて、そこで Assertion の検証と認証可否判断を行う。



Assertion の検証は RP の行うバックエンドの処理なことに注意。

AuthenticatorAssertionResponse の要素の構造は以下の通り。

clientDataJSON の構造, **authenticatorData**, **signature** (\$ yarn test の途中結果)

```
LOG: '----- [Decoding result of elements of AuthenticatorAssertionResponse] -----'
LOG: '> Decoded clientDataJSON: {'
  "challenge": "rHNRQ6copASyNLyFv0Ja...", ← 認証処理開始の際、RP が送付した challenge (base64url)
  "origin": "http://localhost:9876",       ← RP ID
  "type": "webauthn.get"                  ← ユーザ認証の時は webauthn.get 固定。
}'

LOG: '> Base64 authenticatorData: SZYN5Yg0j...' ← 認証器の情報や Credential が紐付けられている RP ID などのデータ

LOG: '> Base64 signature: MEYCIQDI0cKyqpkA...' ← clientDataJSON と authenticatorData に対して作られた署名
```

clientDataJSON のハッシュ値と **authenticatorData** を連結したデータに
対して Credential Private Key で署名生成されている。

⇒ RP はこの連結データに対して signature が正しいものかどうかを検証。

RP は authenticatorData、signature、clientDataJSON に対して以下を検証する。 (=Assertion の検証)

① RP 自身が要求した Assertion 作成なのか？

⇒ clientDataJSON 内部と、RP で保持していたチャレンジの比較

② RP のサービスで認証すべき要求なのか？

⇒ clientDataJSON 内部の origin のチェック

⇒ authenticatorData に含まれる RP ID のチェック

③ 事前登録したユーザ・認証器によって作られた Assertion か？

⇒ Credential ID が紐づいている Attested Credential Public Key を登録ユーザの DB から取得。clientDataJSON のハッシュ値と

authenticatorData について、signature (署名) の正しさを、
Credential Public Key で検証。

テストコードで模擬する RP の Assertion 検証結果を見てみよう。

assertion 検証結果 (\$ yarn test の途中結果)

```
LOG: '----- [Verification result on PublicKeyCredential.AuthenticatorAssertionResponse] -----'  
LOG: '> Verification result: true' ← Assertion の検証成功 (challenge/origin/署名の検証成功)
```

Assertion 検証のソースコード解説は、ただのフォーマット解説のため省略。

まとめ

この資料では以下を行った。

- 認証の基礎と、FIDO2 の概要の紹介
- FIDO2 WebAuthn の概要の紹介
- FIDO2 WebAuthn のユーザ登録・認証についてコードレベルで動作解説

ただし、今回触ってみたことが WebAuthn の全体像ではないことに注意。仕様は日々進化しており、またパラメタもここで掲載したもの以外にも多く存在する。あくまで 1 つの例と考えてほしい。

参考資料: WebAuthn の標準文書・仕様書

- Web Authentication (W3C 勧告)

<https://www.w3.org/TR/webauthn-1>

- Web Authentication (W3C Working Draft)

<https://w3c.github.io/webauthn>

- Web Authentication API (MDN)

https://developer.mozilla.org/en-US/docs/Web/API/Web.Authentication_API