

# End-to-End セキュリティを担保する 2 セキュリティエンジニアリング特論 第4回

栗原 淳

兵庫県立大学大学院

2020-10-22

# はじめに

# はじめに

前回(第3回)は

- End-to-End (E2E) セキュリティの原則と必要性
- Web サイトでの E2E セキュリティ実践のため、JavaScript での暗号 (AES) の利用のさわり

を勉強した。

E2E セキュリティの重要性はわかった。

共通鍵暗号 (AES) を使ってみることもできた。

でも、実際の App で正しく・安全に共通鍵暗号を使うにはどうすべきなのか？

今回は、システムにおいて正しく・安全に共通鍵暗号を使ってみる方法、についてのお話。

### 今回の講義で最終的に学びたいこと

- パスワードを使った共通鍵暗号化はどうすればいいか？<sup>1</sup>
- 固定のマスターシークレット（バイナリ値）<sup>2</sup>を使った共通鍵暗号化はどうすればいいか？<sup>3</sup>

たったこれだけ。

<sup>1</sup>RFC8018 PBES2 <https://tools.ietf.org/html/rfc8018> による AES 暗号化

<sup>2</sup>よくサーバの.env ファイルとかに Base64 で書くアレ。

<sup>3</sup>RFC5869 HKDF <https://tools.ietf.org/html/rfc5869> による鍵導出と AES 暗号化

たったこれだけでも、気をつけなければならない「重要な作法」がある。

お作法を守る・守らないで安全性は大違いなので、注意しなければならない。<sup>4</sup>

今回は、「その作法の担保」を標準技術を使ってやってみる。

---

<sup>4</sup>世の中のソフトウェア、全くお作法を守ってないのが散見されてとても危険。最近だと php の `hash_hkdf()` がお作法守ってなかった(2018年)。

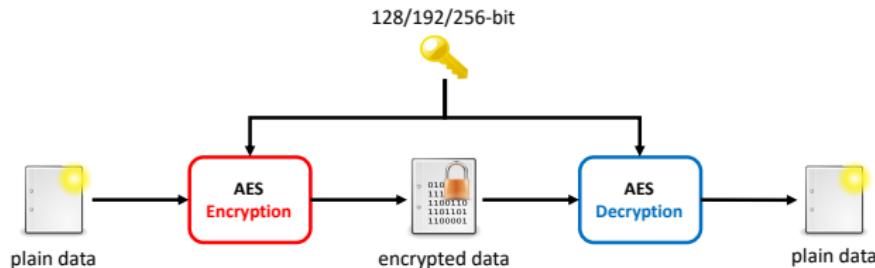
# 共通鍵暗号の使い方 事始め

# AES (Advanced Encryption Standard) とは? [復習]

## AES

米国 NIST の標準<sup>5</sup> 共通鍵暗号アルゴリズム

- 鍵長は 3 種類: 128-bit, 192-bit, 256-bit
- 欧州 NESSIE, 日本 CRYPTREC などの標準規格としても採択
- 現在まで致命的な欠陥は見つかっていない, 安全性の高いアルゴリズム



アルゴリズムとしての AES が標準技術なのは前回した通り. ではその「使い方」は?

<sup>5</sup> NIST FIPS 197 <https://www.nist.gov/publications/advanced-encryption-standard-aes>

# 共通鍵暗号を使うために

共通鍵暗号を使う際に気をつけるお作法は、ざっと3点。

- ① 共通鍵のランダム具合
- ② 共通鍵を総当りする際の大変さ<sup>6</sup>
- ③ 共通鍵暗号の利用モードの安全性

つまりどういうこと？

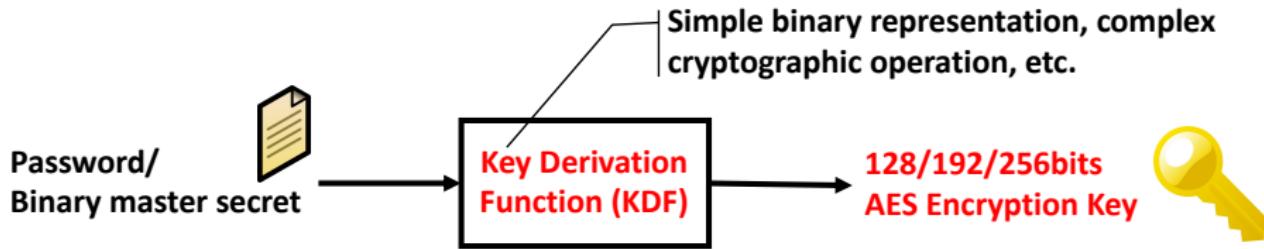
---

<sup>6</sup>1点目と2点目は似ているようで異なる。

# 準備: パスワードとかを使った共通鍵暗号化のポイント

パスワード ≠ 共通鍵暗号化の鍵

パスワードやマスターシークレットを元にして共通鍵暗号化するためには、「パスワード等を変換し、鍵を導出」することが必要



# 1: 共通鍵のランダム具合？

⇒ 過去の利用履歴も含めて「ランダム」であるということ

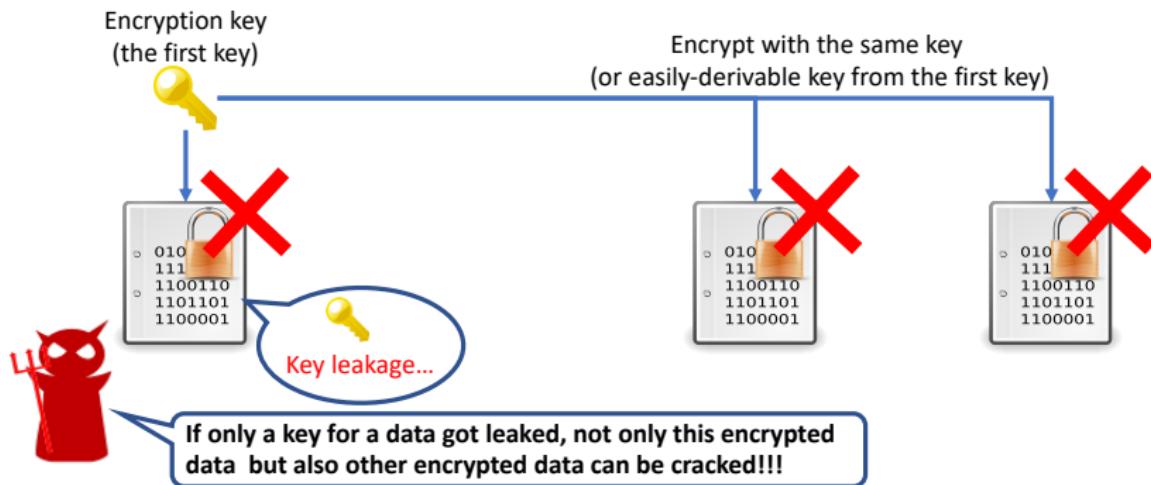
つまり…

- 過去に暗号化を使った鍵は二度と使わない
- 鍵は、過去に利用した鍵から<sup>7</sup>は容易に導出できないものへと毎回ランダムに変更・変換する

<sup>7</sup> および未来に使う鍵からも

…なぜか？

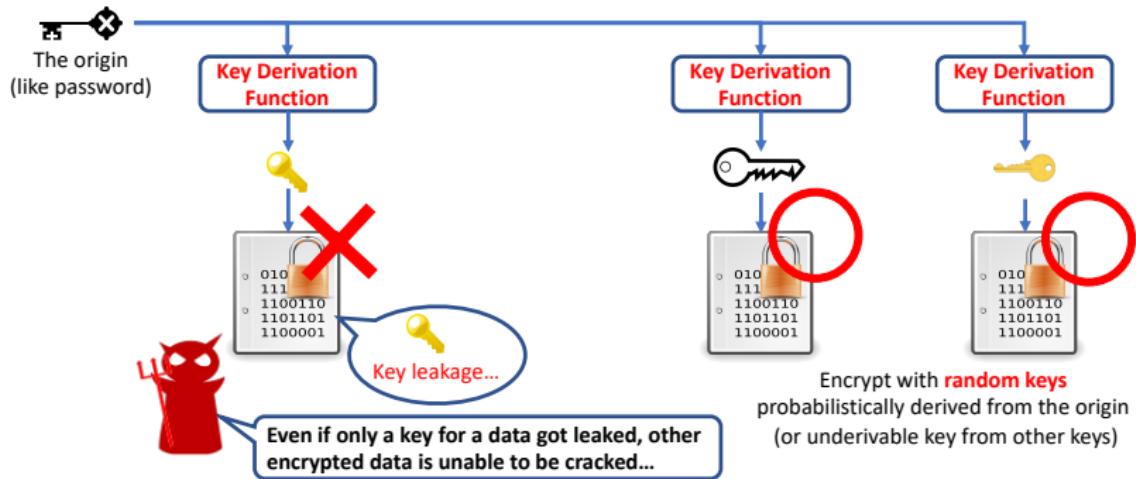
⇒ 鍵が1つ漏れてしまうと、過去の暗号化データまで一網打尽…



なので、万一鍵が1つ漏れちゃったとしても、他の暗号化データにまで影響が出ないことを保証しなきゃならない。

だが、暗号化毎のパスワード等のランダム変更は非現実的.

⇒ 固定パスワード等から不可逆かつランダムに鍵を導出する技術を使ってこれを担保  
⇒ 標準技術 PBKDF2 (RFC8018), HKDF (RFC5869)



※ただし、固定パスワード等そのものが漏洩した場合はこの場合でもアウトなことに注意

## 2: AES で使う鍵を総当たりする際の大変さ？

⇒ 総当たり攻撃のためのコストのこと。

※特にパスワードを使って暗号化する場合に重要

暗号化データに対する総当たり (ブルートフォース) 攻撃

鍵の候補を全通りを一覧で用意して、「当たり」を見つけるまでとにかく  
復号を繰り返すこと。

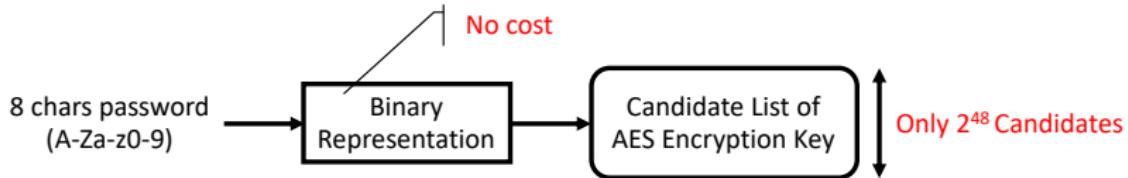
つまり総当たり攻撃のコストは、「ストレージ量」と「計算量(時間)」。  
このコストを払うことが非現実的に高くなれば安全性が担保できない。

8桁パスワードを ASCII コード的に単純バイナリ化で鍵とすると…

大小英数字 8 桁パスワードは  $62^8 < 2^{48}$  通り.

- ⇒ 48bits の全通りの準備は、高々 1.5PB.
- ⇒ ストレージなしでも、パスワード候補を都度バイナリ化するだけで復号を試行可能.
- ⇒ ストレージ・計算量のバランスをとったレインボーテーブルを使った手法も存在.

割と簡単に「当たり = 48bits」が見つかってしまう。<sup>8</sup>



<sup>8</sup>2009 年当時でもスパコンを使って 60 時間とか。今だと GPU で並列化すればもっと高速になる。<https://web.archive.org/web/20180412051235/http://www.lockdown.co.uk/?pg=combi&s=articles>

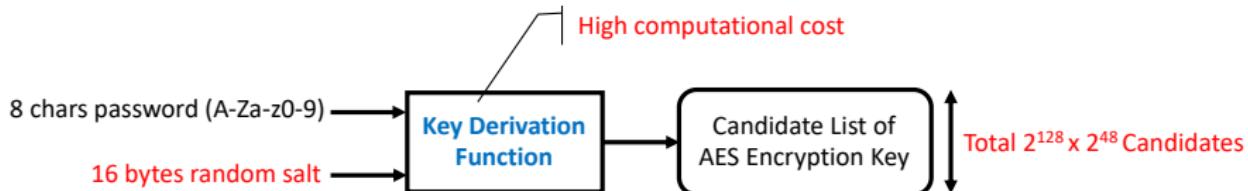
なので、短いパスワード等から鍵を作るときは、総当たりのコスト（ストレージ&時間）が膨大になるような変換をする。

パスワード等から暗号化の鍵を作るとき、

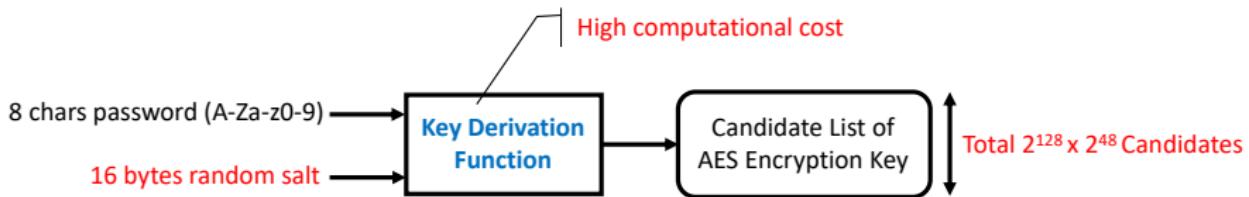
- 毎回使い捨てのランダム値 (Salt と呼ぶ) と混合して、**鍵のランダム性を上げる。**
- 計算コストの高い不可逆の演算を使う。

という処理を行う。<sup>9</sup>

⇒ 標準技術 PBKDF2 により担保可能



<sup>9</sup>PBKDF2



- ランダムな Salt と混合することで、**鍵候補全通りの事前準備のストレージが膨大になる**
- ストレージなしで試行しても、計算コストの高い演算のせいで、**鍵候補を都度生成→復号の計算コストが莫大になる**

「作法 1」と合わせて 1 つの関数で実行することが多いが、AES 暗号化の鍵を作る際に意識する重要なポイント。

### 3: 暗号の利用モードの安全性？

⇒ AES の API で設定できる利用モード ('AES256-CBC' とか) と、  
そのパラメタの適切な設定をしないと致命的な事態に陥る。

#### AES の「利用モード」

AES の処理 1 回で暗号化できるのはたった 16bytes にすぎない。  
長いデータを連続で暗号化するために、**暗号化処理を連続して組み合わせる方法**が利用モード。

「とりあえず AES を使う」ための利用モード設定のポイントは 2 つ

- 初期ベクトル (IV) というパラメタは都度ランダム値にする<sup>10</sup>.
- CTR モード・CBC モードあたりを使う。 ECB モードは絶対に使わない。

前者、「過去に暗号化したデータとの相関をなくす」ために必要なパラメタ設定。

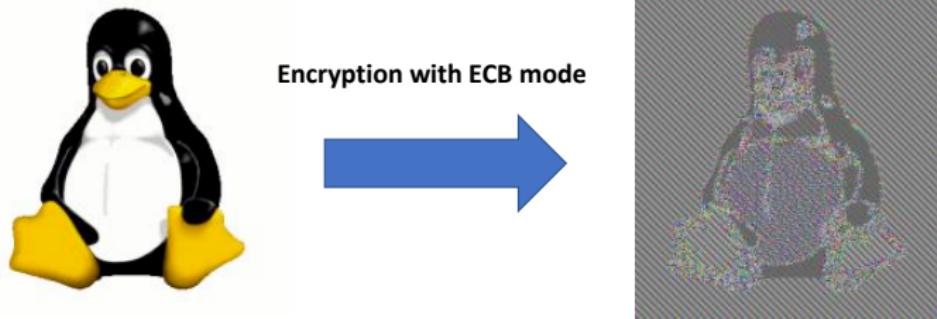
後者、 ECB モードは論外 (これが言いたいこと)。

---

<sup>10</sup> API によって、 ナンス (Nonce) というパラメタもあればそれも。

どうして ECB モードは論外なのか？

- ⇒ 元のデータの中で「同じ値のブロック<sup>11</sup>」は、暗号化データにおいても必ず「同じ値のブロック」になる。
- ⇒ 暗号化されてても中のデータが何かというのが予測可能…



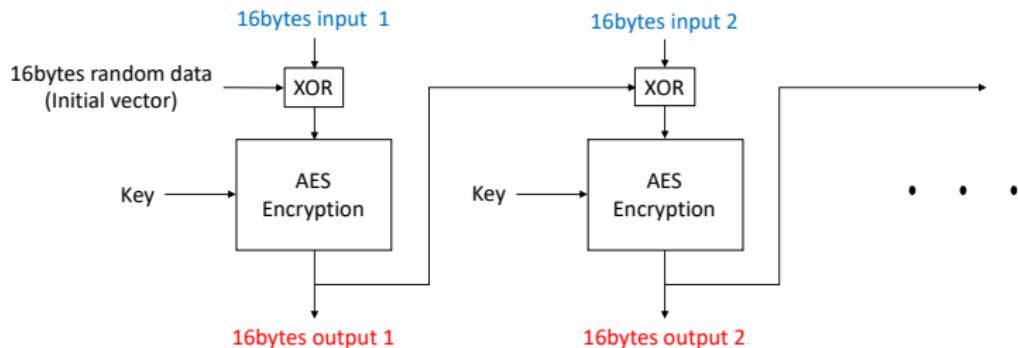
Original images are given by Larry Ewing  
([lewing@isc.tamu.edu](mailto:lewing@isc.tamu.edu))

というわけで、たとえ選べたとしても絶対に ECB モードは利用してはいけない。

<sup>11</sup>1 ブロックは 16Bytes 単位

ECB モードと違って、 CBC モードではそういうことが起きない。

- 先頭の 16Bytes はランダムな初期化ベクトルと混ぜる
- 前の 16Bytes の暗号化データを混ぜて次の 16Bytes を処理



CBC モードの 16Bytes 每の処理

## 暗号利用モードの標準について

- ECB モードは実は ISO, NIST SP800 等では標準化(推奨)されているが、

*In the ECB mode, under a given key, any given plaintext block always gets encrypted to the same ciphertext block. If this property is undesirable in a particular application, the ECB mode should not be used.*

と明示的に「使うべきでない」ケースが述べられている。<sup>12</sup>

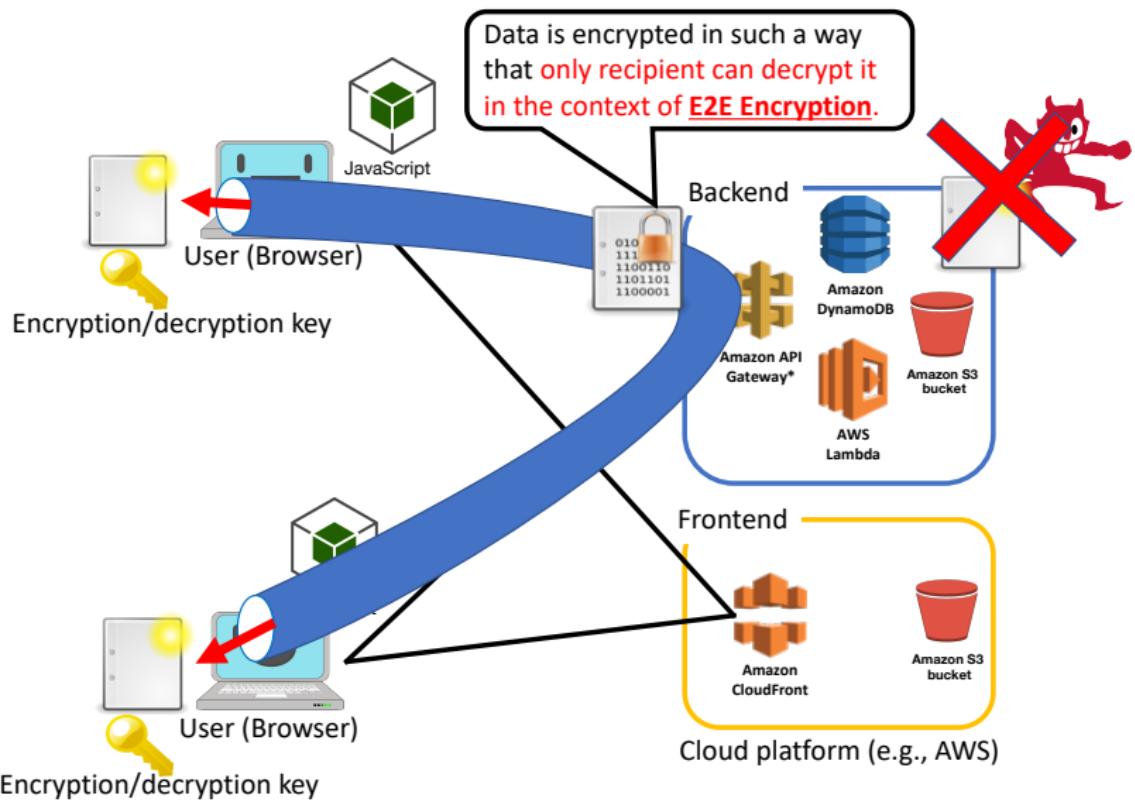
- 他、CBC/OFB/CFB/CTR 等が標準アルゴリズムと同時に利用モードとして標準化されている。
- 標準に載っているので、「一応」暗号ライブラリによっては ECB が Call できるようになっていたりする。注意しなければいけない。

<sup>12</sup><https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf>

# 共通鍵暗号の使い方: まず AES で暗号化を試す

# 今回のセッティング

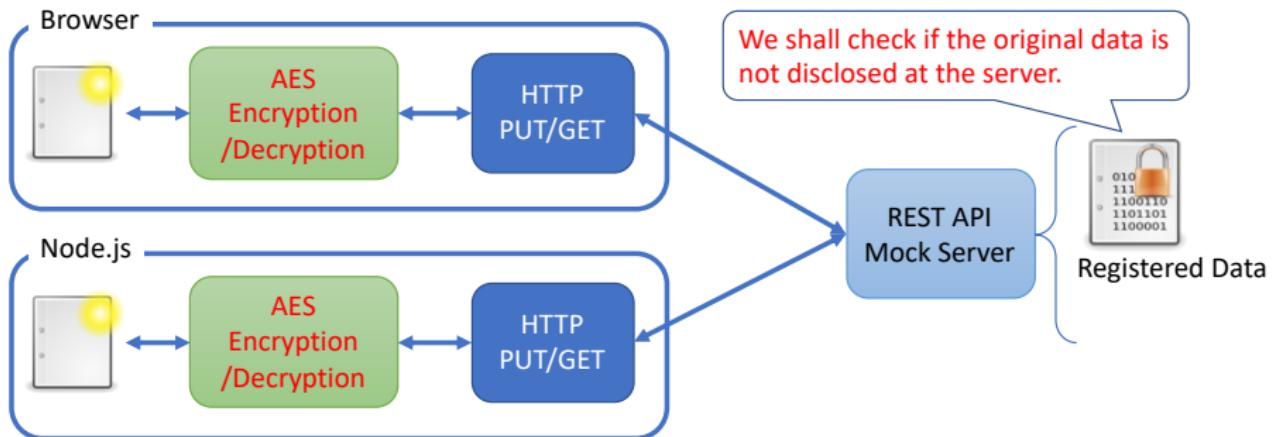
前回同様の REST API サーバを介した E2E 暗号化.



ブラウザ・Node.js をエンドとし、

- 1 「パスワード」「マスターシークレット」から鍵を導出し<sup>13</sup>
- 2 それを使って AES-CBC モードで暗号化して<sup>14</sup>

REST API で暗号化データを登録してみる。



<sup>13</sup>作法 1,2

<sup>14</sup>作法 3

# 環境

以下の環境が前提:

- Node.js ( $\geq v12$ ) がインストール済. yarn が使えること。<sup>15</sup>
- 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.

---

<sup>15</sup>インストールコマンド: `npm i -g yarn`

# JavaScript プロジェクトの準備

- プロジェクトの GitHub リポジトリ<sup>16</sup>を Clone

```
$ git clone  
https://github.com/junkurihara/slides-e2e-security-js.git  
$ cd sample-04
```

- 依存パッケージのインストール

```
$ yarn install
```

- ライブラリのビルド

```
$ yarn build
```

---

<sup>16</sup>[https://github.com/junkurihara/lecture-security\\_engineering](https://github.com/junkurihara/lecture-security_engineering)

# REST API モックサーバの準備

今回は SSL 接続可能な共有サーバを準備済  
(<https://e2e.typeq.org/>)<sup>17</sup>.

別途、検証用のサーバをローカルで立ち上げ可能。

モックサーバの立ち上げ

```
$ yarn start
```

起動すると、localhost の 3000 番ポートで HTTP リクエストを待ち受け開始する。

---

<sup>17</sup>2020/10/22 以降はサーバ停止。モックサーバを利用のこと。

まずはコマンドラインを叩き、Node.js で

- パスワードで AES 暗号化
- マスターシークレットで AES 暗号化
- ECB モードの危険性を実体験

してみる。

※今回は栗原がリリースしているユニバーサル暗号ユーティリティ<sup>18</sup>を使ってサンプルを制作しているので、**ブラウザでも全く同じ Code Snippet を試用可能**。

---

<sup>18</sup><https://github.com/junkurihara/jscu>

# パスワードで暗号化してみる

「yarn execute post -r -p ‘パスワード’ ‘データ’」で暗号化.

sample ディレクトリ以下で実行

```
$ yarn execute post -r -p 'my password' 'my private data' // -r を抜くとローカル  
Register encrypted data to remote server  
Data: my private data  
Password: my password  
Derived key and its related params: // パスワードから生成された鍵とパラメタ  
  Derived key in Base64: fiP4flrlhd3Iwg5M0yln7zNNk4Au9If429n2uvfi43s=  
  PBKDF2 Param - Salt in Base64: zyD7/TGDq3dig3l4zJ5SRzFKVnIjw2KG26XUrMZFkkw=  
  PBKDF2 Param - Hash: SHA-256  
  PBKDF2 Param - Iteration: 2048  
Registered id: 1 // id=1 で暗号化データと鍵導出のパラメタを登録
```

長い鍵 「S4lFVWrVlj4OjPfFRTgVJFfRUI+6LlIw1VooFzG2J5E=」 を短い 「my password」 から生成.

また、同じパスワード・データでも毎回異なる鍵になることを確認する.

登録データは <https://e2e.typeq.org/data> で一覧.

```
[{"data": "1Up45UUO9pkD81q5E5o4w==", "iv": "RkfsY25RlB0MWh5/qnGOjw==", "kdfParams": {"salt": "zyD7/TGDq3dig314zJ5SRzFKVnIjw2KG26XUrMZFkkw=", "hash": "SHA-256", "iterationCount": 2048}, "id": 1}]
```

AES暗号化されたデータ  
AES暗号化に使われるIV  
パスワードから鍵を導出するためのパラメタ

暗号化データ以外、復号側と共有する公開パラメタ：

- AES の CBC モード → IV
- パスワードから鍵の導出 → Salt, iteration 回数, Hash 関数

「yarn execute get -r -p ‘パスワード’ ‘id 番号’」で復号.

```
$ yarn execute get -r -p 'my password' 1 // -r を抜くとローカル  
Retrieve encrypted data to remote server  
Id: 1  
Password: my password  
Derived key and its related params: // 取得した公開パラメタと、生成した鍵  
  Derived key in Base64: fiP4flrlhd3Iwg5M0yln7zNNk4Au9If429n2uvfi43s=  
  PBKDF2 Param - Salt in Base64: zyD7/TGDq3dig3l4zJ5SRzFKVnIjw2KG26XUrMZFkkw=  
  PBKDF2 Param - Hash: SHA-256  
  PBKDF2 Param - Iteration: 2048  
Decrypted data: my private data // 正しく復号された
```

暗号化の時と同一の鍵が生成されたことに注目.

中のコードがどうなっているかは後述.

# マスターシークレット(バイナリ)で暗号化してみる

「yarn execute post -r -m ‘マスターシークレット’ ‘データ’」で暗号化。<sup>19</sup>

sample ディレクトリ以下で実行

```
$ yarn execute gen-secret 32 // まずは Base64 でマスターシークレットを生成する.  
Generated master secret in Base64: mP95WFEv3G/iWsjQKC4mEuEmCkiS8dRK80Q6CpC1bc0=  
  
$ yarn execute post -r -m 'mP95WFEv3G/iWsjQKC4mEuEmCkiS8dRK80Q6CpC1bc0=' 'my  
private data'  
Register encrypted data to remote server  
Data: my private data  
Master secret: mP95WFEv3G/iWsjQKC4mEuEmCkiS8dRK80Q6CpC1bc0=  
Derived key and its related params: // マスターシークレットから生成された鍵とパラメタ  
Derived key in Base64: 1vgTfxp3FEi3kpJiQ6h0vxtDCkdz+u5XQUF1tPm1VMy=  
HKDF Param - Salt in Base64: 8SM9tyXJUX+JGwLswIUnnGyHPL+7hzkSHXaKY7z0AF0=  
HKDF Param - Hash: SHA-256  
Registered id: 2
```

同じマスターシークレット・データでも毎回異なる鍵になることを確認する。

<sup>19</sup>マスターシークレットは Base64

ブラウザで確認してみる。

```
},  
{  
    "data": "GFEtip320wwAt3OpRSOpvq==",  
    "iv": "9htsBwr3D1HmP0Eq6As0WA==",  
    "kdfParams": {  
        "salt": "8SM9tyXJUX+JGwLswIUnnGyHPL+7hzkSHXaKY7z0AF0=",  
        "hash": "SHA-256"  
    },  
    "id": 2  
}  
1
```

AES暗号化されたデータ  
AES暗号化に使われるIV  
パスワードから鍵を導出するためのパラメタ

暗号化データ以外、復号側と共有する公開パラメタ：

- AES の CBC モード → IV
- マスターシークレットから鍵の導出 → Salt, Hash 関数

「yarn execute get -r -m ‘マスターシークレット’ ‘id 番号’」で復号.

```
$ yarn execute get -r -m 'mP95WFEv3G/iWsjQKC4mEuEmCkiS8dRK80Q6CpC1bc0=' 2 // -r を抜くとローカル
Retrieve encrypted data to remote server
Id: 2
Master secret: mP95WFEv3G/iWsjQKC4mEuEmCkiS8dRK80Q6CpC1bc0=
Derived key and its related params: // 取得した公開パラメタと、生成した鍵
    Derived key in Base64: 1vgTfxp3FEi3kpJiQ6h0vxtDCkdz+u5XQUF1tPm1VMY=
    HKDF Param - Salt in Base64: 8SM9tyXJUX+JGwLswIUnnGyHPL+7hzkSHXaKY7z0AF0=
    HKDF Param - Hash: SHA-256
Decrypted data: my private data // 正しく復号された
```

暗号化の時と同一の鍵が生成されたことに注目.

中のコードがどうなっているかは後述.

ブラウザでもパスワード暗号化・マスターシークレットでの暗号化が試せる。

`sample-04/src/post-get-browser.html` を開いて開発者コンソールから試してみよう。

(サンプルコードは html ファイルに記載)

# 危ない暗号化モードで暗号化してみる

ECB モードで暗号化できる API を用意してみたので、それで暗号化してみるとヤバさが目に見えてわかる。

```
$ yarn execute aes-mode-compare '0123456789ABCDEF0123456789ABCDEF' ← 16bytes 每  
random key (Base64): 4gfrl+/0MyFt2ALLEp24sIXyHsyjv1YZZxRj4lkJe9M=  
data (Hex): 3031323334353637383941424344454630313233343536373839414243444546  
AES-ECB (Hex): c871e345b92951236059676b0866c7af c871e345b92951236059676b0866c7af  
...  
AES-CBC (Hex): d34ad4cc8816edcf3ad1a56c355c9067 69c4f525903b607960e377649abef648  
...
```

16bytes 単位で同じ値が出てくるデータを、ECB モードで暗号化してみると ECB モードだと暗号文も同じ値の繰り返しになる。  
⇒ 元のデータが推測しやすくなる。

CBC モードだと暗号化データが繰り返されるようなことはない。

20

ECB モードについては、WebCryptoAPI などではその危険性のためにサポートされていない<sup>21</sup>が、**どんな場合であっても ECB モードの利用は避けて**、CBC モードや CTR モードなどを利用しよう。

---

<sup>20</sup>というか、繰り返しが発生してしまうようなものは ECB だけ。

<sup>21</sup>サンプルコードでは、CBC モードを弄って ECB モードを再現している。

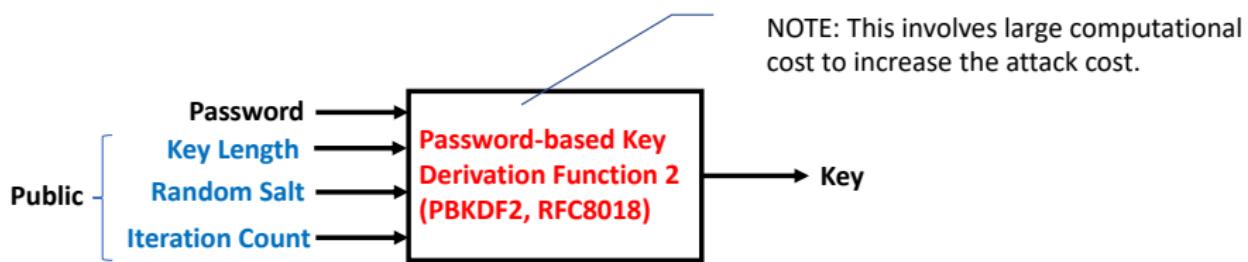
# 共通鍵暗号の使い方: 細かめの解説

# PBKDF2 の使い方 in JavaScript

パスワードから AES の鍵を導出するのに使った。

## PBKDF2: Password-based Key Derivation Function

PKCS #5 v2.1 (RFC8081<sup>22</sup>) にて規定。非推奨の PBKDF1 の置き換え。PBKDF2 を利用した (AES) 暗号化は、Password-based Encryption Scheme 2 (PBES2) と規定される。



他にも、BCrypt など類似のアルゴリズムがある。<sup>23</sup>

<sup>22</sup><https://tools.ietf.org/html/rfc8018>

<sup>23</sup>「パスワードハッシュ化」と「パスワードから鍵導出」で目的は異なれど、必要な機能は一緒。

PBKDF2 は、 WebCrypto API, Node.js Crypto 共にネイティブ実装。

### sample/src/derive-key.js: deriveKeyFromPassword

```
const jscu = getJscu(); // 環境に応じて jscu を script タグで読み込んだり、 require したり。

if(!salt){ // salt が入力されなかったらランダム値を生成。Salt は任意長。
    salt = jscu.random.getRandomBytes(32); // Uint8Array
}
else {
    salt = jseu.encoder.decodeBase64(salt); // Base64 から Uint8Array にデコード
}

const key = await jscu.pbkdf.pbkdf2( // PBKDF2 により鍵導出
    password, // パスワード。
    salt, // 復号側と共有（公開）。
    iterationCount, // 内部処理の反復回数。通常 1000 回以上。復号側と共有（公開）。
    len, // 出力する鍵の長さ。復号側と共有（公開）。
    hash // 内部の HMAC 関数用の Hash 関数名。'SHA-256'。復号側と共有（公開）。
);
```

標準技術が実行環境にネイティブ実装してなかったら自前実装？

実装においてセキュリティホールを産むこともあるので、設計・実装・テストに時間を掛けられないのであれば、闇雲に自力実装すべきではない。よって、技術選択は慎重に。

ただ、攻撃者が全くアクセスできないような箇所で利用する<sup>24</sup> 分には、多くの場合で問題ない。

---

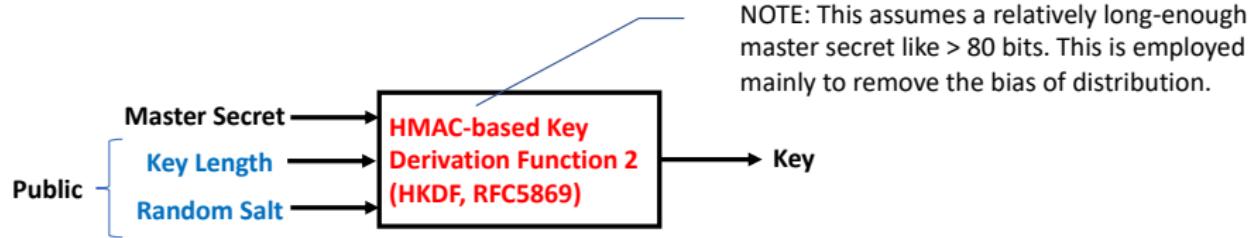
<sup>24</sup> 実装したライブラリを Call する箇所がチャネル攻撃などに無縫のサーバ内の場合、異常なデータを入力されることがないようサニタイジングを正しく行なっている場合、等々

# HKDF の使い方 in JavaScript

マスターシークレットから AES の鍵を導出するのに使った。

## HKDF: HMAC-based Key Derivation Function

RFC8081<sup>25</sup> にて規定。PBKDF2 と違って、鍵の導出計算量を莫大にする効果は薄いので、長めのマスターシークレットを元にする場合に使う。



<sup>25</sup><https://tools.ietf.org/html/rfc5869>

HKDF は、 WebCrypto API でのみネイティブ実装.

### sample/src/derive-key.js: deriveKeyFromMasterSecret

```
const jscu = getJscu(); // 環境に応じて jscu を script タグで読み込んだり、 require したり。  
  
if(!salt){ // salt が入力されなかったらランダム値を生成。Salt は任意長。  
    salt = jscu.random.getRandomBytes(32); // Uint8Array  
}  
else {  
    salt = jseu.encoder.decodeBase64(salt); // Base64 から Uint8Array にデコード。  
}  
  
const keyObj = await jscu.hkdf.compute(  
    masterSecret, // マスターシークレット。  
    hash          // 内部の HMAC 関数用の Hash 関数名。'SHA-256'。復号側と共有（公開）。  
    len,          // 出力する鍵の長さ。復号側と共有（公開）。  
    '',           // 'info' field for RFC5869. This could be always blank.  
    salt          // 復号側と共有（公開）。  
);
```

# 暗号化モードの設定

今回は栗原作成のラッパーを被せているが、CBC モードと CTR モードに加えて、CTR モードを拡張した GCM<sup>26</sup>もネイティブで Call できる。

## sample/src/encrypt.js: encrypt

```
const jscu = getJscu(); // 環境に応じて jscu を script タグで読み込んだり、require する。

const uint8iv = jscu.random.getRandomBytes(16); // ランダム IV の生成。CBC は 16Bytes.

const encrypted = await jscu.aes.encrypt( // AES 暗号化
    jseu.encoder.stringToArrayBuffer(data), // string data の Uint8Array 化
    key, // HKDF/PBKDF で導出した鍵
    { // CBC 暗号化モードを設定
        name: 'AES-CBC',
        iv: uint8iv
    }
);
```

---

<sup>26</sup>GCM(Galois/Counter Mode) は、CTR モードで暗号化したデータに認証タグを付与したもの。暗号化したデータの改ざんを検知できる。

## sample/src/encrypt.js: decrypt

```
const jscu = getJscu(); // 環境に応じて jscu を script タグで読み込んだり、require する。
```

```
const decrypted = await jscu.aes.decrypt( // AES 復号
    jseu.encoder.decodeBase64(data), // Base64 の暗号化データをデコード。
    key, // HKDF/PBKDF で導出した鍵
    { // CBC 暗号化モードを設定
        name: 'AES-CBC',
        iv: jseu.encoder.decodeBase64(iv) // Base64 で与えられた IV をデコード。
    }
);
```

# まとめ

# まとめ

お疲れ様でした。

- 共通鍵暗号化する際の作法を学んだ。
  - 鍵のランダムさを上げる、鍵への攻撃を困難にするために Key Derivation Function として設計された標準技術を適切に使う。  
(※今回紹介した PBKDF2/HKDF 以外の方法もある。<sup>27)</sup>)
  - 利用モードの選択方法。
- 作法を守った JavaScript での実装例を触ってみた。

---

<sup>27</sup>e.g., JOSE 向けの Concat KDF with AESKW <https://tools.ietf.org/html/rfc8037>

# 今後の予定(暫定)

- 1 導入&とりあえず暗号化コードを触ってみる
- 2 共通鍵暗号化によるデータ秘匿を組み上げる
- 3 公開鍵暗号化による鍵秘匿を組み上げる ← 次回
- 4 ハッシュ・MAC・署名、それぞれの使い所と使い方
- 5 全てをまとめて End-to-End セキュリティを担保する

次回の内容:

- 公開鍵暗号を使うコツ（数学的なことはやらない）
- RSA-OAEP
- EDCH-Ephemeral + AES