

# JavaScript による **End-to-End** セキュリティ

## 第 4 回 データの真正性・本人確認のためのテクニック 編

栗原 淳

2019 年 10 月 31 日

はじめに

# はじめに

第 1,2,3 回では

- End-to-End (E2E) セキュリティの原則と必要性
- JavaScript で AES を使った暗号化のお作法
- JavaScript で公開鍵暗号 (RSA/楕円曲線) を使った暗号化のお作法

を勉強した。

今回は、第 3 回の最後に懸案事項だった

「データのやり取りしてる相手って本当に正しい相手？」  
を保証する方法を学んでいく。

## 第3回のおさらい: Ephemeral Scheme

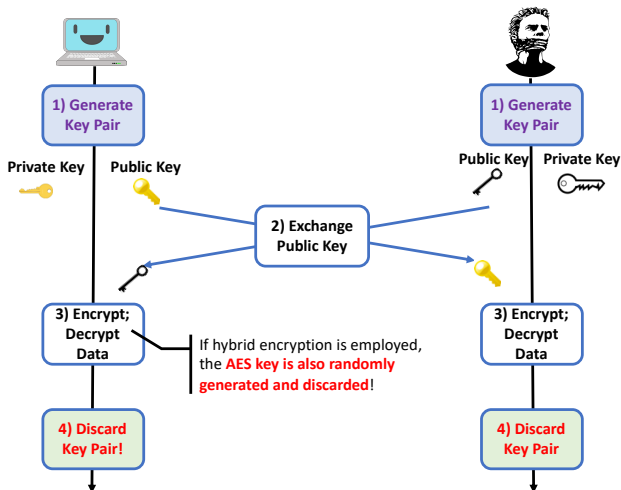
### 公開鍵暗号化の Ephemeral Scheme での運用

公開鍵・秘密鍵ペアを都度生成、1 回限りで使い捨てることで、**Perfect Forward Secrecy**<sup>1</sup> を担保する運用方法。

Perfect Forward Secrecy を守り、End-to-End 暗号化の強固な運用を。

---

<sup>1</sup>長期的に保存されているマスター秘密鍵の漏洩や、一部の暗号化データがクラックされたとしても、**それ以外の過去に暗号化されたデータは復号されてしまうことはない**という概念。



## Ephemeral Scheme のイメージ

まずはじめに、「送られてきた Ephemeral な公開鍵は、本当に自分がやりとりしたい相手の公開鍵か？」の確認が必須。<sup>2</sup>

<sup>2</sup> 意図しない相手の公開鍵で暗号化して機密データを漏らさぬように、ということ。

というわけで、「E2E で安全にデータをやり取りする」ための基礎部分の最後のピースを今日は学ぶ。

### この講義で最終的に学びたいこと

- 本人確認やデータの改ざん防止を担保する方法
  - データ毎に固有の指紋を生成する「ハッシュ」
  - 「共通鍵」を使った改ざん防止方法「MAC」<sup>3</sup>
  - 「公開鍵」を使った本人確認・改ざん防止方法「電子署名」<sup>4</sup>
- そしてその具体的な JavaScript での実装方法・お作法

細かい話もするが、数式は使わない。

「イメージ」と「コードの流れ&その流れの必要性」をつかめるようにする。

<sup>3</sup>HMAC (RFC2104 <https://tools.ietf.org/html/rfc2104>)

<sup>4</sup>RSASSA PKCS#1-v1.5/PSS (PKCS#1 RFC8017 <https://tools.ietf.org/html/rfc8017>), ECDSA (FIPS PUB186-4 <https://csrc.nist.gov/publications/detail/fips/186/4/final>)

### 栗原 淳 (Jun Kurihara)

- (株) ゼタント 主任研究員  
(株) 国際電気通信基礎技術研究所 (ATR) 連携研究員
- 博士 (工学),  
専門: セキュリティ、応用数学、システムアーキテクチャとか
- Web システム (フロントエンド・バックエンド) を作ったり、  
論文他のアルゴリズムを実装したり、研究して論文書いたり、  
セキュリティ技術中心に手広くやっています。
- GitHub: <https://github.com/junkurihara>  
LinkedIn: <https://www.linkedin.com/in/junkurihara>

# 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 に暗号技術を導入したい Web 系のエンジニア

必須ではないが触って楽しむのには必要な事前準備:

- Bash, 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 系ブラウザ and/or Firefox が利用可能なこと



## 今後の予定 (暫定)

- 1 導入&JS の暗号化コードを触ってみる
- 2 AES を正しく・安全に暗号化するには？
- 3 公開鍵暗号はどうやって使う？その使い方のコツは？
- 4 ハッシュ・MAC・署名、それぞれの使い所と使い方は？ ← 今日はココ
- 5 RFC にまつわるあれこれ（証明書・鍵フォーマット・etc... 未定。）

「こういうのを知りたい」というリクエストがあれば是非。

セカンドシーズンも検討中。<sup>5</sup>

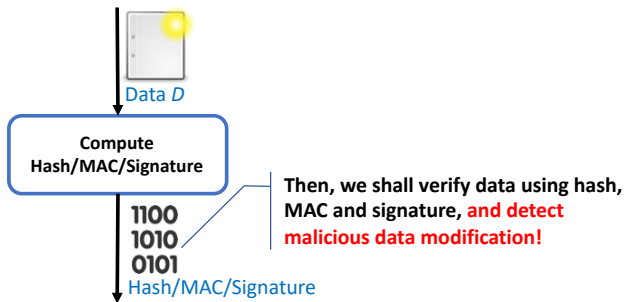
---

<sup>5</sup>場所等変えてもっと来やすい場所へ。。。

# サンプルコードの準備

# 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、JavaScript (Node.js) を使って手元でデータの Hash/MAC/署名をいじってみる。そしてその効果を実感する。



※サンプルコードはブラウザでも動く。

src/commands-browser.html を開くとこれから Node.JS で試すデモが開発者コンソールで実行される。適宜試したり比較すると良い。

※前回のコードの公開鍵に署名をつけたりして Ephemeral Scheme を作ってみると良い。

以下の環境が前提:

- Node.js (> v10) がインストール済。yarn が使えること。<sup>6</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>6</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

## 1 プロジェクトの GitHub リポジトリ<sup>7</sup> を Clone

```
$ git clone https://github.com/zettant/e2e-security-04  
$ cd e2e-security-04/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
```

---

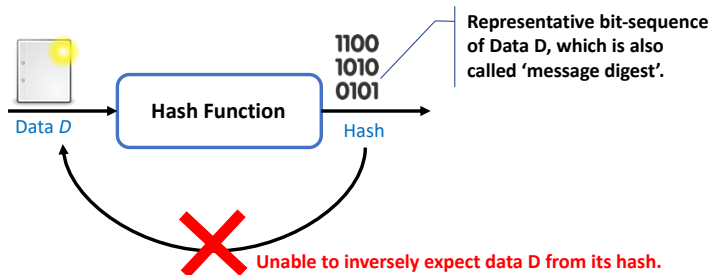
<sup>7</sup><https://github.com/zettant/e2e-security-04>

# データの指紋: Hash

# Hash および Hash 関数とは

## Hash および Hash 関数

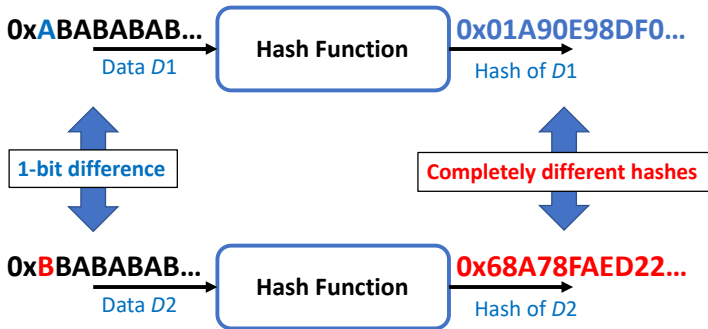
あるデータに対し、そのデータを「代表するビット列」を計算する不可逆の関数を「Hash 関数」。導出したビット列を「Hash」<sup>8</sup>と呼ぶ。



<sup>8</sup>あるいは Hash 値、Message Digest



1 ビットでもデータが異なれば、全く違う Hash が導出される。



# Hash および Hash 関数の役割

同じくデータ固有のビット列を導出する Checksum と似ているが、その用途はより強力で多岐にわたる。

## ■ Checksum

- 通信路上などでのデータの (偶発的な) エラー検知

⇒ データから一意に導ける値・高速な処理が可能なのが必須

## ■ Hash

- データのエラー・改ざん検知
- Hash をデータ実態の代替として署名を生成
- 多数のデータの索引作成<sup>9</sup>
- データの重複検出

⇒ 別のデータ同士で同じ Hash を得ることが困難なのが必須

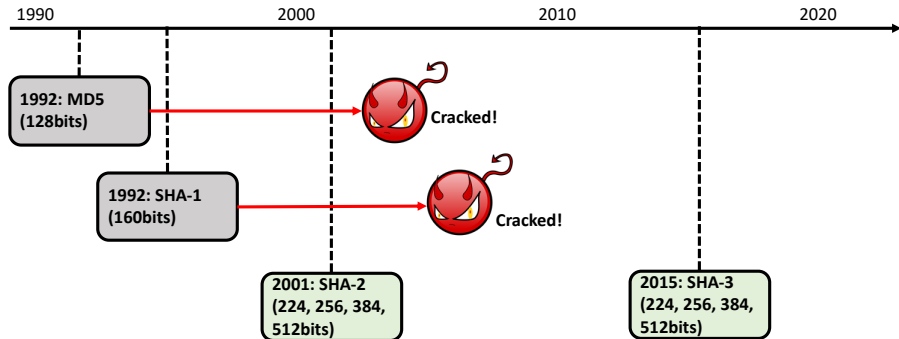
**Checksum  $\subseteq$  Hash** と言える。

---

<sup>9</sup>Hash Table

# Hash 関数の種類

MD5, SHA-1, SHA-2 (SHA-256, 384, 512), SHA-3 という Hash 関数がよく知られている。



MD5、SHA-1 は、「**同じ Hash(指紋) を生成するデータが割と簡単に見つけられる<sup>10</sup>**」という致命的な欠陥が発見されている。

<sup>10</sup> 「衝突」と呼ぶ。MD5 の場合は、 $2^{20}$  程度の計算量でクラック可能。

## Hash 関数の選択について

- 理由がなければ SHA-2 シリーズ以降のものを選択する。
  - bit 長は長いほど、衝突するデータが見つけづらい (=強固)
  - ただし、bit 長が長いほど、計算が重くなる
- SHA-1/MD5 は、基本的に互換性の担保のためだけに利用する。但し、Checksum として使う分には概ね問題ない。何が何でも使うな、というわけではない。

## IE/Edge こぼれ話

X.509 の公開鍵証明書などはまだ SHA-1 が利用されている場合が多々ある。しかし、IE/Edge では互換性の担保を全て無視して SHA-1 のネイティブサポートを全打ち切りしているので、X.509 公開鍵証明書などを JavaScript からネイティブ API を通して扱えない。

**JavaScript** でデータの **Hash** を生成してみる。

Hash 生成のコードはこんな感じ。

# 本人確認の技術

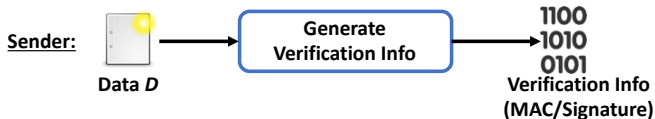
# 「正しい相手から正しく送信されてきたデータか」？

いわゆる「データの真正性と送信元の確認方法」には、大まかに 2 つの方法がある。

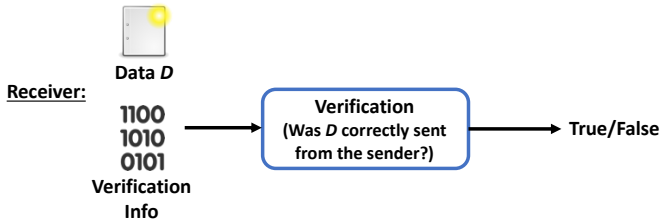
- Message Authentication Code (MAC)
- 署名 (電子署名)



両者とも、送信するデータから MAC/署名という検証用データを生成、元データに付与する形で送信。



受信したデータと、MAC/署名とを突合して、「送信元は意図している相手か?」「データは改ざんされていないか?」を検証。

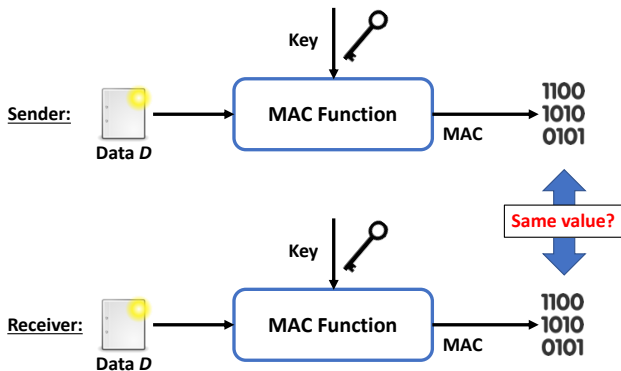


MAC・署名の中身に突っ込む前に、それぞれのざっくりとした定義と pros/cons を説明する。

# Message Authentication Code (MAC)

## MAC によるデータ真正性と送信者の確認

- 送信側・受信側で共有する鍵を使ってデータ・鍵固有のバイナリ (MAC) を生成する方法。
- 受信側で、送信側と同一の MAC が作れるかどうかをチェック。



## MAC の特徴:

- 同じ鍵でも、データが異なれば出力される MAC も異なる。
- 同じデータでも、鍵が異なれば出力される MAC も異なる。



すなわち、受信側で同一の MAC が作れることを確認できれば、

- 鍵を共有する相手から
- 途中の改ざんなしで送られたデータであること

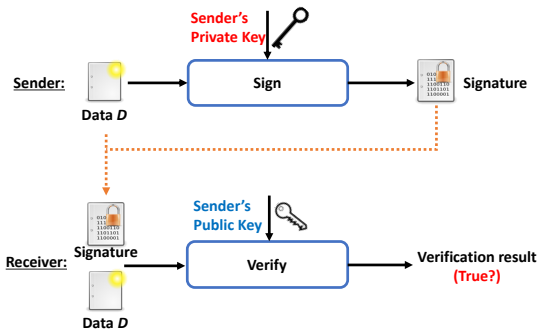
が保証される。

# 署名 (電子署名)

公開鍵・秘密鍵ペアをベースとした技術<sup>11</sup>：

## 署名によるデータ真正性と送信者の確認

- 送信側は公開鍵・秘密鍵ペアを保有。
- 送信側は、データと自分の秘密鍵から署名を生成。
- 受信側は、受信データ、署名と公開鍵の間の一貫性をチェック。



<sup>11</sup> ここでいう公開鍵・秘密鍵ペアは、公開鍵暗号化に使うものと全く一緒の概念。

## 署名の特徴:

- データが改ざんされていたら、検証が失敗 (false が出力)。
- 意図する相手の秘密鍵<sup>12</sup> で署名が作られていなければ、検証が失敗。
- **MAC と違って、秘密の情報 (=鍵) を事前共有しなくて良い**



すなわち、署名技術は、

- 意図する送信者から
- 途中の改ざんなしで送られたデータなことを
- **事前の秘密情報の共有なしで**

保証する。<sup>13</sup>

---

<sup>12</sup> 自分が入手している公開鍵の対となる秘密鍵

<sup>13</sup> 検証用の公開鍵は、信頼できる手段で入手済み、あるいはプリインストールされていると仮定

# MAC と署名の pros/cons

じゃあ署名だけで MAC は不要では？…そういうわけにはいかない。

	Pros	Cons
MAC	<ul style="list-style-type: none"><li>・ 一般的に <b>高速</b><sup>14</sup></li><li>・ 生成する MAC サイズは小さい<sup>15</sup></li></ul>	<ul style="list-style-type: none"><li>・ 鍵の <b>事前共有が必要</b></li></ul>
署名	<ul style="list-style-type: none"><li>・ 鍵の <b>事前共有が不要</b></li></ul>	<ul style="list-style-type: none"><li>・ 一般的に <b>非常に遅い・重い</b></li><li>・ 生成する署名サイズは一般的に <b>大きい</b><sup>16</sup></li></ul>

⇒ AES/公開鍵暗号の関係と全く一緒に、使い所を考えて組み合わせて使う、もしくは場合に応じて使い分ける。

<sup>14</sup> AES (CMAC) とか Hash (HMAC) とかを構成要素としているため。

<sup>15</sup> 通常 128–512bits 程度。

<sup>16</sup> ECDSA は小さく、256–512bits 程度。RSA 系は非常に大きく通常 2048bits 以上。

# 共通鍵を使った改ざん検知・本人確認: MAC



# Message Authentication Code (MAC) 事始め

# Hash-based MAC (HMAC)

## HMAC (RFC2104)<sup>17</sup>

- 鍵付き Hash<sup>18</sup> と呼ばれる、Hash 関数ベースの MAC 生成方法。
- HDKF (RFC5869) などの標準技術や、AWS Signature v4<sup>19</sup> 等、各所で利用されている。

「鍵」と「データ」をまとめて Hash 関数に入れる、と考えると、

- 鍵・データ両者が正しくないと、正しい Hash も生成不能 (=MAC 検証失敗)。
- MAC から鍵・データの情報を逆算することはできない。

という特徴をイメージしやすい。

---

<sup>17</sup><https://tools.ietf.org/html/rfc2104>

<sup>18</sup>Keyed Hash

<sup>19</sup>AWS S3 にクライアントから REST API 経由でアップロードする時に一時的に生成する MAC

# JavaScript で HMAC を実行してみる

コードの中身はこんな感じ。

## その他の MAC (JS じゃビミヨー…)

### Cipher-based MAC (CMAC, NIST SP800-38B<sup>20</sup>)

共通鍵暗号 (e.g., AES) の CBC モードを Hash 関数がわりに使用して MAC を計算する。「前のブロックの暗号文を使って次のブロックを暗号化する」という特徴を応用。

### GMAC (Galois MAC, NIST SP800-38D<sup>21</sup>)

共通鍵暗号 (e.g., AES) の Galois Counter Mode (GCM) で暗号化と同時に生成される MAC。高速に計算できる代数演算<sup>22</sup>を Hash 関数がわりに使用して MAC を計算する。GMAC 単独で利用可。

<sup>20</sup> <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf>

<sup>21</sup> <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>

<sup>22</sup>  $\mathbb{F}[x]/(x^{128} + x^7 + x^2 + x + 1) = \mathbb{F}_{2^{128}}$  上の乗算

と、「標準技術」で「広く利用されている」MAC アルゴリズムはあるが、JS のネイティブ API<sup>23</sup> でサポートされている MAC は、現状 HMAC のみ…

CMAC, GMAC が使いたかったら自力実装 or npmjs.com で見つけて利用する。

---

<sup>23</sup>WebCrypto API, Node.js Crypto

公開鍵を使った改ざん検知・本人確認: 署名

# 署名 事始め



# 署名とは？

データそのものに署名を施すのは厳しいので、「データの指紋」に対してハッシュを施そう。

- RSASSA PKCS1-v1.5
- RSASSA-PSS

RSA なら可能なら PSS を使おう！

# JavaScript で RSASSA-PSS を実行してみる

# ECDSA

# JavaScript で ECDSA を実行してみる

# 運用 Tips

# 署名検証のブートストラップの問題

署名の検証用の公開鍵が正しいことはどうやって保証するの？

⇒ App に固定インストールか、Verisign あたりに署名してもらう必要…

⇒ PKI に頼って検証用の公開鍵の信頼性を担保するしか、現状は方法がない。

# 署名・MACの使い分け

処理の重さで使い分けると幸せになれる。

- 署名は手続きのイニシエーションに使う
- MACは、なんども繰り返すような本人確認に使う

例えば。。

- 1 署名を付与して、ECDH-ephemeralの公開鍵を交換。
- 2 ECDH-ephemeral + AESでMACの鍵を共有。
- 3 以降の大規模データのやり取りはMACで本人確認を実施。

など。



# まとめ

# まとめ

お疲れ様でした。



次回は

# 宣伝: iTransfy by Zettant

簡単・安全にファイル転送ができる

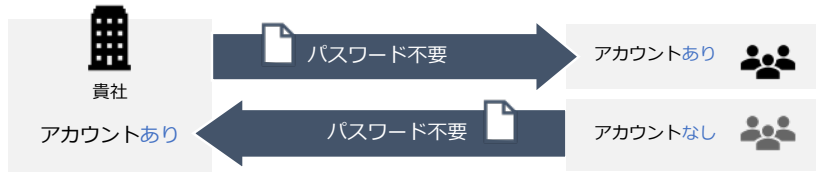


**iTransfy** for biz

<https://www.itransfy.com>

**アカウント登録で、パスワード入力の手間が省けます**

クライアント/協力会社等へファイルを送りたい、また送付してほしい時の手間を軽減



# 宣伝: 株式会社ゼタント



ゼタントはのミッションは、

「自分の身は自分で守ることができる世の中にする」

ことです。

共感してくれる仲間を募集しています！

問合せ先: [recruit@zettant.com](mailto:recruit@zettant.com)

会社 URL: <https://www.zettant.com>