

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^{1),2)}

¹⁾ 株式会社ゼタント

²⁾ 兵庫県立大学

January 26, 2020

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。

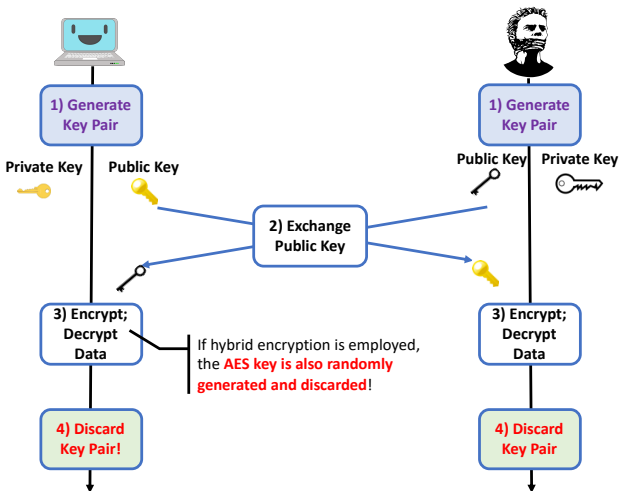
第3回のおさらい: Ephemeral Scheme

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 回限りの暗号化・復号で使い捨てることで、**Perfect Forward Secrecy**¹ を担保する運用方法。

Perfect Forward Secrecy を守ることで、End-to-End 暗号化の強固な運用が可能に。

¹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。²

²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」³
 - 「公開鍵」を使った本人確認・改ざん防止方法「電子署名」⁴
- 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。

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

⁴RSASSA 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>)

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

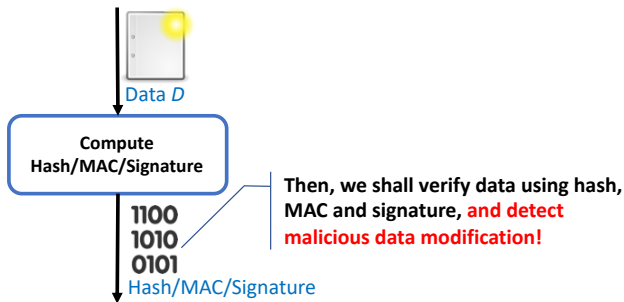
今回の位置付け:

- 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

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。

環境

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。⁵
- 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。

⁵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⁶ 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
```

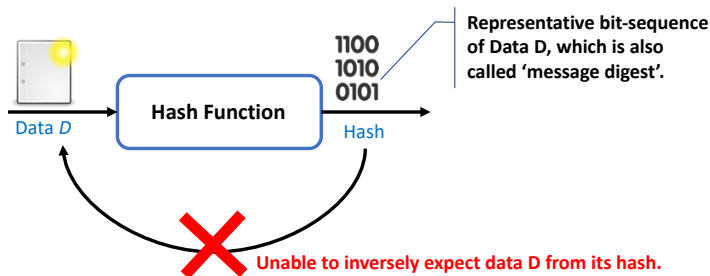
⁶<https://github.com/zettant/e2e-security-04>

データの指紋: Hash

Hash および Hash 関数とは

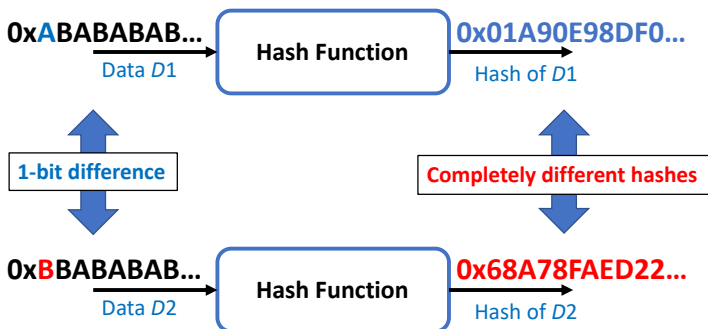
Hash および Hash 関数

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」⁷と呼ぶ。



⁷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⁸
- データの重複検出

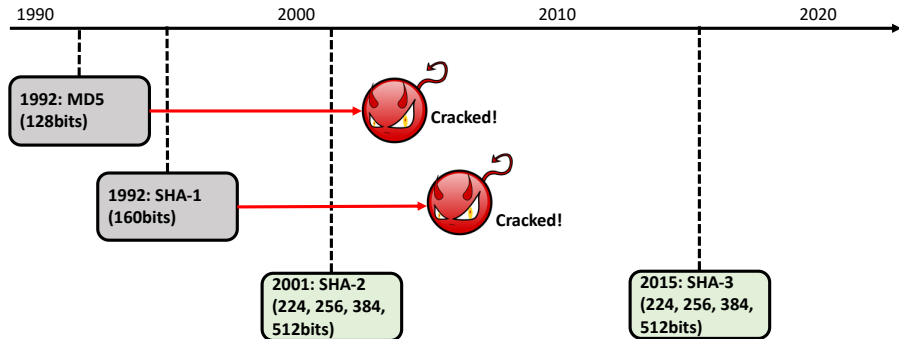
⇒ 別のデータ同士で同じ Hash を得ることが困難なのが必須

Checksum \subseteq Hash と言える。

⁸Hash 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⁹**」という致命的な欠陥が発見されている。

⁹「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。

```
% yarn execute gen-hash 'hello hash world!' // デフォルトの hash 関数は'SHA-256'
<Computed Hash>
70c6b0c909b7a3b4932e6e6d27c6e3c8106b7b9487a4ab9fb27d698b0bee601d // 'SHA-256' で計
算した hash
```

=====

```
% yarn execute gen-hash 'hello hash world!!' // 1 文字増やす
<Computed Hash>
c1a548f16bc6cd013fb76f59c982c6dbc57d390d9a470e09b35d716c7716ab47 // 全く違う hash
```

=====

```
% yarn execute gen-hash 'hello hash world!' -h 'SHA3-256' // -h で hash 関数指定
<Computed Hash>
cb352b3d82d5911b99774fcf534bfd024fc58ef58fb67db14f504931da9a333d
```

=====

ちなみに bash だと、以下で SHA-256 の hash をチェック可能。

```
% echo -n 'hello hash world!' | shasum -a 256
70c6b0c909b7a3b4932e6e6d27c6e3c8106b7b9487a4ab9fb27d698b0bee601d -
```

Hash 生成のコードはこんな感じ。
今回も手前味噌で恐縮だが jscu¹⁰ を利用している。

hash 生成 (src/test-apis.js)

```
// hashName = 'SHA-256', 'SHA-384', 'SHA-512', 'SHA-1', 'SHA3-256', etc...  
  
const jscu = getJscu(); // jscu オブジェクト取得  
const binary = jscu.encoder.stringToArrayBuffer(data); // string を uint8array に  
  
return jscu.hash.compute(binary, hashName); // hash 値の promise を返す
```

Node.js, ブラウザ共に全く同じコードで動作。SHA3 もサポート。

¹⁰<https://github.com/junkurihara/jscu>

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」？

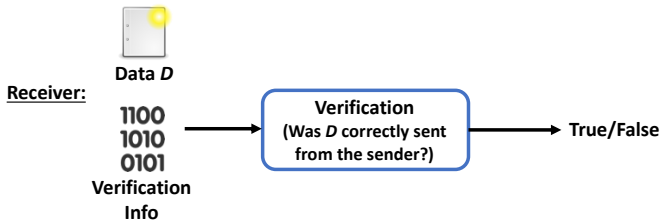
いわゆる「データの真正性と送信元の確認方法」には、大まかに 2 つの方法がある。

- 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。

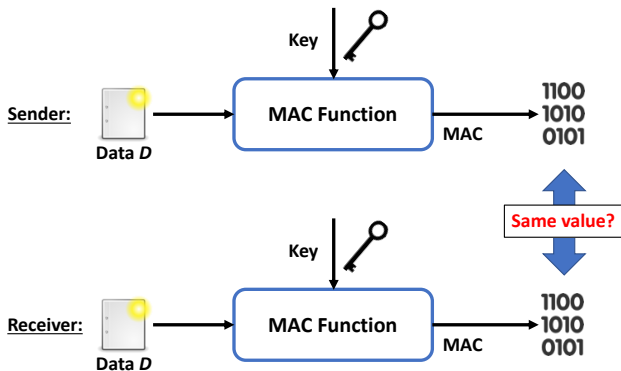


MAC・署名の中身に突っ込む前に、それぞれのざっくりとした定義と pros/cons を説明する。

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

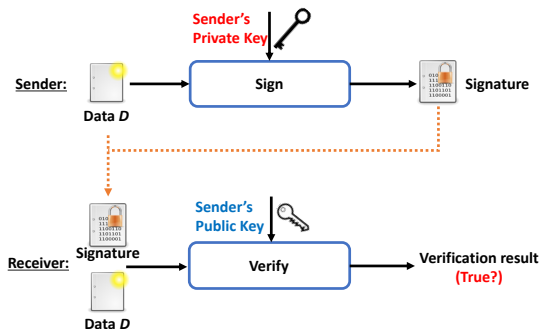
が保証される。

署名 (電子署名)

公開鍵・秘密鍵ペアをベースとした技術¹¹：

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。



¹¹ 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¹²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。¹³

¹² 自分が入手している公開鍵の対となる秘密鍵

¹³ 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">・ 一般的に 高速¹⁴・ 生成する MAC サイズは小さい¹⁵	<ul style="list-style-type: none">・ 鍵の 事前共有が必要
署名	<ul style="list-style-type: none">・ 鍵の 事前共有が不要	<ul style="list-style-type: none">・ 一般的に 非常に遅い・重い・ 生成する署名サイズは一般的に 大きい¹⁶

⇒ 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。

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

¹⁵ 通常 128–512bits 程度。

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

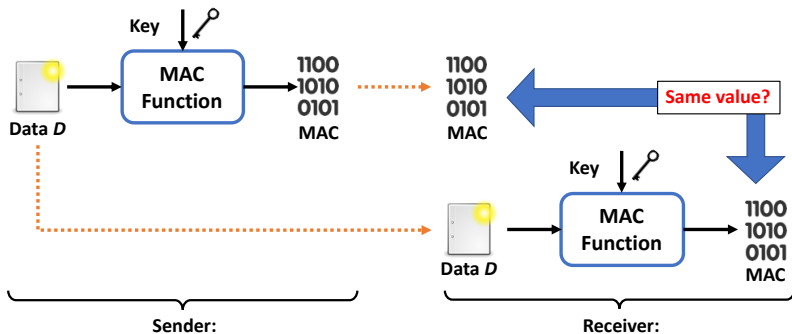
共通鍵を使った改ざん検知・本人確認: MAC

Message Authentication Code (MAC) 事始め

MAC を使った改ざん検知&本人確認手続き

送信側、受信側で秘密の鍵 (バイナリ列) を共有。

- 1 送信側はデータと一緒に、データと鍵から生成した MAC を送信。
- 2 受信側は、鍵と受信したデータから、受け取った MAC と同じものが作れるかどうかをチェック。



MAC を作る標準手法のバリエーション。

- HMAC; Hash-based Message Authentication Code
- CMAC; Cipher-based Message Authentication Code
- GMAC; Galois Message Authentication Code
- etc.

今回は、JS で一番使いやすいと思われる HMAC を取り上げる。

HMAC; Hash-based MAC

HMAC (RFC2104)¹⁷

- 鍵付き Hash¹⁸ 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 v4¹⁹ 等、各所で利用されている。

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

¹⁸ Keyed Hash

¹⁹ 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

「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。

JavaScript で HMAC を実行してみる

```
// HMAC 用の Hex 鍵を生成。鍵長は使う hash のサイズと等しい。
```

```
% yarn execute gen-hex-key 32
```

```
<Generated Hex Key>
```

```
6c9a34e979fc7701330ec75a1bc6acb589ebaf831c7941e042c9ded0b2741d8f
```

```
=====
```

```
// HMAC を生成。デフォルトは SHA-256 で実行。-h で変更可。
```

```
% yarn execute gen-hmac 'hello hmac world!'\
```

```
-k '6c9a34e979fc7701330ec75a1bc6acb589ebaf831c7941e042c9ded0b2741d8f' // Hex 鍵
```

```
<Computed HMAC with SHA-256>
```

```
862e28454f635541ce194d3e4919327c9823830cb7174286aaced5fc61e96a46
```

```
=====
```

```
// HMAC を検証
```

```
% yarn execute verify-hmac 'hello hmac world!'\
```

```
-k '6c9a34e979fc7701330ec75a1bc6acb589ebaf831c7941e042c9ded0b2741d8f' // Hex 鍵
```

```
-m '862e28454f635541ce194d3e4919327c9823830cb7174286aaced5fc61e96a46' // Hex HMAC
```

```
<Verification result of given HMAC>
```

```
true
```

```
=====
```

```
// データが書き換わると検証が失敗
% yarn execute verify-hmac 'hello hmac world!?' \ // ?を追加
-k '6c9a34e979fc7701330ec75a1bc6acb589ebaf831c7941e042c9ded0b2741d8f' \
-m '862e28454f635541ce194d3e4919327c9823830cb7174286aaced5fc61e96a46'

<Verification result of given HMAC>
false
=====

// HMAC が書き換わっても検証が失敗。
% yarn execute verify-hmac 'hello hmac world!' \
-k '6c9a34e979fc7701330ec75a1bc6acb589ebaf831c7941e042c9ded0b2741d8f' \
-m '862e28454f635541ce194d3e4919327c9823830cb7174286aaced5fc61e96a47' //最後を書換

<Verification result of given HMAC>
false
=====
```

もちろん鍵が書き換わっても検証が失敗する。

コードの中身はこんな感じ。

HMAC 生成 (src/test-apis.js)

```
const jscu = getJscu();

// hex string や string を uint8array に。
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const binaryKey = jseu.encoder.hexStringToArrayBuffer(key);

// Promise を返す。hash = 'SHA-256', etc...
return jscu.hmac.compute(binaryKey, binaryData, hash);
```

HMAC 検証 (src/test-apis.js)

```
const jscu = getJscu();

// hex string や string を uint8array に。
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const binaryKey = jseu.encoder.hexStringToArrayBuffer(key);
const binaryMac = jseu.encoder.hexStringToArrayBuffer(mac);

// Promise を返す。hash = 'SHA-256', etc...
return jscu.hmac.verify(binaryKey, binaryData, binaryMac, hash);
```

検証は、「受信側でも MAC を生成→チェック」でも OK。

その他の MAC (JS じゃビミョー…)

CMAC; Cipher-based MAC (NIST SP800-38B²⁰)

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 SP800-38D²¹)

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²²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。

※ CMAC/GMAC 共々暗号化と同時に計算されることが多く、単独で利用するケースはあまり見かけない。

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

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

²² $\mathbb{F}[x]/(x^{128} + x^7 + x^2 + x + 1) = \mathbb{F}_{2^{128}}$ 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 API²³ 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。

²³WebCrypto 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 (電子署名) を使った改ざん検知&本人確認手続き

受信側は、送信側の公開鍵を予めプリインストール。

■ 送信側の処理:

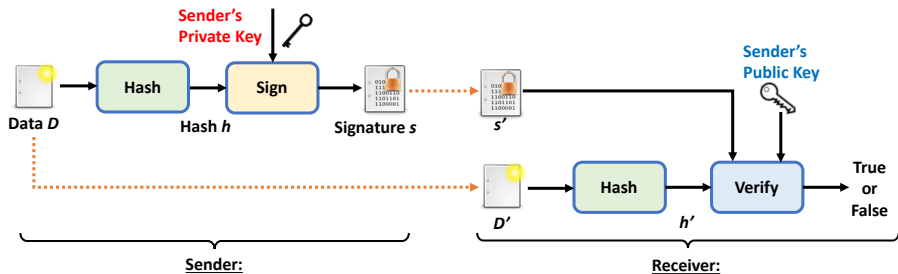
- 1 データ D を Hash 関数で短縮²⁴、hash $h = \text{Hash}(D)$ を導出。
- 2 hash h に対して秘密鍵 SK で署名 $s = \text{Sign}(h, SK)$ を生成。
データ D と署名 s とを一緒に受信側へ送付。

■ 受信側の処理:

- 1 データ D の hash $h = \text{Hash}(D)$ を導出。
- 2 hash h と署名 s の一貫性を、公開鍵 PK で検証。
 $\text{Verify}(h, s, PK) \in \{\text{True}, \text{False}\}$

²⁴データ D そのものに直接署名を施すのは計算量的・データ量的に大変 (e.g. 元データと同じかそれ以上の大きさの署名を作る羽目になる) なので、**データの指紋 (i.e., hash) に対して署名を施す。**

ざっくりフロー図。



このフローは、以下のように考えるとイメージがつきやすい²⁵

- 1 送信側は、hash h を秘密鍵で暗号化して s を生成。
- 2 受信側は、 s を公開鍵で復号して h' を入手。
命題「 $h' = \text{Hash}(D')$ 」が成立するか検証。

²⁵ 但し、常に正しい表現ではないので注意。

署名生成方式の標準方式のバリエーション。

- RSA 暗号をベースとした手法:
 - RSASSA PSS
 - RSASSA PKCS#1-v1.5
- 楕円曲線暗号をベースとした手法:
 - ECDSA
- etc.²⁶

JS で使いやすい RSASSA PSS & PKSC#1-v1.5 と ECDSA について取り上げる。

²⁶Digital Signature Algorithm; DSA (FIPS PUB 186-4
<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>) など

細かく説明する前に…

署名において一番やばいのは、「誰かが自分を騙って署名を作ることができる」などという状況。それを防ぐためには、署名においても「公開鍵サイズの正しい選択」が重要。

前回の復習

暗号化において AES-128bits と同程度の強度の公開鍵サイズは、

- RSA: $\geq 3072\text{bits}$
- ECC: $\geq 256\text{bits}$

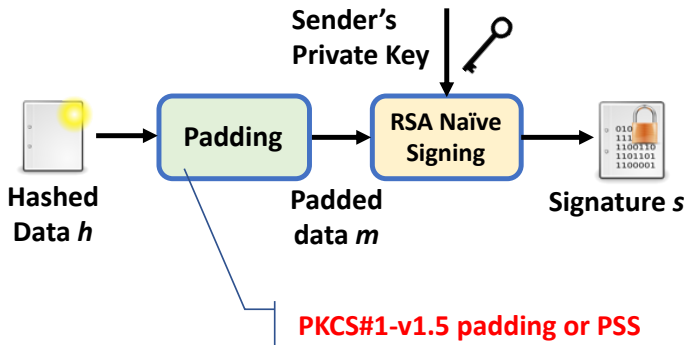
程度。

署名についても暗号化に求めるものと同程度、あるいはそれ以上の鍵サイズを担保して、公開鍵から秘密鍵を作られる (i.e., 自分を騙られる) ことを防ぐ。

RSASSA; RSA Signature Scheme with Appendix

RSASSA PKCS#1-v1.5 と、RSASSA PSS の違い

暗号化と同様に、 $h = \text{Hash}(D)$ に対して署名を作る際、鍵長に合わせたパディングが必要。そのパディングの方法が違う。



RSASSA PKCS#1-v1.5 (RFC8017²⁷)

- RSAES PKCS#1-v1.5 の署名版。PKCS#1 の v1.5 で標準化。SSL/TLS を含め、現在でも広く使われている。
- hash $h = Hash(D)$ を DER エンコードしたデータ T に、公開鍵長まで以下のようなパディングを付与。

$$m = 0x00||0x01||RandomSequence||0x00||T$$

- RSASSA PKCS#1-v1.5 は、hash ではなく生の短いデータに対して署名を生成すると、署名からその元データを戻せちゃうという脆弱性²⁸が知られている。本人確認の意味なし。
- PKCS#1 v2.2 (RFC8017) では、「仕様通りに hash 使っていればとりあえず問題ないが、今後何があるか分からないので **RSASSA-PSS を使うほうが無難**」と記載。

²⁷<https://tools.ietf.org/html/rfc8017>

²⁸Y. Desmedt et al., “A Chosen Text Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes,” in *Proc. CRYPTO 1985*, pp.516–522, 1985.、他。J. Coron (CRYPTO 1999) 等。

RSASSA-PSS (Probabilistic Signature Scheme, RFC8017²⁹)

- RSAES OAEP の署名版。RFC3447/PKCS#1 v2.1 (2003 年) で標準化策定。
- 署名対象データ D に対して、乱数成分 (RandomSalt) を加えつつ Hash 関数を 2 重掛けし、Padded data m を作っている。

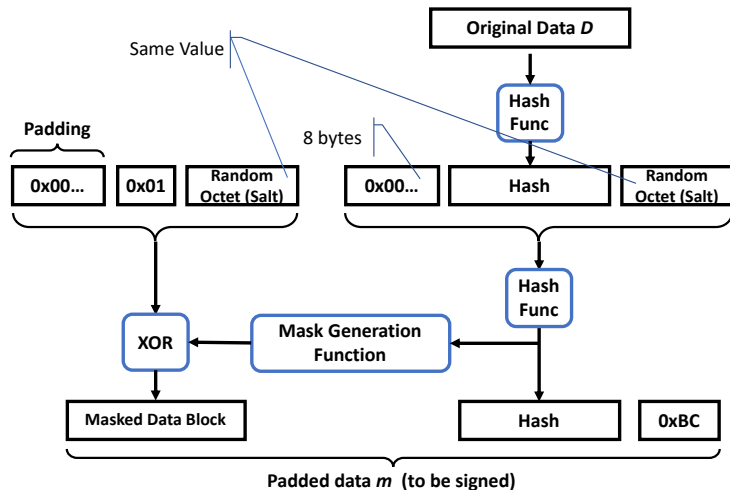
$$h = \text{Hash}(\text{Hash}(D), \text{RandomSalt}),$$
$$m = \text{MaskedDataBlock} || h || 0x\text{BC},$$

MaskedDataBlock はヘッダ。

- RSASSA PKCS#1-v1.5 からの乗り換えが進みつつあるものの **OAEP 同様に未対応の暗号ライブラリも多い**。ただし、RFC 標準的には、可能なら PSS を利用することを推奨。

²⁹<https://tools.ietf.org/html/rfc8017>

参考: PSS のイメージ図



元データ D に加え、padded data m があれば Salt などを逆計算可能。

JavaScript で RSASSA-PSS を実行してみる

では、RFC 推奨となっている PSS を試す。(コピー頑張って)

```
% yarn execute gen-rsa-key // RSA 鍵ペアの生成。-b でビット数指定。
<Generated RSA Public Key>
30820122300d06092a864886f70d01010105000... // 公開鍵
<Generated RSA Private Key>
308204be020100300d06092a864886f70d01010... // 秘密鍵
=====

% yarn execute sign-rsa-pss 'hello rsa-pss world!'\ // RSASSA-PSS 署名生成
-s '308204be020100300d06092a864886f70d0...' // 秘密鍵
<Generated RSASSA-PSS Signature>
6e7c4632f62e354f0ac40f65c92cd3e5bec5f6f... // 生成された署名
=====

% yarn execute verify-rsa-pss 'hello rsa-pss world!'\ // RSASSA-PSS 署名検証
-p '30820122300d06092a864886f70d0101010...' // 公開鍵
-t '6e7c4632f62e354f0ac40f65c92cd3e5bec...' // 受け取った署名
<Verification Result of RSASSA-PSS Signature>
true // 署名検証結果
=====
```

使う hash 関数は-h オプションで切り替えられる。

コードの中身はこんな感じ。
鍵生成は前回と一緒なので省略。

署名生成 (src/test-apis.js)

```
const jscu = getJscu();

// uint8array へ変換
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const privateKeyDer = jseu.encoder.hexStringToArrayBuffer(privateKeyHex);

const privateKey = new jscu.Key('der', privateKeyDer); // DER を読込

// hash = 'SHA-256', saltLength = 32
return jscu.pkc.sign(
  binaryData,
  privateKey,
  hash, // データを縮めるための hash 関数名
  {name: 'RSA-PSS', saltLength}
);
```

RSASSA でも、Hash 生成は署名生成 API 内部に含まれていることに注意。別途 Hash 生成しなくてよい。

署名検証 (src/test-apis.js)

```
const jscu = getJscu();

// uint8array へ変換
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const publicKeyDer = jseu.encoder.hexStringToArrayBuffer(publicKeyHex);
const signature = jseu.encoder.hexStringToArrayBuffer(signatureHex);

const publicKey = new jscu.Key('der', publicKeyDer); // DER 鍵を読み込

// hash = 'SHA-256', saltLength = 32
jscu.pkc.verify(
  binaryData,
  signature,
  publicKey,
  hash, // データを縮めるための hash 関数名
  {name: 'RSA-PSS', saltLength}
);
```

RSASSA-PSS は、Node.js Crypto/WebCrypto 共にネイティブサポートされている。(WebCrypto はブラウザ次第)
しかし他の環境だと、**OAEP 同様、PSS に未対応なライブラリも多く注意が必要**。(最近まで OpenSSL も未対応だった)

ECDSA; Elliptic Curve Digital Signature Algorithm

ECDSA (NIST FIPS 186-4³⁰)

- ECDH の署名版³¹。NIST FIPS 186-3 (2009 年) で標準化。
Bitcoin blockchain にも使われていて、また注目度が上がった。
- RSASSA とは異なり、事前の padding に気を使う必要がない。
アルゴリズム内部で hash h を生成・利用している。すなわち、

$$\text{Signature} = \text{SignECDSA}(D, SK),$$

$$\{\text{True}, \text{False}\} \ni \text{VerifyECDSA}(\text{Signature}, D, PK),$$

という直接の API がアルゴリズムレベルで提供される。³²

- OpenSSL をはじめほとんどの現代的な暗号ライブラリが実装をサポートしている。

³⁰<https://csrc.nist.gov/publications/detail/fips/186/4/final>

³¹正しい表現ではないが、イメージでそう捉えて欲しい。

³²RSASSA も、普通のライブラリでは直接 Sign/Verify 可能な API が提供されるが、その実態は RSA のナイーブな署名生成・検証アルゴリズムのラッパー。

RSASSA と ECDSA の比較。(一般論。実装にもよる。)

	Pros	Cons
RSASSA	<ul style="list-style-type: none">・署名検証が高速。 ECDSA と比較してもかなり速い。	<ul style="list-style-type: none">・署名サイズが大きい。 公開鍵長に等しい (e.g., 3072bits)。・署名生成がかなり低速。
ECDSA	<ul style="list-style-type: none">・署名サイズが小さい。 公開鍵長の倍 (e.g., 512bits (256bits key))。	<ul style="list-style-type: none">・署名生成・検証共に同じくらい低速。

どっちを使えばいいかは利用環境に応じて選択したほうがいい。常に ECDSA を使えばいいとかそういうわけではない。

JavaScript で ECDSA を実行してみる

```
% yarn execute gen-ecc-key // ECC 鍵ペアの生成。-c で楕円曲線パラメタ設定。
<Generated ECC Public Key>
3059301306072a8648ce3d020106082a8648ce3d03010703... // 公開鍵

<Generated ECC Private Key>
308193020100301306072a8648ce3d020106082a8648ce3d... // 秘密鍵
=====

% yarn execute sign-ecdsa 'hello ecDSA world!'\ // ECDSA 署名生成
-s '308193020100301306072a8648ce3d020106082a8648c...' // 秘密鍵

<Generated ECDSA Signature>
58e5d15be4e71f7e6fbd4662cdb31eca463ed855114ef8357bed9.... // 生成された署名
=====

% yarn execute verify-ecdsa 'hello ecDSA world!'\
-p '3059301306072a8648ce3d020106082a8648ce3d03010...' // 公開鍵
-t '58e5d15be4e71f7e6fbd4662cdb31eca463ed855114ef...' // 受け取った署名
<Verification Result of ECDSA Signature>
true // 署名検証結果
=====
```

使う hash 関数は-h オプションで切り替えられる。

コードの中身はこんな感じ。
鍵生成は前回と一緒なので省略。

署名生成 (src/test-apis.js)

```
const jscu = getJscu();

// uint8array へ変換
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const privateKeyDer = jseu.encoder.hexStringToArrayBuffer(privateKeyHex);

const privateKey = new jscu.Key('der', privateKeyDer); // DER を読み込

// hash = 'SHA-256'
return jscu.pkc.sign(
  binaryData,
  privateKey,
  hash, // データを縮めるための hash 関数名
);
```

署名検証 (src/test-apis.js)

```
const jscu = getJscu();

// uint8array へ変換
const binaryData = jseu.encoder.stringToArrayBuffer(data);
const publicKeyDer = jseu.encoder.hexStringToArrayBuffer(publicKeyHex);
const signature = jseu.encoder.hexStringToArrayBuffer(signatureHex);

const publicKey = new jscu.Key('der', publicKeyDer); // DER 鍵を読み込

// hash = 'SHA-256'
jscu.pkc.verify(
  binaryData,
  signature,
  publicKey,
  hash // データを縮めるための hash 関数名
);
```

RSASSA-PSS と異なり、Salt が不要になり、API がよりシンプル！

ECDSA は、Node.js Crypto/WebCrypto 共に（大体）ネイティブサポートされている。

ただし、パラメータによっては実装されていない可能性があるの
で注意。ブラウザ差異だけではなく、仕様がない可能性も。³³

³³ JavaScript では曲線'P-256K' がネイティブではサポートされていないので、Bitcoin blockchain の ECDSA は動作しない。jscu では pure js で補っている。

MAC/署名の運用について

署名検証のブートストラップの問題

Q: 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？

⇒ 現状ではこの問題に対して**仮定なしで OK な万能の解は未知**。
どこかに**仮定、信頼するという起点**を置いた暫定解がある感じ。

暫定解 1: Trust Anchor

End ごとに、**検証用の公開鍵をアプリケーション・端末に固定で事前に埋め込んでおく**。そこだけは起点として無条件に信頼して使う。

※ SSH とか GitHub で行う「ホスト・サービスへの公開鍵の登録」、というのはこの Trust Anchor の登録になる。

暫定解 2: PKI に頼る

さらに Verisign とかに検証用の公開鍵に署名してもらって、公開鍵証明書を作る。公開鍵証明書自身はオンラインで取得するが、その Verisign の署名の検証は Trust Anchor…(Verisign を信頼の起点にする)。

解 2 は、Verisign の鍵 1 つを Trust Anchor にすればいいので鍵を埋め込むような手間がかからないが、お金がかかる³⁴。

³⁴賛否あるだろうが、Let's encrypt (<https://letsencrypt.org/>) を使えば無償でいける。ただし Let's encrypt は厳密な本人確認とせずに署名つけてくれるので…

署名・MAC の使い分け

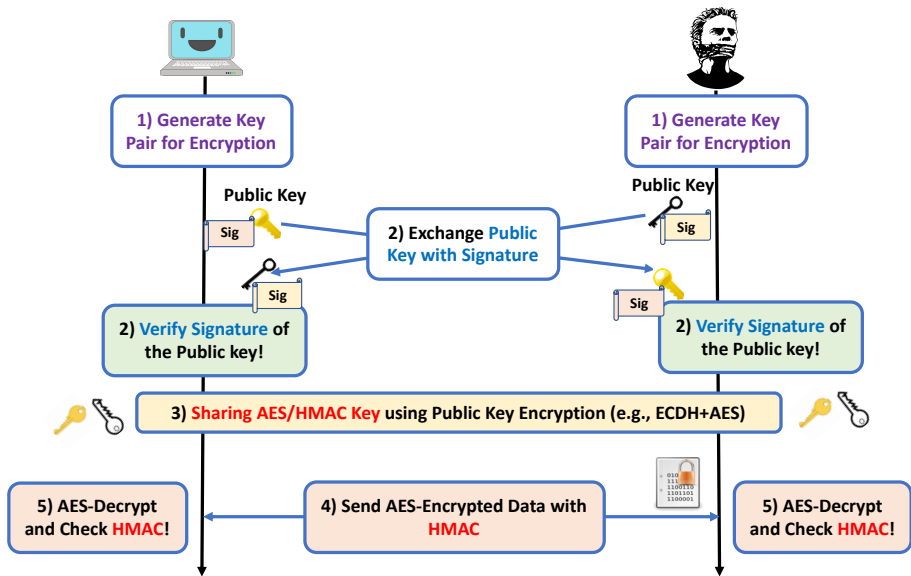
処理の重さで使い分けるのが鉄則。

- 暗号化されたデータのやり取りの手続きで、**署名はエンド間でのイニシエーション**に使う
- **MAC はエンド間でなんども繰り返すような本人確認・データ整合性確認**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 HMAC/AES 暗号化の鍵を共有。
- 3 以降の大規模データのやり取りは AES 復号と同時に HMAC で本人確認を実施。

図にすると:



まとめ

まとめ

お疲れ様でした。

- データの真正性・本人確認のためのテクニックについて学んだ。
 - Hash: データの「指紋」。SHA-2 以降のものを使おう。
 - MAC:
 - 共通鍵ベースの本人確認手法。動作が軽い。
 - JS だにとりあえず HMAC を使う。
 - 署名:
 - 公開鍵ベースの本人確認手法。MAC と比べると動作が重い。
 - RSASSA: 使うなら **RSASSA-PSS がおすすめ**。
 - ECDSA: 大体どの環境でも使える。
- 上記について、JavaScript のコードを実行/中身を覗いてみた。
- 署名はエンド間のやり取りイニシエーション、MAC はその後のデータのやり取りに使うと良い。

宣伝: iTransfy by Zettant

簡単・安全にファイル転送ができる

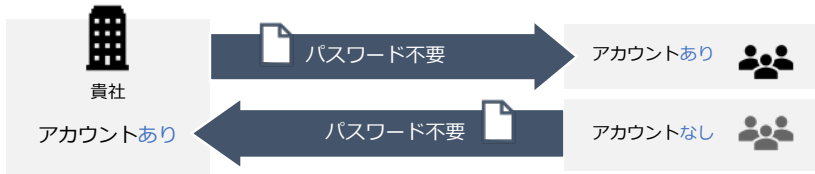


iTransfy for biz

<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

会社 URL: <https://www.zettant.com>