WebCryptoから始める  
暗号化生活

株式会社ムラサメ研究所  
みやたけ　ゆき

# まえがき

# はじめに

目次

目次

[WebCryptoから始める 暗号化生活 1](#_Toc24961496)

[まえがき 2](#_Toc24961497)

[はじめに 3](#_Toc24961498)

[1 基礎知識 7](#_Toc24961499)

[1.1. 暗号化概要 7](#_Toc24961500)

[1.1.1. 暗号化の目的 7](#_Toc24961501)

[1.1.2. 暗号化の方式 7](#_Toc24961502)

[1.1.3. データ形式 8](#_Toc24961503)

[1.1.4. PKI(Public Key Infrastructure)基盤 8](#_Toc24961504)

[1.1.5. 認証局（CA） 9](#_Toc24961505)

[1.1.6. PKCS 9](#_Toc24961506)

[1.2. WebCryptoとは 9](#_Toc24961507)

[1.3. ブラウザ対応状況 10](#_Toc24961508)

[2. ツールの使い方 10](#_Toc24961509)

[2.1. OpenSSH、Puttyのインストール 10](#_Toc24961510)

[2.2. OpenSSH、Puttyの使い方 10](#_Toc24961511)

[2.3. Hashツール 10](#_Toc24961512)

[2.4. PKI.js 10](#_Toc24961513)

[3. WebCryptoを試す 10](#_Toc24961514)

[3.1. Hashを作る 10](#_Toc24961515)

[3.1.1. WebCryptoのインタフェースを取得 10](#_Toc24961516)

[3.1.2. ハッシュ化 11](#_Toc24961517)

[ 結果を確認 13](#_Toc24961518)

[3.2. RSAキーを作成 13](#_Toc24961519)

[3.2.1. PrivateKeyを作成しPEMに変換する 13](#_Toc24961520)

[3.2.2. PrivateKey＆PublicKeyを作成しPEMに変換する 17](#_Toc24961521)

[3.3. RSAキーの内容確認 19](#_Toc24961522)

[3.4. Verify 19](#_Toc24961523)

[3.5. ・・・ 19](#_Toc24961524)

[3.6. 作成したキーを外部サービスで使う 19](#_Toc24961525)

1. 基礎知識
   1. 暗号化概要
      1. 暗号化の目的

我々は知らず知らずのうちに暗号化を利用しています。

無線LAN、SSH、HTTPS、パスワードの保存、クラウドストレージ

数え上げるときりがありません。

暗号化や認証により下記の問題を解決します

* 盗聴  
  アリスがボブに送った秘密のメッセージをイブに読まれること  
  データの暗号化で防ぐ事ができます
* 改ざん  
  アリスがボブに好き（はぁと）って送ったメッセージを  
  ボブに思いをよせていたマロリーに 嫌いと改ざんされること  
  デジタル署名により防ぐ事ができます
* なりすまし  
  イブがボブに、アリスの名を語って嘘のメッセージを送ること  
  信頼性の高い認証局を使う事で防ぐ事ができます
  + 1. 暗号化の方式
* ハッシュ  
  可変の文字列から固定長の文字列を生成する  
  メッセージダイジェストとも呼ばれる  
  ハッシュ化された文字列から元には戻せない  
  処理速度は速い  
  データの改ざんを調べる事ができる  
  MD5、SHA-1、SHA-256等が有名
* 共通鍵方式  
  暗号化、複合化に同じ鍵を使う  
  相手毎に鍵を作るので鍵の管理が大変  
  速度は比較的速い  
  鍵を相手に安全に渡す事が難しいため、下記の公開鍵方式で鍵を暗号化して送る事がある  
  暗号方式はDES、AESなどがある
* 公開鍵方式  
  暗号化に公開鍵、複合化に秘密鍵と対になる2つの鍵を使う  
  秘密鍵がなければ複合化できないので、公開鍵は漏れても問題がない  
  計算に時間がかかるため、共通鍵方式と併用されることが多い  
  RSA、DSA、楕円曲線暗号が使われる  
  + 1. データ形式

データを表す形式は ASN.1にて定義されています

* BER (Basic Encoding Rules)  
  日本語に訳すと、基本的な符号化規則

バイナリ形式の基本的なデータエンコーディングルール

1つのデータをコンテナにして連結させた形式です  
コンテナは、コンテナ種類、サイズ、データで構成されます  
  
ただし、符号化において複数の表現が可能なため、署名において問題が発生します  
そのため、暗号化においては厳密なDERやCERが使われます

* DER (Distinguished Encoding Rules)  
  日本語に訳すと、優れた符号化規則？  
  BERのサブセットでより厳密に正規化された規格です  
  固定長のデータを扱う時に使われます
* CER (Canonical Encoding Rules)  
  日本語に訳すと、正規化された符号化規則？  
  BERのサブセットでより厳密に正規化された規格です  
  可変長データの時に使われます
* PEM (Privacy Enhanced Mail)  
  日本語だと、匿名性を負荷した電子メール？  
  DERやCERはバイナリデータで、そのままでは人間が扱いにくく  
  メールやHTTPにおいて扱いが難しくなるので  
  Base64でバイナリデータをエンコーディングし、ヘッダとフッタを付与したものをPEMと呼びます  
  + 1. PKI(Public Key Infrastructure)基盤

公開鍵方式を利用したセキュリティインフラの事をさします

* X.509  
  証明書や失効リストに関する取り決め
* TLS  
  トランスポート層で暗号化しセキュリティーを付与します  
  SSHやHTTPSなどに使われています
  + 1. 認証局（CA）

公開鍵が改ざんされた場合、悪意のある人にデータが盗まれる可能性がある  
第三者により公開鍵を証明する仕組み

CAの役割は２つ。公開鍵を登録する事と、失効させること。

（図が必要）

アリスがボブに暗号化メッセージを送る時を考えてみます

ボブはアリスに自分の公開鍵を渡します  
アリスはボブの公開鍵を使いメッセージを暗号化しボブに送ります  
秘密鍵はボブしか持っていないため、ボブ以外にはメッセージが解読できません  
  
ところが、悪意を持ったマロリーがボブの公開鍵ではなく自分の公開鍵をアリスに送りました

それを知らないアリスは、マロリーの公開鍵で暗号化してしまいました  
秘密鍵を持っているマロリーはメッセージを手に入れました  
  
上記の問題の解決に、認証局（キャロル）が必要です

ボブはキャロルに公開鍵を登録します  
アリスはキャロルからキャロルが署名したボブの公開鍵をもらいます  
アリスは署名が正しい事を確認し、ボブの公開鍵で暗号化しボブに送ります

たとえマロリーが自分の公開鍵をアリスに渡したとしても、署名でなりすましがバレます

* + 1. PKCS

１～１５？

* 1. WebCryptoとは

Webブラウザで暗号化を行うAPI  
<https://webrtc.org/>

* 1. ブラウザ対応状況

図

1. ツールの使い方
   1. OpenSSH、Puttyのインストール
   2. OpenSSH、Puttyの使い方
   3. Hashツール
   4. PKI.js
2. WebCryptoを試す
   1. Hashを作る

手始めに、SHA-1ハッシュを作成します

WebCryptoのインタフェースを取得し、digest関数を利用します  
digestにはArrayBufferで渡す必要がるのでStringからArrayBufferへ変換します  
digestの戻りはArrayBufferなので、Hexに変換します

WebCryptoはPromiseを使った非同期ライブラリとして提供されます

今回はPromiseを使って処理をします  
Promiseについて不明な人は軽く勉強しておいてください

コードは下記にあります

<https://github.com/YukiMiyatake/WebApiTest/tree/master/WebCrypto/src/CreatePrivatePEM>

* + 1. WebCryptoのインタフェースを取得

|  |
| --- |
| // WebCryptoのライブラリ  let crypto = window.crypto || window.msCrypto;  let subtle = crypto.subtle || crypto.webkitSubtle;  // アルゴリズム  const hashAlg = "SHA-1"; |

WebCryptoのライブラリをwindowから取得します

IEの場合は window.msCrypto.subtleで取得します

Safariの場合は window.crypto.webkitSubtleで取得します  
firefoxの場合は window.crypto.subtleです  
と、ベンダー毎に異なるため、上記のように面倒な取り方となります  
  
ハッシュアルゴリズムは SHA-1にします（SHA-1は強度が弱いので暗号化には向きません）

* + 1. ハッシュ化

subtle.digest 関数を使います

<https://developer.mozilla.org/ja/docs/Web/API/SubtleCrypto/digest>  
  
digest関数へは、ハッシュアルゴリズムとハッシュするデータ（ArrayBuffer）を渡します

対応しているハッシュアルゴリズムは  
**SHA-1、SHA-256、SHA-384、SHA-512** となります

* HTMLコード

|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">  <head>  <script lang="text/javascript" src="mycrypto.js" ></script>  <title>Create Hash</title>  </head>  <body>  <div class="container">  <label >Create Hash</label><br>  <textarea id="msg-text" cols="70" rows="1"></textarea><br>  <textarea id="hash-text" cols="40" rows="1"></textarea><br>  <button type="button" onclick="createHash();">Create</button>  </div>  </body>  </html></body>  </html> |

ハッシュ元メッセージ msg-text、ハッシュダイジェスト hash-text、ハッシュボタン  
だけのシンプルなHTMLです

* StringをArrayBufferに変換

digestに渡すためにStringをArrayBufferに変換します

|  |
| --- |
| // StringをArrayBufferに  function str2ab(str) {  const buf = new ArrayBuffer(str.length);  const bufView = new Uint8Array(buf);  for (let i = 0, strLen = str.length; i < strLen; i++) {  bufView[i] = str.charCodeAt(i);  }  return buf;  } |

* ArrayBufferをHexに

ダイジェストではHexで表示する事が一般的なのでHexに変換します

|  |
| --- |
| // ArrayBufferをHexに  function ab2hex(buf) {  var s = "";  var dataView = new DataView(buf);  for (var i = 0; i < buf.byteLength; i++) {  var hex = dataView.getUint8(i).toString(16);  hex = "00".substr(0, 2 - hex.length) + hex;  s += hex;  }  return s;  } |

* ハッシュ取得

|  |
| --- |
| // Hash化する  function createHash() {  return( Promise.resolve()  .then(() => {  const msg = str2ab(document.getElementById("msg-text").value);  return(subtle.digest(hashAlg, msg));  },  error => Promise.reject((`Error during digest: ${error}`))  )  .then(hash => {  const hex = ab2hex(hash);  document.getElementById("hash-text").value = hex;  })  );  } |

メッセージをArrayBufferに変換し、digest関数でハッシュ化し  
その値をHexに変換し出力します

* 結果を確認

|  |
| --- |
|  |

|  |
| --- |
| $ echo -n hoge | sha1sum  31f30ddbcb1bf8446576f0e64aa4c88a9f055e3c |

完全に一致しました！

* 1. RSAキーを作成
     1. PrivateKeyを作成しPEMに変換する

RSAキーを作って表示してみましょう

今回はPrivateKeyだけ表示させます

GenerateKeyでKeyPairを作成し、PKCS#8にエクスポートし  
ArrayBufferをStringに変換し、Base64にし、64文字で改行をいれヘッダフッタ挿入  
これでPEMが出来上がります

コードは下記にあります

[https://github.com/YukiMiyatake/WebApiTest/tree/master/WebCrypto/src/CreatePrivatePEM](https://github.com/YukiMiyatake/WebApiTest/tree/master/WebCrypto/src/sample3.1)

* KeyPairの作成

|  |
| --- |
| function createCryptoKeys() {  return( Promise.resolve()  .then(() => {  // アルゴリズムを選びKeyPairを作る  let algorithm = {  name: signAlg,  modulusLength: 2048,  publicExponent: new Uint8Array([0x01, 0x00, 0x01]),  hash: {  name: hashAlg  }  };  return subtle.generateKey(algorithm, true, ["sign", "verify"]);  },  error => Promise.reject((`Error during generateKey: ${error}`))  ) |

Htmlから呼ばれる関数を作ります

WebCryptoは非同期ライブラリになっていて、Promiseを使います

Algorithmにはアルゴリズム等を指定します  
RSA以外にもECDSAやHMACなども指定する事ができます

usagesには["sign", "verify"]を指定しておきます  
Algorithmに公開鍵方式を指定した場合はCryptoKeyPair、共通鍵方式の場合はCryptoKeyを返却するPromiseが返ってきます

* キーのエクスポート

|  |
| --- |
| .then(keyPair => {  // PrivateKeyをpkcs#8(秘密鍵)形式にExportする  return (subtle.exportKey(  "pkcs8",  keyPair.privateKey  ));  },  error => Promise.reject((`Error during exportkey: ${error}`))  ) |

generateKeyで作成したCryptoKeyPairよりprivateKeyを使います

それを秘密鍵の標準形式 PKCS#8にExportします

PKCS#8にExportされた値を返すPromiseが生成されます

* 文字列型へ変換しTextareaに表示する

|  |
| --- |
| .then(pkcs8 => {  // ArrayBufferを文字列にし  const pkcs8str = ab2str(pkcs8);  // Base64エンコードを行い64文字毎に改行を行い  const base64 = pac64(btoa(pkcs8str));  // ヘッダとフッタを追加  const pem = `-----BEGIN PRIVATE KEY-----\n${base64}\n-----END PRIVATE KEY-----`;  // Textareaに表示  document.getElementById("pem-text").value = pem;  return (pem);  })  );  } |

ExportしたデータはArrayBufferなので、ab2strでStringに変換する  
次に組み込み関数btoaでBase64にエンコードする  
そしてpac64で64バイト毎に改行を行い  
最後にヘッダ度フッタを追加すればPEM形式の出来上がりです

* ライブラリコード

|  |
| --- |
| // 64文字ごとに改行を入れる  function pac64(pemString)  {  const stringLength = pemString.length;  let resultString = "";  for(let i = 0, count = 0; i < stringLength; i++, count++) {  if( count > 63 ) {  resultString = `${resultString}\r\n`;  count = 0;  }  resultString += pemString[i];  }  return resultString;  }  // ArrayBufferをStringに  function ab2str(buf) {  return String.fromCharCode.apply(null, new Uint8Array(buf));  } |

64バイト毎に改行を入れる関数とArrayBufferをStringにする関数は上記になります

* 実行結果

|  |
| --- |
| テキスト が含まれている画像  自動的に生成された説明 |

PEMの確認をする

* + 1. PrivateKey＆PublicKeyを作成しPEMに変換する

先ほどのコードを元にPublicKeyのPEMも表示させるようにします  
PublicKeyはspkiに変換します

コードは下記にあります

[https://github.com/YukiMiyatake/WebApiTest/tree/master/WebCrypto/src /](https://github.com/YukiMiyatake/WebApiTest/tree/master/WebCrypto/src/sample3.1)CreateKeypairPEM

* PrivateKeyとPublicKeyの変換は並列実行が可能

|  |
| --- |
| // PublicKeyとPrivateKeyのExportは並列で出来る  .then(keyPair => {  return Promise.all([  // Private Key  Promise.resolve()  .then(() => {  // PrivateKeyをpkcs#8(秘密鍵)形式にExportする  return(subtle.exportKey(  "pkcs8",  keyPair.privateKey  ));  })    .then(pkcs8 => {  const pem = ab2pem(pkcs8, "PRIVATE KEY");  return (pem);  }) |

KeyPairを取得するまでは前回と同じです

PrivateKeyとPublicKeyは同時に行っても問題は無いのでPromise.allを使い並列実行してしまいましょう。  
並列実行部分以外は同じです

* PublicKeyの取得

spki形式にExportする以外は同じです

|  |
| --- |
| // Public Key  , Promise.resolve()  .then(() => {  // PublicKeyをspki形式にExportする  return(subtle.exportKey(  "spki",  keyPair.publicKey  ));  })  .then(pkcs8 => {  const pem = ab2pem(pkcs8, "PUBLIC KEY");  return (pem);  })  ]) |

keyPair.publicKeyでPublicKeyは取得できます  
spkiでExportするのがPrivateKeyとの違いです  
ab2pemはArrayBufferからヘッダ付きのPEMまで作成するライブラリ関数です

* TextAreaに表示する

Promise.allは戻り値を配列なので、それらをTextAreaに設定します

|  |
| --- |
| .then(values => {  document.getElementById("private-pem-text").value = values[0];  document.getElementById("public-pem-text").value = values[1];  });  }) |

* 実行結果

PrivateKeyとPublicKeyが表示されました

|  |
| --- |
| スクリーンショット が含まれている画像  自動的に生成された説明 |

* + 1. ハッシュアルゴリズム、署名アルゴリズムの組み合わせ
  1. RSAキーの内容確認
  2. Verify
  3. ・・・
  4. 作成したキーを外部サービスで使う