電子署名（PKI）ハンズオン

～電子署名・タイムスタンプ超入門！～

2015/03/14

JNSA勉強会/AITCオープンラボ

日本ネットワークセキュリティ協会
電子署名ワーキンググループ

先端IT活用推進コンソーシアム
オープンラボ

自己紹介:宮地 直人

所属:有限会社ラング・エッジ プログラマ/取締役
JNSA電子署名WG サブリーダー
AITCクラウド・テクノロジー活用部会 メンバ

バックボーン:
2006年より自社長期署名製品の開発担当。
現在はJNSAで電子署名等の標準化に参加。
フリータイムスタンプ「FreeTSA」開発者。

コンタクト:
mail miyachi@langedge.jp
Facebook fb.me/nao.miyachi
Twitter @le_miyachi

電子署名ハンズオン の目的

クラウドの発展により紙から電子文書への移行が加速しています。電子文書/データの信頼性を守る電子署名やタイムスタンプの技術が注目されてはいますが、「利用が難しい」「情報が少なくて使い方が分からない」と言う話をよく聞きます。

電子署名ハンズオンでは主に技術者向けに、電子署名やＰＫＩの基本をセミナー形式で解説すると共に、ハンズオンにより実際に電子署名やタイムスタンプを体験して頂きます。技術的な内容だけでは無く日本における法的な基盤についても解説して非技術の方でも参考になる内容になっています。

電子署名ハンズオン メニュー

OpenSSLを使って
実際に操作する

- 第1部：電子署名とタイムスタンプ
  1. 電子署名とその基本技術を理解する
  2. ハンズオン1：電子署名を生成してみよう
  3. タイムスタンプ技術を理解する
  4. ハンズオン2：タイムスタンプを取得してみよう
- 第2部：署名検証と長期署名
  5. 電子署名の検証技術とPKIを理解する
  6. ハンズオン3：検証してみよう
  7. 長期署名を理解する
  8. ハンズオン4：長期署名を作ってみよう

Adobe Readerで長期署名
PAdES-Aファイルを作る

事前準備の確認

1. Wi-Fi（インターネット）への接続

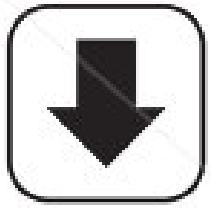
接続環境が無い人は、無線LANが利用可能です。

※ SSIDとパスワードは別途お知らせします。

2. 利用するソフトウェアのインストール

第１部：ハンズオンソフトウェアのダウンロード

※ 詳細は次ページにて。



第２部：Adobe Reader-XI（最新版）

※ 以下ダウンロードサイトからインストール。

https://get.adobe.com/jp/reader/

注意：「オプションのプログラム」は必ず「オフ」！

ハンズオンソフトウェア

本ドキュメントもこのアドレスからダウンロード可能です

- 各環境用ソフトをダウンロードして展開します。

  ダウンロード:　http://eswg.jnsa.org/sandbox/handson/

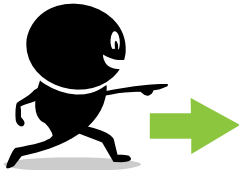
  Windows環境用:　ESig-PKI-handson-win-v100.zip
  MacOS-X環境用:　ESig-PKI-handson-mac-v100.tar.gz

- ハンズオン実施時には各ディレクトリ下に移動してください。

ディレクトリ名	内容
OpenSSL	OpenSSL実行ファイルとテスト用ファイル
Java	Javaソースとテスト用ファイル （Javaソースの文字コードはShift-JISです） ※ Java環境（JDK）は1.7（Java7）以上が必要。
AdobeReader	Adobe Reader/Acrobatのテスト用ファイル ※ Windows環境とMacOS環境で共通です。

電子署名(PKI)ハンズオン

第1部:電子署名とタイムスタンプ



1. **電子署名とその基本技術を理解する**
2. 電子署名を生成する(ハンズオン1)
3. タイムスタンプ技術を理解する
4. タイムスタンプを取得する(ハンズオン2)。

電子署名とデジタル署名

質問:「電子署名」と「デジタル署名」って同じ物？

Wikipediaより:　http://ja.wikipedia.org/wiki/電子署名

電子署名とは、電子文書に付与する、電子的な徴証であり、紙文書における印やサインに相当する役割をはたすものである。（中略）

電子署名を実現する仕組みとしては、公開鍵暗号方式に基づくデジタル署名が有力である。

「電子署名」は役割であり「デジタル署名」は実現する為の技術のこと。「デジタル署名」を使わない「電子署名」も存在すると言うこと。

電子署名法

1. 日欧米で電子署名法と要求される仕組みが異なる

- 日欧は公開鍵暗号（主にRSA方式）を利用したデジタル署名

※ 日本では電子署名法、e-文書法、電子帳簿保存法等で規定。
本ハンズオンでは電子署名≒デジタル署名として解説します。

- 米国は電子証拠（エビデンス）ベースの電子署名

※ 運用ログ（同意の証拠）とシステムログ等を管理して残す。

 日本	 欧州	 米国
デジタル署名		電子エビデンス

2. デジタル署名：公開鍵暗号は秘密鍵漏洩がない前提

- 秘密鍵が署名者から漏洩しない前提で署名者を確定する
- 署名値と署名属性（署名方式・署名者情報）と署名対象で構成

電子署名法（電子署名及び認証業務に関する法律）

- 電子署名法は2001年4月に施行
  1. 電磁的記録の真正な成立の推定（第3条）
  2. 特定認証業務に関する認定制度（第4から16条）

- 「電子情報の信頼性」を確保する法的な裏付け

  本人による電子署名が付与された電子文書は、
  訴訟時の証拠として、紙と同等な証拠能力がある。

- 特定認証業務の認定電子認証局

  → GPKI（政府PKI）と相互認証され電子申請等で利用可能

※電子署名法主務三省の管轄

（総務省、法務省、経済産業省）

余談：電子エビデンス（Electronic Evidence）

米連邦証拠規則における電子エビデンスの証拠能力：

「6. Process or System」より「真正性を示すための11ステップのプロセス」

注：訳例ですので正確には原文を参照！

1. その業務にコンピュータを使用していること。
2. そのコンピュータは信頼できること。
3. その業務において、データの入力手順が既に開発されていること。
4. その手順は正確さを保証し、エラーを確認するための安全装置が組み込まれていること。
5. その業務では、コンピュータ利用状態を適切に維持していること。
6. 証人は、確かにコンピュータから読みだされたデータを持っていること。
7. 証人は、読み出したデータを得るために適切な手順を用いたこと。
8. 証人が読み出しデータを得た時点で、コンピュータが正常な状態で使用可能であったこと。
9. 証人は、読み出しデータの公開を認めること。
10. 証人は、彼又は彼女がその読み出しデータをどのように承認しているか説明すること。
11. もし読み出しデータが奇妙な符号や条件を含むのなら、承認はその符号や条件の意味を説明すること。

参考：電子記録応用基盤研究会（eRAP）平成27年2月成果報告書
http://www.jipdec.or.jp/dupc/forum/faudi/aboutus/26seika01.pdf

余談：電子認証

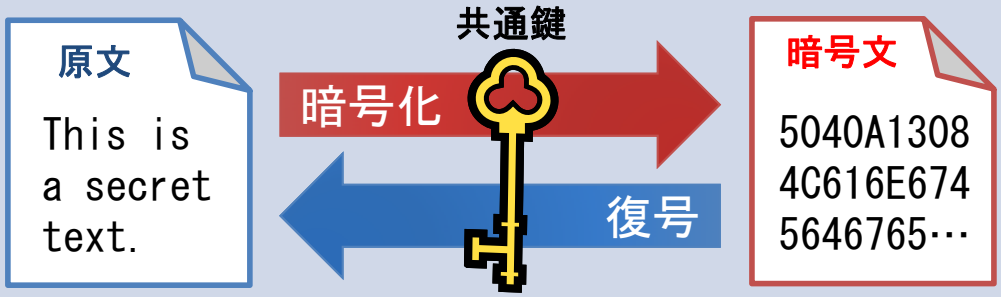
世の中で「電子署名」と「電子認証」の区別があいまい…

- 認証とは、「対象（人や物）の正当性を認め保証する」ことであって署名とは意味が異なる。
- 電子世界でも同様であり、「電子署名」と「電子認証」は本来異なる意味を持つ。
- 「電子署名」では、「認証」を本人確認等で利用する。
  例：認証局は「電子認証サービス」を提供している。
- 「電子認証」ではID/パスワード方式以外に、デジタル署名を使った電子証明書による認証方式もある。
- 米国のクラウド署名は電子認証を使った、非デジタル署名である。

暗号基本：共通鍵暗号と公開鍵暗号

共通鍵暗号

暗号化と復号で**同じ鍵**を利用

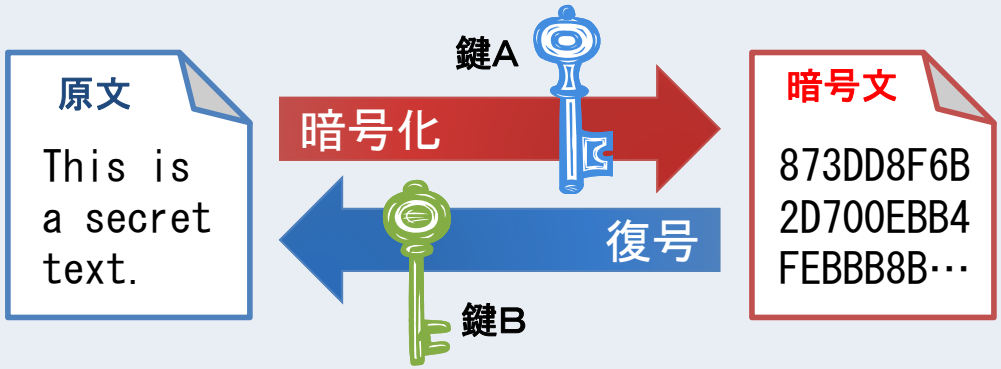


第三者に共通鍵が漏えいすると秘密が保てない。

どうやって共通鍵を共有するのか？

公開鍵暗号

暗号化と復号で**異なる鍵**を利用



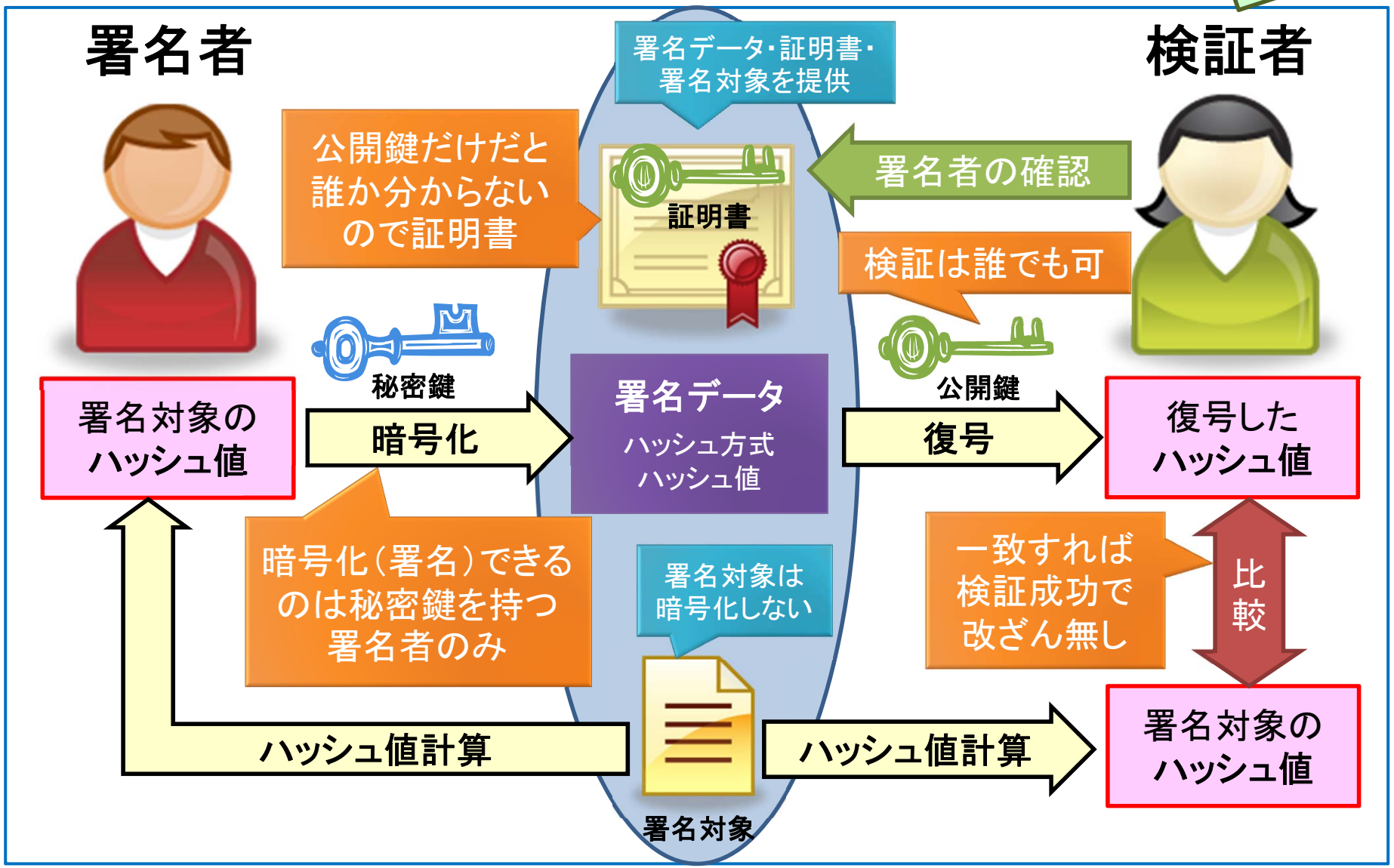
一方の鍵を公開し、対となるもう一方を秘密にする。

秘密鍵で暗号化することで電子署名が実現できる。

	共通鍵暗号	公開鍵暗号
主な暗号方式	AES, Camellia	RSA, ECC（楕円曲線）
一般に計算は	軽くて早い	重くて遅い
主な用途	大量データの暗号化	共通鍵自体の暗号化、電子署名

公開鍵暗号によるデジタル署名

署名（作成）者と検証者が異なる点がとても重要。



電子署名でよく使われるデータ形式

1. ASN.1/BER/DER形式（バイナリ）のCMS署名

- RFC 5652　Cryptographic Message Syntax (CMS)
- 電子署名の世界において最も基本的なデータ形式

2. XML形式（テキスト）のXML署名

- W3C勧告 XML Signature Syntax and Processing (XmlDsig)
- ZIP圧縮を使ったデータ形式でもXML署名の利用が多い
- ただし証明書/CRL等の要素はASN.1/BER(DER)形式を利用

3. PDF形式（バイナリ/テキスト）のPDF署名

- ISO 32000　署名データはASN.1/DERのPKCS#7等を利用

4. JSON形式（テキスト）のJWS(Json Web Signature)

- IETF Internet Draft JSON Web Signature (JWS)

参考：JSON（JavaScript Object Notation）とは？

- XMLに替わりAjax/WebAPI等で使われる軽量データ記述言語
- JavaScriptのオブジェクト表記だったがRFC 7159で標準化

✓ 名前/値のペアの集まりで、オブジェクトや連想配列として実現
✓ オブジェクトは ”{“(左の中括弧) で始まり ”}”(右の中括弧) で終わる
✓ 名前の後に ”:”(コロン)が付き、名前/値ペアは ”,”(コンマ)で区切る
✓ 配列は “[“(左の角括弧) で始まり “]”(右の角括弧) で終わる
✓ 値にオブジェクトや配列を使うことで階層化が可能

```
{
    “event”: “pki-handson”,
    “member”:  [
        { “name”: “Alice”, “age”: 21 },
        { “name”: “Bob”, “age”: 32 }
    ]
}
```

オブジェクト開始
　イベントはpki-handson
　メンバー配列
　　名前はアリス、21歳
　　名前はボブ、32歳
　配列終了
オブジェクト終了

JSONのサンプル

参考：JSON署名・JWS（Json Web Signature）

- URL引数のトークンにも使えるJSONベースの新しい署名形式
- Header/Payload/Cryptoの3パートをBase64化して”.”で接続
  - JWS Header Input：署名アルゴリズム等の属性をJSON記述
  - JWS Payload Input：署名対象のJSON記述
  - JWS Crypto Output：署名値そのもの

ある意味、電子署名の基本的な構成です。

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAi
OjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnV
lfQ.dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

JWS（正確にはJWT）のサンプル

ヘッダ部 Base64展開	署名対象部 Base64展開
{ “typ”: “JWT”, “alg”: “HS256” }	{ “iss”: ”oe”, “exp”: 1300819380, “http://example.com/is_root”: true }

“alg”:”HS256” = HMAC-SHA256
“alg”:”RS256” = RSA-SHA256

ASN.1（Abstract Syntax Notation One）

※ ここからPKIの基本となるフォーマット仕様の説明です。

- ASN.1（**抽象構文**記法1）はプロトコル記述用の言語
- ISOとITU-Tによって規定され、現在はX.680で規定
- PKIデータのほとんどはANS.1で記述定義されている

Car ::= SEQUENCE {
  type PrintableString,
  price INTEGER OPTIONAL,
  color ColorType
}
ColorType ::= INTEGER {
  red (0),
  white (1),
  blue (2)
}
	車(Car)定義、オブジェクト集合
	  種類(type)は文字列
	  価格(price)は整数値でオプション
	  色(color)はColorType
	
	色(ColorType)定義、整数値
	  赤(red)は0
	  白(white)は1
	  青(blue)は2

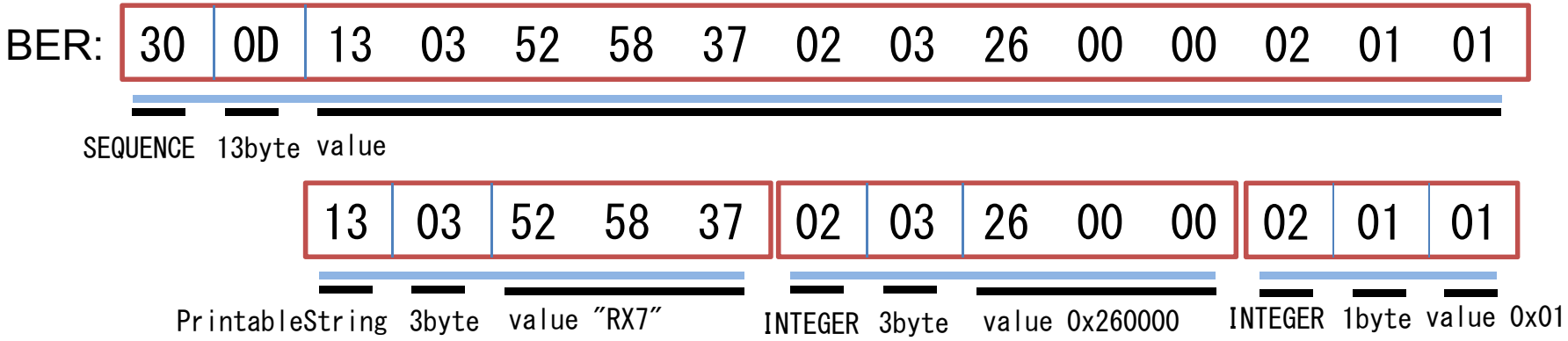
ASN.1記述のサンプル

BER（Basic Encoding Rules）

- BER（基本符号化規則）はASN.1のバイナリエンコード規則
- CCITT X.209, ISO8825-1 として定義
- １要素は、[タグ（次頁参照）] [値長] [値] で表現される

```
Car ::= SEQUENCE {
  type    PrintableString,
  price   INTEGER OPTIONAL,
  color   ColorType
}
```

オブジェクト集合
種類(type)　　= "RX7"(3byte)
価格(price)　 = 0x260000(約249万円)
色(ColorType) = white (1)



BERのサンプル

BERの主なタグ値（種類）

タグ種類	HEX	説明
BOOLEAN	0x01	論理型(true/false)、DERのtrueは1限定
INTEGER	0x02	整数型(符号付、可変長)
BIT STRING	0x03	ビット単位可変長のバイナリ値に使われる
OCTET STRING	0x04	オクテットストリーム
NULL	0x05	値なし
OBJECT IDENTIFIER	0x06	OID　データ型や暗号の識別番号　※後述
UTF8String	0x0C	日本語等利用時に推奨
SEQUENCE	0x10	構造化フラグ0x20等が必要なので通常0x30
SET	0x11	構造化フラグ0x20等が必要なので通常0x31
NumericString	0x12	数字文字列
PrintableString	0x13	表示可能文字列
IA5String	0x16	アスキー文字限定
UTCTime	0x17	YYMMDDhhmmssZ（2000年問題あり）
GeneralizedTime	0x18	YYYYMMDDhhmmss.dZ（推奨）

DER（Distinguished Encoding Rules）

- DER（識別符号化規則）はBERに制限加えたサブセット仕様
- DERは固定長で正規化する規則でX.509等で利用
  - エンコードされるサイズは規定に従う必要がある（ありうる最小サイズ）
  - BitString,OctetString等はプリミティブエンコードする
  - 指定された要素順にソーティングする、他
- 不定長はCER（Canonical Encoding Rules：標準符号化規則）
- PKIの世界ではDERがメインでCERはほぼ無い

OID（Object IDentifier）

➢ OID（オブジェクト識別子）はデータ型や暗号の識別番号

例1	Description	Common name （証明書の所有者名等で使われる一般名）
	OID	2.5.4.3
	ASN.1	{ joint-iso-itu-t(2) ds(5) attributeType(4) commonName(3) }
	BER	06 03 55 04 03　[55 = 2 x 0x28 + 5]
	Information	This OID is defined in Annex A of Rec. ITU-T X.520 | ISO/IEC 9594-6 "The Directory: Selected attribute types".

例2	Description	Hash algorithm identifier SHA-1
	OID	1.3.14.3.2.26　この数値で検索してもだいたい説明ページが見つかる。
	ASN.1	{ iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) hashAlgorithmIdentifier(26) }
	BER	06 05 2B 0E 03 02 1A　[2B = 1 x 0x28 + 3 / 0E = 14 / 1A = 26]
	Information	National Institute of Standards and Technology (NIST). FIPS Pub 180-1: Secure Hash Standard. 17 April 1995.

PKCS（Public Key Cryptography Standards）

➢ RSA Security社策定の公開鍵暗号技術標準規格セット

Number	内容	RFC	補足
PKCS#1	RSA暗号	3447	RSA署名/暗号データの書式標準
PKCS#3	Diffie-Hellman鍵共有		DHを利用した鍵交換の実装仕様
PKCS#5	パスワードに基づく暗号化	2898	パスワードによる暗号化方式仕様
PKCS#7	暗号メッセージ構文 署名データや暗号化データ	2315	PDF署名等で利用 拡張版がCMS（RFC 5652）
PKCS#8	秘密鍵情報構文	5208	秘密鍵のデータフォーマット
PKCS#9	選択された属性タイプ		#6 #7 #8 #10用の属性定義
PKCS#10	証明書署名要求	2986	CAへの証明書要求メッセージ仕様
PKCS#11	暗号トークンインタフェース		ICカード利用時のAPI仕様
PKCS#12	個人情報交換構文		秘密鍵と証明書を格納する パスワードが必要　旧称PFX
PKCS#15	暗号トークン情報書式		ICカード利用時のファイル構造

PEM（Privacy Enhanced Mail）RFC 1421

- IETFで標準化された通信用データ形式

  IETF（Internet Engineering Task Force）は標準化任意団体。

  IETF標準化によりRFC（Request For Comments）化される。

- PEMはBER/DERをBase64化したテキスト

  各要素は”BEGIN 種類”と”END 種類”で囲まれる

```
-----BEGIN CERTIFICATE-----
MIIDOzCCArugAwIBAgIHA41+pMaAATANBgkqhkiG9w0BAQsFADBMMQswCQYDVQQG
EwJKUDERMA8GA1UEChMITGFuZ0VkZ2UxDTALBgNVBAsTBGRlbW8xGzAZBgNVBAMT
   （中略）
V/MNuU4+1jeadIC9icbxiXRcmwCoC1e1ItV3WhznEAeoGua2y8lXBWJYpvFwpGrk
dI1DfeDh98jp3JuoGmLN10732uMAzuw=
-----END CERTIFICATE-----
```

X.509証明書のPEM形式

電子署名付与の流れ

※ すみませんやっとここから電子署名付与の話です…

秘密鍵

署名対象

入力

電子署名
プログラム
（ライブラリ）

出力

埋め込まれる署名対象は
ハッシュ値の場合もある

署名データ

CMS署名

署名対象
署名方式
署名値

XML署名

署名対象
署名方式
署名値

証明書（公開鍵）と関連付いた秘密鍵の利用

➢ PKCS#12形式ファイル

- 認証局発行の場合は最終的にこの形式が多い。
- 利用時にはパスワードが必要。

➢ PEM形式ファイル

- OpenSSLの証明書や鍵等のテキスト保存形式。
- 秘密鍵の利用時にはパスワードが必要。

最も安全！

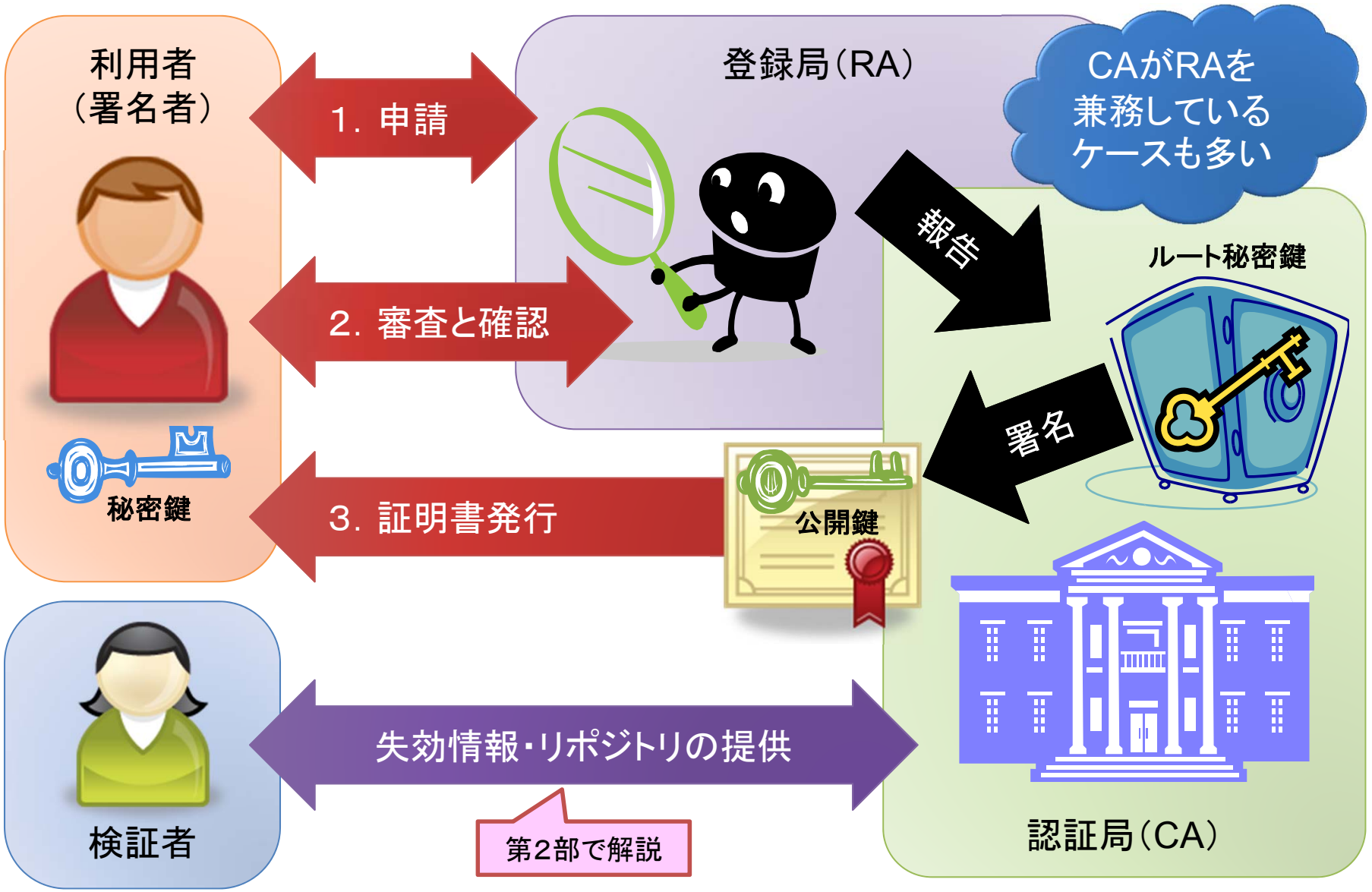
➢ ハードウェア格納（Secure Signature Creation Device）

- ICカード/USBトークン、サーバ側はHardware Security Module。

➢ 証明書ストア（※後述）

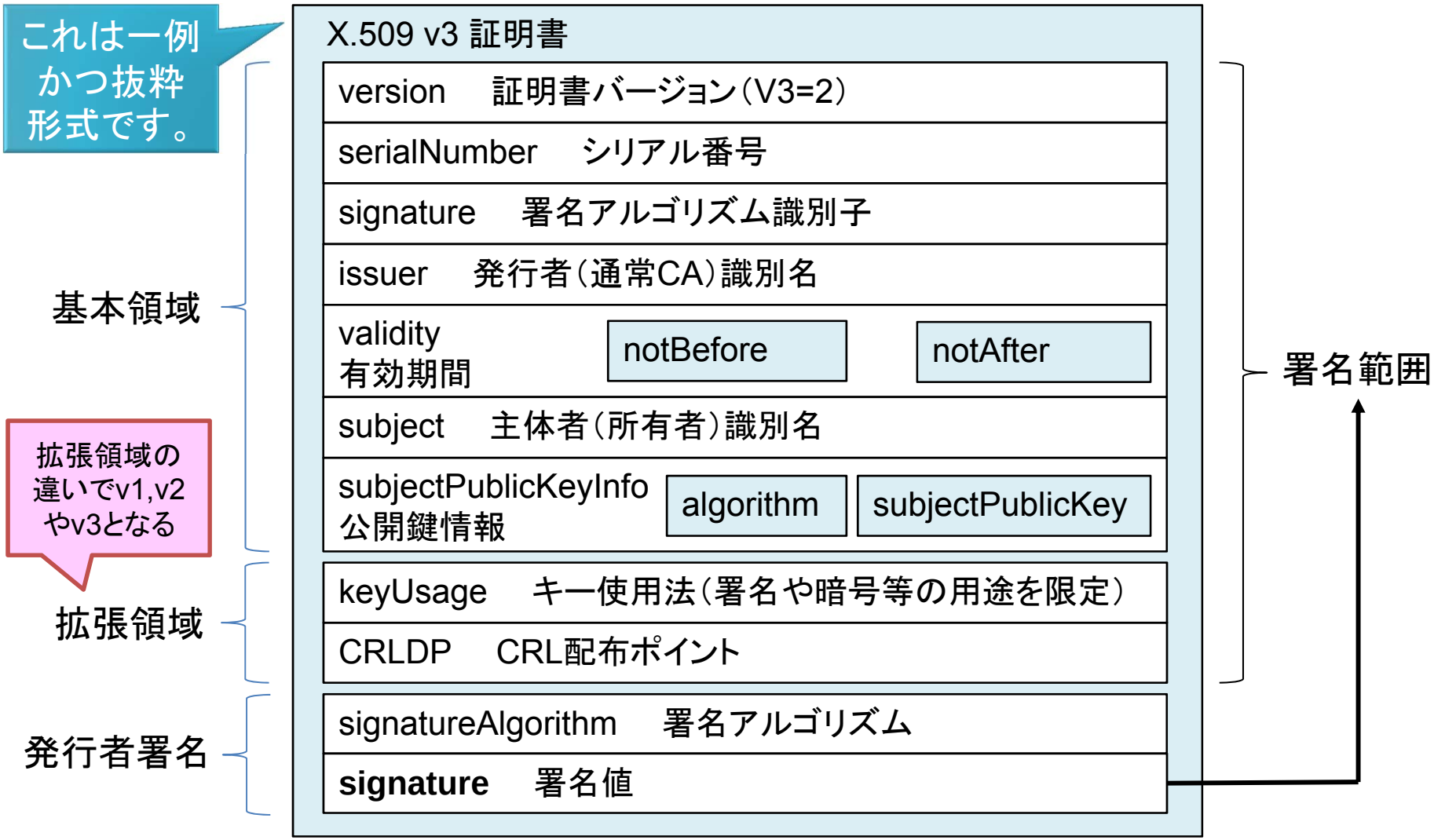
- OSや言語の管理ストアにインストールして利用。
- Windowsではログイン前提なのでパスワード不要。

証明書取得：認証局（CA）と登録局（RA）



X.509 v3 電子証明書 RFC 5280

➢ 発行者/主体者の情報と公開鍵情報に発行者が署名



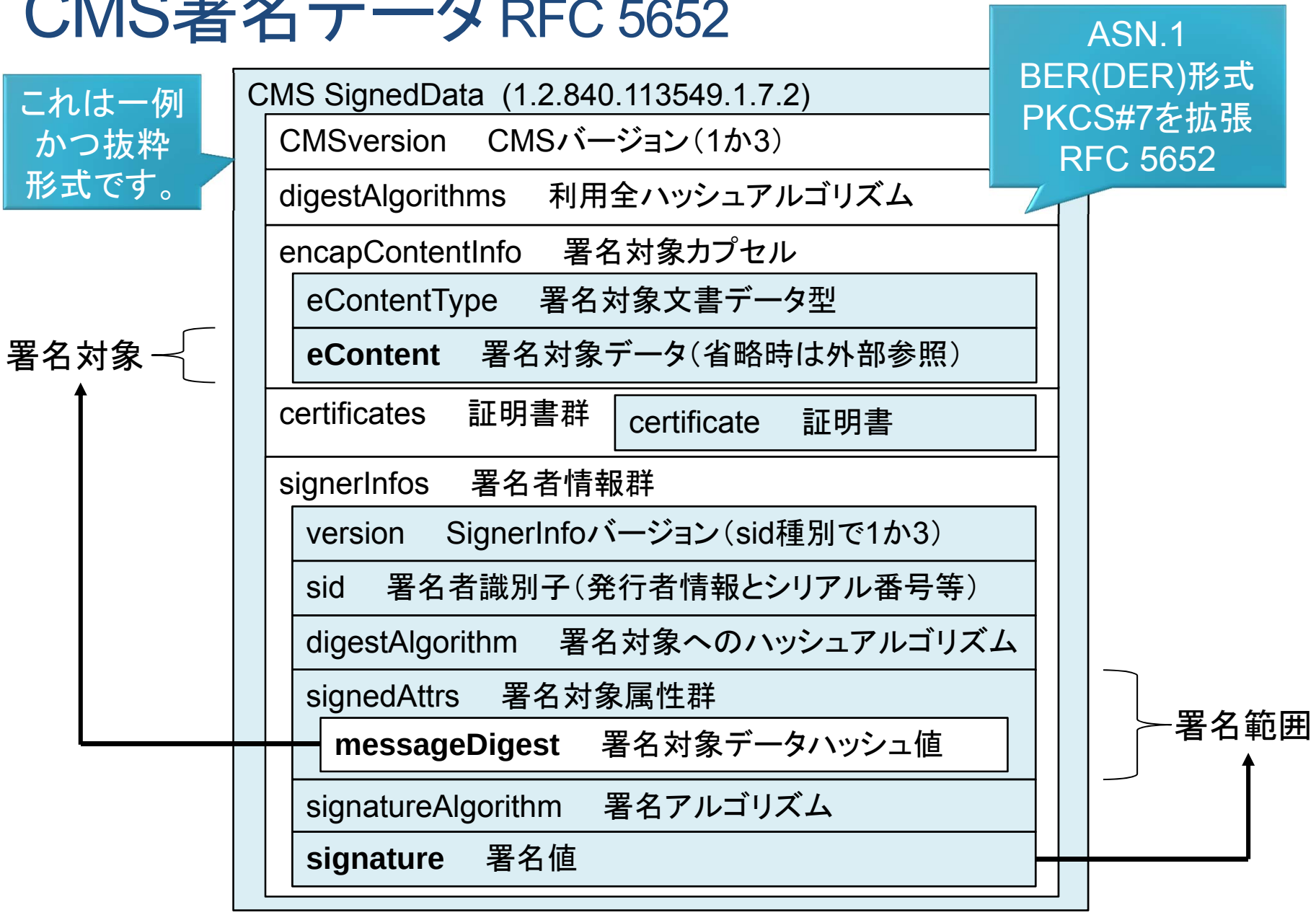
利用可能な認証局（証明書）

- パブリック証明書（公的またはブラウザ/OS等に標準搭載）

特定認証局	電子署名法の認定認証局（PKCS#12・ICカード） 帝国データバンク・セコムトラストシステムズ・ジャパンネット 等
商業登記証明書	法務局発行の電子認証登記所電子証明書（PKCS#12） ※ 法人代表者の証明書（ビジネス向け）
公的個人 認証サービス	個人向けに地方自治体から発行されるJPKI証明書（ICカード） ※ マイナンバーの個人番号カードに移行予定 ※ 署名のみ可能（検証は地方自治体・政府のみ可能）
WebTrust（米） ETSI認定（欧）	JCAN証明書 [WebTrust/ETSI認定] : JIPDEC Symantec（VeriSign） [WebTrust認定] :PKI Class1/Class2証明書（個人向け） :ドキュメントサイニング対応証明書

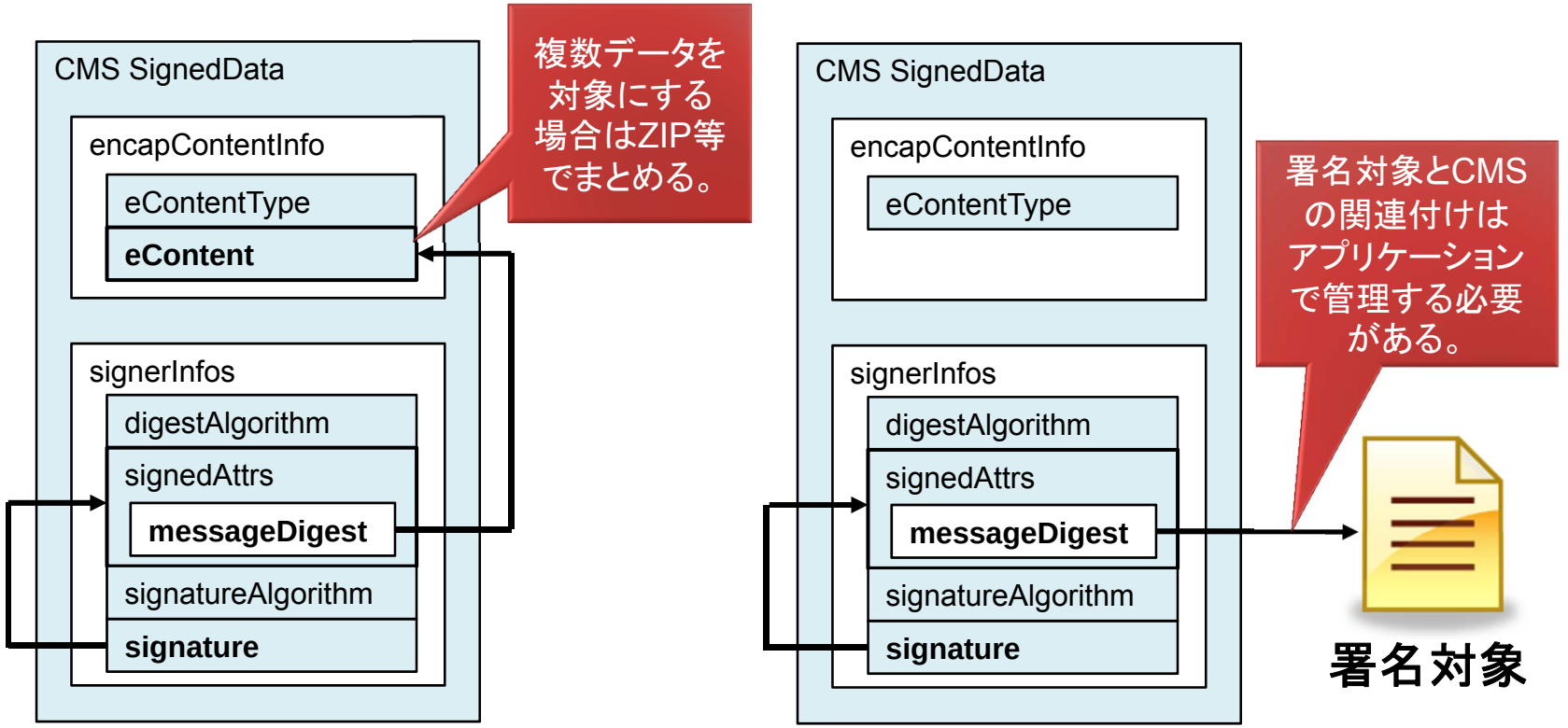
- プライベート証明書（企業や学術向けに発行される）
- 試験用証明書（JNSA PKI SandBox CA）
  OpenSSL利用の認証局でCRL発行しており試験利用可能

CMS署名データ RFC 5652



CMS署名：署名対象の選択

- Attached（内包）とDetached（外包）の2種類
- eContent要素の有無でどちらかの判定が可能



Attached: eContent要素から計算　　Detached: 外部ファイルから計算

XML署名（XmlDsig）データの構造

➢ XML形式の署名データ（ただし証明書等はASN.1/DER）

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
 <SignedInfo>
  <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
  <Reference URI="../aaa.txt">
   <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
   <DigestValue>mDSHbc+wXLFnpcJJU+uljErImxrfV/KPL5OJrxB+6PA=</DigestValue>
  </Reference>
 </SignedInfo>
 <SignatureValue>mioSNze...Av6D3MA==</SignatureValue>
 <KeyInfo Id="MyKeyInfoId">
  <KeyValue>
   <RSAKeyValue>
    <Modulus>n4eFDiIF...RGO9eMQ==</Modulus>
    <Exponent>AQAB</Exponent>
   </RSAKeyValue>
  </KeyValue>
  <X509Data>
   <X509Certificate>MIIEBTCCA...po+oGkQNw==</X509Certificate>
  </X509Data>
 </KeyInfo>
</Signature>
```

署名範囲

署名対象の参照

署名対象のハッシュ値

署名値

鍵情報

バイナリデータ（ASN.1/DER等）は全てBase64化して埋め込む

証明書（ASN.1/DER）

XML署名：署名対象（複数可）の指定

Detached: 外部ファイルをURI指定

DetachedOut.xml

```
<Signature>
  <SignedInfo>
    <Reference URI="target.xml">
  </SignedInfo>
  ...
</Signature>
```

Target.xml

```
<MyData>
  <Data>...</Data>
</MyData>
```

XML形式以外の
任意ファイルを指定可能

Enveloping: 同じファイル内 Object Id指定

Enveloping.xml

```
<Signature>
  <SignedInfo>
    <Reference URI="#MyObjId">
  </SignedInfo>
  ...
  <Object Id="MyObjId">
    <MyData>
      <Data>...</Data>
    </MyData>
  </Object>
</Signature>
```

※良く使われる指定方式

Detached: 同じファイル内の別要素Id指定

DetachedIn.xml

```
<MyData>
  <Data Id="D1">...</Data>
  <MySign>
    <Signature>
      <SignedInfo>
        <Reference URI="#D1">
      </SignedInfo>
      ...
    </Signature>
  </MySign>
</MyData>
```

Enveloped: 同じファイル全体（署名除く）

Enveloped.xml

```
<MyData>
  <Data>...</Data>
  <Signature>
    <SignedInfo>
      <Reference URI="">
        ...<Transform Algorithm="...#enveloped-signature">...
      </Reference>
    </SignedInfo>
    ...
  </Signature>
</MyData>
```

XML署名：正規化 Canonical XML（c14n）

- XML-c14nはXMLデータ正規化に関するW3C勧告
  - 「Canonicalization」のCとnとの間に14文字あることから「c14n」と呼ぶ。
- XMLのXML署名対象はXML-c14nにて正規化する
  - XmlDsig署名対象は<ds:Transform Algorithm="">で変換指定が可能。
  - おまけ：バイナリの署名対象はBase64化の変換指定が必要。

XML-c14n	http://www.w3.org/TR/2001/REC-xml-c14n-20010315
Base64	http://www.w3.org/2000/09/xmldsig#base64

```
<doc>
<a bb="2" aa="1">&#x31;&#x32;&#x33;</a>
</doc>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
  <a aa="1"
     bb="2">123</a>
</doc>
```

c14n

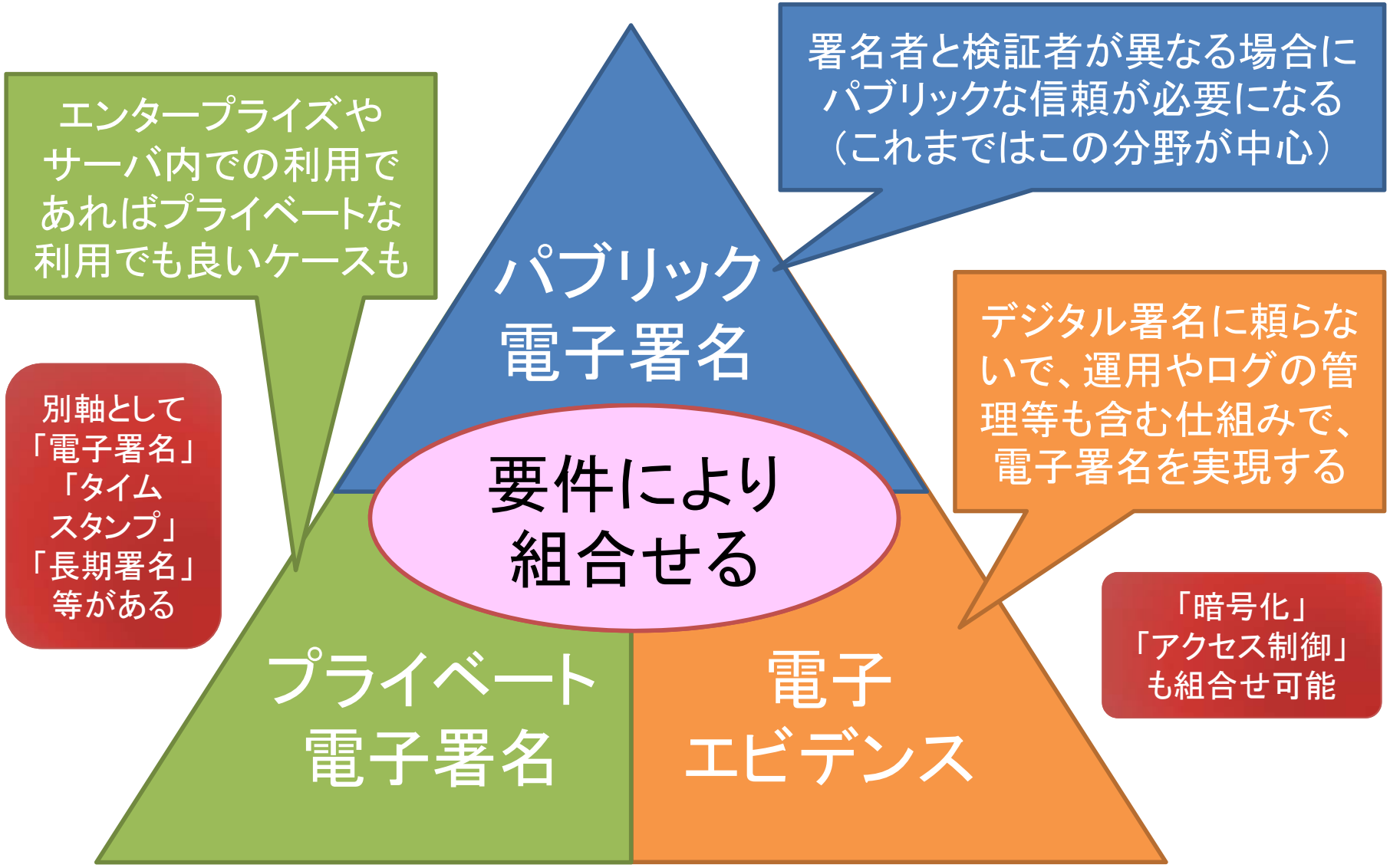
c14n

```
<doc>
<a aa="1" bb="2">123</a>
</doc>
```

見やすいようにdoc要素の改行だけは正規化せず。

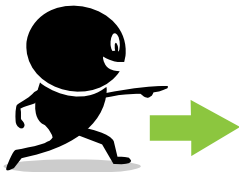
XML正規化のサンプル

付録：広義の電子署名における組合せ



電子署名(PKI)ハンズオン

第1部:電子署名とタイムスタンプ



1. 電子署名とその基本技術を理解する
2. **電子署名を生成する(ハンズオン1)**
3. タイムスタンプ技術を理解する
4. タイムスタンプを取得する(ハンズオン2)

ハンズオン1：電子署名を生成する（概要）

a. OpenSSLを使ってCMS署名ファイルを生成する

※ ハンズオンディレクトリ OpenSSL の下で操作

コマンドの実行は本資料からのコピー＆ペーストで構いません

1-1：OpenSSL用に鍵形式の変換（署名準備）

PKCS#12形式からOpenSSL用のPEM形式へ変換

1-2： CMS署名ファイルの作成（署名付与）

1-3： CMS署名ファイルの検証

b. JavaプログラムでXML署名ファイルを生成する

※ ハンズオンディレクトリ Java の下で操作

1-4：Javaソースのビルドと実行

Java7から標準となったXML署名を利用しています

Javaソースが読める人はソースを確認してください

ハンズオン1-1：OpenSSL用に鍵形式の変換

手順1）秘密鍵と証明書を、PKCS#12形式からOpenSSL用のPEM形式へ変換

```
> openssl pkcs12 -in signer.p12 -clcerts -out signer.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Enter Import Password:test1
MAC verified OK
Enter PEM pass phrase:test1
Verifying - Enter PEM pass phrase:test1
>
```

MacOSは ./openssl

警告は無視

用意されたsigner.p12 のパスワード

test.pem に付けるパスワード（好きな文字列）

繰り返し

➢ openssl pkcs12 引数（PKCS#12の操作）

今回は事前に用意された証明書・秘密鍵を利用する

-in　　：入力（PKCS#12形式の証明書＋秘密鍵）
-clcerts：クライアント（署名者）証明書のみ出力（指定しないと全証明書を出力）
-out　　：出力（PEM形式の証明書＋秘密鍵）

□ PKCS#12形式（バイナリ形式）

RSA Security社策定の公開鍵暗号技術標準規格セットのうち、秘密鍵と証明書を格納する形式。秘密鍵はパスワードが必要。

□ PEM形式（Base64化されたテキスト形式）

テキストエディタで中を見てみよう

OpenSSLの標準的なバイナリを利用する形式。秘密鍵はパスワードが必要。

ハンズオン1-2：CMS署名ファイルの作成（署名付与）

手順2）CMS署名の実行

MacOSは ./openssl

```
> openssl cms -sign -in aaa.txt -out sign.cms -signer signer.pem -outform DER
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for test.pem:test1
>
```

cert.pem に付けたパスワード（手順1で指定）

- openssl cms 引数（CMS署名の操作）
  - -sign　：署名の付与操作を実行
  - -in　　：入力（署名対象ファイルを指定）
  - -out　　：出力（作成するCMS署名ファイルを指定）
  - -signer：署名者指定（PEM形式の証明書＋秘密鍵）
  - -outform：出力形式（DER形式を指定、指定しないとPEM形式になる）

- 署名対象ファイル aaa.txt
  - 内容は “aaa” の3バイトのみのテキストファイル。

MacOS-X

MacOS標準で提供さているopensslは古い0.9.8系が多いので、提供フォルダにある1.0.0系を利用すること（バージョン確認方法 $ openssl version）

ハンズオン1-3：CMS署名ファイルの検証

手順3）CMS署名の結果確認（検証）

MacOSは ./openssl

```
> openssl cms -verify -in sign.cms -inform DER -content aaa.txt -no_signer_cert_verify
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
aaaVerification successful
>
```

sucessful で検証成功

- openssl cms 引数（CMS署名の操作）
  - -verify　：署名の検証操作を実行
  - -in　　：入力（CMS署名ファイルを指定）
  - -inform：入力形式（DERで出力したのでDER形式を指定）
  - -content　：署名対象（署名対象ファイルを指定）
  - -no_signer_cert_verify　：署名証明書を検証しない（署名値のみ検証）

証明書の検証は第2部で説明

◆検証が出来たら aaa.txt を修正（改ざん）して再検証してみる。

```
> openssl cms -verify -in sign.cms -inform DER -content aaa.txt -no_signer_cert_verify
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
？？？？？
>
```

どうなる？？

ハンズオン1-4：電子署名Java/XML署名編

JavaによるXML署名生成と検証

MacOSは ./JavaSignBuild.sh

```
> JavaSignBuild.bat
> javac -classpath .;./commons-codec-1.3.jar SignSample.java
```

コンパイル実行。
common-codecはBase64用

MacOSは ./JavaSignSample.sh

```
> JavaSignSample.bat
> java -classpath .;./commons-codec-1.3.jar SignSample aaa.txt signer.p12 test1 xmldsig.xml
Java XmlDsig Test
target file : aaa.txt
sign cert   : signer.p12
output file : xmldsig.xml
Sign OK.
Verify OK.
Done.

>
```

署名と検証の実行。
引数1：署名対象
引数2：PKCS#12ファイル
引数3：PKCS#12パスワード
引数4：XML署名出力ファイル

検証に成功したら aaa.txt やソースを修正して再度検証してみよう

- XML署名自体は標準でサポートされています（Base64除く）。
- 検証は手抜きで署名者（証明書）は確認していません。
- 出力ファイル xmldsig.xml をIE等のブラウザで開いてみよう。
- Javaソースが読める人は SignSample.java を読もう。

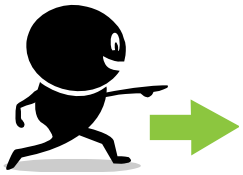
ハンズオン1-5：生成XML署名ファイル（参考）

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
    <Reference URI="aaa.txt">
      <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
      <DigestValue>mDSHbc+wXLFnpcJJU+uljErImxrfV/KPL5OJrxB+6PA=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>wDOn7l..(略)..QvyON9w==</SignatureValue>
  <KeyInfo Id="MyKeyInfoId">
    <KeyValue>
      <RSAKeyValue>
        <Modulus>7RPbdUaj5..(略)..qdzDQqRrQ==</Modulus>
        <Exponent>AQAB</Exponent>
      </RSAKeyValue>
    </KeyValue>
    <X509Data>
      <X509Certificate>MIIEGDCCAwC..(略)..+UKFSbtKk=</X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

xmldsig.xml

電子署名(PKI)ハンズオン

第1部:電子署名とタイムスタンプ



1. 電子署名とその基本技術を理解する
2. 電子署名を生成する(ハンズオン1)
3. **タイムスタンプ技術を理解する**
4. タイムスタンプを取得する(ハンズオン2)

タイムスタンプとは？

- 一般にタイムスタンプと言えば単なる日時データ。
  例：<CreateDate>2014-09-02T16:11:58+09:00</CreateDate>
- 日時データはそのシステム日時を利用している。
  システムが適格に運用されていれば信頼しても良い？
- 後で日時を改ざんされても分からない!?

これ以降タイムスタンプと書けばRFC 3161です。

「PKIベースのタイムスタンプはRFC 3161で規定」

- RFC 3161 over HTTP はサーバ署名の一種
  - 1：第三者サーバで管理された時刻で保証される。
  - 2：電子署名の一種なので後から改ざんができない。
- ✓ サーバ署名なのでHTTP通信で要求・応答を行う。
- ✓ 要求も応答もASN.1/DER形式のバイナリデータ。

電子署名とタイムスタンプの比較

電子署名
で解決

誰が？
認証局が発行した証明書を見ることで信頼された人かどうかを判定できます。

改竄？
改竄があれば数学的な仕組みですぐに判定が出来ます。

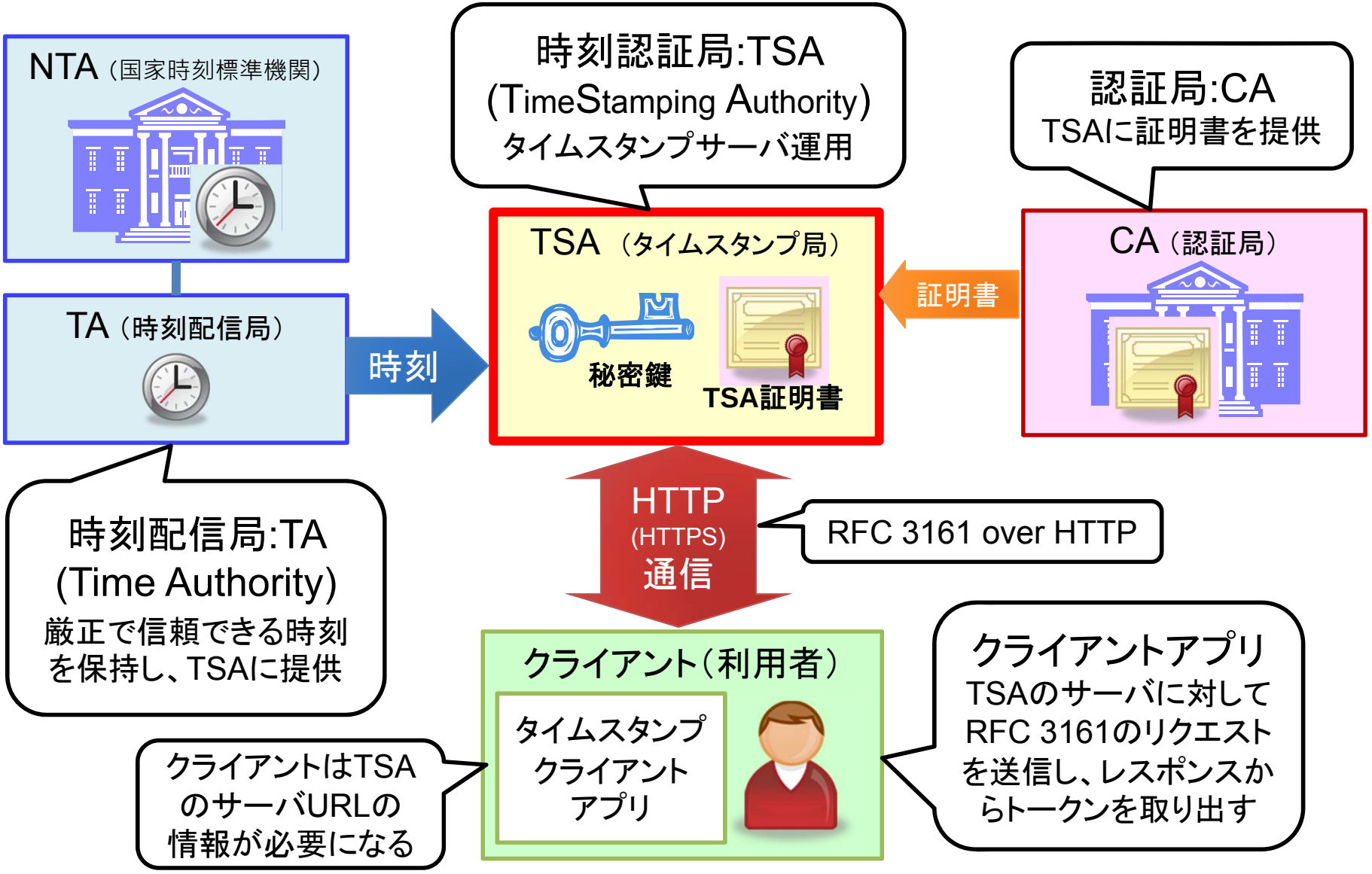
タイムスタンプ
で解決

いつ？
タイムスタンプ局が発行したタイムスタンプに埋め込まれた日時は保証されています。

改竄？
タイムスタンプも電子署名の1種ですので、タイムスタンプだけで改竄判定ができます。

課題：証明書の有効期間
一般に署名で数年、タイムスタンプで10年程度
検証情報の発行は有効期間内のみ＆暗号危殆化

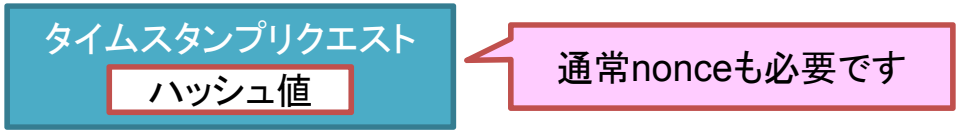
タイムスタンプサービスの構成



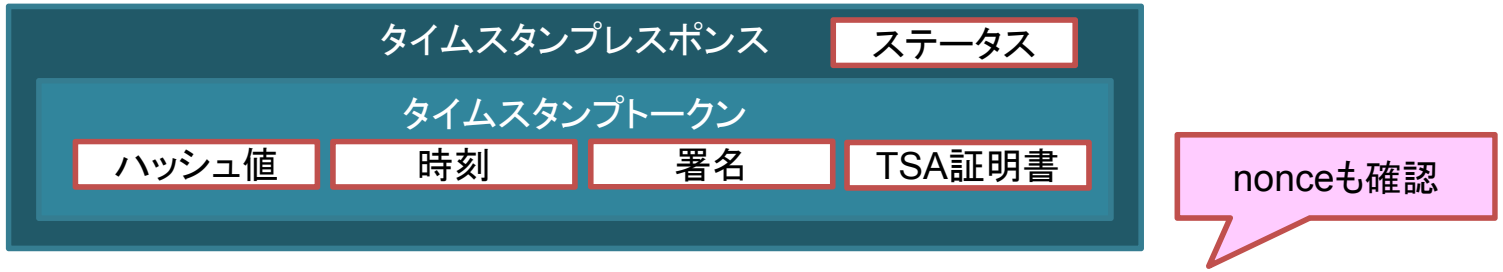
タイムスタンプ取得手順

データは全てASN.1/DER形式 (RFC3161)

1. 対象データのハッシュ値を計算
2. ハッシュ値を埋め込んだタイムスタンプリクエスト生成



3. タイムスタンプリクエストをサーバに送信
4. サーバからタイムスタンプレスポンスを取得



5. タイムスタンプレスポンスのステータスを確認
6. レスポンスからタイムスタンプトークンを取り出し利用

タイムスタンプ リクエスト

```
// SHA-1タイムスタンプリクエスト情報BER定義
byte[] sha1req = {
 0x30, 0x31,                                    // Request SEQUENCE (49バイト)
 0x02, 0x01, 0x01,                              // Version INTEGER (1バイト) value:1
 0x30, 0x1f,                                    // MessageImprint SEQUENCE (31バイト)
 0x30, 0x07,                                    // AlgorithmOID SEQUENCE (7バイト)
 0x06, 0x05,                                    // OID (5バイト)
 0x2b, 0x0e, 0x03, 0x02, 0x1a,                  // OIDSHA1 value: 1.3.14.3.2.26
 0x04, 0x14,                                    // Hash OCTET STRING (20バイト)
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 0x02, 0x08,                                    // Nonce INTEGER (8バイト)
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 0x01, 0x01, 0xff                               // RequestCertificate BOOLEAN (1バイト) value:true
};
```

全て固定長なのでバイナリで準備をしておけば良い。Javaソース参照。

SHA-1なら20バイトのハッシュ値をセットすれば良い

Nonceは8バイト(任意)固定

```
TimeStampReq ::= SEQUENCE  {
 version                  INTEGER  { v1(1) },
 messageImprint           MessageImprint,
 reqPolicy                TSAPolicyId OPTIONAL,
 nonce                    INTEGER OPTIONAL,
 certReq                  BOOLEAN DEFAULT FALSE,
 extensions               [0] IMPLICIT Extensions OPTIONAL
}
```

```
MessageImprint ::= SEQUENCE  {
 hashAlgorithm   AlgorithmIdentifier,
 hashedMessage   OCTET STRING
}
```

タイムスタンプ レスポンス

```
TimeStampResp ::= SEQUENCE  {
 status          PKIStatusInfo,
 timeStampToken TimeStampToken OPTIONAL
}
PKIStatusInfo ::= SEQUENCE {
 status        PKIStatus,
 statusString PKIFreeText OPTIONAL,
 failInfo     PKIFailureInfo OPTIONAL
}
```

成功(granted)ならタイムスタンプトークンが含まれる

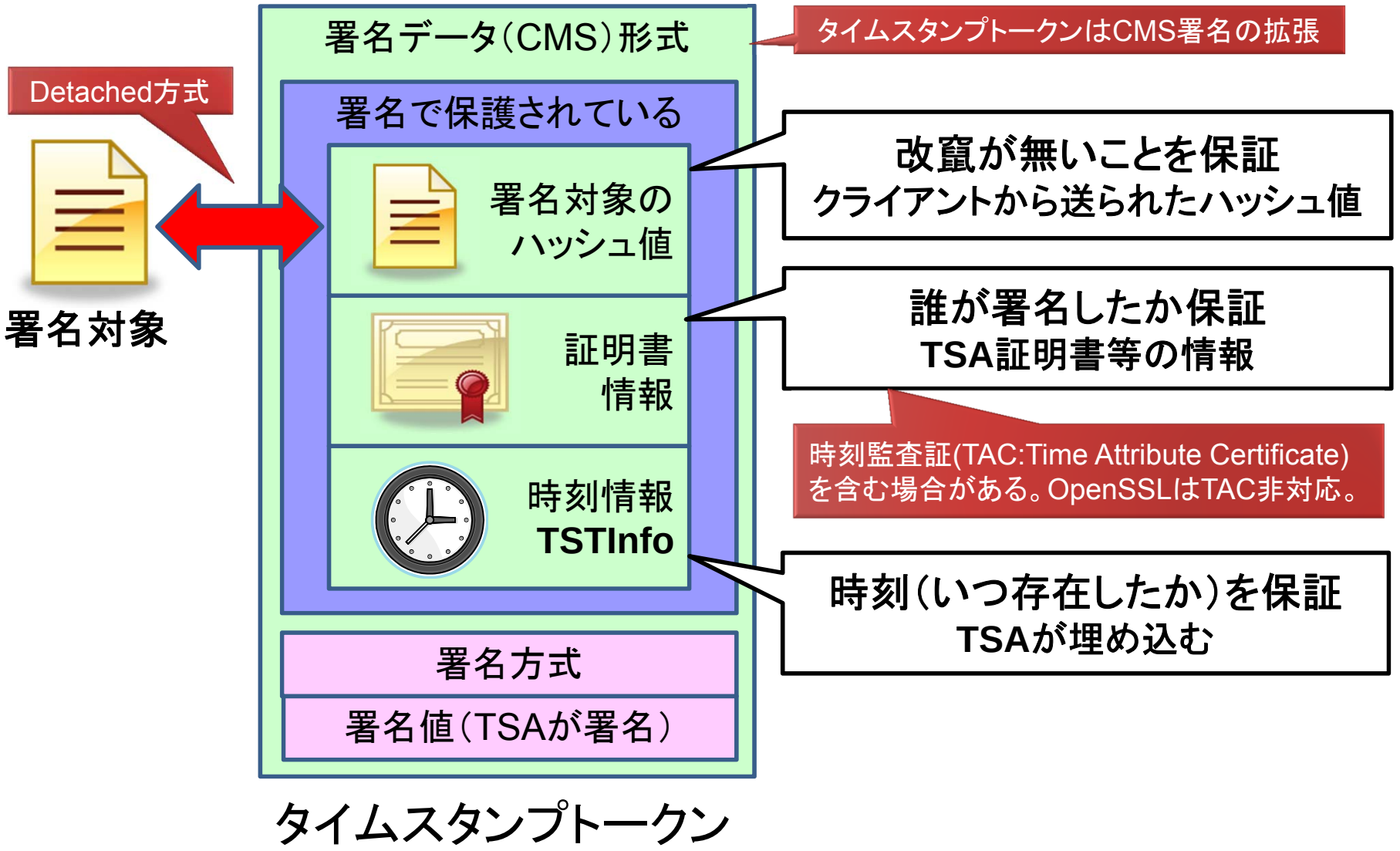
statusをチェックする

```
// レスポンスのステータス
PKIStatus ::= INTEGER {
 granted                (0),    // TSTを含む
 grantedWithMods        (1),    // TSTを含み、プライベート拡張を含む
 rejection              (2),    // TSTを含まず、拒否された
 waiting                (3),    // TSTを含まず、レシートのみ含む
 revocationWarning      (4),    // TSTを含まず、TSA証明書の失効が近い
 revocationNotification (5)     // TSTを含まず、TSA証明書が失効している
}
```

OK

- タイムスタンプのレスポンスは簡単な構造なのでバイナリとして解析すれば、PKIライブラリは無くても利用可能。
- Javaサンプルソースを参照。

タイムスタンプ トークン RFC 3161



利用可能なタイムスタンプサービス

- 時刻認証業務認定事業者（認定タイムスタンプサービス）
  http://www.dekyo.or.jp/tb/list/

アマノタイムスタンプサービス3161 **（アマノ株式会社）** http://www.e-timing.ne.jp/	Type-T　一般向けの基本サービス
	Type-S　VPN高速発行サービス
SEIKO認定タイムスタンプ　サービス **（セイコーソリューションズ株式会社）** https://www.seiko-cybertime.jp/	VPN接続タイプ　帯域保証の定額サービス
	SSL接続タイプ　ネット利用の従量課金制
S.T.E.P Time Carve 時刻認証サービス **（北海道総合通信網株式会社）** http://www.hotnet.co.jp/security/timecarve.html	L2L　HOTnet L2L接続
	C2C　インターネットと専用端末
※ 認定サービスのうちアーカイビング方式は特殊なのでここでは説明はしない。	

- 試験用タイムスタンプサービス（FreeTSA Project）
  http://eswg.jnsa.org/freetsa　（Ruby運用版）
  http://www.langedge.jp/tsa　（Perl運用版）

タイムビジネス認定制度

- タイムビジネス認定制度は2004年11月に策定
- 「タイムビジネスに関わる指針」（総務省）
  - 「タイムスタンプビジネス」のガイドライン
    第三者が証明可能な時刻情報を付与する。
  - 日本データ通信協会による任意の認定制度
    → 技術面ではNICTやTBF（タイムビジネス協議会）と連携
  - 認定タイムスタンプ局は多くの法律の要件
    → e-文書法にも認定タイムスタンプ局の利用が明記

※ 総務省の管轄　総務省は標準時を管理している

FreeTSA:フリータイムスタンプ プロジェクト

- 自由に使えるタイムスタンププロジェクト
- OpenSSLではv1.0.0以降にタイムスタンプのAPIやコマンドが使えるようになった
  - v0.9.8の頃にあったOpenTSAからソース移管された
- FreeTSAはOpenSSL v1.0.0のタイムスタンプ機能を使って実装したタイムスタンプサービス
- JNSA PKI SandBox Projectサービス公開中
  - Ruby用のソースコードも公開予定
  - 以下に詳細情報あり

  http://eswg.jnsa.org/sandbox/freetsa

参考:e-文書法

- e-文書法は2005年4月に施行された
- 適用されるのは各府省が定める約250省令

  ※「e-文書法によって電磁的記録が可能となった規定」

  http://www.kantei.go.jp/jp/singi/it2/others/syourei.pdf
- 適用には要件(省令毎)を満たす必要がある

  見読性:スキャン等入力時やフォーマットの問題

  完全性:電子署名・タイムスタンプや保管の問題

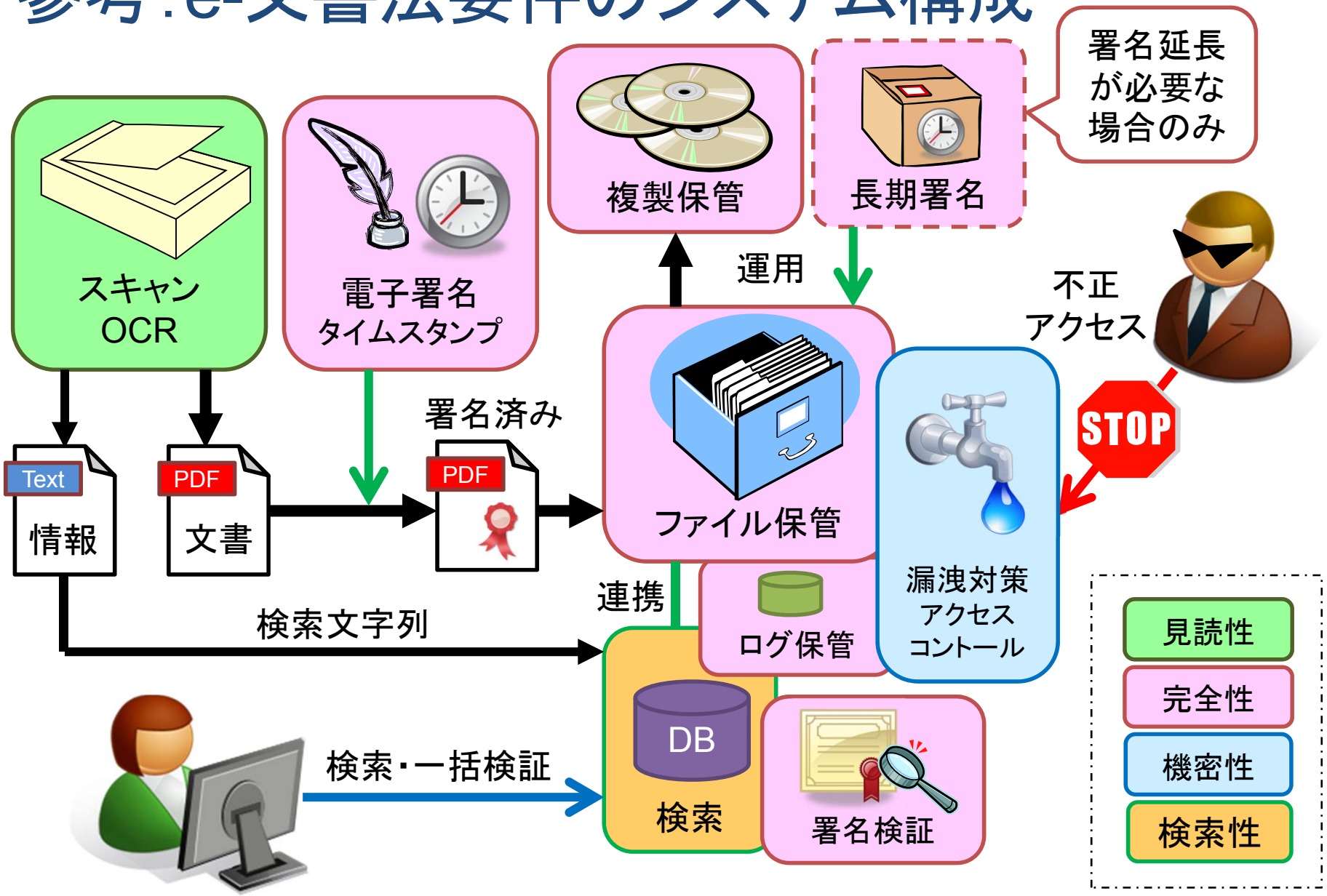
  機密性:漏洩対策・アクセス制御等の運用の問題

  検索性:入力時キーワードセットや検索DBの問題

  ※「文書の電子化・活用ガイド」

  http://www.meti.go.jp/policy/it_policy/e-doc/guide/e-bunshoguide.pdf

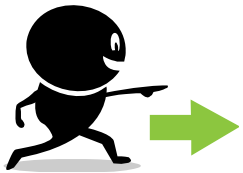
参考：e-文書法要件のシステム構成



電子署名(PKI)ハンズオン

第1部:電子署名とタイムスタンプ

1. 電子署名とその基本技術を理解する
2. 電子署名を生成する(ハンズオン1)
3. タイムスタンプ技術を理解する



4. タイムスタンプを取得する(ハンズオン2)

ハンズオン2：タイムスタンプを取得する（概要）

a. OpenSSLとcURLを使ってタイムスタンプを取得する

※ ハンズオンディレクトリ OpenSSL の下で操作

2-1：タイムスタンプリクエストの生成

2-2： タイムスタンプリクエストの送信

HTTP通信にはcURLプログラムを利用します

2-3： タイムスタンプレスポンスの検証

b. Javaプログラムでタイムスタンプを取得する

※ ハンズオンディレクトリ Java の下で操作

2-4：Javaソースのビルドと実行

タイムスタンプ取得に必要バイナリを生成・解析します

Javaソースが読める人はソースを確認してください

ハンズオン2-1：タイムスタンプリクエストの生成

手順1）タイムスタンプリクエストの生成

MacOSは./openssl

```
> openssl ts -query -data aaa.txt -cert -sha512 -out req.tsq
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Loading 'screen' into random state - done
>
```

警告は無視

- openssl ts 引数（タイムスタンプの操作）
  - -query ：タイムスタンプリクエストの操作を行う
  - -data　：入力（タイムスタンプ対象ファイルを指定）
  - -cert　：返されるタイムスタンプトークンにTSA証明書を含むことを要求
  - -sha512：要求するタイムスタンプのハッシュアルゴリズム指定（SHA-2の512ビット）
  - -out　：出力（作成するタイムスタンプリクエストのファイルを指定）

- タイムスタンプ対象ファイル aaa.txt
  - 内容は “aaa” の3バイトのみのテキストファイル。

- 生成が出来たら req.tsq をASN.1解析して表示してみよう。

```
> openssl asn1parse -in req.tsq -inform DER
？？？？？
>
```

どうなる？？ 47ページも参考に

ハンズオン2-2：タイムスタンプリクエストの送信

手順2）タイムスタンプリクエストのHTTP送信とレスポンスの取得

```
> curl -H "Content-Type:timestamp-query" "http://eswg.jnsa.org/freetsa" --data-binary @req.tsq -o res.tsr
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  3013    0  2911  100   102  26463    927 --:--:-- --:--:-- --:--:-- 30968
>
```

- curl 引数（HTTP通信の操作）
  - -H “arg”　：ヘッダを付ける（timestamp-queryのコンテンツタイプ指定）
  - “URL”　：HTTP通信するするサーバのURLを指定する
  - --data-binary　：入力データをバイナリとしてそのまま送信する
  - @入力ファイル　：指定ファイルを入力データとして利用する
  - -o 出力ファイル　：サーバからの応答を出力ファイルに保存する

- cURL（Client for URLs）

  URLで示される場所からデータを様々なプロトコルを用いて送受信することができるプログラム。Unix/Linuxでは通常標準搭載されている。オープンソースソフト。OpenSSL単体ではHTTP通信が出来ない為に利用している。

  http://curl.haxx.se/

ハンズオン2-3：タイムスタンプレスポンスの検証

手順3）タイムスタンプレスポンスを使った検証

MacOSは./openssl

```
> openssl ts -verify -data aaa.txt -in res.tsr -CAfile root2.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Verification: OK
>
```

OK で検証成功

- openssl ts 引数（タイムスタンプの操作）
  - -verify　：タイムスタンプの検証操作を実行
  - -data　　：タイムスタンプ対象（対象ファイルを指定）
  - -in　　：入力（タイムスタンプレスポンスファイルを指定）
  - -CAfile：ルート証明書（事前に用意済みのPEM形式ルート証明書ファイルを指定）

ルート証明書に関しては第2部で解説する

◆ 検証が出来たら aaa.txt を修正（改ざん）して再検証してみる。

```
> openssl ts -verify -data aaa.txt -in res.tsr -CAfile root2.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
？？？？？
>
```

どうなる？？

問題：OpenSSLではタイムスタンプトークンの抽出・検証ができない。

ハンズオン2-4：タイムスタンプ取得Java編

Javaによるタイムスタンプ取得

```
> JavaTimeStampBuild.bat
> javac TimeStampSample.java
>
> JavaTimeStampSample.bat
> java TimeStampSample aaa.txt http://eswg.jnsa.org/freetsa ts3161.tst
Java TimeStamp Test
target file : aaa.txt
server url  : http://eswg.jnsa.org/freetsa
output file : ts3161.tst
TimeStamp Get OK.
Done.
>
```

MacOSは ./JavaTimeStampBuild.sh

MacOSは ./JavaTimeStampSample.sh

タイムスタンプ対象ファイルは aaa.txt

FreeTSAを利用

タイムスタンプリクエストからタイムスタンプトークンを取り出してます

- PKIライブラリを使わずにタイムスタンプ取得するサンプルです。
- 検証は普通PKIライブラリが必要なので実装していません。
- Javaソースが読める人は TimeStampSample.java を読もう。
- このJavaソースは商用利用も含め自由にお使い頂けます。

※ ただし保証は一切ありませんのであくまで自己責任でお使いください。

ハンズオン2-5：タイムスタンプトークン内容確認

OpenSSLでタイムスタンプトークンをASN.1解析してみる（時間のある人だけで結構です）

```
> ..¥OpenSSL¥openssl asn1parse -in ts3161.tst -inform DER
WARNING: can't open config file: /usr/local/ssl/openssl.cnf
    0:d=0  hl=4 l=2895 cons: SEQUENCE
    4:d=1  hl=2 l=   9 prim: OBJECT            :pkcs7-signedData
  （省略）
 2389:d=9  hl=2 l=   3 prim: OBJECT            :countryName
 2394:d=9  hl=2 l=   2 prim: PRINTABLESTRING   :JP
  （省略）
 2402:d=9  hl=2 l=   3 prim: OBJECT            :organizationName
 2407:d=9  hl=2 l=   4 prim: PRINTABLESTRING   :JNSA
  （省略）
 2417:d=9  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
 2422:d=9  hl=2 l=   4 prim: PRINTABLESTRING   :ESWG
  （省略）
 2432:d=9  hl=2 l=   3 prim: OBJECT            :commonName
 2437:d=9  hl=2 l=  26 prim: PRINTABLESTRING   :JNSA PKI Sandbox CA Root 2
 2465:d=6  hl=2 l=   3 prim: INTEGER           :300001
  （省略）
 2514:d=7  hl=2 l=   9 prim: OBJECT            :signingTime
 2525:d=7  hl=2 l=  15 cons: SET
 2527:d=8  hl=2 l=  13 prim: UTCTIME           :150310043857Z
 2542:d=6  hl=2 l=  35 cons: SEQUENCE
  （省略）
>
```

MacOSは ../OpenSSL/openssl …

TSA証明書情報

タイムスタンプ時刻

第1部終了:ちょっと一休み

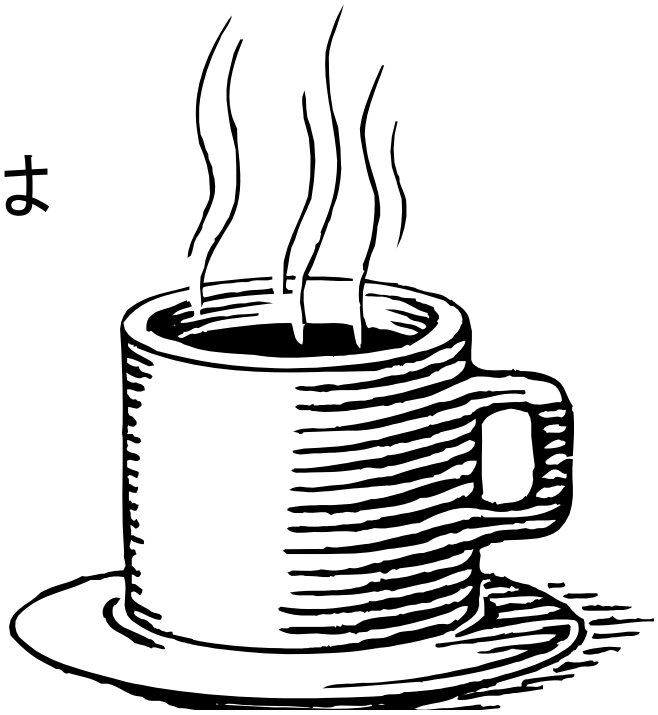
電子署名はむずかしい話ばかりで疲れますね(^_^;
いきなり全部の理解は難しいと思いますので、まずは

ASN.1/BER(DER) というバイナリ形式がある

とだけ覚えておきましょう。

もう一つ秘密鍵は他人に漏れては
いけない情報です。運用が重要。

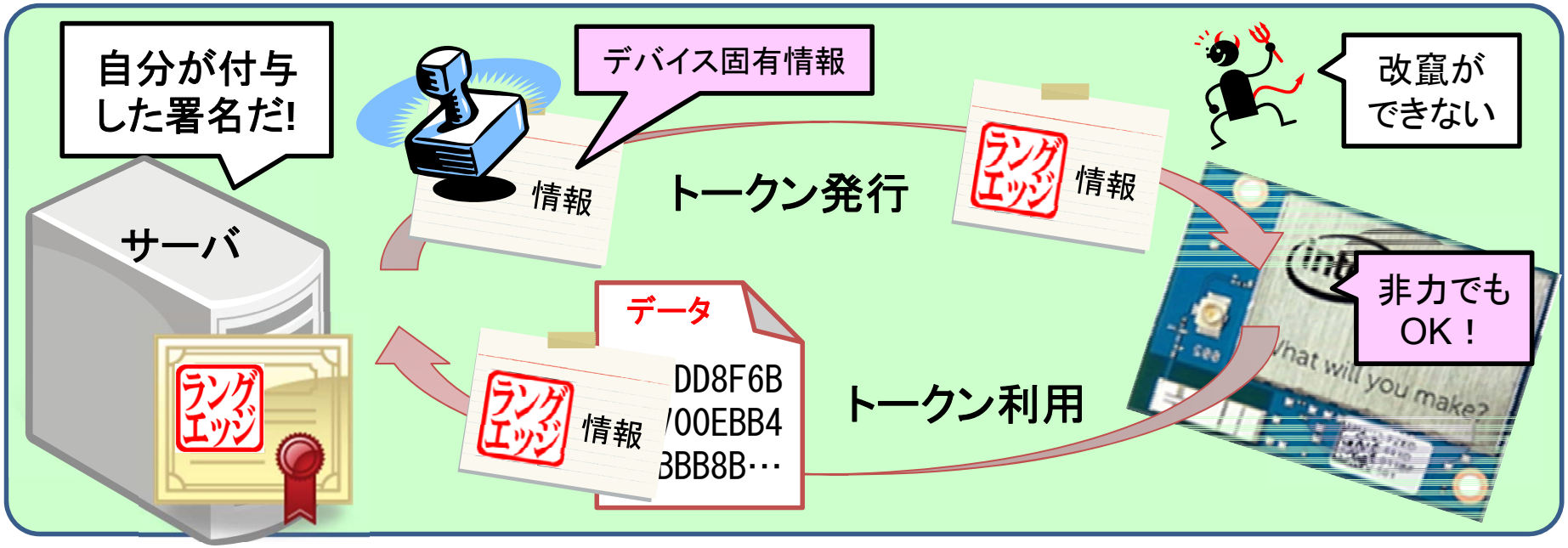
あとは必要に応じて検索すれば
色々と見つかると思います。
次は検証と長期署名のお話です！



付録：クラウド/IoT時代の電子署名応用

➢ 認証トークン（電子チケット）

自分で署名したトークン（チケット）を配布して利用時にトークンの署名を確認して判断。



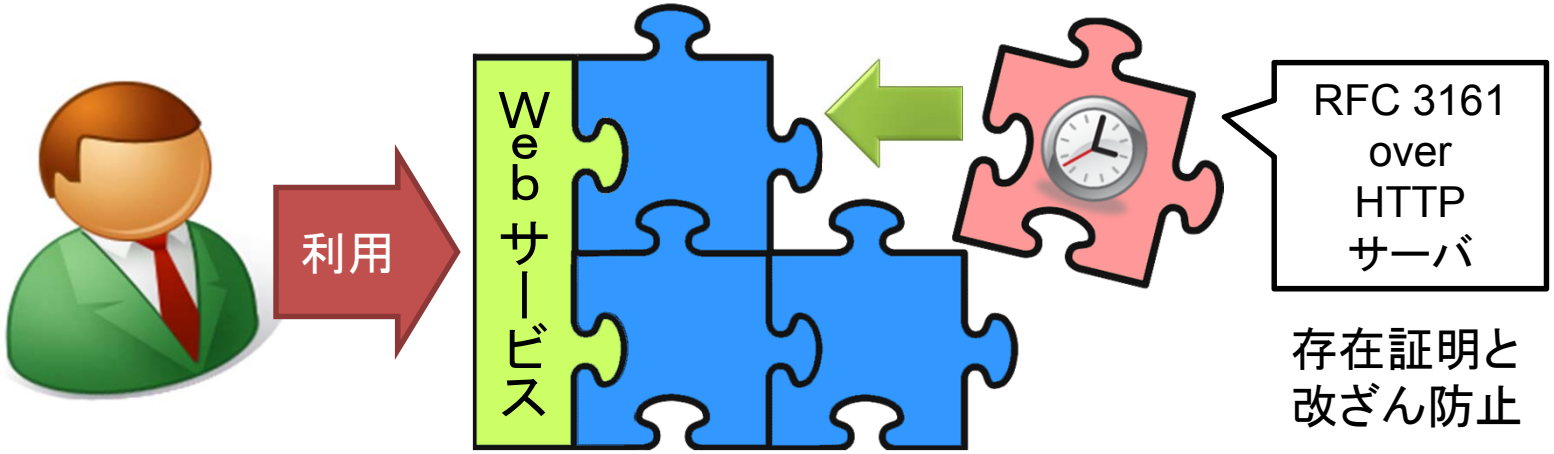
➢ 公開データの作成者確認

公開データの配布元が電子署名して公開してくれるならば、不正な第三者が作成したデータとの区別が付く。

付録：クラウド/IoT時代のタイムスタンプ応用

➢ タイムスタンプサーバとのマッシュアップ

マッシュアップ：複数のサービスをAPIで接続して新たなサービスを提供する仕組み。
RFC 3161 over HTTP は、データこそASN.1/DER形式だが、APIとして利用可能なので、
RESTfulなAPIのように、他のクラウドサービスとのマッシュアップ利用が可能。
ハンズオンで見たようにタイムスタンプ取得だけなら簡単なプログラミングで可能。

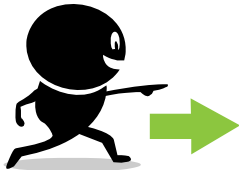


➢ システムログ等の改ざん防止

システムログの保管時にタイムスタンプを付与しておくことで、ログの改ざんができなくなるのでシステム監査等のコンプライアンス遵守に有効。

電子署名（PKI）ハンズオン

第2部：署名検証と長期署名



5. 電子署名の検証技術を理解する

6. 検証してみよう（ハンズオン3）
7. 長期署名を理解する
8. 長期署名を作ってみよう（ハンズオン4）

電子署名の検証には何が必要か？

1. 署名値（改ざん有無）の確認

○こっちは簡単！

- 数学的な処理なので環境依存はない（誰がやっても一致）
- 署名データ形式（XMLならスキーマ等）や属性等も確認

2. 署名者（証明書）の確認

×こっちが難しい！

- 誰が署名したか正しい署名者か確認する
- 環境に依存する処理なので正しい環境構築が必要
- 大きく分けて以下3項目を確認

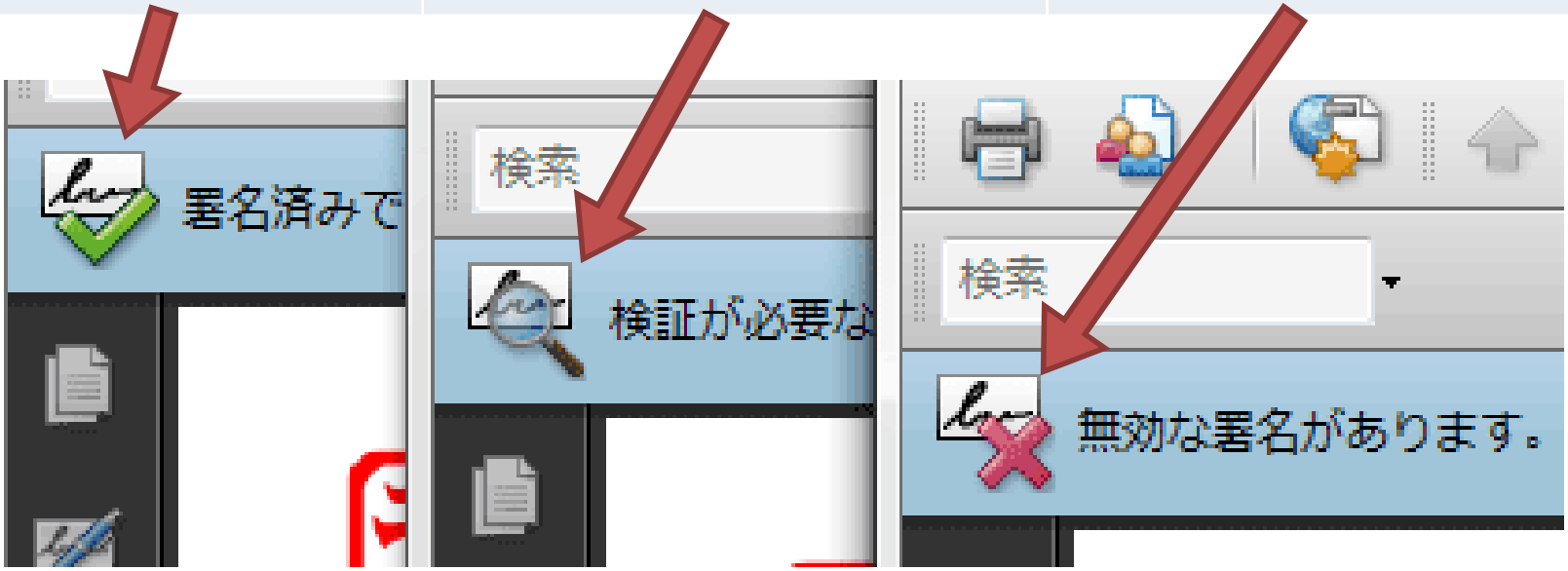
A) 認証パス（証明書チェーン）の構築

B) トラストアンカー（ルート証明書）の確認

C) 失効状況の確認（CRLとOCSP）

検証結果の種類

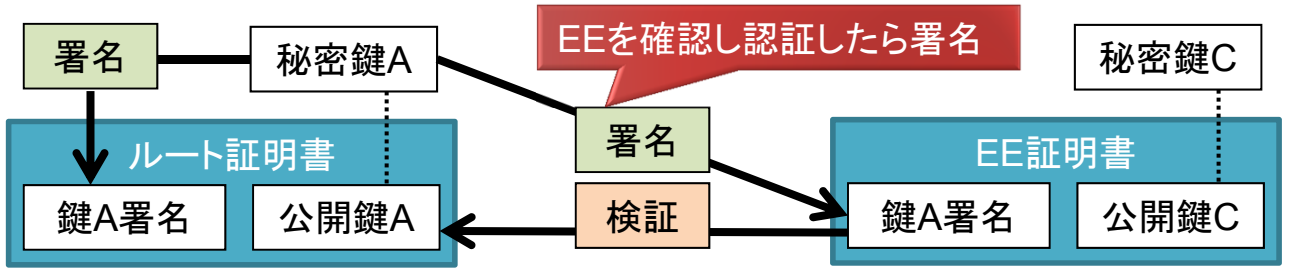
Verified (Valid)	Indeterminate (Unknown)	Failed (Invalid)
○ 検証正常	△ 検証不明	× 検証異常
全ての署名が正常である 暗号検証（改ざん無し） 証明書検証（有効）	暗号検証は正常（改ざん無し） 証明書の有効性が確認できない ※ 検証情報が不足している	暗号検証が異常（改ざんあり）か、 証明書が無効（失効等）だった ※ 明らかに異常な状態



Acrobat 9 における検証結果の各状態の表示

認証パス(証明書チェーン)の構築

- 証明書は公開鍵を含み、認証者から署名を受ける



1. 証明書の署名によるチェーンで認証パスを構成
2. 認証パスの最終点は署名者(EE:EndEntity)証明書
3. 認証パスの開始点は自己署名(オレオレ)証明書
4. 開始点と最終点の間にCA証明書がある場合もある

中間CA証明書がある認証パス例

トラストアンカー（ルート証明書）の確認

- 認証パスのルート証明書は自己署名（オレオレ）証明書
- 信頼されたルート証明書をトラストアンカーと呼ぶ
- 自己署名のルート証明書はどうすれば信頼できる？

1. 公開されている指紋（ハッシュ値）と自分で比較する
   - 認証局のリポジトリでは通常ルート証明書の指紋値を公開している
   - 面倒だし利用者が正しく確認すると言う前提が必要になる問題がある
2. アプリやOSが標準で信頼している証明書を信頼
   - Windows環境ならWindows証明書ストアの「信頼されたルート証明機関」
   - FireFoxやAdobe Reader等も独自に信頼済みルート証明書を持つ
   - 通常はWebTrustやETSIの認定を受けた認証局である必要がある
3. 証明書検証サーバ等が信頼している証明書を信頼
   - 例：日本のGPKIは政府内ネット内（非公開）の証明書検証サーバを利用

証明書の失効

証明書は様々な理由で有効期限内でも失効する

- 紛失や盗難による秘密鍵の漏えい

  クレジットカードと同じで不正利用される可能性があるのですぐに失効させる必要がある。

- 利用している暗号方式の危殆化
- 記載事項が変更された：住所や所属の変更等
- その他の理由で破棄や必要性が無くなった場合

失効情報を検証者に提供する方法は2つある

CRL　：失効した証明書情報一覧を提供

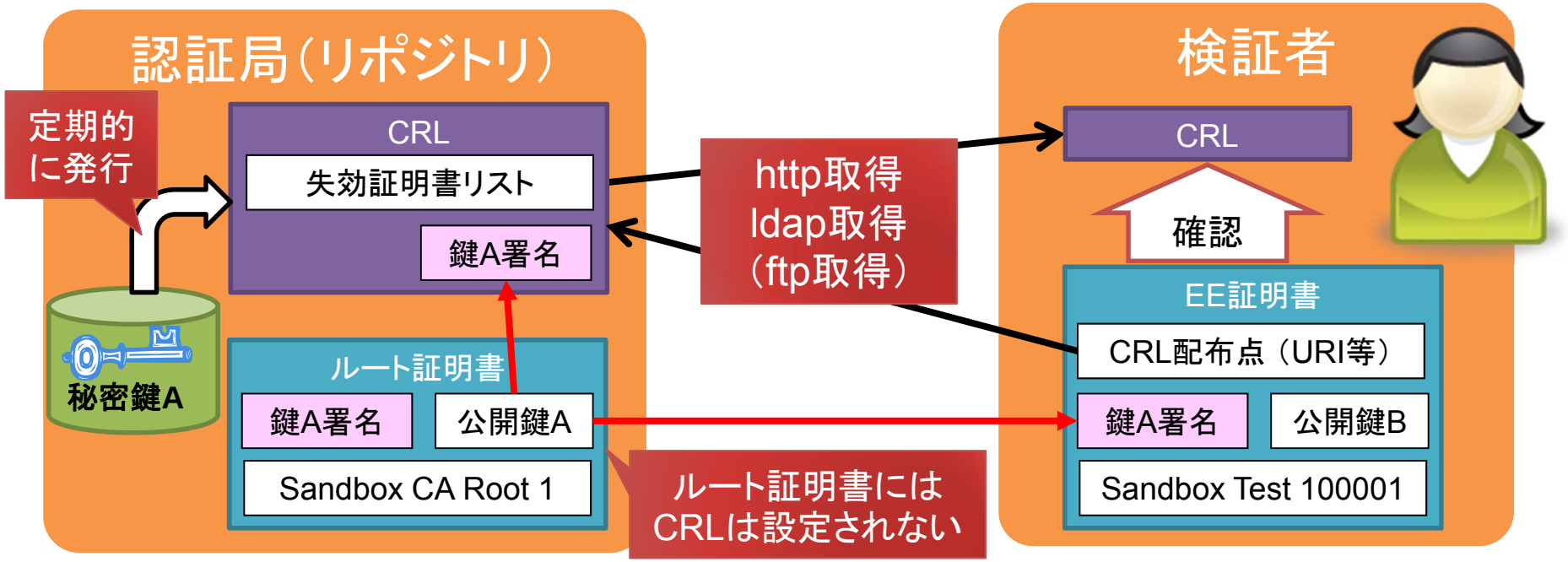
OCSP：検証対象の証明書の未失効を確認

CRL（失効情報リスト）RFC 5280

CRL（Certificate Revocation List）は、失効した証明書のリストを提供するASN.1/DER形式のファイル。

※ CRLは通常は証明書と同じ親証明書で署名する

➢ リストに検証する証明書が未記載なら有効！



CRL（失効情報リスト）の構造

CRLの発行周期は認証局運用規程（CPS）等で確認できる。ポリシによっては不定期の場合もあり。

これは一例かつ抜粋形式です。

ルート秘密鍵

X.509 CRL

signature	署名アルゴリズム識別子
issuer	発行者（通常CA）識別名
thisUpdate	CRL発行日
nextUpdate	次CRL発行予定日
CertificateSerialNumber	失効証明書シリアル番号
Time	失効日時
crlEntryExtensions	CRLエントリ拡張（失効理由等）
signatureAlgorithm	署名アルゴリズム
signature	署名値

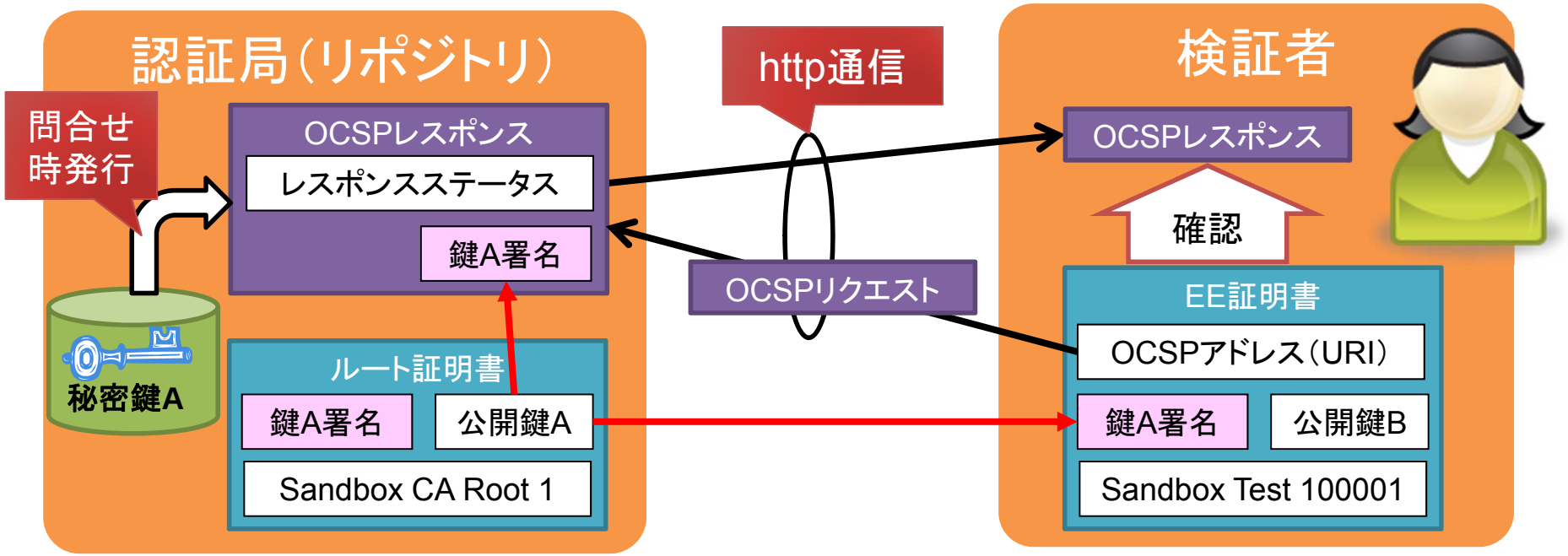
署名範囲

OCSP（オンライン証明書状態プロトコル）RFC 2560

OCSP（Online Certificate Status Protocol）は、問い合せた証明書のステータス（状態）を返すhttpプロトコル。

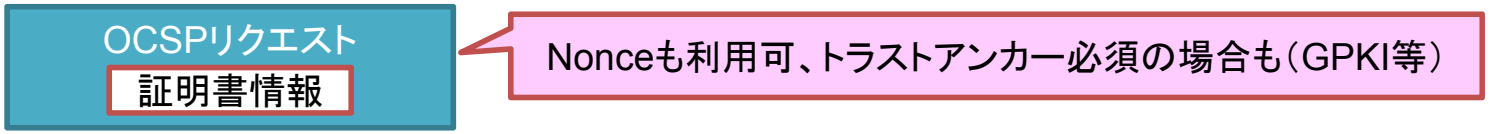
※ OCSPも通常は証明書と同じ親証明書で署名する

➢ 証明書の有効/無効のステータス情報を返す！

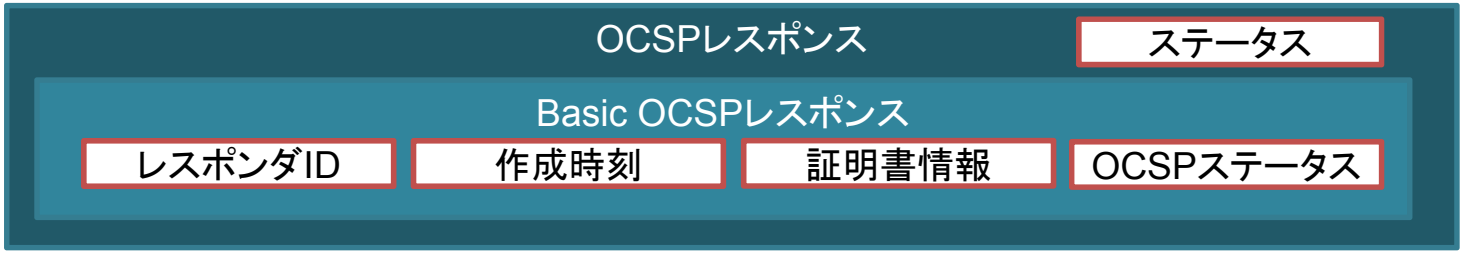


OCSPの取得手順

1. 証明書情報を埋め込んだOCSPリクエスト生成



2. OCSPリクエストをサーバに送信
3. サーバからOCSPレスポンスを取得



4. OCSPレスポンスのステータスを確認
5. BasicOCSPレスポンスのOCSPステータスを確認

※ 長期署名等では直接OCSPレスポンスを利用保管

CRLとOCSPの比較

	CRL	OCSP
CA・サーバの負荷	○軽い： 定期的に生成してファイルをWebサーバに置くだけ	×重い： APIの開発と提供が必要で、サーバ運用も必須
ファイルサイズ	×大きい（失効数に依存）： 数百Kバイトになる場合も	○小さい： 通常は常に一定
失効の反映と更新	通常定期的： CAのポリシ次第	DB直結はリアルタイム： ただしCRLベースが多く、その場合はCRLと同じになる
複数証明書	○同じ発行元の証明書は同じCRLを利用できる	×証明書毎に取得が必要 ※ キャッシュ化できない
その他	仕組みがシンプルなので、なんだかんだ言ってもよく使われている（デファクト）	商業登記証明書で利用、GPKI/LGPKIではOCSPを拡張し独自証明書検証サーバを提供、日本のCAでの利用は少ない

タイムスタンプ局(TSA)証明書

- タイムスタンプ局の署名証明書も検証する必要がある
  - A) 認証パス(証明書チェーン)の構築
  - B) トラストアンカー(ルート証明書)の確認
  - C) 失効状況の確認(CRLとOCSP)
- 証明書は通常用途(keyUsage)が指定されている

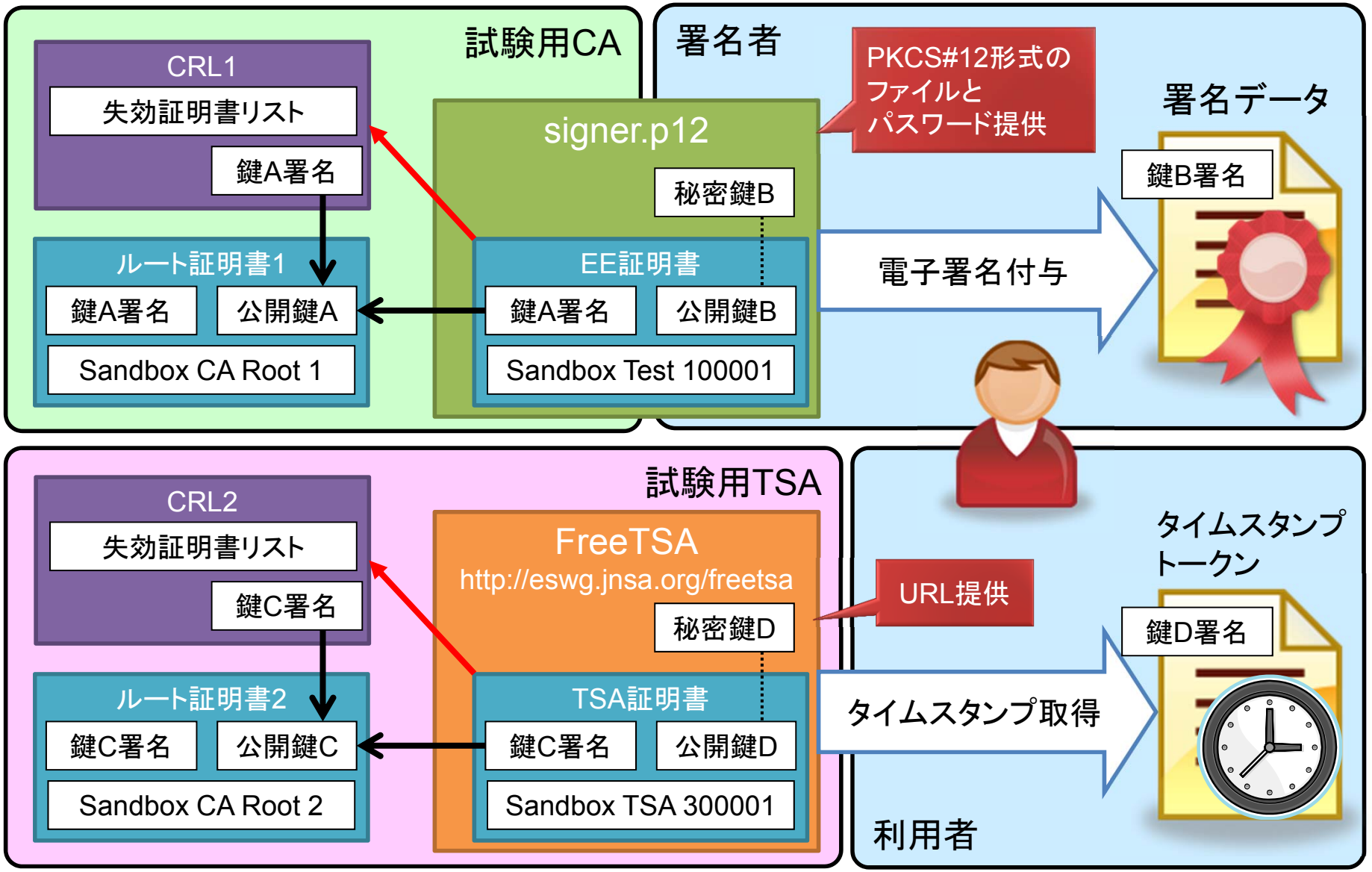
  digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment ...

  ※ 用途が指定されていたら不正用途で無いことを確認する
- タイムスタンプ局証明書はV3拡張用途を指定する

  extendedKeyUsage: timeStamping

  ※ タイムスタンプ署名証明書は拡張用途に timeStampling が必要
- 署名＋タイムスタンプは、2つの認証パス検証が必要

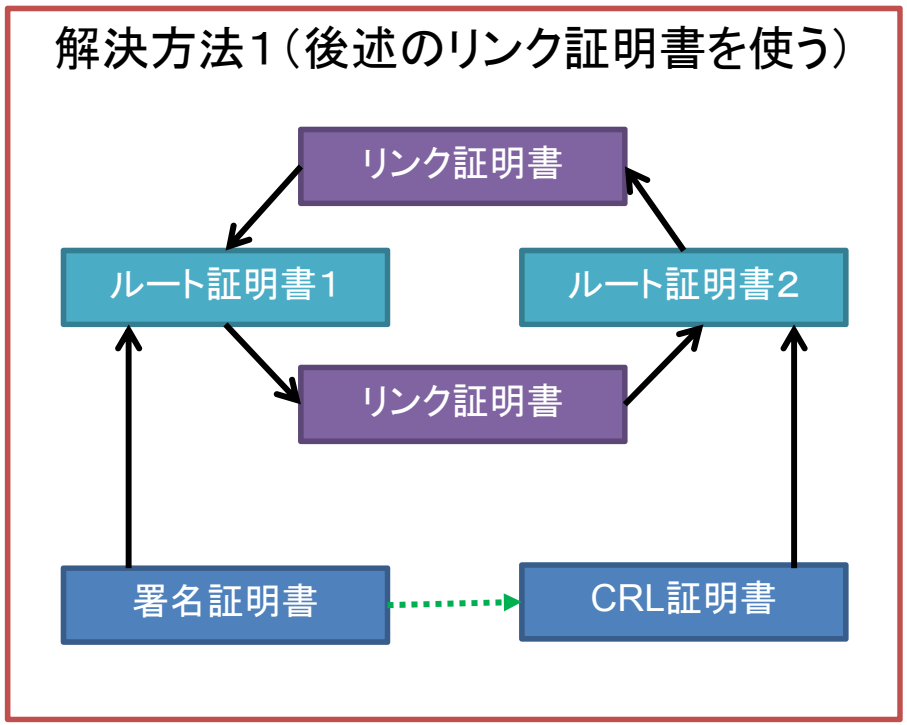
例：ハンズオン環境のCAとTSA

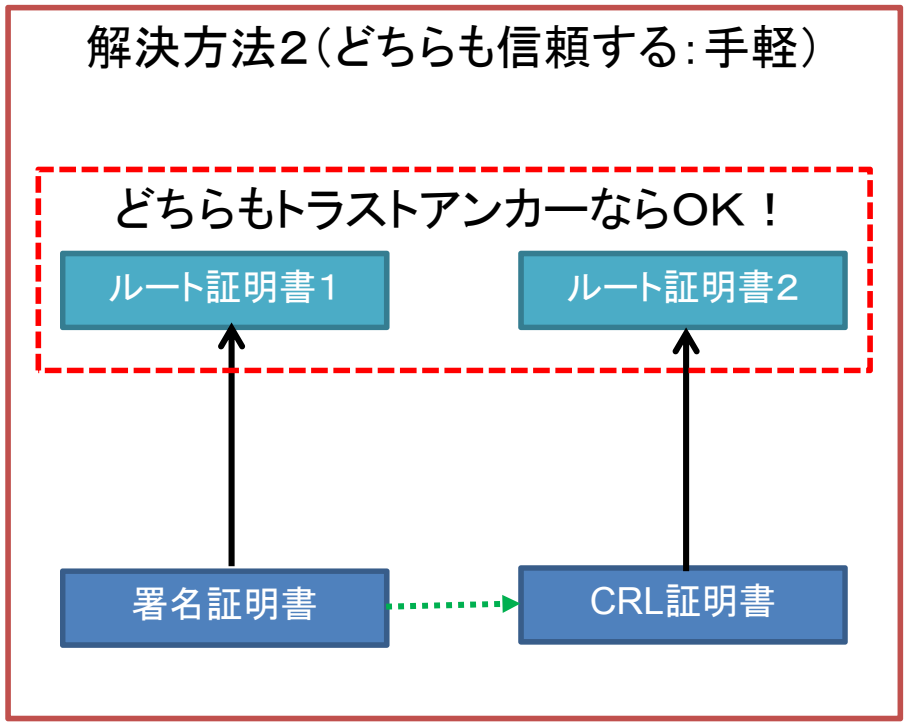


CRL(OCSP)と署名のルート証明書が違う

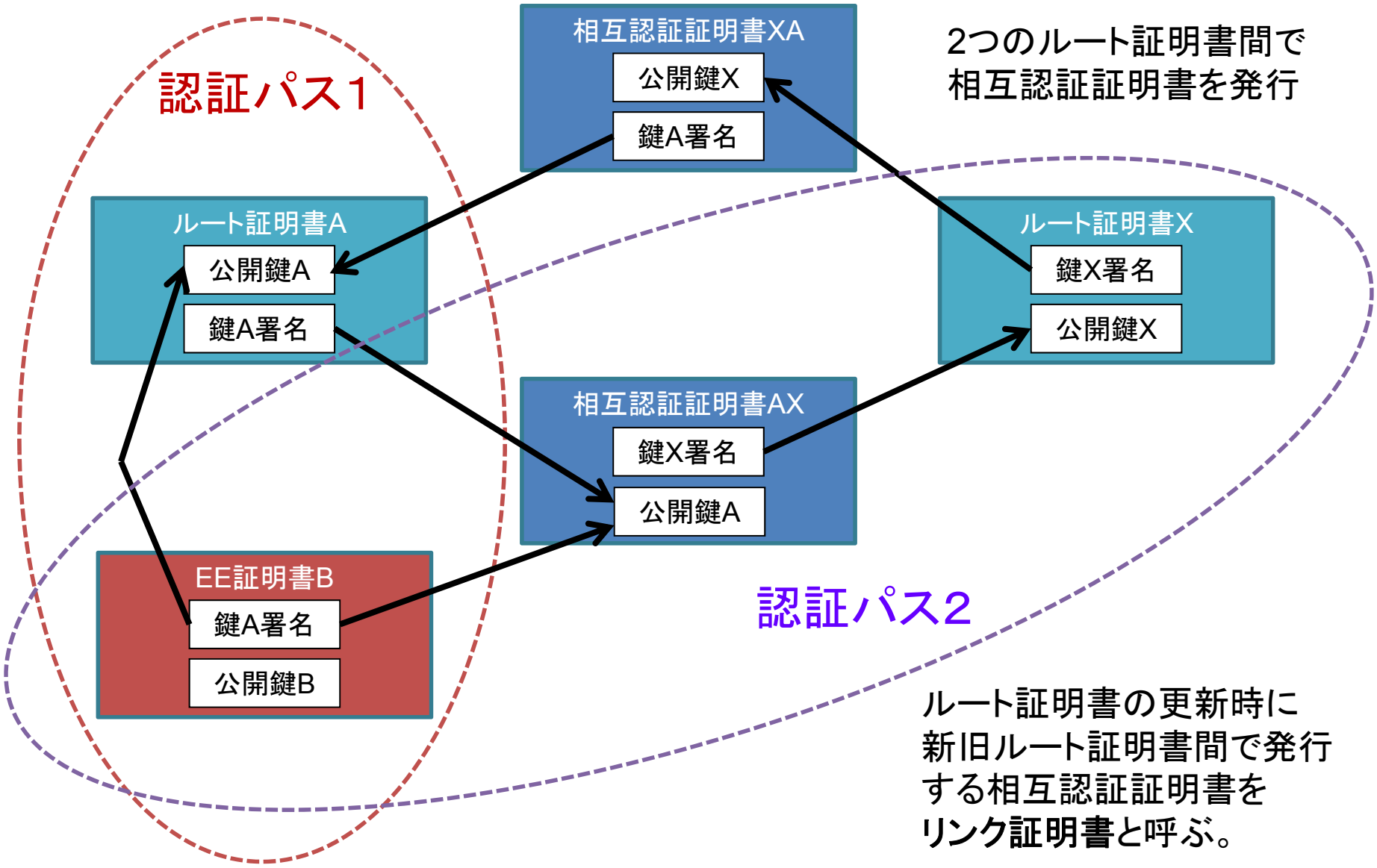
新しいルート証明書に切り替わり、新旧等複数のルート証明書が併存する場合に生じる場合あり。

- 商業登記証明書は毎年新ルート証明書を発行する為毎年この現象が…正直勘弁して欲しい。(※Acrobat未対応)





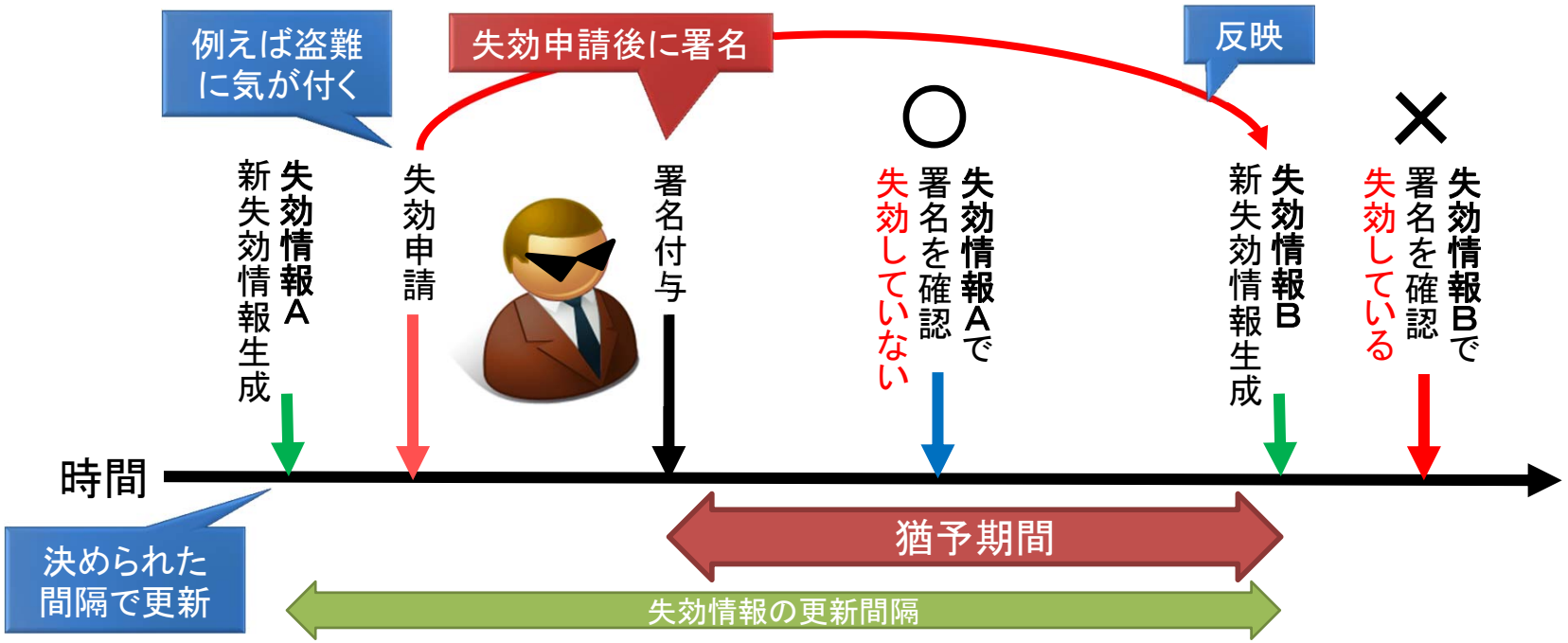
信頼点接続(相互認証)と2つの認証パス



失効の猶予期間（Grace Period）

- 失効情報は一定期間（数時間～数日）毎に更新される
  - 更新周期は認証局のポリシー（CPS）に記載されている。
- 失効申請をしても反映されるのは早くても次の更新時
- 署名時刻から失効更新までの期間を失効の猶予期間と呼ぶ

通常チェックする条件：署名時刻 ＜ 失効情報生成時刻



参考：公開鍵基盤（PKI：Public Key Infrastructure）

1. 公開鍵基盤（PKI）は、利用者の身元について「信頼できる第三者」（Trusted Third Party）が審査を行い、保証を実現する仕組みのことである。[by Wikipedia]
2. PKIは電子署名以外にSSL認証等で使われている。
3. ここまで勉強してきた、認証パス・失効確認・トラストアンカーは、SSL認証でも同じ仕組み。
4. トラストアンカーを管理している認証局（CA）に対して不正な要求を受け入れさせて偽造証明書を発行させるような攻撃が近年増加している。
5. ネットワークのセキュリティの基礎となる公開鍵基盤（PKI）も基本知識として勉強しよう。

参考：日本のPKI（公開鍵基盤）

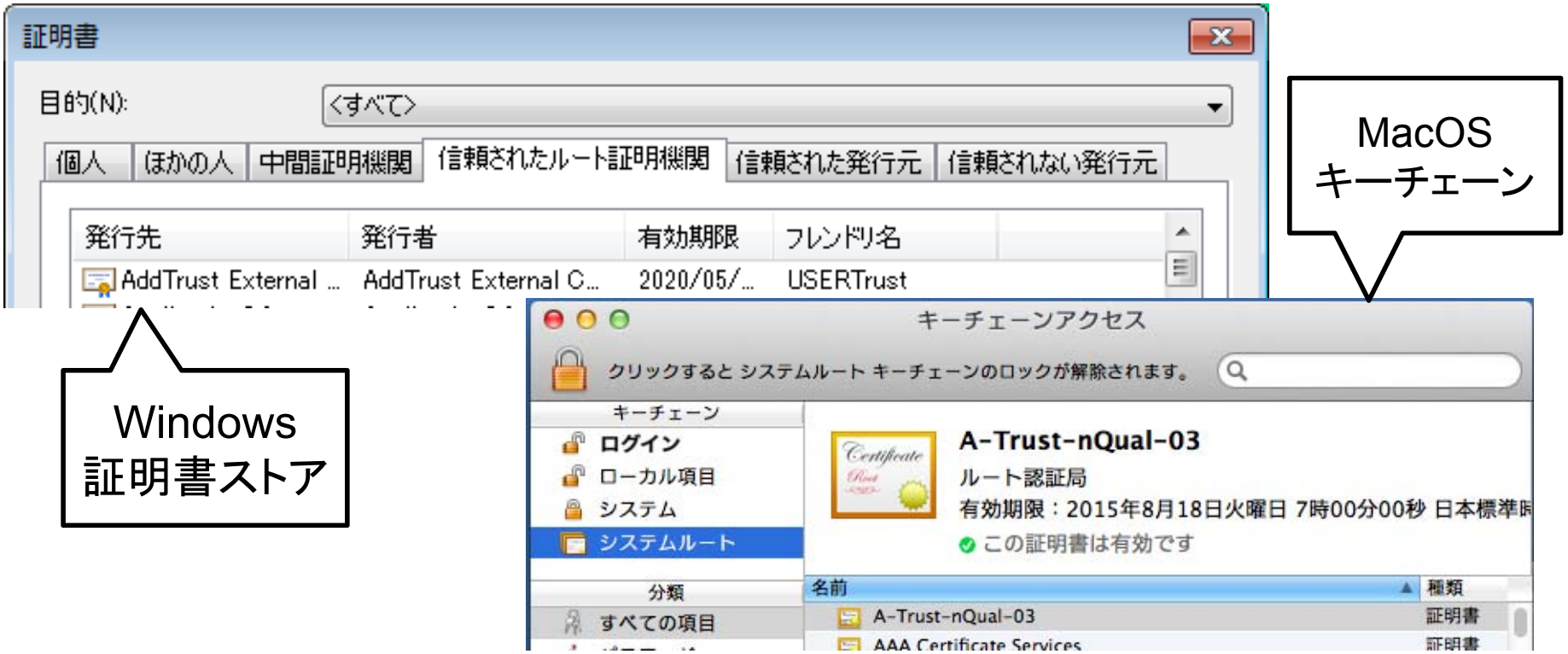
PKI		概要
GPKI	相互認証接続	日本政府PKI、官職証明書等
LGPKI		地方自治体PKI、自治体職員向け
JPKI		公的個人、住民向け（マイナンバー）
商業登記証明書		法務局、法人代表者向け
特定認証局		士業、民間個人向け（電子署名法）
HPKI		保険医療福祉、医者や薬剤師等
UPKI		大学学術、職員・教員・学生向け
WebTrust/ETI認定		OS/ブラウザ/アプリへの組み込み JCAN証明書やSymantec（VeriSign）

正確には
ブリッジCA
による接続

証明書ストア（信頼点や秘密鍵の管理）

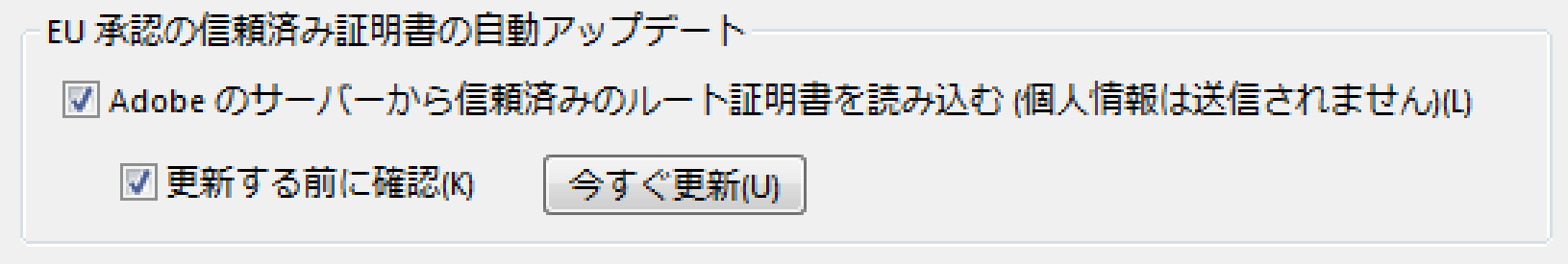
質問：ブラウザはどうやってSSL証明書を信頼する？

ブラウザは独自の証明書ストアを持ち、その中で信頼点（トラストアンカー）の管理を行っている。それぞれの開発元が認めた信頼点のみ登録されている。



多数の信頼点をリスト化（トラストリスト）

1. 信頼された組織が信頼点を管理してリスト化
2. トラストリストに含まれていれば信頼できる
3. 欧州（EU）では各国のトラストリストをまとめて管理
4. 最近Adobe Reader/Acrobatでも採用された



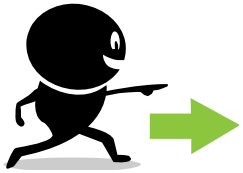
Adobe Reader の[環境設定]-[信頼性管理マネジャー画面]

5. MSやAdobeもEUのトラストリストと協調している
6. 日本でも同様の取り組みに期待したい

→ 最近少し動きがあるようです！期待しましょう！

電子署名（PKI）ハンズオン

第2部：署名検証と長期署名



5. 電子署名の検証技術を理解する
6. **検証してみよう（ハンズオン3）**
7. 長期署名を理解する
8. 長期署名を作ってみよう（ハンズオン4）

ハンズオン3：検証してみよう（概要）

a. OpenSSLを使ってCMS署名ファイルを検証する

※ ハンズオンディレクトリ OpenSSL の下で操作

3-1：期限切れと失効の証明書で署名（異常ケースの用意）

3-2：最新CRL取得とCAfileの作成（前準備）

3-3：CMS検証1（CRLをチェックしない）

3-4：CMS検証2（CRLをチェックする）

3-5：CRLの中を見てみよう

b. OpenSSLを使って証明書ファイルを検証する

※ ハンズオンディレクトリ OpenSSL の下で操作

3-6：CMSから証明書を取り出す

3-7：証明書の中を見てみよう

ハンズオン3-1：期限切れと失効の証明書で署名

手順1）CMS署名の実行

MacOSは ./openssl

```
> openssl cms -sign -in aaa.txt -out sign_rev.cms -signer revoked.pem -outform DER
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for test.pem:test2
> openssl cms -sign -in aaa.txt -out sign_exp.cms -signer expired.pem -outform DER
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Loading 'screen' into random state - done
Enter pass phrase for test.pem:test3
>
```

➢ 検証に利用するCMSファイル（署名データ）一覧

署名データ	署名証明書	ステータス	署名証明書	pswd
sign.cms	○ 有効	CRLを検証しても正常判定 失効もしていない	signer.pem	test1
sign_rev.cms	× 失効済み	CRLを検証すると異常判定 失効しているが有効期限内	revoked.pem	test2
sign_exp.cms	× 期限切れ	検証時刻から異常判定 notAfter: 2015-03-07	expired.pem	test3

ハンズオン3-2：最新CRL取得とCAfileの作成

手順2）CRL取得とCAfileの作成

CRL配布URLは証明書にCRLDPとして書かれている

```
> curl http://eswg.jnsa.org/sandbox/freeca/ca1.crl -o new.crl
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   434  100   434    0     0   3981      0 --:--:-- --:--:-- --:--:--  4617
> openssl crl -in new.crl -inform DER -out crl.pem -outform PEM
WARNING: can't open config file: /usr/local/ssl/openssl.cnf
>
```

最新CRLを取得

取得CRLをPEM形式に変換

MacOSは ./openssl

```
// Windowsの場合
> copy root1.pem ca1.pem
   1 個のファイルをコピーしました。
> type crl.pem >> ca1.pem
>
```

ルート証明書をca1.pemにコピー

CRLのPEMをca1.pemに追加

```
// MacOSの場合
$ cp root1.pem ca1.pem
$ cat crl.pem >> ca1.pem
$
```

ルート証明書をca1.pemにコピー

CRLのPEMをca1.pemに追加

- openssl crl 引数（CRLファイルの操作）
  - -in / -inform　：入力ファイルと、入力ファイルフォーマット（DER/PEM）の指定
  - -out / -outform　：出力ファイルと、出力ファイルフォーマット（DER/PEM）の指定

ハンズオン3-3：CMS検証（CRLをチェックしない）

手順3）CRLをチェックしないCMS署名の検証

MacOSは ./openssl

```
> openssl cms -verify -in sign.cms -inform DER -content aaa.txt -CAfile ca1.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
aaaVerification successful
>
> openssl cms -verify -in sign_rev.cms -inform DER -content aaa.txt -CAfile ca1.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
aaaVerification successful
>
> openssl cms -verify -in sign_exp.cms -inform DER -content aaa.txt -CAfile ca1.pem
Verification failure
4796:error:2E099064:CMS routines:CMS_SIGNERINFO_VERIFY_CERT:certificate verify e
rror:.¥crypto¥cms¥cms_smime.c:304:Verify error:certificate has expired
>
```

ハンズオン1でも試した通り正常

失効しているはずなのに正常！

証明書の期限切れ！

- openssl cms 引数（CMSファイルの操作）
  - -verify ：署名の検証操作を実行
  - -in ：入力（CMS署名ファイルを指定、**revo.cms** は失効証明書で署名済み）
  - -inform：入力形式（DERで出力したのでDER形式を指定）
  - -content ：署名対象（署名対象ファイルを指定）
  - -CAfile：ルート証明書他（事前に用意済みのPEM形式ルート証明書ファイルを指定）

ハンズオン3-4：CMS検証（CRLをチェックする）

手順4）正常なCMS署名のCRLも含めた検証

MacOSは ./openssl

```
> openssl cms -verify -in sign.cms -inform DER -content aaa.txt -CAfile ca1.pem -crl_check
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
aaaVerification successful
>
> openssl cms -verify -in sign_rev.cms -inform DER -content aaa.txt -CAfile ca1.pem -crl_check
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Verification failure
6232:error:2E099064:CMS routines:CMS_SIGNERINFO_VERIFY_CERT:certificate verify error:
.¥crypto¥cms¥cms_smime.c:304:Verify error:certificate revoked
>
> openssl cms -verify -in sign_exp.cms -inform DER -content aaa.txt -CAfile ca1.pem -crl_check
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Verification failure
4796:error:2E099064:CMS routines:CMS_SIGNERINFO_VERIFY_CERT:certificate verify error:
.¥crypto¥cms¥cms_smime.c:304:Verify error:certificate has expired
>
```

CRLをチェックしても正常！

-crl_check を追加

証明書失効のエラー！

証明書の期限切れ！

- openssl cms 引数（CMSファイルの操作）
  - -verify ：署名の検証操作を実行
  - -CAfile：ルート証明書他（ルート証明書と事前取得済みのCRLも含んでいる）
  - **-crl_check** ：CRLをチェックする（CRLはCAfile:ca1.pemに事前格納済み）

ハンズオン3-5：CRLの中を見てみよう

手順5）crl.pem中のCRLの確認

MacOSは ./openssl

```
> openssl crl -in crl.pem -text
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Certificate Revocation List (CRL):
        Version 1 (0x0)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: /C=JP/O=JNSA/OU=ESWG/CN=JNSA PKI Sandbox CA Root 1
        Last Update: Mar  8 19:56:01 2015 GMT
        Next Update: Mar 18 19:56:01 2015 GMT
Revoked Certificates:
    Serial Number: 100002
        Revocation Date: Feb 26 09:00:00 2015 GMT
    Signature Algorithm: sha256WithRSAEncryption
         95:77:49:f0:6b:3a:99:11:d4:e2:23:87:75:c1:c5:bf:f3:2c:
             （以下略）
>
```

CRL発行者名は証明書の発行者名と通常同じ

このCRLの発行日時と次の発行予定日時

失効証明書のシリアル番号と失効日時の情報！

- openssl crl 引数（CRLの操作）
  - -in　　：入力（CRLファイルを指定、crl.pem に格納、テキストエディタで見てみよう）
  - -text　：CRLの内容をテキスト表示する

ハンズオン3-6：CMSから証明書を取り出す

手順6）CMSファイルから署名証明書の取得

MacOSは ./openssl

```
> openssl cms -verify -in sign_rev.cms -inform DER -content aaa.txt -CAfile ca1.pem -certsout revo_cert.pem
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
aaaVerification successful
>
```

-certsout を追加

※ ここでは失効している証明書を取り出しています。

- openssl cms 引数（CMSファイルの操作）
  - -verify ：署名の検証操作を実行
  - -in ：入力（CMS署名ファイルを指定、**revo.cms** は失効証明書で署名済み）
  - -inform：入力形式（DERで出力したのでDER形式を指定）
  - -content ：署名対象（署名対象ファイルを指定）
  - -CAfile：ルート証明書他（ルート証明書と事前取得済みのCRLも含んでいる）
  - **-certsout** ：証明書を出力する（出力するPEMファイルを指定）

ハンズオン3-7：証明書の中を見てみよう

手順7）取り出した証明書の中を見てみよう

MacOSは ./openssl

```
> openssl x509 -in revo_cert.pem -text
WARNING: can't open config file: C:¥openssl-1.0.1g/ssl/openssl.cnf
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1048578 (0x100002)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=JP, O=JNSA, OU=ESWG, CN=JNSA PKI Sandbox CA Root 1
        Validity
            Not Before: Feb 25 06:50:14 2015 GMT
            Not After : Feb 22 06:50:14 2025 GMT
        Subject: C=JP, O=JNSA, OU=ESWG, CN=JNSA PKI Sandbox Test 100002
        Subject Public Key Info:
            （以下略）
>
```

証明書のシリアル番号

証明書の発行者名

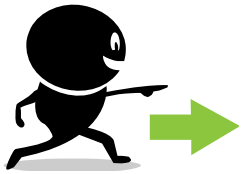
証明書の有効期限

- openssl x509 引数（X.509証明書の操作）
  - -in　　：入力（証明書ファイルを指定、テキストエディタで見てみよう）
  - -text　：証明書の内容をテキスト表示する

◆ 時間があれば正常/期限切れの証明書も表示してみよう。

電子署名（PKI）ハンズオン

第2部：署名検証と長期署名



5. 電子署名の検証技術を理解する
6. 検証してみよう（ハンズオン3）
7. **長期署名を理解する**
8. 長期署名を作ってみよう（ハンズオン4）

長期署名（有効期間を延長する）

電子署名済み

長期署名化

長期署名済み

① 検証情報の埋め込み

検証情報を長期保管する

証明書の有効期間後も検証可能

② アーカイブタイムスタンプ追加

最新の暗号方式でタイムスタンプを追加し保護する事で暗号危殆化に対応

電子署名（タイムスタンプ）のみ：

- 証明書の有効期間のみ有効
- 電子署名で数年、タイムスタンプで10年程度で良いなら、電子署名やタイムスタンプのみで良い
- 有効期限切れ時の扱いはグレー？

検証情報の埋め込み

- 検証情報とは？

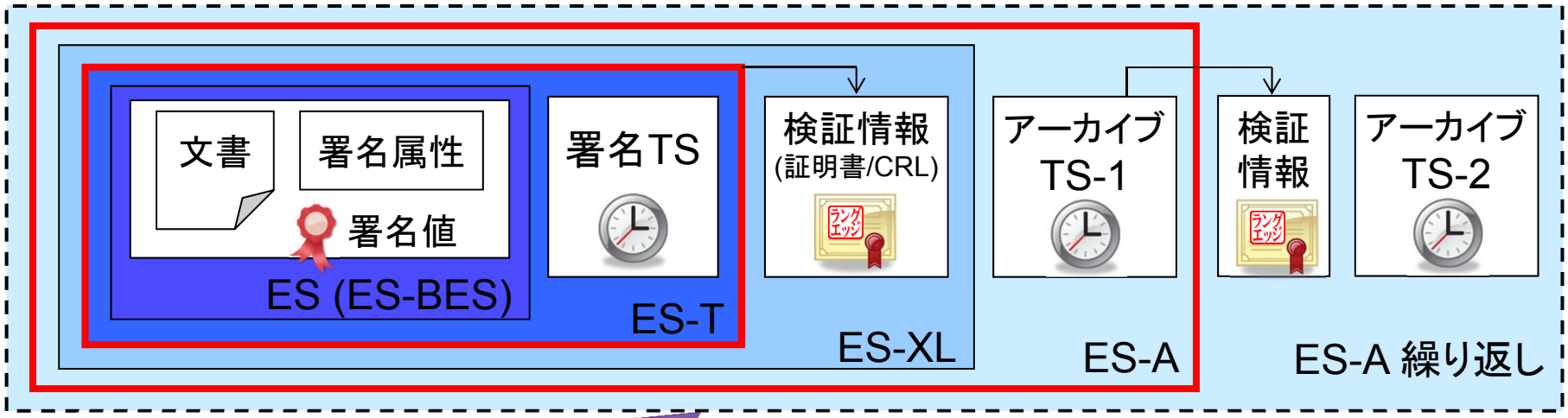
証明書	認証パスを構築する為に必要となる全ての証明書を埋め込む。 認証パスは署名証明書とTSA証明書の最低でも2つが必要となる。
失効情報	全証明書で必要となる失効情報として、CRLかOCSPレスポンスを埋め込む。

- アーカイブタイムスタンプを重ねる場合

  直前のアーカイブタイムスタンプのTSA証明書に関する検証情報を埋め込む必要がある。

長期署名フォーマット階層構造

ES (ES-BES/ES-EPES)	電子署名の付与（EPESはポリシ追加型）
ES-T	署名タイムスタンプの付与 - ISO/JIS定義
ES-X Long (ES-XL)	検証情報の埋め込み（認証パスと失効情報）
ES-A	アーカイブタイムスタンプの付与 - ISO/JIS定義

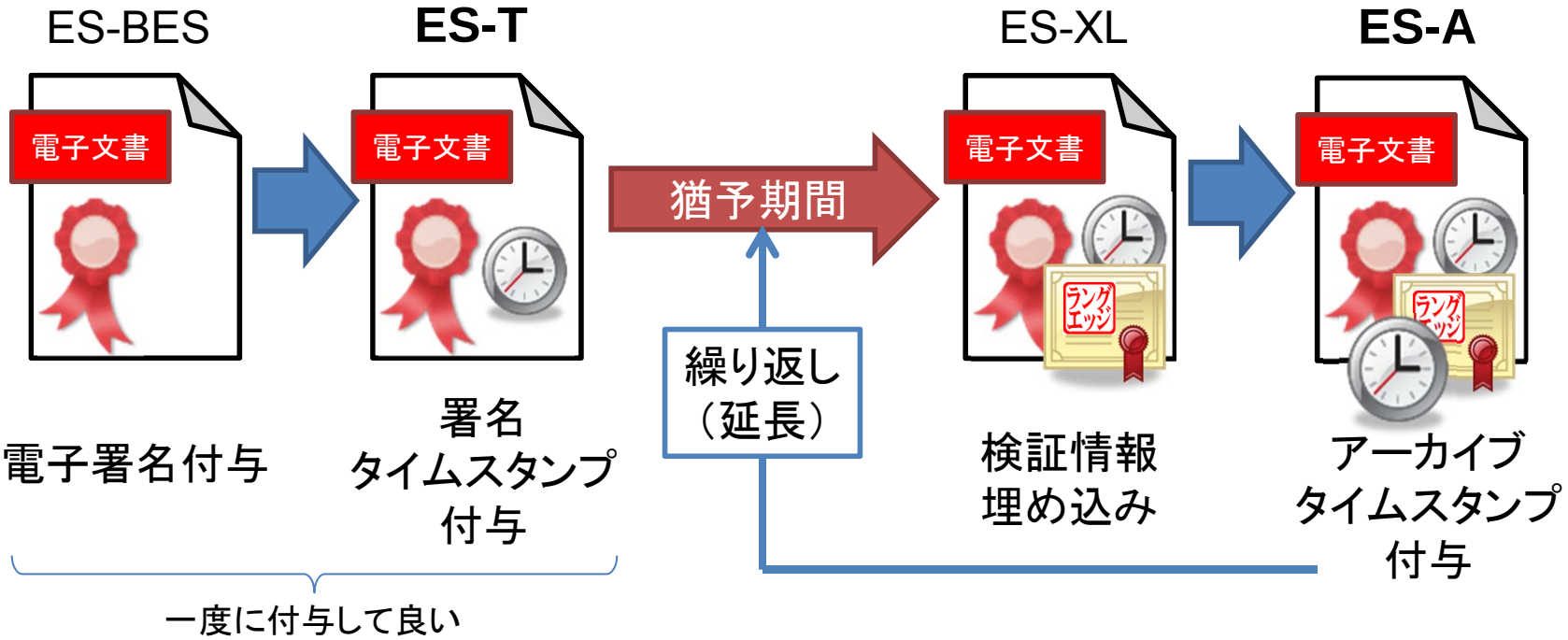


長期署名フォーマットは成長して行くフォーマットである（だんだん大きくなる）

長期署名の運用

※ 長期署名化の前に猶予期間（Grace Period）が必要

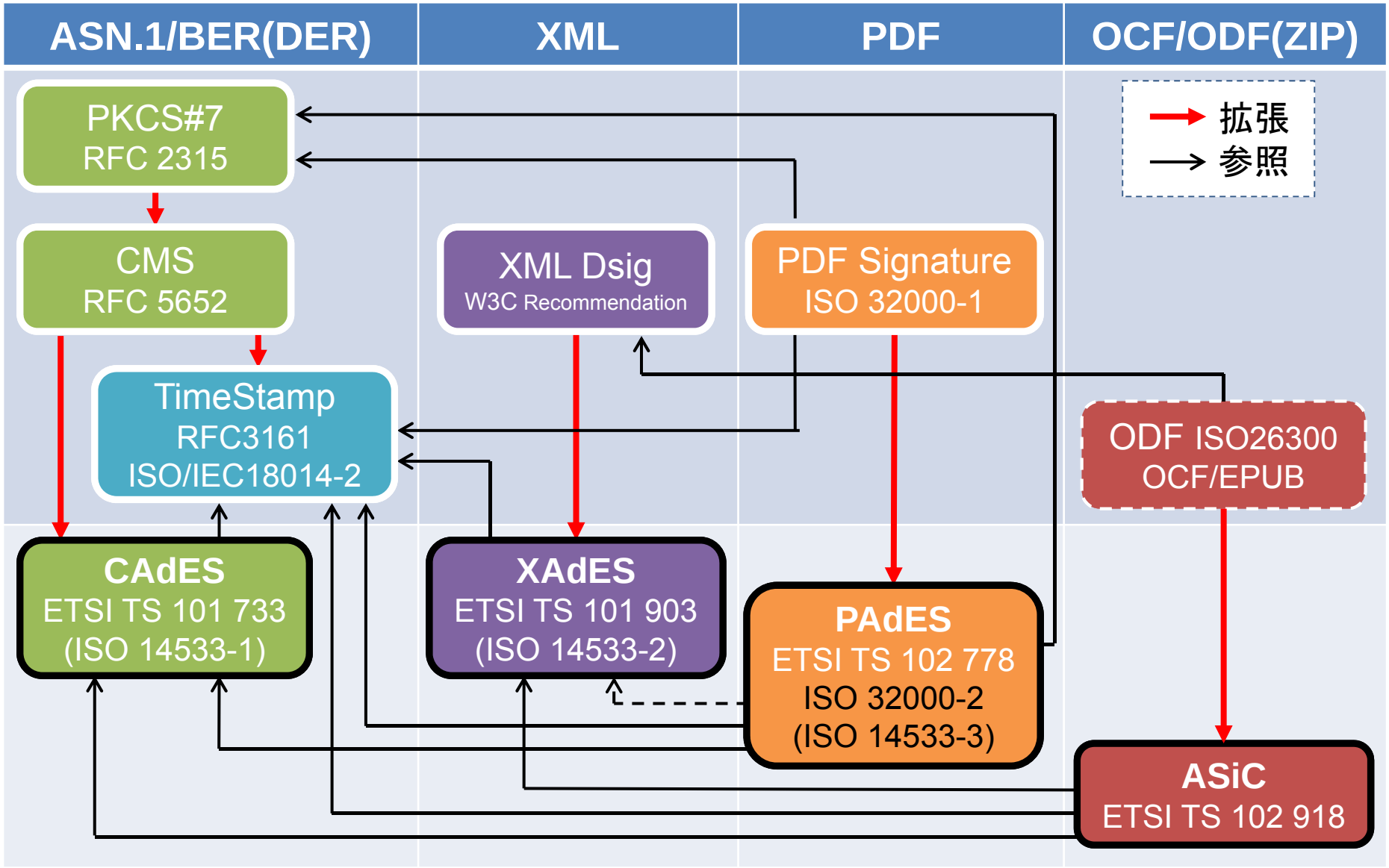
- 署名時点で失効していないことは通常は署名翌日に分かる！
  → 正確には認証局の運用規定の失効情報更新期間に依存する
- 長期署名化の検証情報の埋め込みは署名翌日以降にする
  → 長期署名化の運用フローを考える必要がある



長期署名フォーマット種類

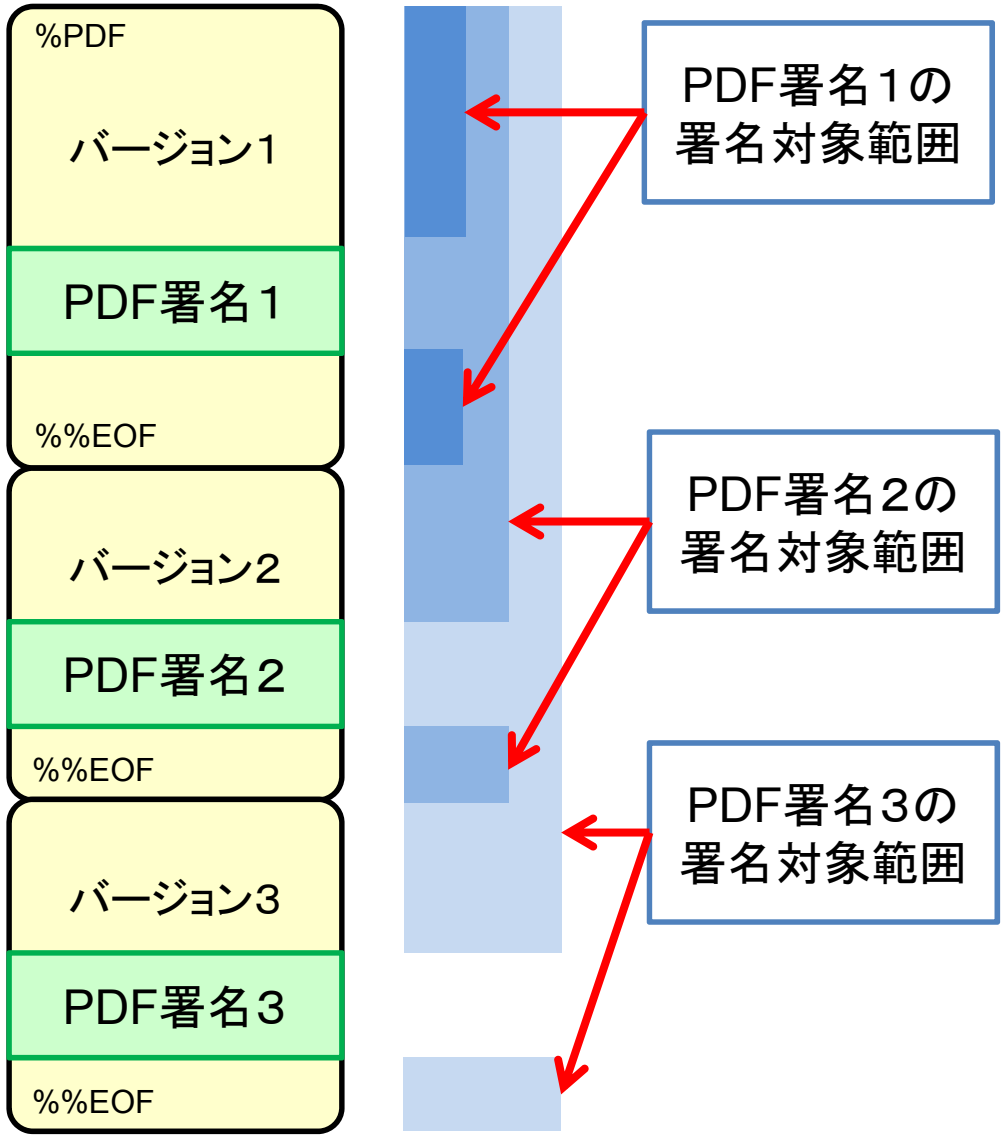
名称	ベース	署名対象	標準化
CAdES (2000年)	ASN.1 BER/DER	バイナリ他 ZIP化し複数指定可能	ETSI TS 101 733 (ETSI EN 319 122) JIS X5092 ISO14533-1
XAdES (2002年)	XML	URI指定可能データ 複数指定可能 （OOXML/ODFも採用）	ETSI TS 101 903 (ETSI EN 319 132) JIS X5093 ISO14533-2
PAdES (2009年)	PDF (CAdES)	PDFドキュメント 署名外観が利用可能	ETSI TS 102 778 (ETSI EN 319 142) 未JIS化 ISO32000-2予定 ISO14533-3予定
ASiC (2011年)	ZIP (XAdES/CAdES)	ZIPドキュメント内データ 複数指定可能	ETSI TS 102 918 (ETSI EN 319 162) 未JIS化 / 未ISO ETSIで仕様策定中

署名データフォーマットの系譜

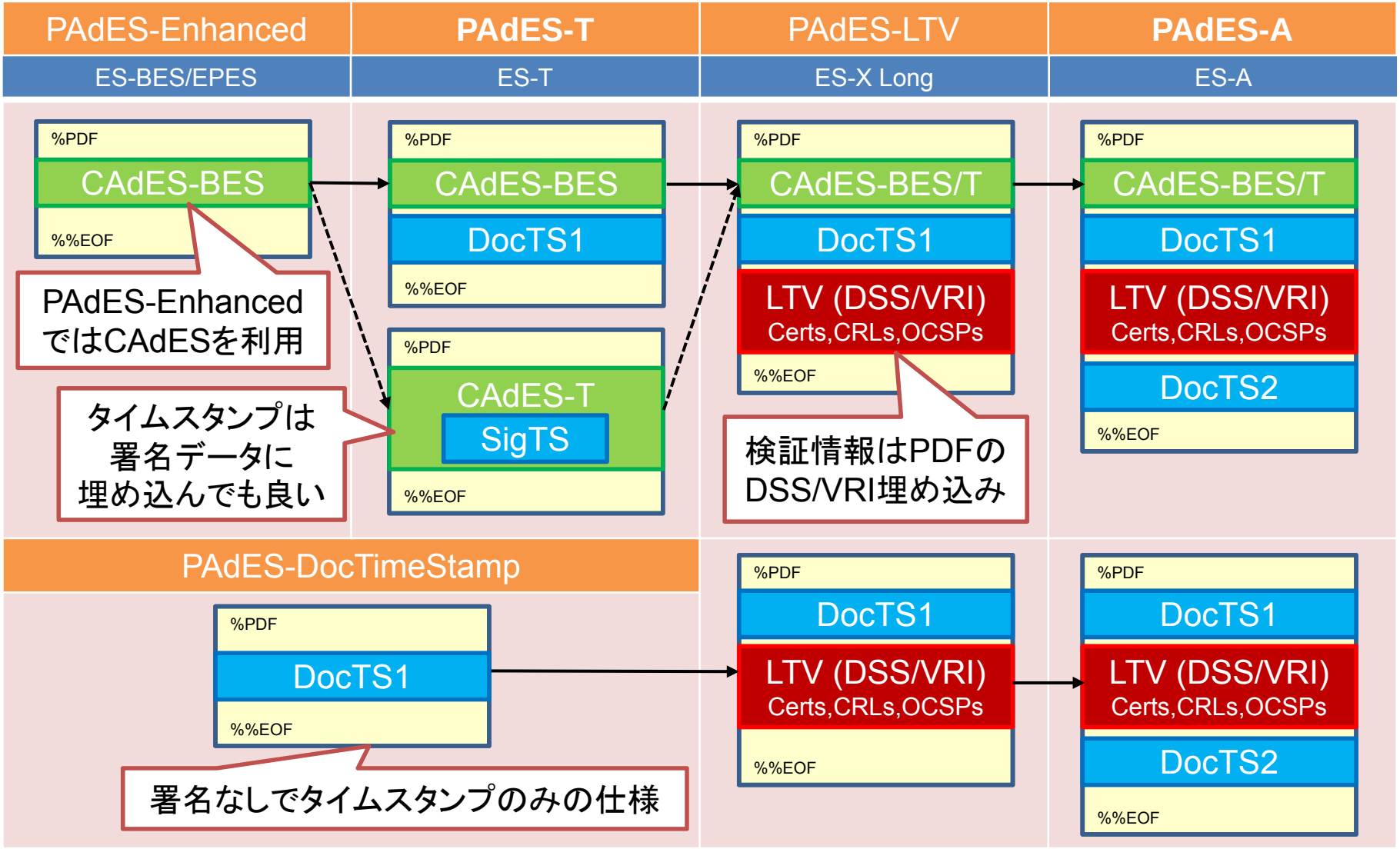


PDF署名 ISO 32000-1 / PAdES-Basic

- PDFの署名はPDFの増分更新と言う仕組みを使っている。
- ファイルの最後に新たに署名を追加して行くシリアル署名になる。
- PAdES以前及びPAdES-BasicのPDF署名は署名データにPKCS#7形式を利用。
- 署名対象範囲は署名毎に固定領域になる。



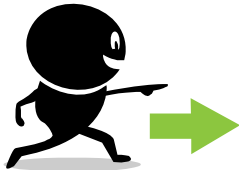
PAdES長期署名 ISO 32000-2 / ISO 14533-3



電子署名（PKI）ハンズオン

第2部：署名検証と長期署名

5. 電子署名の検証技術を理解する
6. 検証してみよう（ハンズオン3）
7. 長期署名を理解する



8. 長期署名を作ってみよう（ハンズオン4）

ハンズオン4：長期署名を作ってみよう（概要）

a. AdobeReaderを使ってPAdES-Tを作成/検証する

※ ハンズオンディレクトリ AdobeReaderの下のファイルを利用

4-1～3：前準備と利用する証明書/タイムスタンプのセット

4-4～5：AdobeReaderでES-T（PAdES-T）作成

4-6～11：AdobeReaderでES-T（PAdES-T）検証

b. AdobeReaderを使ってPAdES-Aを作成/検証する

4-12：AdobeReaderでES-XL（PAdES-LTV※）作成

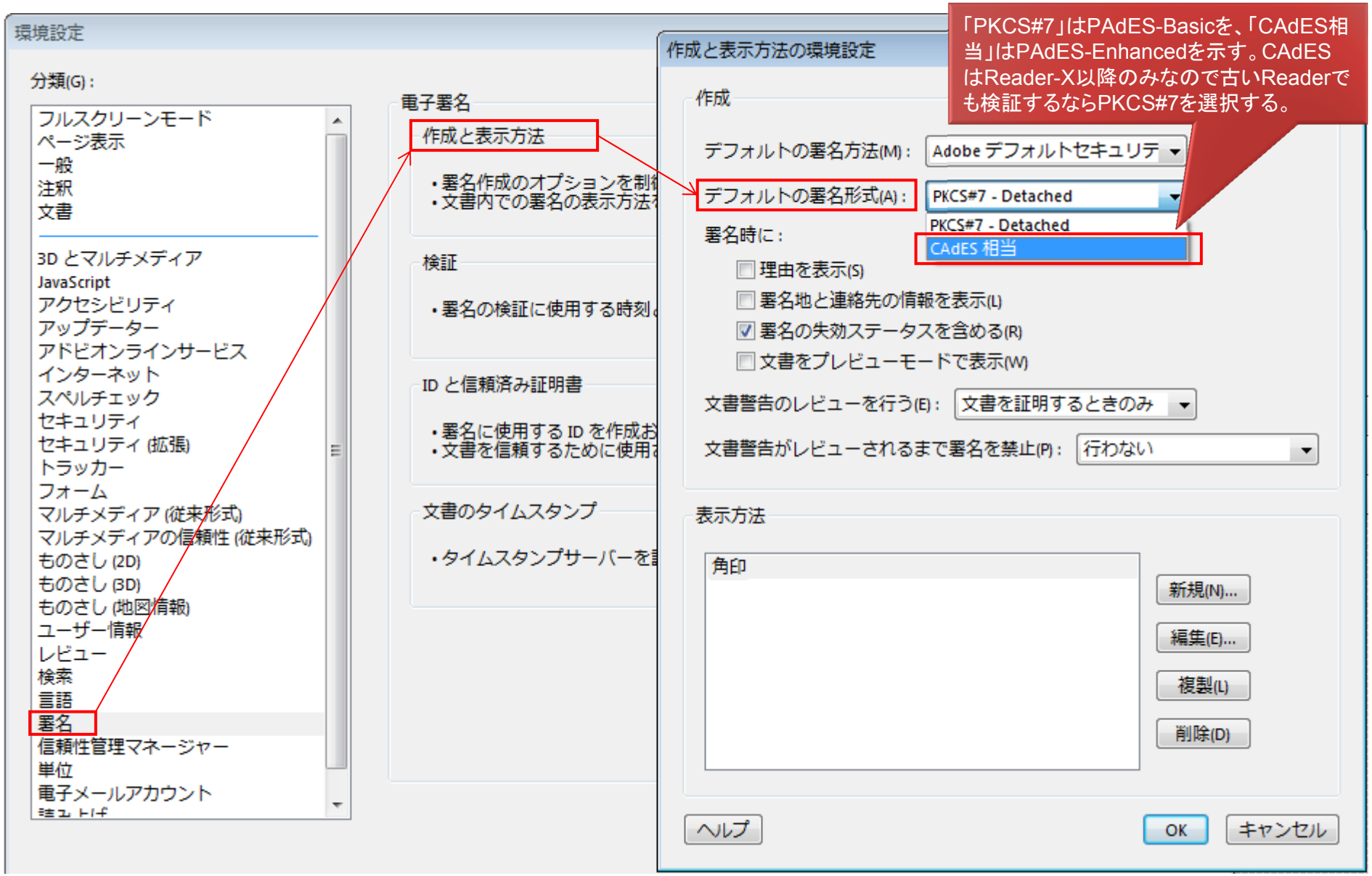
※ PAdESではES-XLをPAdES-LTVと呼ぶ

4-13：AdobeReaderでES-A（PAdES-A）作成

4-14：AdobeReaderでES-A（PAdES-A）検証

- 後片付け：トラストアンカーを削除する（とても重要!）

ハンズオン4-1：Adobe Readerの環境設定



ハンズオン4-2：利用する証明書のセット

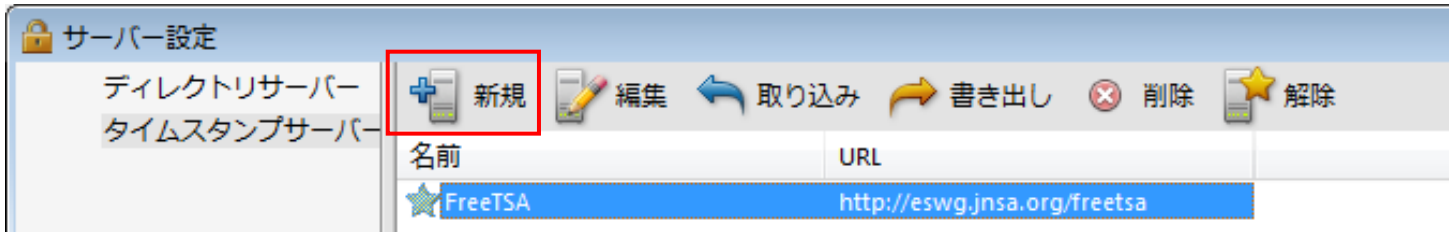
➢ Adobe ReaderにPKCS#12ファイルをセットする

1. [環境設定]-[署名]-[IDと信頼済み証明書]-[詳細]
2. [IDを追加]-[ファイル]-[次へ]
3. signer.p12 と パスワード ”test1” を指定して [次へ]
4. 確認画面が出るので[完了]

ハンズオン4-3：利用するタイムスタンプのセット

➢ Adobe Readerにタイムスタンプサーバをセットする

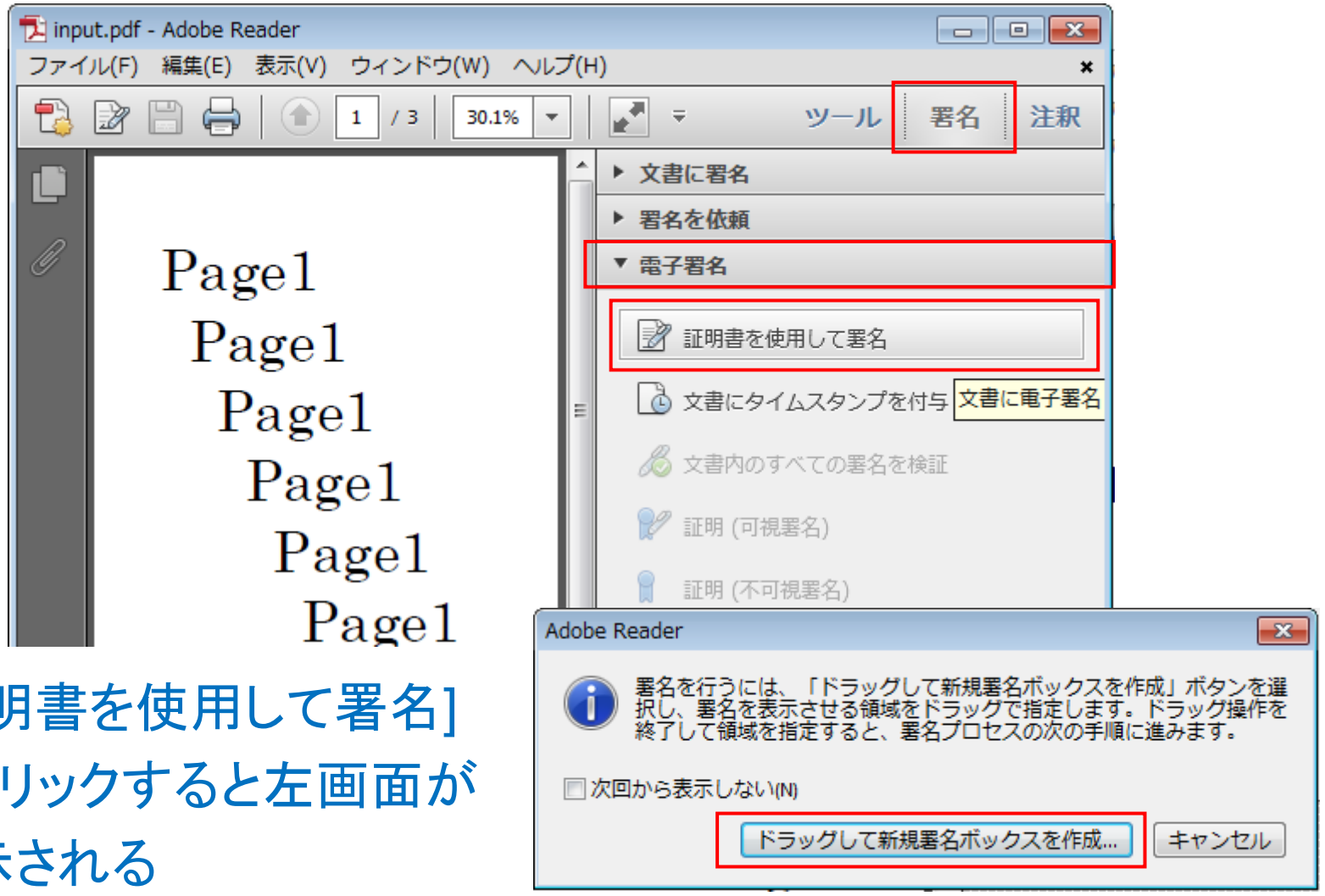
1. [環境設定]-[署名]-[文書のタイムスタンプ]-[詳細]
2. [新規] で任意の名前（例：FreeTSA）を入力
3. URL http://eswg.jnsa.org/freetsa を指定して [OK]
4. [デフォルトに設定] する　確認画面が出たら [OK]



※ これで署名時には自動的に署名タイムスタンプが付きます。

ハンズオン4-4：Adobe Readerで ES-T 作成1

1. PDFファイルを開き、ツールバー[署名]タブの[電子署名]を開く

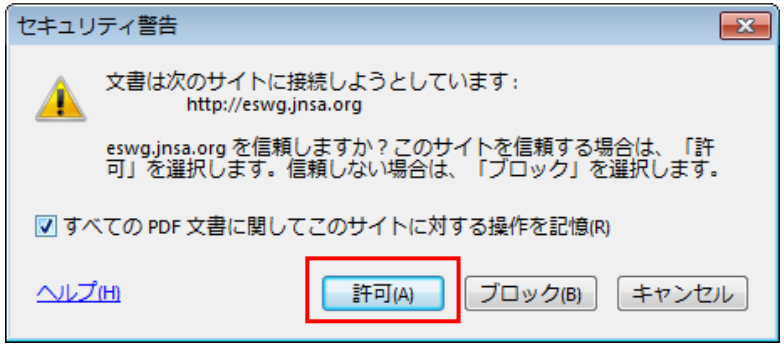


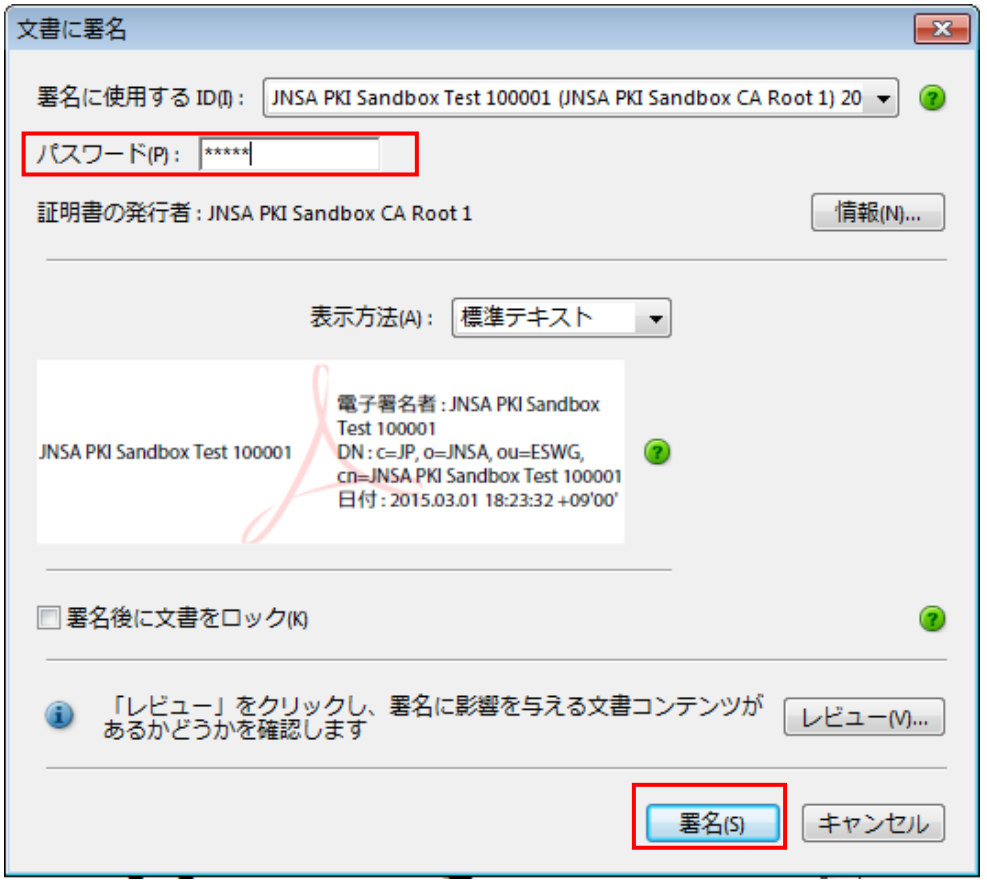
2. [証明書を使用して署名]をクリックすると左画面が表示される

ハンズオン4-5：Adobe Readerで ES-T 作成2

3. [ドラッグして新規署名ボックスを作成…]をクリックし範囲指定
4. [文書に署名]画面で[JNSA PKI Sandbox Test 100001]を選択
   パスワードを 指定し（パスワード: ”test1” ）、[署名]ボタンをクリック
5. [名前を付けて保存]と
   なるので任意の名前で
   保存（例：”signed.pdf”）

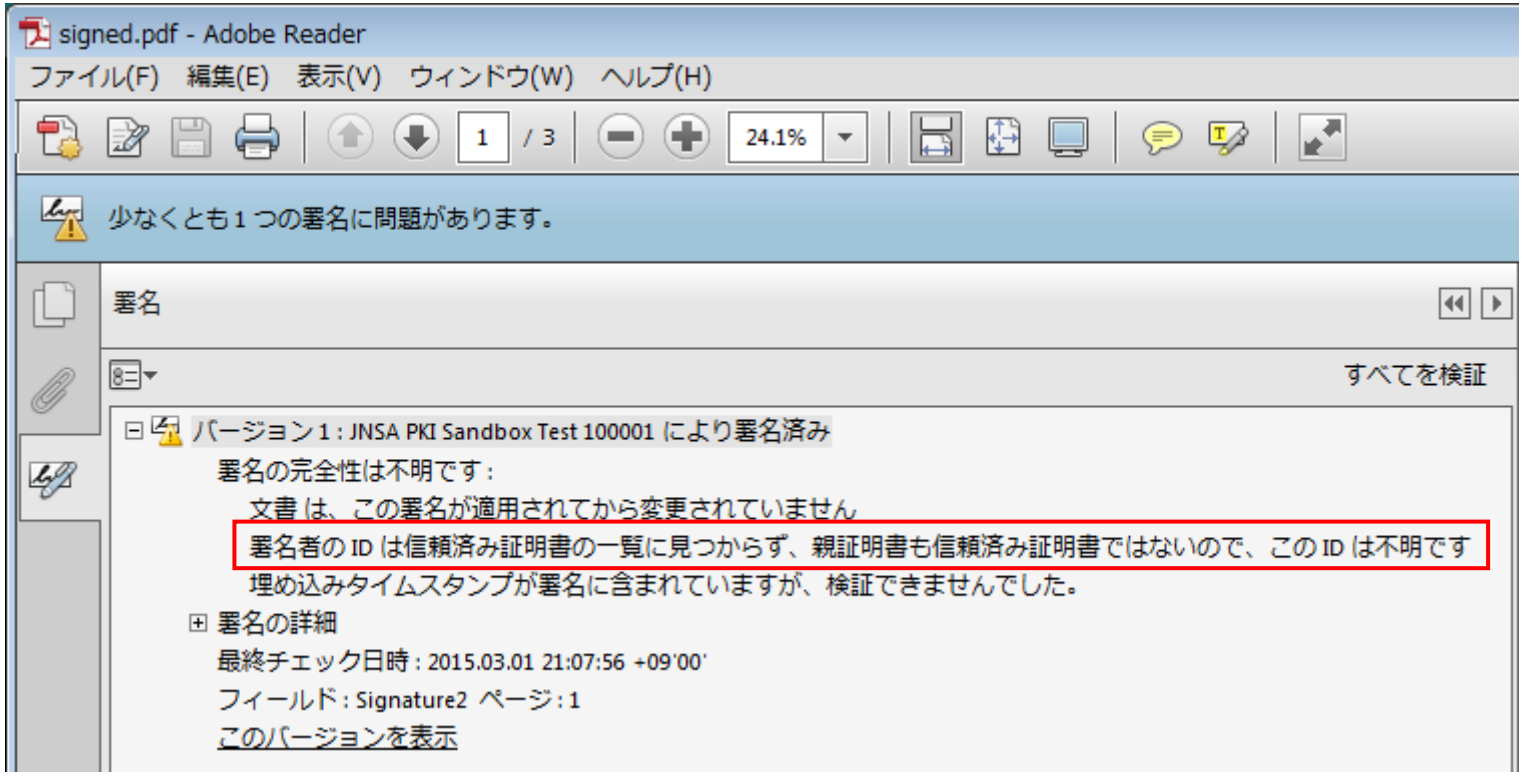
※ セキュリティ警告が
出たら許可する。





ハンズオン4-6：Adobe Readerで ES-T 検証

6. 署名バーに[少なくとも1つの署名に問題があります。]と表示がされるので[署名パネル]をクリックして開く

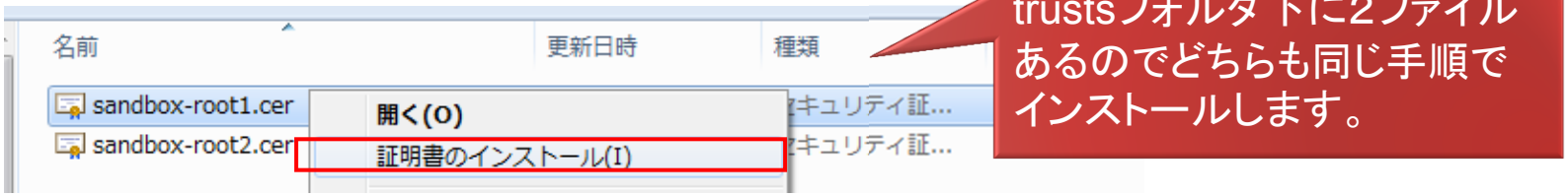


- エラーになっているのはトラストアンカーが無いか、信頼されていない為。

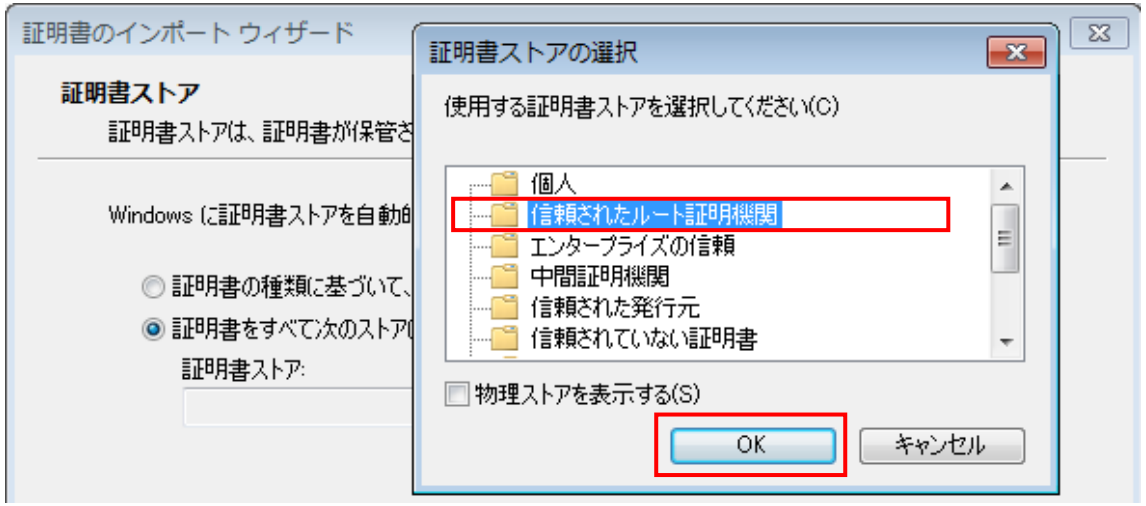
ハンズオン4-7：トラストアンカーWindows編1

- アドビ独自の証明書ストアもありますが、まずここではWindows証明書ストアへトラストアンカーをインストールしてみます。

1. Windowsの場合にはエクスプローラから右クリックで[証明書のインストール]を実行する。

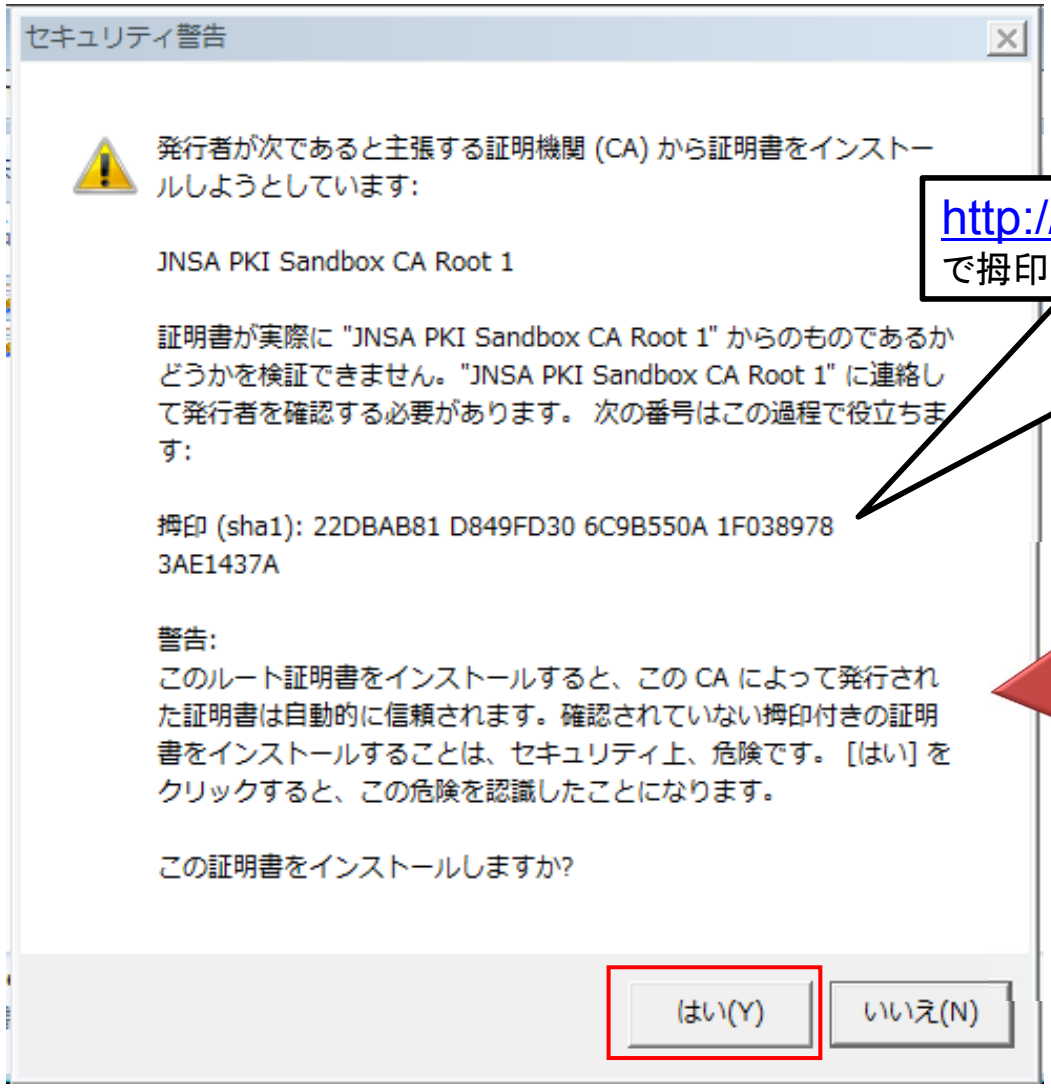


2. 証明書ストアでは[証明書をすべて次のストアに配置する]を指定して[信頼されたルート証明機関]を選択



ハンズオン4-8：トラストアンカーWindows編2

3.　セキュリティ警告が表示されるので確認して[はい]をクリック。



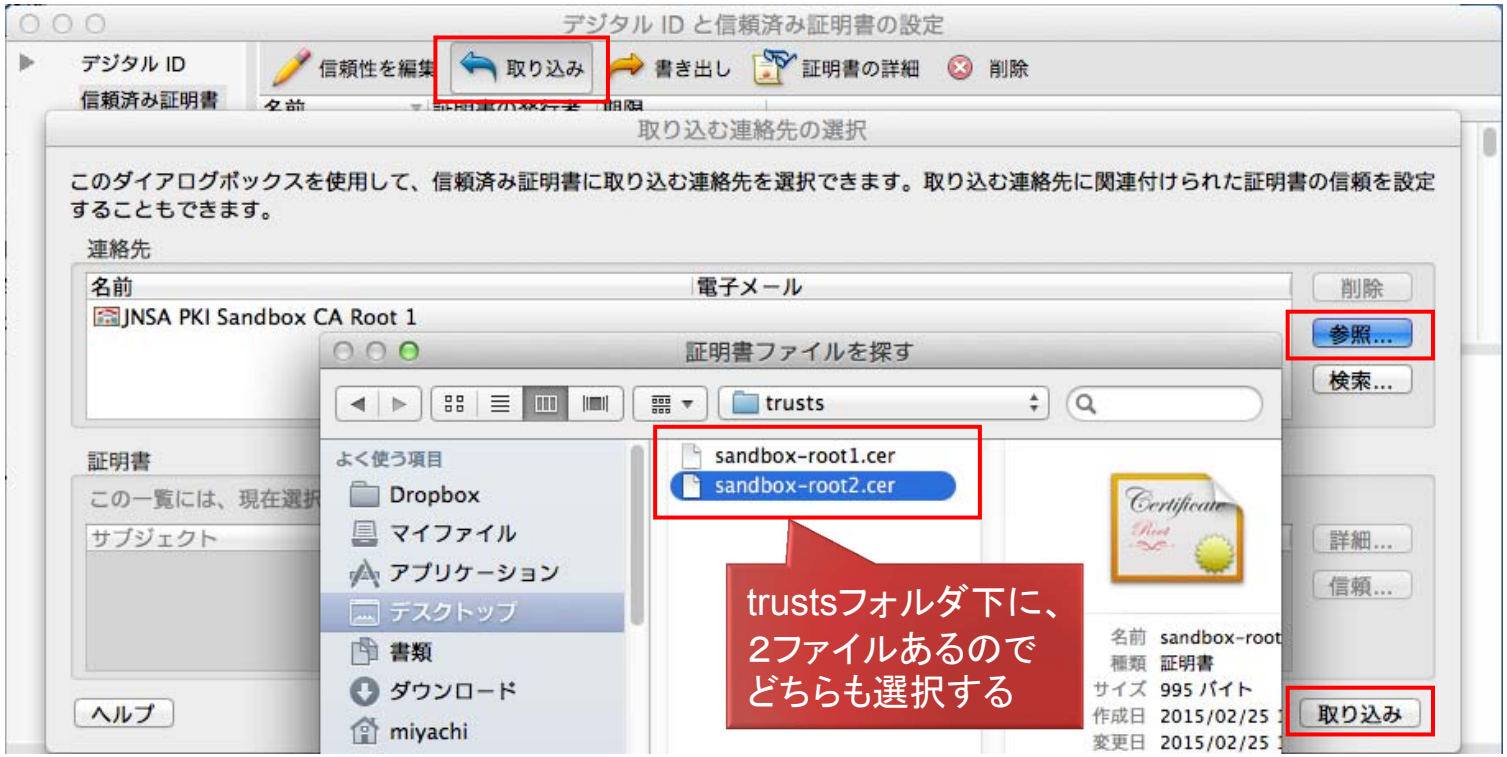
http://eswg.jnsa.org/sandbox/freeca/
で拇印は確認可能（通常、CAのリポジトリで公開）

ここにも書かれていますがルート証明書をインストールすると、このルートから発行された証明書が自動的に信頼されるのでとても危険です。今回は試験の為にインストールしますがこの警告が出た場合には注意が必要です。

ハンズオン4-9：トラストアンカーAdobe編1

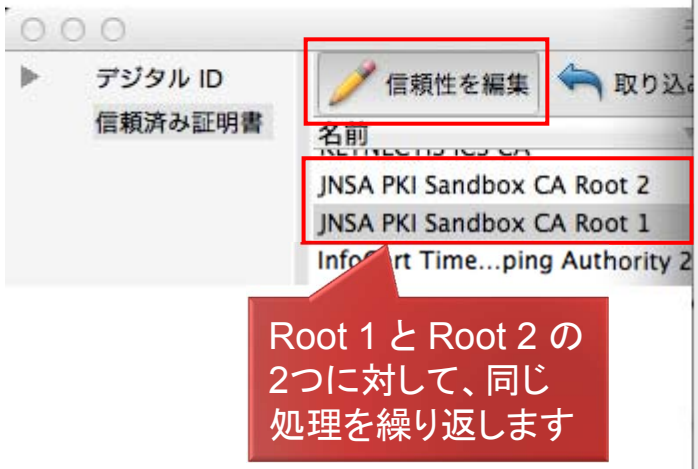
- アドビ独自の証明書ストアへトラストアンカーをインストールする。MacOSの場合はこちらの作業が必要です（Windowsでも可能）。

1. [環境設定][署名][IDと信頼済み証明書]から[信頼済み証明書]を選択して[取り込み]を開き[参照…]をクリック
2. ルート証明書2ファイルを指定して[取り込み]ボタンで取り込む

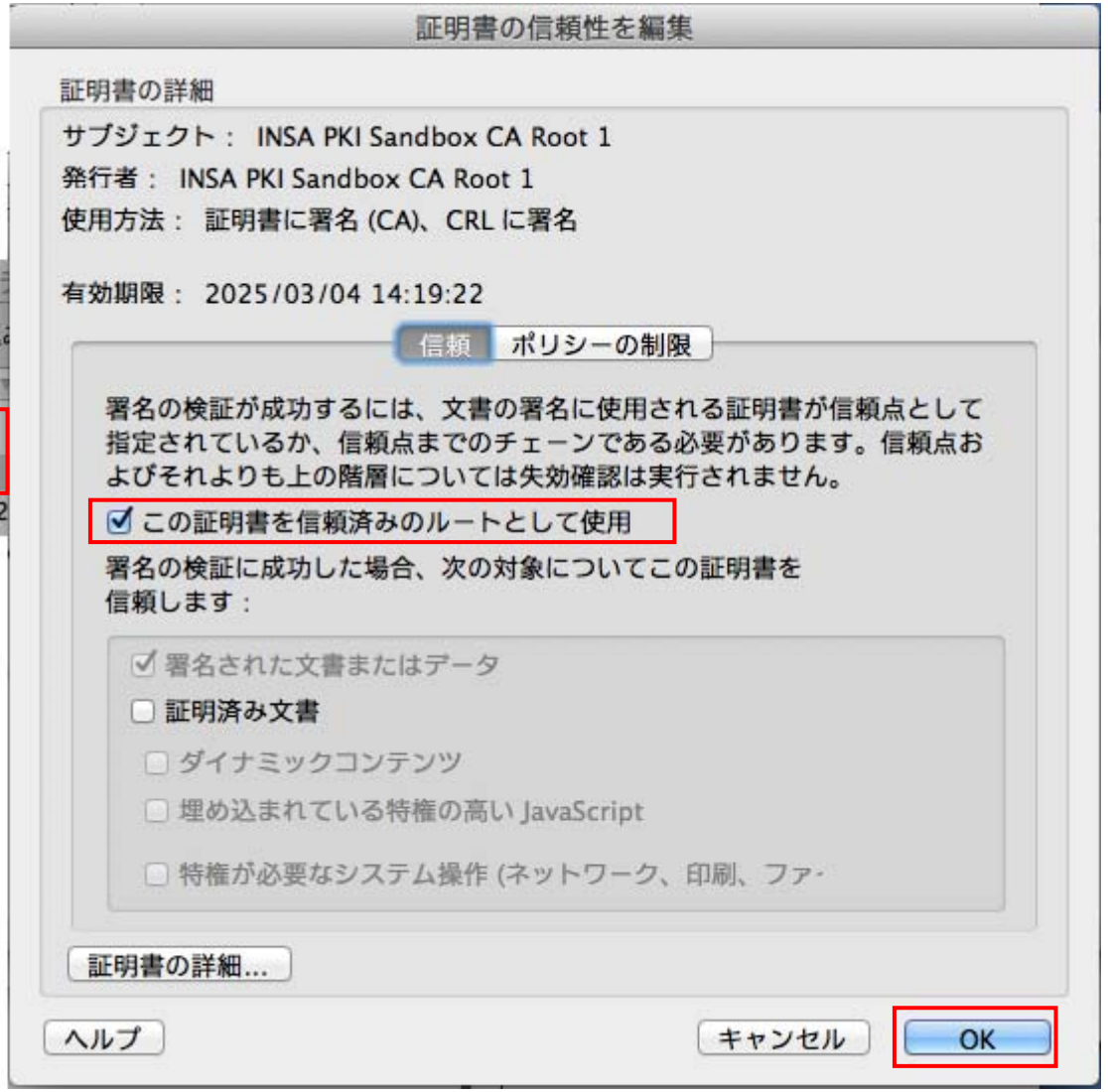


ハンズオン4-10：トラストアンカーAdobe編2

3. 取り込んだ証明書を選択し[信頼性を編集]をクリック



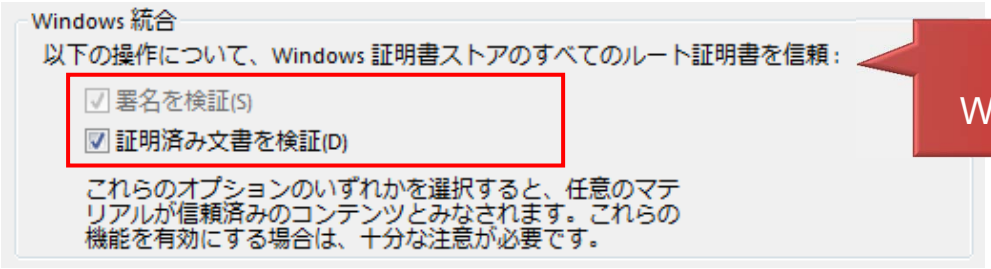
3. [この証明書を信頼済みのルートとして使用]をオンにして[OK]をクリック



ハンズオン4-11：Adobe Readerで ES-T 再検証

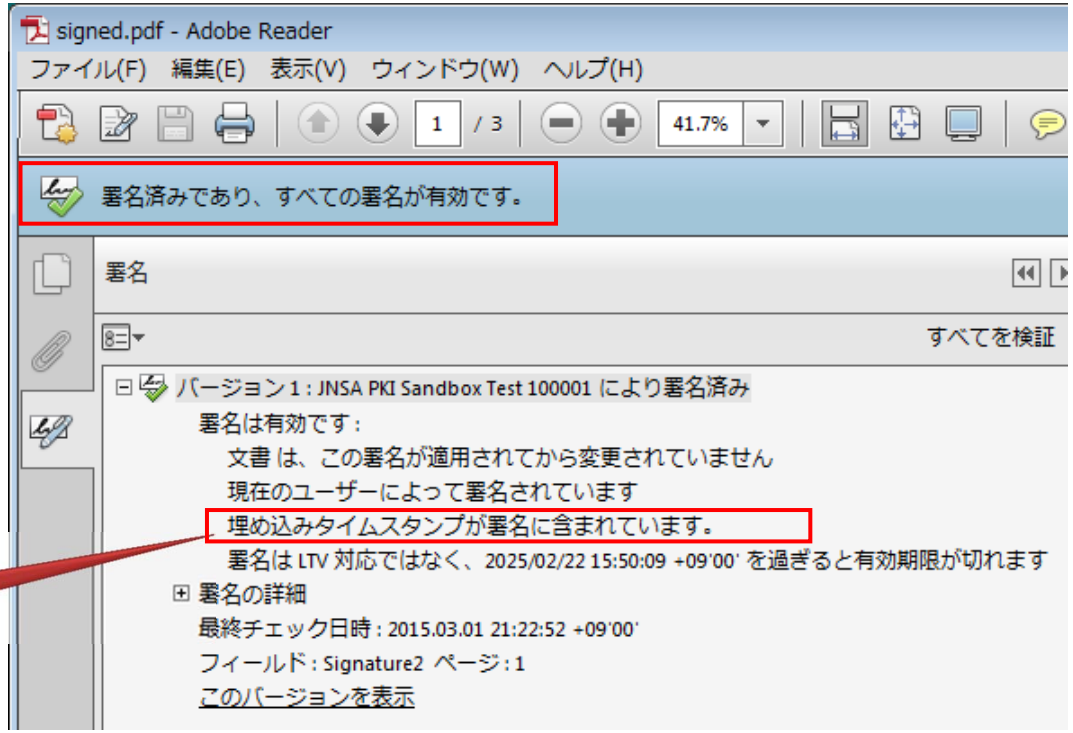
1. [環境設定]-[署名]-[検証]-[詳細]をクリック
2. [Windows統合]の項目を全てチェックする

MacOSでは1と2の設定は不要（設定項目が無い）



Windows環境では、Windows証明書ストアを信頼！

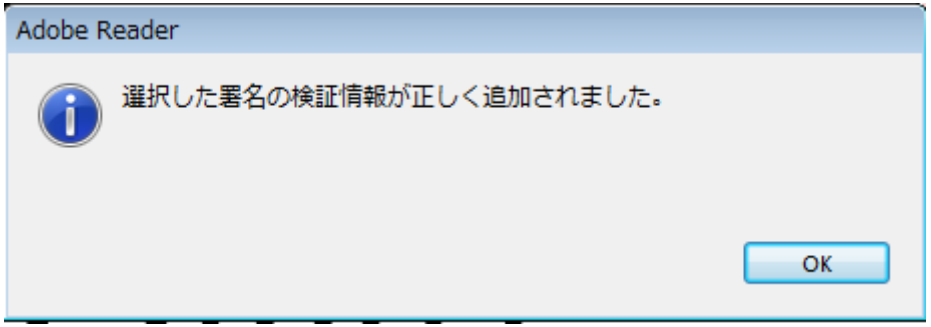
3. [すべてを検証]で再検証すると署名が有効になるはず。これでES-Tができました。

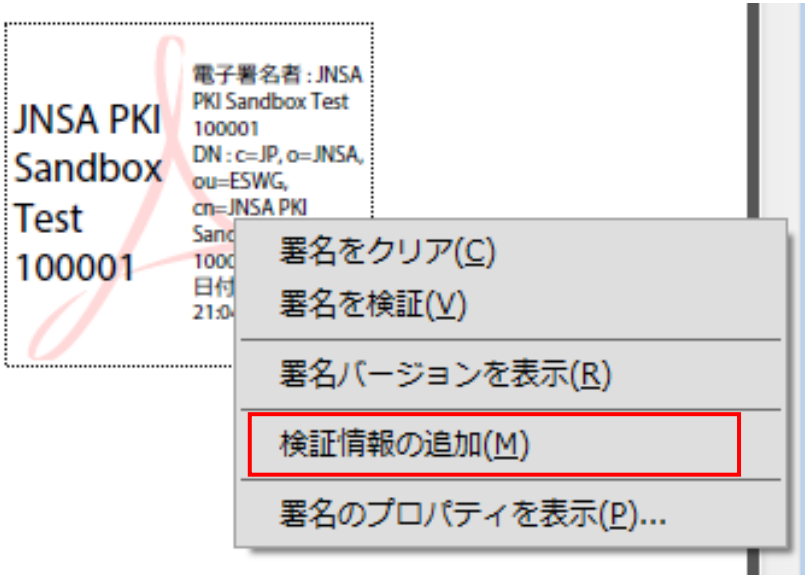


署名タイムスタンプも付いている。

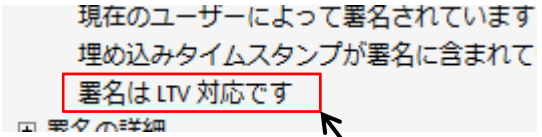
ハンズオン4-12：Adobe Readerで ES-XL 作成

1. 署名部でマウスの右クリックから[検証情報の追加]を実行。正常に追加された場合には「正しく追加されました」と表示される。





2. [閉じる]時に保存する

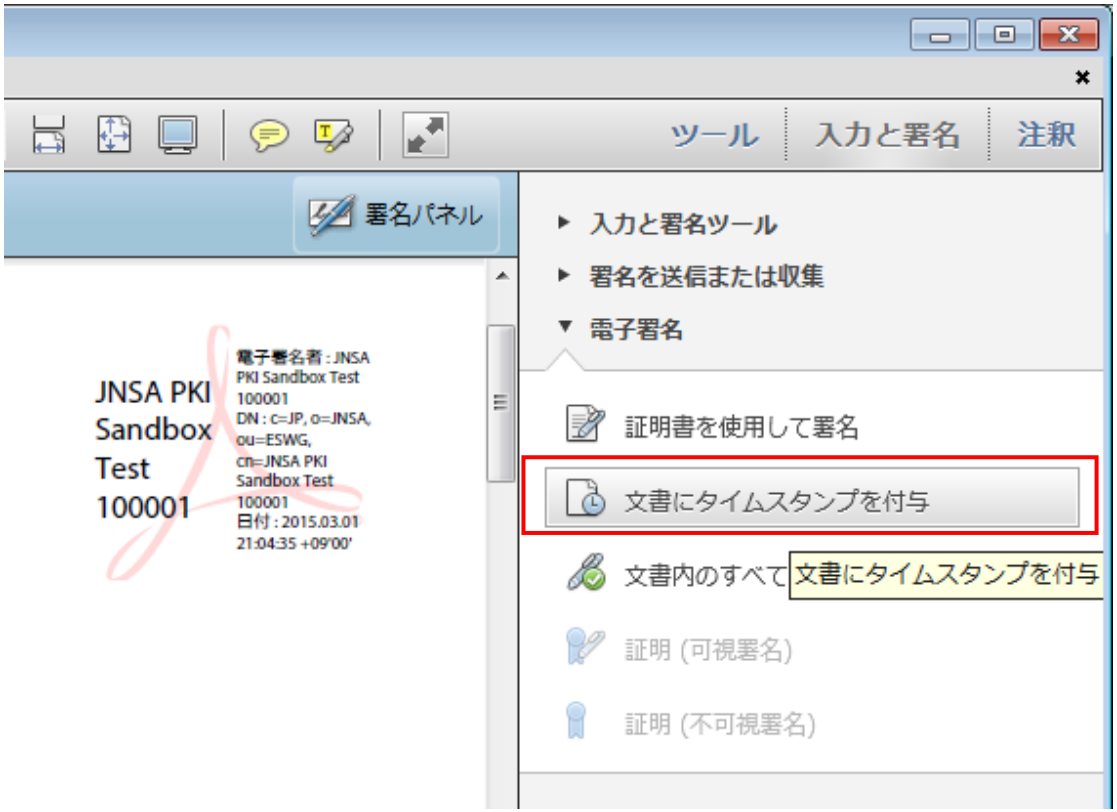


3. これでES-XLができました

※ PAdESではPAdES-LTVと呼ぶ

ハンズオン4-13：Adobe Readerで ES-A 作成

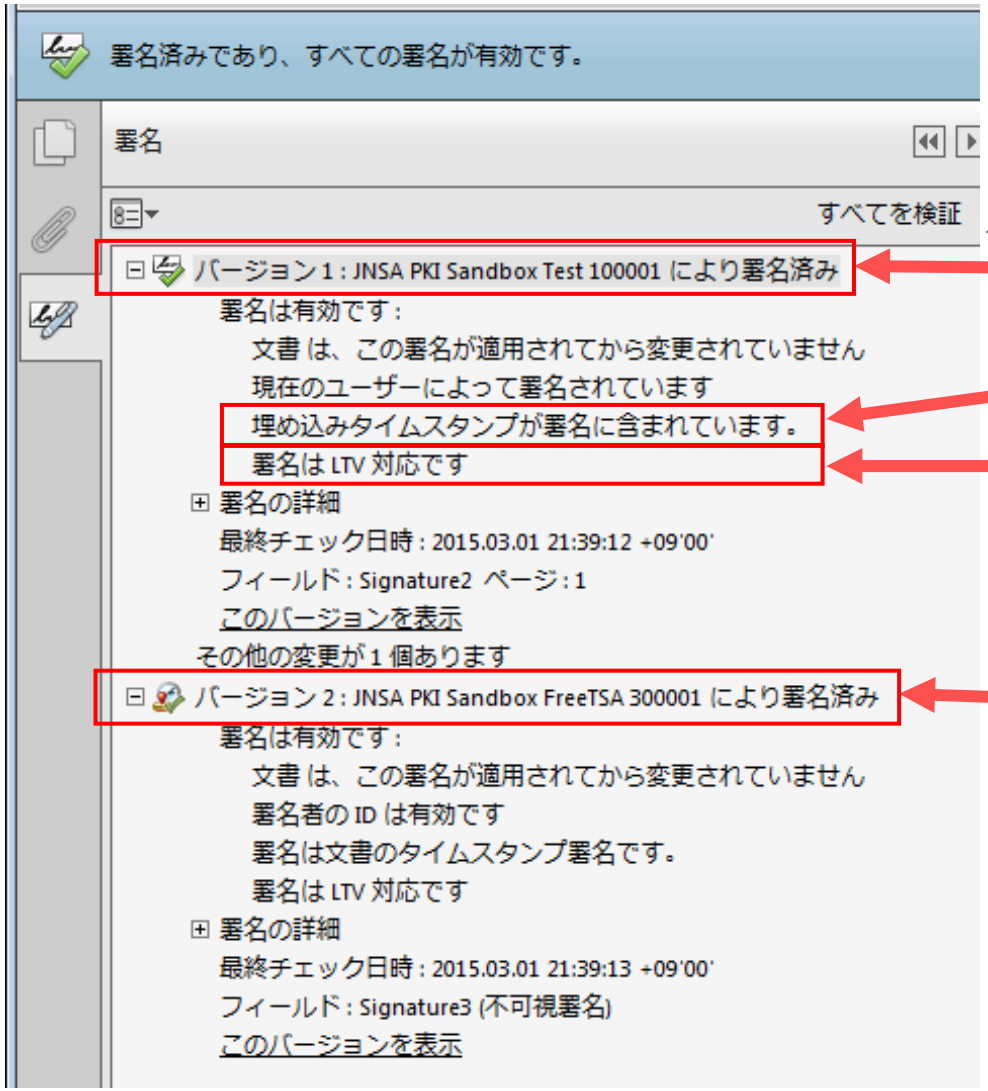
1. [電子署名]から[文書にタイムスタンプを付与]を実行する



2. [名前を付けて保存]でファイルを保存する
3. これでES-Aができました

ハンズオン4-14：Adobe Readerで ES-A 検証

1.　ES-Aファイルを開き[署名パネル]を開く



課題：これがPAdES-Aだとは長期署名の知識が無いと分からない…

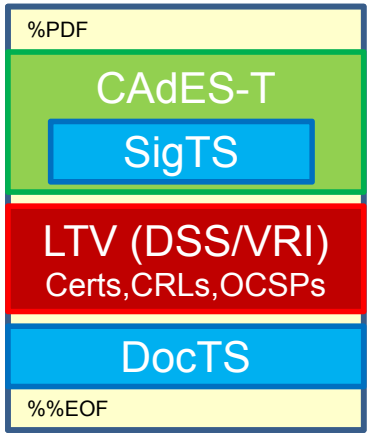
署名(ES-BES)済み

署名タイムスタンプ(ES-T)済み

検証情報埋め込み(ES-XL)済み

アーカイブタイムスタンプ(ES-A)済み

ES-Aになっているので長期保管が可能

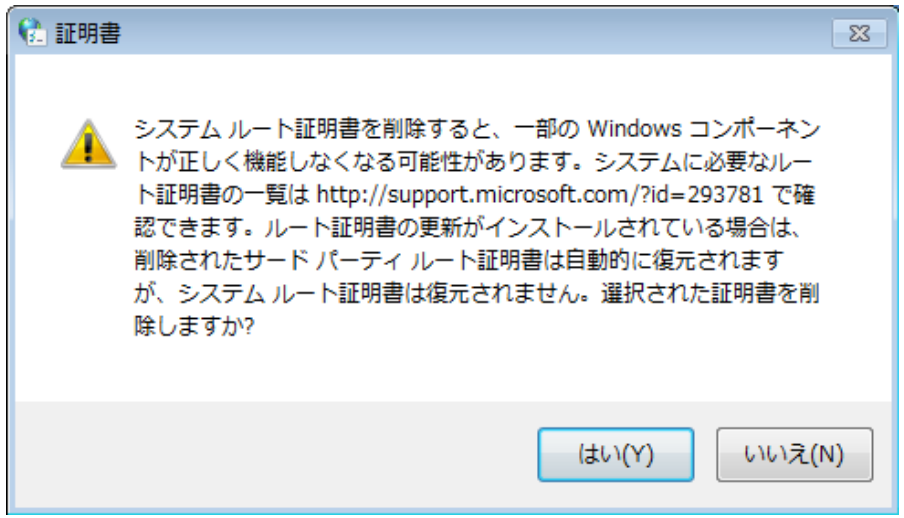


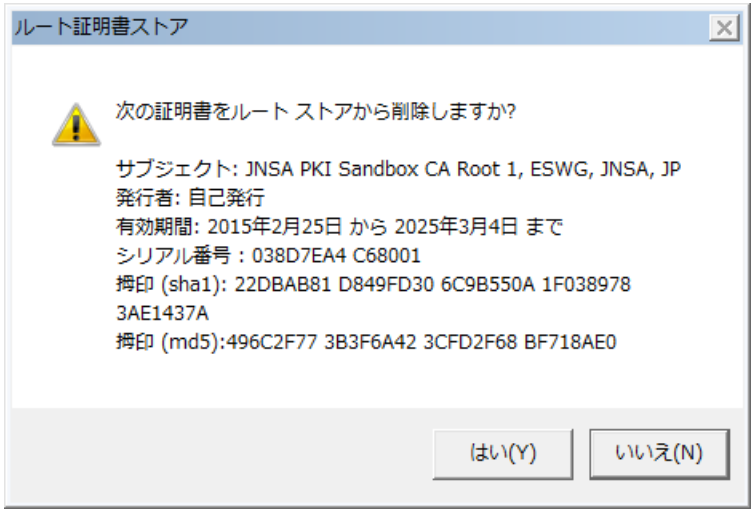
PAdES-Aイメージ

Windows：トラストアンカー削除（とても重要！）

1. [コントロールパネル]-[ネットワークとインターネット]-[インターネット オプション] を開く
2. [コンテンツ]タブから[証明書]をクリック
3. [信頼されたルート証明機関]タブを選択
4. [発行先]と[発行者]から[JNSA PKI Sandbox CA Root 1]を選択して[削除]をクリックし確認が出たら全て[はい]をクリック

お片付け完了するまでがPKIです！

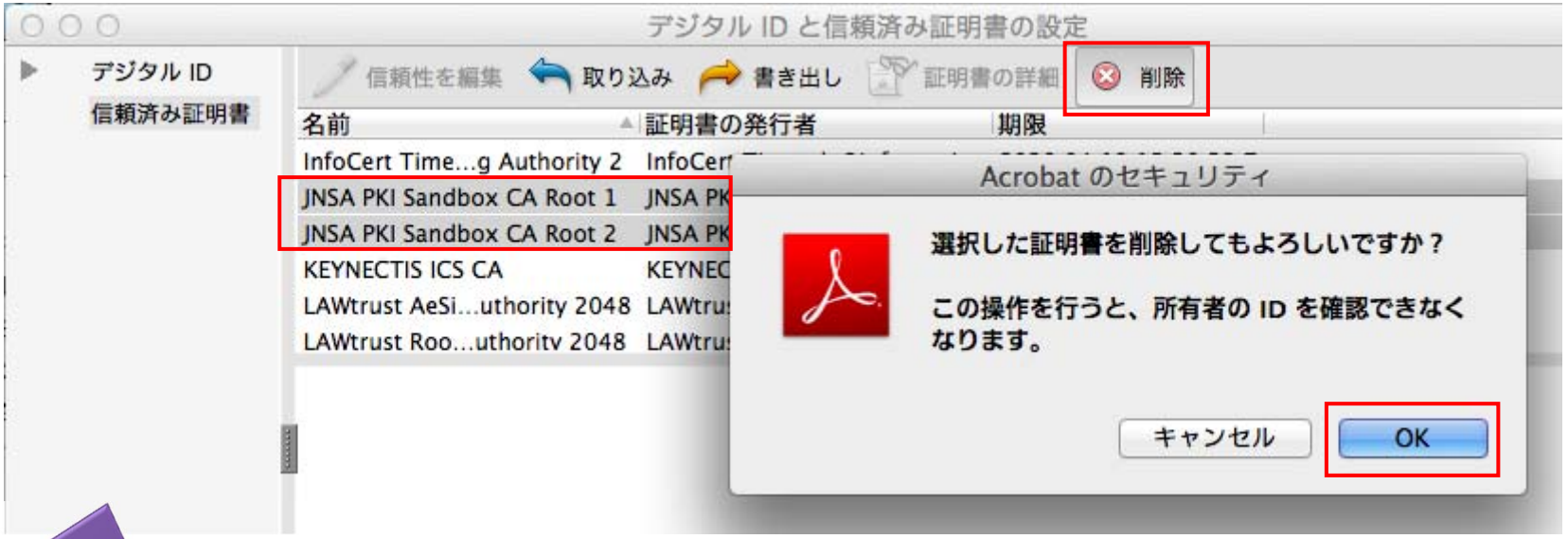




5. [JNSA PKI Sandbox CA Root 2]についても同様に削除する

Adobe:トラストアンカー削除（とても重要！）

1. [環境設定][署名][IDと信頼済み証明書]から[信頼済み証明書]を選択して[JNSA PKI Sandbox CA Root 1] と[JNSA PKI Sandbox CA Root 2]を名前から選択（個別選択して削除可能）
2. [削除]をクリックして削除する



お片付け完了するまでがPKIです！

第2部終了：お疲れさまでした！

以上で今回の電子署名ハンズオンは全て終了です。
電子署名作成とタイムスタンプ取得はできましたか？

「意外に簡単だった。」

なんて感想を持って頂けると嬉しいです。
少しでも電子署名が身近になったら
成功なんですがどうでしょう？

何にせよお疲れさまでした！

あとはビール飲んで打上げだ～！
飲みながら何でもご意見ください！

JNSA PKI SandBox Project

参考

http://eswg.jnsa.org/sandbox/

電子署名・タイムスタンプの技術を普及促進する為に誰でも自由に使えるPKIの遊び場を作るプロジェクトです！現在は以下を公開！

FreeTSA Project

試験用に使えるRFC 3161タイムスタンプサービス
※ 本日のハンズオンでも利用しました！

試験用の証明書発行やTSA証明書用の試験認証局
※ 本日のハンズオンでも利用しました！

ESig-PKI HandsOn

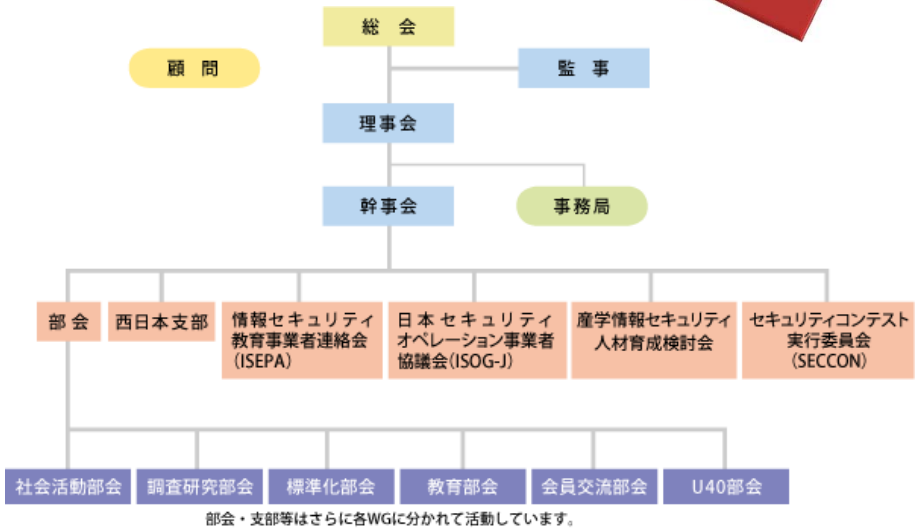
PKI SandBox の環境を使った勉強会と資料の公開
※ 本日のハンズオンで利用した資料他です！

日本ネットワークセキュリティ協会

英名：ＪＮＳＡ（Japan Network Security Assosiation）
所属：標準化部会 / 電子署名ワーキンググループ
詳細： http://www.jnsa.org/

「ＪＮＳＡ」は、ネットワークセキュリティシステムに携わるベンダーが結集しネットワーク・セキュリティの必要性を社会にアピールし、かつ、諸問題を解決していきます。

参考



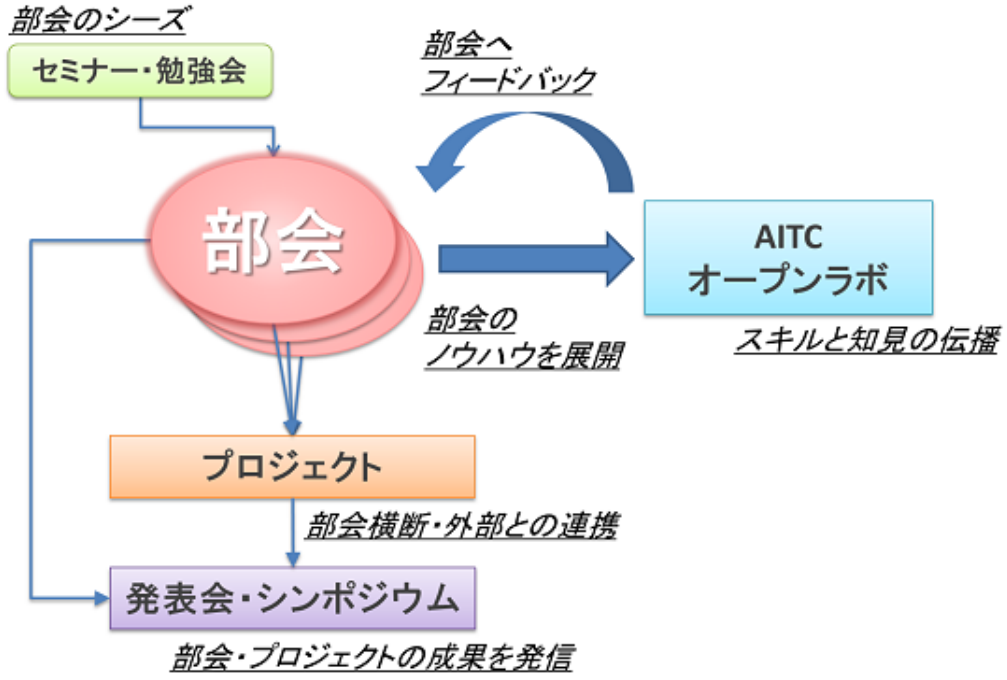
JNSA電子署名WG Facebookページ： https://www.facebook.com/eswg.jnsa.org

先端IT活用推進コンソーシアム

英名：AITC（Advanced IT Consortium to Evaluate, Apply and Drive）
由来：XMLコンソーシアムの後継団体
所属：クラウド・テクノロジー活用部会
詳細：http://aitc.jp/

参考

「先端IT活用推進コンソーシアム」は企業における先端ITの活用および先端ITエキスパート技術者の育成を目指す団体です。



http://eswg.jnsa.org/sandbox/handson/

Thank you !