JCS - JSON Clear Text Signature

Although XML Signatures are extremely flexible they come at a price: limited interoperability and mobile platform support. The case for XML has also been considerably weakened by REST which more or less has replaced SOAP for web-based systems used by for example Google and Facebook. In REST-based systems response-data is usually in JSON format.

Converting to JSON

Due to the reasons above I felt a need converting the XML-based KeyGen2 system to JSON. However, there is currently no direct counterpart to XML DSig's enveloped signatures which forced me into developing such a system. It turned out that less than 3,000 lines of Java code were required to *Encode*/*Decode* and *Sign*/*Verify* JSON data:

https://code.google.com/p/openkeystore/source/browse/#svn%2Flibrary%2Ftrunk%2Fsrc%2Forg%2Fwebpki%2Fjson

I can now safely retire my 200,000+ lines Android port of Xerces ☺

Things that make JSON signatures simpler than XML DSig include:

- No confusing attribute versus element canonicalization rules
- No namespaces
- No defaults
- No XPath
- No SOAP envelopes
- No WS-Security framework

The JSON parser mentioned does a pretty good job for supporting conformance verification with intended messages though registered message types as well through strict type control and *checks for missing references*.

Together with additional tests performed at application-level the system should be comparable to XML schema although the declarative mode of course is sadly missing.

Obviously there are complex systems that live or die by the use of XML DSig and XML Schema but KeyGen2 is hopefully not one of them…

Sample Signature

```
{
 "@context": "http://example.com/signature",
 "Now": "2013-09-13T13:17:08+02:00",
 "Order":
   {
     "Currency": "USD",
     "VAT": 1.45,
     "OrderLines":
       [{
          "Units": 1,
          "SKU": "TR-46565666",
          "UnitPrice": 4.50
        },
        {
          "Units": 3,
          "SKU": "JK-56566655",
          "UnitPrice": 39.99
        }]
   },
 "EscapeMe": "\u000F\u000aA\u0042\\\"\/",
 "Signature":
```

```
      {
        "Algorithm": "http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256",
        "KeyInfo":
          {
            "PublicKey":
              {
                "EC":
                  {
                    "NamedCurve": "http://xmlns.webpki.org/sks/algorithm#ec.p256",
                    "X": "lNxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk=",
                    "Y": "LmTlQxXB3LgZrNLmhOfMaCnDizczC/RfQ6Kx8iNwfFA="
                  }
              }
          },
        "SignatureValue":
"MEUCIAeai8SH3aLo6Mp6Fmv1Emz1GZzfn17TgJJQDfiD1wV1AiEAoObC/bKLA0HAC0RjEaz/WQGKlYVdSim77Soqjxic9+g="
      }
  }
```

Signature Scope

The scope of a signature (=what is actually signed) comprises all properties and values including possible child objects of the JSON object holding the `Signature` property minus the `SignatureValue` name-value pair.

Canonicalization

Precondition: Valid JSON data as described on http://www.json.org has been received.

The canonicalization steps are as follows:

- Whitespace is removed which in practical terms means removal of all characters outside of quoted strings having a value <= ASCII space (0x32)
- The \/ escape sequence is honored on input but is treated as a degenerate equivalent to /
- Unicode escape sequences (\uhhhh) within quoted strings are normalized. If the Unicode value falls within the traditional ASCII control character range (0x00 - 0x1f), it must be rewritten in lower-case hexadecimal notation unless it is one of the pre-defined JSON escapes (\n etc.) because the latter have precedence. If the Unicode value is outside of the ASCII control character range, it must be replaced by the actual Unicode character
- The original property order must be preserved. Zero-length properties are not allowed
- The JSON object associated with the `Signature` is recreated using the actual *textual* data. Rationale: *Numbers are ambiguously defined in JSON* which means that encoding and decoding most likely will differ among JSON implementations. For monetary data numbers like 1.00 are more or less standard, in spite of the trailing zero being redundant. There is another, more subtle issue as well. If a sender for example assigns a large number such as 0.99999999999999999 to a JSON property there is a possibility that a receiver due to limitations in arithmetic precision (like using 32-bit floating point) rather interprets it as 1.0. To cope with these potential problems, a compliant parser must preserve the original textual representation of numbers internally in order to perform proper canonicalization

The sample signature has the following canonicalization data:

```
{"@context":"http://example.com/signature","Now":"2013-09-13T13:17:08+02:00","Order":{"Currency":"U
SD","VAT":1.45,"OrderLines":[{"Units":1,"SKU":"TR-46565666","UnitPrice":4.50},{"Units":3,"SKU":"JK-
56566655","UnitPrice":39.99}]},"EscapeMe":"\u000f\nAB\\\"/","Signature":{"Algorithm":"http://www.w3
.org/2001/04/xmldsig-more#ecdsa-sha256","KeyInfo":{"PublicKey":{"EC":{"NamedCurve":"http://xmlns.we
bpki.org/sks/algorithm#ec.p256","X":"lNxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk=","Y":"LmTlQxXB3Lg
ZrNLmhOfMaCnDizczC/RfQ6Kx8iNwfFA="}}}}}
```

The text in **red** and **blue** highlight consequences of the canonicalization scheme.

Supported Signature Types

The JSON Signature scheme supports the following key types as indicated by the `KeyInfo` object:

- *RSA keys*:

```
{
  "PublicKey":
    {
      "RSA":
        {
          "Modulus": "tF3wS3naI41hzUm2q ... Yhr+a1Jhh6VpgKY4R2FlJi9Ow==",
          "Exponent": "AQAB"
        }
    }
}
```

- *EC keys:*

```
{
  "PublicKey":
    {
      "EC":
        {
          "NamedCurve": "http://xmlns.webpki.org/sks/algorithm#ec.p256",
          "X": "TQL/LgkOykT65MeeYhHCPEHoowrYckIdfGnaNYPUnLA=",
          "Y": "CuiM80A5/bAkxqnEiYkat2V+0udAk1sfn7txOx4pNR4="
        }
    }
}
```

- *X.509 certificates.* Note that if there are multiple certificates in the `X509CertificatePath` array they must: 1) Belong to a single path. 2) Be *sorted* with the signature certificate as the *first* element.

```
{
  "SignatureCertificate":                                          ┐
    {                                                              │
      "Issuer": "CN=Demo Sub CA,DC=webpki,DC=org",                 ├ Optional
      "SerialNumber": 1377713637130,                               │
      "Subject": "CN=example.com,O=Example Organization,C=US"      │
    },                                                             ┘
  "X509CertificatePath":
    [
      "MIIClzCCAX+gAwIBAgIGAUD ... aO0ixD+q5P2OszRBYG3uk9W/uNIHdoyQn19w=="
    ]
}
```

- *Symmetric keys*

```
{
  "KeyID": "hj65-9grt-076sl"
}
```

In addition to in-line keys, `KeyInfo` may also point to an external location holding a public key or an X.509 certificate path in PEM format:

```
{
  "URL": "http://example.com/my-key.pem"
}
```

Version Attribute

In similarity to CMS, a `Signature` object may also carry an *optional* `Version` property which by default has the value `"http://xmlns.webpki.org/jcs/v1"`.

Multiple Signatures

Since JSON properties are single-valued the described scheme does not automatically support multiple signings of the same object. It would be technically possible to rather use an array of signatures *but that would also greatly complicate canonicalization*.

However, there is a workaround which fits most real-world scenarios using multiple signatures and that is using wrapping signatures like the following:

```
{
    {
      "@context": "http://example.com/test-multiple-signatures",
      "Now": "2013-08-30T07:56:08+02:00",
      "ID": "lADU_sO067Wlgoo52-9L",
      "STRINGS": ["One","Two","Three"],
      "Signature":
        {
              ...
        }
    },
  "Signature":
    {
          ...
    }
}
```

That is, there is in this scheme no difference between multiple signatures and counter-signatures**.**

Acknowledgements

Highly appreciated feedback has been provided by Manu Sporny, Jim Klo, James Manger, Jeffrey Walton, David Chadwick, Jim Schaad, David Waite, Douglas Crockford and others.

Other JSON Signature Solutions

The IETF JOSE WG (http://tools.ietf.org/wg/jose) has defined a JSON signature scheme called JWS.

The primary reason why I haven't adopted JWS for KeyGen2 is because it is based on *in-line signatures using Base64-encoded payloads*.

Although certainly working Base64-encoded messages disrupts readability making the switch from XML to JSON unnecessary painful for schemes where the *message* is the core and a signature only is there to vouch for the message's authenticity. The following shows how a JWS-based conversion of the sample message could look like:

```
{
  "payload": "dTzJcZgb ... QWBBRaQnES",
  "signatures":
    [{
          Signature data
    }]
}
```

Yet another scheme which is quite similar to this specification is something known as "HTTP Keys":

https://payswarm.com/specs/source/http-keys

The authors of HTTP Keys also created their own signature scheme for multiple reasons, with clear-text messaging as one objective.

Disclaimer: This document does not represent a standard of any kind.
It might at best be useful as input to a future standardization process

Author: Anders Rundgren

anders.rundgren.net@gmail.com

2013-09-13