JSign - Clear Text JSON Signatures

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 above I felt a need to convert the currently XML-based KeyGen2 system to JSON. However, there is currently no counterpart to XML DSig's enveloped signatures which made me develop 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

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

The JSON parser mentioned also 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.

Sample Signature

```
{
  "@context": "http://example.com/signature",
  "Now": "2013-08-31T14:58:31+02:00",
  "HRT":
    {
      "RTl": "67",
      "YT":
        {
          "HTL": "656756#",
          "INTEGER": -689,
          "Fantastic": false
        },
      "er": "33"
    },
  "ARR": [],
  "BARR":
    [{
       "HTL": "656756#",
       "INTEGER": -689,
       "Fantastic": true
     },
     {
       "HTL": "656756#",
```

```
          "INTEGER": -689,
          "Fantastic": false
        }],
    "ID": "8OUbPTamxSHpn6iytuDf",
    "STRINGS": ["One","Two","Three"],
    "EscapeMe": "A\\\n\"",
    "Intra": 78,
    "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":
"MEUCIF32SJ8n+NCr0GSPu6E4ZGgr69iAERphslIyIoUQ9TGfAiEAqevMXFnlJrVIZMc81S3KvDZEZIIDWdslEMrYMNNsNkA="
      }
  }
```


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

The principle for canonicalization is simple: All textual data of the signature scope is used after *removing whitespace*. Textual data means that if a sender puts 0.99999999999999999999 in a message and the receiver would after reading the value interpret it as 1.0, the parser must still always keep the original representation internally in order to perform proper canonicalization. In addition, *properties must be sorted in descending* UTF-8 order.

The sample signature has the following canonicalization data:


```
{"Signature":{"KeyInfo":{"PublicKey":{"EC":{"Y":"LmTlQxXB3LgZrNLmhOfMaCnDizczC/RfQ6Kx8iNwfFA=","X":
"lNxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk=","NamedCurve":"http://xmlns.webpki.org/sks/algorithm#
ec.p256"}}},"Algorithm":"http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256"},"STRINGS":["One","Tw
o","Three"],"Now":"2013-08-31T14:58:31+02:00","Intra":78,"ID":"8OUbPTamxSHpn6iytuDf","HRT":{"er":"3
3","YT":{"INTEGER":-689,"HTL":"656756#","Fantastic":false},"RTl":"67"},"EscapeMe":"A\\\n\"","BARR":
[{"INTEGER":-689,"HTL":"656756#","Fantastic":true},{"INTEGER":-689,"HTL":"656756#","Fantastic":fals
e}],"ARR":[],"@context":"http://example.com/signature"}
```


Version Attribute

In similarity to CMS there is an *optional* `Version` attribute which by default has the value `http://xmlns.webpki.org/jsign/v1`.

Supported Signature Types

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

- *Asymmetric keys*. RSA and EC:

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

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

- *X.509 certificates*

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

- *Symmetric keys*

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

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**.**

Other JSON Signature Solutions

The IETF JOSE WG 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.

Author: Anders Rundgren

anders.rundgren.net@gmail.com

2013-08-31