Web Services: Security 2

Endre András, Ferran Delgado, Gauthier Picalausa

Overview

1. Quick Recap
2. WS-family specifications
   - WS-Security
   - WS-Authorization
   - WS-SecureConversation
3. OpenID
4. OAuth
5. Amazon S3

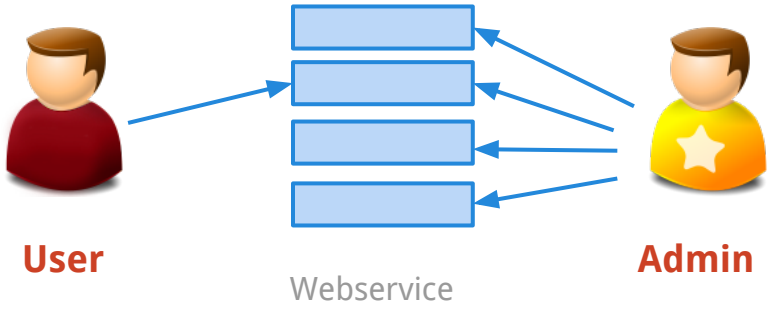
Recap

- **Authentication**

  Confirming identity

- **Authorization**

  Access rights



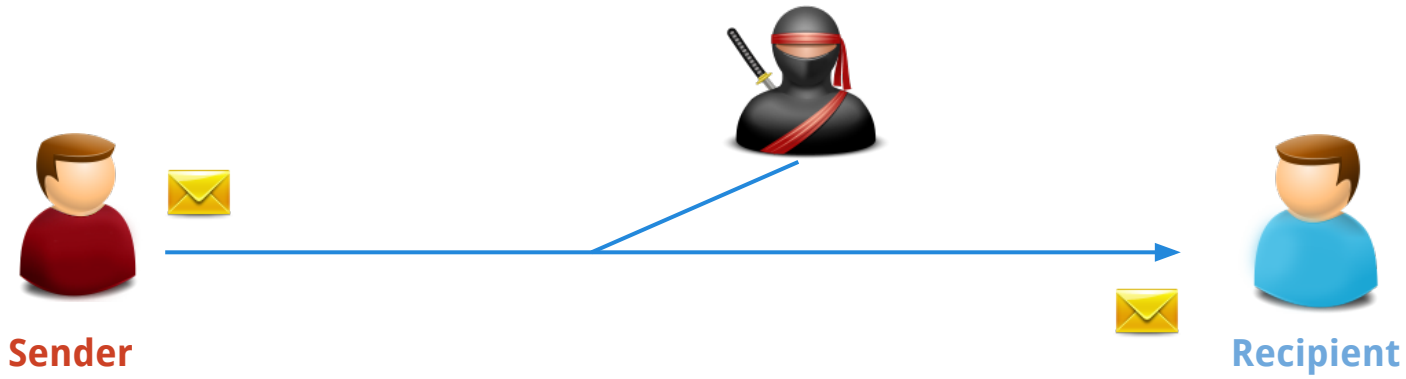
Recap

- **Integrity**

  The property that data has not been modified.

- **Confidentiality**

  The property that data is not made available to unauthorized individuals

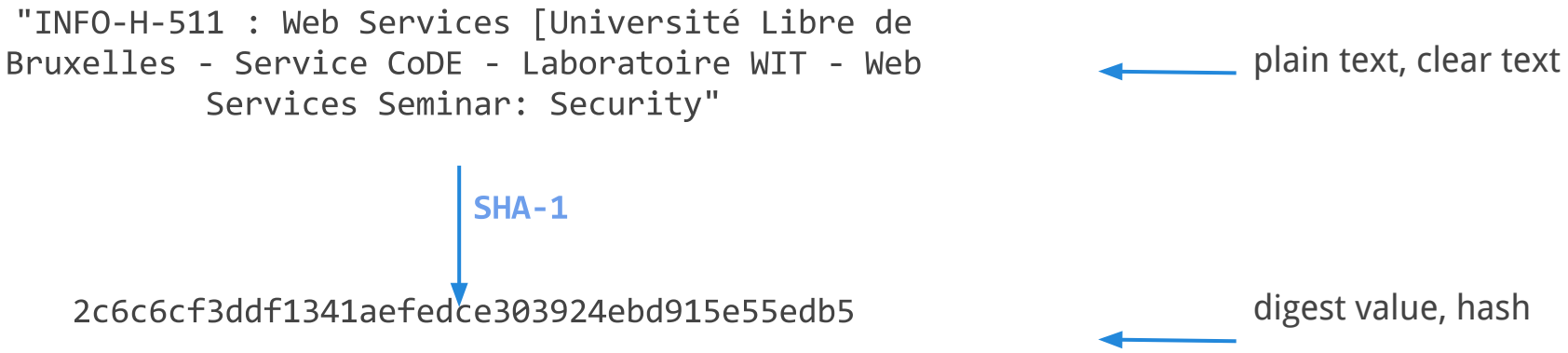


Recap

- **Hash function**

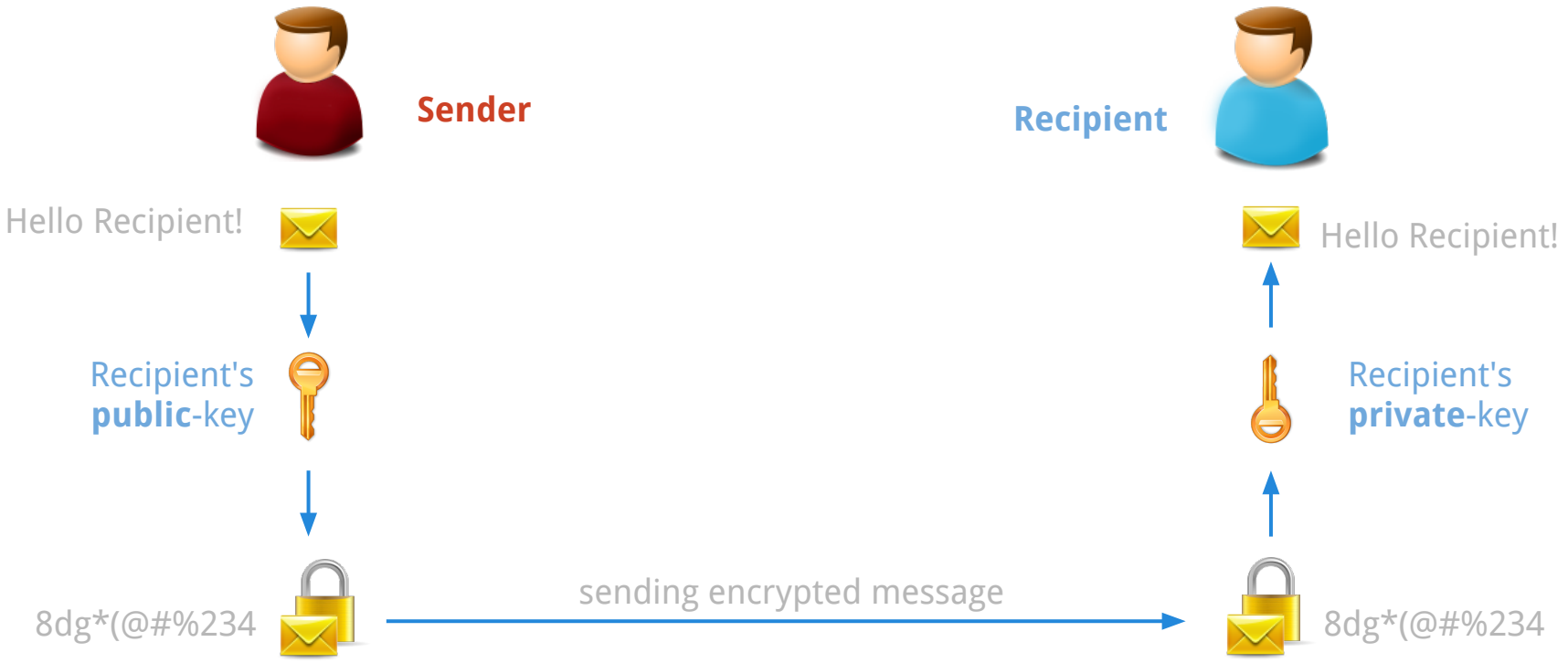
  Maps a large data set to a smaller fixed length data set.

  E.g. MD5, SHA-1,SWIFFT

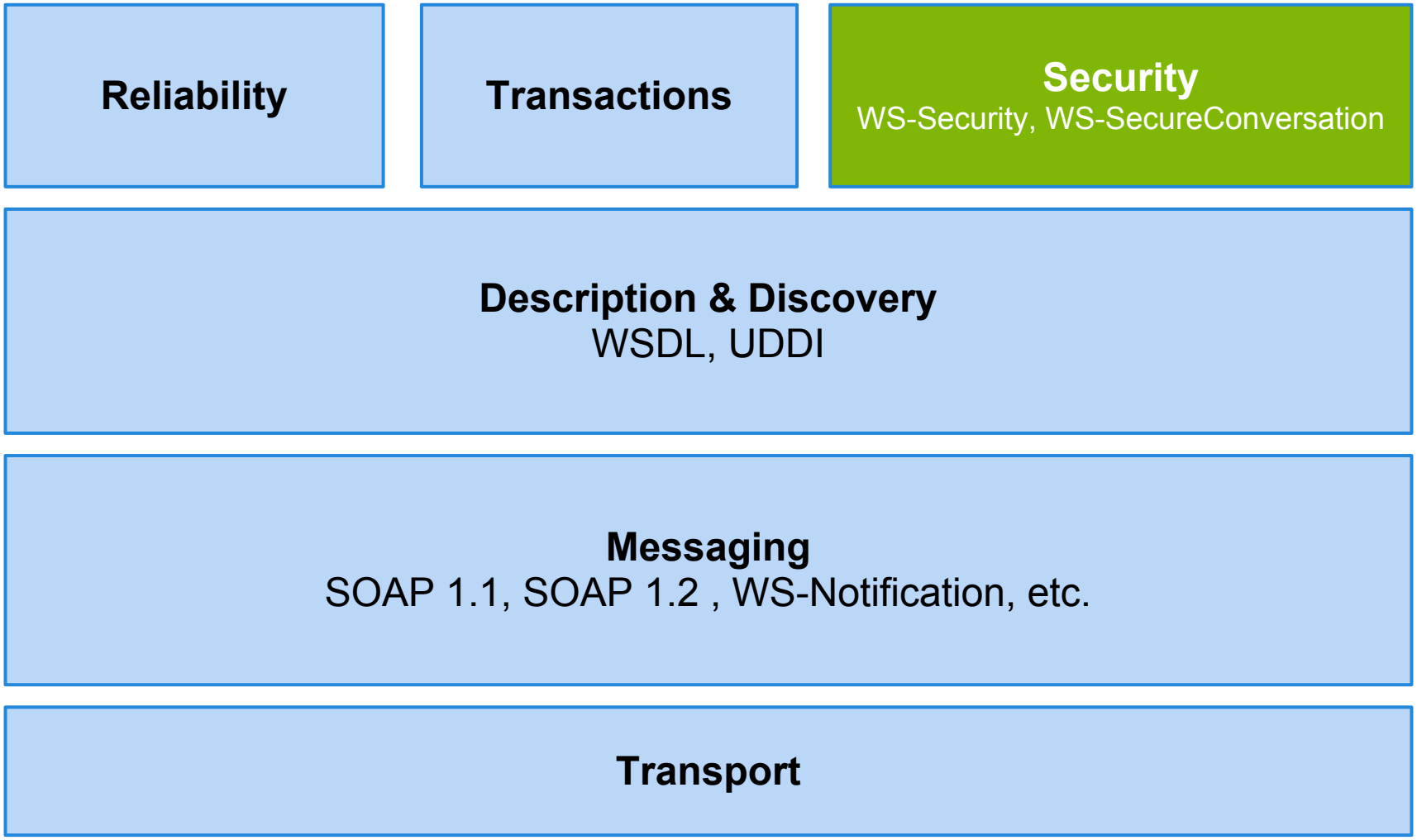


Recap

- **Public-key cryptography (asymmetric)**



WS-Security in WS-*



Why WS-Security is needed?

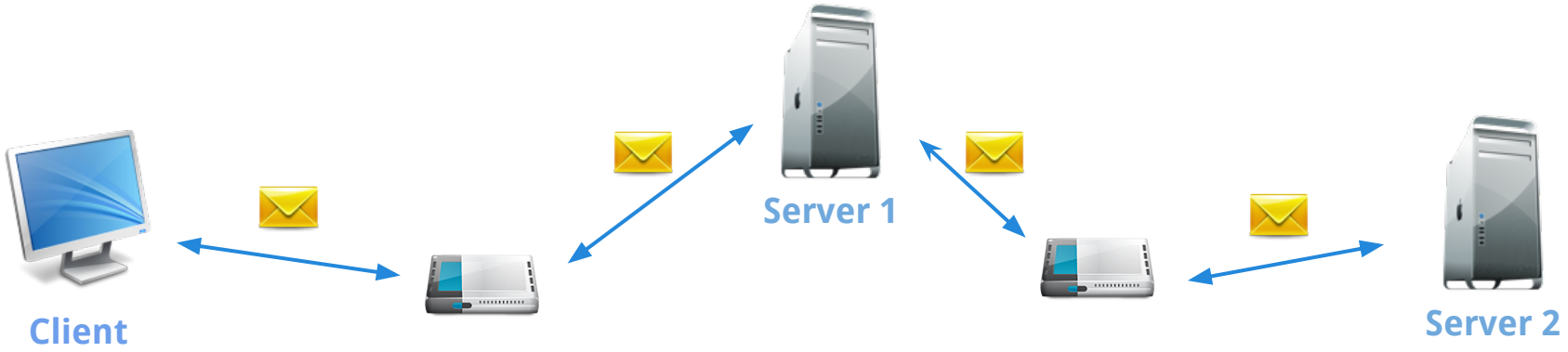
1. The simple case

Communication between two endpoints

Over HTTP, one can authenticate the caller, sign the message, and encrypt the contents of the message.

Why WS-Security is needed?

2. SOAP to solve bigger problems



The identity, integrity, and security of the message and the caller need to be preserved over multiple hops

We need end-to-end security, rather than point-to-point

What is WS-Security?

It uses existing standards and specifications

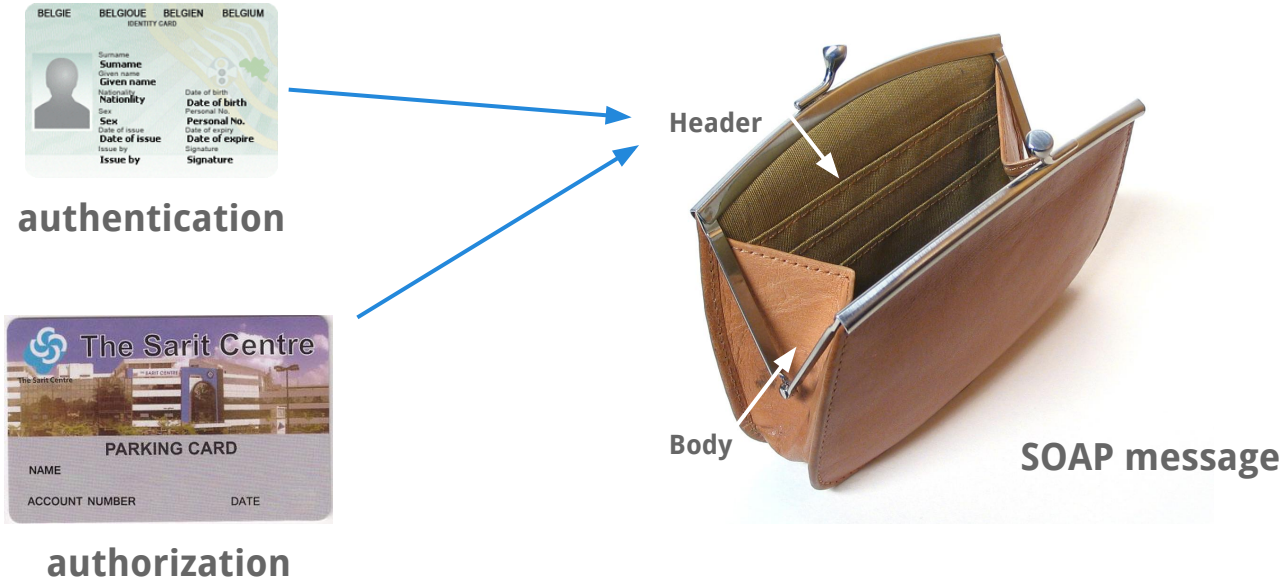
- **Authentication**
  Kerberos, X.509, username and password

- **Encrypting and signing messages**
  XML Encryption and XML Signature

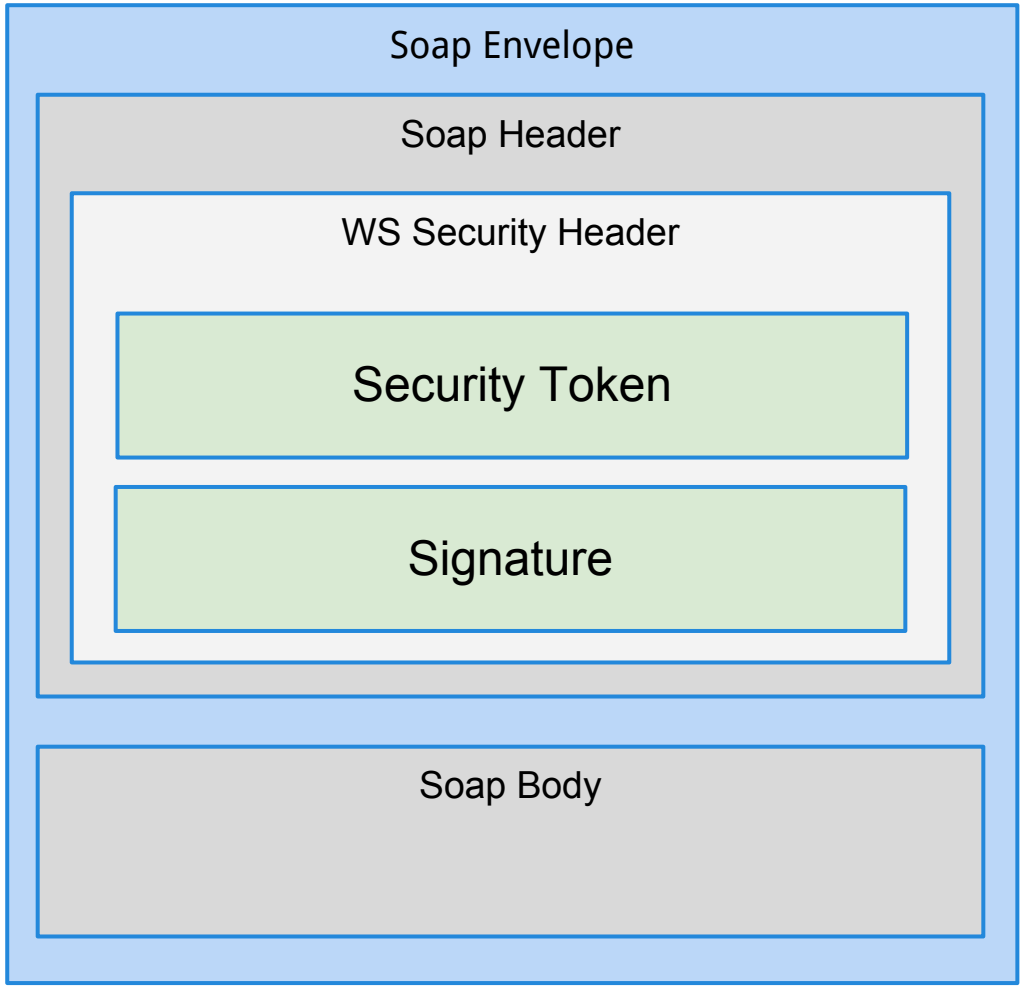
- **Preparing documents to be signed and encrypted**
  XML Canonicalization

What is WS-Security?

Is a way (framework) that describes how to embed existing specifications into a SOAP messages.



WSS in SOAP header



Simplified Example

```
<env:Header>
    <wsse:Security>
        <wsse:UsernameToken>
            <wsse:Username>scott</wsse:Username>
            <wsse:Password Type="wsse:PasswordDigest">
                KE6QugOpkPyT3Eo0SEgT30W4Keg=
            </wsse:Password>
        </wsse:UsernameToken>
    </wsse:Security>
</env:Header>
```

WS-Security credential management

Two special elements covers all:

- **<UsernameToken>**
  Transferring simple user credentials

- **<BinarySecurityToken>**
  Embedding binary security tokens such as *Kerberos Tickets* or *X.509 Certifications*

BinarySecurityToken Example

```
<env:Header>
    <wsse:Security>
        <wsse:BinarySecurityToken
             ValueType="wsse:X509v3"
             EncodingType="wsse:Base64Binary"
            Id="SecurityToken-f49bd662-59a0-401a">
               MIIHdjCCB...
        </wsse:BinarySecurityToken>
    </wsse:Security>
</env:Header>
```

wsu:Timestamp & wsu:Id


```
<wsu:Timestamp>
    <wsu:Created wsu:Id="Id-2af5d5bd-1f0c-4365">
       2002-08-19T16:15:31Z
    </wsu:Created>
    <wsu:Expires wsu:Id="Id-4c337c65-8ae7-439f-b4f0">
       2002-08-19T16:20:31Z
    </wsu:Expires>
</wsu:Timestamp>
```

message is valid for the next 5 min.

Encryption example

```
<soap:Header>
    <wsse:Security soap:mustUnderstand="1" >
        <xenc:ReferenceList>
            <xenc:DataReference URI="#EncryptedContent-f6f50b24" />
        </xenc:ReferenceList>
    </wsse:Security>
</soap:Header>
```

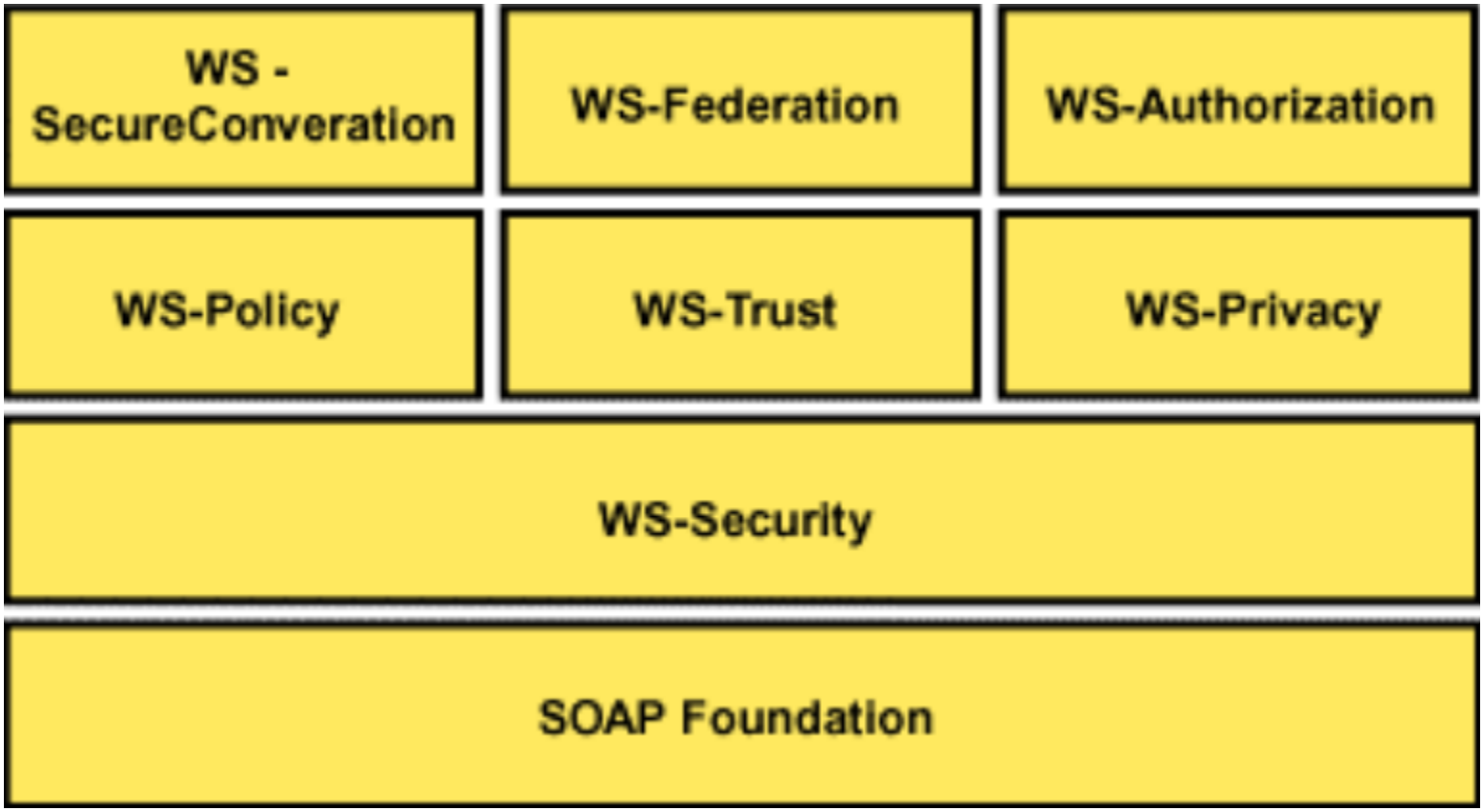
Encryption example <soap:body>

```
<xenc:EncryptedData
    Id="EncryptedContent-f6f50b24"
     Type="http://www.w3.org/2001/04/xmlenc#Content">
    <xenc:EncryptionMethod Algorithm="w3.org/..tripledes-cbc" />

    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <KeyName>Symmetric Key</KeyName>
   </KeyInfo>

    <xenc:CipherData>
        <xenc:CipherValue>
            InmSSXQcBV5UiT...  Y7RVZQqnPpZYMg==
        </xenc:CipherValue>
    </xenc:CipherData>
</xenc:EncryptedData>
```

Encryption example <soap:body>

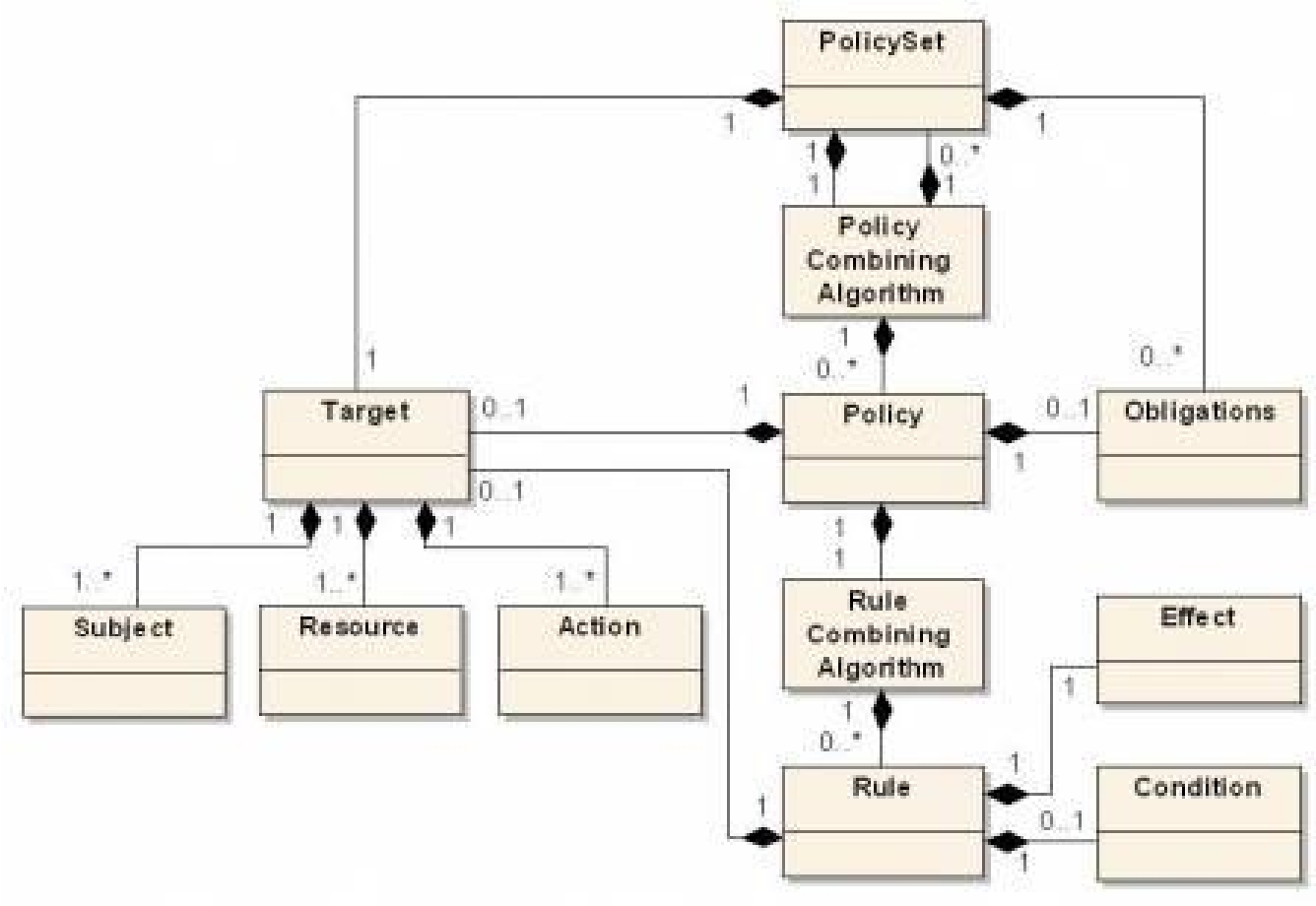


WS-Authorization

- Very Similar to XACML
- Manages Data and authorization policies
  Makes authorization decisions
- Add role-based authorization
- Current Status : Unknown

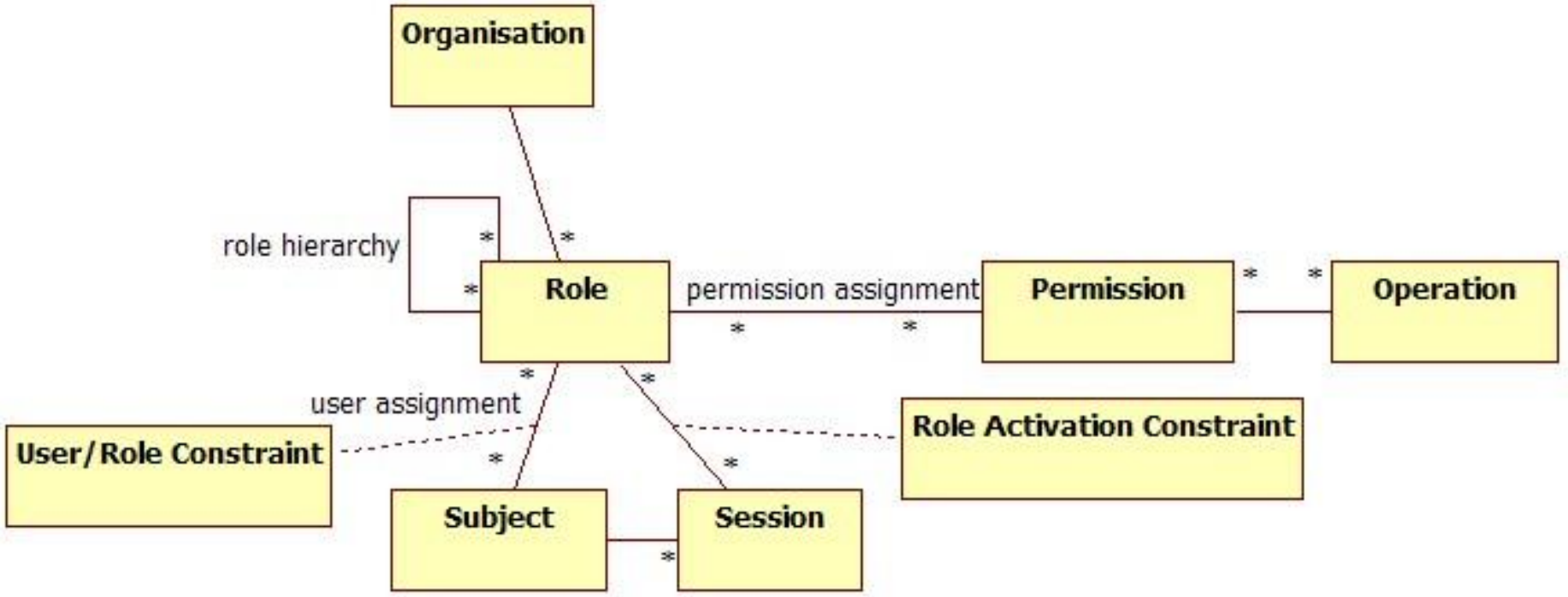
WS-Authorization

- Recap : XACML

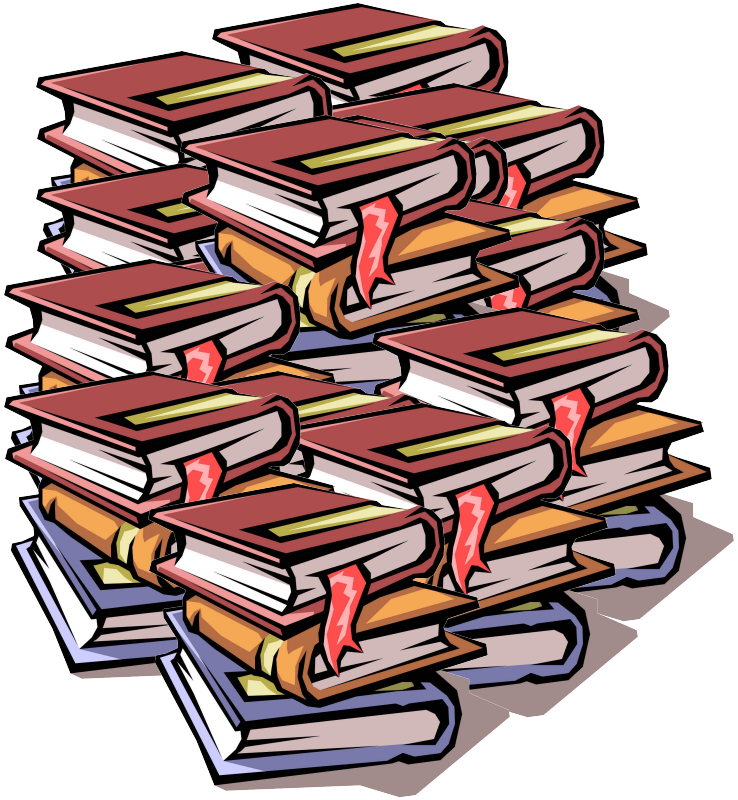


WS-Authorization

- Role based authorization



WS-SecurityConversation



WS-SecureConversation

A security context provides a way to provide session based security, rather than establishing new keys for every message

WS-SecureConversation

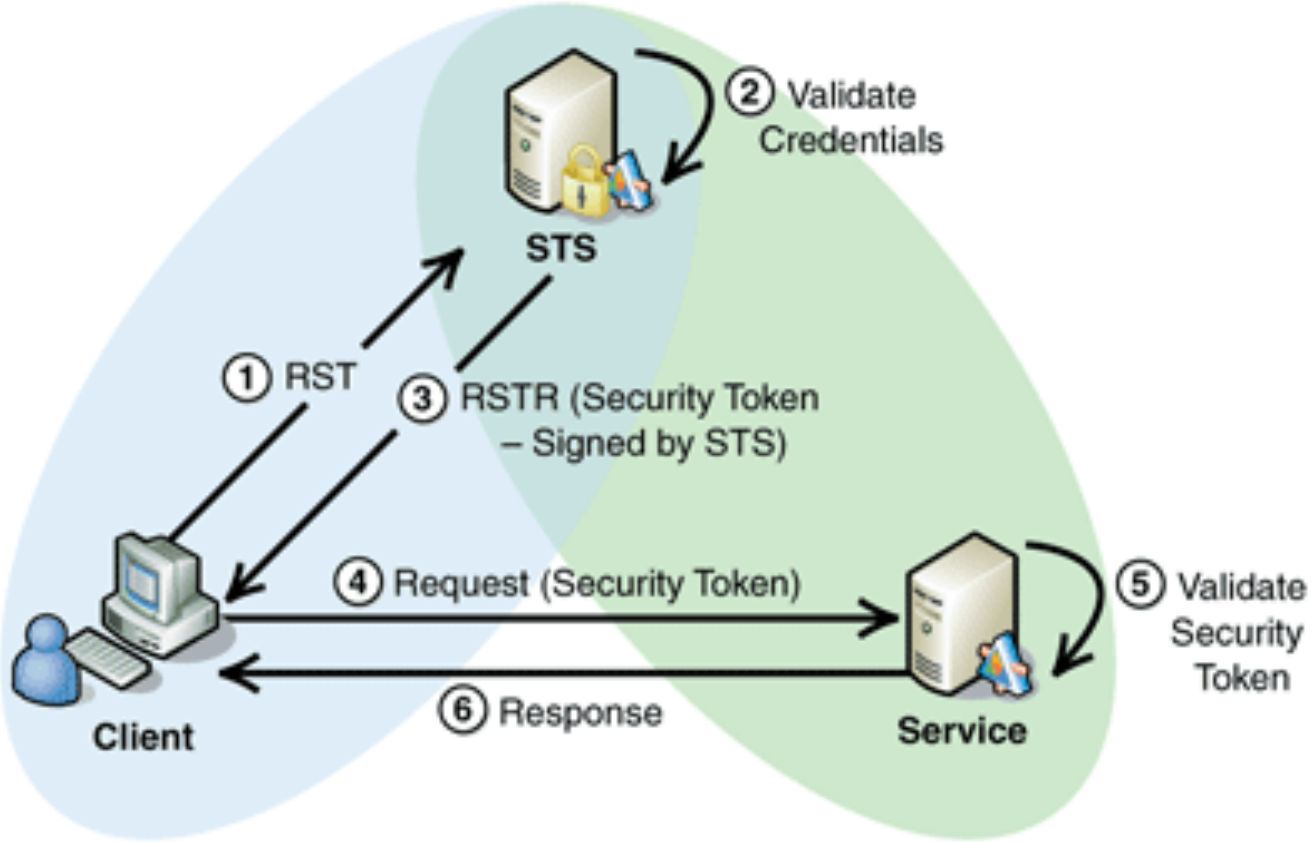
```
<wsc:SecurityContextToken
 xmlns:wsu='http://docs.oasis-open.org/wss/2004/01/...'
 xmlns:wsc='http://schemas.xmlsoap.org/ws/2004/04/sc'
 wsu:Id='MySCT'>
    <wsc:Identifier></wsc:Identifier>
    <wsu:Created>2004-05-07T21:24:06</wsu:Created>
    <wsu:Expires>2004-05-08T09:24:06</wsu:Expires>
</wsc:SecurityContextToken>
```

WS-SecureConversation: Establishing Security Contexts

- Created by a security token service
- Created by one of the communicating parties
- Created through negotiation

WS-SecureConversation: Establishing Security Contexts

- Created by a security token service



WS-SecureConversation: Establishing Security Contexts

- Created by one of the communicating parties

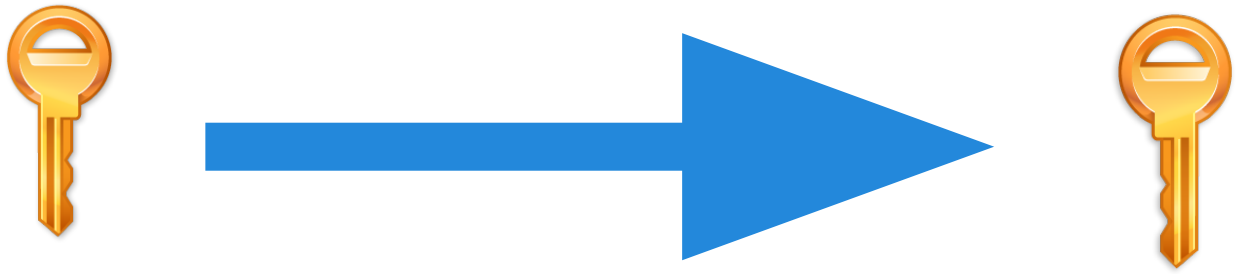
  a. The initiator creates a security context token and sends it to the other parties on a message

  b. The recipient can then choose whether or not to accept the security context token.

WS-SecureConversation: Establishing Security Contexts

- Created through negotiation
  - The initiating party sends a <RequestSecurityToken> request to the other party
  - A <RequestSecurityTokenResponse> is returned.
  - Repeat the above 2 steps until a final response containing a <SecurityTokenReference> and a <ProofTokenReference> is received.

WS-SecureConversation: Derived Key



WS-SecureConversation: Derived Key

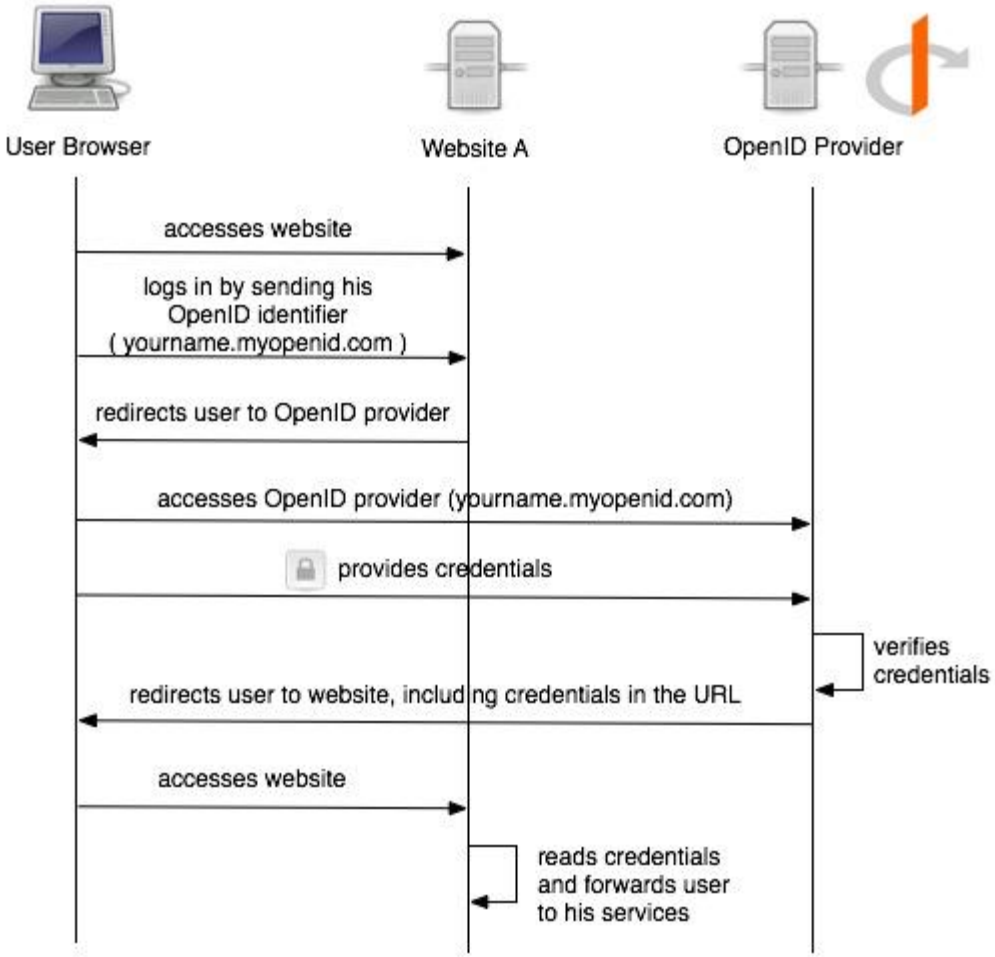
```
<wsc:DerivedKeyToken
 xmlns:wss='http://docs.oasis-open.org/wss/2004/01/oasis-2...'
 xmlns:wsu='http://docs.oasis-open.org/wss/2004/01/oasis-200...'
 xmlns:wsc='http://schemas.xmlsoap.org/ws/2004/04/sc'
 Algorithm='http://schemas.xmlsoap.org/ws/2004/04/security/sc/dk/p_sha1'
 >
 <wss:SecurityTokenReference>
     <wss:Reference URI='#MySCT'
          ValueType='http://schemas.xmlsoap.org/ws/2004/04/security/sc/sct' />
     </wss:SecurityTokenReference>
     <wsc:Length>16</wsc:Length>
     <wsc:Label>MyLabelText</wsc:Label>
     <wsc:Nonce>CXrD7xEjbLQzN6au+RRfTQ==</wsc:Nonce>
</wsc:DerivedKeyToken>
```

A lot of ids !


OFFICIAL PASSPORT
CERTIFICATE OF IDENTITY
DOCUMENT OF IDENTITY
PASSPORT
DIPLOMATIC PASSPORT
EMERGENCY PASSPORT

OpenID

OpenID


User Browser
Website A
OpenID Provider
accesses website
logs in by sending his OpenID identifier (yourname.myopenid.com)
redirects user to OpenID provider
accesses OpenID provider (yourname.myopenid.com)
provides credentials
verifies credentials
redirects user to website, including credentials in the URL
accesses website
reads credentials and forwards user to his services

OpenID

Is OpenID secure?

OpenID

List of the providers

Name	Ease of us	Security	Remembers info	Multiple profiles	Antiphishing	Password protected
MyOpenId	8	9	**X**	**X**	**X**	**X**
Google	7	4	**X**			**X**
Yahoo!	10	4			**X**	**X**
VeriSign	7	7	**X**		**X**	**X**
wordPress	5	1	**X**		**X**	**X**

OAuth

"You would like people to get access to the books they want but not the employee list"

- Authorize partial access
- New role : resource owner

OAuth - history

- 2006 : Blaine Cook, OpenID and Twitter
- 2007 : OpenAuth google group
  AOL OpenAuth protocol
- 4th December 2007 : OAuth is released

OAuth - history

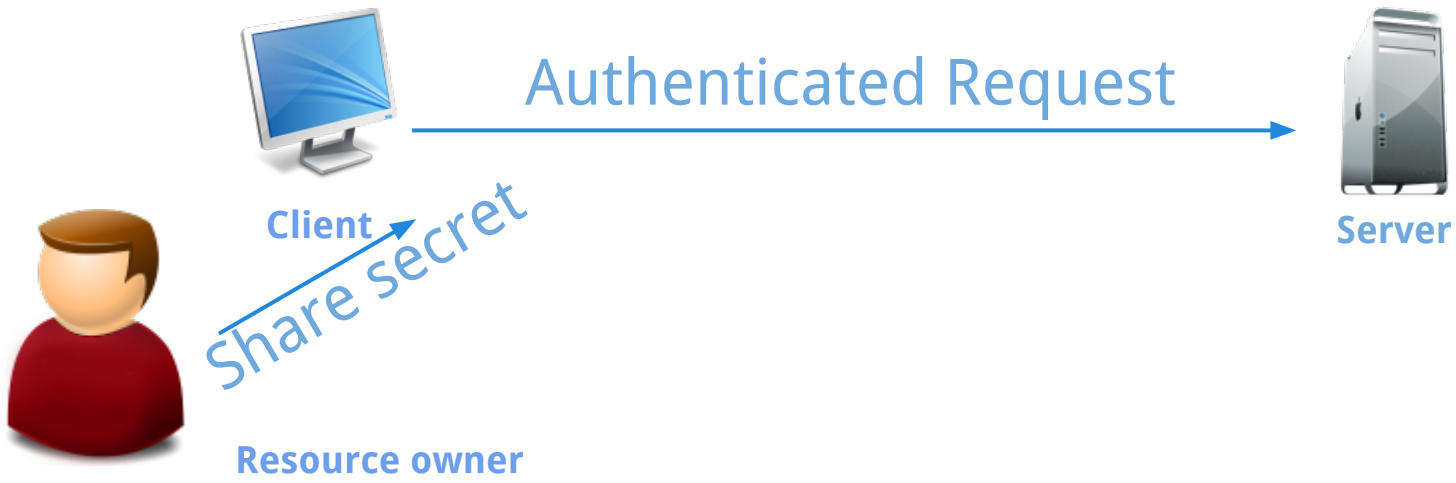
- June 2009 : First revision
- March 2009 : OAuth Working Group created

  OAuth Core 1.0 RFC Edition
- November 2009 : OAuth 2.0 project
- October 2012

OAuth - Terminology

Traditionally

OAuth



OAuth - Terminology

- **Protected Resource**

"A protected resource is a resource stored on (or provided by) the server which requires authentication in order to access it. Protected resources are owned or controlled by the resource owner. Anyone requesting access to a protected resource must be authorized to do so by the resource owner (enforced by the server)."

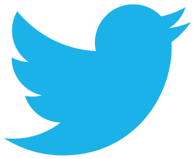
OAuth - Terminology

- **Credentials**
  - Client
  - Temporary
  - Token

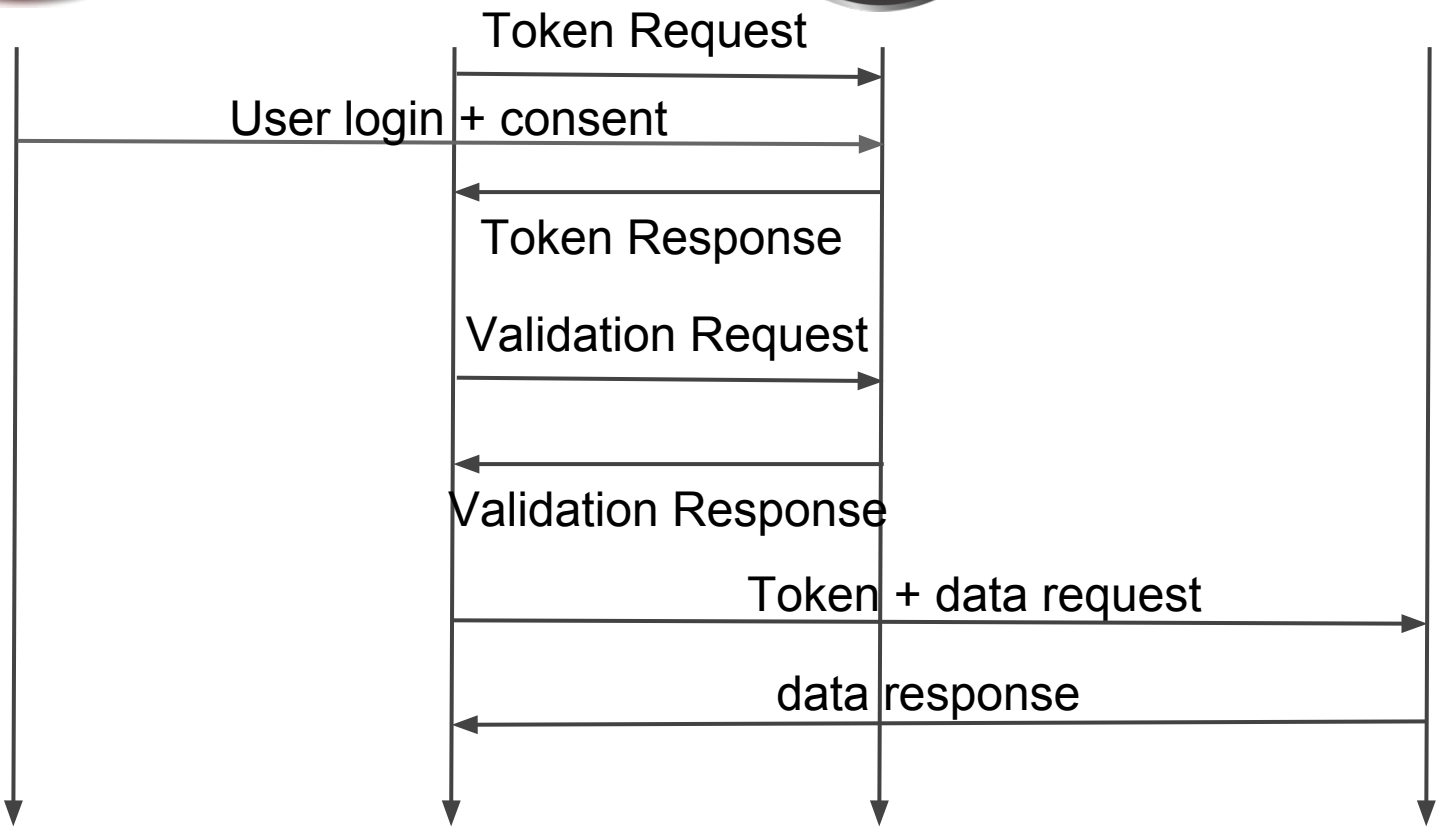
OAuth - Security

- Basic
- Credentials (client + token)
- Signature and hash
- TimeStamp and Nonce

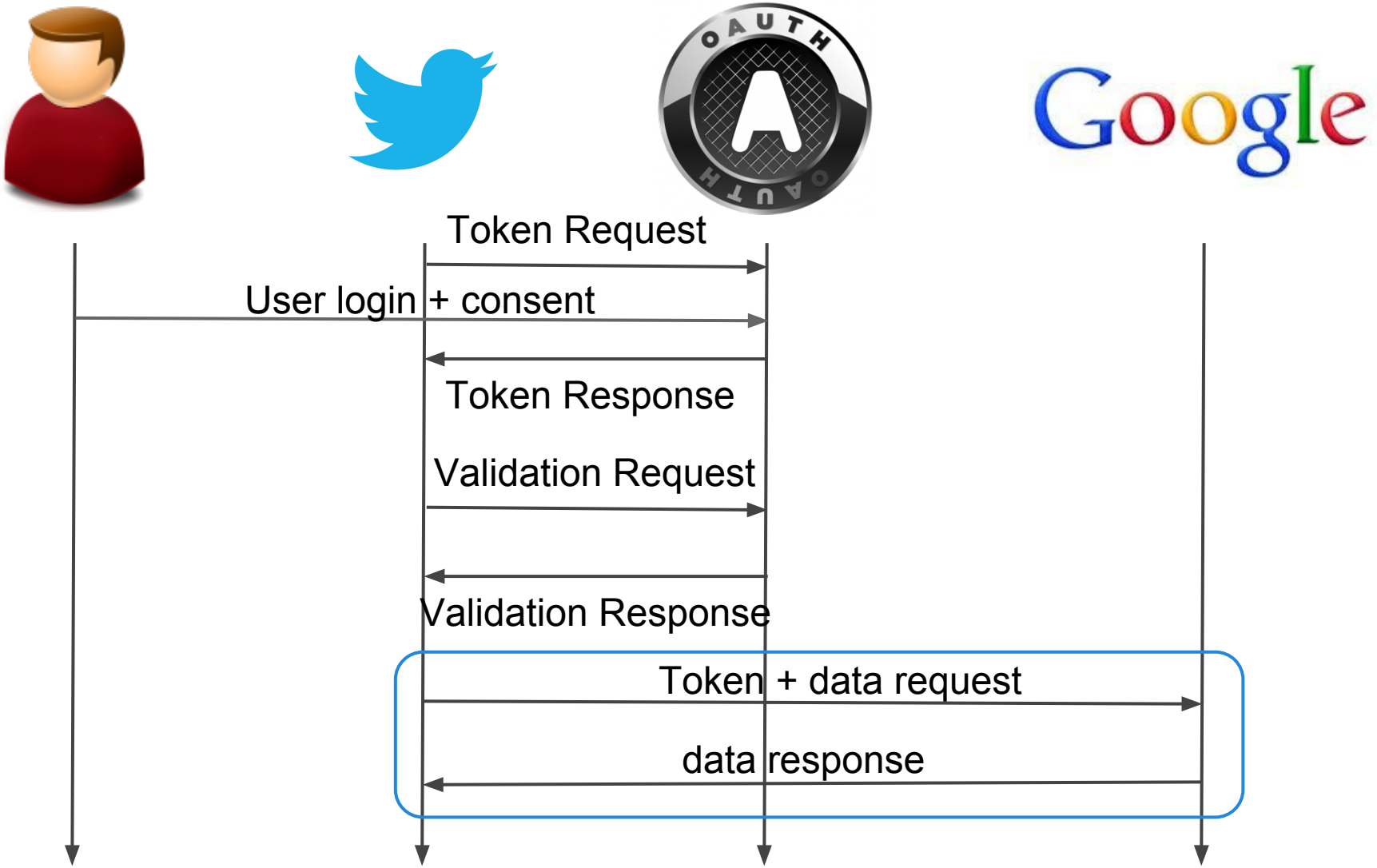
OAuth - Credential security



Google



OAuth - Credential security



OAuth - Data request

OAuth header - sent

realm	Bleh
oauth_consumer_key	dpf43f3p2l4k3l03
oauth_token	nnch734d00sl2jdk
oauth_nonce	kllo9940pd9333jh
oauth_timestamp	1191242096
oauth_signature	tv90v06QjdPVw3c5EoEAN4/hoW4=
oauth_signature_method	HMAC-SHA1
oauth_version	1.0

OAuth - Data request

Http request

```
GET /photos?size=original&file=vacation.jpg HTTP/1.1
Host: photos.example.net:80
Authorization: OAuth realm="http://photos.example.net/photos",
    oauth_consumer_key="dpf43f3p2l4k3l03",
    oauth_token="nnch734d00sl2jdk",
    oauth_nonce="kllo9940pd9333jh",
    oauth_timestamp="1191242096",
    oauth_signature_method="HMAC-SHA1",
    oauth_version="1.0",
    oauth_signature="tR3%2BTy81lMeYAr%2FFid0kMTYa%2FWM%3D"
```

OAuth - Data Access

- Url
- OAuth header
- 'application/x-www-form-urlencoded' POST body

OAuth2

- Not backwards compatible
- Aims to improve where 1.0 was limited and confusing
- New flows

OAuth2

- Bearer tokens
- Simplified signatures
- Short-lived token with long-lived authorisation

OAuth2 - By Eran Hammer

- **Unbounded tokens** - In 2.0, the client credentials are no longer used. This means that tokens are no longer bound to any particular client type or instance.

- **Bearer tokens** - 2.0 got rid of all signatures and cryptography at the protocol level. Instead it relies solely on TLS. This means that 2.0 tokens are inherently less secure as specified.

OAuth2 - By Eran Hammer

- **Expiring tokens** - developers now need to implement token state management. The reason for token expiration is to accommodate encrypted tokens which can be authenticated by the server without a database lookup. Because such tokens are self-encoded, they cannot be revoked and therefore must be short-lived to reduce their exposure.

OAuth2 - example

- Google Example

Amazon S3

- **S3 = Simple Storage Service**
  Amazon S3 provides storage through web services interfaces (REST, SOAP, and BitTorrent)

- **Buckets**
  http://s3.amazonaws.com/`bucket`/`key`

S3: Authenticate REST requests

- **Amazon S3 REST API uses a custom HTTP scheme** based on a keyed-HMAC (Hash Message Authentication Code) for authentication

- **At registration developers get:**
  AWS Access Key ID
  AWS SecretAccess Key

S3: REST requests

```
GET /photos/puppy.jpg HTTP/1.1
Host: johnsmith.s3.amazonaws.com
Date: Mon, 26 Mar 2007 19:37:58 +0000

Authorization: AWS AKIAIOSFODNN7EXAMPLE:frJIUN8DYpKDtOLCwo//yllqDzg=
```

S3: Signature

```
Signature = Base64( HMAC-SHA1(SecretAccessKey,UTF-8-Encode(StringToSign)));

StringToSign = HTTP-Verb + "\n" +
    Content-MD5 + "\n" +
    Content-Type + "\n" +
    Date + "\n" +
    CanonicalizedAmzHeaders +
    CanonicalizedResource;

CanonicalizedResource = [ "/" + Bucket ] +
    <HTTP-Request-URI, from the protocol name up to the query string>
```

S3: GET request example

AWSAccessKeyId: AKIAIOSFODNN7EXAMPLE
AWSSecretAccessKey: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

Request

```
GET /photos/puppy.jpg HTTP/1.1
Host: johnsmith.s3.amazonaws.com
Date: Tue, 27 Mar 2007 19:36:42 +0000

Authorization: AWS AKIAIOSFODNN7EXAMPLE:
bWq2s1WEIj+Ydj0vQ697zp+IXMU=
```

String to sign

```
GET\n
\n
\n
Tue, 27 Mar 2007 19:36:42 +0000\n
/johnsmith/photos/puppy.jpg
```

Thank you for your attention

WS-Security signing
OpenID symmetric keys
OAuth 1.0 hashing
cryptography REST API
SOAP confidentiality
authentication authorization
integrity encryption
WS-* OAuth 2.0