Send comments to:
Phillip Hallam-Baker, Senior Author
401 Edgewater Place, Suite 280
Wakefield MA 01880
Tel 781 245 6996 x227
Email: pbaker@verisign.com

Security Assertions Markup Language

Straw-man Architecture

Phillip Hallam-Baker *VeriSign*

Draft Version 0.1: February 16th 2001

Printed on Friday, February 16, 2001

Security Assertions Markup Language

Version 0.1

Table Of Contents

Printed on Friday, February 16, 2001

Table of Figures

Executive Summary

A straw-man architecture is proposed to elucidate the architectural implications of the requirements implicit in the SAML use cases document.

1 Introduction

This paper presents a number of architectural ideas suggested by proposals in the SAML use cases group.

The terminology chosen is intentionally separate from that employed in the use cases for reasons that will become apparent.

2 Abstract Data Flow

The SAML protocol specification supports transfer of security assertions between an issuing server, a relying server and a client (Figure 1).

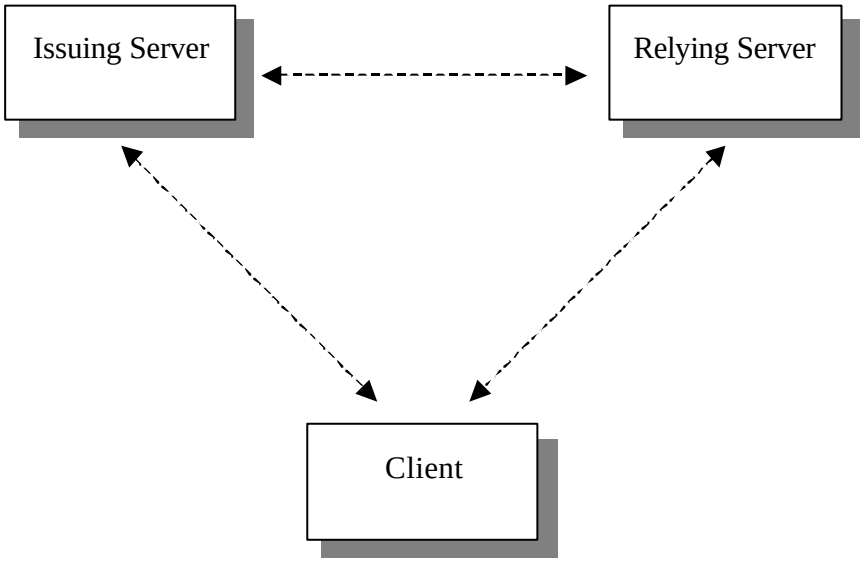

Figure 1: Parties to the protocol

2.1 Client

The application program that is requesting access to a resource. In each case it is the client that initiates a sequence of protocol messages.

2.2 Issuing Server

Also known as a *Policy Decision Point* (PDP), the issuing server is responsible for issuing assertions and assertion tickets.

2.3 Relying Server

Also known as a *Policy Enforcement Point* (PEP), the relying server acts on information encoded in assertions and assertion tickets to determine whether a party is to be allowed access to a resource or not.

Although a resource controller may subordinate policy decisions to a remote resource we exclude by definition the possibility of subordinating policy enforcement[1].

2.4 Configurations

The data-flow between the parties should be independent of the communication graphs supported. In particular the client may not have direct access to the issuing server and all interactions between the issuing server and the client may be intermediated by the relying server acting as a proxy Figure 2.

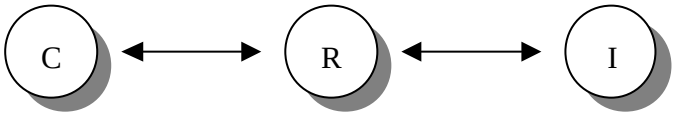

Figure 2: Relying Server acts as Proxy to Issuing Server

A wide variety of network configurations may be implemented involving multiple issuing servers and multiple relying servers. Some of these configurations are discussed in detail in section 6 below.

3 Data Objects

3.1 Security Assertion

An XML data structure that makes a security assertion. Typical assertions include:

- The party with account ID Alice has the *Plumber* right.

[1] The effect of PEP subordination may be achieved through Ford/Weiner type schemes in which access to an encrypted resource is controlled by controlling access to the relevant keying material. However for the purposes of this standard we may simply consider the cryptographic keying material to be the resource to which access is actually being controlled and thus subordination is not taking place for the purposes of this specification.

- The Party with the account ID Alice is permitted to access resource X
- Any party with the Plumber right is permitted to access resource X

We employ the XTASS framework to represent assertions. XTASS provides a very general framework for encoding of assertions, some parts of which do not address problems within SAML scope (for example management of embedded root keys).

X-TASS assertions may encode authorization data in one of two ways:

1) As a URI identifying either a resource itself (i.e. a URL of the resource) or a rights identifier associated with the resource (e.g. via a URN). The mapping of rights identifiers to resources themselves may be achieved using SAML or through another mechanism outside the scope of the specification.

2) By incorporating additional elements into the assertion that are defined in a separate schema (this is not currently in the X-TASS schema but should be).

All X-TASS assertions share a common set of XML elements specifying information about the assertion, including:

- A URI that uniquely identifies the assertion
- Status of the assertion
- Validity interval
- Conditions placed on validity

3.2 Ticket

A *ticket* is compact data structure that identifies a particular assertion. A ticket MAY be authenticated and MAY carry encrypted data.

The principal purpose of tickets is to support the constraints imposed by zero footprint clients. It is not possible to encode all the information encoded in an assertion in the minimal space available in a URL fragment or HTTP cookie.

A second use of tickets is to provide a lightweight means of communicating cryptographic keying material in the manner of Kerberos [Kerberos].

A possible syntax for encoding tickets is provided in Appendix A . Issuing servers and relying servers may use a different ticket format by private agreement however.

For architectural purposes it is desirable that tickets have the following properties:

- Be compact, allowing the minimum data set to be encoded in 64 bytes or less.

- Support authentication by means of a shared key
  [Could add option to do a DSA signature]

- Support encryption by means of a shared key

- Specify the account identifier of the party to whom the ticket was issued and whether the identifier was authenticated.

- Allow encoding of authentication data (e.g. a shared key established between the client and issuing server)

- Be extensible to allow applications to encode data from arbitrary XML assertion elements.

3.3 Meta-Assertion

An assertion that modifies the status of one or more previously issued assertions.

4 Constraints

4.1 Zero Footprint Client Constraints

A 'zero footprint' client is defined for design purposes to be a Web browser supporting a lowest common denominator feature set, i.e. requiring no feature not supported by both Internet Explorer 4.0 and Netscape Navigator 3.5 and not requiring active code such as a plug-in, Java Applet or Active-X control.

A zero footprint client would not receive assertions but MAY receive a ticket encoded in either a Cookie or a URL.

4.2 SAML Aware Client Constraints

A SAML aware client supports handling of SAML messages directly. Support may be integrated into the client application or provided by a client plug in, applet or Active-X control.

4.3 Server Constraints

Simpler is better. However if there is a choice to be made between implementation or configuration complexity in the client and complexity in the server, the latter is to be favored in most cases.

5 Protocol Exchanges

5.1 Assertion / Ticket Issue Request

The client authenticates itself to the Issuing server by some means (typically username/password or public key authentication scheme).

5.2 Access Query

Request Data

- The party or class of parties requesting access
- The specific resource or class of resources for which access is requested
- The context of the request

Examples

- Is the party that presented ticket *T* allowed access to resource *X* via security protocol *P*?
- What rights are associated with ticket *T*?
- Is a party with the *Plumber* right permitted to access resource *X*?

Response

- Access Permitted; Party with ticket *T* may access resource *X* in time interval *I*.
- Access Denied; Party with ticket *T* may not access resource *X*
- Party with ticket *T* has the *Plumber* right.
- A Party with the *Plumber* right may access resource *X*.

5.3 Access Account Query

A future revision of the protocol could extend the access query transaction to add in accounting so that each time a resource was accessed an adjustment was made to the relevant account.

Examples

- Printer charging for each page printed
- Pay per view

- Detect excessive access to secure resources

5.4 Session Management / Distributed Log Out

An important special requirement is the need to ensure that parties whose authentication credentials have been withdrawn can no longer access resources through cached credentials.

Support for this requirement inevitably involves a greater degree of complexity than the case in which an issued assertion is never revoked until it expires. Either the relying server must query the status of an assertion each and every time it is used or the issuing server must pro-actively notify all relying servers whenever an assertion is revoked.

We propose the use of XTASS Tier 2, meta-assertions to support notification of status changes. These allow support for both the 'push' and 'pull' models of status notification.

5.4.1 Status Pull Model

In the status pull model the relying server queries the status of the assertion each time it is retrieved from the cache. The server queried may be either the original issuing server or another server indicated in the assertion:

❶ Client makes request for resource X
Relying Server makes initial query to Issuing server,
Issuing Server returns an assertion with status Valid and containing a Verify element as a Condition.
Relying server returns resource X to client
Assertion is cached

❷ Relying server makes second request for resource X
Relying server retrieves assertion from cache
Relying server request status from service specified in Verify clause
Verify service returns status Valid
Relying server returns resource X to client

❸, ❹ As in case ❷

❺ Relying server makes second request for resource X
Relying server retrieves assertion from cache
Assertion has expired, Relying Server makes new query to issuing server
etc.

❻ Relying server makes second request for resource X
Relying server retrieves assertion from cache
Relying server request status from service specified in Verify clause
Verify service returns status Invalid
Assertion deleted from cache Relying Server makes new query to issuing server
etc.

Note that it is generally desirable to support negative caching in a protocol so that negative results are stored as well as positive. However the mere revocation of an assertion does not in the general case indicate that all statements made by the assertion is false.

5.4.2 Status Push Model

In many cases it is desirable to avoid the need to introduce a status validation exchange for each transaction. It is inefficient in the general case to continuously distribute status updates for all assertions and notification protocols are subject to denial of service attacks. The X.509/PKIX family of specifications has given rise to a large number of CRL management options including Version 1 CRLs, delta CRLs, Certificate Distribution Points and Scoped CRLS.

The X-TASS meta-assertion model defines a single simple compact data structure that allows any of the CRL management options of X.509/PKIX to be employed. This structure is used in a manner analogous to an Access Control List and consists of a list of statements specifying the current status of one or more assertions identified by URI. Each statement may be market terminal or non-terminal. The list of statements is processed in order until the first matching statement that is marked terminal is found or the end of the list is encountered.

Example, assertion identifier is `urn:abd/323`, the assertion list is:

- `First="urn:abd/100" Last="urn:abd/500" status="Valid" Terminal="False"`
  Matches, status is Valid, Rule is not marked terminal so processing continues.

- `First="urn:abd/325" status="Invalid" Terminal="True"`
  Does not match, ignore.

- `First="urn:abd/323" status="Invalid" Terminal="True"`
  Matches, status is Invalid, Rule marked terminal so processing is complete.

- `First="urn:abd/105" status="Invalid" Terminal="True"`
  Processing is complete, rule is ignored

Protocol Example:

❶ Client makes request for resource X
Relying Server makes initial query to Issuing server,
Issuing Server returns an assertion with status Valid and containing a Listen element as a Condition.
Relying server returns resource X to client
Assertion is cached

❷ Relying server makes second request for resource X
Relying server retrieves assertion from cache
Verify service returns status Valid
Relying server returns resource X to client

❸, ❹ as in case ❷

❺ Issuer notifies Relying Server that the assertion status is Invalid
Relying server deletes assertion from cache

❻ Relying server makes second request for resource X
Relying server retrieves assertion from cache
Verify service returns status Valid, Access Denied
Relying server returns refusal to client

Depending on implementation the issuing service may or may not track dependencies on particular assertions.

5.5 Push-me-Pull-you Model

An implementation may combine the push and pull models. Relying servers may employ the pull model to request status updates on cached assertions from a local responder acting as a gateway. The gateway responder acts as a listener for status updates distributed under push model and tracks local dependencies on particular assertions.

6 Network Configurations

6.1 Multiple Issuing Servers

The access control process is typically divided into a series of distinct stages, for example the access control mechanism embedded in the VMS operating system incorporates the following stages:

Authentication
Alice authenticates herself by entering her username and password at the login prompt

Rights Authorization
The ALICE account has the rights identifiers FINANCE_SUPER, FINANCE_ADMIN

Resource Authorization
The database file is associated with an Access Control List that specifies that read access is permitted to accounts with the rights identifier FINANCE_ADMIN, write access requires the rights identifier FINANCE_SUPER

Similar multi-stage authorization schemes are supported by other operating systems. Although the need for multiple authorization steps is well established the number of steps required and distinction between those steps is not.

The lack of a sharp distinction between multiple authorization steps strongly suggests that the issue should be left to implementation decision. The protocol SHOULD support an application implementation in which a sequence of authorization messages are employed but should not enforce an arbitrary distinction in the degree of granularity specified.

The configuration in Figure 3 shows an example of a configuration in which two separate Issuing servers address different aspects of authorization. Issuing server 1 (possibly an external resource) returns information corresponding to 'Authorization Rights', Issuing server 2 (likely to be an internal enterprise resource) returns information specific to a particular resource request.

In the example shown Alice (*A*) requests access to resource *X* controlled by the relying server. The relying server first queries Issuing Server 1 to discover that Alice has the *Plumber* right. The relying server then queries Issuing Server 2 to determine whether a party with the *Plumber* right may access the resource X.

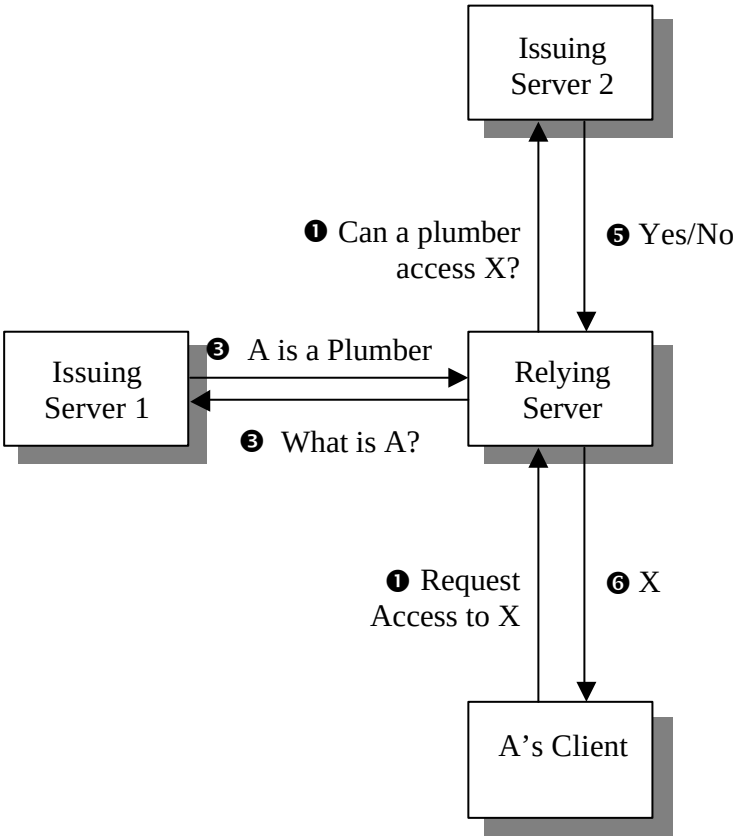

Figure 3 Configuration in which separate issuing servers address different levels of authorization

Alternative configurations may be employed to achieve the same objective:

- The relying server may direct all requests to a single internal resource that acts as an issuing server and processes referrals automatically.

- The relying server may be responsible for handling referrals itself, however the initial request might be 'Can A access X' and the response be 'Indeterminate, a *Plumber* can access X'

6.2 Multiple Relying Servers

In many circumstances a ticket must be acceptable to multiple relying servers Figure 2.

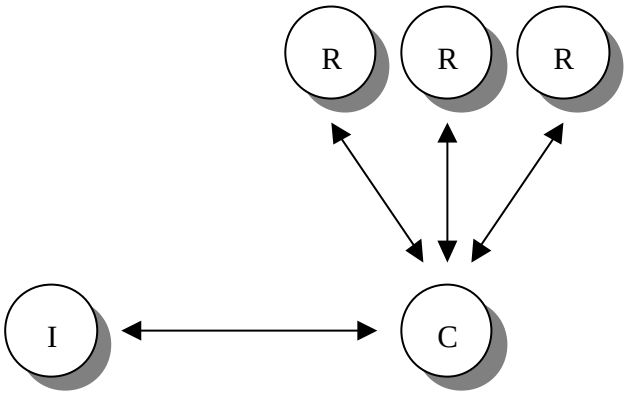

Figure 4: Multiple Relying Servers Rely on Assertion Issued by a Single Issuing Server

6.3 Issuing Server is the Relying Server

In many extant Web applications the same server performs the functions of the issuing server and the relying server. The first time the client visits a Web site it authenticates itself using username and password and receives a token (cookie, URL fragment) that is used to authenticate further access to the site.

The introduction of multiple servers supporting an application of this type leads to a need for an interoperable standard for exchanging security assertions between the servers Figure 4.

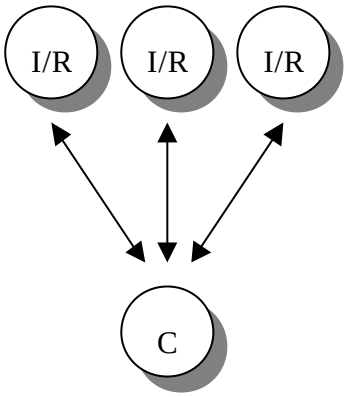

Figure 5: A Collection of Servers Act in Both Issuing and Relying Mode

7 References

[Kerberos]

[S2ML-Use]

[XTASS]

[XKMS]

8 Acknowledgements

Had discussions with Jeremy, Carlisle, Steve, Eve, Dave. Also based on use cases by Darren, Evan and co.

Appendix A Ticket Encoding Syntax

It may be argued that the ticket encoding syntax can be left to private agreement between servers.

- Allow encoding of any SAML assertion data element
- Place no arbitrary restrictions on the lengths of data objects
- Require minimal overhead, allowing a ticket to be encoded in 64 bytes of data
- Identify the version number of the ticket encoding

A.1 Self-terminating Integer Encoding

Fields representing data length and tag values are encoded using a simple self-terminating length encoding. In this encoding values are encoded as a sequence of octets. The most significant bit of the last octet in sequence is set, the most significant bit of each of the leading octets is clear. The value of the integer is encoded in the lower 7 bits of the octet sequence, with the first octet being the least significant.

Examples:

Integer	Data (hexadecimal)			
0	80			
1	81			
2	81			
127	FF			
128	00	81		
16383	7F	FF		= 127 + 128 * 127
2097151	7F	7F	FF	= 127 + 128 * 127 + 128^2 * 127

Data may be encoded using the following procedure:

```
Encode (integer v, out octet s[], out integer i)
  while (v > 127)
    s [i] = octet (v & 127)
    v = v / 128
    i = i + 1
  s [i] = (128 + v)
  i = i + 1
```

Data may be decoded using the following procedure:

```
Decode (octet s[], out integer v, out integer i)
  integer base = 1
  v = 0
  while (s[i] < 128)
    v = v + s[i] * base
    i = i + 1
    base = base * 128
  v = v + (s[i] – 128) * base
  i = i + 1
```

Additional code may be required to perform range checking if the language does not support integers of indefinite size.

A.2 Envelope Format

Field	Length (bytes.bits)	Max	Description
Version	0.4	0.4	Equals 0 for this version
Encryption Suite	0.4	0.4	0 = AES encryption with HMAC-SHA1

Key ID length	1	1	
Key ID	1	20	
Body length	1	2	
Body	22	256+	
Checksum length	1	1	
Checksum	12	20	
Total	39	301	

A.3 Body Data

Body data is encoded as a sequence of Tag, Length Data triplets where the tag values are specified as follows:

Tag Value	Description
0	SHA-1 hash of the assertion
1	Locator for assertion
2	Authenticated account identifier
3	Unauthenticated account identifier
4	Expiry date and time (format TBD)
5	Symmetric keying material
…	To be specified

Both tag and length are encoded using the self-terminating integer encoding

Examples:

```
8094 16E4 C8F6 681D C786 560B 9012 712C 602E 348F 39EE
```

Tag is 0 (SHA-1 hash of the assertion), Length is 20 bytes, and data is C8F6 39EE.

```
8285 "Alice"
```

Tag is 2 (Authenticated account identifier), Length is 5 bytes, and data is `Alice`.