OASIS Security Assertion Markup Language (SAML) Versioning Issues and Considerations

Draft 02, 6 March 2003

Document identifier:

draft-cantor-versioning-02

Location:

http://www.oasis-open.org/committees/security/docs/

Author:

Scott Cantor, The Ohio State University and Internet2 (cantor.2@osu.edu)

Contributors:

The author would like to acknowledge contributions to this document from several members of the Liberty Alliance Technical Expert Group, including but not limited to:

Jeff Hodges, John Kemp, Jonathan Sergent, Xavier Serret, Tom Wason

Their contributions do not imply agreement with any or all of the positions taken by this document.

Abstract:

This document defines useful terminology and explores some of the issues facing the committee and SAML implementers as the standard prepares to move beyond 1.0. Various possible approaches to versioning the standard and its components are discussed along with their implications as perceived by the author.

Status:

This is currently an individual submission that reflects contributions from the listed parties and other committee members, but does not reflect the consensus of the SSTC.

If you are on the security-services@lists.oasis-open.org list for committee members, send comments there. If you are not on that list, subscribe to the security-services-comment@lists.oasis-open.org list and send comments there. To subscribe, send an email message to security-services-comment-request@lists.oasis-open.org with the word "subscribe" as the body of the message.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Security Services TC web page (http://www.oasis-open.org/committees/security/).

Table of Contents

1 Introduction

This document discusses some of the considerations facing the SSTC during the transition from the 1.0 document set to 1.1 and beyond. Versioning of XML standards is not a particularly well understood process, (at least not by the author), and there are a significant number of different components of the standard which can affect and be affected by the versioning activity. The author is also an implementer of the standard, and is conscious of the effects that different approaches may have on the maintainability and compatibility of different implementation strategies. Finally, creating a set of definitions for somewhat informal descriptions of goals or characteristics of the versioning process will help more precisely communicate the intent of the committee.

Where there are established practices and definitions in place, they should be brought to the attention of the committee so that a fuller understanding and a better outcome might be achieved.

While versioning can be discussed in different contexts, the purpose of this document is to explore versioning on a technical basis for implementers and deployers, and not at the level at which political and marketing considerations may impact nomenclature.

Critical points being made will be highlighted in bold face for casual readers.

2 Key Questions and Assumptions

Before delving into terminology and detail, it's useful to lay the groundwork for evaluating some of the conclusions reached later by laying out a handful of key considerations that influence the problem space. What are the questions that must be answered? What are the critical decision points that lead to different conclusions?

- How do we define terms like "major version", "minor version", and "compatibility"?
- How minor is "minor"? Minor to whom? Implementers? Deployers? Specification authors?
- When XML is used to define protocols, are the schemas defined frozen at the time that implementations ship? Or should implementations understand and account for schema change?
- What are the security implications of schemas and schema evolution?
- Is a namespace change "minor" if the schema remains largely the same?
- Are parallel implementations a reasonable burden of minor version compatibility?
- Is runtime interoperability between new implementations and deployed implementations important? How important is it?

Some of these questions are answered directly or indirectly in the sections that follow. Many of these questions are subjective, and thus the author's opinion as to their answers leads to the conclusions reached in various places.

For clarity and context, I find myself proceeding from these general principles:

- Runtime interoperability is king. Robust evolution implies the freedom to evolve protocols without building numerous redundant implementations or requiring continuous redeployment.
- If XML is to be used to define protocols, then it should be used in a fashion consistent with its design and philosophy (if anybody can agree what that is, anyway).
- Namespace change is a big hammer that shouldn't be used to drive small nails.
- Schema change is a complex but ultimately powerful evolutionary tool, but it requires careful design and consideration, well-defined specifications (leading to predictable implementations), and security analysis, especially in a security specification.

I specifically disagree at least in part with these assumptions that I have encountered during discussion with others:

- Schemas cannot be modified at all once published without changing the namespace.
- A new schema copied from an existing schema into a new namespace (usually with some relationship between the namespace names) should be considered a minor "compatible" revision of the original schema.
- A specification incorporating two sets of schema and processing rules for the same basic information is backward compatible in a useful way with a specification containing one of the two sets.

Essentially, I take issue with those assumptions as not providing an especially useful framework to discuss anything other than what I perceive to be major revisions of a specification. They do serve such a purpose; however I think the scope of impact of the changes that would be permissible under those assumptions are extremely large, and violate the spirit of a minor revision. They also do not seem to me especially useful in scoping the specification changes that could be introduced in a minor revision, which is another purpose for framing a revision as minor instead of major.

Once a new namespace is introduced, all bets are off. Any change could be made without impacting the ability of an implementation to process older messages, because it essentially splits the implementation in two with respect to conformance. The best that might be said of such an approach is that there probably will be significant commonality of code if the new schema strongly resembles the old, but it's difficult to gauge that without also considering semantics.

3 Terminology and Implications

Where possible, agreeing to a set of common definitions will help clarify discussions within the committee and more clearly communicate the intent of the committee to the interested community. Guiding principles can be established for the versioning activity with reference to such definitions so that questions need not be answered as to why one approach or another was taken in deference to another. The committee might even see fit to publish a statement of intent to follow certain guidelines when advancing the specification so that future versions evolve in a consistent and predictable way. This will serve the interests of users and implementers alike, and may speed the evolutionary process.

In certain cases, it is useful to explore the implications of these definitions on each other and the versioning process.

3.1 Implementation Types

For the purpose of discussing the effect that various kinds of changes have on the implementations of an XML specification, it is useful to classify such implementations into categories that distinguish the degree and form of impact of those changes.

3.1.1 Validating

A validating implementation is one that applies XML schema validation to incoming messages before passing them to a higher level processing engine. This validation process does not necessarily rely on the W3C XML Schema validation process specifically; other schema languages exist and provide similar features. It merely implies that before further examination of messages, an automated processing step insures that the message fully conforms to the syntax required by the specification. Any violation of that syntax renders a message unrecognizable and in error.

A validating implementation is by definition unable to process messages defined by a newer version of a specification if the newer version adds any content, optional or mandatory, to any messages defined by the older version. The exception is in the event that the older version includes schema wildcard content placeholders that permit unknown content to appear, though care must be taken with respect to the namespace(s) in which that unknown content is placed.

3.1.2 Non-Validating

A non-validating implementation is one that does not apply XML schema validation to incoming messages before passing them to a higher level processing engine. It may or may not apply tests of well-formed-ness to incoming messages.

A non-validating implementation is obligated to accept any incoming message which adheres to the syntax defined by the specification. **It is impossible to know in the absence of additional information whether messages which deviate from that syntax will be accepted by a non-validating implementation.** However, unless such an implementation implements a large degree of manual processing that largely duplicates the work performed by a schema validator, it is unlikely that certain kinds of invalid messages would be detected and rejected. Whether this is a violation of the conformance rules defined by a specification depends on those rules. **Historically, the ability to process messages that take syntactic liberties with a specification has been deemed a virtue, and a sign of robustness.**

3.2 Compatibility

The primary purpose behind versioning at a technical level is to communicate to specification implementers and to the implementations themselves an expectation of compatibility (or of

incompatibility). The possibility of compatibility in the face of (and despite) change must exist in order for fine-grained versioning to make sense. Compatibility should be addressed at both syntactic and semantic levels, independently. **Different versioning mechanisms may address syntax, semantics, or both.**

3.2.1 Backward Compatibility

If we say that two versions of the specification are backward compatible, then the messages and/or semantics defined by the older version are consumable by schemas and implementations of the newer version.

However, a very important assumption that underlies much of the rest of this document is that it is not a reasonable definition of backward compatibility to presume that a new specification can simply incorporate any and all schema and processing rules of an older specification while subsequently redefining significant portions of that schema in a new namespace so that changes can be made. While such an implementation might be called backward compatible, functionally, the specification does not assist in the effort to remain compatible and is more properly termed a major revision that requires both the old and new versions to be implemented side by side.

3.2.2 Forward Compatibility

If we say that two versions of the specification are forward compatible, then the messages and/or semantics defined by the newer version are consumable by schemas and implementations of the older version.

3.3 Versions

A specification such as SAML can be described on several different levels, each having a potentially independent version, though the committee may choose to intrinsically link one or more of these versions so that they are revised in concert. **It is important to identify each of the different versioning mechanisms, and clarify which are intended to be independent and which are intended to reflect one another.**

3.3.1 Major and Minor Version

In most cases, the various versioning mechanisms will represent either formally (by explicitly distinguishing) or informally (using a conventional notation such as *major.minor*) the notion of both a major and minor version. This is common to many specifications and should connote the usual general intent. **However, the exact scope of changes that would constitute a minor revision seems vaguely defined in the XML arena.** Precise definitions of "compatibility" or "understanding a message" are hard to come by.

Major versions should represent fundamental changes to the information being versioned that do not imply a possibility of compatibility in syntax, semantics, or implementation. Higher major versions may be a superset or a subset of functionality present in lower major versions.

Minor versions should represent less significant changes to the information being versioned that imply specific expectations of compatibility. Higher minor versions must be a superset, and must be backward compatible with lower minor versions. Furthermore, higher minor versions should be forward compatible with lower minor versions to the greatest extent possible.

Most especially, an implementation must be able to treat a message with a higher minor version as though it were of a lower minor version, or be able to recognize explicitly when it cannot. Without this capability, there is much less advantage to maintaining minor version compatibility, and no effective difference between a major and minor revision beyond higher level concepts of change scope and a general sense that a minor revision should require fewer implementation changes than a major revision. Useful perhaps, but far less useful than the real runtime interoperability that some degree of forward compatibility offers.

For this to be possible, syntactic compatibility must be maintained throughout all minor version changes, and new semantics must be optional to implement and must either be optional to process or be communicated as required to process. Adding syntactic extensions with required semantics is only possible if the original version permits a syntax that can communicate required vs. optional semantics in a forward compatible way, such as the "mustUnderstand" attribute in **[SOAP]** or the Condition element processing rules in **[SAMLCore]**.

The effect of such a mechanism is to permit syntactically compatible but semantically incompatible extensions to be introduced, while maintaining well-defined behavior in older versions. Newer messages without mandatory-to-process extensions can then be processed by older implementations as though they were of the older version, satisfying the rule above.

3.3.2 Namespace Version

The most coarse (and somewhat implicit) versioning mechanism available to an XML specification is the namespace(s) in which the elements and attributes that make up the specification's XML syntax are placed. Namespaces are opaque strings to an XML processor. While it is common (though not universal) practice to include version information or date information in a namespace URI, such information is not used directly by an XML processor, and is only visible to an XML application in a manual fashion.

If a namespace in a specification is replaced by another, this should constitute a major version change to that part of the specification.

Note that namespaces by themselves were not originally formulated as a versioning mechanism. From an XML perspective, the element "foo" in two different (even similarly named) namespaces were intended to bear no relationship to one another. It seems that as schemas and data typing have become more pervasive, this picture has become muddier.

3.3.3 Schema Definition

If an XML schema, in whatever schema language, is defined as a normative part of a specification, then the syntax rules defined by that schema form the definition of the messages permitted by the specification. In most cases, a schema is bound permanently to a particular XML namespace. The namespace cannot be changed without effectively creating a new schema that is not related to the old one in XML terms. **Strictly speaking, a subsequent revision of the specification could choose to modify or add to that schema** (without changing the namespace, since that would constitute a replacement of the original schema). **If such modification is not permitted, schema evolution and forward compatibility become competing goals.**

It should be clear that if content is removed from the schema or if cardinalities decrease, it is likely that backward compatibility will not be possible. Further, any addition or increase in cardinality to a schema will break forward compatibility, unless the addition is a new message that does not relate to an older message or is used in a new way (as part of a new profile, for example).

3.3.4 Schema Version

A seemingly little-used feature of **[XSD]** is the "version" attribute that can be placed on the schema element in a schema definition. **There are *no* normative processing rules defined for an XML processor or a schema validator with respect to this attribute.** As an example, consider the version value placed in the normative schema defined for **[XSD]** itself, "Id: XMLSchema.xsd,v 1.48 2001/04/24 18:56:39 ht Exp". Suffice to say, this does not appear to be intended for consumption by any typical kind of versioning algorithm, apart from an identity test equivalent to a namespace comparison.

Further, consider that the value of the version attribute would not be used by a validating implementation during schema validation unless additional steps were taken to examine the schema; a non-validating implementation would quite likely never see such a value, since it is by definition not using the schema directly.

3.3.5 Message Version

A message version is defined as in-band content that identifies the major and/or minor version of a message. The version information is carried as content within the message, rather than as part of the message definition. Message versioning seems primarily useful as a way to communicate semantic distinctions between messages with a common syntax.

To see why, consider a strategy in which the message version is revised in concert with the message's primary namespace (i.e. the namespace of the root element of the versioned message). A typical implementation, whether validating or not, using either a SAX or DOM processing model, is likely to see the namespace before it has a chance to examine the message version, and thus can just as easily base any processing decisions on the namespace.

Since minor revisions must have some degree of common syntax to remain backward compatible, message versioning would seem to be a significant vehicle for indicating minor revisions. The other versioning mechanisms tend to imply syntactic change, and would generally be considered major revisions.

3.3.6 Specification Version

A specification version is applied by the specification's approving body to the set of normative syntactic and semantic rules that govern the messages defined by the specification. It may be reflected by the other kinds of versioning attached to the content of the specification, discussed in the previous sections, or it may be independent of them. In fact, many different versions of various types may coexist within a single specification.

Political and marketing considerations seem best suited for resolution with this kind of versioning. Ultimately, it has little or no technical impact and should not imply anything about the other version changes it might encompass. For example, a major revision technically might be marketed as a minor revision because little new functionality is introduced, merely corrections to technical problems, security holes, etc.

4 Schema Changes and Versioning

XML is of course designed to be extensible (duh!), a goal furthered (but also sometimes complicated) by the extension facilities described by **[XSD]**. Understanding the implications of different kinds of extension techniques on the versioning process is one of the most important pieces of the versioning puzzle. This might suggest guidelines that can be followed in deciding when and how to add extensions in subsequent specification versions.

The following set of examples describe a variety of potential changes to a specification and explore how those changes would seem to impact the specification, versioning, and implementations.

4.1 Adding a Global Element or Type

When adding a new globally visible (or root) element to a specification, the definition could be added to an existing namespace or a new namespace. Either approach would be backward compatible, but neither would be forward compatible. **Thus, either approach would constitute a minor revision of the specification.** With respect to implementations, neither a validating nor a non-validating implementation of the original version could process the new definition usefully, regardless of how the definition was added.

4.2 Adding a Required Element or Attribute to an Existing Type

If the existing type contains a schema wildcard that permits the addition of the new element or attribute in the location at which it is added, then this would be a forward compatible change. It would not, however, be backward compatible, since older messages would not carry the required information and would not be considered valid. **Therefore this cannot be considered a minor revision**. This holds regardless of what namespace is used to define the new element or attribute.

4.3 Adding an Optional Element or Attribute to an Existing Type

If the existing type contains a schema wildcard that permits the addition of the new element or attribute in the location at which it is added, then this would be a forward compatible change. It would always be backward compatible even without a wildcard, since older messages would not carry the new information and would still be considered valid. **At the syntactic level, then, this is a minor revision of the specification.**

However, an additional consideration must be whether the semantics of the optional information are mandatory or optional to implement. If the extension has optional semantics, then forward compatibility holds. If the extension has mandatory semantics, then forward compatibility does not hold. **Additionally, a mandatory extension would violate the rule that a message of a higher minor version be treatable as being of a lower minor version or identified as an error.** This should be held distinct from a case in which the message can communicate the semantics of an extension within the message, rather than relying on version information to do so, such as in **[SOAP]**. In such a case, while the message may not be usable by the older implementation, it can be recognized as being invalid without any knowledge of what the extension is.

Thus, adding an optional extension with optional semantics could be considered a minor revision, but adding one with required semantics in which the version is used to communicate those semantics could not be.

Consider as well, however, how implementations might react to such an extension. A validating implementation of the older version would be likely to reject any message that contained such an extension, unless the original schema permitted arbitrary extension via a wildcard. The newer message could not be processed as if it were an older one, and would be in error. **Thus, use of wildcards seems essential to permit minor revisions to add optional extensions in a useful way.** A non-validating

implementation is largely an unknown. It might be able to ignore the extension and still process the message, or it might find an incongruity that would cause it to reject the message, particularly if the extension were an element added in the middle of a content model. Adding optional attributes would probably not break a non-validating implementation.

4.4 Removing an Element or Attribute

Any time an existing piece of information is removed from the schema, backward compatibility cannot be maintained, since older messages would no longer be valid. **Therefore this cannot be considered a minor revision**.

4.5 Extending an Existing Complex Type

[XSD] permits various kinds of extensibility when defining types so that a schema can relate newer types to older types in a well-understood fashion, in part ostensibly as an aid to implementers. Extending a type is a different kind of change from directly modifying the content of an existing type, so it deserves specific examination.

In general, extending a complex type is not that different from defining a new stand-alone type. The extended type may be defined in an existing namespace or by defining a new one. **In either case, the effect on versioning is based more on how the type is to be used.** If the new type is a new top level message, then the discussion in section 3.1 is relevant. If the new element type is to be referenced specifically by the content model of an existing element, then this constitutes an addition, and sections 3.2 or 3.3 would apply, depending on the cardinality of the new element.

If the new type is intended to appear in place of an existing element of the base type, then the content model containing the base type is left unchanged. Thus, backward compatibility is maintained. Forward compatibility cannot be maintained, since the new type cannot be recognized by the older version; there is no way to identify the relationship between the new type and the base type.

This could still be considered a minor revision, however, because even if the new element type has mandatory semantics, it cannot be ignored by an older implementation, since it is not hidden inside a wildcarded content model, as is the case in section 3.3. Specifically, the version information is not used to communicate the element's mandatory semantics; the element itself does this job. As an example, see the "Condition" element base type in **[SAMLCore]**. **Further, if the new element type is intended for use in profiles or interactions that are not in the scope of the older version, the impact on older versions is likely to be minimal.**

4.6 Changing a Simple Type

When changing a simple type, such as an attribute's value type, compatibility seems to depend on a comparison of the value spaces of the old and new types. If the new type is a restriction of that value space (such as restricting a string into a URI), then the change is forward compatible, but is not backward compatible, and therefore is not a minor revision.

If the new type is an expansion or extension of that value space (such as adding to an enumeration or expanding a URI into a string), then the change is backward compatible but not forward compatible and could be considered a minor revision. However, such a minor revision would again lead to errors in older implementations rather than useful processing of the message. More seriously, in a non-validating implementation, one might imagine dangerous error conditions such as underflow or overflow leading to significant problems that are best avoided. **It seems prudent to avoid expansion of a simple type's value space in a minor revision.**

5 Security Considerations

Rather than a general security discussion, I would prefer to focus on an important consideration in the context of schema evolution. Schemas form a contract that implementations can follow (whether manually or using automated tools) to document assumptions about message syntax and semantics. The interpretation of any XML vocabulary is subject to those assumptions. Certainly in the context of SAML, for example, policy may well be enforced on the basis of the information in messages, and the audit trail of those messages may be significant.

It does not seem reasonable to require that messages contain a detached signature over the schema and/or specification documents on which they rely, nor has any use of XML schemas that I'm aware of been predicated on such a step. Some degree of trust is implied in the body that governs a schema, and it is certainly reasonable practice for an implementation to control and document the schema on which it relies.

Yet, it does not seem necessary to therefore conclude that once published, such a schema forms an unalterable contract. Just as contracts can be amended with the consent of the parties, so too should schemas be permitted to evolve in ways that are useful to the schema's consumers. This does not mean that all such changes are appropriate. The contract implied by the schema should be a two-way street. Syntax changes should be accommodated in the design of the schema, and if the semantics or default behaviors of the schema change, messages should be capable of signaling this explicitly through version changes.

Short of errata (and possibly not even then), it seems a strong likelihood that any change to a schema has to be reflected in some visible sign of evolution so that interoperability becomes a voluntary act, and not a consequence of successful parsing through happenstance. Let the implementation rely on the schema to the extent that it deems appropriate and let it react to changes as strictly as it wishes, without using code breakage as the switch.

6 A Modest Proposal

I would suggest that this document lays out a fairly precise starting point for discussing what one could and could not expect to change in a minor or major revision, provided a few basic guidelines are followed, though I acknowledge the guidelines may themselves be contentious, given the assumptions made previously in many areas.

My general thoughts:

- Message version information (in SAML, the MajorVersion and MinorVersion attributes) should be revised independently of any other versioning mechanism, and in lockstep with the specification version.
- Semantic-only changes should be reflected and documented by changes to the message version.
- Namespaces might contain version-oriented data in their names, but any change to a namespace name should be considered a major revision. That is, moving definitions into a new namespace from an old namespace would be reserved for major revisions.
- There should be no mandatory relationship between the namespace version and the message or specification version.
- Care should be taken when modifying definitions in an existing namespace, but this should not be uniformly outlawed. It is likely to be useful only in isolated cases, however, and requires a more liberal use of wildcards than exists in SAML 1.0.
- Avoid needless namespace creation when adding relatively orthogonal or forward compatible changes to the data model.
- Examine proposed schema changes in detail to understand their compatibility implications and the best strategy for implementing them based on the type of revision under discussion.

I believe these guidelines lead to a reasonable process for defining scopes of work, making technical decisions, and evolving schemas while minimizing code impact and maximizing the utility of any given implementation across revisions of a specification.

7 References

The following are cited in the text of this document:

[SAMLCore] Phillip Hallam-Baker et al., *Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML)*, http://www.oasis-open.org/committees/security/, OASIS, May 2002.

[SAMLBind] Prateek Mishra et al., *Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML)*, http://www.oasis-open.org/committees/security/, OASIS, May 2002.

[SOAP] Various., *Simple Object Access Protocol (SOAP) 1.1*, http://www.w3.org/TR/SOAP/, W3C Note, May 2000.

[XSD] David C. Fallside et al., *XML Schema*, http://www.w3.org/XML/Schema#dev, W3C Recommendation, May 2001.