Network Working Group
INTERNET-DRAFT
September 1999
Expires: March 2000
draft-ietf-trade-hiroshi-dom-hash-03.txt

Hirosi Maruyama
Kent Tamura
Naohiko Uramoto
IBM

Digest Values for DOM (DOMHASH)

Hiroshi Maruyama
Kent Tamura
Naohiko Uramoto

Status of This Document

This draft, file name draft-ietf-trade-hiroshi-dom-hash-03.txt, is intended to be become an Informational RFC. Distribution of this document is unlimited. Comments should be sent to the authors or the IETF TRADE working group mailing list <ietf-trade@elistx.com>.

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months. Internet-Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet-Drafts as reference material or to cite them other than as a ``working draft'' or ``work in progress.''

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

To view the entire list of current Internet-Drafts, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories as listed at <http://www.ietf.org/shadow.html>.

Abstract

This internet draft defines a clear and unambiguous definition of digest (hash) values of the XML objects regardless of the surface

string variation of XML. This definition can be used for XML digital signature as well efficient replication of XML objects.

Table of Contents

1. Introduction

The purpose of this document is to give a clear and unambiguous definition of digest (hash) values of the XML objects [XML]. Two subtrees are considered identical if their hash values are the same, and different if their hash values are different.

There are at least two usage scenarios of DOMHASH. One is as a basis for digital signatures for XML. Digital signature algorithms normally require hashing a signed content before signing. DOMHASH provides a concrete definition of the hash value calculation.

The other is to use DOMHASH when synchronizing two DOM structures [DOM]. Suppose that a server program generates a DOM structure which is to be rendered by clients. If the server makes frequent small changes on a large DOM tree, it is desirable that only the modified parts are sent over to the client. A client can initiate a request by sending the root hash value of the structure in the cache memory. If it matches with the root hash value of the current server structure, nothing needs be sent. If not, then the server compares the client hash with the older versions in the server's cache. If it finds one that matches the client's version of the structure, then it locates differences with the current version by recursively comparing the hash values of each node. This way, the client can receive only an updated portion of a large structure without requesting the whole thing.

One way of defining digest values is to take a surface string as the input for a digest algorithm. However, this approach has several drawbacks. The same internal DOM structure may be represented in may different ways as surface strings even if they strictly conform to the XML specification. Treatment of white spaces, selection of character encodings, entity references (i.e., use of ampersands), and so on have impact on the generation of a surface string. If the implementations of surface string generation are different, the hash values would be different, resulting in unvalidatable digital signatures and unsuccessful detection of identical DOM structures. Therefore, it is desirable that digest of DOM is defined in the DOM terms -- that is, as an unambiguous algorithm operating on a DOM tree. This is the approach we take in this specification.

Introduction of namespace is another source of variation of surface string because different namespace prefixes can be used for representing the same namespace URI [URI]. In the following example, the namespace prefix "edi" is bound to the URI "http://ecommerce.org/schema" but this prefix can be arbitrary chosen without changing the logical contents as shown in the second example.

```
<?xml version="1.0"?>
<root xmlns:edi='http://ecommerce.org/schema'>
    <edi:order>
        :
    </edi:name>
</root>


<?xml version="1.0"?>
<root xmlns:ec='http://ecommerce.org/schema'>
    <ec:order>
        :
    </ec:name>
</root>
```

The DOMHash defined in this document is designed so that the choice of the namespace prefix does not affect the digest value. In the above example, both the "root" elements will get the same digest value.

2. Digest Calculation

2.1. Overview

Hash values are defined on the DOM type Node. We consider the following four node types that are used for representing a DOM document structure:

1. Element
2. Attr
3. ProcessingInstruction
4. Text

Comment nodes and Document Type Definitions (DTDs) do not participate in the digest value calculation. This is because DOM does not require a conformant processor to create data structures for these. DOMHash is designed so that it can be computed with any XML processor conformant to the DOM or SAX [SAX] specification.

Nodes with the node type EntityReference must be expanded prior to digest calculation.

The digest values are defined recursively on each level of the DOM tree so that only a relevant part needs to be recalculated when a small portion of the tree is changed.

Below, we give the precise definitions of digest for these types. We describe the format of the data to be supplied to a hash algorithm using a figure and a simple description, followed by a Java code fragment using the DOM API and the JDK 1.1 Platform Core API only. Therefore, the semantics should be unambiguous.

As the rule of thumb, all strings are to be in UTF-16 in the network byte order (Big Endian) with no byte order mark. If there is a sequence of text nodes without any element nodes in between, these text nodes are merged into one by concatenating them. A zero-length text node is always ignored.

Note that validating and non-validating XML processors may generate different DOM trees from the same XML document, due to attribute normalization and default attributes. If DOMHash is to be used for testing logical equivalence between two XML documents (as opposed to DOM trees), it may be necessary to normalize attributes and supply default attributes prior to DOMHash calculation.

Some legacy character encodings (such as ISO-2022-JP) have certain ambiguity in translating into Unicode. This is again dependent on XML processors. Treatment of such processor dependencies is out of scope of this document.

2.2. Namespace Considerations

To avoid the dependence on the namespace prefix, we use "expanded names" to do digest calculation. If an element name or an attribute name is qualified either by a explicit namespace prefix or by a default namespace, the name's LocalPart is prepended by the URI of the namespace (the namespace name as defined in the NameSpace specification [NAM]) and a colon before digest calculation. In the following example, the default qualified name "order" is expanded into "http://ecommerce.org/schema:order" while the explicit qualified name "book:title" is expanded into "urn:loc.gov:books:title" before digest calculation.

```
<?xml version="1.0"?>

<root xmlns='http://ecommerce.org/schema'
         xmlns:book='urn:loc.gov:books'>
    <order>
       <book:title> ... </book:title>
        :
    </name>
</root>
```

We define an expanded name (either for element or attribute) as follows:

> If a name is not qualified, the expanded name is the name itself.
>
> If a name is qualified with the prefix "xmlns", the expanded name is undefined.
>
> If a name is qualified either by default or by an explicit namespace prefix, the expanded name is URI bound to the namespace + ":" + LocalPart

In the following example code, we assume that the getExpandedName() method (which returns the expanded name as defined above) is defined in both Element and Attr interfaces of DOM.

Note that the digest values are not defined on namespace declarations. In other words, the digest value is not defined for an attribute when

> - the attribute name is "xmlns", or
> - the namespace prefix is "xmlns".

In the above example, the two attributes which are namespace declarations do not have digest values and therefore will not participate in the calculation of the digest value of the "root" element.

2.3. Definition with Code Fragments

The code fragments in the definitions below assume that they are in implementation classes of Node. Therefore, a methods call without an explicit object reference is for the Node itself. For example, getData() returns the text data of the current node if it is a Text node. The parameter digestAlgorithm is to be replaced by an identifier of the digest algorithm, such as "MD5" [MD5] and "SHA-1" [SHA].

The computation should begin with a four byte integer that represents the type of the node, such as TEXT_NODE or ELEMENT_NODE.

2.3.1. Text Nodes

The hash value of a Text node is computed on the four byte header

followed by the UTF-16 encoded text string.

- TEXT_NODE (3) in 32 bit network-byte-ordered integer
- Text data in UTF-16 stream (variable length)

```
public byte[] getDigest(String digestAlgorithm) {
    MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
    md.update((byte)0);
    md.update((byte)0);
    md.update((byte)0);
    md.update((byte)3);
    md.update(getData().getBytes("UnicodeBigUnmarked"));
    return md.digest();
}
```

Here, MessageDigest is in the package java.security.*, one of the built-in packages of JDK 1.1.

2.3.2. ProcessingInstruction Nodes

A ProcessingInstruction (PI) node has two components: the target and the data. Accordingly, the hash is computed on the concatenation of both, separated by 'x0000'. PI data is from the first non white space character after the target to the character immediately preceding the "?>".

- PROCESSING_INSTRUCTION_NODE (7) in 32 bit network-byte-ordered integer

- PI target in UTF-16 stream (variable length)
- 0x00 0x00
- PI data in UTF-16 stream (variable length)

```
public byte[] getDigest(String digestAlgorithm) {
    MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
    md.update((byte)0);
    md.update((byte)(0);
    md.update((byte)0);
    md.update((byte)7);
    md.update(getName().getBytes("UnicodeBigUnmarked"));
    md.update((byte)0);
    md.update((byte)0);
    md.update(getData().getBytes("UnicodeBigUnmarked"));
    return md.digest();
}
```

2.3.3. Attr Nodes

The digest value of Attr nodes are defined similarly to PI nodes, except that we need a separator between the expanded attribute name and the attribute value. The '0x0000' value in UTF-16 is allowed nowhere in an XML document, so it can serve as an unambiguous separator. The expanded name must be used as the attribute name because it may be qualified. Note that if the attribute is a namespace declaration (either the attribute name is "xmlns" or its prefix is "xmlns"), the digest value is undefined and the getDigest() method should return null.

- ATTRIBUTE_NODE (2) in 32 bit network-byte-ordered integer
- Expanded attribute name in UTF-16 stream (variable length)
- 0x00 0x00
- Attribute value in UTF-16 stream (variable length)

```
  public byte[] getDigest(String digestAlgorithm) {
      if (getNodeName().equals("xmlns")
              || getNodeName().startsWith("xmlns:"))
          return null;
      MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
      md.update((byte)0);
      md.update((byte)0);
      md.update((byte)0);
      md.update((byte)2);
      md.update(getExpandedName().getBytes("UnicodeBigUnmarked"));
      md.update((byte)0);
      md.update((byte)0);
      md.update(getValue().getBytes("UnicodeBigUnmarked"));
      return md.digest();
  }
```

2.3.4. Element Nodes

Element nodes are the most complex because they consist of other nodes recursively. Hash values of these component nodes are used to calculate the node's digest so that we can save computation when the structure is partially changed.

First, all the attributes except for namespace declarations must be collected. This list is sorted by the expanded attribute names. The sorting is done in ascending order in terms of the UTF-16 encoded expanded attribute names, using the string comparison operator defined as String#compareTo() in Java. The semantics of this sorting operation should be clear.

- ELEMENT_NODE (1) in 32 bit network-byte-ordered integer

- Expanded element name in UTF-16 stream (variable length)
- 0x00 0x00
- A number of non-namespace-declaration attributes in 32 bit network-byte-ordered unsigned integer
- Sequence of digest values of non-namespace-declaration attributes, sorted by String#compareTo() for attribute names
- A number of child nodes (except for Comment nodes) in 32bit network-byte-ordered unsigned integer
- Sequence of digest values of each child node except for Comment nodes (variable length) (A sequence of child texts is merged to one text. A zero-length text and Comment nodes are not counted as child)

```
  public byte[] getDigest(String digestAlgorithm) {
      MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      DataOutputStream dos = new DataOutputStream(baos);
      dos.writeInt(ELEMENT_NODE);//This is stored in network byte order
      dos.write(getExpandedName().getBytes("UnicodeBigUnmarked"));
      dos.write((byte)0);
      dos.write((byte)0);
      // Collect all attributes except for namespace declarations
      NamedNodeMap nnm = this.getAttributes();
      int len = nnm.getLength()
              // Find "xmlns" or "xmlns:foo" in nnm and omit it.
      ...
      dos.writeInt(len);    // This is sorted in the network byte order
      // Sort attributes by String#compareTo() on expanded attribute names.
      ...
      // Assume that `Attr[] aattr' has sorted Attribute instances.
      for (int i = 0;  i < len;  i ++)
          dos.write(aattr[i].getDigest(digestAlgorithm));
      Node n = this.getFirstChild();
      // Assume that adjoining Texts are merged,
      // there is  no 0-length Text, and
      // comment nodes are removed.
      len = this.getChildNodes().getLength();
      dos.writeInt(len);    // This is stored in the network byte order
      while (n != null) {
          dos.write(n.getDigest(digestAlgorithm));
          n = n.getNextSibling();
      }
      dos.close();
      md.update(baos.toByteArray());
      return md.digest();
  }
```

3. Discussion

The definition described above can be efficiently implemented with any XML processor that is conformant to either DOM and SAX specification. Reference implementations are available on request.

4. Security Considerations

DOMHASH is expected to be used as the basis for digital signatures and other security and integrity uses. It's appropriateness for such uses depends on the security of the hash algorithm used and inclusion of the fundamental characteristics it is desired to check in parts of the DOM model incorporated in the digest by DOMHASH.

References

[DOM] - "Document Object Model (DOM), Level 1 Specification", October 1998, http://www.w3.org/TR/REC-DOM-Level-1/

[MD5] - RFC 1321 - R. Rivest, "The MD5 Message-Digest Algorithm", April 1992.

[NAM] - Tim Bray, Dave Hollander, Andrew Layman, "Namespaces in XML", http://www.w3.org/TR/1999/REC-xml-names-19990114.

[SAX] - David Megginson, "SAX 1.0: The Simple API for XML", http://www.megginson.com/SAX/, May 1998.

[SHA] - (US) National Institute of Standards and Technology, "Federal Information Processing Standards Publication 180-1: Secure Hash Standard", 17 April 1995.

[URI] - RFC 2396 - T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", August 1998.

[XML] - Tim Bray, Jean Paoli, C. M. Sperber-McQueen, "Extensible Markup Language (XML) 1.0", http://www.w3.org/TR/1998/REC-xml-19980210

Author's Address

Hiroshi Maruyama,
IBM Research, Tokyo Research Laboratory
email: maruyama @ jp.ibm.com

Kent Tamura,
IBM Research, Tokyo Research Laboratory
email: kent @ trl.ibm.co.jp

Naohiko Uramoto,
IBM Research, Tokyo Research Laboratory
email: uramoto @ jp.ibm.com

Expiration and File Name

This draft expires March 2000.

Its file name is draft-ietf-trade-hiroshi-dom-hash-03.txt.