Frequently Asked Questions - C++

Questions

Answers

1. Compiling and Using the Library

1.1. Is OpenSSL required?

The main development work for the library is done using OpenSSL, so this is the recommended option. However, a Windows Crypto API interface is also now provided.

It is also possible to implement interfaces for other cryptographic libraries and pass them into the xml-security-c library during initialisation (via the *XSECPlatformUtils::Initialise()* call).

1.2. Does the library provide a full C++ wrapper for OpenSSL?

The C++ crypto interface layer provided for the library provides only the smallest subset of cryptographic functions necessary for the library to make calls to the provided library. Applications will need to work directly with OpenSSL (or other libraries) to read and manipulate encryption keys that should then be wrapped in XSECCrypto* objects and passed into the library.

1.3. What is WinCAPI?

WinCAPI is the developmental interface being built to give users of the library access to the Windows Cryptographic library.

It is *not* a C API wrapper for the overall library.

1.4. Is Xalan required?

The library can be compiled without linking to Xalan-c. However doing so will disable support for XPath and XSLT transformations.

To disable Xalan-c support either use --without-xalan when running configure on UNIX, or use the VC++ "without Xalan" settings.

1.5. Are versions of Xalan prior to 1.6 supported?

No. Whilst the functionality required is available in prior versions, the location of include files changed in 1.6. A decision was made in version 1.0.0 of xml-security-c to update the source to support these new locations.

1.6. I sign a document and when I try to verify using the same key, it fails

After you have created the XMLSignature object, before you sign the document, you *must* embed the signature element in the owning document (which is returned by the call to `DSIGSignature::createBlankSignature(...)`) before calling the `DSIGSignature::sign()` method,

During canonicalisation of the SignedInfo element, the library looks at the parent and ancestor nodes of the Signature element to find any namespaces that the SignedInfo node has inherited. Any that are found are embedded in the canonical form of the SignedInfo. (This is not true when Exclusive Canonicalisation is used, but it is still good practice to insert the element node prior to the sign() method being called).

If you have not embedded the signature node in the document, it will not have any parent or ancestor nodes, so it will not inherit their namespaces. If you then embed it in the document and call `verify()`, the namespaces will be found and the canonical form of SignedInfo will be different to that generated during `sign()`.

1.7. How does the library identify Id attributes?

During a signing operation, finding the correct Id attribute is vital. Should the wrong Id Attribute be used, the wrong part of the document will be identified, and what the user signs will not be what they expect to sign.

The preferred method (and the method the library uses first) of finding an Id is via the DOM Level 2 call *DOMDocument::getElementById()*. This indicates to the library that the Id has

been explicitly identified via a schema, DTD or during document building. However, if this call fails, the library will then search the document for attributes named "Id" or "id" with the appropriate value. The first one found will be used as document fragment identifier.

As this is a potential security exposure, this behaviour can be disabled using a call to *DISGSignatures::setIdByAttributeName(false)*. There are also methods provided to modify the list of attributes that will be searched. However it is recommended that these methods not be used, and DOM attributes of Type=ID be used.

Note:

In version 1.1, the library defaults to searching for Id attributes by name if a search by Id fails. As this is a potential security risk, this behaviour may be changed in a future version of the library.