ISO/IEC JTC 1/SC 22/WG 23 N 0357

Preliminary working draft, “Code Signing for Source Code”

Date	7 September 2011
Contributed by	Larry Wagoner
Original file name	prelim_WD_code_signing_090611.doc
Notes	Replaces N0318

The following is a preliminary working draft related to a New Work Item Proposal which has not yet been approved. It is offered as an illustration of what the proposed project might produce.

Strawman INTERNATIONAL STANDARD

ISO/IEC xxxxx

Information technology—Programming languages, their environments and system software interfaces—Code signing for source code

1. Scope

This document uses a language and environment neutral description to define the application program interfaces (APIs) and supporting data structures necessary to support the signing of code and executables. It is intended to be used by both applications developers and systems implementers.

The following areas are outside the scope of this specification:

- Graphics interfaces
- Object or binary code portability
- System configuration and resource availability

2. Normative References

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14750:1999, Information technology -- Open Distributed Processing -- Interface Definition Language

3. Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

[TBD]

4. Conformance

An implementation of code signing conforms to this International Standard if it provides the interfaces specified in Clause 6.

Clause 5 is informative, providing an overview of the concepts of code signing. Annex A, also informative, provides a possible scenario of usage for the interfaces specified in Clause 6.

5. Concepts

Code signing is the process of digitally signing scripts and executable objects that verifies the author or origin and guarantees that the signed code has not been tampered with or corrupted since it was signed by use of a cryptographic hash.

Code signing provides several valuable functions,

- code signing can provide security when deploying,
- code signing can provide a digital signature mechanism to verify the identity of the author or build system,
- code signing can provide multi signatures, allowing an audit trail of the signed object,
- code signing will provide a checksum to verify that the object has not been modified,
- code signing can provide versioning information, and
- code signing can store other meta data about an object.

Code Signing identifies to customers the responsible party for the code and confirms that it has not been modified since the signature was applied. In traditional software sales where a buyer can physically touch a package containing software, the buyer can confirm the source of the application and its integrity by examining the packaging. However, most software is now procured via the Internet. This is not limited to complete applications as code snippets, plug-ins, add-ins, libraries, methods, drivers, etc. are all downloaded over the Internet. Verification of the source of the software is extremely important since the security and integrity of the receiving systems can be compromised by faulty or malicious code. In addition to protecting the security and integrity of the software, code signing provides authentication of the author, publisher or distributor of the code, and protects the brand and the intellectual property of the developer of the software by making applications uniquely identifiable and more difficult to falsify or alter.

When software (code) is associated with a publisher's unique signature, distributing software on the Internet is no longer an anonymous activity. Digital signatures ensure accountability, just as a manufacturer's brand name ensures accountability with packaged software. Distributions on the Internet lack this accountability and code signing provides a means to offer accountability. Accountability can be a strong deterrent to the distribution of harmful code. Even though software may be acquired or distributed from an untrusted site or a site that is unfamiliar, the fact that it is written and signed by someone known and trusted allows the software to be used with confidence.

Multiple signatures for one piece of code would be needed in some cases in order to create a digital trail through the origins of the code. Consider a signed piece of code. Someone should be able to modify a portion of the code, even if just one line or even one character, without assuming responsibility for the remainder of the code. A recipient of the code should be able to identify the responsible party for each portion of the code. For instance, a very trustworthy company A produces a driver. Company B modifies company A’s driver for a particular use. Company B is not as trusted or has an unknown reputation. The recipient should be able to determine exactly what part of the code originated with company A and what was added or altered by company B so as to be able to concentrate their evaluation on the sections of code that company B either added or altered. This necessitates a means to keep track of the modifications made from one signature to the next.

An alternative scenario is software offered by company B that contains software from company A. Company B does not alter company A's software, but incorporates it into a package or suite of software. It would be useful to a customer to be able to identify the origin of each portion of the software.

6. Structures and APIs

6.1 General

The APIs described below are intended to be language and platform independent. A particular language implementation will need to specify, for instance, an appropriate convention for specifying options and determine how error reporting will be done.

The APIs are described with a syntax independent of any particular programming language, using the Interface Description Language (IDL) provided by ISO/IEC 14750:1999.

Note: the APIs are expressed using camel case (e.g. *isIntTrue* instead of underscores *is_int_true*). Particular language implementations may prefer to implement the APIs using underscores. Either is acceptable as long as the implementation is consistent within the language implementation.

6.2 certCreate

Notional Syntax

boolean certCreate (string certificateFile, string certificateDirPath)

Description

CertCreate creates in the directory *certificateDirPath* the file *certificateFile* that contains a certificate that complies with ITU-T X.509.

Returns

CertCreate returns TRUE if the certificate was successfully created and FALSE otherwise.

Errors

If the *certificateFile* cannot be created, *CertCreate* will report an error.

If *certificateDirPath* is an invalid path, *CertCreate* will report an error.

6.3 certSignCode

Notional Syntax

boolean certSignCode (certStruct myCertificate, keyStruct myPrivateKey, string sourceFilename, string sourceDirPath, boolean overwriteCurrentSignature, enum hashType signatureAlgorithm, string signFilename, string signDirPath)

Description

CertSignCode generates a digital signature (encrypted hash) of the source code file *sourceFilename* in directory *sourceDirPath* using public certificate *myCertificate* and private key *myPrivateKey*. The default hashing algorithm for signing shall be SHA-1. Alternative hashing functions that are specified in ISO/IEC 10118:2004 could be used instead and would be indicated through the enumerated type *signatureAlgorithm*. The digital signature and publisher's certificate are stored in the directory *signDirPath* in the file *signFilename*. By convention, the signature filename *signFilename* should be of the form "filename.ds". If *signFilename* already exists in the directory *signDirPath*, then *overwrite* must be set to TRUE or *certSignCode* will return an error that the file could not be created since it already exists.

Returns

CertSignCode returns TRUE if the digital signature was successfully created and FALSE otherwise.

Errors

If *signFilename* exists and *overwrite* is FALSE, *certSignCode* will report that the signature operation could not be completed since sign*Filename* already exists.

If *myCertificate* or *myPrivateKey* are in an unknown format or do not contain proper keys, *certSignCode* will report that the signature operation could not be completed since a key could not be read or used.

6.4 certSignWrap

Notional Syntax

boolean certSignWrap (certStruct myCertificate, keyStruct myPrivateKey, string originalSourceFilename, string originalSourceDirPath, string modifiedSourceFilename, string modifiedSourceDirPath, enum hashType signatureAlgorithm, string signFilename, string signDirPath)

Description

Incorporates changes to the previously signed file *originalSourceFilename* in directory *originalSourceDirPath* in such a way that the changes can be unwrapped at a later date in order to revert to a previously signed version. *CertSignWrap* generates a digital signature (encrypted hash) of the source code file *modifiedSourceFilename* in directory modifiedSourceDirPath using public certificate *myCertificate* and private key *myPrivateKey*. The default hashing algorithm for signing shall be SHA-1. Alternative hashing functions that are specified in ISO/IEC 10118:2004 could be used instead and would be indicated through the enumerated type *signatureAlgorithm*. The digital signature, publisher's certificate and changes between the current version and the previous version are added to the file *signFilename* in directory *signDirPath*.

Returns

CertSignWrap returns TRUE if the signature was successfully created and FALSE otherwise.

Errors

If a signature for *originalSourceFilename* does not exist, *certSignWrap* will report that the signature wrapping could not be completed because a signature does not exist and that a signature file would need to be created before the operation could be completed.

If there are no differences between the contents of *originalSourceFilename* and *modifiedSourceFilename*, *certWrap* will report that the signature operation could not be completed since there have not been any changes to the source code file.

If the hash of *originalSourceFilename* does not match the encrypted hash stored within *originalFile.ds*, *certSignWrap* will report that the *originalFile* differs from the file which was signed and that the signature operation could not be completed.

6.5 certHash

Notional Syntax

boolean certHash (string sourceFilename, string sourceDirPath, enum hashType signatureAlgorithm)

Description

CertHash generates a digital finger print (hash) of the source code contained in file *sourceFilename* in directory *sourceDirPath*. The default hashing algorithm for signing shall be SHA-1. Alternative hashing functions that are specified in ISO/IEC 10118:2004 could be used instead and would be indicated through the enumerated type *signatureAlgorithm*.

Returns

CertHash returns TRUE if the hash was successfully generated and FALSE otherwise.

Errors

TBD

6.6 certDecryptSignature

Notional Syntax

boolean certdecryptsignature (certStruct myCertificate, keyStruct myPrivateKey, string signFilename, string signDirPath)

Description

CertDecryptSignature decrypts the digital signature of the source code file contained in *signFilename* using *myCertificate* and *myPrivateKey*.

Returns

CertDecryptSignature returns TRUE if the digital signature was successfully decrypted and FALSE otherwise.

Errors

If the signature file *signFilename* does not exist, *certDecryptSignature* will report that the signature could not be verified because the signature file is missing.

If the signature file exists yet does not contain the properly formatted signature and public key components, *certDecryptSignature* will report that the signature file is corrupt.

6.7 certVerifySignature

Notional Syntax

boolean certVerifySignature (certStruct myCertificate, keyStruct myPrivateKey, string signFilename, string signDirPath)

Description

CertVerifySIgnature verifies the latest digital signature of the source code file sign*Filename* in directory *signDirPath* is valid and returns either an indication that the “signature is valid” or “signature is not valid”. This accomplishes in one step what *certHash()* and *certDecryptSignature()* do in multiple steps. Note that the hashing algorithm is inferred by the length of the signed hash and thus need not be specified by the user.

Returns

CertVerifySignature returns TRUE if the signature is valid and FALSE otherwise.

Errors

If the signature file does not exist, *certVerifySignature* will report that the signature file is missing.

If the signature file exists but does not contain the properly formatted signature and public key components, *certVerifySignature* will report that the signature file is corrupt.

6.8 certUnwrap

Notional Syntax

boolean certUnwrap (string signatureFile, string signatureFileDirPath, string sourceFilename, string sourceDirPath, string newSignatureFile, string newSignatureDirPath, string newSourceFilename, string newSourceDirPath)

Description

CertUnwrap reverts a previously signed file to the last previously signed version. *CertUnwrap* will remove the most recent signature for *sourceFilename* in *sourceDirPath* from the file *signatureFile* in directory *signatureFileDirPath* and the most recent set of changes in order to revert to the next most recent signature and file. If *newSignatureFile* and newSignatureFileDirPath are non-NULL, certUnwrap places modified the signature file in *newSignatureFile* inside directory newSignatureDirPath instead of modifying the contents of *signatureFile*. If *sourceFilename* and sourceDirPath non-Null, then the unwrapped file contents are placed in *sourceFilename* in *sourceDirPath*.

After the operation is complete, the user should run *certverifysignature* to ensure the files they are viewing is the previous version of source code and has a valid signature.

Returns

CertUnwrap returns TRUE if the unwrapping was successful and FALSE otherwise.

Errors

If the signature file does not contain a valid signature or is missing any components such as certificates or file differences, *cerUnwrap* will report that the unwrap operation could not be completed.

If only one of *newSignatureFile* and *newSignatureFileDirPath* is NULL, an error is generated.

If only one of *sourceFilename* and *sourceDirPath* is NULL, an error is generated.

Annex A

(Informative)

A possible method of operation

This annex describes one possible way of using the interfaces specified in Clause 6 of this International Standard.

1. **Publisher obtains a Code Signing Digital ID (Software Publishing Certificate) from a global certificate authority**

   (how one obtains a Code Signing Digital ID may be out of scope and might be better left to other standards bodies such as the World Wide Web Consortium (W3C))

   A software publisher's request for certification is sent to the Certification Authority (CA). It is expected that the CAs will have Web sites that walk the applicant through the application process. Applicants will be able to look at the entire policy and practices statements of the CA. The utilities that an applicant needs to generate signatures should also be available.

   Digital IDs can be either issued to a company or an individual. In either case, the global certificate authority must validate the identification of the company and applicant. Validation for applicants would be in the form of a federally issued identification for applicants and a Dun & Bradstreet number. Tables 1 and 2, respectively, contain the criteria for a commercial and individual code signer.

   Proof of identification of an applicant must be made. Simply trusting the applicant’s ID via a web site is insufficient. Additional verification of the applicant’s ID should be commensurate with the application process for a federally issued ID, such as a passport. Sending in a federally issued ID, such as a passport, to the CA would be sufficient for proof of identification.

   The applicant must generate a key pair using either hardware or software encryption technology. The public key is sent to the CA during the application process. Due to the identity requirements, the private key must be sent by mail or courier to the applicant.

Identification	Applicants must submit their name, address, and other material along with a copy of their federally issued id that proves their identity as corporate representatives. Proof of identify requires either personal presence or registered credentials.

Agreement	Applicants must agree to not distribute software that they know, or should have known, contains viruses or would otherwise harm a user's computer or code.
Dun & Bradstreet Rating	Applicants must achieve a level of financial standing as indicated by a D-U-N-S number (which indicates a company's financial stability) and any additional information provided by this service. This rating identifies the applicant as a corporation that is still in business. (Other financial rating services are being investigated.) Corporations that do not have a D-U-N-S number at the time of application (usually because of recent incorporation) can apply for one and expect a response in less than two weeks.

Table 1: Criteria for Commercial Code Publishing Certificate

Identification	Applicants must submit their name, address, and other material along with a copy of their federally issued id that proves their identity as citizens of the country where they reside. Information provided will be checked against an independent authority to validate their credentials.
Agreement	Applicants must agree that they cannot and will not distribute software that they know, or should have known contains viruses or would otherwise maliciously harm the user's computer or code.

Table 2: Criteria for Individual Code Publishing Certificate

2. **Publisher develops code or modifies previously signed code**

3. **Calculate a hash of the code and create a new file containing the encrypted hash, the publisher's certificate and the code**

A one-way hash of the code is produced using *certsigncode*, thereby signing the code. The hash and publisher's certificate are inserted stored in a separate file.

In order to be able to verify the integrity of previously signed code, it must be possible to identify the responsible party for each section of code. When new code modifies or in some way encapsulates previously signed code, the original code must be able to be identified so that its signature can be checked. Therefore, iterative changes to code must be able to be reversed to identify previously signed versions.

4. **The digitally signed file is transmitted to the recipient**

5. **The recipient produces a one-way hash of the code**

6. **Using the publisher's public key contained within the publisher's Digital ID and the digital signature algorithm, the recipient browser decrypts the signed hash with the sender's public key**

7. **The recipient compares the two hashes**

   If the signed hash matches the recipient's hash, the signature is valid and the document is intact and hasn't been altered since it was signed.

   Software that has multiple signings must be able to be "unwrapped" in order to recreate previously signed versions. Iterative changes to code can be reversed to identify previously signed versions through the use of *certunwrap*.

Bibliography

Code-Signing Best Practices, http://msdn.microsoft.com/en-us/windows/hardware/gg487309.aspxJuly 25, 2007

Code Signing Certificate FAQ, http://www.verisign.com/code-signing/information-center/certificates-faq/index.html, 2011

Code Signing for Developers - An Authenticode How-To, Tech-Pro.net, http://www.tech-pro.net/code-signing-for-developers.html, 2011.

Oliver Goldman, *Code Signing in Adobe AIR*, Dr. Dobb's, September 1, 2008.

How Code Signing Works, https://www.verisign.com/code-signing/information-center/how-code-signing-works/index.html , 2011.

Introduction to Code Signing, http://msdn.microsoft.com/en-us/library/ms537361(VS.85).aspx, June 21, 2011.

Steve Mansfield-Devine, *A Matter of Trust*, Network Security, Vol 2009, Issue 6, June 2009.

Regina Gehne, Chris Jesshope, Jenny Zhang, *Technology Integrated Learning Environment: A Web-based Distance Learning System*, AI-ED'95, 7th World Conference on Artificial Intelligence in Education, 2001.

Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine, *Survivable Key Compromise in Software Update Systems*, The 17th ACM Conference on Computer and Communications Security, 2010.

Deb Shinder, *Code Signing: Is it a Security Feature?*, WindowSecurity.com, http://www.windowsecurity.com/articles/Code-Signing.html?printversion ,June 9, 2005.