Contents

Changes made to the 05-Nov-1999 version:
1. Description of ECDSA calculation (Appendix B)
2. Addition of an SCR (3.1, Appendix D)
3. Corrected authenticated attributes templates; added template for random nonce usage (6.2)
4. Character set clarification (5.1.2)
5. Clarification regarding hash input (6)

1. Scope

Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope for the WAP Forum is to define a set of standards to be used by service applications. The wireless market is growing very quickly and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and fast/flexible service creation, WAP defines a set of protocols in transport, session and application layers. For additional information on the WAP architecture, refer to *Wireless Application Protocol Architecture Specification* [WAPARCH].

This document specifies the library interface for WMLScript [WMLScript] to provide cryptographic functionality of a WAP client. In addition this document specifies a signed content format to be used to convey signed data to/from WAP devices. This functionality complements transport layer security provided by [WAPWTLS].

The notation and other conventions related to describing a WMLScript library are according to [WMLScript] and [WMLSSL].

2. Document Status

This document is available online in the following formats:

- PDF format at http://www.wapforum.org/.

2.1 Copyright Notice

2.2 Errata

Known problems associated with this document are published at http://www.wapforum.org/.

2.3 Comments

Comments regarding this document can be submitted to the WAP Forum in the manner published at http://www.wapforum.org/.

3. References

3.1 Normative references

[ASN1] ISO/IEC 8824-1:1995 Information technology – Abstract Syntax Notation One (ASN.1) – Specification of basic notation.

[DER] ISO/IEC 8825-2:1995 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).

[ECMA262] Standard ECMA-262: "ECMAScript Language Specification", ECMA, June 1997

[IEEE754] ANSI/IEEE Std 754-1985: "IEEE Standard for Binary Floating-Point Arithmetic". Institute of Electrical and Electronics Engineers, New York (1985).

[PKCS1] PKCS #1: RSA Encryption Standard", version 1.5, RSA Laboratories, November 1993.

[PKCS7] PKCS #7: Cryptographic Message Syntax Standard, version 1.5, RSA Laboratories, November 1993.

[PKCS9] PKCS #9: Selected Attribute Types, version 1.1, RSA Laboratories, November 1993.

[PKCS15] PKCS #15: Cryptographic Token Information Standard", version 1.0, RSA Laboratories, April 1999. URL: ftp://ftp.rsa.com/pub/pkcs/pkcs-15/pkcs15v1.doc

[RFC1521] "MIME (Multipurpose Internet Mail Extensions), Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", N. Borenstein, et al, September 1993. URL: ftp://ftp.isi.edu/in-notes/rfc1521.txt

[RFC1738] "Uniform Resource Locators (URL)", T. Berners-Lee, et al., December 1994. URL: ftp://ftp.isi.edu/in-notes/rfc1738.txt

[RFC1808] "Relative Uniform Resource Locators", R. Fielding, June 1995. URL: ftp://ftp.isi.edu/in-notes/rfc1808.txt

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. URL: ftp://ftp.isi.edu/in-notes/rfc2119.txt

[RFC2459] "Internet X.509 Public Key Infrastructure, Certificate and CRL Profile", R. Housley, at al., January 1999. URL: ftp://ftp.isi.edu/in-notes/rfc2459.txt

[RFC2560] "X.509 Internet Public Key Infrastructure: Online Certificate Status Protocol – OCSP", M. Myers, R. Akney, A. Malpani, S. Galperin, and C. Adams; IETF RFC 2560, June 1999.

[UNICODE] "The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. URL: http://www.unicode.org/

[UTF8] "UTF-8, a transformation format of Unicode and ISO 10646", F. Yergeau, January 1998. URL:ftp://ftp.isi.edu/in-notes/rfc2279.txt

[WAPARCH] "Wireless Application Protocol Architecture Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[WAPWIM] "WAP Identity Module", WAP-260-WIM, WAP Forum Ltd. URL:http//www.wapforum.org/

[WAPWTLS] "Wireless Transport Layer Security", WAP-261-WTLS, WAP Forum Ltd. URL: http://www.wapforum.org/

[WML] "Wireless Markup Language Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[WMLScript] "WMLScript Language Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/
[WMLSSL] "WMLScript Standard Libraries Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/
[WAPCREQ] "Specification of WAP Conformance Requirements", WAP-221-CREQ, WAP Forum Ltd, URL:http//www.wapforum.org/"
[X9.62] “The Elliptic Curve Digital Signature Algorithm (ECDSA)”, ANSI X9.62 Working Draft, September 1998.

3.2 Informative References

[JavaScript] "JavaScript: The Definitive Guide", David Flanagan. O'Reilly & Associates, Inc. 1997
[RFC2068] "Hypertext Transfer Protocol - HTTP/1.1", R. Fielding, et al., January 1997. URL: ftp://ftp.isi.edu/in-notes/rfc2068.txt
[WAE] "Wireless Application Environment Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/
[WSP] "Wireless Session Protocol", WAP Forum, 1998. URL: http://www.wapforum.org/
[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210", T. Bray, et al, February 10, 1998. URL: http://www.w3.org/TR/REC-xml

4. Definitions and Abbreviations

4.1 Definitions

The following are terms and conventions used throughout this specification.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Please refer to [WMLScript] and [WMLSS] for WMLScript related terminology.

4.2 Abbreviations

For the purposes of this specification, the following abbreviations apply:

API	Application Programming Interface
CA	Certification Authority
ECMA	European Computer Manufacturer Association
HTTP	HyperText Transfer Protocol [RFC2068]
LSB	Least Significant Bits
MSB	Most Significant Bits
PKCS	Public-Key Cryptography Standards
RFC	Request For Comments
RSA	Rivest Shamir Adleman public key algorithm
SHA	Secure Hash Algorithm
UI	User Interface
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
WSP	Wireless Session Protocol
WTLS	Wireless Transport Layer Security
WTP	Wireless Transport Protocol
WAP	Wireless Application Protocol
WAE	Wireless Application Environment
WTA	Wireless Telephony Applications
WTAI	Wireless Telephony Applications Interface
WBMP	Wireless BitMaP
WIM	WAP Identity Module

5. Cryptographic Library Description

Name: Crypto

Library ID: 6

Description: This library contains cryptographic functions.

The current library specification supports digital signature functionality . Other functionality (like encryption/decryption or symmetric key based MAC) may be added in future versions.

5.1 signText

5.1.1 Introduction

Many kinds of applications, e.g., electronic commerce, require the ability to provide persistent proof that someone has authorised a transaction. Although WTLS [WAPWTLS] provides transient client authentication for the duration of a WTLS connection, it does not provide persistent authentication for transactions that may occur during that connection. One way to provide such authentication is to associate a digital signature with data generated as the result of a transaction, such as a purchase order or other financial document.

To support this requirement, the browser provides a WMLScript function, Crypto.signText, that asks the user to sign a string of text. A call to the signText method displays the exact text to be signed and asks the user to confirm that. After the data has been signed and both the signature and the data have been sent across the network, the server can extract the digital signature and validate it, and possibly store it for accountability purposes.

The browser SHOULD use special signature keys that are distinct from authentication keys used for WTLS. A WIM [WAPWIM] may be used for private key storage and signature computation.

5.1.2 signText function definition

Function: *signedString* = Crypto.signText(*stringToSign, options, keyIdType, keyId*)

Function ID: 16

Description: The function requests that a user digitally signs a text string. The calling script provides the text to sign (*stringToSign*) which MUST be displayed to the user. The user may choose either to cancel or approve the signing operation. If several certificates are available that match the criteria indicated in parameters, the choices should be indicated to the user, using e.g., labels of the certificates. If the user approves the operation, the browser MUST ask for user verification information for the private key (e.g., the WIM PIN for a non-repudiation key). If the user enters the correct information, signText signs the specified string and returns *signedString* to the script as String, formatted as base-64 [RFC1521] encoding of `SignedContent`.

Parameters: *stringToSign* = String

A string which MUST be displayed to the user. In case the string is a concatenation of strings in different character sets, the implementation has to convert the string to a certain single encoding, before processing in this function. The recommended encoding method is UTF-8 [UTF8].

options = Integer

Contains several option values, ORed together:

0x0001 – INCLUDE_CONTENT. If this option is set, the browser MUST include the *stringToSign* in the result.

0x0002 – INCLUDE_KEY_HASH. If this option is set, the browser MUST include the hash of the public key corresponding to the signature key in the result.

0x0004 – INCLUDE_CERTIFICATE. If this option is set, the browser MUST include the certificate or a URL of the certificate in the result (whether the browser includes the certificate content or a URL depends on which one is available). If the browser does not have access to a certificate, it MUST return “error:noCert”.

keyIdType = Integer

Indicates the type of a key identifier:

0 – NONE. No key identifier is supplied. The browser may use any key and certificate available.

1 – USER_KEY_HASH. A SHA-1 hash of the user public key is supplied in the next parameter. The browser MUST use the signature key that corresponds to the given public key hash or, if this key is not available, return “error:noCert”.

2 – TRUSTED_KEY_HASH. A SHA-1 hash of a trusted CA public key (or multiple of them) is supplied in the next parameter. The browser MUST use a signature key that is certified by the indicated CA (or some of them). If no such key is available, the browser MUST return “error:noCert”.

keyId = String

Identifies the key in a way based on the previous parameter.

For a SHA-1 public key hash, contains the 20-byte hash. Multiple values may be concatenated. Number of elements in the list is implied by the length of the parameter.

Return value: String or Invalid.

The content of the return string is the following

- in case of a succesful operation, the base-64 [RFC1521] encoding of `SignedContent`
- if there is no proper certificate or signature key available, the string “error:noCert”
- if the user cancelled the operation, the string “error:userCancel”

Exceptions: Errors in parameters, encoding or internal errors result in an `invalid` return value.

Example:

```
var foo = Crypto.signText("Bill of Sale\n------------------\n3
Bolognese $18.00\n1 Pepperoni  $7.00\n4 Lemonade   $6.00\n----
--------------\nTotal Price $31.00",
0, 1,
"\x37\x00\xB6\x96\x37\x75\xE3\x93\x48\x74\xD3\x98\x47\x53\x94\
x34\x58\x97\xB5\xD6");
// The application indicates the signature key
```

5.1.3 Handling of Certificates

For verification of the digital signature, the server must have access to a user’s certificate that is signed by a Certification Authority (CA) recognised by the server. There are several possibilities for how the server can get access to the user’s certificate:

1. The certificate is appended to the signature.
2. The public key hash is appended to the signature. The server is able to fetch the corresponding certificate from a certificate service.
3. A URL of the certificate is appended to the signature. The server is able to fetch the certificate using internet methods.
4. The server knows the user certificate based on a previous data exchange with the user, e.g., a previous digital signature.

5.1.4 Implementation using the WIM

This chapter describes how to implement the signText function using the WIM [WAPWIM].

A non-repudiation key is used for signing. This implies usage of a an authentication object used for this key only, and that the verification requirement cannot be disabled. E.g., in case of a PIN, the PIN MUST be entered separately for each signature operation.

The PKCS#15 key ID (commonObjectAttributes.id) has the value of the public key hash. So, it can be used to find the proper key or certificate, if the key is identified by USER_KEY_HASH. The certificate issuer public key hash (PKCS15CommonCertificateAttributes.requestId) can be used to find a proper certificate, if it is identified by TRUSTED_KEY_HASH.

Labels, contained in entries that describe private keys and certificates (commonObjectAttributes.label) SHOULD be used to display options to use for signing.

For a smart card implementation, the procedure is described in [WAPWIM], chapter 11.4.6.

6. Format of SignedContent

This section defines a format for transmission of signed content to/from WAP devices. It is described below using WTLS presentation [WAPWTLS]. Hash values of authenticated attributes are computed using a PKCS#7 template to provide end-to-end authentication between WAP clients and devices supporting the PKCS#7 standard for signed data representation.

```
enum {null(0), rsa_sha_pkcs1(1), ecdsa_sha_p1363(2), (255)}
DataSignatureAlgorithm;
```

Item	Description
null	No signature present.
rsa_sha_pkcs1	The signature is calculated according to [PKCS1] (see Appendix B), using octet string output.
ecdsa_sha	The signature is calculated according to [X9.62], using octet string output.

```
struct {
  DataSignatureAlgorithm algorithm;
  switch (algorithm) {
    case null: struct {};
    default: opaque signature<0..2^16-1>;
  };
} Signature;
```

```
enum { implicit(0), sha_key_hash(1), wtls_certificate(2),
x509_certificate(3), x968_certificate(4), certificate_url(5), (255)}
SignerInfoType;
```

Item	Description
implicit	The signer is implied by the content.
sha_key_hash	The SHA-1 hash of the public key, encoded as specified in [WAPWTLS].
wtls_certificate	A WTLS certificate.
x509_certificate	An X.509v3 certificate.
x968_certificate	An X9.68 certificate.
certificate_url	A URL where the certificate is located.

```
struct {
  SignerInfoType signer_info_type;
  switch (signer_info_type) {
```

```
     case implicit: struct{};
    case sha_key_hash:
      opaque hash[20];
    case wtls_certificate:
      WTLSCertificate;
    case x509_certificate:
      opaque x509_certificate<0..2^16-1>;
    case x968_certificate:
        opaque x968_certificate<0..2^16-1;
    case certificate_url:
      opaque url<0..255>;
  };
} SignerInfo;


enum {text(1), data(2), (255)} ContentType;
```

Item	Description
text	Encoded text (according to character set).
data	Encoded data (encoding indicated by content_encoding parameter, see below).

```
enum (false(0), true(1)} Boolean;

struct {
  ContentType content_type;
  uint16 content_encoding;
  Boolean content_present;
  switch (content_present) {
    case false: struct{};
    case true: opaque content<0..2^16-1>;
  };
} ContentInfo;
```

Item	Description
content_type	The type of the content that was signed.
content_encoding	For text type of content, indicates the character set used to encode the text before signing (IANA assigned character set number, see [WAPWSP]). The recommended character set is UTF-8 [UTF8]. Note that the hash is calculated over the encoded text (no length indication, terminating character or character set indicator is included). For data type of content, indicates a specific content type (assigned values are not defined yet).
content_present	Indicates if the content is present in the structure.
content	Content.

```
enum { gmt_utc_time(1), signer_nonce(2), (255) } AttributeType;
```

Item	Description
gmt_utc_time	The current time and date in UTC format (see Appendix C). Only the 12 actual date/time octet values are included; the trailing 'Z', indicating GMT or Zulu, is omitted since it is implicit.
signer_nonce	A nonce generated by the signer. This attribute MAY be used by devices that do not have an internal clock.

```
struct {
  AttributeType attribute_type;
  switch (attribute_type) {
    case gmt_utc_time: uint8[12];
    case signer_nonce: opaque signer_nonce[8];
  }
} AuthenticatedAttribute;
```

```
struct {
  uint8 version;
  Signature signature;
  SignerInfo signer_infos<0..2^16-1>;
  ContentInfo content_info;
  AuthenticatedAttribute authenticated_attributes<0..255>;
} SignedContent;
```

Item	Description
version	Version of the SignedContent structure. For this specification the version is 1.
signature	Signature
signer_infos	Information on the signer. This may contain zero items (in case the signer is implicit). Also, there may be multiple items of SignerInfo present (public key hash and a certificate).
content_info	Information about the content being signed. The actual content is optionally included in the structure.
authenticated_attributes	Attributes that are included in the signature.

6.1. Usage with signText

The result returned by signText is formatted as `SignedContent`. The original *stringToSign* is optionally included in the structure. It is the responsibility of the application that the verifying party (server) will have access both to the original text and the signature. The text may be generated in the

server and cached there. Or, if the text is generated in the client (e.g., based on user input), it should be included in the structure.

The verification service must take the character set into account. If the original service generated the *stringToSign*, it is necessary to convert that to a proper character set encoding.

6.2. Hash Calculation and Relationship to PKCS#7 SignedData

The signed content type is defined so as to allow end-to-end authentication of signed content based on PKCS#7 [PKCS7] signed data structures. A proxy server or gateway may accept a PKCS#7 signed data object and convert to the WAP signed content type without violating the end-to-end integrity of the signature. This is done by compressing the PKCS#7 header (by representing it in WTLS encoding format) without information loss. Since the mobile device can reconstruct the original header with any authenticated attributes it can verify the original signature.

When a mobile device is sending signed content it constructs the PKCS#7 header using a static template and filing in the relevant attribute values. The hash is computed as specified in [PKCS7]. The mobile device then formats and sends the SignedContent type. This allows a proxy or gateway to convert this back to PKCS#7 format for transmission to a server. In this way we achieve both bandwidth efficiency and limited parsing requirements on the mobile device while enabling end-to-end signed content verification with servers not supporting the WAP signed content type.

The hash calculation on the mobile device is performed as defined in [PKCS7], using the signer's authenticated attributes. This requires that the input for the hash calculation is represented in ASN.1 DER encoding. As shown below, complex DER encoding is not required, since the length of the values are known beforehand. An implementation needs only the (static) PKCS#7 DER structure, filing in the variable fields. It need not understand the specifics of the ASN.1 encoding.

According to [PKCS7], the mandatory authenticated attributes are the contentType and messageDigest attributes (hash of the original data). Additionally, either signing time or a random nonce MUST be used as an authenticated attribute. Signing time is recommended. A random number MAY be used by implementations that do not support real time clock.

The message-digesting process computes a message digest on the content together with the signer's authenticated attributes. The initial input to the message-digesting process is the value of the content being signed.

The authenticated attributes are the following [PKCS9].

Attribute	**OID**	**OID in Binary**
contentType	pkcs9-3	2a 86 48 86 f7 0d 01 09 03
messageDigest	pkcs9-4	2a 86 48 86 f7 0d 01 09 04
signingTime	pkcs9-5	2a 86 48 86 f7 0d 01 09 05
signerNonce	pkcs9-25-3	2a 86 48 86 f7 0d 01 09 19 03

To calculate the hash, the signer uses the following buffer as a template:

```
31 5d
   30 18
      06 09 2a 86 48 86 f7 0d 01 09 03 – contentType
      31 0b
         06 09 2a 86 48 86 f7 0d 01 07 01 -- data
   30 1c
      06 09 2a 86 48 86 f7 0d 01 09 05 – signingTime
      31 0f
         17 0d XX XX XX XX XX XX XX XX XX XX XX XX -- UTCTime
   30 23
      06 09 2a 86 48 86 f7 0d 01 09 04 – messageDigest
      31 16
         04 14 XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
XX
               -- SHA-1 digest
```

In order to construct the input for hash calculation, the following steps are performed

- use initially a 95-byte buffer as above (bytes 1...95)
- replace bytes 46...58 with the value of UTC time expressed as YYMMDDHHMMSSZ (ASCII-encoded)
- replace bytes 76...95 with the 20-byte value of the SHA-1 hash of the content value

The next step is to calculate the hash from the above 95-byte buffer. Finally, the signature is calculated.

Note that the PKCS#7 contentType “data” is used for both text and data content types specified in the beginning of this chapter.

Note also that since the input to the message digesting process is the content value (and not for example "ContentInfo" as defined in Section 6), the information about the character set associated with the content is not protected. Implementations should therefore take steps to ensure that it is not possible for an attacker to harmfully manipulate this character set information.

For verification, the above structure needs to be constructed based on values transmitted in SignedContent: content_type,
gmt_utc_time.

Note that the authenticated attributes are included in the in ascending order compared as octet strings.

A proxy server MAY construct a PKCS#7 [PKCS7] SignedData object based on a received SignedContent object. The motivation of doing this would be that some internet or other service applications may require a PKCS#7 formatted object to verify the signature. The conversion to PKCS#7 is based on the original text, the signature and a certificate.

A proxy server MAY also convert a PKCS#7 SignedData object to a SignedContent object for transmission to a mobile device.

When the mobile device receives (e.g. over WSP) a SignedContent object (containing text or any type of data), it should verify the signature and be able to present information on the signer and the result

of verification: if it was succesful, or if it failed with different reasons, like invalid signature or inability to verify the signer's certificate. When the SignedContent object contains signed text, the original text and result of verification must be presented in a manner which is distinctive from texts generated by applications using e.g. WML or WMLScript.

When using an 8-byte signerNonce instead of signingTime (as described above), the following buffer is used as a template:

```
31 59
   30 18
      06 09 2A 86 48 86 F7 0D 01 09 03    -- contentType
      31 0B
         06 09 2A 86 48 86 F7 0D 01 07 01  -- data
   30 18
      06 0A 2A 86 48 86 F7 0D 01 09 19 03  -- signerNonce
      31 0A
         04 08 XX XX XX XX XX XX XX XX   -- randomNonce
   30 23
      06 09 2A 86 48 86 F7 0D 01 09 04     -- messageDigest
      31 16
         04 14 XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
XX
               -- SHA-1 digest
```

In order to construct the input for hash calculation, the following steps are performed

- use initially a 91-byte buffer as above (bytes 1...91)
- replace bytes 47...54 with the value of the 8 byte nonce.
- replace bytes 72...91 with the 20-byte value of the SHA-1 hash

The next step is to calculate the hash from the above 91-byte buffer. Finally, the signature is calculated.

Appendix A Library Summary

The libraries and their library identifiers:

Library name	**Library ID**	**Page**
Crypto	6	8

The libraries and their functions:

Crypto library	**Function ID**
SignText	16

Appendix B Signature Calculation

B.1 ECDSA Signature Calculation

ECDSA signature calculation for signText is based on ANSI X9.62 [X9.62].

Three aspects of the ECDSA signature process are described:
- the input to the signature process;
- the signature process itself; and
- the output from the signature process.

The input to the ECDSA signature process is one of the completed templates specified in Section 6.2.

Note that ANSI X9.62 regards hashing as an integral part of the signing process – thus the completed template will be hashed using SHA-1 as required during the signing process.

The ECDSA signature process is then performed on the completed template as specified in Section 5.3 of ANSI X9.62.

The output of the ECDSA signature process is a pair of integers r and s. Here the ECDSA signature is converted to a byte string for inclusion in the “Signature” field of “SignedContent” as specified in Section 6 as follows: Convert the integer r to an octet string R and the integer s to an octet string S using the conversion routine specified in Section 4.3.1 of ANSI X9.62 [X9.62]. Both R and S should be the same length as the length needed to represent the order of the base point G. The signature is represented as the concatenation of R and S: R | S.

Note that the signature will subsequently be re-encoded using the ASN.1 syntax for an ECDSA signature specified in Section of ANSI X9.62 if the “SignedContent” format specified in Section 6 is converted to PKCS7 CMS format.

Recommended curves for use with ECDSA are described in Appendix A of WTLS [WAPWTLS].

B.2 RSA PKCS#1 Signature Calculation

The calculation is based on [PKCS1], chapter 10.1. It consists of three steps: message digesting (hashing), data encoding and RSA encryption. (The fourth step, octet-string-to-bit-string conversion is not necessary here.)

The message (the text being signed) is digested using SHA-1 [SHA1]. The 20-byte output and a SHA-1 algorithm identifier shall be combined into an ASN.1 [ASN1] value of type DigestInfo, described below, which shall be DER-encoded [DER] to give an octet string, the data.

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm DigestAlgorithmIdentifier,
    digest Digest }

DigestAlgorithmIdentifier ::= AlgorithmIdentifier

Digest ::= OCTET STRING
```

`digestAlgorithm` identifies the message-digest algorithm. For this application, it should associate the SHA-1 algorithm. The object identifier is the following

```
sha-1 OBJECT IDENTIFIER ::=
  { iso(1) identified-organization(3)  oiw(14) secsig(3) 2 26 }
```

The BER encoding of the above is: `2b 0e 03 02 1a`

`digest` is the result of the message digesting process, ie, the message digest.

The BER encoding of `DigestInfo` is

```
30 21                       -- SEQUENCE (DigestInfo)
  30 09                     -- SEQUENCE (AlgorithmIdentifier)

    06 05 2b 0e 03 02 1a      -- digestAlgorithm = sha-1
    05 00                   -- parameters = NULL
  04 14                     -- OCTET STRING (digest)
    xx xx xx xx             -- digest value
    xx xx xx xx
    xx xx xx xx
    xx xx xx xx
    xx xx xx xx
```

where the last 20 bytes is the message digest. So, in order to implement the BER-encoded `DigestInfo`, it is sufficient to concatenate the constant 15 bytes and the 20 bytes of the hash.

The resulting data (BER-encoded `DigestInfo`), is encrypted with the signer's private key as described in [PKCS1] section 7, using the block type 1. The resulting octet string, is the signature.

Appendix C UTC Time

The universal time type, UTCTime, is a standard ASN.1 type intended for international applications where local time alone is not adequate. UTCTime specifies the year through the two low order digits and time is specified to the precision of one minute or one second. UTCTime includes either Z (for Zulu, or Greenwich Mean Time) or a time differential.

For the purposes of this profile, UTCTime values MUST be expressed Greenwich Mean Time (Zulu) and MUST include seconds (i.e., times are YYMMDDHHMMSSZ), even where the number of seconds is zero. Conforming systems MUST interpret the year field (YY) as follows:

Where YY is greater than or equal to 50, the year shall be interpreted as 19YY; and where YY is less than 50, the year shall be interpreted as 20YY.

The above usage is as is specified in [RFC2459].

For transmission in the signed content `AuthenticatedAttribute` type (`gmt_utc_time`) the trailing ‘Z’ is omitted as it is implicit.

Appendix D Static Conformance Requirement

This static conformance requirement [WAPCREQ] lists a minimum set of functions that can be implemented to help ensure that WMLScript Crypto Library implementations will be able to inter-operate. The “Status" column indicates if the function is mandatory (M) or optional (O).

D.1 Client Options

Item	Function	Subfunction	Reference	Status	Requirement
WMLSCrypt-C-001	SignText	Function supported with at least one signature algorithm	5.1	M	WMLSCrypt-C-002 OR WMLSCrypt-C-003
WMLSCrypt-C-002		RSA	6	O	
WMLSCrypt-C-003		ECDSA	6	O	
WMLSCrypt-C-004	Use of WIM		5.1.4	O	WIM:MCF AND WIM-C-002 AND WIM-C-042

D.2 Script Encoder Options

Item	Function	Subfunction	Reference	Status	Requirement
WMLSCrypt-S-001	SignText		5.1	M	

D.3 Application options

Item	Function	Subfunction	Reference	Status	Requirement
WMLSCrypt-A-001	SignText output (SignedContent) verification	Signature verification supported with at least one signature algorithm	5.1	M	WMLSCrypt-A-002 OR WMLSCrypt-A-003
WMLSCrypt-A-002		RSA	6	O	
WMLSCrypt-A-003		ECDSA	6	O	