



TMS Cryptography Pack

DEVELOPERS GUIDE

Contents

Contents..... 2

Availability..... 3

Online references..... 3

Description..... 5

AES (modes ECB-CBC-OFB-CTR-CTS) 6

AES MAC 10

AES GCM..... 12

AES GCM..... 15

AES NI..... 18

RSA 20

ECDSA, EdDSA, ECDH and ECIES..... 25

SALSA 29

xCHACHA20 31

SHA-2 33

SHA-3 35

SPECK..... 38

PBKDF2 41

HKDF 43

Blake2 45

~~RIPEMD-160~~ 47

Argon2 48

TLSH 50

Converter class..... 53

X509 certificates 58

X509 CSR 65

PKCS11..... 68

PEM Certificates..... 73

PKCS#12, PFX, P12 Certificates..... 74

Abstract Syntax Notation 1 support files 75

Cryptographic Message Syntax support file 76

XAdES 78

CAdES..... 83

PAdES 87

Random number generators 90

ZIP encryption and decryption 91

Encrypt an ini file..... 93

Generate a self-decrypting file 95

Troubleshooting 97

Annex A: certificate formats 99

Availability

TMS Cryptography Pack is available as VCL and FMX component set for Delphi and C++Builder. It can also be used in `console` mode.

TMS Cryptography Pack is available for RAD Studio (DELPHI and C++ Builder) 10.4 Sydney, 11.0, 11.1 Alexandria, 12.0, 12.1, 12.2, 12.3 Athens, 13.0 Florence.

TMS Cryptography Pack has been designed for and tested with: Windows 10, Windows 11, OSX 10.12.2, iOS 17.6, and Android 12 or newer. It may also work on other platforms and version.

TMS Cryptography Pack supports following targets: Win32, Win64, Android32, Android64, OSX64, OSX-ARM64, iOS32, iOS64 (**Linux with no guarantees**).

Online references

TMS software website:

<https://www.tmssoftware.com>

TMS Cryptography Pack page:

<https://www.tmssoftware.com/site/tmscrypto.asp>

TMS Cryptography Pack is available separately and also as part of:

TMS ALL-ACCESS:

<https://www.tmssoftware.com/site/tmsallaccess.asp>

Technical support is provided through:

<https://support.tmssoftware.com/>

GENERAL WARNING

This version of the TMS CP library is the first to be fully written in DELPHI whereas the previous ones were written in C, with DELPHI wrappers.

The original C code of the cryptographic algorithms was ported to DELPHI in a straightforward way, keeping the 'C' conventions to ensure line to line comparison in debugging (when relevant). This low-level code will gradually evolve to better use DELPHI concepts, and in some cases reduce redundancies.

Therefore, it is not encouraged to use the low-level cryptographic algorithms (although the overall performance would be better), mainly located in 'XXXCore' files because the APIs may change over time. It is recommended to use classes from the 'XXXObj' files.

It also known that some algorithms can be improved from a performance standpoint, especially with the use of parallel computing. RSA is a particular example that cumulates impediments on Windows. More improvements will be made over time.

NOTE 1: The SHA2, SHA3 and BLAKE2B classes have been restructured. SHA2 hash sizes have now to be set with THashSize = (hs224, hs256, hs384, hs512, hs1024) INSTEAD OF TSHAHSIZE = (hs256, hs384, hs512).

NOTE 2: some APIs have been changed to ensure better uniformity across cryptographic interfaces and Users may have to adjust existing code to take those changes into account, especially with the new 'inputFormat' property (see README-FIRST.txt).

NOTE 3: XAdES has been modified to take national specifics into account. **Work is still in progress.**

NOTE 4: Many constants have been moved to CryptoConst.pas and it necessary to add this files to 'uses'.

NOTE 5: check the relevant *.pas file for the latest version of the class content. There ARE discrepancies with this manual.

NOTE 6: there is a conflict in names for the C++ version due to Wincrypt.h statements for cryptographic primitives on Windows. This can only be fixed manually at this stage.

Description

TMS Cryptography Pack is a software library that provides various algorithms used to encrypt, sign and hash data. This library has been developed by Cyberens (www.cyberens.fr).

This manual provides a complete description of how to use the library and its various features. Each section corresponds to an algorithm used in cryptography and a class in the TMS Cryptography Pack.

The different algorithms are the following:

- ✓ AES (modes ECB-CBC-OFB-CTR-CTS)
- ✓ AES MAC
- ✓ AES GCM
- ✓ AES NI (as of 5.1.1.0)
- ✓ SPECK (128/192/256-bit key sizes and CBC mode with PKCS#7 padding only)
- ✓ RSA
- ✓ ECDSA and EdDSA (ed25519, ed448, ed 511187, P-521, P-384 and P-256)
- ✓ ECIES (ed25519 is in, others will be added in a future release)
- ✓ SALSAs
- ✓ SHA-2
- ✓ SHA-3
- ✓ PBKDF 2
- ✓ HKDF
- ✓ Blake2B
- ✓ Argon2
- ✓ Decoding of X509 certificates and PKIX containers
- ✓ Decoding of PFX key containers
- ✓ Generation of X509 self-signed certificates
- ✓ Generation of X509 CSR (support for RSA 2048 / SHA 256, EC P-256, P-384 and P-521)
- ✓ XAdES
- ✓ CAdES
- ✓ PAdES
- ✓ xChaCha20
- ✓ TLSH
- ✓ ZIP encryption services supporting the AE-2 requirements

Compatibility is no longer ensured with public key cryptographic operations using SHA1. However, SHA1 has been kept for interoperability with legacy certificates and other cryptographic libraries, including ZIP.

AES (modes ECB-CBC-OFB-CTR-CTS)

AES or Advanced Encryption Standard is a symmetric encryption algorithm. It has become a standard since 2002 in the USA, described in NIST FIPS PUB 197. Its input is a 128-bit message (plain text or clear text), and its output is a 128-bit cipher text. Depending on the version, the key length is either 128 bits, 192 bits or 256 bits. To encrypt messages of different lengths, we use different encryption modes:

- ✓ ECB (Electronic Code Book): it is the simplest mode. The message to encrypt is divided into blocks of 128 bits and each block is encrypted separately with the same key (*this mode should not be used because of possible collisions*).
- ✓ CBC (Cipher Block Chaining): it XORs the 128-bit first block of clear text with a 128-bit initialisation vector. Then it encrypts the result with AES. For each new block, it uses the previous cipher text as the initialisation vector.
- ✓ OFB (Output Feedback): an initialisation vector is encrypted with AES, then XORed with the first block of clear text, to obtain the first block of cipher text. Then this encrypted initialisation vector is reused as the initialisation vector for the next block.
- ✓ CTR (Counter): it encrypts a counter, which is incremented for each block. Then each counter is XORed with a block of clear text to obtain a block of cipher text.
- ✓ CTS (Cipher Text Stealing): similar to CBC but with the last 2 blocks inverted and chopped to the exact plaintext length (clear text length must be greater than 16 bytes).

These modes are described in the NIST Special Publication 800-38A.

The AES class is:

```

TAESKeyLength = (kl128, kl192, kl256);
TAESType = (atECB, atCBC, atOFB, atCTR, atCTS);
TIVMode = (rand, userdefined);
TPaddingMode = (PKCS7, nopadding);

TAESEncryption = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType;
        paddingMode: TPaddingMode; outputFormat: TConvertType;
        uni: TUnicode); overload;
    Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType;
        paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode;
        IV: string); overload;
    Destructor Destroy; override;
    function Encrypt(s: string): string;
    function Decrypt(s: string): string; overload;
    function Decrypt(s: string, var o: string): Integer; overload;
    procedure EncryptFileW(s, o: string);
    function DecryptFileW(s, o: string): Integer;
    procedure EncryptStream(s: TStream; var o: TStream);
    function DecryptStream(s: TStream; var o: TStream): Integer;
published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
        kl128;
    property AType: TAESType read FType write FType default atcbc;
    property inputFormat: TConvertType read FInputFormat write FInputFormat
    
```

```

    default raw;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
        default hexa;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property paddingMode: TPaddingMode read FPaddingMode write FPaddingMode
        default PKCS7;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
end;

```

The constructors and the destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TAESKeyLength; key: string; AType: TAESType; paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor with IVMode = rand¹
- **Constructor** Create(keyLength: TAESKeyLength; key: string; AType: TAESType; paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode; IV: string); **overload**; the constructor with IVMode = userdefined²
- **Destructor** Destroy; **override**; to zeroize the key

The public methods are:

- **function** Encrypt(s: string): string; to encrypt a string s
- **function** Decrypt(s: string): string; to decrypt a string s
- **function** Decrypt(s: string; var o: string): Integer; to decrypt a string s and return 0 if success and error code if failure
- **procedure** EncryptFileW(s, o: string); to encrypt a file whose path is s and the encrypted file path is o
- **function** DecryptFileW(s, o: string); to decrypt a file whose path is s and the decrypted file path is o and return 0 if success and error code if failure
- **procedure** EncryptStream(s: TStream; var o: TStream); to encrypt the stream s into the stream o
- **function** DecryptStream(s: TStream; var o: TStream); to decrypt the stream s into the stream o and return 0 if success and error code if failure

The properties are:

- **property** Key: string read FKey write SetKey; to read and write the key
- **property** KeyLength: TAESKeyLength read FKeyLength write SetKeyLength; to read and write the key length in bits (128, 192 or 256 bits)
 - **NOTE: KeyLength needs to be set before the Key is assigned to prevent an exception from being raised if there is a mismatch**
- **property** AType: TAESType read FType write FType; to read and write the encryption mode (ECB, CBC, OFB, CTR or CTS)

¹ The Initialisation Vector (IV) is randomly generated within the class.

² The user has to provide the IV, either by generating it or reusing it from another source.

- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section – **all input data shall have the same format: key IV, data**)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** IVMode: TIVMode **read** FIVMode **write** FIVMode; to read and write the IV mode, userdefined or rand
- **property** IV: **string read** FIV **write** SetIV; to read and write the IV of 16 bytes if the IV mode is userdefined (in rand mode, the IV is randomly generated and added to the encrypted text)
- **property** PaddingMode: TPaddingMode **read** FPaddingMode **write** FpaddingMode; to read and write the padding mode, PKCS7 or nopadding. In PKCS7, the length of the encrypted text is always the length of the clear text + 16 bytes (plus 16 bytes in the case of rand IV mode). In nopadding mode, the length of the clear text must be a multiple of 16 bytes, and no padding is added to the clear text
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer or the input file name has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during encryption / decryption of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate that the progress changes

Example of encryption with AES

```
var
  aes: TAESEncryption;
  cipher: string;

begin
  aes := TAESEncryption.Create;
  aes.AType := atCBC; // CBC is however the default mode
  aes.KeyLength := kl256;
  aes.Unicode := yesUni;
  aes.Key := '12345678901234567890123456789012'; // 32 bytes
  aes.InputFormat := raw;
  aes.OutputFormat := hexa;
  aes.PaddingMode := TpaddingMode.PKCS7;
  aes.IVMode := TIVMode.rand; // This will generate a random IV
  cipher := aes.Encrypt('test');
  aes.Free;
end;
```

Example of decryption with AES and the same parameters as in encryption

```
var
  aes: TAESEncryption;
  cipher, cleartext: string;
  Con: TConvert;

begin
  aes := TAESEncryption.Create;
  aes.AType := atCBC; // CBC is however the default mode
  aes.KeyLength := kl256;
  aes.Unicode := yesUni;
```

```
aes.Key := '12345678901234567890123456789012'; // 32 bytes

aes.InputFormat := raw;
aes.OutputFormat := raw; // we want the initial text as a 'raw' string
                        // when decrypted

aes.PaddingMode := TpaddingMode.PKCS7;
aes.IVMode := TIVMode.rand; // This will get the random IV from the cryptogram
Conv := TConvert.Create(hexa); // because the previous output was in hex
cipher := Conv.FormatToChar(cipher); // we need to convert it to 'raw'

cleartext := aes.Decrypt(cipher); // key, IV + cryptogram are now 'raw'
..Conv.Free;
aes.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES MAC

To produce a message authentication code (MAC), we use the MAC mode. It is described in the NIST Special Publication 800-38B.

The AES MAC class is:

```

TAESKeyLength = (kl128,kl192,kl256);
TAESMAC = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(keyLength: TAESKeyLength; key: string;
    tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode);
    overload;
  Destructor Destroy; override;
  function Generate(s: string): string;
  function Verify(s, t: string): integer;
  function GenerateFromFile(s: string): string;
  function VerifyFromFile(s, t: string): integer;
  function GenerateFromStream(s: TStream): string;
  function VerifyFromStream(s: TStream; t: string): integer;
published
  property key: string read FKey write SetKey;
  property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
    kl128;
  property tagSizeBits: integer read FTagSizeBits write SetTagSizeBits default
    128;
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property Unicode: TUnicode read FUni write FUni default yesUni;
  property Progress: Integer read FProgress write SetProgress;
  property OnChange: TNotifyEvent write FOnChange;
end;

```

The constructors and the destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to set all the properties
- **Destructor** Destroy; **override**; to zero the key

The public methods are:

- **function** Generate(s: string): string; to generate a tag from a string s
- **function** Verify(s, t: string): Integer; to verify the tag t from the string s
- **function** GenerateFromFile(s: string): string; to generate a tag from a file the path of which is s
- **function** VerifyFromFile(s, t: string): Integer; to verify a tag t from a file the path of which is s

- **function** `GenerateFromStream(s: TStream): string`; to generate a tag from stream s
- **function** `VerifyFromStream(s: TStream; t: string): integer`; to verify the tag t from stream s

The properties are:

- **property** `Key: string read FKey write SetKey`; to read and write the key
- **property** `KeyLength: TAESKeyLength read FKeyLength write SetKeyLength`; to read and write the key length in bits (128, 192 or 256 bits)
- **property** `TagSizeBits: Integer read FTagSizeBits write SetTagSizeBits`; to read and write the tag length in bits (<= 128 bits)
- **property** `InputFormat: TConvertType read FInputFormat write FInputFormat`; to read and write the input format of the data (see Converter class section – **all input data shall have the same format: key IV, data**)
- **property** `OutputFormat: TConvertType read FOutputFormat write FoutputFormat`; to read and write the output format of the data (see Converter class section)
- **property** `Unicode: TUnicode read FUni write FUni`; to indicate whether the input buffer or the input file name has Unicode characters
- **property** `Progress: Integer read FProgress write SetProgress`; to indicate progress during generation / verification of the tag of a stream
- **property** `OnChange: TNotifyEvent write FOnChange`; to indicate that the progress changes

Example of how to generate a tag from a `string` with AES MAC

```
var
  aesmac: TAESMAC;
  tag: string;
begin
  aesmac := TAESMAC.Create;
  aesmac.KeyLength := kl256;
  aesmac.Key := '12345678901234567890123456789012';
  aesmac.TagSizeBits := 128;
  aesmac.InputFormat := raw;
  aesmac.OutputFormat := hexa;
  aesmac.Unicode := noUni;
  tag := aesmac.Generate('test');
  aesmac.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES GCM

The last mode is the Galois Counter Mode (GCM), that encrypts the message using the CTR mode and produces a tag using a hash function. It is described in NIST Special Publication 800-38D. This mode allows the user to verify the integrity of some additional data, without encrypting it.

The AES-GCM class is:

```

TAESKeyLength = (kl128, kl192, kl256);
TIVMode = (rand, userdefined);

TAESGCM = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
        tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode);
        overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
        tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode;
        IVLength: integer; IV: string); overload;
    Destructor Destroy; override;
    function EncryptAndGenerate(s, a: string): string;
    function DecryptAndVerify(s, a: string; var o: string): integer;
    procedure EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath,
        tagPath: string);
    function DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath,
        tagPath: string): integer;
    procedure EncryptAndGenerateFromStream(inputStream: TStream;
        var outputStream: TStream; addDataStream: TStream;
        var tagStream: TStream);
    function DecryptAndVerifyFromStream(inputStream: TStream;
        var outputStream: TStream; addDataStream: TStream;
        var tagStream: TStream): integer;
published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
        kl128;
    property tagSizeBits: integer read FTagSizeBits write SetTagSizeBits default
        128;
    property inputFormat: TConvertType read FInputFormat write FInputFormat
        default raw;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
        default hexa;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property IVLength: integer read FIVLength write SetIVLength;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
    property UseOldGCM: boolean read FUseOldGCM write FUseOldGCM default false;
end;

```

The constructors and the destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor with IVMode = rand
- **Constructor** Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode; IVLength: integer; IV: string); **overload**; the constructor with IVMode = userdefined
- **Destructor** Destroy; **override**; to zero the key

The methods are:

- **function** EncryptAndGenerate(s, a: string): string; to encrypt string s and generate a tag from s and additional data a (the output string and the tag are concatenated)
- **function** DecryptAndVerify(s, a: string; var o: string): integer; to decrypt string s and verify the tag (contained in s) associated with s and the additional data a, the resulting string is the o string. This function returns 0 if the decryption succeeded and an error code if it failed.
- **procedure** EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath, tagPath: string); to encrypt a file whose path is inputPath into a file whose path is outputPath and generate a tag (associated with the inputPath file and the addDataPath file) into the file tagPath
- **function** DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath, tagPath: string): Integer; to decrypt a file whose path is inputPath into a file which path is outputPath and verify the tag, associated with the outputPath file and the addDataPath file, whose path is tagPath.
- **procedure** EncryptAndGenerateFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream); to encrypt the stream inputStream into outputStream and generate a tag (associated with inputStream and addDataStream)
- **function** DecryptAndVerifyFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream): integer; to decrypt the stream inputStream into outputStream and verify the tag tagStream associated with outputStream and addDataStream

The properties are:

- **property** Key: string **read** FKey **write** SetKey; to read and write the key
- **property** KeyLength: TAESKeyLength **read** FKeyLength **write** SetKeyLength; to read and write the key length in bits (128, 192 or 256 bits)
- **property** TagSizeBits: Integer **read** FTagSizeBits **write** SetTagSizeBits; to read and write the tag length in bits (<= 128 bits)
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section – **all input data shall have the same format: key IV, data**)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)

- **property** IVMode: TIVMode **read** FIVMode **write** FIVMode; to read and write the IV mode, userdefined or rand.
- **property** IV: **string** **read** FIV **write** SetIV; to read and write the IV of IVLength bytes if the IV mode is userdefined (in rand mode, the IV (12 bytes) is randomly generated and added to the encrypted text)
- **property** IVLength: integer **read** FIVLength **write** SetIVLength; to read and write the IV length in bytes if the IVMode is userdefined
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during encryption / decryption of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate progress changes
- **property** UseOldGCM: boolean **read** FUseOldGCM **write** FUseOldGCM **default** false; to use the old version of AES GCM before 3.5 version of the library

Example of how to encrypt with AES-GCM

```
var
  aesgcm: TAESGCM;
  cipher: string;
begin
  aesgcm:= TAESGCM.Create;
  aesgcm.TagSizeBits:= 128;
  aesgcm.KeyLength:= kl256;
  aesgcm.Key:= '12345678901234567890123456789012';
  aesgcm.InputFormat:= raw;
  aesgcm.OutputFormat:= base64;
  aesgcm.IVMode:= TIVMode.rand;
  aesgcm.Unicode := yesUni;
  cipher:= aesgcm.EncryptAndGenerate('test', '');
  aesgcm.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES GCM

The last mode is the Galois Counter Mode (GCM), that encrypts the message using the CTR mode and produces a tag using a hash function. It is described in NIST Special Publication 800-38D. This mode allows the user to verify the integrity of some additional data, without encrypting it.

The AES-GCM class is:

```

TAESKeyLength = (kl128, kl192, kl256);
TIVMode = (rand, userdefined);

TAESGCM = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
        tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode);
        overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
        tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode;
        IVLength: integer; IV: string); overload;
    Destructor Destroy; override;
    function EncryptAndGenerate(s, a: string): string;
    function DecryptAndVerify(s, a: string; var o: string): integer;
    procedure EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath,
        tagPath: string);
    function DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath,
        tagPath: string): integer;
    procedure EncryptAndGenerateFromStream(inputStream: TStream;
        var outputStream: TStream; addDataStream: TStream;
        var tagStream: TStream);
    function DecryptAndVerifyFromStream(inputStream: TStream;
        var outputStream: TStream; addDataStream: TStream;
        var tagStream: TStream): integer;
published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
        kl128;
    property tagSizeBits: integer read FTagSizeBits write SetTagSizeBits default
        128;
    property inputFormat: TConvertType read FInputFormat write FInputFormat
        default raw;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
        default hexa;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property IVLength: integer read FIVLength write SetIVLength;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
    property UseOldGCM: boolean read FUseOldGCM write FUseOldGCM default false;
end;

```

The constructors and the destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TAESKeyLength; key: **string**; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor with IVMode = rand
- **Constructor** Create(keyLength: TAESKeyLength; key: **string**; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode; IVLength: integer; IV: **string**); **overload**; the constructor with IVMode = userdefined
- **Destructor** Destroy; **override**; to zero the key

The methods are:

- **function** EncryptAndGenerate(s, a: **string**): **string**; to encrypt string s and generate a tag from s and additional data a (the output string and the tag are concatenated)
- **function** DecryptAndVerify(s, a: **string**; var o: **string**): integer; to decrypt string s and verify the tag (contained in s) associated with s and the additional data a, the resulting string is the o string. This function returns 0 if the decryption succeeded and an error code if it failed.
- **procedure** EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath, tagPath: **string**); to encrypt a file whose path is inputPath into a file whose path is outputPath and generate a tag (associated with the inputPath file and the addDataPath file) into the file tagPath
- **function** DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath, tagPath: **string**): Integer; to decrypt a file whose path is inputPath into a file which path is outputPath and verify the tag, associated with the outputPath file and the addDataPath file, whose path is tagPath.
- **procedure** EncryptAndGenerateFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream); to encrypt the stream inputStream into outputStream and generate a tag (associated with inputStream and addDataStream)
- **function** DecryptAndVerifyFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream): integer; to decrypt the stream inputStream into outputStream and verify the tag tagStream associated with outputStream and addDataStream

The properties are:

- **property** Key: **string** **read** FKey **write** SetKey; to read and write the key
- **property** KeyLength: TAESKeyLength **read** FKeyLength **write** SetKeyLength; to read and write the key length in bits (128, 192 or 256 bits)
- **property** TagSizeBits: Integer **read** FTagSizeBits **write** SetTagSizeBits; to read and write the tag length in bits (<= 128 bits)
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section – **all input data shall have the same format: key IV, data**)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)

- **property** IVMode: TIVMode **read** FIVMode **write** FIVMode; to read and write the IV mode, userdefined or rand.
- **property** IV: **string** **read** FIV **write** SetIV; to read and write the IV of IVLength bytes if the IV mode is userdefined (in rand mode, the IV (12 bytes) is randomly generated and added to the encrypted text)
- **property** IVLength: integer **read** FIVLength **write** SetIVLength; to read and write the IV length in bytes if the IVMode is userdefined
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during encryption / decryption of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate progress changes
- **property** UseOldGCM: boolean **read** FUseOldGCM **write** FUseOldGCM **default** false; to use the old version of AES GCM before 3.5 version of the library

Example of how to encrypt with AES-GCM

```
var
  aesgcm: TAESGCM;
  cipher: string;
begin
  aesgcm:= TAESGCM.Create;
  aesgcm.TagSizeBits:= 128;
  aesgcm.KeyLength:= kl256;
  aesgcm.Key:= '12345678901234567890123456789012';
  aesgcm.InputFormat:= raw;
  aesgcm.OutputFormat:= base64;
  aesgcm.IVMode:= TIVMode.rand;
  aesgcm.Unicode := yesUni;
  cipher:= aesgcm.EncryptAndGenerate('test', '');
  aesgcm.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES NI

The AES New Instructions (NI) uses X86-64 CPU instructions to provide a fast AES implementation. The TAESNI class proposes the CBC and GCM modes only and supports file encryption and decryption.

There is no visual component for this class.

Examples of the TAESNI class are provided in the Demo directory of the library.

```
TAESNI = class
  private
    FIncContext: TAESContext; // encryption
    FDecContext: TAESContext; // decryption
    FKeySize: TAESKeyLength;
    FType: TAESType;
    FIVMode: TIVMode;
    FIV: TIV;
    FInputFormat: TConvertType;
    FProgress: Integer;
    FOnChange: TNotifyEvent;

    procedure PrepareDecryptionKey;
    procedure ApplySingleBlockIMC(var Block: T128BitBlock);

    procedure SetIV(const Value: string);
    function GetIV: string;
    procedure SetProgress(P: integer);

    procedure FinalizeGCMTag(var AuthBlock: T128BitBlock; const H, JEnc:
      T128BitBlock; DataLen: Int64);

  public
    constructor Create(aKey: string; eType: TAESType; Format: TConvertType);
      overload;
    constructor Create(aKey: string; eType: TAESType); overload;

    procedure EncryptBlock(const Input; var Output);
    procedure DecryptBlock(const Input; var Output);

    function EncryptCBC(const Data: TBytes): TBytes;
    function DecryptCBC(const Data: TBytes): TBytes;

    procedure EncryptFileCBC(const InFile, OutFile: string);
    procedure DecryptFileCBC(const InFile, OutFile: string);

    procedure EncryptFileGCM(const InFile, OutFile: string);
    procedure DecryptFileGCM(const InFile, OutFile: string);

  published
    property AType: TAESType read FType write FType default atCBC;
    property inputFormat: TConvertType read FInputFormat write FInputFormat
      default raw;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read GetIV write SetIV;
    property Progress: integer read FProgress write SetProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
end;
```

Abstract from the example programme.

```

procedure TMainForm.TestFileEncryptIV;
var
    AES: TAESNI;
    Key: TBytes;
    Conv: TConvert;

begin
    if OpenDialog.Execute = false then
        Exit;

    // This is a constant key from the NIST test vectors.
    // Use your own key for a real use.
    Key := TBytes.Create(
        $60, $3d, $eb, $10, $15, $ca, $71, $be,
        $2b, $73, $ae, $f0, $85, $7d, $77, $81,
        $1f, $35, $2c, $07, $3b, $61, $08, $d7,
        $2d, $98, $10, $a3, $09, $14, $df, $f4);

    Conv := TConvert.Create(raw);
    AES := TAESNI.Create(Conv.TBytesToString(Key), atCBC);
    AES.IVMode := rand;
    try
        // Encryption: IV generated and stored in 'data.enc'
        AES.EncryptFileCBC(OpenDialog.FileName, '.\document.enc');
        MainMemo.Lines.Add('PDF encrypted');

    finally
        AES.Free;
        Conv.Free;
    end;
end;

```

All AES NIfunctions/procedures are located in the AESNI.pas file.

RSA

RSA is an asymmetric encryption algorithm and a signature algorithm, described in 1977 by Ronald Rivest, Adi Shamir et Leonard Adleman.

RSA is an algorithm that can encrypt and decrypt, on the one hand, and sign and verify a signature, on the other hand. Nevertheless, it is recommended to use the AES to encrypt a message, a string or file, rather than RSA for performance reasons.

To encrypt some data with RSA, it is necessary to use the public key of the recipient and to decrypt, it is necessary to use the recipient's private key.

To sign some data with RSA, it is necessary to use the sender's private key, and the recipient verifies the signature with the public key of the sender. **This implementation is limited to signing/verifying messages and strings of less than 1024 bytes**, for practical reasons³. For longer strings and messages, SignFile shall be used with the string or message stored in an intermediate file.

The RSA algorithm is not secure in its initial form. To make it secure, there are two options. The first one is to use OAEP for encryption and PSS for signature. OAEP has been developed as a padding scheme. Similarly, for the signature, the secure form is called PSS. The second is to use PKCS v1.5 padding scheme for encryption and signature. OAEP, PSS and v1.5 are described in the PKCS#1 v2.2 standard.

In this implementation of the algorithm, only RSA keys of 2048-, 3072- or 4096-bit length are authorised.

This RSA algorithm supports sha256, sha384 and sha512 as hash functions. Sha1 is only used for X509 and PFX certificate verification (this is transparent in the validation process).

NOTE 1: SHA1 has been removed as of version 5 of TMS CP but can be used in the Mask Generation Function (MGF).

NOTE 2: RSA key generation can be a slow process, especially with long keys. It is not recommended to generate keys on iOS or Android platforms as the process can be terminated by the OS before a key pair is generated.

NOTE 3: properties MGFhash and OAEPlabel have been added in 5.0.9.1 to improve interoperability with other libraries.

The RSA class is:

```
TRSAKeyLength = (kl2048, kl3072, kl4096);
TRSAEncType = (oaep, epkcs1_5);
TRASignType = (pss, spkcs1_5);
TRSAHashfunction = (sha1, sha256, sha384, sha512);

TRSAEncSign = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
        publicExp: string; hashF: TRSAHashFunction;
        outputFormat: TConvertType; uni: TUnicode); reintroduce; overload;
```

³ This limitation will be removed in a future release.

```

Constructor Create(keyLength: TRSAKeyLength; modulus: string;
    publicExp: string; hashF: TRSAHashFunction;
    outputFormat: TConvertType; uni: TUnicode; encT: TRSAEncType);
    reintroduce; overload;
Constructor Create(keyLength: TRSAKeyLength; modulus: string;
    publicExp: string; hashF: TRSAHashFunction;
    outputFormat: TConvertType; uni: TUnicode; signT: TRSASignType);
    reintroduce; overload;
constructor Create(keyLength: TRSAKeyLength; modulus: string;
    publicExp: string; privateExp: string; hashF: TRSAHashFunction;
    outputFormat: TConvertType; uni: TUnicode; CT: Boolean);
    reintroduce; overload;
constructor Create(keyLength: TRSAKeyLength; modulus: string;
    publicExp: string; privateExp: string; hashF: TRSAHashFunction;
    outputFormat: TConvertType; uni: TUnicode;
    encT: TRSAEncType; CT: Boolean); reintroduce; overload;
constructor Create(keyLength: TRSAKeyLength; modulus: string;
    publicExp: string; privateExp: string; hashF: TRSAHashFunction;
    outputFormat: TConvertType; uni: TUnicode;
    signT: TRSASignType; CT: Boolean); reintroduce; overload;
Destructor Destroy; override;
procedure GenerateKeys;
procedure GenerateKeysX509Compatible(var dp, dq, p, q, inverseQ: string);
function Encrypt(m: string): string;
function Decrypt(m: string): string;
function Sign(m: string): string;
function Verify(m: string; s: string): Integer;
function SignFile(filePath: string): string;
function VerifySignatureFile(filePath, s: string): Integer;

```

```

procedure FromCertificate(CertStr: string);
procedure FromCertificateFile(CertFile: string);
procedure FromPrivateKey(KeyStr: string);
procedure FromPrivateKeyFile(KeyFile: string);
procedure FromPublicKey(KeyStr: string);
procedure FromPublicKeyString(KeyStr: string);
procedure FromPublicKeyFile(KeyFile: string);

```

published

```

property modulus: string read FModulus write SetModulus;
property PublicExponent: string read FPublicExponent
    write SetPublicExponent;
property PrivateExponent: string read FPrivateExponent
    write SetPrivateExponent;
property keyLength: TRSAKeyLength read FKeyLength write SetKeyLength
    default kl2048;
property hashFunction: TRSAHashFunction read FHashFunction write
    FHashFunction default sha256;
property MGFhash: TRSAHashFunction read FMGFHash write
    FMGFHash default sha256;
property OAEPlabel: string read FOAEPlabel write FOAEPlabel;
property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
property Unicode: TUnicode read FUni write FUni default yesUni;
property passwd: string read Fpw;
property withOpenSSL: Boolean read FwithOpenSSL write SetwithOpenSSL
    default false;
property constantTime: Boolean read FConstantTime write FConstantTime

```

```

    default true;
    property signType: TRSASignType read FSignType write FSignType default pss;
    property encType: TRSAEncType read FEncType write FEncType default oaep;
end;

```

The constructors and the destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TRSAKeyLength; modulus: string; publicExp: string; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to instantiate a public key
- **Constructor** Create(keyLength: TRSAKeyLength; modulus: string; publicExp: string; privateExp: string; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to instantiate a key pair
- **Destructor** Destroy; **override**; to zeroize the keys

The public methods are:

- **procedure GenerateKeys**; to generate the modulus, the public exponent and the private exponent
- **procedure GenerateKeysX509Compatible**(var dp, dq, p, q, inverseQ: string); to generate the modulus, the public exponent (fixed value 65537) and the private exponent, and also the private variables dp, dq, p, q and inverse (for interoperability with other libraries)
- **function Encrypt**(m: string): string; to encrypt the string m
- **function Decrypt**(m: string): string; to decrypt the string m
- **function Sign**(m: string): string; to sign the string m
- **function Verify**(m: string; s: string): Integer; to verify the signature s of the string m
- **function SignFile**(filePath: string): string; to sign a file
- **function VerifySignatureFile**(filePath, s: string): Integer; to verify the signature s of a file
- **procedure FromCertificate**(CertStr: string); to import a public key from a Certificate blob (base64, hex or raw format)
- **procedure FromCertificateFile**(CertFile: string); to import a public key from a Certificate file (PEM or other format)
- **procedure FromPrivateKey**(KeyStr: string); to import a key set (all parameters) from a private key (base64, hex or raw format)
- **procedure FromPrivateKeyFile**(filePath, Password: string); to import a key set (all parameters) from a private key file (PKIX format)
- **procedure FromPublicKey**(KeyStr: string); to import a public key from a blob (base64, hex or raw format)
- **procedure FromPublicKeyString**(key: string); to import a public key from a string (PKIX format)
- **procedure FromPublicKeyFile**(filePath, Password: string); to import a public key from a file (PKIX format)

The properties are:

- **property** Modulus: `string read FModulus write SetModulus`; to read and write the modulus
- **property** PublicExponent: `string read FPublicExponent write SetPublicExponent`; to read and write the public exponent (16 bytes)
- **property** PrivateExponent: `string read FPrivateExponent write SetPrivateExponent`; to read and write the private exponent
- **property** prime1: `string read FPrime1 write FPrime1`;
- **property** prime2: `string read FPrime2 write FPrime2`;
- **property** exponent1: `string read FExponent1 write FExponent1`;
- **property** exponent2: `string read FExponent2 write FExponent2`;
- **property** coefficient: `string read FCoefficient write FCoefficient`;

- **property** KeyLength: `TRSAKeyLength read FKeyLength write SetKeyLength`; to read and write the key length in bits (2048, 3072 or 4096 bits)
- **property** hashFunction: `TRSAHashFunction read FHashFunction write FHashFunction`; to choose the hash function (sha256, sha384 or sha512) used to hash in the different algorithms
- **property** MGFhash: `TRSAhashfunction read FMGFhash write FMGFhash default sha256`;
- **property** OAEPlabel: `string read FOAEPlabel write FOAEPlabel`;

- **property** InputFormat: `TConvertType read FInputFormat write FInputFormat`; to read and write the input format of the data (see Converter class section)
- **property** OutputFormat: `TConvertType read FOutputFormat write FOutputFormat`; to read and write the output format of the data (see Converter class section)
- **property** Unicode: `TUnicode read FUni write FUni`; to indicate whether the input buffer has Unicode characters
- **property** passwd: `string read Fpw Fpw write Fpw`; to get and set the password to decrypt an encrypted private key
- **property** ConstantTime: `boolean read FconstantTime write FConstantTime`; to indicate whether the encryption/decryption/signature/verification uses constant time implementation.
- **property** signType: `TRSASignType read FSignType write FSignType default pss`; to indicate whether the signature function uses PSS or PKCS v1.5 mode.
- **property** encType: `TRSAEncType read FEncType write FEncType default oaep`; to indicate whether the encryption function uses OAEP or PKCS v1.5 mode
- **property** pssSaltLen: `Integer read FPSSSaltLen write FPSSSaltLen default 20`;
- **property** PSSMGFtype: `Integer read FPSSMGFtype write FPSSMGFtype default 0`;

Example of how to encrypt with RSA

```
var
  rsa: TRSAEncSign;
  cipher: string;
begin
  rsa:= TRSAEncSign.Create;
  rsa.KeyLength:= kl2048;
  rsa.InputFormat:= raw; // default
  rsa.OutputFormat:= base64;
  rsa.GenerateKeys; // generates a key pair assigned to the object
  rsa.Unicode := noUni;
  rsa.encType := oaep;
  rsa.OAEPlabel := '1234';
  cipher:= rsa.Encrypt('test'); // input is raw
  rsa.Free;
end;
```

Example of how to sign with RSA

```
var
  rsa: TRSAEncSign;
  signature: string;
begin
  rsa:= TRSAEncSign.Create;
  rsa.KeyLength:= kl2048;
  rsa.InputFormat:= raw; // default
  rsa.OutputFormat:= base64;
  rsa.GenerateKeys;
  rsa.Unicode := yesUni;
  rsa.hashFunction := sha256;
  rsa.signType := pss;
  signature:= rsa.Sign('test');
  rsa.Free;
end;
```

All RSA functions/procedures are stored in the RSAObj file.

ECDSA, EdDSA, ECDH and ECIES

EdDSA is a digital signature algorithm using Edwards elliptic curves. It has been developed by a team directed by Daniel J. Bernstein. Three algorithms are implemented in this library, EdDSA25519, which public key is encoded on 256 bits, Ed448, which public key is encoded in 456 bits and EdDSA511187, which public key is encoded on 512 bits⁴.

Elliptic Curve Integrated Encryption Scheme (ECIES) is an asymmetric encryption scheme using elliptic curves. In this library, we have used Edwards curves, Curve25519 and Curve511187 because they ensure a good security level and allow us to reuse a part of the algorithms already implemented for EdDSA25119 and EdDSA511187.

ECDSA is a digital signature algorithm using elliptic curves cryptography. NIST standardized three curves, P-256, P-384 and P-521 in July 2013 in FIPS 186-4.

ECDH is an anonymous key agreement protocol that allows two parties, each having an elliptic-curve public-private key pair, to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key. The key, or the derived key, can then be used to encrypt subsequent communications using a symmetric-key cipher. It is a variant of the Diffie-Hellman protocol using elliptic-curve cryptography. ECDH is only implemented for the P-256, P-384 and P-521 curves.

The ECC (Elliptic Curve Cryptography) class is:

```
TECCType = (cc25519, cc448, cc511187, p256, p384, p521);
TNaCl = (naclno, naclyes);

TECCEncSign = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(AType: TECCType; PublicKey: string; NaCl: TNaCl;
        outputFormat: TConvertType; uni: TUnicode); overload;
    Constructor Create(AType: TECCType; PublicKey: string; PrivateKey: string;
        NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); overload;
    Destructor Destroy; override;

    procedure GenerateKeys;
    function Encrypt(m: string): string;
    function Decrypt(m: string): string;
    function Sign(m: string): string;
    function Verify(m: string; s: string): integer;
    function SignFile(filePath: string): string;
    function VerifySignatureFile(filePath, s: string): Integer;
    function GenerateSharedSecret(PeerPublicKey: string): string;
    procedure FromCertificate(CertStr: string);
    procedure FromCertificateFile(CertFile: string);
    procedure FromPrivateKey(KeyStr: string);
    procedure FromPrivateKeyFile(KeyFile: string);
    function FromPublicKey(KeyStr: string): string;
    function FromPublicKeyFile(KeyFile: string): string;

    // Added in 5.0.7.0
```

⁴ Ed511187 is not used in any standard yet, to our knowledge. There is no OID for this curve and therefore operations on certificates and PKIX objects at large are not possible.

```

procedure ToPrivateKeyFile(KeyFile: string);

// Added in 5.0.3.0
function FromSignatureFile(FilePath: string): string;
procedure ToSignatureFile(FilePath, s: string);
function FromJWKFile(FilePath: string): TECJWK;
function FromJWK(FilePath: string): TECJWK;
procedure ToJWK(FilePath, s1, s2: string); overload;
procedure ToJWK(FilePath: string; inJWK: TECJWK); overload;

published
property ECType: TECType read FECType write SetType default cc25519;
property PublicKey: string read FPublicKey write SetPublicKey;
property PrivateKey: string read FPrivateKey write SetPrivateKey;
property Context: string write SetContext;
property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
property NaCl: TNaCl read FNaCl write FNaCl default NaClno;
property Unicode: TUnicode read FUni write FUni default yesUni;
property Legacy: boolean read FLegacy write FLegacy default true;
property Password: string read FPassword write FPassword;
end;

```

The constructors and destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(AType: TECType; PublicKey: string; NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to instantiate a public key
- **Constructor** Create(AType: TECType; PublicKey: string; PrivateKey: string; NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to instantiate a key pair
- **Destructor** Destroy; **override**; to zero the keys

The public methods are:

- **procedure GenerateKeys**; to generate the public and private keys
- **function Encrypt**(m: string): string; to encrypt string m
- **function Decrypt**(m: string): string; to decrypt string m
- **function Sign**(m: string): string; to sign string m
- **function Verify**(m: string; s: string): Integer; to verify the signature s of string m
- **function SignFile**(filePath: string): string; to sign a file
- **function VerifySignatureFile**(filePath, s: string): Integer; to verify the signature s of a file
- **function GenerateSharedSecret**(PeerPublicKey: string): string; ECDH algorithm to compute a shared secret with an other entity, who gave us his/her public key. The other entity can compute the same secret key with our public key.
- **procedure FromCertificate**(CertStr: string); import a public key from a base64 certificate string

- **procedure** `FromCertificateFile`(CertFile: `string`); import a public key from a PEM certificate file
- **procedure** `FromPrivateKey`(KeyStr: `string`); import a private key from a base64 private key string
- **procedure** `FromPrivateKeyFile`(KeyFile: `string`); import a private key from a PEM private key file
- **function** `FromPublicKey`(KeyStr: `string`): `string`; extract a public key from a PEM public key
- **function** `FromPublicKeyFile`(KeyFile: `string`): `string`; extract a public key from a PEM public key file
- **procedure** `ToPrivateKeyFile`(KeyFile: `string`); save a private key to a PEM private key file

The properties are:

- **property** `ECCType`: `TECCType` `read` `FECCType` `write` `SetType`; to read and write the curve name
- **property** `PublicKey`: `string` `read` `FPublicKey` `write` `SetPublicKey`; to read and write the public key
- **property** `PrivateKey`: `string` `read` `FPrivateKey` `write` `SetPrivateKey`; to read and write the private key
- **property** `Context`: `string` `write` `SetContext`; is a specific parameter for the curve (see 8.3 of RFC 8032)
- **property** `OutputFormat`: `TConvertType` `read` `FOutputFormat` `write` `FoutputFormat`; to read and write the output format of the data (see Converter class section)
- **property** `NaCl`: `TNaCl` `read` `FNaCl` `write` `FNaCl`; to use an EdDSA algorithm interoperable with NaCl software library (available only for ed25519)
- **property** `Unicode`: `TUnicode` `read` `FUni` `write` `FUni`; to indicate whether the input buffer has Unicode characters
- **property** `Legacy`: `boolean` `read` `FLegacy` `write` `FLegacy` default `true`; to use the class as in 4.3.3
- **property** `Password`: `string` `read` `FPasswd` `write` `FPasswd`; to encrypt or decrypt PKIX file containing keys

Example of how to encrypt with ECIES

```
var
ecc: TECCEncSign;
cipher: string;
begin
ecc:= TECCEncSign.Create;
ecc.ECCType:= cc25519; // THIS IS THE ONLY SUPPORTED CURVE
                        // AS FOR NOW FOR ENCRYPTION
                        // OTHER CURVES CAN BE USED TO SIGN AND VERIFY
ecc.OutputFormat:= base64;
ecc.Unicode:= noUni;
ecc.NaCl := naclno;
ecc.GenerateKeys();
cipher:= ecc.Encrypt('test');
ecc.Free;
end;
```

Example of how to sign with EdDSA

```
var
  ecc: TECCEncSign;
  signature: string;
begin
  ecc:= TECCEncSign.Create;
  ecc.ECCType:= cc25519;
  ecc.OutputFormat:= base64;
  ecc.Unicode:= yesUni;
  ecc.NaCl := naclno;
  ecc.GenerateKeys;
  signature:= ecc.Sign('test');
  ecc.Free;
end;
```

Example of how to share a secret key with ECDH from a PEM peer public key

```
var
  ecc: TECCEncSign;
  sharedSecret: string;
begin
  ecc:= TECCEncSign.Create;
  ecc.ECCType:= p256;
  ecc.OutputFormat:= base64;
  ecc.Unicode:= yesUni;
  ecc.GenerateKeys;
  sharedSecret:= ecc.GenerateSharedSecret(ecc.FromPublicKey(PeerPublicKey));
  ecc.Free;
end;
```

All ECC functions/procedures are located in the ECCObj file.

SALSA

Salsa20 is a stream encryption algorithm proposed by Daniel Bernstein.

The SALSA class is:

```
TSalsaKeyLength = (skl128, skl256);

TSalsaEncryption = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(keyLength: TSalsaKeyLength; key: string;
    outputFormat: TConvertType; uni: TUnicode); overload;
  Destructor Destroy; override;
  function Encrypt(s: string): string;
  function Decrypt(s: string): string;
  procedure EncryptFile(s, o: string);
  procedure DecryptFile(s, o: string);
  procedure EncryptStream(s: TStream; var o: TStream);
  procedure DecryptStream(s: TStream; var o: TStream);
published
  property key: string read FKey write SetKey;
  property keyLength: TSalsaKeyLength read FKeyLength write SetKeyLength
    default skl128;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property Unicode: TUnicode read FUni write FUni default yesUni;
  property Progress: Integer read FProgress write SetProgress;
  property OnChange: TNotifyEvent write FOnChange;
end;
```

The constructors and destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(keyLength: TSalsaKeyLength; key: string; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to set all the parameters
- **Destructor** Destroy; **override**; to zeroize the key

The public methods are:

- **function** Encrypt(s: string): string; to encrypt the string s
- **function** Decrypt(s: string): string; to decrypt the string s
- **procedure** EncryptFile(s, o: string); to encrypt the file whose path is s in the file whose path is o
- **procedure** DecryptFile(s, o: string); to decrypt the file whose path is s in the file whose path is o
- **procedure** EncryptStream(s: TStream; var o: TStream); to encrypt the stream s in the stream o
- **procedure** DecryptStream(s: TStream; var o: TStream); to decrypt the stream s in the stream o

The properties are:

- **property** Key: `string` **read** FKey **write** SetKey; to read and write the key
- **property** KeyLength: `TSalsaKeyLength` **read** FKeyLength **write** SetKeyLength; to read and write the key length in bits (256 or 512 bits)
- **property** OutputFormat: `TConvertType` **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** Unicode: `TUnicode` **read** FUni **write** FUni; to indicate whether the input buffer or the file name has Unicode characters
- **property** Progress: `Integer` **read** FProgress **write** SetProgress; to indicate progress during encryption / decryption of a stream
- **property** OnChange: `TNotifyEvent` **write** FOnChange; to indicate progress changes

Example of how to encrypt with SALSA

```
var
  salsa: TSalsaEncryption;
  cipher: string;
begin
  salsa := TSalsaEncryption.Create;
  salsa.KeyLength := skl128;
  salsa.Key := '0123456789012345';
  salsa.Unicode := yesUni;
  salsa.OutputFormat := hexa;
  cipher := salsa.Encrypt('test');
  salsa.Free;
end;
```

All SALSA functions/procedures are located in the SALSAObj file.

xCHACHA20

xChaCha20 is another stream encryption algorithm proposed by Daniel Bernstein.

The xChaCha20 class is:

```
TXChaCha = class(TTMSCryptBase)
  private
    FCtx: XChaChaCtx;
    FKey: string;
    FIV: string;

    procedure DoSetUp;
    procedure Hchacha20(var output: TBytes; input, key: TBytes);
    procedure Setkey(Value: string);
    procedure SetIV(Value: string);
    procedure QUARTERROUND(var a,b,c,d: UInt32);
    function PLUS(v: UInt32; w: UInt32): UInt32;
    function PLUSONE(v: UInt32): UInt32;
    procedure UInt32ToByte(var p: Tbytes; i: integer; v: UInt32);
    function ByteToUInt32(p: Tbytes; I: integer): UInt32;

  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create(InKey, InIV: array of byte); reintroduce; overload;
    Destructor Destroy; override;

    procedure SetCounter(counter: TBytes);
    procedure EncryptBytes(PlainText: TBytes; var CipherText: TBytes); overload;
    procedure EncryptBytes(PlainText: array of byte;
      var CipherText: TBytes); overload;
    procedure DecryptBytes(CipherText: TBytes; var PlainText: TBytes);
    procedure KeyStreamBytes(var KeyStream: TBytes);
    procedure Clear;

  published
    property key: string read FKey write SetKey;
    property IV: string read FIV write SetIV;
    // property Progress: Integer read FProgress write SetProgress;
    // property OnChange: TNotifyEvent write FOnChange;
end;
```

The constructors and destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create(InKey, InIV: array of byte); **reintroduce**; **overload**;
- **Destructor** Destroy; **override**; to zeroize the key

The public methods are:

- **procedure** SetCounter(counter: TBytes); to set the internal counter to a specific number
- **procedure** EncryptBytes(PlainText: TBytes; var CipherText: TBytes); **overload**; to encrypt a stream of bytes
- **procedure** EncryptBytes(PlainText: array of byte; var CipherText: TBytes); **overload**; to encrypt a stream of bytes

- **procedure** `DecryptBytes`(CipherText: `TBytes`; var PlainText: `TBytes`); to decrypt a stream of bytes
- **procedure** `KeyStreamBytes`(var KeyStream: `TBytes`); to get a key stream
- **procedure** `Clear`; to clear the cipher context and reset

The properties are:

- **property** `Key`: `string read` `FKey` `write` `SetKey`; to read and write the key
- **property** `IV`: `string read` `FIV` `write` `SetIV`; to read and write the IV
- **property** `Progress`: `Integer read` `FProgress` `write` `SetProgress`; to indicate progress during encryption / decryption of a stream (on hold)
- **property** `OnChange`: `TNotifyEvent write` `FOnChange`; to indicate progress changes (on hold)

Example of how to encrypt with xCHACHA20

```
var
  chacha: TXChaCha;
  cipher: TBytes;
begin
  chacha := TXChaCha.Create;
  chacha.Key := '01234567890123450123456789012345';
  cipher := salsa.EncryptBytes('test', cipher);
  chacha.Free;
end;
```

All xCHACHA20 functions/procedures are located in the xChaCha20 file.

SHA-2

SHA-2 (Secure Hash Algorithm) is a family of hash functions that have been designed by the National Security Agency (NSA) of the USA, on the model of the now deprecated SHA-1 and SHA-0 functions. The algorithms of the SHA-2 family, SHA-224, SHA-256, SHA-384 and SHA-512 are described and published along with SHA-1 in FIPS 180-2 (Secure Hash Standard). In this library, only SHA-256, SHA-384 and SHA-512 have been implemented.

SHA-2 is described in NIST FIPS PUB 180-4.

The SHA2 class is:

```
TSHA2Hash = class(TSHA2)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
    uni: TUnicode); overload;
  function Hash(s: string): string;
  function HashFile(s: string): string;
  function HashStream(s: TStream): string;
  function HMAC(s, k: string): string;
  function VerifyHMAC(s, k, h: string): Integer;
published
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property Unicode: TUnicode read FUni write FUni default yesUni;
  property Progress: Integer read FProgress write SetProgress;
  property OnChange: TNotifyEvent write FOnChange;
end;
```

It inherits from TSHA2 in SHA2Core.pas that has 2 properties.

- **property** hashSize: THashSize **read** FHashSize **write** SetHashSize; to set the hash size
- **property** HBlockSize: integer **read** FHBlockSize **write** FHBlockSize **default** 32; to set the hash block size (transparent to the User)

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to set all the parameters

The public methods are:

- **function** Hash(s: string): string; to hash string s
- **function** HashFile(s: string): string; to hash the file which path is s
- **function** HashStream(s: TStream): string; to hash stream s
- **function** HMAC(s, k: string): string; to generate a hmac from string s and key k
- **function** VerifyHMAC(s, k, h: string): Integer; to verify the hmac h associated with the string s and key k

The properties are:

- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during hashing of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate that the progress changes

Example of how to hash with SHA-2

```
var
  sha2: TSHA2Hash;
  hash: string;
begin
  sha2:= TSHA2Hash.Create;
  sha2.HashSize:= hsha256;
  sha2.InputFormat:= raw;
  sha2.OutputFormat:= hexa;
  sha2.Unicode:= noUni;
  hash:= sha2.Hash('test');
  sha2.Free;
end;
```

Example of how to generate a HMAC with SHA-2

```
var
  sha2: TSHA2Hash;
  hash: string;
  k: string;
begin
  sha2:= TSHA2Hash.Create;
  sha2.HashSize:= hsha256;
  sha2.InputFormat:= raw;
  sha2.OutputFormat:= hexa;
  sha2.Unicode := yesUni;
  k:= '0123456789012345';
  hash:= sha2.HMAC('test', k); // data and k shall have the same format
                                // (raw here)
  sha2.Free;
end;
```

All HASH functions/procedures are located in the HashObj file.

SHA-3

SHA-3 comes from the NIST hash function competition which selected the Keccak algorithm on October 2, 2012. It is not intended to replace SHA-2 but to provide an alternative following the possibilities of attacks on the deprecated MD5, SHA-0 and SHA-1 standards. This library allows hashing with the SHA-3 standard algorithm but also with the SHA-3 XOF (Extendable-Output Function) algorithm which allows to have a variable length output. SHA-3 is described in FIPS PUB 202. This library includes the SHA3 derived functions cSHAKE, KMAC and TupleHash, described in NIST Special Publication (SP) 800-185.

cSHAKE allows a user to add a salt to the hashed output. KMAC provides a pseudorandom function and a keyed hash function with variable-length outputs. TupleHash provides a function that hashes tuples of input [strings](#) correctly and unambiguously.

The SHA3 class is:

```
TSHA3Type = (tsha, txof);

TSHA3Hash = class(TSHA3Core)
  private
    FHashSize: Integer;
    FType: TSHA3Type;
    FVersion: Integer;
    FUni: TUnicode;
    FInputFormat: TConvertType;
    FOutputFormat: TConvertType;
    FProgress: Integer;
    FOnChange: TNotifyEvent;
    SHA3: TSHA3Core;

    procedure SetHashSize(const Value: Integer);
    procedure SetType(const Value: TSHA3Type);
    procedure SetVersion(const Value: Integer);
    procedure SetProgress(P: Integer);

  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
      uni: TUnicode); overload;
    Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
      uni: TUnicode; version: Integer); overload;
    function Hash(s: string): string;
    function HashFile(s: string): string;
    function cSHAKEHash(s, salt: string): string;
    function KMACHash(k, s, salt: string): string;
    function TupleHash(s: array of string; salt: string): string;
    function HMAC(s, k: string): string;
    function VerifyHMAC(s, k, h: string): Integer;
  published
    property AType: TSHA3Type read FType write SetType default tsha;
    default 256;
    property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
```

```
property Unicode: TUnicode read FUni write FUni default yesUni;
property Progress: Integer read FProgress write SetProgress;
property OnChange: TNotifyEvent write FOnChange;
end;
```

The class inherits from TSHA3Core and has 4 properties:

- **property** hashSize: Integer **read** FHashSize **write** SetHashSize to read and write the number of bits (224, 256, 384 or 512 bits in classical SHA-3, any value in extended SHA-3) of the hash
- **property** xof: boolean **read** FXOF **write** SetXOF default false;
- **property** Shake: TShake **read** FShake **write** FShake default hss128;
- **property** version: Integer **read** FVersion **write** FVersion default 256;

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor in tsha mode
- **Constructor** Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode; version: Integer); **overload**; the constructor in txof mode

The public methods are:

- **function** Hash(s: string): string; to hash the string s
- **function** HashFile(s: string): string; to hash the file which path is s
- **function** cSHAKEHash(s, salt: string): string; to hash string s with a salt – to be used with txof mode
- **function** KMACHash(k, s, salt: string): string; to hash string s with a salt and key k – to be used with txof mode
- **function** TupleHash(s: array of string; salt: string): string; to hash the array of strings s with a salt – to be used with txof mode
- **function** HMAC(s, k: string): string; to generate a hmac from string s and key k
- **function** VerifyHMAC(s, k, h: string): Integer; to verify the hmac h associated with string s and key k

The properties are:

- **property** Version: Integer **read** FVersion **write** SetVersion; to read and write the version (256 or 512) in case of extended type
- **property** AType: TSHA3Type **read** FType **write** SetType; to read and write the type, classical or extended.
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the output format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FOutputFormat; to read and write the output format of the data (see Converter class section)
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer or the file name has Unicode characters

- `property` Progress: Integer `read` FProgress `write` SetProgress; to indicate progress during hashing of a stream
- `property` OnChange: TNotifyEvent `write` FOnChange; to indicate that the progress changes

Example of how to hash with SHA-3

```
var
  sha3: TSHA3Hash;
  hash: string;
begin
  sha3:= TSHA3Hash.Create;
  sha3.AType:= txof;
  sha3.Version:= 512;
  sha3.InputFormat:= raw;
  sha3.OutputFormat:= base64;
  sha3.Unicode := yesUni;
  hash:= sha3.Hash('test');
  sha3.Free;
end;
```

Example of how to generate a HMAC with SHA-3

```
var
  sha3: TSHA3Hash;
  hash: string;
  k: string;
begin
  sha3:= TSHA3Hash.Create;
  sha3.AType:= tsha;
  sha3.InputFormat:= raw;
  sha3.OutputFormat:= hexa;
  sha3.Unicode := yesUni;
  k:= '0123456789012345';
  hash:= sha3.HMAC('test', k); // default hash size is 256 bits
  sha3.Free;
end;
```

All HASH functions/procedures are located in the HashObj file.

SPECK

Speck is a family of lightweight block ciphers publicly released by the National Security Agency (NSA) in June 2013. Speck has been optimized for performance in software implementations. Speck is an Add-Rotate-Xor (ARX) cipher.

Speck supports a variety of block and key sizes. A block is always two words, but the words may be 16, 24, 32, 48 or 64 bits in size. The corresponding key is 2, 3 or 4 words. The round function consists in two rotations, adding the right word to the left word, xoring the key into the left word, then and xoring the left word to the right word.

To encrypt a message with many blocks, we will use the following modes (like the AES):

- ECB (Electronic Code Book), PKCS#7 padding
- CBC (Cipher Block Chaining), PKCS#7 padding

The version only supports 64 bit words (128 bit blocks) and 128, 192 and 256 bit keys.

The SPECK class is:

```
TSPECKType = (stECB, stCBC);
TSPECKIVMode = (rand, userdefined);
TSPECKPaddingMode = (PKCS7, nopadding);

TSPECKEncryption = class(TTMSCryptBase)
public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(key: string; AType: TSPECKType;
        OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode;
        uni: TUnicode); overload;
    Constructor Create(key: string; AType: TSPECKType;
        OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode;
        IV: string); overload;
    Destructor Destroy; override;
    function Encrypt(s: string): string;
    function Decrypt(s: string): string;
    procedure EncryptFileW(s, o: string);
    procedure DecryptFileW(s, o: string);
    procedure EncryptStream(s: TStream; var o: TStream);
    procedure DecryptStream(s: TStream; var o: TStream);
published
    property key: string read FKey write SetKey;
    property AType: TSPECKType read FType write FType default stcbc;
    property InputFormat: TConvertType read FInputFormat write FInputFormat
        default raw;
    property OutputFormat: TConvertType read FOutputFormat write FOutputFormat
        default hexa;
    property IVMode: TSPECKIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property paddingMode: TSPECKPaddingMode read FPaddingMode
        write FPaddingMode default PKCS7;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
end;
```

The constructors and destructor are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(key: string; AType: TSPECKType; OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode); **overload**; the constructor to set the parameters with IVMode = rand
- **Constructor** Create(key: string; AType: TSPECKType; OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode; IV: string); **overload**; the constructor to set the parameters with IVMode = userdefined
- **Destructor** Destroy; **override**; to zero the key

The public methods are:

- **function** Encrypt(s: string): string; to encrypt string s
- **function** Decrypt(s: string): string; to decrypt string s
- **procedure** EncryptFilew(s, o: string); to encrypt the file which path is s and the encrypted file path is o
- **procedure** DecryptFilew(s, o: string); to decrypt the file which path is s and the decrypted file path is o
- **procedure** EncryptStream(s: TStream; var o: TStream); to encrypt stream s into stream o
- **procedure** DecryptStream(s: TStream; var o: TStream); to decrypt stream s into stream o

The properties are:

- **property** Key: string **read** FKey **write** SetKey; to read and write the key
- **property** AType: TSPECKType **read** FType **write** FType; to read and write the encryption mode (ECB, CBC or OFB)
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** IVMode: TSPECKIVMode **read** FIVMode **write** FIVMode; to read and write the IV mode, userdefined or rand.
- **property** IV: string **read** FIV **write** SetIV; to read and write the IV of FwordSizeBits/4 bytes if the IV mode is userdefined (in rand mode, the IV is randomly generated and added to the encrypted text)
- **property** PaddingMode: TSPECKPaddingMode **read** FPaddingMode **write** FpaddingMode;
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer or the file name has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during encryption / decryption of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate that the progress changes

Example of how to encrypt with SPECK

```
var
  speck: TSPECKEncryption;
  cipher: string;
begin
  speck := TSPECKEncryption.Create;
  speck.AType := stCBC;
  speck.Key := '0123456789012345';
  speck.InputFormat := raw;
  speck.OutputFormat := hexa;
  speck.Unicode := noUni;
  speck.PaddingMode := TSPECKPaddingMode.PKCS7;
  speck.IVMode := TSPECKIVMode.rand;
  cipher := speck.Encrypt('test');
  speck.Free;
end;
```

All SPECK functions/procedures are located in the SpeckObj file.

PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is a key derivation function that is part of RSA Laboratories' Public-Key Cryptography Standards (PKCS) series, specifically PKCS#5 v2.0, also published as Internet Engineering Task Force's RFC 2898. It replaces an earlier standard, PBKDF1, which could only produce derived keys up to 160 bits long.

PBKDF2 applies a pseudorandom function, such as a cryptographic hash, cipher, or HMAC, to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult and is known as key stretching.

It is described in NIST Special Publication 800-132.

The PBKDF2 class is:

```
THashFunction = (hsha1, hsha2, hsha3);

TPBKDF2KeyDerivation = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(outputSizeBits: Integer; salt: string; counter: Integer;
    outputFormat: TConvertType; uni: TUnicode, hashF: THashFunction;
    hashSB: Integer); overload;
  function GenerateKey(s: string): string;
published
  property outputSizeBits: Integer read FOutputSizeBits
    write SetOutputSizeBits default 128;
  property Salt: string read FSalt write FSalt;
  property counter: Integer read FCounter write FCounter default 10000;
  property hashFunction: THashFunction read FHashFunction write FHashFunction
    default hsha2;
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
    default 256;
  property Unicode: TUnicode read FUni write FUni default yesUni;
end;
```

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(outputSizeBits: Integer; salt: string; counter: Integer; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to set all the parameters

The public method is:

- **function** GenerateKey(s: string): string; to generate a key from a password s

The properties are:

- **property** OutputSizeBits: Integer **read** FOutputSizeBits **write** SetOutputSizeBits; to read and write the output length in bits
- **property** Salt: string **read** FSalt **write** FSalt; to read and write the salt
- **property** Counter: Integer **read** FCounter **write** FCounter; to read and write the number of iterations of the algorithm
- **property** hashFunction: THashFunction **read** FHashFunction **write** FHashFunction; to read and write the hash function used into PBKDF2 algorithm
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FOutputFormat; to read and write the output format of the data (see Converter class section)
- **property** hashSizeBits: Integer **read** FHashSizeBits **write** SetHashSizeBits; to read and write the number of output bits of the hash function used in PBKDF2 algorithm
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer has Unicode characters

Example of how to generate a key from a password with PBKDF2

```
var
  pbkdf2: TPBKDF2KeyDerivation;
  output: string;
begin
  pbkdf2:= TPBKDF2KeyDerivation.Create;
  pbkdf2.OutputSizeBits:= 1024;
  pbkdf2.Counter:= 10000;
  pbkdf2.Unicode:= yesUni;
  pbkdf2.InputFormat:= raw; // password and salt
  pbkdf2.OutputFormat:= base64;
  pbkdf2.Salt:= '012345678901234567890123456789012345678901234567890123456789';
  output:= pbkdf2.GenerateKey('test123');
  pbkdf2.Free;
end;
```

PBKDF2 functions/procedures are located in the PBKDF2.pas file.

HKDF

HKDF (HMAC-based Extract-and-Expand Key Derivation Function) is a simple key derivation function (KDF) based on a hash-based message authentication code (HMAC). The main approach HKDF follows is the "extract-then-expand" paradigm, where the KDF logically consists of two modules: the first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key, and then the second stage "expands" this key into several additional pseudorandom keys (the output of the KDF).

It can be used, for example, to convert shared secrets exchanged via Diffie–Hellman into key material suitable for use in encryption, integrity checking or authentication.

It is formally described in RFC 5869.

The HKDF class is:

```
THashFunction = (hsha2, hsha3);

THKDFKeyDerivation = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(outputFormat: TConvertType; uni: TUnicode,
    hashF: THashFunction; hashSB: Integer); overload;
  function Extract(s, salt: string): string;
  function Expand(s, info: string; len: Integer): string;
published
  property hashFunction: THashFunction read FHashFunction write FHashFunction
    default hsha2;
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
    default 256;
  property Unicode: TUnicode read FUni write FUni default yesUni;
end;
```

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(outputFormat: TConvertType; uni: TUnicode; hashF: THashFunction; hashSB: Integer); **overload**; the constructor to set all parameters

The public methods are:

- **function** Extract(s, salt: string): string; to generate a pseudo random key from a message s and a salt
- **function** Expand(s, info: string; len: Integer): string; to generate a key of length len from the resulting key of Extract function, s, and an info string

The properties are:

- **property** hashFunction: THashFunction **read** FHashFunction **write** FHashFunction; to read and write the hash function used into HKDF algorithm
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** hashSizeBits: Integer **read** FHashSizeBits **write** SetHashSizeBits; to read and write the number of output bits of the hash function used in HKDF algorithm
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer has Unicode characters

Example of how to generate a key from a password with HKDF

```
var
  hkdf: THKDFKeyDerivation;
  PRK, OKM: string;
begin
  hkdf:= THKDFKeyDerivation.Create;
  hkdf.Unicode:= yesUni;
  hkdf.InputFormat:= raw; // password and salt
  hkdf.OutputFormat:= base64;
  hkdf.hashFunction := hsha2;
  hkdf.hashSizeBits := 256;
  PRK:= hkdf.Extract('test123', '123456');
  OKM := hkdf.Expand(PRK, 'WebPush: info', 32);
  hkdf.Free;
end;
```

All KEY DERIVATION functions/procedures are located in the HashObj file.

Blake2

BLAKE2 is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3, yet is at least as secure as the latest standard SHA-3. BLAKE2 has been adopted by many projects due to its high speed, security, and simplicity. BLAKE2 is specified in RFC 7693. We have chosen to implement BLAKE2b that is optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes.

The Blake2B class is:

```
TBlake2BHash = class(TBlake2Core)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(hashSizeBytes: Integer; key: string;
    outputFormat: TConvertType; uni: TUnicode); overload;
  function Hash(s: string): string;
  function HashFile(s: string): string;
  function HashStream(s: TStream): string;
published
  property hashSizeBytes: Integer read FHashSizeBytes write SetHashSizeBytes
    default 16;
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;
  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;
  property key: String read FKey write SetKey;
  property Unicode: TUnicode read FUni write FUni default yesUni;
  property Progress: Integer read FProgress write SetProgress;
  property OnChange: TNotifyEvent write FOnChange;
end;
```

The class inherits from TBlake2Core.

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(hashSizeBytes: Integer; key: string; outputFormat: TConvertType; uni: TUnicode); **overload**; the constructor to set all the parameters

The public methods are:

- **function** Hash(s: string): string; to hash string s
- **function** HashFile(s: string): string; to hash a file whose path is s
- **function** HashStream(s: TStream): string; to hash stream s

The properties are:

- **property** HashSizeBytes: Integer **read** FHashSizeBytes **write** SetHashSizeBytes; to read and write the hash size in bytes

- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the input format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FoutputFormat; to read and write the output format of the data (see Converter class section)
- **property** Key: String **read** FKey **write** SetKey; to read and write the optional key
- **property** Unicode: TUnicode **read** FUni **write** FUni; to indicate whether the input buffer or the file name has Unicode characters
- **property** Progress: Integer **read** FProgress **write** SetProgress; to indicate progress during hashing of a stream
- **property** OnChange: TNotifyEvent **write** FOnChange; to indicate that the progress changes

Example of how to hash a **string** with Blake2B

```
var
  blake2B: Tblake2BHash;
  output: String;
begin
  blake2B:= Tblake2BHash.Create;
  try
    blake2B.Key:= '';
    blake2B.HashSizeBytes:= 64;
    blake2B.InputFormat:= raw; // password
    blake2B.OutputFormat:= hexa;
    blake2B.Unicode := yesUni;

    output:= blake2B.Hash('ABCDEFGH');

  finally
    blake2B.Free;
  end;
end;
```

All HASH functions/procedures are in the Blake2BCore.pas file.

~~RIPEMD-160~~

NOTE: This algorithm is not in version 5 of TMS CP. It will NOT be added back. Other hash functions will be considered for version 5.3 (e.g., RIPEMD-256, Blake3).

Argon2

Argon2 is a key derivation function that was selected as the winner of the Password Hashing Competition (PHC) in July 2015. It was designed by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich from the University of Luxembourg. Argon2 provides two related versions:

- Argon2d maximizes resistance to GPU cracking attacks.
- Argon2i is optimized to resist side-channel attacks.

We have chosen to implement Argon2d with no parallelism.

The Argon2 class is:

```
TArgon2KeyDerivation = class(TBlake2Core)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(outputSizeBytes: Integer; salt: string; counter: Integer;
    outputFormat: TConvertType; memory: Integer; uni: TUnicode); overload;
  function GenerateKey(s: string): string;
published
  property outputSizeBytes: UInt32 read FVariables.OutputSizeBytes
    write FVariables.OutputSizeBytes default 32;
  property inputFormat: TConvertType read FInputFormat write FInputFormat
    default raw;

  property outputFormat: TConvertType read FOutputFormat write FOutputFormat
    default hexa;

  property Salt: string read FVariables.Salt write FVariables.Salt;
  property counter: UInt32 read FVariables.Counter
    write FVariables.Counter;

  property memory: UInt32 read FVariables.Memory write FVariables.Memory;
end;
```

The class inherits from TBlake2Core.

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(outputSizeBytes: Integer; salt: string; counter: Integer; outputFormat: TConvertType; memory: Integer; uni: TUnicode); **overload**; the constructor to set all the parameters

The public method is:

- **function** GenerateKey(s: string): string; to generate a key from a password s

The properties are:

- **property** OutputSizeBytes: UInt32 **read** FVariables.OutputSizeBytes **write** FVariables.OutputSizeBytes; to read and write the output length in bits (default is 32)
- **property** Salt: string **read** FVariables.Salt **write** FVariables.Salt; to read and write the salt (16 bytes in string form)

- **property** Counter: UInt32 **read** FVariables.Counter **write** FVariables.Counter; to read and write the number of iterations of the algorithm (minimum 1)
- **property** InputFormat: TConvertType **read** FInputFormat **write** FInputFormat; to read and write the output format of the data (see Converter class section)
- **property** OutputFormat: TConvertType **read** FOutputFormat **write** FOutputFormat; to read and write the output format of the data (see Converter class section)
- **property** Memory: Integer **read** FVariables.Memory **write** FVariables.Memory; to read and write the amount of memory you want to use in KB (minimum 8)

Example of how to generate a key from a password with Argon2

```
var
  argon2: TArgon2KeyDerivation;
  output: String;
begin
  argon2 := TArgon2KeyDerivation.Create;
  try
    argon2.InputFormat := raw; // key and salt
    argon2.OutputFormat := base64;
    argon2.OutputSizeBytes := 64;
    argon2.Counter := 10;
    argon2.Memory := 16;
    argon2.Unicode := yesUni;
    argon2.Salt := 'ABCDEFGHJKLMNOP';

    output := argon2.GenerateKey('toto23');
  finally
    argon2.Free;
  end;
end;
```

Argon2d functions/procedures are in the Argon2d.pas file.

TLSH

TLSH is an acronym for Trend Micro Locality Sensitive. The TMS Cryptographic Pack version of TLSH is a simplified version of Trend Micro's Jonathan Oliver, Chun Cheng, and Yanggui Chen paper and Git Repository:

- <https://documents.trendmicro.com/assets/wp/wp-locality-sensitive-hash.pdf>
- <https://github.com/trendmicro/tlsh>

TLSH generates a hash value which can be used for similarity comparisons. Similar objects will have similar hash values which allows for the detection of similar objects by comparing their hash values. Note that the byte stream should have enough complexity. For example, a byte stream of identical bytes will not generate a hash value.

NOTE: TLSH is NOT a cryptographic hash function.

The TLSH class is:

```
Ttlsh = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; reintroduce; overload;
  destructor Destroy; override;

  function Checksum(k: integer): integer;
  function BucketValue(Bucket: integer): integer;
  function ModDiff(X, Y, R: integer): integer;
  function HDistance(Length: integer; T: Ttlsh): integer;

  function GetHash: string; overload;
  function GetHash(Buffer: array of byte; Size: integer): string; overload;
  function GetHash(FileName: String): string; overload;
published
  property tlsHash: string read GetHash;
  property comment: string read FComment;
  property Lvalue: integer read getLvalue default 0;
  property Q1ratio: integer read getQ1ratio default 0;
  property Q2ratio: integer read getQ2ratio default 0;
  property TLSHversion: string read getVersion;
  property ShowVersion: boolean read FVer write FVer default False;
end;

function FileDistance(MyFile1, MyFile2: String; LenDiff: boolean): integer;
```

The constructors are:

- **Constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **Constructor** Create; reintroduce; overload; the default constructor

The public methods are:

- **function** Checksum(k: integer): integer; return the k-th value of checksum array
- **function** BucketValue(Bucket: integer): integer; return the Bucket-th value of Bucket array

- **function** ModDiff(X, Y, R: **integer**): **integer**; the minimum number of steps between X and Y on a circular queue of size R
- **function** HDistance(Length: **integer**; T: Ttlsh): **integer**; return the distance between the current TLSH object and the T one, truncated by Length elements.
- **function** GetHash: **string**; **overload**; return the Tlsh hash value
- **function** GetHash(Buffer: **array of byte**; Size: **integer**): **string**; **overload**; return the hash of Buffer with size Size.
- **function** GetHash(FileName: **String**): **string**; **overload**; return the hash of the file FileName.

The properties are:

- **property** tlshHash: **string read** GetHash; the resulting hash
- **property** comment: **string read** FComment; an error message explaining why the hash value is empty
- **property** Lvalue: **integer read** getLvalue **default** 0; to get LValue TLSH parameter that is a representation of the logarithm of the byte string length (modulo 256)
- **property** Q1ratio: **integer read** getQ1ratio **default** 0; Q1ratio is the rightmost part of the third byte that is constructed out of two 16 bit quantities derived from the quartiles: q1, q2 and q3
- **property** Q2ratio: **integer read** getQ2ratio **default** 0; Q2ratio is the leftmost part of the third byte that is constructed out of two 16 bit quantities derived from the quartiles: q1, q2 and q3
- **property** TLSHversion: **string read** getVersion; to get the TLSH version
- **property** ShowVersion: **boolean read** FVer **write** FVer **default** **False**; to set whether we put the version in the first byte of resulting hash

There is also a function outside the class:

- **function** FileDistance(MyFile1, MyFile2: **String**; LenDiff: **boolean**): **integer**; to compute the TLSH distance between MyFile1 and MyFile2 files

Example of how to hash a file with TLSH

```
var
  tlsh: TTLSH;
  output: String;
begin
  tlsh:= TTLSH.Create;
  try
    tlsh.ShowVersion:= false;
    output:= tlsh.GetHash('myfile.txt');
  finally
    tlsh.Free;
  end;
end;
```

Example of how to compare 2 files with TLSH

```
var
  tlsh: TTLSH;
  output: Integer;
begin
  tlsh:= TTLSH.Create;
  try
    output := FileDistance('myfile1.txt', 'myfile2.txt', false);
  finally
    tlsh.Free;
  end;
end;
```

All TLSH functions/procedures are in the Tlsh file.

Converter class

NOTE: Please check MiscObj.pas for the latest series of procedures and functions.

To display the output binary data of the library functions on a screen, we need to convert them in a printable format. We have chosen four formats:

- Hexadecimal format: consists in replacing each 4-bit block by a symbol in the list 0, ..., 9, A, ..., F.
- Base64 format: consists in replacing each 6-bit block by a symbol in the list a, ..., z, A, ..., Z, 0, ..., 9, + and / (the symbol = is used in complement when the length of the data is not a multiple of 3 bytes).
- Base64url format: the same as Base64 with - in place of + and _ in place of /, to be compatible with URLs.
- Base32 format: consists in replacing each 5-bit block by a symbol in the list A, ..., Z, 2, ..., 7 (the symbol = is used in complement when the length of the data is not a multiple of 8 bytes).

We add the raw format to have an output format compatible with the input of some functions.

The TConvert class contains the following:

```
TConvertType = (base64, hexa, base64url, base32, raw);
TUnicode = (noUni, yesUni);

TConvert = class(TTMSCryptBase)
public
  Constructor Create(AOwner: TComponent); overload; override;
  Constructor Create; overload;
  Constructor Create(AType: TConvertType); overload;

  function CharToFormat(charstring: PAnsiChar; charlen: Integer): string;
  function FormatToChar(str: string): PAnsiChar;
  function UnicodeToPAnsiChar(str: string): PAnsiChar;
  function PAnsiCharFromUnicodeLength(str: string): Integer;
  function StringToBuffer(str: string; u: boolean; var msgLen: Integer)
    : PAnsiChar;
  function StringToBufferA(str: string; u: boolean): PAnsiChar;
  function StringToBufferA(str: string; u: TUnicode): PByte;
  function UnicodeStringToFormat(str: string): string;
  function FormatToUnicodeString(str: string): string;
  function StringToByteArray(str: string): TArray<Byte>;
  function TestUnicode(str: string): Integer;
  function StringToUnicode(str: string): string;
  function StringToFormat(charstring: string): string;
  function FormatToString(str: string): string;
  function OutputFormatLength(charlen: Integer): Integer;
  function CharLength(charstring: string): Integer;
  function Base64ToHexa(base64String: string): string;
  function HexaToBase64(hexaString: string): string;
  function Base64ToBase64url(inString: string): string;
  function Base64urlToBase64(inString: string): string;
  function Base64urlToHexa(inString: string): string;
```

```

function HexaToBase64url(inString: string): string;
function Base32ToHexa(base32String: string): string;
function HexaToBase32(hexaString: string): string;
function Base32ToBase64url(inString: string): string;
function Base64urlToBase32(inString: string): string;
function Base32ToBase64(inString: string): string;
function Base64ToBase32(inString: string): string;
function KeyRSAOpenSSLToKeyTRSAEncSign(strKey: string): string;
function KeyTRSAEncSignToKeyRSAOpenSSL(strKey: string): string;
function Base58Encode(const value: uint64): string;
function Base58Decode(const encoded: string): uint64;
function TBytesToString(const t: TBytes): string;
function StringToTBytes(const str: string): TBytes;
function RandomString(len: Integer): string;

procedure ToCharString(InHexString: string; var OutCharString: string);
overload;
procedure ToCharString(InHexString: string; var OutCharArray: TBytes);
overload;
function ToCharString(InHexString: string): string; overload;

procedure ToHexString(InCharString: string; var OutHexString: string);
overload;
procedure ToHexString(InCharString: string; var OutHexArray: TBytes);
overload;
function ToHexString(InCharString: string): string; overload;

procedure ToBase32String(InCharString: string; var Base32String: string);
procedure ToBase64String(InCharString: string; var Base64String: string);
procedure ToBase64urlString(InCharString: string; var Base64String: string);
procedure Base32ToChar(Base32String: string; var OutCharString: string);
procedure Base64ToChar(Base64String: string; var OutCharString: string);
procedure Base64urlToChar(Base64urlString: string; var OutCharString:
string);
procedure FormatBeforeConvert(InCharString: string; var OutCharStringString:
string); overload;
function FormatBeforeConvert(InCharString: string; aType: TConvertType):
string; overload;
function StringToByteArray(str: string; u: TUnicode): TBytes; overload;
function UnicodeToByteArray(str: string): TBytes;

function BigHexToBigInt(s: string): string;
function Add(x, y: string): string;
function HexToInt(a: char): integer;

function Reverse(inString: string): string;
function ToUtf8(s: string): string;

published
property AType: TConvertType read FType write FType default hexa;
end;

```

The constructors are:

- **Constructor** Create(AOwner: TComponent); **overload**; **override**; the default constructor from the TComponent class
- **Constructor** Create; **overload**; the default constructor
- **Constructor** Create(AType: TConvertType); **overload**; the constructor to set the type

The public methods are:

- **function CharToFormat**(charstring: string): string; to convert a string to a string in the format defined by AType.
- **function FormatToChar**(str: string): string; to convert a formatted string to a raw string
- **function UnicodeToPAnsiChar**(str: string): PAnsiChar; to convert an Unicode string to a PAnsiChar with only UTF8 characters values
- **function PAnsiCharFromUnicodeLength**(str: string): Integer; to compute the length of the Unicode string in byte
- **function StringToBuffer**(str: string; u: TUnicode; var msgLen: Integer) : PAnsiChar; to convert a string to a PAnsiChar (or PByte for mobile platforms) and return the length of PAnsiChar in msgLen value.
- **function StringToBufferA**(str: string; u: TUnicode): PAnsiChar; to convert a string to a PAnsiChar (or PByte for mobile platforms).
- **function UnicodeStringToFormat**(str: string): string; to convert a Unicode string to a formatted string (hexa, base64, etc.)
- **function FormatToUnicodeString**(str: string): string; to convert a formatted string to an Unicode string
- **function StringToByteArray**(str: string): TArray<Byte>; to convert a string to a byte array
- **function TestUnicode**(str: string): Integer; to test whether a string has Unicode characters
- **function StringToUnicode**(str: string): string; to convert an ANSI string to an Unicode string
- **function StringToFormat**(charstring: string): string; to convert a raw string to a string in the format defined by AType
- **function FormatToString**(str: string): string; to convert a string in the format defined by AType to a raw string
- **function OutputFormatLength**(charLen: Integer): Integer; to compute the length of the formatted string from the length of the binary data
- **function CharLength**(charstring: string): Integer; to compute the length of the binary data from the formatted string
- **function Base64ToHexa**(base64String: string): string; to convert a string in base64 format to a string in hexadecimal format
- **function HexaToBase64**(hexaString: string): string; to convert a string in hexadecimal format to a string in base64 format
- **function Base64ToBase64url**(inString: string): string; to convert a string in base64 format to a string in base64url format
- **function Base64urlToBase64**(inString: string): string; to convert a string in base64url format to a string in base64 format
- **function Base64urlToHexa**(inString: string): string; to convert a string in base64url format to a string in hexadecimal format
- **function HexaToBase64url**(inString: string): string; to convert a string in hexadecimal format to a string in base64url format
- **function Base32ToHexa**(base32String: string): string; to convert a string in base32 format to a string in hexadecimal format
- **function HexaToBase32**(hexaString: string): string; to convert a string in hexadecimal format to a string in base32 format

- **function** `Base32ToBase64url`(inString: string): string; to convert a string in base32 format to a string in base64url format
- **function** `Base64urlToBase32`(inString: string): string; to convert a string in base64url format to a string in base32 format
- **function** `Base32ToBase64`(inString: string): string; to convert a string in base32 format to a string in base64 format
- **function** `Base64ToBase32`(inString: string): string; to convert a string in base64 format to a string in base32 format
- **function** `KeyRSAOpenSSLToKeyTRSAEncSign`(strKey: string): string; to convert an RSA key (the modulus, the public exponent or the private exponent) in OpenSSL format to an RSA key usable in TRSAEncSign (bytes are set in the reverse order)
- **function** `KeyTRSAEncSignToKeyRSAOpenSSL`(strKey: string): string; to convert an RSA key (the modulus, the public exponent or the private exponent) in TRSAEncSign format to an RSA key in OpenSSL format
- **function** `Base58Encode`(const value: uint64): string; to convert an uint64 to a string in Base58
- **function** `Base58Decode`(const encoded: string): uint64; to convert a string in Base58 to the corresponding uint64
- **function** `TBytesToString`(const t: TBytes): string; to convert a TBytes into a string where each byte is a character
- **function** `StringToTBytes`(const str: string): TBytes; to convert a string of bytes into a TBytes
- **function** `RandomString`(len: Integer): string; to generate a random byte string of length len
- **procedure** `ToCharString`(InHexString: string; var OutCharString: string); to convert a raw string to a hex string
- **procedure** `ToCharString`(InHexString: string; var OutCharArray: TBytes); to convert a raw string to a byte array
- **procedure** `ToHexString`(InCharString: string; var OutHexString: string); to convert a hex string to a raw string
- **procedure** `ToHexString`(InCharString: string; var OutHexArray: TBytes); to convert a byte array to a raw string
- **function** `BigHexToBigInt`(s: string): string; to convert 'long' strings representing hexadecimal values to 'long' strings representing integers

The property is:

- **property** `AType`: TConvertType **read** `FType` **write** `Ftype`; to read and write the type of the conversion: hexa, base64, base64url, base32 or raw

Example of how to use AES with a key in TBytes form

Let b be a TBytes array of 16 bytes.

```
var
  aes: TAESEncryption;
  conv: TConvert;
```

```
str: string;

begin
  aes:= TAESEncryption.Create;
  conv:= TConvert.Create;
  try
    aes.AType := atcbc;
    aes.KeyLength := kl128;
    aes.InputFormat := raw;
    aes.OutputFormat := hexa;
    aes.Key := conv.TBytesToString(b);
    aes.IVMode := TIVMode.rand;
    aes.PaddingMode := TPaddingMode.PKCS7;
    aes.Unicode := yesUni;

    str := AES.Encrypt('test');

  finally
    aes.Free;
    conv.Free;
  end;
end;
```

All CONVERSION functions/procedures are in the MiscObj file.

Again, check **MiscObj.pas** for the latest updates.

X509 certificates

X.509 is a standard that defines the format of public key certificates. X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS, the secure protocol for browsing the web. They are also used in offline applications, like electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, or an organization, or an individual), and is either signed by a certificate authority or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

In the library, you can decode an X509 certificate to display all the fields and verify the signature if the algorithm is supported. You can also generate a self-signed certificate (only on MacOS and Windows platforms) and sign a Certificate Signing Request (CSR).

The demo on Windows of these functionalities is now included in the Demo folder (or at <https://www.tmssoftware.com/site/freetools.asp> for the executable).

The signature algorithms supported by TMS CP are: RSA-SHA256, RSA-SHA384, RSA-SHA512, ECDSA-SHA256 (corresponding to P-256 curve), ECDSA-SHA384 (corresponding to P-384 curve), ECDSA-SHA512 (corresponding to P-521 curve). All RSA algorithms are RSA PKCS#1 v1.5, with a key length of 2048 or 4096 bits. Moreover, RSA-SHA1 is supported for decoding.

The **(incomplete)** X509 certificate class is⁵:

```
TX509Certificate = class(TTMSCryptBase)
public
  constructor Create(AOwner: TComponent); overload; override;
  constructor Create; reintroduce; overload;
  constructor Create(RootCAPath: string); reintroduce; overload;

{$IF defined(MSWINDOWS)}
  // PKCS#12/PFX/p12 functions - to be moved to PFX.pas in a future release
  procedure ProcessAuthSafe(Content: string);
  procedure ProcessGeneralSequence(Content: string);
  function ProcessMacData(Content: string): string;
  function DecodePFXSequence(Content: string): integer;
  function HMacGenKey(idByte: integer; Size: integer): string;
  function HmacAuthenticatedSafe(KHash, s: string): string;
  procedure ComputeHash(inBlock: TBytes; var outHash: TBytes);
{$IFEND}

{$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
  procedure GenerateSelfSigned;
  procedure GenerateSelfSigned(kfp: string); overload;
{$IF defined(MSWINDOWS)}

  procedure DecodeCertFromTextFile(TextFilePath: string);
  procedure DecodeCertFromPEM(PEMfilePath: string);

  procedure DecodeCertFromPFX(PFXfilePath: string; Password: string;
```

⁵ See class for latest updates.

```

    PathToOpenSSL: string);
procedure DecodeCertAndKeyFromPFX(PFXFilePath: string; Password: string;
    PathToOpenSSL: string; KeyPath: string);
procedure ExportToPFX(PFXFilePath: string; Password: string;
    PathToOpenSSL: string);
{$IFEND}
{$IFEND}
procedure Decode;
procedure Reset;

procedure SignCSR(CSRFilePath: string; outputPath: string); overload;
function SignCSR(CSR: string): string; overload;

procedure RSAExtractPrivateKey(KeyStr: string);
procedure ECCEExtractPrivateKey(KeyStr: string);

function GenerateKeyFile(keySizeBits: integer; modulus, publicExponent,
    privateExponent, prime1, prime2, dp, dq, qinv: string; KeyFilePathChar:
    string): integer; overload;
function GenerateKeyFile(keySizeBits: integer; PublicKey, PrivateKey,
    KeyFilePathChar: string): integer; overload;

function GenerateEncryptedKeyFile(keySizeBits: integer; modulus,
    publicExponent, privateExponent, prime1, prime2, dp, dq, qinv: string;
    password: string; KeyFilePathChar: string): integer; overload;
function GenerateEncryptedKeyFile(keySizeBits: integer; PublicKey,
    PrivateKey, password, KeyFilePathChar: string): integer; overload;

function GeneratePublicKeyFile(keySizeBits: integer; modulus,
    publicExponent, KeyFilePathChar: string): integer;

function GenerateSelfSignedCertificate(information: CERTINFSTRUCT;
    CertFilePathChar: string; modulus,
    privateExponent, publicExponent: string): integer; overload;
function GenerateSelfSignedCertificate(information: CERTINFSTRUCT;
    CertFilePathChar, PrivateKey, PublicKey: string): integer; overload;

published
property KeyFilePath: string read FKeyFilePath write setKeyFilePath;
property CrtFilePath: string read FCrtFilePath write setCrtFilePath;
property signatureAlgorithm: TSignAlgo read FSignatureAlgorithm
    write setSignatureAlgorithm default TSignAlgo.sa_sha256rsa;
property hashFunction: TX509HashFunction read FHashFunction
    write setHashFunction default TX509HashFunction.sha256;

property SubjectCountryName: string read FSubjectCountryName write
SetCountryName;
property IssuerCountryName: string read FIssuerCountryName;
property SubjectStateName: string read FSubjectStateName write
FSubjectStateName;
property IssuerStateName: string read FIssuerStateName;
property SubjectLocalityName: string read FSubjectLocalityName
    write FSubjectLocalityName;
property IssuerLocalityName: string read FIssuerLocalityName;
property SubjectOrganizationName: string read FSubjectOrganizationName
    write FSubjectOrganizationName;
property IssuerOrganizationName: string read FIssuerOrganizationName write
FIssuerOrganizationName;
property IssuerOrganizationIdentifier: string read
FIssuerOrganizationIdentifier

```

```

    write FIssuerOrganizationIdentifier;
    property IssuerOrganizationUnitName: string read FIssuerOrganizationUnitName
        write FIssuerOrganizationUnitName;
    property SubjectOrganizationIdentifier: string read
FSubjectOrganizationIdentifier
        write FSubjectOrganizationIdentifier;
    property SubjectOrganizationalUnitName: string read
FSubjectOrganizationUnitName
        write FSubjectOrganizationUnitName;
    property IssuerOrganizationalUnitName: string read
FIssuerOrganizationalUnitName
        write FIssuerOrganizationalUnitName;
    property SubjectCommonName: string read FSubjectCommonName write
FSubjectCommonName;
    property SubjectGivenName: string read FSubjectGivenName write
        FSubjectGivenName;
    property SubjectSurname: string read FSubjectSurname write FSubjectSurname;
    property IssuerCommonName: string read FIssuerCommonName;
    property IssuerSerialNumber: string read FIssuerSerialNumber;
    property SubjectSerialNumber: string read FSubjectSerialNumber;

    property PKIPolicyIssuer: string read FPKIPolicyIssuer write
        FPKIPolicyIssuer;
    property CertificatePolicies: string read FCertificatePolicies write
        FCertificatePolicies;
    property QcpNatural: boolean read FQcpNatural default false;

    property isCA: boolean read FISCA write FISCA default false;
    property IssuerAltName: TStringList read FIssuerAltName write
        FIssuerAltName;
    property SubjectAltName: TStringList read FSubjectAltName write
        FSubjectAltName;
    property IssuerEmailAddress: string read FIssuerEmailAddress;
    property SubjectEmailAddress: string read FSubjectEmailAddress;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property X509version: string read FX509Version;
    property serialNumber: string read FSerialNumber;
    property notBefore: string read FnotBefore;
    property notAfter: string read FnotAfter;
    property EncryptionAlgorithm: string read FEncryptionAlgorithm;
    property PublicKeyModulus: string read FPublicKeyModulus write
        FPublicKeyModulus;
    property Exponent: string read FExponent write FExponent;
    property PublicKey: string read GetPublicKey write SetPublicKey; // ECC
    property PrivateKey: string read GetPrivateKey write SetPrivateKey; // ECC
    property IsSignatureValid: string read FISignatureValid;
    property BitSizeEncryptionAlgorithm: Integer
        read FBitSizeEncryptionAlgorithm write setBitSizeEncryptionAlgorithm;
    property PolicyInformation: string read FPolicyInformation write
        SetPolicyInformation;
    property ecCurve: string read FECCurve;
    property RootCAPath: string read FRootCAPath write FRootCAPath;
    property CrtRaw: string read FCrtRaw write FCrtRaw;
    property CrtStr: string read FCrtStr write FCrtStr;
    property KeyStr: string read FKeyStr write FKeyStr;
    property PSSParam: TPSSParams read FPSSParam write FPSSParam;
    // RSA parameters
    property OutputFormat: TConvertType read FOutputFormat write FOutputFormat
        default hexa;

```

```

property InputFormat: TConvertType read FInputFormat rite FInputFormat
default raw;
property RSASignType: TRSASignType read GetRSASignType write SetRSASignType
default spkcs1_5;
    // PFX values
property Pfx: TPFXParams read FPfx write FPfx;
end;

```

The constructors are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor
- **constructor** Create(RootCAPath: string); reintroduce; overload; the constructor to set the path to the folder containing the CA certificates

The public methods are:

- **procedure GenerateSelfSigned**; to generate a self-signed certificate (only available on desktop platforms)
- **procedure DecodeCertFromPFX**(PFXFilePath: string; Password: string; PathToOpenSSL: string); to decode a certificate from a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platforms)
- **procedure DecodeCertAndKeyFromPFX**(PFXFilePath: string; Password: string; PathToOpenSSL: string; KeyPath: string); to decode a certificate and save the private key (in KeyPath file) from a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platforms)
- **procedure ExportToPFX**(PFXFilePath: string; Password: string; PathToOpenSSL: string); to export the certificate and the private key in a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platforms)
- **procedure Decode**; to decode the fields of a certificate and verify the signature
- **procedure SignCSR**(CSRFilePath: string; outputFilePath: string); to sign a CSR in PEM format in CSRFilePath and write the output in PEM format in outputFilePath.
- **function SignCSR**(CSR: string): string; sign a CSR string and return the output certificate

The properties are:

- **property** KeyFilePath: string **read** FKeyFilePath **write** setKeyFilePath; to set and get the path to the private key file
- **property** CrtFilePath: string **read** FCrtFilePath **write** setCrtFilePath; to read and write the path to the certificate file
- **property** signatureAlgorithm: TSignAlgo **read** FSignatureAlgorithm **write** setSignatureAlgorithm; to read and write the path to the certificate file
- **property** hashFunction: TX509HashFunction **read** FHashfunction **write** SetHashfunction **default** sha256; to read and write the hash function

- `property` `countryName`: string read `FSubjectCountryName` write `SetCountryName`; to read and write the subject country name field
- `property` `IssuerCountryName`: string read `FIssuerCountryName`; to read and write the issuer country name field
- `property` `stateName`: string read `FSubjectStateName` write `FSubjectStateName`; to read and write the subject state name field
- `property` `IssuerStateName`: string read `FIssuerStateName`; to read and write the issuer state name field
- `property` `localityName`: string read `FSubjectLocalityName` write `FSubjectLocalityName`; to read and write the subject locality name field
- `property` `IssuerLocalityName`: string read `FIssuerLocalityName`; to read and write the issuer locality name field
- `property` `OrganizationName`: string read `FSubjectOrganizationName` write `FSubjectOrganizationName`; to read and write the subject organization name field
- `property` `IssuerOrganizationName`: string read `FIssuerOrganizationName`; to read and write the issuer organization name field
- `property` `OrganizationUnitName`: string read `FSubjectOrganizationUnitName` write `FSubjectOrganizationUnitName`; to read and write the subject organization unit name field
- `property` `IssuerOrganizationUnitName`: string read `FIssuerOrganizationUnitName`; to read and write the issuer organization unit name field
- `property` `commonName`: string read `FSubjectCommonName` write `FSubjectCommonName`; to read and write the subject common name field (mandatory)
- `property` `IssuerCommonName`: string read `FIssuerCommonName`; to read and write the issuer common name field (mandatory)
- `property` `AltName1`: string read `FAltName1` write `FAltName1`; to read and write the first alternative name field
- `property` `AltName2`: string read `FAltName2` write `FAltName2`; to read and write the second alternative name field
- `property` `AltName3`: string read `FAltName3` write `FAltName3`; to read and write the third alternative name field
- `property` `AltName4`: string read `FAltName4` write `FAltName4`; to read and write the fourth alternative name field
- `property` `AltName5`: string read `FAltName5` write `FAltName5`; to read and write the fifth alternative name field
- `property` `AltName6`: string read `FAltName6` write `FAltName6`; to read and write the sixth alternative name field
- `property` `isCA`: boolean write `FISCA`; to set whether the certificate is a CA one
- `property` `Unicode`: `TUnicode` read `FUni` write `FUni` default `yesUni`; to indicate whether the input buffer or the file name has Unicode characters
- `property` `version`: string read `FVersion`; to read the version of the certificate
- `property` `serialNumber`: string read `FSerialNumber`; to read the serial number of the certificate
- `property` `notBefore`: string read `FnotBefore`; to read the date of emission of the certificate
- `property` `notAfter`: string read `FnotAfter`; to read the date of validity of the certificate

- `property` EncryptionAlgorithm: string read FEncryptionAlgorithm; to read the encryption algorithm of the certificate
- `property` publicKey: string read FPublicKey; to read the public key of the certificate
- `property` modulus: string read FModulus; to read the modulus of the certificate
- `property` IsSignatureValid: string read FISSignatureValid; to read the validity of the certificate
- `property` BitSizeEncryptionAlgorithm: Integer read FBitSizeEncryptionAlgorithm write FBitSizeEncryptionAlgorithm; to read and write the bit size of the encryption algorithm of the certificate
- `property` ecCurve: string read FECCurve; to read the curve type of the certificate
- `property` RootCAPath: string read FRootCAPath write FRootCAPath; to read and write the path to the folder containing the CA certificates
- `property` CrtStr: string read FCrtStr write FCrtStr; to read and write the content in base64 of the certificate (between the ---BEGIN CERTIFICATE--- and ---END CERTIFICATE--- lines)
- `property` KeyStr: string read FKeyStr write FKeyStr; to read and write the content in base64 of the private key (between the ---BEGIN EC PRIVATE KEY--- (or ---BEGIN RSA PRIVATE KEY---) and ---END EC PRIVATE KEY--- (or ---END RSA PRIVATE KEY---) lines)
- `property` PSSParam: string read FPSSParam; to read the RSA PSS parameters of a certificate, i.e., the MGF function and the salt length.

Example of how to generate a self-signed X509 certificate

```
var
  X509Certificate1: TX509Certificate;
begin
  X509Certificate1 := TX509Certificate.Create;
  try
    X509Certificate1.RootCAPath := '.\RootCA\';
    X509Certificate1.KeyFilePath := '.\mykey.key';
    X509Certificate1.CrtFilePath := '.\mycert.crt';
    X509Certificate1.signatureAlgorithm := TSignAlgo.sa_sha256rsa;
    X509Certificate1.BitSizeEncryptionAlgorithm := 2048;
    X509Certificate1.countryName := 'BE';
    X509Certificate1.stateName := 'Flanders';
    X509Certificate1.localityName := 'Wevelgem';
    X509Certificate1.OrganizationName := 'TMS Software';
    X509Certificate1.commonName := 'TMS Software certificate';
    X509Certificate1.GenerateSelfSigned;
  finally
    X509Certificate1.Free;
  end;
end;
```

Example of how to parse an X509 certificate

```
var
  X509Certificate1: TX509Certificate;
  ts: TStringList;
  I: integer;
begin
```

```

X509Certificate1 := TX509Certificate.Create;
ts := TStringList.Create;
try
  X509Certificate1.RootCAPath := '\\RootCA\\';
  X509Certificate1.CrtFilePath := '\\mycert.crt';
  X509Certificate1.Decode;
  ts.Add('Version: ' + X509Certificate1.version);
  ts.Add('Serial number: ' + X509Certificate1.serialNumber);
  ts.Add('Signature algorithm: ' + TabSignAlgo
    [Integer(X509Certificate1.signatureAlgorithm)]);
  ts.Add('not before: ' + X509Certificate1.notBefore);
  ts.Add('not after: ' + X509Certificate1.notAfter);
  ts.Add('Subject country name: ' + X509Certificate1.countryName);
  ts.Add('Subject state or province name: ' + X509Certificate1.stateName);
  ts.Add('Subject locality name: ' + X509Certificate1.localityName);
  ts.Add('Subject organization name: ' + X509Certificate1.OrganizationName);
  ts.Add('Subject organization unit name: ' +
    X509Certificate1.OrganizationUnitName);
  ts.Add('Subject common name: ' + X509Certificate1.commonName);
  ts.Add('Issuer country name: ' + X509Certificate1.IssuercountryName);
  ts.Add('Issuer state or province name: ' +
    X509Certificate1.IssuerstateName);
  ts.Add('Issuer locality name: ' + X509Certificate1.IssuerlocalityName);
  ts.Add('Issuer organization name: ' +
    X509Certificate1.IssuerOrganizationName);
  ts.Add('Issuer organization unit name: ' +
    X509Certificate1.IssuerOrganizationUnitName);
  ts.Add('Issuer common name: ' + X509Certificate1.IssuerCommonName);

  for I := 0 to X509Certificate1.AltName.Count - 1 do
    ts.Add('Alternative name ' + IntToStr(I + 1) + ': ' +
      X509Certificate1.AltName.Strings[I]);

  ts.Add('Asymmetric encryption algorithm: ' +
    X509Certificate1.EncryptionAlgorithm + ' ' +
    IntToStr(X509Certificate1.BitSizeEncryptionAlgorithm) + ' bits');
  ts.Add('Modulus: ' + X509Certificate1.Modulus);
  ts.Add('Curve: ' + X509Certificate1.ecCurve);
  ts.Add('Public key: ' + X509Certificate1.publicKey);
  ts.Add(X509Certificate1.IsSignatureValid);
finally
  X509Certificate1.Free;
end;
end;

```

All X509 Certificate generation and parsing functions are in the X509Obj.pas file.

X509 CSR

CSR means for Certificate Signing Request. It is a message sent from an applicant to a certificate authority to apply for a digital identity X.509 certificate. It usually contains the public key for which the certificate should be issued, identifying information (such as a domain name) and integrity protection (e.g., a digital signature). The most common format for CSRs is the PKCS #10 specification.

In this library, you can generate a CSR in PKCS#10 format (only for Desktop platform) and decode them.

RSA uses PKCS#1 v1.5 for signatures.

The X509 CSR class inherits from X509Certificate and is:

```
TX509CSR = class(TX509Certificate)
private
  FCsrFilePath: string;
  FIsSignatureValid: string;
  FCsrStr: string;
  FParameters: TCSRParams;

  procedure setCsrFilePath(path: string);
  function GenerateKeyFile(keySizeBits: integer; modulus, publicExponent,
    privateExponent, prime1, prime2, dp, dq, qinv: string; KeyFilePathChar:
string): integer; overload;
  function GenerateKeyFile(keySizeBits: integer; PublicKey, PrivateKey,
    KeyFilePathChar: string): integer; overload;
  function GenerateCertificateRequest(modulus, publicExponent,
privateExponent: string; information: CERTINFSTRUCT; CSRFilePath: string):
integer; overload;
  function GenerateCertificateRequest(PublicKey, PrivateKey: string;
information: CERTINFSTRUCT; CSRFilePath: string): integer; overload;

public
  constructor Create(AOwner: TComponent); overload; override;
  constructor Create; reintroduce; overload;
  {$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
  procedure Generate;
  {$IFEND}
  procedure Decode;

published
  property CsrFilePath: string read FCsrFilePath write setCsrFilePath;
  property IsSignatureValid: string read FIsSignatureValid;
  property CsrStr: string read FCsrStr write FCsrStr;
  property Parameters: TCSRParams read FParameters write FParameters;
end;
```

The constructors are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor

The public methods are:

- **procedure** Generate; to generate a CSR (only available on desktop platforms)

- **procedure** `Decode`; to decode the fields of a CSR and verify the signature

The properties are:

- **property** `KeyFilePath`: string **read** `FKeyFilePath` **write** `setKeyFilePath`; to set and get the path to the private key file
- **property** `CrtFilePath`: string **read** `FCrtFilePath` **write** `setCrtFilePath`; to read and write the path to the certificate file
- **property** `signatureAlgorithm`: `TSignAlgo` **read** `FSignatureAlgorithm`, **write** `setSignatureAlgorithm`; to read and write the path to the certificate file
- **property** `hashFunction`: `TX509HashFunction` **read** `FHashfunction` **write** `SetHashfunction` **default** `sha256`; to read and write the hash function
- **property** `countryName`: string **read** `FSubjectCountryName` **write** `SetCountryName`; to read and write the subject country name field
- **property** `stateName`: string **read** `FSubjectStateName` **write** `FSubjectStateName`; to read and write the subject state name field
- **property** `localityName`: string **read** `FSubjectLocalityName` **write** `FSubjectLocalityName`; to read and write the subject locality name field
- **property** `OrganizationName`: string **read** `FSubjectOrganizationName` **write** `FSubjectOrganizationName`; to read and write the subject organization name field
- **property** `OrganizationUnitName`: string **read** `FSubjectOrganizationUnitName` **write** `FSubjectOrganizationUnitName`; to read and write the subject organization unit name field
- **property** `commonName`: string **read** `FSubjectCommonName` **write** `FSubjectCommonName`; to read and write the subject common name field (mandatory)
- **property** `Unicode`: `TUnicode` **read** `FUni` **write** `FUni` **default** `yesUni`; to indicate whether the input buffer or the file name has Unicode characters
- **property** `version`: string **read** `FVersion`; to read the version of the certificate
- **property** `EncryptionAlgorithm`: string **read** `FEncryptionAlgorithm`; to read the encryption algorithm of the certificate
- **property** `publicKey`: string **read** `FPublicKey`; to read the public key of the certificate
- **property** `modulus`: string **read** `FModulus`; to read the modulus of the certificate
- **property** `IsSignatureValid`: string **read** `FIsSignatureValid`; to read the validity of the certificate
- **property** `BitSizeEncryptionAlgorithm`: Integer **read** `FBitSizeEncryptionAlgorithm` **write** `FBitSizeEncryptionAlgorithm`; to read and write the bit size of the encryption algorithm of the certificate
- **property** `ecCurve`: string **read** `FECCurve`; to read the curve type of the certificate
- **property** `CsrStr`: string **read** `FCsrStr` **write** `FCsrStr`; to read and write the content in base64 of the CSR (between the ---BEGIN CERTIFICATE REQUEST--- and ---END CERTIFICATE REQUEST--- lines)
- **property** `KeyStr`: **string read** `FKeyStr` **write** `FKeyStr`; to read and write the content in base64 of the private key (between the ---BEGIN EC PRIVATE KEY--- (or ---BEGIN RSA PRIVATE KEY---) and ---END EC PRIVATE KEY--- (or ---END RSA PRIVATE KEY---) lines)

Example of how to generate a X509 CSR

```
var
```

```
X509CSR1: TX509CSR;
begin
  X509CSR1 := TX509CSR.Create;
  try
    X509CSR1.KeyFilePath := '.\mykey.key';
    X509CSR1.CsrFilePath := '.\mycsr.csr';
    X509CSR1.signatureAlgorithm := TSignAlgo.sa_sha256rsa;
    X509CSR1.BitSizeEncryptionAlgorithm := 2048;
    X509CSR1.countryName := 'BE';
    X509CSR1.stateName := 'Flanders';
    X509CSR1.localityName := Wevelgem;
    X509CSR1.OrganizationName := TMS Software;
    X509CSR1.commonName := TMS certificate';
    X509CSR1.Generate;
  Finally
    X509CSR1.Free;
  end;
end;
```

Example of how to parse an X509 CSR

```
var
  X509CSR1: TX509CSR;
  ts: TStringList;
begin
  X509CSR1 := TX509CSR.Create;
  ts := TStringList.Create;
  try
    X509CSR1.CsrFilePath := '.\mycsr.csr';
    X509CSR1.Decode;
    ts.Add('Version: ' + X509CSR1.version);
    ts.Add('Signature algorithm: ' + TabSignAlgo
      [Integer(X509CSR1.signatureAlgorithm)]);
    ts.Add('Subject country name: ' + X509CSR1.countryName);
    ts.Add('Subject state or province name: ' + X509CSR1.stateName);
    ts.Add('Subject locality name: ' + X509CSR1.localityName);
    ts.Add('Subject organization name: ' + X509CSR1.OrganizationName);
    ts.Add('Subject organization unit name: ' +
      X509CSR1.OrganizationUnitName);
    ts.Add('Subject common name: ' + X509CSR1.commonName);
    ts.Add('Asymmetric encryption algorithm: ' +
      X509CSR1.EncryptionAlgorithm + ' ' +
      IntToStr(X509CSR1.BitSizeEncryptionAlgorithm) + ' bits');
    ts.Add('Modulus: ' + X509CSR1.Modulus);
    ts.Add('Curve: ' + X509CSR1.ecCurve);
    ts.Add('Public key: ' + X509CSR1.publicKey);
    ts.Add(X509CSR1.IsSignatureValid);
  Finally
    X509CSR1.Free;
  end;
end;
```

All X509 CSR generation and parsing functions are in the X509Obj.pas file.

PKCS11

PKCS#11 is the programming interface to create and manipulate cryptographic tokens.

The PKCS #11 standard defines a platform-independent API to cryptographic tokens, such as hardware security modules (HSM) and smart cards, and names the API itself "Cryptoki" (from "cryptographic token interface" and pronounced as "crypto-key" - but "PKCS #11" is often used to refer to the API as well as the standard that defines it).

The API defines most commonly used cryptographic object types (RSA keys, X.509 Certificates, AES keys, etc.) and all the functions needed to use, create/generate, modify and delete those objects.

Each token has a different DLL filename to access to PKCS11 library functions. You need to know the filename of your driver to use our component. There is a list of known driver filenames here:

<http://wiki.ncryptoki.com/Known-PKCS-11-modules.ashx>

Obviously, you need to know the PIN code of your token if you would like to use secret or private keys.

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of "slots". Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is "present in the slot" (typically) when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots and applications can connect to all these tokens.

In our library, we suppose that one slot = one token. To use the component, you need to select which slot index you want to use into the list of available slots. The "first" token into this slot is automatically selected. Tell us if you have more than one token into a slot.

In TMS Cryptography Pack, there are 3 files for PKCS11:

- PKCS11Values.pas that contains all Cryptoki constants and types
- PKCS11Library.pas that contains all Cryptoki basic functions, the documentation of these functions is available here: <http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html>
- PKCS11Obj.pas that contains a component class to use easily the Cryptoki functions. This class is TPKCS11, described below. You can ONLY use it on **Windows** platforms.

```
type
  TPKCS11param = packed record
    isToken: boolean;
    SlotIndex: Integer;
    CertIndex: Integer;
    DLLPath: string;
    Pin: string;
  end;

TPKCS11 = class(TTMSCryptBase)
public
  constructor Create(AOwner: TComponent); overload; override;
  constructor Create; reintroduce; overload;
```

```

constructor Create(DLLPath: string); reintroduce; overload;

function ListSlots: TStringList;
function ListTokens: TStringList;
function ListObjects: TStringList;
function ListCertificates: TStringList;
function ListPrivateKeys: TStringList;
function ListPublicKeys: TStringList;
function ListSecretKeys: TStringList;
function ListMechanisms: TStringList;

procedure OpenSession;
procedure CloseSession;
procedure Login(PIN: string);
procedure Logout;
procedure SetPIN(oldPin: string; newPIN: string);
procedure InitPIN(PIN: string);

function CertificatesIndex: TIndexArray;
function SecretKeysIndex: TIndexArray;
function PrivateKeysIndex: TIndexArray;
function PublicKeysIndex: TIndexArray;

function PrivateKeyIndexFromID(id: string): Integer;
function PublicKeyIndexFromID(id: string): Integer;
function CertificateIndexFromID(id: string): Integer;
function GetObjectID(index: Integer): string;
function ExtractCertificate(index: Integer): string;

function SignWithPrivateKey(s: string): string;
function SignWithSecretKey(s: string; algorithm: TMACAlgorithm): string;
function VerifyWithPublicKey(s, signature: string): Integer;
function VerifyWithSecretKey(s, signature: string;
    algorithm: TMACAlgorithm): Integer;

function EncryptWithPublicKey(s: string; algo: TAsymEncAlgo): string;
function AESEncryptWithSecretKey(s: string; mode: TAESMode;
    IV: string): string;
function DecryptWithPrivateKey(s: string; algo: TAsymEncAlgo): string;
function AESDecryptWithSecretKey(s: string; mode: TAESMode;
    IV: string): string;

function ShowKey(index: Integer): TStringList;
function ShowCert(index: Integer): TStringList;

procedure GenerateAESKey(keyLength: Integer);
procedure GenerateGenericSecretKey(keyLength: Integer);

function IsCertificate(index: Integer): boolean;
function IsPublicKey(index: Integer): boolean;
function IsPrivateKey(index: Integer): boolean;
function IsSecretKey(index: Integer): boolean;

published
property currentSlotIndex: Integer read FcurrentSlotIndex
    write setCurrentSlot default -1;
property currentObjectIndex: Integer read FcurrentObjectIndex
    write setCurrentObject default -1;
property outputFormat: TConvertType read FoutputFormat write FoutputFormat
    default base64;

```

```
property DLLpath: string read FDLLPath write setDLLPath;  
end;
```

The constructors are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor
- **constructor** Create(DLLPath: string); reintroduce; overload; the constructor to set the DLL file path, required to access to the token library

The public methods are:

- **function** ListSlots: TStringList; to list the slots, i.e., the cryptographic devices that are active
- **function** ListTokens: TStringList; to list the tokens that are present on the current slot. We suppose that one slot = one token.
- **function** ListObjects: TStringList; to list all objects on the current slot
- **function** ListCertificates: TStringList; to get the details of the certificates present on the current slot
- **function** ListPrivateKeys: TStringList; to get the details of the private keys present on the current slot
- **function** ListPublicKeys: TStringList; to get the details of the public keys present on the current slot
- **function** ListSecretKeys: TStringList; to get the details of the secret keys present on the current slot
- **function** ListMechanisms: TStringList; to get the list of the mechanisms of the token, i.e. the available algorithms.
- **procedure** OpenSession; to open a session
- **procedure** CloseSession; to close a session
- **procedure** Login(PIN: string); to log in
- **procedure** Logout; to log out
- **procedure** SetPIN(oldPin: string; newPIN: string); to change the PIN
- **procedure** InitPIN(PIN: string); to set the PIN if the token does not have one
- **function** CertificatesIndex: TIndexArray; to get the index of certificates in list objects
- **function** SecretKeysIndex: TIndexArray; to get the index of secret keys in list objects
- **function** PrivateKeysIndex: TIndexArray; to get the index of private keys in list objects
- **function** PublicKeysIndex: TIndexArray; to get the index of public keys in list objects
- **function** PrivateKeyIndexFromID(id: string): Integer; to get the index of a private key from its ID
- **function** PublicKeyIndexFromID(id: string): Integer; to get the index of a public key from its ID
- **function** CertificateIndexFromID(id: string): Integer; to get the index of a certificate from its ID
- **function** GetObjectID(index: Integer): string; to get the ID of an object
- **function** ExtractCertificate(index: Integer): string; to extract the certificate in base64 format

- **function** `SignWithPrivateKey(s: string): string`; to sign the string `s` with the current private key (see property `currentObjectIndex` to set the private key)
- **function** `SignWithSecretKey(s: string; algorithm: TMACAlgorithm): string`; to sign the string `s` with the current secret key (see property `currentObjectIndex` to set the private key) and using a MAC algorithm
- **function** `VerifyWithPublicKey(s, signature: string): Integer`; to verify the signature of the string `s` with the current public key (see property `currentObjectIndex` to set the public key)
- **function** `VerifyWithSecretKey(s, signature: string; algorithm: TMACAlgorithm): Integer`; to verify the signature of the string `s` with the current secret key (see property `currentObjectIndex` to set the secret key) and using a MAC algorithm
- **function** `EncryptWithPublicKey(s: string; algo: TAsymEncAlgo): string`; to encrypt a string `s` with the current public key (see property `currentObjectIndex` to set the public key) and using an asymmetric encryption algorithm
- **function** `AESEncryptWithSecretKey(s: string; mode: TAESMode; IV: string): string`; to encrypt a string `s` with the current secret key (see property `currentObjectIndex` to set the secret key) and using AES with a mode and an IV (if not ECB mode).
- **function** `DecryptWithPrivateKey(s: string; algo: TAsymEncAlgo): string`; to decrypt a string `s` with the current private key (see property `currentObjectIndex` to set the private key) and using an asymmetric encryption algorithm
- **function** `AESDecryptWithSecretKey(s: string; mode: TAESMode; IV: string): string`; to decrypt a string `s` with the current secret key (see property `currentObjectIndex` to set the secret key) and using AES with a mode and an IV (if not ECB mode).
- **function** `ShowKey(index: Integer): TStringList`; to show the details of a key (public, secret or public)
- **function** `ShowCert(index: Integer): TStringList`; to show the details of a certificate
- **procedure** `GenerateAESKey(keyLength: Integer)`; to generate a non-persistent AES key on the token.
- **procedure** `GenerateGenericSecretKey(keyLength: Integer)`; to generate a non-persistent Generic secret key on the token.
- **function** `IsCertificate(index: Integer): boolean`; to know whether the object number `index` is a certificate
- **function** `IsPublicKey(index: Integer): boolean`; to know whether the object number `index` is a public key
- **function** `IsPrivateKey(index: Integer): boolean`; to know whether the object number `index` is a private key
- **function** `IsSecretKey(index: Integer): boolean`; to know whether the object number `index` is a secret key

The properties are:

- **property** `currentSlotIndex: Integer` **read** `FcurrentSlotIndex` **write** `setCurrentSlot` **default** `-1`; to get and set the current slot index
- **property** `currentObjectIndex: Integer` **read** `FcurrentObjectIndex` **write** `setCurrentObject` **default** `-1`; to get and set the current object (private key, public key, secret key or certificate)

- **property** outputFormat: TConvertType **read** FoutputFormat **write** FoutputFormat **default** base64; to get and set the outputFormat (also the format of the signature and the decrypt input)
- **property** DLLpath: string **read** FDLLPath **write** setDLLPath; to get and set the path fo the DLL filename

Example

```

var
  p11: TPKCS11;
  ts, ts2: TStringList;
  I, j: integer;
begin
  p11 := TPKCS11.Create('aetpkss1.dll');
  try
    p11.currentSlotIndex := 0;
    p11.Login('123456');
    ts := TStringList.Create;
    try
      ts := p11.ListObjects;
      for i := 0 to ts.Count - 1 do
        Memo1.Lines.Add(ts.Strings[i]);
      for i := 0 to ts.Count - 1 do
        begin
          if p11.IsPrivateKey(i) or p11.IsSecretKey(i) or p11.IsPublicKey(i) then
            begin
              ts2 := p11.ShowKey(i);
              try
                for j := 0 to ts2.Count - 1 do
                  Memo1.Lines.Add(ts2.Strings[j]);
                Finally
                  ts2.Free;
              end;
            end;
          if p11.IsCertificate(i) then
            begin
              ts2 := p11.ShowCert(i);
              try
                for j := 0 to ts2.Count - 1 do
                  Memo1.Lines.Add(ts.Strings[j]);
                Finally
                  ts2.Free;
              end;
            end;
          end;
        Finally
          ts.Free;
        end;
      Finally
        p11.Free;
      end;
    end;
end;

```

The PKCS11Param record is used into the XAdES, CAdES and PAdES class to set the required property to sign documents with cryptographic token.

PEM Certificates

Private Enhanced Mail (PEM) is a legacy standard for certificates used by many applications and services.

TMS CP generates certificates, CSR and keys in PKIX container formats. Keys can be encrypted or in the clear or encrypted at User's discretion.

Supported containers are:

```
'-----BEGIN CERTIFICATE-----'
```

```
'-----END CERTIFICATE -----'
```

```
'-----BEGIN PRIVATE KEY-----'
```

```
'-----END PRIVATE KEY-----'
```

```
'-----BEGIN ENCRYPTED PRIVATE KEY-----'
```

```
'-----END ENCRYPTED PRIVATE KEY-----'
```

```
'-----BEGIN RSA PRIVATE KEY-----'
```

```
'-----END RSA PRIVATE KEY-----'
```

```
'-----BEGIN EC PRIVATE KEY-----'
```

```
'-----END EC PRIVATE KEY-----'
```

```
'-----BEGIN CSR-----'
```

```
'-----END CSR -----'
```

Relevant methods are in the X509Certificate class.

PKCS#12, PFX, P12 Certificates

PKCS#12 standard for certificates used by Windows.

The PKCS#12 Class in TMS Cryptography Pack version 5.0.0.0 is a placeholder for future functions.

However, it is possible to use PFX certificates from the X509Certificate class.

This version only allows the parsing of PFX certificates in clear or encrypted, but for the 'certBag' and 'pkcs8ShroudedKeyBag' types.

The generation of PFX certificates will be added in a later release.

All PKCS#12 parsing and verification functions are in the X509Obj.pas file.

Abstract Syntax Notation 1 support files

X509v3, PEM, PKCS#12 certificates and Cryptographic Message Syntax (CMS) require the use of the Abstract Syntax Notation version 1 (ASN1) to encode and decode their content.

As of TMS Cryptography Pack v5.0.0.0, ASN1 functions have been separated and moved to ASN1Core.pas, OIDValues.pas and CMSCore.pas files, with many additions.

Some ASN1 function are still hard coded in the X509Certificate class but will be rearranged in future releases.

```
TASN1 = class(TTMSCryptBase)
private
    FPassword: string;
    FRsaPk: TRSAPrivateKey;
    FECPk: TECPrivateKey;
    FData: TX509Data;
    FSignatureAlgorithm: TSignAlgo;
    FHashFunction: TX509HashFunction;

public
    function UpdatePosition(ByteString: string; var Position: integer): UInt32;
    procedure OIDDecode(RawString: string; var OID: string); overload;
    function OIDDecode(RawString: string): string; overload;
    procedure OIDEncode(OID: string; var RawString: string); overload;
    function OIDEncode(OID: string): string; overload;
    function ExtractEncryptedPrivateKey(KeyStr: string): string;
    function ExtractRSAPrivateKey(KeyStr: string): string;
    function ExtractECCPrivateKey(KeyStr: string): string;
    procedure ParsePkiMessage(PkiMessage: string);
    procedure ProcessBag(BagBlock: string);
    function AssertAsnTag(AsnBlock: string; AsnTag: char;
        Position: integer): integer;
    function GetAsnTag(AsnBlock: string; Position: integer): char;
    function GetOid(AsnBlock: string; var AsnBlockLength,
        Position: integer): string;
    function AssertAsnNull(AsnBlock: string; Position: integer): integer;
    procedure ExtractAlgorithms(AlgSequence: string);
    function Pkcs7TimeStamp(InputHash: string): string;
    function LengthToTag(len: integer): string;
    function DoCertFields(Certificate: TStringList): string;
    function DecodeASNSequence(str: string): string;

    constructor Create; reintroduce; overload;
    destructor Destroy; reintroduce; overload;

    property Password: string read FPassword write FPassword;
    property RSAKeySet: TRSAPrivateKey read FRsaPk;
    property ECKeySet: TECPrivateKey read FECPk;
    property XDataSet: TX509Data read FData;
    property SignatureAlgorithm: TSignAlgo read FSignatureAlgorithm;
end;
```

Cryptographic Message Syntax support file

The Cryptographic Message Syntax (CMS) requires the use of the Abstract Syntax Notation version 1 (ASN1) to encode and decode its content.

As of TMS Cryptography Pack v5.0.0.0, A CMS class has been added with the creation of CMSCore.pas file.

```

TCms = class(TTMSCryptBase)
private
    FCountryName: string;
    FIssuerOrganisation: string;
    FOrganizationalUnitName: string;
    FIssuerOrganisationIdentifier: string;
    FIssuerCommonName: string;
    FSubjectCommonName: string;
    FSerialNumber: string;
    FContentType: string;
    FMessageDigest: string;
    FPDFRevocationInfoArchival: string;
    FSigningCertificateV2: string;
    FSignatureValue: string;
    FBitSizeEncryptionAlgorithm: integer;
    FSignedAttributes: string;
    FPublicKey: string;
    FExponent: string;
    FSignatureAlgorithm: TSignAlgo;
    FHashAlgorithm: string;
    FCertificate: string;
    FCertificateDigest: string;
    FSigningTime: string;
    FNotBefore: string;
    FNotAfter: string;
    FASN1: TASN1;

    function ReadDataSet: TX509Data;
    function ParseSigningCertificate(Cert: string): integer;

public
    procedure DecodeFromFile(FilePath: string);
    procedure Decode(CMSstring: string);
    function PreProcess(CertificateBlob: string): boolean;

    constructor Create; reintroduce; overload;
    destructor Destroy; reintroduce; overload;

published
    property PublicKey: string read FPublicKey;
    property Exponent: string read FExponent;

```

```
property SignatureAlgorithm: TSignAlgo read FSignatureAlgorithm;
property HashAlgorithm: string read FHashAlgorithm;
property SignedAttributes: string read FSignedAttributes write
FSignedAttributes;
property SignatureValue: string read FSignatureValue;
property BitSizeEncryptionAlgorithm: integer read FBitSizeEncryptionAlgorithm
write FBitSizeEncryptionAlgorithm;
property MessageDigest: string read FMessageDigest;
property Certificate: string read FCertificate write FCertificate;
property CertificateDigest: string read FCertificateDigest write
FCertificateDigest;
property SubjectCommonName: string read FSubjectCommonName;
property NotBefore: string read FNotBefore;
property NotAfter: string read FNotAfter;
property XDataSet: TX509Data read ReadDataSet;
end;
```

XAdES

XAdES (short for "**XML Advanced Electronic Signatures**") is a set of extensions to XML-DSig recommendation making it suitable for advanced electronic signatures. W3C and ETSI maintain and update XAdES together.

While XML-DSig is a general framework for digitally signing documents, XAdES specifies precise profiles of XML-DSig making it compliant with the European eIDAS regulation (*Regulation on electronic identification and trust services for electronic transactions in the internal market*). EIDAS is legally binding in all EU member states since July 2014. An electronic signature that has been created in compliance with eIDAS has the same legal value as a handwritten signature.

There are 6 profiles for XAdES: XAdES-BES, XAdES-T, XAdES-C, XAdES-X, XAdES-X-L, XAdES-A. TMS Cryptography Pack supports **only** the XAdES-BES profile.

There are 3 signature formats:

- *detached*: the signature is detached from the document to sign.
- *enveloping*: the signature includes the XML document (encoded in base 64).
- *enveloped*: the signature is added to the XML document to sign (with specific tags).

As of 5.0.9.0, a new XML template has been added in XAdESTemplate.pas. This template is specific to the Spanish Government regulations for invoices. To use it, you must set the new TXAdES class Template property to xSPEnveloped (see last example of this section).

As of 5.0.9.3, xPIEnveloped has been added for the Polish KSeF requirements. A specific template is bound to this for EC keys and specific processing is performed when required to ensure national requirements are satisfied.

As of 5.1.1.7, the following templates are available:

```
TTemplate = (  
    xBasic,           // standard template for XAdES-B, RSA, SHA  
    xSpEnveloped,    // template for Spanish Facturae, XAdES-B, RSA-SHA1, SHA1  
    xPIEnveloped,    // template for Polish KSeF, EC P-256, SHA-256  
    xSpEnvelopedV2,  // template for Spanish Facturae, XAdES-BBB, RSA-SHA1, SHA1  
    xSpEnvelopedTS  // template for Spanish Facturae, XAdES-BBB + Timestamp, RSA-  
    SHA1, SHA1  
);
```

These profiles correspond to ETSI standards:

- ETSI TS 101 903 v1.4.2 compliance for xBasic
- ETSI TS 101 903 v1.4.2 compliance with SHA1 for Spanish regulations
- ETSI TS 101 903 v1.4.2 compliance and adaptations for Polish regulations
- ETSI EN 319 132-1 v1.0.0 Building Blocks and Baseline (BBB) compliance with SHA1 for Spanish regulations
- ETSI EN 319 132-1 v1.0.0 Building Blocks and Baseline, plus time stamping, compliance with SHA1 for Spanish regulations

Note that ETSI EN 319 132-1 v1.0.0 Building Blocks and Baseline, plus time stamping, requires the use of an online timestamping server. Using this profile implies that the End User has an active Internet connection.

The XAdES class is:

```

TTemplate = (xBasic, xSpEnveloped, xPlEnveloped, xSpEnvelopedV2,
xSpEnvelopedTS);

TRole = (xSupplier, xCustomer, xThirdParty);

TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);
TPackaging = (detached, enveloping, enveloped);
TTemplate = (xBasic, xSpEnveloped, xPlEnveloped);
TRole = (xSupplier, xCustomer, xThirdParty);

TXAdES = class(TAdES)
Private
    FID: string;
    FCanonicalizedSignedInfo: string;
    FXMLDocument: IXMLDocument;
    FTemplate: TTemplate;
    FRole: TRole;

    function GenerateID: string;
    function NextTag(xmlldoc: string; var position: integer): string;
    function DeleteSignatureNode(var withspaces: string; xpathvalue: string):
        string;
    function GetNode(XmlNode: IXMLNode; NodeName: string): IXMLNode;
    function BuildXmlString(ARootNode: IXMLNode): string;
    function BuildAttributes(ANode: IXMLNode): string;
    procedure OrganizeNamespaces(var ARootNode: IXMLNode; tsxmlns: TStringList);
    procedure AddPrefix(var ARootNode: IXMLNode; prefix: string; xmlns: string);
    function ExtractNodeName(NodeName: string): string;
    function ExtractPrefix(Attribute: string): string;
    function ExtractNodeValue(NodeName: string): string;
    function ComputeSignatureMethod: string;
    function HashString(toHash: string; HashFunc: string; uni: boolean): string;
    function HashFile(toHashFile: string; HashFunc: string): string;
    procedure DeleteNameSpaceChildren(var LNode: IXMLNode; prefix,
        xmlns: string);
    function HTMLEncode(input: string): string;
    function ExtractNode(QName: string; FileStrings: TStringList): TStringList;
    function CleanCert(InCert: string): string;
    function IssuerSerialV2(Cert: TX509Certificate): string;
    procedure AddXmlNs(ts: TStringList; ns: string);

Public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; overload;
    constructor Create(Cert: TX509Certificate); reintroduce; overload;
    constructor Create(KeyFilePath: string; CertPath: string); reintroduce;
        overload;
    destructor Destroy; override;
    function ExtractCertificate: string;
    procedure GenerateSignature(FilePath: string; OutputFilePath: string);
    procedure ChangeSignatureTagLocation(SignatureFilePath: string;
        parentTag: string; AdditionalParentTags: TStringList;
        OutputFilePath: string);
    function VerifySignature(FilePath: string): integer;
    function VerifyError(err: Integer): string;

```

```
published
property Template: TTemplate read FTemplate write FTemplate default xBasic;
property Role: TRole read FRole write FRole default xSupplier;
end;
```

The constructors are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor
- **constructor** Create(Cert: TX509Certificate); reintroduce; overload; the constructor to set the X509 certificate
- **constructor** Create(KeyFilePath: string; CertPath: string); reintroduce; overload; the constructor to set the path to the private key file and the path to the certificate file

The public methods are:

- **procedure** GenerateSignature(FilePath: string; OutputFilePath: string); to generate an XAdES signature of FilePath in the file OutputFilePath
- **procedure** ChangeSignatureTagLocation(SignatureFilePath: string; parentTag: string; AdditionalParentTags: TStringList; OutputFilePath: string); to change the location of the signature, i.e. to put the signature into parentTag with AdditionalParentTags as parents.
- **function** VerifySignature(FilePath: string): integer; to verify the signature
- **function** VerifyError(err: Integer): string; to display the error associated to the error code err

The properties inherited from TAdES, are:

- **property** SignatureMethod: TSignatureMethod read FSignatureMethod; to read the signature method
- **property** KeyFilePath: string read FKeyFilePath write FKeyFilePath; to read and write the path to the private key file
- **property** Packaging: TPackaging read FPackaging write FPackaging default enveloping; to read and write the packaging detached, enveloping or enveloped
- **property** CertFilePath: string read FCertFilePath write SetCertPath; to read and write the path to the certificate file
- **property** ErrorDetails: string read FErrorDetails; to read the error details of verifying signature
- **property** PathToOriginalFile: string read FPathToOriginalFile write FPathToOriginalFile; to read and write the path to the folder containing the original file
- **property** Progress: integer read FProgress write SetProgress default 0; to set and get the progress of the generation or verification process
- **property** OnChange: TNotifyEvent read FOnChange write FOnChange; to indicate that the progress changes
- **property** PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param; to get and set the required parameters to use the XAdES object with a cryptographic token

The properties are:

- `property` Template: to select the signature template
- `property` Role: to select the signatory role

Example of how to generate a XAdES signature

```
var
  XAdES: TXAdES;
begin
  XAdES := TXAdES.Create;
  try
    XAdES.KeyFilePath:= '.\mykey.key';
    XAdES.CertFilePath:= '.\mycert.crt';
    XAdES.Packaging := enveloping;
    XAdES.PathToOriginalFile := ExtractFilePath(XAdES.KeyFilePath);
    XAdES.GenerateSignature('.\test.txt', '.\signature.xml');
  finally
    XAdES.Free;
  end;
end;
```

Example of how to verify a XAdES signature

```
var
  XAdES: TXAdES;
  Err : Integer;
  S, t: string;
begin
  XAdES := TXAdES.Create;
  try
    XAdES.PathToOriginalFile := '.';
    err := XAdES.VerifySignature('.\signature.xml');
    s:= XAdES.VerifyError(err);
    if err < 0 then
      t := XAdES.ErrorDetails;
  finally
    XAdES.Free;
  end;
end;
```

Example of how to sign an XML document with a cryptographic token in compliance with the Spanish Government invoice regulations

```
var
  XAdES: TXAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  XAdES := TXAdES.Create;
  XAdES.Template := xSpEnveloped; // Set Template

  try
    XAdES.Packaging := enveloped;
    XAdES.PKCS11Param.isToken := true;
    XAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
```

```
try
  XAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
finally
  PKCS11.Free;
end;
XAdES.PKCS11Param.DLLPath := 'aetpkss1.dll';
XAdES.PKCS11Param.Pin := '123456';
XAdES.GenerateSignature('.\\test.xml', '\\test_with_signature.xml');
Finally
  XAdES.Free;
end;
end;
```

All XAdES generation and verification functions are in the XAdESObj.pas file.

Caninicalization functions are in XMLCanon.pas and templates are located in XAdESTemplate.pas.

NOTE: Templates should not be modified. However, if you want to add or remove text in a template, make sure you adjust the relevant index (string position in template) in signature generation and verification functions.

CAAdES

CAAdES (short for "CMS Advanced Electronic Signatures") is a set of extensions to Cryptographic Message Syntax (CMS) signed data making it suitable for advanced electronic signatures. ETSI maintains and updates CAAdES.

CMS is a general framework for Electronic Signatures for various kinds of transactions like purchase requisition, contracts or invoices. CAAdES specifies precise profiles of CMS signed data making it compliant with the European eIDAS regulation (Regulation on electronic identification and trust services for electronic transactions in the internal market).

There are 4 profiles for CAAdES: CAAdES-B, CAAdES-T, CAAdES-LT, CAAdES-LTA. TMS Cryptography Pack supports **only** the CAAdES-B profile.

There are 2 signature formats:

- detached: the signature is detached from the document to sign.
- enveloping: the document to sign is added to the signature.

The CAAdES class is:

```
TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
    ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);
TPackaging = (detached, enveloping, enveloped);

TCAAdES = class(TTMSCryptBase)
public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(KeyFilePath: string; CertPath: string);
        reintroduce; overload;
    constructor Create(KeyFilePath: string; CertPath: string; fp: boolean);
        reintroduce; overload;
    constructor Create; reintroduce; overload;
    constructor Create(fp: boolean); reintroduce; overload;
    constructor Create(Cert: TX509Certificate; fp: boolean); reintroduce;
        overload;
    function GenerateSignature(FilePath: string; OutputFilePath: string;
        AdditionalData: string): string;
    function VerifySignature(FilePath: string; OriginalFile: string): integer;
    function VerifyError(err: integer): string;
{$IF defined(MSWINDOWS)}
    procedure LoadCertAndKeyFromPKCS12(FilePath: string; Password: string;
        PathToOpenSSL: string);
{$IFEND}
    function GetFileMIMEType(const AFileName: String): String;
{$IF defined(MSWINDOWS)}
    property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;
{$IFEND}
published
    property SignatureMethod: TSignatureMethod read FSignatureMethod;
    property KeyFilePath: string read FKeyFilePath write FKeyFilePath;
    property Packaging: TPackaging read FPackaging
        write FPackaging default enveloping;
    property CertFilePath: string read FCertPath write setCertPath;
    property ErrorDetails: string read FErrorDetails;
    property PathToOriginalFile: string read FPathToOriginalFile write
        FPathToOriginalFile;
    property Progress: integer read FProgress write SetProgress default 0;
```

```
property OnChange: TNotifyEvent read FOnChange write FOnChange;
property ForPAdES: boolean read FForPAdES write FForPAdES;
end;
```

The constructors are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor
- **constructor** Create(KeyFilePath: string; CertPath: string); reintroduce; overload; the constructor to set the path to the private key file and the path to the certificate file
- **constructor** Create(KeyFilePath: string; CertPath: string; fp: boolean); reintroduce; overload; the constructor to set the path to the private key file, the path to the certificate file and the ForPAdES property
- **constructor** Create(fp: boolean); reintroduce; overload; the constructor to set the ForPAdES property
- **constructor** Create(cert: TX509Certificate; fp: boolean); reintroduce; overload; the constructor to set the X509 certificate and the ForPAdES property

The public methods are:

- **function** GenerateSignature(FilePath: string; OutputFilePath: string; AdditionalData: string): string; to generate a CAdES signature of FilePath concatenated to AdditionalData in the file OutputFilePath
- **function** VerifySignature(FilePath: string; OriginalFile: string): integer; to verify the signature
- **function** VerifyError(err: Integer): string; to display the error associated to the error code err
- **procedure** LoadCertAndKeyFromPKCS12(FilePath: string; Password: string; PathToOpenSSL: string); to import a pfx encrypted file. We need the password to decrypt it and the folder path to the openssl.exe file. This function is available only on Windows
- **function** GetFileMIMETYPE(const AFileName: String): String; to know the MIME type of a file

The properties are:

- **property** SignatureMethod: TSignatureMethod read FSignatureMethod; to read the signature method
- **property** KeyFilePath: string read FKeyFilePath write FKeyFilePath; to read and write the path to the private key file
- **property** Packaging: TPackaging read FPackaging write FPackaging default enveloping; to read and write the packaging detached, enveloping or enveloped
- **property** CertFilePath: string read FCertFilePath write SetCertFilePath; to read and write the path to the certificate file
- **property** ErrorDetails: string read FErrorDetails; to read the error details of verifying signature
- **property** PathToOriginalFile: string read FPathToOriginalFile write FPathToOriginalFile; to read and write the path to the folder containing the original file
- **property** Progress: integer read FProgress write SetProgress default 0; to set and get the progress of the generation or verification process

- **property** OnChange: TNotifyEvent **read** FOnChange **write** FOnChange; to indicate that the progress changes
- **property** ForPADES: boolean **read** FForPADES **write** FForPADES; to indicate whether the CAdES signature is used for PADES
- **property** PKCS11Param: TPKCS11Param **read** GetPKCS11Param **write** SetPKCS11Param; to get and set the required parameters to use the CAdES object with a cryptographic token

Example of how to generate a CAdES signature

```
var
  CAdES: TCAdES;
begin
  CAdES := TCAdES.Create;
  try
    CAdES.KeyFilePath:= '.\mykey.key';
    CAdES.CertFilePath:= '.\mycert.crt';
    CAdES.Packaging := enveloping;
    CAdES.GenerateSignature('.\test.txt', '.\signature.p7m');
  finally
    CAdES.Free;
  end;
end;
```

Example of how to verify a CAdES signature

```
var
  CAdES: TCAdES;
  Err : Integer;
  S, t: string;
begin
  CAdES := TCAdES.Create;
  try
    err := CAdES.VerifySignature('.\signature.p7m', '.\test.txt');
    s:= CAdES.VerifyError(err);
    if err < 0 then
      t := XAdES.ErrorDetails;
  finally
    CAdES.Free;
  end;
end;
```

Example of how to sign a document with a cryptographic token

```
var
  CAdES: TCAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  CAdES := TCAdES.Create;
  try
    CAdES.Packaging := enveloping;
    CAdES.PKCS11Param.isToken := true;
    CAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
  try
    CAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
```

```
finally
  PKCS11.Free;
end;
CADES.PKCS11Param.DLLPath := 'aetpkss1.dll';
CADES.PKCS11Param.Pin := '123456';
CADES.GenerateSignature('.\test.txt', '.\signature.p7m');
Finally
  CADES.Free;
end;
end;
```

All CADES generation and verification functions are in the CADESObj.pas file.

PADES

PADES (*PDF Advanced Electronic Signatures*) is a set of restrictions and extensions to PDF and ISO 32000-1 making it suitable for Advanced Electronic Signature. This is published by ETSI as TS 102 778.

While PDF and ISO 32000-1 provide a framework for digitally signing their documents, PAdES specifies precise profiles making it compliant with the European eIDAS regulation (Regulation on electronic identification and trust services for electronic transactions in the internal market).

To sign a PDF by several signers, you need to sign original PDF by signer 1. Then you need to sign the signed PDF by signer 2, etc.

The PAdES class is:

```

TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
    ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);

TPAdES = class(TTMSCryptBase)
public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(KeyFilePath: string; CertPath: string);
        reintroduce; overload;
    constructor Create; reintroduce; overload;
    destructor Destroy; override;
    procedure GenerateSignature(FilePath: string; OutputFilePath: string);
        overload;
    procedure GenerateSignature(InStream: TStream; OutStream: TStream);
        overload;
    function VerifySignature(FilePath: string; OriginalFile: string): integer;
    function VerifyError(err: integer): string;
{$IF defined(MSWINDOWS)}
    procedure LoadCertAndKeyFromPKCS12(FilePath: string; Password: string;
        PathToOpenSSL: string);
{$IFEND}
    function GetFileMIMEType(const AFileName: String): String;
{$IF defined(MSWINDOWS)}
    property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;
{$IFEND}
published
    property SignatureMethod: TSignatureMethod read FSignatureMethod;
    property KeyFilePath: string read FKeyFilePath write FKeyFilePath;
    property CertFilePath: string read FCertPath write setCertPath;
    property ErrorDetails: string read FErrorDetails;
    property PathToOriginalFile: string read FPathToOriginalFile write
        FPathToOriginalFile;
    property Progress: integer read FProgress write SetProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
end;

```

The constructors and destructor are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor

- **constructor** Create(KeyFilePath: `string`; CertPath: `string`); reintroduce; over load; the constructor to set the path to the private key file and the path to the certificate file
- **destructor** Destroy; **override**; the default destructor

The public methods are:

- **procedure** GenerateSignature(FilePath: `string`; OutputFilePath: `string`); to generate a PAdES signature of FilePath in the file OutputFilePath
- **procedure** GenerateSignature(InStream: `TStream`; OutStream: `TStream`); to generate a PAdES signature of InStream in the stream OutStream
- **function** VerifySignature(FilePath: `string`; OriginalFile: `string`): `integer`; to verify the signature
- **function** VerifyError(err: `Integer`): `string`; to display the error associated to the error code err
- **procedure** LoadCertAndKeyFromPKCS12(FilePath: `string`; Password: `string`; PathToOpenSSL: `string`); to import a pfx encrypted file. We need the password to decrypt it and the folder path to the openssl.exe file. This function is available only on Windows.
- **function** GetFileMIMETYPE(const AfileName: `String`): `String`; to know the MIME type of a file

The properties are:

- **property** SignatureMethod: `TSignatureMethod` **read** FSignatureMethod; to read the signature method
- **property** KeyFilePath: `string` **read** FKeyFilePath **write** FKeyFilePath; to read and write the path to the private key file
- **property** Packaging: `TPackaging` **read** FPackaging **write** FPackaging **default** enveloping; to read and write the packaging detached, enveloping or enveloped
- **property** CertFilePath: `string` **read** FCertFilePath **write** SetCertPath; to read and write the path to the certificate file
- **property** ErrorDetails: `string` **read** FErrorDetails; to read the error details of verifying signature
- **property** PathToOriginalFile: `string` **read** FPathToOriginalFile **write** FPathToOriginalFile; to read and write the path to the folder containing the original file
- **property** Progress: `integer` **read** FProgress **write** SetProgress **default** 0; to set and get the progress of the generation or verification process
- **property** OnChange: `TNotifyEvent` **read** FOnChange **write** FOnChange; to indicate that the progress changes
- **property** PKCS11Param: `PPKCS11Param` **read** GetPKCS11Param **write** SetPKCS11Param; to get and set the required parameters to use the PAdES object with a cryptographic token

Example of how to generate a PAdES signature

```
var
  PAdES: TPAdES;
begin
  PAdES := TPAdES.Create;
```

```

try
  PAdES.KeyFilePath:= '.\mykey.key';
  PAdES.CertFilePath:= '.\mycert.crt';
  PAdES.GenerateSignature('.\test.pdf', '.\signature.pdf');
Finally
  PAdES.Free;
end;
end;

```

Example of how to verify a PAdES signature

```

var
  PAdES: TPAdES;
  Err : Integer;
  S, t: string;
begin
  PAdES := TPAdES.Create;
  Try
    err := PAdES.VerifySignature('.\signature.pdf');
    s:= PAdES.VerifyError(err);
    if err < 0 then
      t := PAdES.ErrorDetails;
  Finally
    PAdES.Free;
  end;
end;

```

Example of how to sign a PDF document with a cryptographic token

```

var
  PAdES: TPAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  PAdES := TPAdES.Create;
  try
    PAdES.Packaging := enveloped;
    PAdES.PKCS11Param.isToken := true;
    PAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
    try
      PAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
    finally
      PKCS11.Free;
    end;
    PAdES.PKCS11Param.DLLPath := 'aetpkss1.dll';
    PAdES.PKCS11Param.Pin := '123456';
    PAdES.GenerateSignature('.\test.pdf', '.\test_with_signature.pdf');
  Finally
    PAdES.Free;
  end;
end;

```

All PAdES generation and verifying functions are in the PAdESObj.pas file.

Random number generators

To generate random integers or random buffers, you can use the following functions (in RandomCore.pas):

```
procedure GetRandomBytes(var Output: array of UInt8; Len: integer);
procedure GetRandomBytes(var Output: array of UInt8);
function GetRandomNumber: UInt32;
function GetRandomNumber(MinValue, MaxValue: UInt32): Integer;
function GetRandomString(Len: Integer): string;
function GetRandomString(Len: Integer; FromChar, ToChar: Byte): string;
function GetRandomInt(): UInt32;
function GetRandomLongInt(): UInt64;
```

GetRandomYYYY fill the OutPut or Result with Len (or a fixed number of) random characters. On Windows, the functions use the same algorithm from the API, but in the other targets, they use /dev/random for RandomBuffer and /dev/urandom for RandomUBuffer.

Moreover, you can generate a random string by using the RandomString method of TConvert class.

ZIP encryption and decryption

To encrypt and decrypt ZIP archives with the AE-1 requirements, you can use the following functions (in ZIPCObj.pas):

```
TZCClass = class(TTMSCryptBase)
private const
    NumberOfIterations = 1000;
    ShaOneSizeBits     = 160;
    ShaOneSizeBytes    = 20;
    AuthSaltSizeBytes  = 10;
    SaltSizeBytes       = 16;
    EncryptionFieldSizeBytes = 11;
    ExtraWindowsFieldSizeBytes = 36;

private
    FFilePath: string;
    FPassword: string;
    FKey: TBytes;
    FSalt: TBytes;
    FEncryptionType: TEncryptionType;
    FFileList: TStringList;
    FZIPFile: boolean;
    FDestination: string;

    FAuthSalt: TBytes;
    FVerifCode: TBytes;

    procedure Encrypt(inBytes: TBytes; var outBytes: TBytes);
    procedure Decrypt(inBytes: TBytes; var outBytes: TBytes);
    procedure GetCentralDirectoryHeader(Buffer: TBytes; var P: integer; var CDH:
TZipEndOfCentralHeader);
    procedure GetDirectoryContent(Buffer: TBytes; var P: integer; var CDH: array
of TZCentralDirectoryHeader);
    procedure GetFileHeader(Buffer: TBytes; var P: integer; var FH:
TZCentralDirectoryHeader);
    function CRC32(Buffer: TBytes; N: UInt32; CRC: UInt32): UInt32;
    function DateTimeZipDateTime(DateTime: TDateTime): UInt32;
    procedure InitKeyAndSalt(const APassword: string; Value: TBytes);
    function ParseZipFile: integer;

protected
    procedure Init(const APassword: string; var Data: TBytes; AEncrypt: Boolean);
    procedure GetSeedAndCode(var Seed: TBytes; var Code: TBytes);
    function ComputeAuthenticationCode(CryptoGram: TBytes): TBytes;

public
    Constructor Create(FilePath: string; Pwd: string);
    Destructor Destroy;

    procedure EncryptZipArchive;
    procedure DecryptZipArchive;
    procedure DecryptZipFile(FilePath: string);
```

published

```
property Password: string read FPassword write FPassword;  
property EncryptionType: TEncryptionType read FEncryptionType  
    write FEncryptionType default etae2;  
property FileList: TStringList read FFileList write FFileList;  
property Destination: string read FDestination write FDestination;  
property Progress: Integer read FProgress write SetProgress default 0;  
property OnChange: TNotifyEvent read FOnChange write FOnChange;  
end;
```

Example code snippets for the ZIP encryption services are provided at <https://www.tmssoftware.com/site/blog.asp?post=2384>

The class doesn't contain all possible services for ZIP files and is limited to the AE-2 requirements for crypto. It has only been tested on Windows and would require adaptations to run on OSX.

Encrypt an ini file

Included in the TMS Cryptography Pack is also the non-visual class TEncryptedIniFile that offers the capability to store application settings in an encrypted INI file. TEncryptedIniFile descends from TMemInifile, so it inherits all methods to read and write various types (string, number, Boolean, ...) to an INI file. The encryption and decryption is done in memory, so at no time, the file system 'sees' an unencrypted file. TEncryptedIniFile uses internally AES 256-bit encryption. Further, the only difference with a regular TINIFile class is the added encryption key parameter in the constructor of TEncryptedIniFile.

The class definition looks like:

```
TEncryptedIniFile = class(TMemInifile)
private
  FFileName: string;
  FEncoding: TEncoding;
  FKey: string;
  FOnDecryptError: TNotifyEvent;
  FUni: TUnicode;
  FOutputFormat: TConvertType;

  procedure LoadValues;
public
  constructor Create(const FileName: string; const Key: string); overload;
  constructor Create(const FileName: string; const Encoding: TEncoding;
    CaseSensitive: Boolean); overload; override;
  constructor Create(const FileName: string; const Key: string; const Uni:
    TUnicode; const InputFormat: TConvertType; const OutputFormat:
    TConvertType); overload; override;
  procedure UpdateFile; override;
published
  property OnDecryptError: TNotifyEvent read FOnDecryptError
    write FOnDecryptError;
end;
```

A sample to use this class to read data back from such encrypted file is here:

```
const
  aeskey = 'anijd54dee1c3e87e1de1d6e4d4e1de3';
var
  mi: TEncryptedIniFile;
begin
  try
    mi := TEncryptedIniFile.Create('.settings.cfg', aeskey);
    try
      FTPUserNameEdit.Text := mi.ReadString('FTP', 'USER', '');
      FTPPasswordNameEdit.Text := mi.ReadString('FTP', 'PWD', '');
      FTPPortSpin.Value := mi.ReadInteger('FTP', 'PORT', 21);
      mi.WriteDateTime('SETTINGS', 'LASTUSE', Now);
      mi.UpdateFile;
    finally
      mi.Free;
    end;
  except
    ShowMessage('Error in encrypted file. Someone tampered with the file?');
  end;
end;
```

To ensure backward compatibility with TMS CP 2.5.1 and older, we added a third constructor:

```
constructor Create(const FileName: string; const Key: string; const Uni:
TUnicode; const InputFormat: TConvertType; const OutputFormat: TConvertType);
overload; override;
```

An ini file encrypted with TMS CP 2.5.1 can be decrypted by using the following code:

```
const
  aeskey = 'anijd54dee1c3e87e1de1d6e4d4e1de3';
var
  mi: TEncryptedIniFile;
begin
  try
    mi := TEncryptedIniFile.Create('.settings.cfg', aeskey, noUni, noFormat,
base64);
  except
    MessageDlg('Error: ' + mi, mtError, [mbOK], 0);
  end;
  FreeAndNil(mi);
end;
```

The TEncryptedIniFile class is in the TMSEncryptedIniFile file.

Generate a self-decrypting file

Included in the TMS Cryptography Pack is also the component TLockFile that offers the capability to generate a self-decrypted executable, i.e., a file able to decrypt itself. The user chooses a file and a password. The execute method encrypts this file with the password and add it as a resource of an executable (loaded from the resource of the current program) able to decrypt the file. The process uses AES-256, Argon2 and SHA-256. This component is available only on Windows platforms.

The component uses a resource named unlockfileres.RES where the unlocking executable is loaded.

You can find a demo of TLockFile in the TMS Software website:

<https://www.tmssoftware.com/site/freetools.asp#lockfile>

The component definition looks like:

```
TLockFile = class(TTMSCryptbase)
public
  constructor Create(AOwner: TComponent); overload; override;
  constructor Create; reintroduce; override;
  constructor Create(UnlockFilePath: string); reintroduce; overload;
  destructor Destroy; override;
  procedure Execute(encFile, password, outputFile: string);
published
  property Progress: Integer read FProgress write SetProgress default 0;
  property OnChange: TNotifyEvent read FOnChange write FOnChange;
  property UnlockFileExePath: string read FUnlockFileExePath
    write FUnlockFileExePath;
end;
```

The constructors and destructor are:

- **constructor** Create(AOwner: TComponent); overload; override; the default constructor from the TComponent class
- **constructor** Create; reintroduce; overload; the default constructor
- **constructor** Create(UnlockFilePath: string); reintroduce; overload; the constructor to set the path to the loaded resource
- **destructor** Destroy; override; the default destructor

The public methods are:

- **procedure** Execute(encFile, password, outputFile: string); to encrypt the file encFile with the password and save the generated executable in outputFile.

The properties are:

- **property** Progress: integer read FProgress write SetProgress default 0; to set and get the progress of the locking process
- **property** OnChange: TNotifyEvent read FOnChange write FOnChange; to indicate that the progress changes
- **property** UnlockFileExePath: string read FUnlockFileExePath write FUnlockFileExePath; to read and write the path to the folder containing the loaded resource

Example of how to use TLockFile

```
var
  TMSLockFile: TLockFile;
begin
  TMSLockFile := TLockFile.Create;
  try
    TMSLockFile.UnlockFileExePath := TPath.GetHomePath;
    TMSLockFile.Execute('.\\file.txt', 'password123!', '\\file_encrypted.exe');
  finally
    TMSLockFile.Free;
  end;
end;
```

Troubleshooting

[CHECK LATEST DISCUSSIONS AND TIPS AT
<https://support.tmssoftware.com/c/vcl/tms-cryptography-pack/29>]

There are several potential issues when running the various demos included in TMS Cryptography Pack.

From version 5.0.0

There are limitations with XAdES as the templates and computations do not satisfy all possible national variants.

Versions prior to 5.0.0

Known bugs

- The XAdES ‘canonicalization’ function is not fully compliant to the standard. In some cases the signature cannot be verified by other tools or libraries.
- The RSA 4096-bit operations can cause a ‘buffer overflow’ (in the C part of the code).

RandomDLL.DLL

It is necessary to copy this DLL in the appropriate folder to run the Windows 64 demo and to use the library in Windows 64 applications for RAD Studio version under 10.2.1.

Copy RandomDLL.dll from the Win64 directory of TMS Cryptography Pack:

- to C:\Windows\SysWOW64 if you are running 32 bit Windows
- or to C:\Windows\System32 if you are running 64 bit Windows

For versions after 10.2.1, you can bypass the use of RandomDLL.dll by uncommenting the line `// {$DEFINE IDEVERSION1021}` in `tmscrypto.inc` file.

libTMSCLib.a

Some error messages contain “... libTMSCLib.a not found”. In this case, the search path for the libraries needs to be updated. Go to Project->Options, then Search Path and click on “...” to update your list with the directory location of libTMSCLib.a (for instance “FULL TMS INSTALLATION PATH\iOSDevice64”).

C++ demo on Android and iOS

To use the C++ demo, you need to add the .a file to the project for Android or iOS target.

iOS Simulator

The TMS Cryptography Pack does not support the iOS Simulator because we generate .a files from C code and we cannot generate .a file for iOS Simulator target with RAD Studio.

Import a public/private key from an OpenSSL file

To do use the TRSAEncSign methods to import a public/private key from an OpenSSL file, you must have OpenSSL installed on Windows or OSX. For Android/iOS, OpenSSL is included in the libcrypto.a/libcrypto.so.1.0.0 (or more recent) and libssl.a/libssl.so.1.0.0 (or more recent) in libAndroid, libiOSDevice32 and libiOSDevice64 folders.

Bcrypt error message with C++ Builder

If you encounter an error message with Bcrypt functions and you use C++ Builder, you need to add bcrypt.lib to your project.

Windows XP Compilation

If you want to compile your application for Windows XP, you need to uncomment `//{$DEFINE WINXP}` in tmscrypto.inc file.

Import a certificate from a PFX file or export a PFX file from a certificate

These methods use openssl.exe and are only available for Windows platforms. There are openssl.exe files in libWin32 and libWin64 folder for both platforms. You need to set the environment variable OPENSSL_CONF to the path to the openssl.cnf file. You can add the following line to your project to set this variable:

```
ShellExecute(0, 'open', PChar('set'), PChar('OPENSSL_CONF=' +  
ExpandFileName(PathToOpenSSLConf + 'openssl.cnf')), nil, SW_SHOW);
```

Annex A: certificate formats

TMS CP supports electronic signatures using several certificate formats. These formats vary in complexity and may have many different options that are however not all supported by the library. The supported certificate formats are: X509, PEM (partially) and PKCS#12 (very partially).

All fields are encoded in one of the ASN.1 variants (see below) and converted to Base64 for *convenience*.

TMS CP contains a simple ASN.1 parser that is usually good enough for certificates. However, because of the number of options and versions, decoding sometimes fails due to unknown fields or unknown object identifiers. You can report these failures to our technical support team⁶:

<https://support.tmssoftware.com/c/vcl/tms-cryptography-pack/29>

These formats are also used to store private keys that are usually encrypted with a key derived from a password. TMS CP only supports AES for key encryption and there is no method (except for dictionary and brute-force attacks) to recover a *lost* password.

Here is a quick description of the certificate formats.

X509 Certificate Format

X509v3 is the current ISO standard for cryptographic certificates. Like many standards, X509 proposes options that may be difficult to decode with a basic parser, the main difficulty being the decoding of the endless list of Object Identifiers (OIDs), as OIDs may be created by local authorities or vendors and may not always be registered (or the online database may not be updated in a timely manner).

Good sources for OIDs are <https://oid-rep.orange-labs.fr/> or <http://www.oid-info.com/>

X509v3 is the standard for certificates stored in cryptographic tokens using the PKCS#11 standard for the provision of cryptographic services.

We recommend the use of PKCS#11 tokens to sign documents (at least official ones) as these tokens are issued by recognized Certification Authorities and provide extra integrity protection measures of the certificate and keys.

PEM Certificate Format

PEM, which stands for privacy-enhanced mail, is a legacy but popular format used by certificate authorities (CAs), to issue SSL certificates, and by various applications for end users.

PEM files contain ASCII (or Base64) encoding data and the certificate files can bear a .pem, .crt, .cer, or .key extension. A PEM file may contain several certificates, such as the server certificate, the intermediate certificate and the private key. A PEM file may also only contain a private key (with the associated public key) or just a public key.

Each PEM file contains a “-BEGIN XXX” - and “-END XXX” statement. For example:

- A certificate is contained between the -----BEGIN CERTIFICATE----- and --- --END CERTIFICATE----- statements.
- A private key is contained between the -----BEGIN RSA PRIVATE KEY----- and -----END RSA PRIVATE KEY----- statements.

⁶ Note that this is not a commitment to fix the issue, especially for deprecated algorithms.

PEM certificates may contain information beyond the BEGIN – END statement boundaries.

More information can be found in RFC 7468 (Textual Encodings of PKIX, PKCS, and CMS Structures).

P7B/PKCS#7 Certificate Format

Certificates in P7B or PKCS#7 formats are encoded in Base64 and they usually have .p7b or .p7c file extension. Contrary to PEM files, PKCS#7 files can only store certificates and no private keys.

These certificates are contained between the “-----BEGIN PKCS7-----” and “-----END PKCS7-----” statements and have been superseded par PEM certificates but can still be found in legacy applications.

More information can be found in RFC 7468 (Textual Encodings of PKIX, PKCS, and CMS Structures).

DER Certificate Format

DER stands for “distinguished encoding rules and is one of the binary ASN.1 formats. A DER certificate file, with a .der extension, is simply a binary form of a PEM certificate, that is encoded in Base64. A DER file can include certificates and private keys of all types.

Certificates can also have .cer extension, which stands for Canonical Encoding Rules, another binary form of ASN.1.

PFX/P12/PKCS#12 Certificate Format

The PFX (or P12 or PKCS#12) format is a more recent format, originally designed by Microsoft. It is described in RFC 7292 (PKCS #12: Personal Information Exchange Syntax v1.1) and compliant files can store a server certificate, an intermediate certificate and an associated private key in a single password-protected pfx or .p12 file.

PFX is by far the most complex (and cumbersome) format to decode due to the many possible options and cases that the standard allows.

TMS CP only addresses a subset of these options, in practice files where only one certificate and one private key are stored.