DataAccess
WORLDWIDE

DataFlex Security library v1.0.0

DATA ACCESS WORLDWIDE

Chapter 1 Introduction

During the last few years numerous cyberattacks have made the headlines: ransomware, DDOS attacks, and election manipulation is becoming more popular every day. The people at Data Access saw this happening and took their time to create a library that has everything on board to improve security of DataFlex applications drastically.

This manual is intended for DataFlex developers. Prior knowledge of security or encryption are useful, but not necessary to implement better security. That said, we strongly encourage you to read more on the subject. Good security is not only about techniques, but mostly about making the right decision for each specific scenario. Even though this library can make your applications secure, it is the way you apply the functionality that makes or breaks your security. Data Access can therefore not take any responsibility for the implementation choices you make, nor for any bugs or security holes present in the library.

Design goals

During the last few years Data Access has gained significant experience on security in software, which has resulted in the release of the DataFlex Security library (DFSecurity). This library has a solid yet flexible design based on a number of core values.

Simplicity

The library's public interface was designed to be very easy to use. Data Access realized that many software developers do not have intimate knowledge of security, nor should they not need to. The library contains all the functionality you need, and this manual will guide you to the right class to use for your use case.

Flexibility

The world of cyber-security continually evolves at an astonishing rate. The encryption algorithms of today may no longer be sufficiently secure tomorrow. Computers become faster, bugs are discovered in software libraries, and weaknesses are found in algorithms. The security library was designed to be flexible and enable you to upgrade your software to stronger parameters, better algorithms, or even different security engines without any significant changes to your code.

Stability

A large number of automated unit tests assure *every* function the library exposes is tested before release. This ensures correct implementation of each hashing or encryption algorithm, and prevents code from breaking after an update.

Before you begin

It is useful to know a few concepts on security – this will help you understand the design choices made.

Cryptographic algorithms act on raw binary data. In DataFlex this is best represented by the UChar array data type and the DFSecurity library uses this data type a lot. If you need to store such binary data somewhere, you may want to convert it to a string value using base64 or base32 encoding. The security library's global object ghoSecurity provides methods for this.

The code samples in this document have been simplified for clarity. Error handling is kept to a minimum, unless it is needed to illustrate the current topic.

Make sure you have the latest DataFlex Studio installed on your system, and that you can run the Order Entry example application without any issues. Web Application samples additionally require IIS to be installed, and the WebOrderMobile example workspace is a good one to test all technical requirements.

Chapter 2 Adding the library to your workspace

As mentioned before, the DataFlex Security library has been designed to be both easy to use and flexible. These principles are already apparent when you add it to your workspace. The DFSecurity Core library should be added to your workspace using the *Maintain Libraries…* option in the *Tools* menu of the DataFlex Studio.

Once the library is added to the workspace you can use the DFSecurity base classes, but there is no real functionality yet. The library does not include any direct support for any security engine – you need to add one or more additional libraries. The reason for this choice is simple: flexibility.

Currently, two cryptography engines are available and supported:

- CNG – the Microsoft Cryptography API: Next Generation[1] (CNG). This engine is the successor to the original CryptAPI (wincrypt), and is also readily available in Windows, since Vista and Server 2008. It contains a number of stronger algorithms, which usually makes it a better choice than wincrypt.
- Libsodium – a cross-platform free and open-source library (DLL) supporting a number of highly secure and modern algorithms. If you are free in choosing your algorithms, this engine is a good choice.

It is advisable to use the best algorithm you can when implementing new functionality, but sometimes software needs to communicate with other software and the required algorithms for that are not available. Luckily DFSecurity supports the usage of multiple engines in parallel, so you can add support for any algorithm needed by adding multiple libraries to your workspace.

[1] Yes, that is its official name.

Chapter 3 First steps - generic hashes

We will gently introduce the various ideas and concepts of the library to you step by step. This chapter fully discusses the simple generic hashes and how to use them.

Generic hashes

Generic hashes are fixed-length fingerprints for an arbitrary long message. A tiny change to the message (even a single bit) will result in a massively different hash value. When using a good algorithm, it is virtually impossible to change a message without changing the hash[2].

Generic hashes are mainly used for file integrity checking, or creating unique identifiers to index data.

Security theory

Hashes are generated from a *message*. It is not possible to directly recover the message from a hash. Unfortunately this is not entirely true for every application. Using generic hashes for storing passwords for instance, is a very bad idea. Modern computers can calculate millions of hashes per second, so rainbow tables[3] or a dictionary attack will allow a hacker to find a lot of matches with little effort.

Design choice – static or dynamic objects?

Historically it is very common in DataFlex to use statically defined objects. This means an object is defined in code with an *Object* command, and it can be referenced by its name. DataFlex also supports dynamic objects, instantiated by the *Create* and *CreateNamed* methods. These return an object handle that can be used to reference it, and the object must be destroyed manually as well (using the *destroy* method). The Security library aims to supports both syntaxes for every public class, and the generic hashes are a very good example illustrating the reasons behind that decision.

Generating a file hash (static object approach)

One common use case for a generic hash is a file hash. Minor changes to a file will result in a completely different hash, so this hash can be used to check a file’s integrity. The code below is a simple example of this.

The *cSecureHash* class exposes the generic hashing functionality. Generating a hash consists of three phases:

1. Initialize the engine and set it up for the desired algorithm.
2. Feed raw data to the hash object. This can be done multiple times.
3. Finalize the hash and get the resulting value.

[2] Some algorithms are vulnerable to length extension attacks though…
[3] We cannot explain everything in this document. Search the internet for more information about a subject if you are interested.

```
Use DFSecurity_CNG.pkg

Object oFileHash is a cSecureHash
  Set piHashImplementation to C_SEC_HASH_CNG_SHA256
  Send Initialize

  Function HashForFile String sPathAndFileName Returns UChar[]
    UChar[] ucaData
    UChar[] ucaHash

    Direct_Input sPathAndFileName
    While (not(SeqEOF))
      Read_Block ucaData 4096
      Send Update ucaData
    Loop
    Close_Input

    Get Finalize to ucaHash

    Function_Return ucaHash
  End_Function

End_Object
```

In this example the chosen hash algorithm is SHA256 and the CNG engine is used. This is clearly defined by the *piHashImplementation* property which is set to a *C_SEC_HASH_** constant. These constants are provided by the various engine libraries, so depending on which engine libraries you use additional constants may be available. The object is initialized at object creation, which means that you cannot change the *piHashImplementation* or other properties later on.

Once the hash object is initialized, the *Update* method adds data to the current hash calculation. This means that a single hash object can generate a single hash at any time, until it is finalized. You can call *Update* once or multiple times, depending on your needs. This can save some memory, or you can provide a “Cancel” option somehow.

When all the data has been fed to the hash object, *Finalize* will return the resulting hash value. The size of this hash value is dependent on the algorithm used: SHA256 will return 256 bits (32 bytes), but for example MD5 will return only 16 bytes. Libsodium supports the blake2b algorithm, which can be configured to return at least 16 and at most 64 bytes. The *cSecureHash* class has a *piOutputBytes* property that can be used to configure this.

Hash object reuse

When you have finalized the hash, the object is immediately available for reuse. The next call to *Update* will start a new hash. Some engines do not natively support this, but the DFSecurity engine wrappers work around such limitations.

Reuse is only supported with the same settings, such as the engine and algorithm. It is bad practice to use a single hash object for different types because it can easily lead to confusion. If you need to be flexible it is better to create and destroy objects dynamically.

Generating a file hash (dynamic object approach)

The same file hashing as before can be implemented using dynamic object creation. This is mostly useful when the engine and/or algorithm must be flexible during program execution.

```
Use DFSecurity_CNG.pkg

Function HashForFile String sPathAndFileName Integer iHashImpl Returns UChar[]
  Handle  hoHash
  UChar[] ucaData
  UChar[] ucaHash

  Get Create (RefClass(cSecureHash)) to hoHash
  Set piHashImplementation of hoHash to iHashImpl
  Send Initialize of hoHash

  Direct_Input sPathAndFileName
  While (not(SeqEOF))
    Read_Block ucaData 4096
    Send Update of hoHash ucaData
  Loop
  Close_Input

  Get Finalize of hoHash to ucaHash
  Send Destroy of hoHash

  Function_Return ucaHash
End_Function

...

Get HashForFile sFile C_SEC_HASH_CNG_SHA256 to ucaFileHash
```

It is very important not to forget to destroy the object when you're done. If you don't, you will eventually run out of handles or memory.

Chapter 4 **Keyed hashes**

In some situations a hash value is desired, but it should be hard for anyone to generate unless it is a trusted party. For instance, when sending a file across the network a hacker may insert ransomware into it and generate a new hash value. If generic hashes are used you will not be able to detect the tampering. That is where keyed hashes such as Hash-based Message Authentication Codes (HMAC) are useful.

A keyed hash can be generated very fast, but also includes a secret key in the process. It is not possible for a hacker to change the message as well as the HMAC within reasonable time, unless he has the key.

Generating a keyed hash (static object approach)

DFSecurity uses the same *cSecureHash* class for keyed hashes, but requires the *piHashAlgorithm* to be a keyed algorithm.

```
Use DFSecurity_Libsodium.pkg

Object oKeyedDataHash is a cSecureHash
  Set piHashImplementation to C_SEC_HASH_LIBSODIUM_BLAKE2b
  Set piOutputBytes to 16  // default is 32

  Procedure Initialize
    UChar[] ucaKey

    Send ReadTheKeyFromASecureLocation (&ucaKey)

    Forward Send Initialize (&ucaKey)
  End_Procedure

  Function HashForData UChar[] ucaData Returns UChar[]
    UChar[] ucaHash

    Send Update ucaData
    Get Finalize to ucaHash

    Function_Return ucaHash
  End_Function

End_Object
```

In this example the chosen hash algorithm is blake2b and the libsodium engine is used. To illustrate how to change a hash length, the default hash length is reduced from the default 32 to 16. Each engine can provide several keyed hash algorithms – CNG for instance provides the well-known HMAC-SHA1.

The most important part is to add an UChar array parameter containing the secret key to *Initialize*. This secret key must be kept secret at all times, and it should be possible to change this key without recompiling the application. In this example, a fictional method *ReadTheKeyFromASecureLocation* retrieves the key and returns it.

Once the hash object is initialized, everything is the same as for a normal keyless hash. The engine wrappers make sure you never need to provide the key again while the object lives.

Security theory

Note that the key variable is passed by reference. This ensures that we do not have multiple copies of the key in memory. The library will overwrite it with zeroes before returning from Initialize, so it will only be in memory for a very short period of time.

Generating a keyed hash (dynamic object approach)

Similar to the generic hash, a keyed hash can be created using the dynamic approach. This can be most useful when many different keys (e.g. user specific) are in use, or when the key is generated from some form of key agreement.

Chapter 5 Passcode storage methods

Even though everyone still uses the term password, we believe passcode is more correct. A single word is not good enough anymore. Users should use a long random code or a passphrase.

Passcode storage methods are only useful for user authentication – not for storing a password your system needs, e.g. for sending e-mails. You can store and verify a passcode, but you cannot retrieve the original passcode from the stored value.

Proper passcode storage methods are very different from generic or keyed hashes. Over the past few years we learned that many users choose passwords that are very easily guessed[4]. Dictionary attacks have become very effective due to the speed of hash calculations.

Passcode hashing functions derive a secret key of any size from a passcode and a salt, and have a number of characteristics:

- The generated key has the size defined by the application, no matter what the password length is.
- The same password hashed with the same parameters will always produce the same output.
- The same password hashed with different salts will produce different outputs.
- The function deriving a key from a password and a salt is CPU intensive and intentionally requires a fair amount of memory. Therefore, it mitigates brute-force attacks by requiring a significant effort (and cost) to verify each password.

In practice many inexperienced software developers make mistakes when implementing password hashing, either because they reuse salts, don't use a cryptographically secure random number generator for the salts, or choose a generic hash algorithm instead of a specialized password hashing algorithm. The Security library therefore hides most technical aspects from the developers, because the only two things you should care about are:

- Turn the plaintext password into some secure data to store.
- Verify a plaintext password against that stored data.

Those two steps are implemented by the *StorageString* and *Verify* methods of the *cSecurePasscodeStorageMethod* class.

[4] We strongly advise checking new passcodes against a list of known bad passcodes, such as the HaveIBeenPwned database, and rejected them if they are in there.

Basic password hash usage

```
Use DFSecurity_Libsodium.pkg

Object oPasscodeStore is a cSecurePasscodeStorageMethod
  Set piPasscodeHashImplementation to C_SEC_PWHASH_LIBSODIUM_ARGON2I
  Set piMemLimit to crypto_pwhash_argon2i_MEMLIMIT_INTERACTIVE // default
  Set piOpsLimit to crypto_pwhash_argon2i_OPSLIMIT_INTERACTIVE // default
  Send Initialize
End_Object

Procedure Main
  Boolean bOk
  String  sPasswordStorageString
  UChar[] ucaPassword

  Move (StringToUCharArray(“Password”)) to ucaPassword // the worst password...
  Get StorageString of oPasscodeStore (&ucaPassword) to sPasswordStorageString
  // ucaPassword is emptied automatically for security reasons

  Move (StringToUCharArray(“Password”)) to ucaPassword
  Get Verify of oPasswordHash (&ucaPassword) sPasswordStorageString to bOk

  If bOk Send Info_Box “The password is correct.”
  Else Send Stop_Box “Wait, what...!? Impossible!”
End_Procedure
```

Creating a cSecurePasscodeStorageMethod object is quite similar to a generic hash. Note that the hash implementation must be a *C_SEC_PWHASH_** constant, which is a different set than the supported generic hashes.

Password hashing algorithms can be tweaked to take a certain amount of time to calculate on specific hardware. It is advised for user logins to aim for about a second – this is hardly noticeable for the user logging in, while it is a real pain for brute force attackers. The library comes with a separate project template that you can use to find an estimate of optimal parameters, which is discussed in the next subsection.

One thing that may seem counterintuitive is that the *StorageString* and *Verify* methods assume the password to be provided as a binary data type – not a string. The reason for this is that the hashing algorithm actually does not care, and strings have the nasty habit of being encoded in OEM, or ANSI, or UTF-8, and so on. The Security library does the safest thing: make no assumptions. It is your responsibility to provide consistent data.

The storage string however *is* a string. This is not a problem, because every engine wrapper must make sure the characters are plain 7-bit ASCII, meaning it is the same for OEM, ANSI, or UTF-8.

Passcode storage parameter estimation tool

File -> New -> Project... -> “Passcode Storage Parameter Estimation Project”

Build & Run.

WebApp integration

In order to use this improved passcode storage scheme in a standard WebApp there are several other things that need to be done:

- Add functionality to the cWebAppUserDataDictionary class so that *Set Field_Changed_Value* of a password field includes the proper storage conversion.
- Expand the *WebAppUser.Password* field length (200 or more to be future-safe).
- A change to the *SessionManager* method *ComparePasswords*.
- Several views to properly handle the flow for a password reset.

With this setup it is very straightforward to upgrade the storage method so that new and changed password are stored using an improved algorithm or higher complexity. If desired the storage method properties can be made configurable via the front-end so that recompilation is not even needed. Such an exercise is beyond the scope of this document.

cWebAppUserDataDictionary example

```
Object oMyMethod is a cSecurePasscodeStorageMethod
    Set piPasscodeHashImplementation to C_SEC_PWHASH_LIBSODIUM_ARGON2I
    Set piMemLimit to (256*1024*1024)
    Set piOpsLimit to 3
    Send Initialize
End_Object

Class cWebAppUserDataDictionary is a DataDictionary
    Import_Class_Protocol cSecurePasscodeStorageMethod_DD_Mixin

    Procedure Construct_Object
        Forward Send Construct_Object

        Send Define_cSecurePasscodeStorageMethod_DD_Mixin

        ...

        Set Field_PasscodeStorageObject Field WebAppUser.Password to oMyMethod
    End_Procedure

    ...

End_Class
```

The *cSecurePasscodeStorageMethod_DD_Mixin* class adds the *Field_PasscodeStorageObject* field property to a DD class. When this property is set for a field, every time the *Field_Changed_Value* or *Field_Current_Value* of that field is set, this value is routed through the defined *cSecurePasscodeStorageMethod* and the storage string is put into the value instead of the raw plaintext value.

New users should not have an active password – not even a difficult one. As long as the field remains empty it will not be verifiable. This means the user will have to perform a password reset in order to activate his login.

SessionManager extension

```
Function ComparePasswords String sUserPass String sEnteredPass Returns Boolean
    Boolean bMatch
    UChar[] ucaPasscode

    Move (StringToUCharArray(sEnteredPass)) to ucaPasscode
    Send SecureStringOverwrite of ghoSecurity (&sEnteredPass)

    Get VerifyPasscode of ghoSecurity (&ucaPasscode) sUserPass to bMatch

    Function_Return bMatch
End_Function
```

By default this example code will route the passcode verification to the correct engine for verification. If you still have (some) plaintext passwords in your database you can set the *pbFallbackToPlaintext* property of *ghoSecurity* to true. This is *not* meant for production systems, but may be useful during development. You should upgrade existing plaintext passwords to a secure storage method as soon as possible.

Chapter 6 One-Time Passwords

The security library has built-in support for both Time-Based (TOTP) and Hash-Based (HOTP) one-time passwords, as defined in RFC's 4226 and 6238. In addition, the newer FIDO U2F standard, which uses hardware tokens, can be implemented as well for web applications.

OTPs can be used as a second factor for the authentication processes. TOTPs are becoming mainstream and a multitude of smartphone apps support it. HOTPs are more often supported by hardware, such as USB security keys or smartcards. FIDO U2F is a newer standard using USB or NFC tokens that is more secure and easier to use than the other methods. Unfortunately browser support may be problematic – especially on smartphones.

HOTPs can also be used as a second factor for machine users, such as a web service consumer. This ensures that intercepted communication does not enable the attacker to make any successful queries. Using HOTP is better than TOTP for this scenario, because it mitigates the risk of a replay attack.

Usability considerations

TOTPs use the current time as an input for the calculation. If the user's TOTP calculator time significantly deviates from the server's time the tokens will be reported as invalid. The library provides a configurable number of earlier and future tokens to be accepted as well. These default values seem to work just fine when users use their smartphone and the server is kept on time using ntp (network time protocol).

HOTPs are counter-based. Both the client and the server keep track of the counter's value and sometimes a mismatch occurs. The library accepts a configurable number of counter values to be considered valid, and reports back the new counter value in the *piCounter* property. It is your responsibility to store this counter value in the database, resynchronizing is with the client. HOTP implementations must accept 2 or more OTP's with consecutive counter values to mitigate brute force attacks.

Using the default settings and above recommendations, issues will rarely occur.

Security considerations

Both the calculation of an OTP and its validation requires possession of the secret. OTPs are therefore only useful when used between two separate entities, such as logging in onto a web application. It is *not suitable* for use in local Windows applications, because the program needs access to the secret and thus a hacker can obtain it.

WebApp integration

To simplify integration of one-time passwords into DataFlex web applications we included a custom web control: c2FAWebGroup. This group does not contain any visible controls, so developers are free to define those themselves.

First, the JavaScript files for using this control must be added to your index.html. You may also want to include the necessary files for the cWebQrCode class, which allows users to scan the TOTP secret with their smartphone.

```
<script src="Custom/u2f-api.js"></script>
<script src="Custom/2FAWebGroup.js"></script>

<script src="qrcode/qrcode.min.js"></script>
<script src="Custom/WebQrCode.js"></script>
```

Now you can add a c2FAWebGroup to your login view and the user profile view. We strongly encourage you to create it from the Class Palette in the Studio. This uses our template which includes a number of to-dos for proper usage.

Setting up TOTP or FIDO U2F is complicated and cannot easily be presented in this document. Please refer to the online course about security in the DataFlex Learning Center for a complete set of videos and a sample workspace[5].

[5] This is not yet available at the time of writing. Release has been planned for later in 2018.

Chapter 7 **Encryption**

Encryption is a powerful method for keeping data secret. The basics of encryption are relatively easy to comprehend, but choosing the right parameters (algorithm, chaining method, key length, etc.) is a lot trickier.

The security library supports several forms of encryption, which are both forms of symmetric key algorithms. These algorithms use the same key for both encrypting and decrypting the data.

Terminology

In the example I chose to use the AES *block algorithm*. An encryption algorithm consists of the data manipulations that are performed to turn readable data into something unrecognizable, and of course the manipulations needed to turn it back into readable form. Block algorithms do not manipulate a bit or byte at a time, but an entire block of data. For AES these blocks are 16 bytes (128 bits) in size.

If the length of the data to be encrypted does not equal a multitude of the block size, the last block will need to be expanded. This is called padding. If the data length does equal a multitude of the block size an entire padding block will be added for technical reasons. This means that the encrypted data is always a bit larger than the plaintext data.

Each block algorithm describes how to encrypt and decrypt a block, but data very often is larger than a single block. If the same block operation is performed for each consecutive block this may result in recognizable patterns. There are some clear examples of this on the internet[6]. To increase randomness a number of chaining methods have been defined. Most of these take the result of the previous block as an additional input factor for the next. *Electronic Codebook* (ECB) is the method that you should avoid at all times. CBC is quite good and the most commonly used method, which is why we chose to make it available in the security library.

If multiple messages start with the same plaintext data, the encrypted data for each message will start the same as well. This is undesirable[7]. To mitigate this problem the block should be initialized with something random – an *initialization vector* (IV). This IV must be generated randomly for every piece of data and does not have to stay secret – you *need* to store/distribute the IV with the encrypted data and use it for decryption as well. The security library by default generates a new IV every time you start encrypting, so you don't have to worry about that.

Encrypting data using AES-CBC

The sample below generates a new random key to encrypt and decrypt an array of data using AES in CBC chaining mode with a 256 bits key. Note that the block length will still be 128 bits – the key length does not change that.

[6] Wikipedia has more information: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
[7] In very specific situations this behavior may be required, but it should be a conscious decision.

```
Object oMyEncryptionMethod is a cSecureSymmetricKeyEncryptionMethod
    // ToDo: Set the symmetric key encryption implementation.
    Set piEncryptImplementation to C_SEC_SYMENC_CNG_AES256_CBC

    Procedure Initialize
        Integer iKeyLen
        UChar[] ucaKey

        // generate a new key
        Get MinimumKeyLength of ghoSecurity (piEncryptImplementation(Self)) ;
            to iKeyLen
        Get RandomData of ghoSecurity iKeyLen to ucaKey
        Forward Send Initialize (&ucaKey)
    End_Procedure

    Send Initialize
End_Object

Procedure TestEncryptAndDecrypt UChar[] ucaData
    Handle  hoDec
    Handle  hoEnc
    UChar[] ucaCipher
    UChar[] ucaIV
    UChar[] ucaPlain

    // encrypt ucaData
    Get NewEncryptor of oMyEncryptionMethod to hoEnc
    Get pucaIV of hoEnc to ucaIV
    Get EncryptChunk of hoEnc ucaData to ucaCipher
    Send Destroy of hoEnc

    // decrypt ucaCipher
    Get NewDecryptor of oMyEncryptionMethod ucaIV to hoDec
    Get DecryptChunk of hoDec ucaCipher to ucaPlain
    Send Destroy of hoDec

    // compare results
    If (IsSameArray(ucaPlain, ucaData)) Begin
        Send Info_Box "Encryption and decryption were successful."
    End
    Else Begin
        Send Info_Box "Something went terribly wrong here..."
    End
End_Procedure
```

Each encryption method object consists of an encryption implementation (algorithm, chaining mode, key size) and a key. This key only needs to be loaded once and the library will keep it as safe as possible.

Encryption is performed using an *Encrypter* object. This object generates a new random IV upon creation. One or more calls to *EncryptChunk* will turn plaintext data into encrypted ciphertext. Make sure each chunk of data is a multiple of the block size, unless it is the last chunk.

Decryption is performed by a *Decrypter* object that requires the IV used at encryption time. The *DecryptChunk* method follows the same rules as *EncryptChunk*.

Encrypting data using AES-GCM

Sometimes it is now enough to keep data secret. You may want to make sure it has not been damaged in transit or tampered with. It is quite simple to use a generic or keyed hash algorithm for this, but it is not so simple to prove whether the combination of encrypted data and the hash is safe from manipulation.

Luckily for us, some brilliant people have come up with methods to do it right. AES-GCM is an example of an *Authenticated Encryption* mode. When used correctly it will encrypt data and calculate a secure authentication tag (a sort of hash) at the same time. While decrypting it uses that authentication tag to verify that the message is complete and unchanged. It is safe to distribute the authentication tag with your encrypted data.

GCM uses the IV a bit differently than CBC – it consists of a random part (usually called a *nonce*) and a sequential counter. The nonce is usually 12 bytes, leaving 4 bytes for the counter. The advantage of using AES-GCM is not only that it provides both encryption and a hash, but also that recent CPUs have been optimized for handling it. AES-GCM with CPU hardware acceleration is blazing fast.

The sample below generates a new random key to encrypt and decrypt an array of data using AES in GCM chaining mode with a 256 bits key. Note that the block length will still be 128 bits and the nonce is 12 bytes.

```
Object oMyAuthEncMethod is a cSecureSymmetricKeyEncryptionMethod
    // ToDo: Set the authenticated encryption implementation.
    Set piEncryptImplementation to C_SEC_AUTHENC_CNG_AES256_GCM

    ...

End_Object

Procedure TestEncryptAndDecrypt UChar[] ucaData
    Boolean bIsAuthentic
    Handle  hoDec
    Handle  hoEnc
    UChar[] ucaAuthTag
    UChar[] ucaCipher
    UChar[] ucaNonce
    UChar[] ucaPlain

    // encrypt and hash ucaData
    Get NewEncryptor of oMyAuthEncMethod to hoEnc
    Get pucaNonce of hoEnc to ucaNonce
    Get EncryptLastChunk of hoEnc ucaData to ucaCipher
    Get AuthenticationTag of hoEnc to ucaAuthTag
    Send Destroy of hoEnc

    // decrypt and verify ucaCipher
    Get NewDecryptor of oMyAuthEncMethod ucaNonce ucaAuthTag to hoDec
    Get DecryptLastChunk of hoDec ucaCipher to ucaPlain
    Get IsAuthentic of hoDec to bIsAuthentic
    Send Destroy of hoDec
    // compare results
    If not bIsAuthentic Send Info_Box "Something went terribly wrong here..."
    Else Send Info_Box "Encryption and decryption were successful."
End_Procedure
```

In the example for AES-CBC we compared the original data with the decrypted data to determine whether it was correct. In real life this is not possible and you would have to trust that the decrypted output is correct. With AES-GCM the authentication tag helps you to make sure the message can be trusted.

Additional authenticated data (AAD)

Sometimes it is useful or required to verify the authenticity of some unencrypted data in addition to the encrypted data. This may for example include some unencrypted message headers needed for routing the encrypted message over a network. If the authenticity of these headers is important they may be supplied to the encryption method as additional authenticated data (AAD). This data is included in the authentication tag calculation, but not in the encrypted data.

The example below is a minor variation of the previous one, with the additional of some AAD. The changes have been highlighted.

```
...

Procedure TestEncryptAndDecrypt UChar[] ucaData
    Boolean bIsAuthentic
    Handle  hoDec
    Handle  hoEnc
    UChar[] ucaAAD
    UChar[] ucaAuthTag
    UChar[] ucaCipher
    UChar[] ucaNonce
    UChar[] ucaPlain

    Move (StringToUCharArray("From Me, To You")) to ucaAAD

    // encrypt and hash ucaData
    Get NewEncryptor of oMyAuthEncMethod to hoEnc
    Set pucaAAD of hoEnc to ucaAAD
    Get pucaNonce of hoEnc to ucaNonce
    Get EncryptLastChunk of hoEnc ucaData to ucaCipher
    Get AuthenticationTag of hoEnc to ucaAuthTag
    Send Destroy of hoEnc

    // decrypt and verify ucaCipher
    Get NewDecryptor of oMyAuthEncMethod ucaNonce ucaAuthTag ucaAAD to hoDec
    Get DecryptLastChunk of hoDec ucaCipher to ucaPlain
    Get IsAuthentic of hoDec to bIsAuthentic
    Send Destroy of hoDec

    ...
End_Procedure
```

Chapter 8 Supported algorithms

When advice below mentions “Do not use” this means it should not be used for any new software. These algorithms can of course be used for backwards compatibility or communication with external software. Also when you do not wish to include libsodium, you may want to choose unadvised algorithms in CNG – at your own risk of course.

Generic hash algorithms

Algorithm	Hash length (bytes)	CNG	libsodium	Advice/notes
MD2	16	X		Do not use.
MD4	16	X		Do not use.
MD5	16	X		Do not use.
SHA1	20	X		Do not use.
SHA256	32	X	X	
SHA384	48	X		Provides some protection against length extension attacks.
SHA512	64	X	X	
blake2b	16-64 (32)		X	Use if possible. Very strong, very fast, and safe against hash length extension attacks.

Keyed hash algorithms

Key lengths are arbitrary, but usually should match the algorithm’s block length. If the key is larger than a single block length, usually a generic hash of the key is used.

Algorithm	Hash length (bytes)	CNG	libsodium	Advice/notes
HMAC-MD5	16	X		
HMAC-SHA1	20	X		
HMAC-SHA256	32	X	X	
HMAC-SHA384	48	X		
HMAC-SHA512	64	X	X	
HMAC-SHA512256	32		X	Truncated HMAC-SHA512.
blake2b	16-64 (32)		X	Use if possible. Very strong, very fast.

Passcode storage method algorithms

Algorithm	CNG	libsodium	Advice/notes
PBKDF2(SHA1)	X		Ignores *piMemLimit*. Use an opslimit > 100.000!
PBKDF2(SHA256)	X		Ignores *piMemLimit*. Use an opslimit > 100.000!
scrypt		X	
argon2i		X	
argon2id		X	Use if possible. Best resistance against numerous forms of attacks.

Encryption algorithms

Algorithm	CNG	libsodium	Advice/notes
AES-ECB-128	X		Do not use.
AES-ECB-192	X		Do not use.
AES-ECB-256	X		Do not use.
AES-CBC-128	X		
AES-CBC-192	X		
AES-CBC-256	X		
AES-GCM-128	X		Authenticated Encryption (AE)
AES-GCM-192	X		Authenticated Encryption (AE)
AES-GCM-256	X		Authenticated Encryption (AE)