

# Yubikey Server COM API

---

YubiKey device server-side interface component

**Version: 1.0**

**Date: 9th June 2010**

## **Disclaimer**

The contents of this document are subject to revision without notice due to continued progress in methodology, design, and manufacturing. Yubico shall have no liability for any error or damages of any kind resulting from the use of this document.

The Yubico Software referenced in this document is licensed to you under the terms and conditions accompanying the software or as otherwise agreed between you or the company that you are representing.

## **Trademarks**

Yubico and YubiKey are trademarks of Yubico AB.

## **Contact Information**

Yubico AB  
Mäster Samuelsgatan 60, 8tr  
111 21 STOCKHOLM  
Sweden  
[info@yubico.com](mailto:info@yubico.com)

# Contents

<b>1</b>	<b>Document Information</b>	<b>4</b>
1.1	Purpose	4
1.2	Audience	4
1.3	Related documentation	4
1.4	Document History	4
1.5	Definitions	4
<b>2</b>	<b>Introduction</b>	<b>5</b>
2.1	Yubikey Server API features	5
2.2	Programming model	5
<b>3</b>	<b>Using the API</b>	<b>6</b>
3.1	Data representation and data exchange	7
3.2	Yubico OTP API – IYubicoOTP interface	8
3.3	OATH-HOTP API – IHmacSha1 interface	11
3.4	HMAC-SHA1 API – IHmacSha1 interface	13
3.5	Random number API – IRandomGen interface	14
<b>4</b>	<b>Test container tutorial</b>	<b>15</b>
4.1	Data buffer and encoding (IYubicoOTP)	16
4.2	Decoding Yubikey OTP output (IYubicoOTP)	16
4.3	Decoding Yubico OTP binary response (IYubicoOTP)	17
4.4	Creating HMAC-SHA1 (IHmacSha1)	17
4.5	Generating a OATH-HOTP (IHmacSha1)	17
4.6	Verifying a OATH-HOTP (IHmacSha1)	18
4.7	Decoding and verifying a HOTP with Token Identifier (IHmacSha1)	18
4.8	Creating random string (IRandomGen)	18

# 1 Document Information

---

## 1.1 Purpose

The purpose of the server interface component is to allow easy implementation of Yubikey functionality into server-side applications. The component provides a toolbox with functions to support Yubikey OTP, OATH-HOTP and HMAC-SHA1 implementations.

The component is not intended as a “stand-alone” utility kit and the provided sample code is provided as boilerplate code only.

## 1.2 Audience

Programmers and systems integrators.

## 1.3 Related documentation

- *The YubiKey Manual* – Usage, configuration and introduction of basic concepts
- *YubiKey Configuration Utility* – The Configuration Tool for the YubiKey
- *Yubikey Configuration API* – Yubikey configuration COM API
- *Yubikey Client API* – Client-side support COM API
- Yubico online forum – <http://forum.yubico.com>

## 1.4 Document History

Date	Version	Author	Activity
2007-07-10	1.0	JE	First draft

## 1.5 Definitions

Term	Definition
YubiKey device	Yubico’s authentication device for connection to the USB port
USB	Universal Serial Bus
API	Application Programmer Interface
COM	Component Object Model – a component based programming model developed by Microsoft.
ActiveX	A definition on top of COM, primarily targeted for user interface extensions in a Web-browser.
Callback	Function in the User Program called by the API
AES	Advanced Encryption Standard. A NIST approved symmetric encryption algorithm.
OATH-HOTP	Initiative for Open Authentication (RFC 4226)
HMAC	Hash-based Message Authentication Code
SHA-1	Secure Hash Algorithm 1

## 2 Introduction

---

In order to support integration of Yubikey functionality on the server-side in a client-server setting, certain functions are required for validating received OTP, either when using Yubico OTP mode or OATH-HOTP mode. With the introduction of challenge-response functionality in Yubikey firmware 2.2, additional server-side functionality is required to issue a challenge and decode the response.

To greatly simplify application development, Yubico provides a high-level server-side support component based on Microsoft's COM/ActiveX technology. With this approach, a wide range of programming languages, scripting environments and software packages can perform necessary server-side cryptographic validation operations using unified interfaces.

This document assumes knowledge about the YubiKey, its functions and intended usage as well as basic challenge-response concepts.

The document further assumes working knowledge of COM, and at least one programming language that supports COM components. Provided examples are developed with Microsoft Visual Studio .NET 2008.

The component is designed for the Microsoft Windows Win32 environment and works with Windows versions from Windows 2000 and onwards. Integration with Microsoft's .NET programming model is straightforward. Refer to appropriate .NET documentation of integration of COM components.

### 2.1 Yubikey Server API features

The Yubikey Server API implements the following functional blocks:

- **Yubico OTP decoding and validation**  
Decoding and decryption of Yubico OTPs
- **OATH-HOTP generation and validation**  
Generation and validation of OATH-HOTPs
- **HMAC-SHA1 generation**  
Generation of HMAC-SHA1
- **Random number generation**  
Generation of cryptographically secure random numbers
- **Data format conversion**  
Conversion between different data formats

### 2.2 Programming model

By using COM/ActiveX, most programming languages and third-party tools can interface to the Yubikey via the YubiServerAPI Component through uniform interfaces with standard data representation. In other words, the component can be used by any programming language and development tool supporting COM/ActiveX. Examples include Visual C++, Visual Basic, Delphi, Microsoft Office (VBA) and Internet Explorer VB Script.

A COM programming tutorial is beyond the scope of this document, but application samples are provided for VB.NET and Visual C++/MFC.

## 3 Using the API

---

The YubiKey configuration API is provided as a COM/ActiveX component, where methods and properties are exposed by three different interfaces:

- **YubiServerAPI.IYubicoOTP**  
Provides Yubico OTP functionality
- **YubiServerAPI.IHmacSha1**  
Provides OATH-HOTP and HMAC-SHA1 functionality
- **YubiServerAPI.IRandomGen**  
Provides random number generation functionality

The component follows the data types defined by the COM Automation model which provides maximum flexibility and interoperability.

A COM component needs to be registered with the operating system in order to be used. This is typically done by an installation tool, where the Self registration function is used. The component can be explicitly registered with the regsvr32 utility: Type **regsvr32 YubiServerAPI.dll** under the Start/Run menu.

Integration of COM components varies between different tools and languages, but the following steps describe the typical workflow of using the API of the YubiServerAPI Component:

1. **Provide a reference to the component and the desired interface**  
The development tool needs access to the YubiServerAPI component's Type Library, which contains the interface descriptions. The Type Library is embedded in the component itself; there is no separate .tlb file.
2. **Instantiate the component's particular interface**  
The instantiation phase gives a "handle" to the selected YubiServerAPI interface in the component.

The following examples use a "pseudo-code" notation, omitting the COM object reference and its instantiation for the sake of clarity. Function prototypes are shown in VB notation such as

```
Property myFunction(myParameter As parameterType) As returnType
```

### 3.1 Data representation and data exchange

To support simple scripting languages and to allow maximum flexibility for each particular application, input and output data is passed as COM/Automation VARIANTS. These VARIANTS can be configured to hold strings (BSTRs), unsigned integers or byte arrays.

#### Property `dataBuffer` As `VARIANT`

A global property is provided to set the appropriate data encoding

#### Property `dataEncoding` As `ysENCODING`

The following data encoding formats are available for the VARIANTS used:

- Hexadecimal string – `ysENCODING_BSTR_HEX`  
Lower-case hexadecimal digits without spacing, e.g. `6b6c3132`
- Hexadecimal string with spaces – `ysENCODING_BSTR_HEX_SPACE`  
Lower-case hexadecimal digits with spacing, e.g. `6b 6c 31 32`
- Modhex string – `ysENCODING_BSTR_MODHEX`  
Yubico Modhex format, e.g. `hnhrebed`
- Base64 string – `ysENCODING_BSTR_BASE64`  
Base64 encoded string, e.g. `a2wxMg==`
- ASCII string – `ysENCODING_BSTR_ASCII`  
ASCII (MBCS / non-Unicode) encoded string, e.g. `k112`
- Unsigned 16-bit integer – `ysENCODING_UINT16`  
16-bit USHORT/UINT16 integer = 2 bytes
- Unsigned 32-bit integer – `ysENCODING_UINT32`  
32-bit ULONG/UINT32 integer = 4 bytes
- Byte array – `ysENCODING_BYTE_ARRAY`  
SAFEARRAY of bytes (UINT8/UI1) with dynamic length
- NULL / Nothing  
Represents an empty string holding zero bytes

Input and output data is handled by the means of a global data buffer which can be set or read at any time. This model further allows data conversion “on the fly”, such as the following pseudo-code

```
dataBuffer = 0x4711          -- Hexadecimal 4711
dataEncoding = ysENCODING_BSTR_MODHEX
print dataBuffer            -- prints fibb
dataEncoding = ysENCODING_BSTR_HEX_SPACE
print dataBuffer            -- prints 47 11
dataEncoding = ysENCODING_UINT32
print dataBuffer            -- prints 0x00004711
```

Integers (16- and 32 bits) are handled as Big Endian (high byte first), just as they appear in a byte string.

### 3.2 Yubico OTP API – IYubicoOTP interface

Decoding a Yubico OTP (with public ID) typically comprise the following steps:

1. Convert the Modhex string to binary bytes
2. Extract the public ID part (and customer prefix if applicable)
3. Retrieve the AES key for this public ID from a database
4. Decrypt the OTP part using this AES key
5. Verify the embedded OTP checksum
6. Verify the private ID
7. Verify the counters with the previous value stored in the database
8. Verify the timer delta (if applicable)

The IYubicoOtp interface provides two functions for (a) parsing the OTP string and (b) decoding it using a supplied AES key:

```
Property otpParse(VARIANT otpString) As ysRETICODE
Property otpDecode(VARIANT aesKey) As ysRETICODE
```

Using the IYubicoOTP interface simplifies this overall procedure into the following pseudo-code example:

```
dataEncoding = ysENCODING_BSTR_MODHEX
returnCode = otpParse(myModhexOtpString)
if returnCode = ysRETICODE_OK Then
  print publicId
  -- Get AES key for publicId -> myAesKey
  returnCode = otpDecode(myAesKey)
  if returnCode = ysRETICODE_OK Then
    if privateId(Nothing) = myPrivateId Then
      if longCounter > myLastCounter Then
        myLastCounter = longCounter
        print "OTP validated successfully"
      Else
        print "Replayed OTP"
      Endif
    Else
      print "Invalid private ID"
    Endif
  Else
    print "Could not decode OTP"
  End If
Else
  print "Could not parse OTP"
End If
```

In the case where a Yubico OTP has been received in challenge-response mode, the first decoding step is not used. In this case, store the received OTP in the `dataBuffer` prior to calling `otpDecode`

```
dataBuffer = myOtpResponse
returnCode = otpDecode(myAesKey)
if returnCode = ysRETCODE_OK Then
    if privateId(myStoredPrivateId) = mySentChallenge Then
        print "Successfully authenticated"
    Else
        print "Response does not match challenge"
    End if
Else
    print "Could not decode OTP"
End If
```

Once the OTP has been successfully parsed and decoded, the application-level validation steps could be performed by using the following helper functions/properties:

Properties valid after a successful `otpParse` call:

**Property publicId As Object**

Returns the decoded public identity from a parsed OTP string

**Property customerPrefix As UShort**

Returns the decoded 16-bit customer prefix from a parsed OTP string

**Property deviceIdentity As UInteger**

Returns the decoded 20-bit device identity from a parsed OTP string

**Property dataBuffer As Object**

Holds the OTP part of the decoded OTP string

Properties valid after a successful `otpDecode` call: (assuming a valid OTP being stored in `dataBuffer` prior to the call)

**Property `privateId(Mask as Object) As Object`**

Returns the private identity from a decoded OTP. Optionally, an XOR mask with the pre-set UID string can be specified when using challenge-response mode. Set the Mask parameter to NULL/Nothing otherwise

**Property `useCounter As Ushort`**

Returns the `useCounter` field from a decoded OTP

**Property `sessionCounter As Ushort`**

Returns the `useCounter` field from a decoded OTP

**Property `longCounter As UInteger`**

Returns the `useCounter` and `sessionCounters` as a single sequential number, ready for compare operations and to be stored as "last counter" value.

**Property `timestamp As UInteger`**

Returns the `timestamp` field from a decoded OTP. This value can optionally be used to determine the delta time between two sequentially received OTPs.

**Property `millisecondTimer As UInteger`**

Returns a free-running millisecond timer from the operating system. This value can be used to verify the `timestamp` delta value between two sequentially received OTPs.

### 3.3 OATH-HOTP API – IHmacSha1 interface

The IHmacSha1 interface provides methods for OATH-HOTP generation and verification, using a fixed 20 byte (160 bit) secret.

HOTPs can either be sent in plain form as 6- or 8 digits numeric values or together with a Token Identifier according to the openauthentication.org token. The parsing routine can the decode the full OTP string and split it into a token identifier with OMP, TT and MUI fields and the HOTP.

The IHmacSha1 interface provides two functions for (a) parsing the HOTP string (eventually including a token identifier) and (b) decoding it using a supplied secret:

```
Property hotpParse(String otpString) As ysRETCODE
```

The `hotpParse` breaks up a OTP string into a `tokenIdIdentifier` (if applicable) and a HOTP value and its derived `customerPrefix` and `deviceIdentity` parts.

```
Property tokenIdIdentifier as String (read only)
```

```
Property customerPrefix as UShort (read only)
```

```
Property deviceIdentity as UInteger (read only)
```

The parsed HOTP value is stored in the `dataBuffer` property as Ascii string (`ysENCODING_BSTR_ASCII`). The `hotpDigits` property is updated accordingly with 6 or 8 digits.

Prior to verifying a HOTP, the secret must be set using the `hmacSecret` property

```
Property hmacSecret As Object
```

Once the `hmacSecret` and HOTP value is set, either explicitly by assigning `dataBuffer` with the HOTP value or implicitly by a prior call to the `hotpParse` function, the HOTP can be verified.

A start value for the moving factor and a maximum number of values to search ahead needs to be specified.

```
Property
```

```
hotpVerify(startSearch as UInteger, maxSearch as UShort)  
As Integer
```

The function returns -1 if the HOTP was not found within the specified search range. A value  $\geq 0$  means that the HOTP was found.

Using the IHmacSha1 interface simplifies this overall procedure into the following pseudo-code example:

```
returnCode = hotpParse(myHotpString)
if returnCode = ysRETCODE_OK Then
  print tokenId
  -- Get secret for tokenId -> mySecret
  hmacSecret = mySecret
  movingFactor = hotpVerify(0, 100) -- Window of 100 HOTPs
  if MovingFactor >= 0 Then
    print MovingFactor -- Success
  Else
    print "Could not decode HOTP"
  End If
Else
  print "Could not parse HOTP/token identifier"
End If
```

### 3.4 HMAC-SHA1 API – IHmacSha1 interface

The IHmacSha1 interface provides methods for HMAC-SHA1 generation using a fixed 20 byte (160 bit) secret.

Generation of a 20 byte (160 bits) HMAC-SHA1 value is straight-forward. First, set the secret by assigning the `hmacSecret` property:

**Property hmacSecret As Object**

When set, set the HMAC input data (up to 64 bytes) via the `dataBuffer` property:

**Property dataBuffer As Object**

When the input data is set, call the `hmacSha1` method

**Sub hmacSha1 ()**

This calculates the HMAC-SHA1 value and stores the result into the `dataBuffer` property

Using the IHmacSha1 interface to generate a HMAC-SHA1 could be done as shown by the following pseudo-code example:

```
hmacSecret = myHmacSecret  
dataBuffer = myDataBuffer  
hmacSha1 ()
```

### 3.5 Random number API – IRandomGen interface

A simple method of generating cryptographically secure random numbers is provided with the IRandomGen interface. The function relies on the Win32 CryptGenRandom function to generate the random number.

Simply set the **dataEncoding** property to the desired output format and then call the **randomGen** method, specifying the desired number of bytes requested.

```
Sub randomGen(numDigits As UShort)
```

Then, read the **dataBuffer** property to get the generated random number.

Using the IRandomGen interface to generate a random value could be done as shown by the following pseudo-code example:

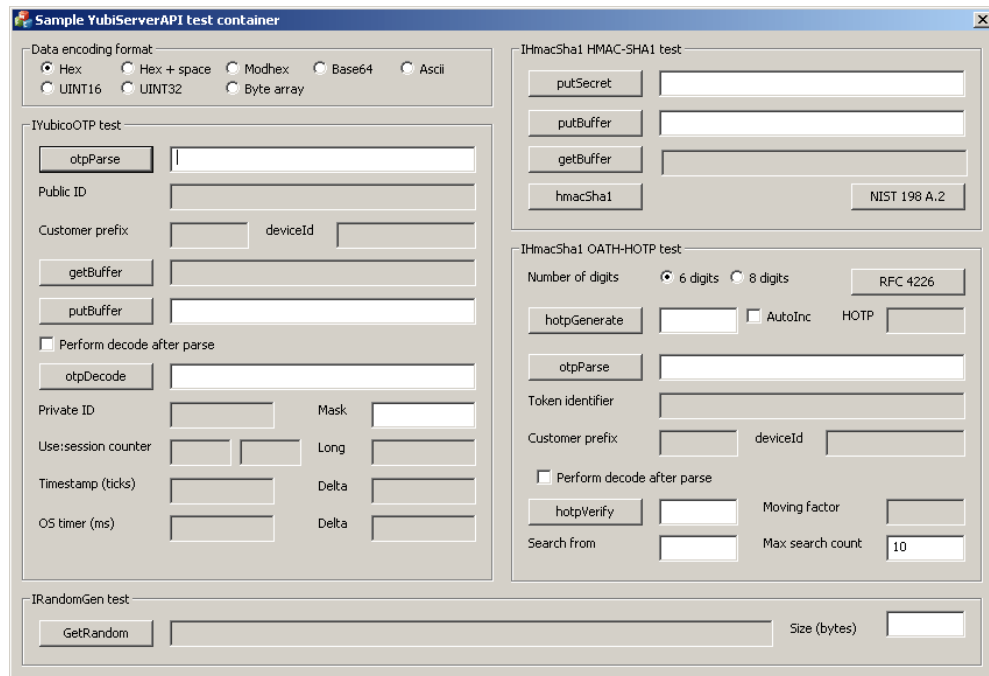
```
dataEncoding = ysENCODING_HEX  
randomGen(6)  
print dataBuffer - The random number buffer is displayed
```

## 4 Test container tutorial

The MFC test container can be used to test the Yubikey Server API functionality and to understand the concepts.

The MFC/VC++ source code is provided as a “boilerplate” template that can be used for test and/or further application development. Microsoft Visual Studio 2008 or later is required to build the test application. A reference to the YubiServerAPI.dll COM component is done at the #import statement in the MFCTestDlg.h file. Change the project search path to the target location if necessary.

Start the MFC test container executable



## 4.1 Data buffer and encoding (IYubicoOTP)

Enter a hexadecimal string in the putBuffer text field. Push getBuffer and the same string will appear in the getBuffer read-only text field. The returned data type is appended by the application for debug purposes.

1. Click Ascii in the Data encoding format box
2. Enter an Ascii string such as ABC123 in the PutBuffer field and push PutBuffer
3. Click Hex in the Data encoding format box
4. Click GetBuffer and the hexadecimal representation 414243313233 appears together with the VARIANT return type VT\_BSTR

This functionality is used to exchange data in the challenge-response transactions but can also be used for simple conversion between different string formats and data types.

Set data:

1. Set dataEncoding to the appropriate ysENCODING\_xxx type
2. Set dataBuffer to the desired data

Get data

1. Set dataEncoding to the appropriate ysENCODING\_xxx type
2. Get the dataBuffer holding data in the selected format

The same functionality is implemented in all three interfaces

## 4.2 Decoding Yubikey OTP output (IYubicoOTP)

The normal setting is a Modhex encoded Yubico OTP in text format being received as output from a Yubikey.

1. Place the cursor in the otpParse field
2. Insert a Yubikey (with known AES key) and push the Yubikey button
3. If the Yubikey sends out an ENTER keystroke, the otpParse button is automatically pushed. Otherwise push it manually
4. The decoded public id and derived customer prefix and device identities are filled in. The decoded OTP part has been stored in the dataBuffer
5. Now decode the string by entering the AES key in the otpDecode field and press otpDecode
6. All decoded fields are filled in

### 4.3 Decoding Yubico OTP binary response (IYubicoOTP)

In challenge-response mode a raw binary buffer is used as input rather than a text OTP. The decoding stage is therefore omitted and an additional response verification step is added

1. Enter received binary response in the putBuffer field and push putBuffer
2. Enter the known private id in the mask field
3. Enter the AES key in the otpDecode field and push otpDecode
4. The decoded fields are filled in

### 4.4 Creating HMAC-SHA1 (IHmacSha1)

The HMAC-SHA1 function can be used in HMAC-SHA1 challenge-response authentication.

1. Enter the HMAC-SHA1 secret (20 bytes fixed value) and push putSecret
2. Enter the input data (up to 64 bytes) in the putBuffer field and push putBuffer
3. Push hmacSha1
4. The HMAC-SHA1 is shown in the getBuffer field

Pushing NIST 198 A.2 enters a test vector from the NIST PUB 198 where the output is 0922d3405faa3d194f82a45830737d5cc6c75d24

### 4.5 Generating a OATH-HOTP (IHmacSha1)

As the OATH-HOTP algorithm is based on HMAC-SHA1, the OATH-HOTP functions are present in this interface.

1. Enter the OATH-HOTP in the putSecret field and push putSecret
2. Check the desired number of output digits, 6 or 8
3. Enter the moving factor in the hotpGenerate field and push hotpGenerate
4. The HOTP is shown in the HOTP field

Checking AutoInc causes the moving factor to be automatically incremented each time hotpGenerate is pushed

Pushing RFC4226 enters the test vector from the RFC 4226 specification

## 4.6 Verifying a OATH-HOTP (IHmacSha1)

The OATH-HOTP decoding/verification allows a window for moving factor search can be specified

1. Enter the OATH-HOTP in the putSecret field and push putSecret
2. Enter the moving factor search start value
3. Enter the maximum number of iterations that shall be made in the search ahead
4. Enter the OATH-HOTP value in the HOTP field and push hotpVerify
5. The found moving factor is shown in the moving factor field

## 4.7 Decoding and verifying a HOTP with Token Identifier (IHmacSha1)

The hotpParse function can be used to decode a full OTP including a Token Identifier.

1. Place the cursor in the otpParse field
2. Insert a Yubikey (with known secret) and push the Yubikey button
3. If the Yubikey sends out an ENTER keystroke, the otpParse button is automatically pushed. Otherwise push it manually
4. The decoded token identifier and derived customer prefix and device identities are filled in. The decoded HOTP part has been stored in the dataBuffer
5. Now decode the HOTP by entering the secret in the putSecret field and press putSecret
6. Push hotpVerify
7. The found moving factor is shown in the moving factor field

## 4.8 Creating random string (IRandomGen)

The random generator can be used to create challenges in a challenge-response setting.

1. Set the data encoding by checking the desired radio button
2. Enter desired number of bytes in result
3. Push GetRandom
4. The random output is shown in the GetRandom field