General page information

aspEasyCrypt

Last modified : 03/02/2004 (Version 2.0)

What's aspEasyCrypt

This is an ASP component for use with the IIS Server from Microsoft, the main future is that can hash, cipher and make checksum on strings and on files. It supports 22 types of hash, 5 checksums and 37 different ciphers.
With this component you can make strong security pages, sign page or messages, make the programs to exchange keys, making public and private keys, you have all the standards to cipher and hash it.
See the documentation and some examples on how to do it.

This component is based on the DEC component by Hagen Reddmann that is also freeware, that's why this component is freeware, you only have to pay if you want to use it on a commercial site, this will remove the printed label on the html. Enjoy it!

This component is **freeware** , the only limitation is when hashing or ciphering it will display a message on the html page, if you want to remove this message then you will have to pay 50 US$. I think is a small limitation because everyone can use it with no limitation of time and functionality.

For additional information contact us at: support@mitdata.comor visit our web site at: www.mitdata.com

Prices & Conditions

All products when registered have the following features:

- Free email support for all of our products
- Minor, Major upgrades and bug fixes
- Notification by email when newer versions are released

Prices and conditions

To order one product, click on the price.

Product	**Code to Order it**	**Price per/ Server**	**Site License**	Full Mail Support	Free Updates for minor releases	Free Updates for major releases	Special Prices for major releases	Bug fixes e-mail notifications
aspEasyCrypt	33561	$ USD 20	$ 100 USD	**Yes**	**Yes**	**No**	**Yes**	**Yes**

N/A = Not applicable

Email usfor special conditions and quantities prices.
The site license enables you to install it on all servers / machines from your company with out any restrictions or develop your own selling software with our component included

History

aspEasyCrypt

Last modified : 03/02/2004

History

2.0 (04/February/2004)
- Some code has been optimized
- Encoding string was not working fine when calling it alone
- Added two new functions for calculating, management of standard checksums files in md5 and sfv format

1.21 (23/12/2002)
- Formats where not working
- Added the debug property information

1.2 (17/09/2002)
- Cyphers were getting wrong ID for using from
- New installation program to make it more easy to install and using it, even registering is more easy with aspEasyREG program.

1.13 (08/02/2001)
- Revised and added the possibility to use it on the client side enhancing the security.

1.0 (01/08/2000)
- First version

Encryption Technology

This document is intended as a quick primer on cryptographic technologies of potential interest to activists. As communities of resistance become stronger and more effective the agents of repression become more determined and draconian in their methods to maintain the status quo. Central to these efforts is intelligence gathering (they need to gather as much as they can since they have so little of their own) This is often accomplished by ease dropping on private communications. To thwart these Orwilli an methods consider using the spooks' own tools against them. Well implemented cryptography is the silver bullet of privacy North American governments have long recognized this and enacted laws to restrict strong crypto, but with the rise of the internet Strong crypto has become as readily available.

While many different algorithm and implementations exist, the ones presented here are considered standards because they have been extensively reviewed by the internet community worldwide. Their strengths and weaknesses have been documented. Other algorithms and implementations from indeterminate sources should be avoided since they may contain compromised or flawed code which could result in the unauthorized access to your data. The most notorious of these, SKIPJACK, was authored by the NSA with an int entional backdoor and marketed as secure crypto.

- Relative Strength Comparisons of Encryption Algorithms -			
Type	**Security Level***	**Implementation**	**Speed**
Idea	Military Grade	128 bit Shared Secret	Fast
Blowfish	Military Grade	256 to 448 bit Shared Secret	Fastest
DES	Low	40 to 56 bit Shared Secret	Fast
RSA	Military Grade	2048 bit Public Key	Very Slow
MD5	High	128 bit Message Digest	Slow
SHA	High	160 bit Message Digest	Slow
*** Depends on the length key used, evaluated at the current implementations max key length**			

Types of Encryption Algorithms

Shared Secret / Symmetric

This is the classical type of encryption. The same password is used to encrypt the message as is used to decrypt the message. While many different algorithms exist the ones presented below are considered standards because they have been extensively rev iewed by the internet community worldwide and their strengths/weaknesses are well documented. It should be noted that current US law forbids the export of (symmetric) ciphers utilizing keys in excess of 56 bits.

IDEA-
Swiss Algorithm very strong it is the backbone to PGP very fast up to 128 bit key
Advantage: Tested algorithm, high security, fast.

Blowfish-
Relatively new 64bit block cipher Invented by cryptographer Bruce Schneier current implementations are available to 448bit keys It is several orders of magnitude stronger than DES, .

RC4-
Fast stream cipher, it is the crypto behind SSL and is considered secure in the 128 bit implementations. The "export grade" version is worthless.

DES-
Industry Standard fast but is relatively insecure. It is the most widely used encryption algorithm in the world. Generally it should be avoided when possible.

Public Key / Asymmetric

These are a relatively recent class of algorithms discovered in the late 70's.. They are constructed of a two part key. One key is used to encrypt the data while the other is key is used to decrypt the data. Given one key and an encrypted message it is mathematically very difficult to determine the other key.

RSA-
Public Key encryption relying on the fact that as numbers get geometrically larger the time it takes to factor them increases exponentially . It is the public key scheme in PGP.

One Way Digests

These are hashing algorithms that translate an indefinite length of data into a fixed length unique hash. They are useful in digitally sealing a message since it is next to impossible that two different messages can have the same "fingerprint" (digest) hence given a message fingerprint you can be sure of the message integrity

MD5 128 bit message digest used to ensure that messages haven't been altered

SHA 160 bit message digest used to ensure that messages haven't been altered

How Secure is secure?

This question is often a raised by people.

There are several ways to break encryption. The most straight forward is to key space the algorithm, that is guess every possible combination. A well engineered algorithm (such as those presented here) should be relatively immune to the currently known forms crypto analysis.

DES:
The Data Encryption Standard typically comes in two strengths 40 bit and 56 bit. The 40 bit variety (known as secure socket layer) is the same encryption that web browsers use to protect credit card orders over the internet. 40 bit strength means there are 2^40 possible keys. On average 2^39 keys must be tried before a correct match will be found. Computer chips currently exist for about $10 US that are capable of testing 200 million DES keys/second. Such a card would be capable of trying half the possible keys in 5 hours. If the forces of evil were to spend $300,000 US to build a purpose built computer they could recover a 40 bit key in 0.07 seconds. Ten million dollars would get it to them in 0.005 seconds. (This is twice as fast as a computer can write to its hard drive) 40 bit DES offers no protection from large corporations or governments, and 56 bit provides only slightly more protection.

IDEA:
Using a similar technique as outlined above on a 128 bit IDEA key it would take approximately 2.2 x 10^24 seconds. If the planet earth is assumed to be 3 billion years old, it would take over 20 thousand times longer than the planet has been in existence to key space a 128 IDEA key.

This could be thought of differently. If the amount of energy required to change a 0 to 1 (remember that all computers are binary adding machine at the chip level) is XXXX to key space a 128 bit IDEA key on average would require that 2^127 bits be flipped. If we assume a super efficient computer with almost no thermal loss (<-currently this is very unrealistic) it would take XXXX joules of energy to keyspace a 128 bit key. By comparison when a star the size of the sun goes super nova it is estimated to produce XXXX joules of energy. Not even the NSA (the FBI evil big brother) could afford that electric bill.

RSA:
Recent advances in factoring theory have significantly reduced the time required to factor large numbers, however with current technology to factor a 2048 bit key would take approximately 4 x 10^14 mips-years. That is a computer capable of one million instructions per second would have to work for 10^14 years to crack the key. In general if your computer can handle it, in RSA bigger is better and much bigger is much better.

From these examples it should be evident that brute forcing "strong encryption" is impractical. When faced with strong encryption the forces of darkness must resort to other methods. These fall into two categories: crypto analysis (guessing the message content based on relative frequency of letters in a language) and password attacks. Most well conceived algorithms (such as IDEA and Blowfish) are relatively immune to crypto analysis. Password attacks are a function of the user. The strongest encryption w ill not protect against a poorly chosen password!

Passwords should be chosen that make it harder to brute force the password than brute forcing the algorithm.

Since there are 94 possible characters (not including alt-shift characters) it would require 20 COMPLETELY RANDOM characters to create a password that was as strong as 128 cipher. Password attacks often amount to trying a specially crafted wordlist to see if any of the words match your password. If you use good passphrase hygiene they won't. Remember your freedom and the freedom of others communicating with you may depend on it.

Resources:

cryptome.org

Algorithm

Set the algorithm you want to use, if you specify a cipher algorithm then you must use the cipher methods.
Those are the supported algorithm, from 0 to 63 different hashes, from 64 to 127 different checksums and from 128 up to 256 there are different ciphers. Many will be supported on to future, see the following list of supported algorithm:

List of supported algorithm

Syntax

Crypt .Algorithm = OptionValues

Parameter

(Read / Write) Integer

Example

```
<%
  const csHash_MD5 = 1
  const fmtHEX = 1
  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
 Crypt.Algorithm = csHash_MD5
 'Hash a string
 Crypt.Hash "Hello world", 0
  response.Write "Result HEX: " + Crypt.Digest( fmtHEX ) +
"<br>"
  set Crypt = nothing
%>
```

Description

Returns the Algorithm selected plus a small description of it, like legal purpose or if it's free to use the algorithm selected.

Syntax
Desc = Crypt .Description

Parameter
(Read) String

Example

```
<%
  set Crypt =
server.createobject("aspCrypt.EasyCRYPT")
 Crypt.Algorithm = csHash_MD5
 response.Write "Hash to use " + Crypt.Description +
"<br>"
  set Crypt = nothing
%>
```

Mode

You specifies the mode you want to use for ciphering:

cmCTS : Cipher Text Stealing, a Variant from cmCBC, but relaxes the restriction that the DataSize must be a mulitply from BufSize, this is the Defaultmode , fast and Bytewise
cmCBC : Cipher Block Chaining
cmCFB : K-bit Cipher Feedback, here is K = 8 -> 1 Byte
cmOFB : K-bit Output Feedback, here is K = 8 -> 1 Byte
cmECB : * Electronic Codebook
cmCTSMAC : Build a Message Authentication Code in cmCTS Mode
cmCBCMAC : Build a CBC-MAC
cmCFBMAC : Build a CFB-MAC

Syntax
Crypt .Mode = Mode

Parameter
(Read / Write) Integer

NVersion

Returns the version number of the component, you can check it to provide compatibility between different versions of the component.

Syntax
Ver = Crypt .NVersion

Parameter
(Read) Integer

Returning values
100 - this is version 1.0
112 - for version 1.12
120 - for version 1.20
Example

```
<%
  if Crypt.NVersion < 112 then
    response.write "This version is not supported by our source code get a newer
one"
 end if
%>
```

Results

Returns the EnCodeString or the DeCodeString, next versions will include the hash returning string. You can also use the returned string when using those functions.

Syntax
Result = Crypt .Results

Parameter
(Read) String

Example

```
<%
  const csCipher_Blowfish= 129
  const cmCTS = 0
  const fmtNone = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
  'Cipher a String
 TestCipher = "This string will be Ciphered"
 TestCipherEnCoded = ""
 MyKey = "This is my secret key"
  response.Write "<b>Start a Cipher Test of '" + TestCipher + "'</b><br>"
 Crypt.Algorithm = csCipher_Blowfish
 Crypt.Mode = cmCTS
  response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 TestCipherEnCoded = Crypt.EnCodeString( TestCipher, fmtNone )
  response.Write "CODED: '" + TestCipherEnCoded + "'<br>"
  response.Write "CODED: '" + Crypt.Results + "'<br>"
  set Crypt = nothing
%>
```

Version

Returns de version of the component in verbose format.

Syntax
Version = Crypt .Version

Parameter
(Read) String

Example

```
<%
  set Crypt = server.createobject("aspCrypt.EasyCRYPT"")
  if not isObject (Crypt) then
    response.write "Easy Crypt is not installed.<br>"
  else
    response.write "Easy Crypt version installed: " & Crypt.Version &
"<br>"
  end if
  set Crypt = nothing
%>
```

Create

It initializes the creation of the component when using it on the client side.
When creating the component on JavaScript or Visual Basic without using ASP Server then you have to use this function to initializes all variables.

****NOTE***:** From version 2.0 there is no need to call this function, has been left just to be cross compatible with lower versions

CreateFileSum

Creates a checksum file that can be used later to check if the file has been altered. I commonly used on Internet when downloading big files to known if the download is correct.

It supports 2 standards, **md5** and **sfv** ;

The md5 is the best one because you get an high precession check but is quite slow and only accepts on entry per file, you will have x .md5 files for each file specified.
Parameter type is 0

The sfv format is not so efficient has it's on md5, but it's faster and on the checksums.sfv will have all file entries on just one file.
Parameter type is 1

Syntax
Crypt .CreateFileSum(Pattern, Type)

Example in VBS:

```
 const csFSUM_MD5 = 0
 const csFSUM_SFV = 1

 set Crypt = server.createobject("aspCrypt.EasyCRYPT")
' Create the .MD5 file of the files
crypt.CreateFileSum "*.exe", csFSUM_MD5
' Create the SVF file of the files
crypt.CreateFileSum "*.exe", csFSUM_SFV
 set Crypt = nothing
```

CheckFileSum

Checks a file checksum if it's correct or damaged.

It supports 2 standards, **md5** and **sfv** ;

The md5 is the best one because you get an high precession check but is quite slow and only accepts on entry per file, you will have x .md5 files for each file specified.
Parameter type is 0

The sfv format is not so efficient has it's on md5, but it's faster and on the checksums.sfv will have all file entries on just one file.
Parameter type is 1

ErrorCodes:

0	Checksum failed
1	Checksum correct
99	Unknown
100	File not found
101	Checksum File not found
102	Checksum entry not found
103	Unknown checksum type

Syntax
ErrorCode = *Crypt* .CheckFileSum(ChecksumFile, Filename)

Example in VBS:

```
  set Crypt = server.createobject(" aspCrypt.EasyCRYPT ")
  ' Check the .MD5 checksum
  MsgBox " Result: " & crypt.CheckFileSum(" setupcrypt.md5 ", "
setupcrypt.exe ")
 ' Check the SVF checksum
  MsgBox " Result: " & crypt.CheckFileSum(" checksums.sfv ", "
setupcrypt.exe ")
  set Crypt = nothing
```

Counter

Only for testing purpose but can be useful for others users so I let it in. If you pass -1 to the size then you reset the counter and starts to count until you recall the function with 0 or another quantity then you get in the return the time enlapsed.

```
Crypt.Counter( -1 )
...
  crypt, encrypt, hash it ...
...
response.Write "Time enlapsed: " & Crypt.Counter( 0 )
```

If you pass the size of the file that you have processed then you will get more statistical information, like size per second worked.

Does not print the license information on the freeware version

DecodeFile

This copies the source file to the target one, can't be the same, decoding with the algorithm specified and with the initkey as a password. This makes the inverse process to the encode function.

Syntax

Crypt .DecodeFile(SourceFile, TargetFile)

Example

```
<%
  const csCipher_IDEA = 131
  const cmCTS = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
 'Cipher a File
 MyKey = "This is my secret key"
 Crypt.Algorithm = csCipher_IDEA
 Crypt.Mode = cmCTS
 response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 Crypt.EnCodeFile "c:\autoexec.bat", "c:\autoexec.enc"
 Crypt.initKey( MyKey )
 Crypt.DeCodeFile "c:\autoexec.enc", "c:\autoexec.dec"
  set Crypt = nothing
%>
```

DecodeString

Returns the decoded string for the text passed on the function with the format specified, see the different formats to return.
(Only fmtNone is supported)

Syntax
Result = *Crypt* .DecodeString(StringtoDecode, format)

Example

```
<%
  const csCipher_Blowfish = 129
  const cmCTS = 0
  const fmtNone = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
  'Cipher a String
 TestCipher = "This string will be Ciphered"
 TestCipherEnCoded = ""
 MyKey = "This is my secret key"
  response.Write "<b>Start a Cipher Test of '" + TestCipher + "'</b><br>"
 Crypt.Algorithm = csCipher_Blowfish
 Crypt.Mode = cmCTS
  response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 TestCipherEnCoded = Crypt.EnCodeString( TestCipher, fmtNone )
  response.Write "CODED: '" + TestCipherEnCoded + "'<br>"
  response.Write "CODED: '" + Crypt.Results + "'<br>"
 Crypt.initKey( MyKey )
  response.Write "DECODED: '" + Crypt.DeCodeString( TestCipherEnCoded, fmtNone ) +
"'<br>'"
  response.Write "DECODED: '" + Crypt.Results + "'<br>"

  set Crypt = nothing
%>
```

Digist

With this function you will get the results of the hash function, you can specify a format for the output:

```
fmtNone = -1  ' With out conversion
fmtCopy = 0  ' As it
fmtHEX = 1 ' In hexadecimal
fmtMIME64 = 2 ' in Mime64 format
fmtUU = 3 ' in UU Code
fmtXX = 4 ' in XX code
```

Syntax

Result = *Crypt* .Digist(Format)

Example

```
<%
  const csHash_MD5 = 1
  const fmtNone = -1  ' With out conversion
  const fmtCopy = 0  ' As it
  const fmtHEX = 1 ' In hexadecimal
  const fmtMIME64 = 2 ' in Mime64 format
  const fmtUU = 3 ' in UU Code
  const fmtXX = 4 ' in XX code

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
 Crypt.Algorithm = csHash_MD5
 'Hash a string
 Crypt.Hash "Hello world", 0
  response.Write "Result Copy: " + Crypt.Digest( fmtCopy ) + "<br>"
  response.Write "Result HEX: " + Crypt.Digest( fmtHEX ) + "<br>"
  response.Write "Result Mime64: " + Crypt.Digest( fmtMIME64 ) + "<br>"
  response.Write "Result EncodeUU: " + Crypt.Digest( fmtUU ) + "<br>"
  response.Write "Result EncodeXX: " + Crypt.Digest( fmtXX ) + "<br>"
  set Crypt = nothing
%>
```

EncodeFile

This copies the source file to the target one encoding with the algorithm specified and with the initkey as a password.
Note: Source and target can not be the same name

Syntax

Crypt .EncodeFile(SourceFile, TargetFile)

Example

```
<%
  const csCipher_IDEA = 131
  const cmCTS = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
 'Cipher a File
 MyKey = "This is my secret key"
 Crypt.Algorithm = csCipher_IDEA
 Crypt.Mode = cmCTS
 response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 Crypt.EnCodeFile "c:\autoexec.bat", "c:\autoexec.enc"
 Crypt.initKey( MyKey )
 Crypt.DeCodeFile "c:\autoexec.enc", "c:\autoexec.dec"
  set Crypt = nothing
%>
```

EncodeString

Returns the encoded string for the text passed on the function with the format specified, see the different formats to return.
(Only fmtNone is supported, will add more on the future)

Syntax
Result = *Crypt* .EncodeString(Stringtoencode, format)

Example

```
<%
  const csCipher_Blowfish = 129
  const cmCTS = 0
  const fmtNone = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
  'Cipher a String
 TestCipher = "This string will be Ciphered"
 TestCipherEnCoded = ""
 MyKey = "This is my secret key"
  response.Write "<b>Start a Cipher Test of '" + TestCipher + "'</b><br>"
 Crypt.Algorithm = csCipher_Blowfish
 Crypt.Mode = cmCTS
  response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 TestCipherEnCoded = Crypt.EnCodeString( TestCipher, fmtNone )
  response.Write "CODED: '" + TestCipherEnCoded + "'<br>"
  response.Write "CODED: '" + Crypt.Results + "'<br>"
  set Crypt = nothing
%>
```

Hash

This functions will hash a string or a file, first you must have specified the algorithm to use.

```
Hash "Hi this string will be hashed", 0
```

If you pass on type the number 0 then it will assume that text is a string, if you pass 1 then it will assume it's a file and will load the file for hashing it.
If you want to see the returned hash then you will have to invoke the Digist function to retrieve the data, the hash only made calculation and store all data in digist function.

```
Hash "index.htm", 1
if Digist( fmtHEX ) <> "C3A110" then response.write "Someone has altered the main page without permissions, maybe cracked?"
```

Syntax
Crypt .Hash(File or String, HashType)

InitKey

Set the key to use for encoding or decoding, this function is only for cipher. Use this function to set the password to use before making any cipher function.

Note: When bigger is the key better protection you get

Syntax

Crypt .InitKey(PasswordKey)

Example

```
<%
  const csCipher_IDEA = 131
  const cmCTS = 0

  set Crypt = server.createobject("aspCrypt.EasyCRYPT")
 'Cipher a File
 MyKey = "This is my secret key"
 Crypt.Algorithm = csCipher_IDEA
 Crypt.Mode = cmCTS
 response.Write "Algorith Desc: " + Crypt.Description + "<br>"
 Crypt.initKey( MyKey )
 Crypt.EnCodeFile "c:\autoexec.bat", "c:\autoexec.enc"
 Crypt.initKey( MyKey )
 Crypt.DeCodeFile "c:\autoexec.enc", "c:\autoexec.dec"
  set Crypt = nothing
%>
```

Include file

```
<%
' INCLUDE FILE aspEasyCrypt ( www.mitdata.com )
Rem Hash constants
const csHash_MD4 = 0
const csHash_MD5 = 1
const csHash_SHA = 2
const csHash_SHA1 = 3
const csHash_RipeMD128 = 4
const csHash_RipeMD160 = 5
const csHash_RipeMD256 = 6
const csHash_RipeMD320 = 7
const csHash_Haval128 = 8
const csHash_Haval160 = 9
const csHash_Haval192 = 10
const csHash_Haval224 = 11
const csHash_Haval256 = 12
const csHash_Sapphire128 = 13
const csHash_Sapphire160 = 14
const csHash_Sapphire192 = 15
const csHash_Sapphire224 = 16
const csHash_Sapphire256 = 17
const csHash_Sapphire288 = 18
const csHash_Sapphire320 = 19
const csHash_Snefru = 20
const csHash_Square = 21
const csHash_Tiger = 22

R em Checksums
const csHash_XOR16 = 64
const csHash_XOR32 = 65
const csHash_CRC16_CCITT = 66
const csHash_CRC16_Standard = 67
const csHash_CRC32 = 68

Rem Cipher
const csCipher_3Way = 128
const csCipher_Blowfish= 129
const csCipher_Gost = 130
const csCipher_IDEA = 131
const csCipher_Q128 = 132
const csCipher_SAFER_K40 = 133
const csCipher_SAFER_SK40 = 134
const csCipher_SAFER_K64 = 135
const csCipher_SAFER_SK64 = 136
const csCipher_SAFER_K128 = 137
const csCipher_SAFER_SK128 = 138
const csCipher_SCOP = 139
const csCipher_Shark = 140
const csCipher_Square = 141
const csCipher_TEA = 142
const csCipher_TEAN = 143
const csCipher_Twofish = 144
const csCipher_Cast128 = 145
const csCipher_Cast256 = 146
const csCipher_1DES = 147
const csCipher_2DES = 148
const csCipher_2DDES = 149
```

```
const csCipher_3DES = 150
const csCipher_3DDES = 151
const csCipher_3TDES = 152
const csCipher_DESX = 153
const csCipher_Diamond2 = 154
const csCipher_Diamond2Lite = 155
const csCipher_FROG = 156
const csCipher_Mars = 157
const csCipher_Misty = 158
const csCipher_NewDES = 159
const csCipher_RC2 = 160
const csCipher_RC4 = 161
const csCipher_RC5 = 162
const csCipher_RC6 = 163
const csCipher_Rijndael = 164
const csCipher_Sapphire = 165

Rem Cipher Modes
const cmCTS = 0
const cmCBC = 1
const cmCFB = 2
const cmOFB = 3
const cmECB = 4
const cmCTSMAC = 5
const cmCBCMAC = 6
const cmCFBMAC = 7

Rem Format Objects
const fmtNone = -1
const fmtCopy = 0
const fmtHEX = 1
const fmtMIME64 = 2
const fmtUU = 3
const fmtXX = 4

Rem HashTipe
const st HashString= 0
const HashFILE = 1

Rem Checksums types
const csFSUM_MD5 = 0
const csFSUM_SFV = 1



%>
```

Sample

```
<%

    set Crypt = server.createobject (" aspCrypt.EasyCRYPT ")
    response.Write " <p><b> " + Crypt.Version + " </b></p> "
    ' Hash a String
    TestHash = " This string will be hashed "
    response.Write " <b>Start a Hash Test of '" + TestHash + "' </b><br> "
    Crypt.Algorithm = csHash_MD5
    response.Write " Algorith Desc: " + Crypt.Description + " <br> "
    Crypt.Counter( -1 )
    Crypt.Hash TestHash, HashString
    response.Write " Result HEX: " + Crypt.Digest( fmtHEX ) + " <br> "
    response.Write " Time: " + Crypt.Counter( 0 ) + " <br> "
    response.Write " <br> "



%>
```