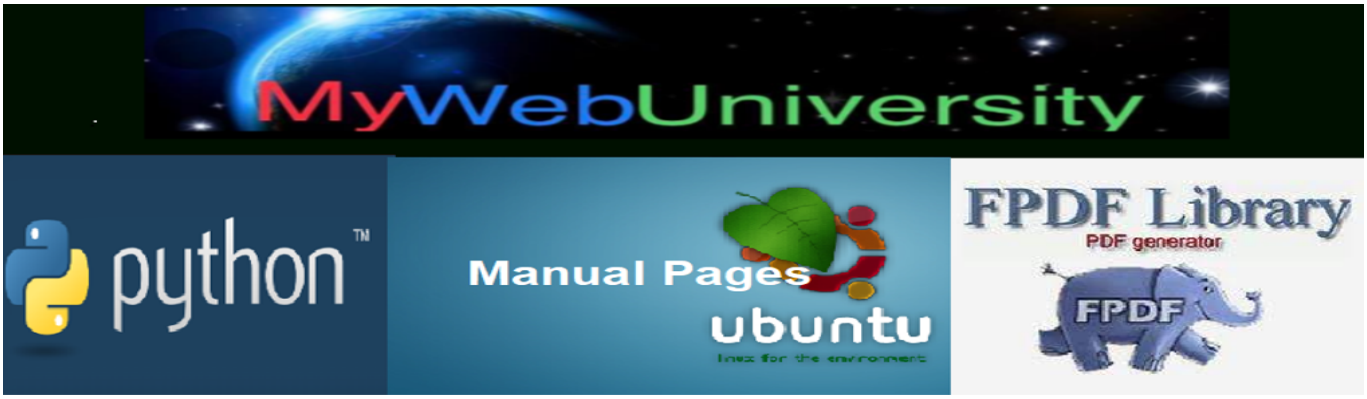

Rocky Enterprise Linux 9.2 Manual Pages on command 'Crypt::CBC.3pm'

$ man Crypt::CBC.3pm

Crypt::CBC(3pm) User Contributed Perl Documentation Crypt::CBC(3pm)

NAME

Crypt::CBC - Encrypt Data with Cipher Block Chaining Mode

SYNOPSIS

```
use Crypt::CBC;
$cipher = Crypt::CBC->new( -pass   => 'my secret password',
                           -cipher => 'Cipher::AES'
                          );
# one shot mode
$ciphertext = $cipher->encrypt("This data is hush hush");
$plaintext  = $cipher->decrypt($ciphertext);
# stream mode
$cipher->start('encrypting');
open(F,"./BIG_FILE");
while (read(F,$buffer,1024)) {
    print $cipher->crypt($buffer);
}
print $cipher->finish;
# do-it-yourself mode -- specify key && initialization vector yourself
$key    = Crypt::CBC->random_bytes(8);  # assuming a 8-byte block cipher
$iv     = Crypt::CBC->random_bytes(8);
$cipher = Crypt::CBC->new(-pbkdf       => 'none',
                          -key         => $key,
```

```
                        -iv          => $iv);
    $ciphertext = $cipher->encrypt("This data is hush hush");
    $plaintext  = $cipher->decrypt($ciphertext);
    # encrypting via a filehandle (requires Crypt::FileHandle>
    $fh = Crypt::CBC->filehandle(-pass => 'secret');
    open $fh,'>','encrypted.txt" or die $!
    print $fh "This will be encrypted\n";
    close $fh;
```

DESCRIPTION

This module is a Perl-only implementation of the cryptographic cipher block chaining mode (CBC). In combination with a block cipher such as AES or Blowfish, you can encrypt and decrypt messages of arbitrarily long length. The encrypted messages are compatible with the encryption format used by the OpenSSL package.

To use this module, you will first create a Crypt::CBC cipher object with new(). At the time of cipher creation, you specify an encryption key to use and, optionally, a block encryption algorithm. You will then call the start() method to initialize the encryption or decryption process, crypt() to encrypt or decrypt one or more blocks of data, and lastly finish(), to pad and encrypt the final block. For your convenience, you can call the encrypt() and decrypt() methods to operate on a whole data value at once.

new()

```
    $cipher = Crypt::CBC->new( -pass   => 'my secret key',
                               -cipher => 'Cipher::AES',
                             );
    # or (for compatibility with versions prior to 2.0)
    $cipher = new Crypt::CBC('my secret key' => 'Cipher::AES');
```

The new() method creates a new Crypt::CBC object. It accepts a list of -argument => value pairs selected from the following list:

Argument	Description
-pass,-key	The encryption/decryption passphrase. These arguments are interchangeable, but -pass is preferred ("key" is a misnomer, as it is not the literal encryption key).

-cipher The cipher algorithm (defaults to Crypt::Cipher:AES), or
a previously created cipher object reference. For
convenience, you may omit the initial "Crypt::" part
of the classname and use the basename, e.g. "Blowfish"
instead of "Crypt::Blowfish".

-keysize Force the cipher keysize to the indicated number of bytes. This can be used
to set the keysize for variable keylength ciphers such as AES.

-chain_mode The block chaining mode to use. Current options are:
'cbc' -- cipher-block chaining mode [default]
'pcbc' -- plaintext cipher-block chaining mode
'cfb' -- cipher feedback mode
'ofb' -- output feedback mode
'ctr' -- counter mode

-pbkdf The passphrase-based key derivation function used to derive
the encryption key and initialization vector from the
provided passphrase. For backward compatibility, Crypt::CBC
will default to "opensslv1", but it is recommended to use
the standard "pbkdf2"algorithm instead. If you wish to interoperate
with OpenSSL, be aware that different versions of the software
support a series of derivation functions.
'none' -- The value provided in -pass/-key is used directly.
This is the same as passing true to -literal_key.
You must also manually specify the IV with -iv.
The key and the IV must match the keylength
and blocklength of the chosen cipher.
'randomiv' -- Use insecure key derivation method found
in prehistoric versions of OpenSSL (dangerous)
'opensslv1' -- [default] Use the salted MD5 method that was default
in versions of OpenSSL through v1.0.2.
'opensslv2' -- [better] Use the salted SHA-256 method that was
the default in versions of OpenSSL through v1.1.0.
'pbkdf2' -- [best] Use the PBKDF2 method that was first
introduced in OpenSSL v1.1.1.

More derivation functions may be added in the future. To see the
supported list, use the command

perl -MCrypt::CBC::PBKDF -e 'print join "\n",Crypt::CBC::PBKDF->list'

-iter If the 'pbkdf2' key derivation algorithm is used, this specifies the number of
hashing cycles to be applied to the passphrase+salt (longer is more secure).
[default 10,000]

-hasher If the 'pbkdf2' key derivation algorithm is chosen, you can use this to provide
an initialized Crypt::PBKDF2::Hash object.
[default HMACSHA2 for OpenSSL compatability]

-header What type of header to prepend to the ciphertext. One of

'salt' -- use OpenSSL-compatible salted header (default)

'randomiv' -- Randomiv-compatible "RandomIV" header

'none' -- prepend no header at all
(compatible with prehistoric versions
of OpenSSL)

-iv The initialization vector (IV). If not provided, it will be generated
by the key derivation function.

-salt The salt passed to the key derivation function. If not provided, will be
generated randomly (recommended).

-padding The padding method, one of "standard" (default),
"space", "oneandzeroes", "rijndael_compat",
"null", or "none" (default "standard").

-literal_key [deprected, use -pbkdf=>'none']
If true, the key provided by "-key" or "-pass" is used
directly for encryption/decryption without salting or
hashing. The key must be the right length for the chosen
cipher.
[default false)

-pcbc [deprecated, use -chaining_mode=>'pcbc']

Whether to use the PCBC chaining algorithm rather than
the standard CBC algorithm (default false).

-add_header [deprecated; use -header instead]

Whether to add the salt and IV to the header of the output

cipher text.

-regenerate_key [deprecated; use -literal_key instead]

Whether to use a hash of the provided key to generate the actual encryption key (default true)

-prepend_iv [deprecated; use -header instead]

Whether to prepend the IV to the beginning of the encrypted stream (default true)

Crypt::CBC requires three pieces of information to do its job. First it needs the name of the block cipher algorithm that will encrypt or decrypt the data in blocks of fixed length known as the cipher's "blocksize." Second, it needs an encryption/decryption key to pass to the block cipher. Third, it needs an initialization vector (IV) that will be used to propagate information from one encrypted block to the next. Both the key and the IV must be exactly the same length as the chosen cipher's blocksize.

Crypt::CBC can derive the key and the IV from a passphrase that you provide, or can let you specify the true key and IV manually. In addition, you have the option of embedding enough information to regenerate the IV in a short header that is emitted at the start of the encrypted stream, or outputting a headerless encryption stream. In the first case, Crypt::CBC will be able to decrypt the stream given just the original key or passphrase. In the second case, you will have to provide the original IV as well as the key/passphrase.

The -cipher option specifies which block cipher algorithm to use to encode each section of the message. This argument is optional and will default to the secure Crypt::Cipher::AES algorithm. You may use any compatible block encryption algorithm that you have installed. Currently, this includes Crypt::Cipher::AES, Crypt::DES, Crypt::DES_EDE3, Crypt::IDEA, Crypt::Blowfish, Crypt::CAST5 and Crypt::Rijndael. You may refer to them using their full names ("Crypt::IDEA") or in abbreviated form ("IDEA").

Instead of passing the name of a cipher class, you may pass an already-created block cipher object. This allows you to take advantage of cipher algorithms that have parameterized new() methods, such as Crypt::Eksblowfish:

```
my $eksblowfish = Crypt::Eksblowfish->new(8,$salt,$key);
my $cbc         = Crypt::CBC->new(-cipher=>$eksblowfish);
```

The -pass argument provides a passphrase to use to generate the encryption key or the literal value of the block cipher key. If used in passphrase mode (which is the default),

-pass can be any number of characters; the actual key will be derived by passing the passphrase through a series of hashing operations. To take full advantage of a given block cipher, the length of the passphrase should be at least equal to the cipher's blocksize.

For backward compatibility, you may also refer to this argument using -key.

To skip this hashing operation and specify the key directly, provide the actual key as a string to -key and specify a key derivation function of "none" to the -pbkdf argument.

Alternatively, you may pass a true value to the -literal_key argument. When you manually specify the key in this way, should choose a key of length exactly equal to the cipher's key length. You will also have to specify an IV equal in length to the cipher's blocksize.

These choices imply a header mode of "none."

If you pass an existing Crypt::* object to new(), then the -pass/-key argument is ignored and the module will generate a warning.

The -pbkdf argument specifies the algorithm used to derive the true key and IV from the provided passphrase (PBKDF stands for "passphrase-based key derivation function"). Valid values are:

"opensslv1" -- [default] A fast algorithm that derives the key by
combining a random salt values with the passphrase via
a series of MD5 hashes.

"opensslv2" -- an improved version that uses SHA-256 rather
than MD5, and has been OpenSSL's default since v1.1.0.
However, it has been deprecated in favor of pbkdf2
since OpenSSL v1.1.1.

"pbkdf2" -- a better algorithm implemented in OpenSSL v1.1.1,
described in RFC 2898 L<https://tools.ietf.org/html/rfc2898>

"none" -- don't use a derivation function, but treat the passphrase
as the literal key. This is the same as B<-literal_key> true.

"nosalt" -- an insecure key derivation method used by prehistoric versions
of OpenSSL, provided for backward compatibility. Don't use.

"opensslv1" was OpenSSL's default key derivation algorithm through version 1.0.2, but is susceptible to dictionary attacks and is no longer supported. It remains the default for Crypt::CBC in order to avoid breaking compatibility with previously-encrypted messages. Using this option will issue a deprecation warning when initiating encryption. You can suppress the warning by passing a true value to the -nodeprecate option.

It is recommended to specify the "pbkdf2" key derivation algorithm when compatibility with older versions of Crypt::CBC is not needed. This algorithm is deliberately computationally expensive in order to make dictionary-based attacks harder. As a result, it introduces a slight delay before an encryption or decryption operation starts.

The -iter argument is used in conjunction with the "pbkdf2" key derivation option. Its value indicates the number of hashing cycles used to derive the key. Larger values are more secure, but impose a longer delay before encryption/decryption starts. The default is 10,000 for compatibility with OpenSSL's default.

The -hasher argument is used in conjunction with the "pbkdf2" key derivation option to pass the reference to an initialized Crypt::PBKDF2::Hash object. If not provided, it defaults to the OpenSSL-compatible hash function HMACSHA2 initialized with its default options (SHA-256 hash).

The -header argument specifies what type of header, if any, to prepend to the beginning of the encrypted data stream. The header allows Crypt::CBC to regenerate the original IV and correctly decrypt the data without your having to provide the same IV used to encrypt the data. Valid values for the -header are:

"salt" -- Combine the passphrase with an 8-byte random value to generate both the block cipher key and the IV from the provided passphrase. The salt will be appended to the beginning of the data stream allowing decryption to regenerate both the key and IV given the correct passphrase. This method is compatible with current versions of OpenSSL.

"randomiv" -- Generate the block cipher key from the passphrase, and choose a random 8-byte value to use as the IV. The IV will be prepended to the data stream. This method is compatible with ciphertext produced by versions of the library prior to 2.17, but is incompatible with block ciphers that have non 8-byte block sizes, such as Rijndael. Crypt::CBC will exit with a fatal error if you try to use this header mode with a non 8-byte cipher. This header type is NOT secure and NOT recommended.

"none" -- Do not generate a header. To decrypt a stream encrypted in this way, you will have to provide the true key and IV

manually.

The "salt" header is now the default as of Crypt::CBC version 2.17. In all earlier versions "randomiv" was the default.

When using a "salt" header, you may specify your own value of the salt, by passing the desired 8-byte character string to the -salt argument. Otherwise, the module will generate a random salt for you. Crypt::CBC will generate a fatal error if you specify a salt value that isn't exactly 8 bytes long. For backward compatibility reasons, passing a value of "1" will generate a random salt, the same as if no -salt argument was provided.

The -padding argument controls how the last few bytes of the encrypted stream are dealt with when they not an exact multiple of the cipher block length. The default is "standard", the method specified in PKCS#5.

The -chaining_mode argument will select among several different block chaining modes. Values are:

'cbc' -- [default] traditional Cipher-Block Chaining mode. It has the property that if one block in the ciphertext message is damaged, only that block and the next one will be rendered un-decryptable.

'pcbc' -- Plaintext Cipher-Block Chaining mode. This has the property that one damaged ciphertext block will render the remainder of the message unreadable

'cfb' -- Cipher Feedback Mode. In this mode, both encryption and decryption are performed using the block cipher's "encrypt" algorithm. The error propagation behaviour is similar to CBC's.

'ofb' -- Output Feedback Mode. Similar to CFB, the block cipher's encrypt algorithm is used for both encryption and decryption. If one bit of the plaintext or ciphertext message is damaged, the damage is confined to a single block of the corresponding ciphertext or plaintext, and error correction algorithms can be used to reconstruct the damaged part.

'ctr' -- Counter Mode. This mode uses a one-time "nonce" instead of an IV. The nonce is incremented by one for each block of plain or ciphertext, encrypted using the chosen algorithm, and then applied to the block of text. If one

bit of the input text is damaged, it only affects 1 bit
of the output text. To use CTR mode you will need to
install the Perl Math::Int128 module. This chaining method
is roughly half the speed of the others due to integer
arithmetic.

Passing a -pcbc argument of true will have the same effect as -chaining_mode=>'pcbc', and is included for backward compatibility. [deprecated].

For more information on chaining modes, see <http://www.crypto-it.net/eng/theory/modes-of-block-ciphers.html>.

The -keysize argument can be used to force the cipher's keysize. This is useful for several of the newer algorithms, including AES, ARIA, Blowfish, and CAMELLIA. If -keysize is not specified, then Crypt::CBC will use the value returned by the cipher's max_keylength() method. Note that versions of CBC::Crypt prior to 2.36 could also allow you to set the blocksie, but this was never supported by any ciphers and has been removed.

For compatibility with earlier versions of this module, you can provide new() with a hashref containing key/value pairs. The key names are the same as the arguments described earlier, but without the initial hyphen. You may also call new() with one or two positional arguments, in which case the first argument is taken to be the key and the second to be the optional block cipher algorithm.

start()

```
$cipher->start('encrypting');
$cipher->start('decrypting');
```

The start() method prepares the cipher for a series of encryption or decryption steps, resetting the internal state of the cipher if necessary. You must provide a string indicating whether you wish to encrypt or decrypt. "E" or any word that begins with an "e" indicates encryption. "D" or any word that begins with a "d" indicates decryption.

crypt()

```
$ciphertext = $cipher->crypt($plaintext);
```

After calling start(), you should call crypt() as many times as necessary to encrypt the desired data.

finish()

```
$ciphertext = $cipher->finish();
```

The CBC algorithm must buffer data blocks internally until they are even multiples of the

encryption algorithm's blocksize (typically 8 bytes). After the last call to crypt() you should call finish(). This flushes the internal buffer and returns any leftover ciphertext.

In a typical application you will read the plaintext from a file or input stream and write the result to standard output in a loop that might look like this:

```
$cipher = new Crypt::CBC('hey jude!');
$cipher->start('encrypting');
print $cipher->crypt($_) while <>;
print $cipher->finish();
```

encrypt()

```
$ciphertext = $cipher->encrypt($plaintext)
```

This convenience function runs the entire sequence of start(), crypt() and finish() for you, processing the provided plaintext and returning the corresponding ciphertext.

decrypt()

```
$plaintext = $cipher->decrypt($ciphertext)
```

This convenience function runs the entire sequence of start(), crypt() and finish() for you, processing the provided ciphertext and returning the corresponding plaintext.

encrypt_hex(), decrypt_hex()

```
$ciphertext = $cipher->encrypt_hex($plaintext)
$plaintext  = $cipher->decrypt_hex($ciphertext)
```

These are convenience functions that operate on ciphertext in a hexadecimal representation. encrypt_hex($plaintext) is exactly equivalent to unpack('H*',encrypt($plaintext)). These functions can be useful if, for example, you wish to place the encrypted in an email message.

filehandle()

This method returns a filehandle for transparent encryption or decryption using Christopher Dunkle's excellent Crypt::FileHandle module. This module must be installed in order to use this method.

filehandle() can be called as a class method using the same arguments as new():

```
$fh = Crypt::CBC->filehandle(-cipher=> 'Blowfish',
                             -pass  => "You'll never guess");
```

or on a previously-created Crypt::CBC object:

```
$cbc = Crypt::CBC->new(-cipher=> 'Blowfish',
```

```
                        -pass  => "You'll never guess");
    $fh  = $cbc->filehandle;
```

The filehandle can then be opened using the familiar open() syntax. Printing to a filehandle opened for writing will encrypt the data. Filehandles opened for input will be decrypted.

Here is an example:

```
    # transparent encryption
    open $fh,'>','encrypted.out' or die $!;
    print $fh "You won't be able to read me!\n";
    close $fh;
    # transparent decryption
    open $fh,'<','encrypted.out' or die $!;
    while (<$fh>) { print $_ }
    close $fh;
```

get_initialization_vector()

```
    $iv = $cipher->get_initialization_vector()
```

This function will return the IV used in encryption and or decryption. The IV is not guaranteed to be set when encrypting until start() is called, and when decrypting until crypt() is called the first time. Unless the IV was manually specified in the new() call, the IV will change with every complete encryption operation.

set_initialization_vector()

```
    $cipher->set_initialization_vector('76543210')
```

This function sets the IV used in encryption and/or decryption. This function may be useful if the IV is not contained within the ciphertext string being decrypted, or if a particular IV is desired for encryption. Note that the IV must match the chosen cipher's blocksize bytes in length.

iv()

```
    $iv = $cipher->iv();
    $cipher->iv($new_iv);
```

As above, but using a single method call.

key()

```
    $key = $cipher->key();
    $cipher->key($new_key);
```

Get or set the block cipher key used for encryption/decryption. When encrypting, the key is not guaranteed to exist until start() is called, and when decrypting, the key is not guaranteed to exist until after the first call to crypt(). The key must match the length required by the underlying block cipher.

When salted headers are used, the block cipher key will change after each complete sequence of encryption operations.

salt()

```
$salt = $cipher->salt();
$cipher->salt($new_salt);
```

Get or set the salt used for deriving the encryption key and IV when in OpenSSL compatibility mode.

passphrase()

```
$passphrase = $cipher->passphrase();
$cipher->passphrase($new_passphrase);
```

This gets or sets the value of the passphrase passed to new() when literal_key is false.

$data = random_bytes($numbytes)

Return $numbytes worth of random data. On systems that support the "/dev/urandom" device file, this data will be read from the device. Otherwise, it will be generated by repeated calls to the Perl rand() function.

cipher(), pbkdf(), padding(), keysize(), blocksize(), chain_mode()

These read-only methods return the identity of the chosen block cipher algorithm, the key derivation function (e.g. "opensslv1"), padding method, key and block size of the chosen block cipher, and what chaining mode ("cbc", "ofb" ,etc) is being used.

Padding methods

Use the 'padding' option to change the padding method.

When the last block of plaintext is shorter than the block size, it must be padded.

Padding methods include: "standard" (i.e., PKCS#5), "oneandzeroes", "space", "rijndael_compat", "null", and "none".

standard: (default) Binary safe

pads with the number of bytes that should be truncated. So, if blocksize is 8, then "0A0B0C" will be padded with "05", resulting in "0A0B0C0505050505". If the final block is a full block of 8 bytes, then a whole block of "0808080808080808" is appended.

oneandzeroes: Binary safe

pads with "80" followed by as many "00" necessary to fill the block. If the last block is a full block and blocksize is 8, a block of "8000000000000000" will be appended.

rijndael_compat: Binary safe, with caveats

similar to oneandzeroes, except that no padding is performed if the last block is a full block. This is provided for compatibility with Crypt::Rijndael's buit-in MODE_CBC. Note that Crypt::Rijndael's implementation of CBC only works with messages that are even multiples of 16 bytes.

null: text only

pads with as many "00" necessary to fill the block. If the last block is a full block and blocksize is 8, a block of "0000000000000000" will be appended.

space: text only

same as "null", but with "20".

none:

no padding added. Useful for special-purpose applications where you wish to add custom padding to the message.

Both the standard and oneandzeroes paddings are binary safe. The space and null paddings are recommended only for text data. Which type of padding you use depends on whether you wish to communicate with an external (non Crypt::CBC library). If this is the case, use whatever padding method is compatible.

You can also pass in a custom padding function. To do this, create a function that takes the arguments:

```
$padded_block = function($block,$blocksize,$direction);
```

where $block is the current block of data, $blocksize is the size to pad it to, $direction is "e" for encrypting and "d" for decrypting, and $padded_block is the result after padding or depadding.

When encrypting, the function should always return a string of <blocksize> length, and when decrypting, can expect the string coming in to always be that length. See _standard_padding(), _space_padding(), _null_padding(), or _oneandzeroes_padding() in the source for examples.

Standard and oneandzeroes padding are recommended, as both space and null padding can potentially truncate more characters than they should.

Comparison to Crypt::Mode::CBC

The CryptX modules Crypt::Mode::CBC, Crypt::Mode::OFB, Crypt::Mode::CFB, and Crypt::Mode::CTR provide fast implementations of the respective cipherblock chaining modes (roughly 5x the speed of Crypt::CBC). Crypt::CBC was designed to encrypt and decrypt messages in a manner compatible with OpenSSL's "enc" function. Hence it handles the derivation of the key and IV from a passphrase using the same conventions as OpenSSL, and it writes out an OpenSSL-compatible header in the encrypted message in a manner that allows the key and IV to be regenerated during decryption.

In contrast, the CryptX modules do not automatically derive the key and IV from a passphrase or write out an encrypted header. You will need to derive and store the key and IV by other means (e.g. with CryptX's Crypt::KeyDerivation module, or with Crypt::PBKDF2).

EXAMPLES

Three examples, aes.pl, des.pl and idea.pl can be found in the eg/ subdirectory of the Crypt-CBC distribution. These implement command-line DES and IDEA encryption algorithms using default parameters, and should be compatible with recent versions of OpenSSL. Note that aes.pl uses the "pbkdf2" key derivation function to generate its keys. The other two were distributed with pre-PBKDF2 versions of Crypt::CBC, and use the older "opensslv1" algorithm.

LIMITATIONS

The encryption and decryption process is about a tenth the speed of the equivalent OpenSSL tool and about a fifth of the Crypt::Mode::CBC module (both which use compiled C).

BUGS

Please report them.

AUTHOR

Lincoln Stein, lstein@cshl.org

SEE ALSO

perl(1), CryptX, Crypt::FileHandle, Crypt::Cipher::AES, Crypt::Blowfish, Crypt::CAST5, Crypt::DES, Crypt::IDEA, Crypt::Rijndael