# 

# Pgcrypto

# *THEORY*

The pgcrypto module provides cryptographic functions for PostgreSQL.

## General Hashing Functions

### digest()

digest(data text, type text) returns bytea

digest(data bytea, type text) returns bytea

Computes a binary hash of the given data. type is the algorithm to use. Standard algorithms are md5, sha1, sha224, sha256, sha384 and sha512. If pgcrypto was built with OpenSSL, more algorithms are available, as detailed in [**Table F-20**](https://www.postgresql.org/docs/9.1/pgcrypto.html#PGCRYPTO-WITH-WITHOUT-OPENSSL).

If you want the digest as a hexadecimal string, use encode() on the result. For example:

CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$

SELECT encode(digest($1, 'sha1'), 'hex')

$$ LANGUAGE SQL STRICT IMMUTABLE;

### hmac()

hmac(data text, key text, type text) returns bytea

hmac(data bytea, key text, type text) returns bytea

Calculates hashed MAC for data with key key. type is the same as in digest().

This is similar to digest() but the hash can only be recalculated knowing the key. This prevents the scenario of someone altering data and also changing the hash to match.

If the key is larger than the hash block size it will first be hashed and the result will be used as key.

## Password Hashing Functions

The functions crypt() and gen\_salt() are specifically designed for hashing passwords. crypt() does the hashing and gen\_salt() prepares algorithm parameters for it.

The algorithms in crypt() differ from usual hashing algorithms like MD5 or SHA1 in the following respects:

1. They are slow. As the amount of data is so small, this is the only way to make brute-forcing passwords hard.
2. They use a random value, called the *salt*, so that users having the same password will have different encrypted passwords. This is also an additional defense against reversing the algorithm.
3. They include the algorithm type in the result, so passwords hashed with different algorithms can co-exist.
4. Some of them are adaptive — that means when computers get faster, you can tune the algorithm to be slower, without introducing incompatibility with existing passwords.

[**Table F-17**](https://www.postgresql.org/docs/9.1/pgcrypto.html#PGCRYPTO-CRYPT-ALGORITHMS) lists the algorithms supported by the crypt() function.

**Table F-17. Supported Algorithms for**crypt()

| **Algorithm** | **Max Password Length** | **Adaptive?** | **Salt Bits** | **Description** |
| --- | --- | --- | --- | --- |
| bf | 72 | yes | 128 | Blowfish-based, variant 2a |
| md5 | unlimited | no | 48 | MD5-based crypt |
| xdes | 8 | yes | 24 | Extended DES |
| des | 8 | no | 12 | Original UNIX crypt |

### crypt()

crypt(password text, salt text) returns text

Calculates a crypt(3)-style hash of password. When storing a new password, you need to use gen\_salt() to generate a new salt value. To check a password, pass the stored hash value as salt, and test whether the result matches the stored value.

Example of setting a new password:

UPDATE ... SET pswhash = crypt('new password', gen\_salt('md5'));

Example of authentication:

SELECT pswhash = crypt('entered password', pswhash) FROM ... ;

This returns true if the entered password is correct.

### gen\_salt()

gen\_salt(type text [, iter\_count integer ]) returns text

Generates a new random salt string for use in crypt(). The salt string also tells crypt() which algorithm to use.

The type parameter specifies the hashing algorithm. The accepted types are: des, xdes, md5 and bf.

The iter\_count parameter lets the user specify the iteration count, for algorithms that have one. The higher the count, the more time it takes to hash the password and therefore the more time to break it. Although with too high a count the time to calculate a hash may be several years — which is somewhat impractical. If the iter\_count parameter is omitted, the default iteration count is used. Allowed values for iter\_count depend on the algorithm and are shown in [**Table F-18**](https://www.postgresql.org/docs/9.1/pgcrypto.html#PGCRYPTO-ICFC-TABLE).

**Table F-18. Iteration Counts for**crypt()

| **Algorithm** | **Default** | **Min** | **Max** |
| --- | --- | --- | --- |
| xdes | 725 | 1 | 16777215 |
| bf | 6 | 4 | 31 |

For xdes there is an additional limitation that the iteration count must be an odd number.

To pick an appropriate iteration count, consider that the original DES crypt was designed to have the speed of 4 hashes per second on the hardware of that time. Slower than 4 hashes per second would probably dampen usability. Faster than 100 hashes per second is probably too fast.

[**Table F-19**](https://www.postgresql.org/docs/9.1/pgcrypto.html#PGCRYPTO-HASH-SPEED-TABLE) gives an overview of the relative slowness of different hashing algorithms. The table shows how much time it would take to try all combinations of characters in an 8-character password, assuming that the password contains either only lower case letters, or upper- and lower-case letters and numbers. In the crypt-bf entries, the number after a slash is the iter\_count parameter of gen\_salt.

**Table F-19. Hash Algorithm Speeds**

| **Algorithm** | **Hashes/sec** | **For**[a-z] | **For**[A-Za-z0-9] |
| --- | --- | --- | --- |
| crypt-bf/8 | 28 | 246 years | 251322 years |
| crypt-bf/7 | 57 | 121 years | 123457 years |
| crypt-bf/6 | 112 | 62 years | 62831 years |
| crypt-bf/5 | 211 | 33 years | 33351 years |
| crypt-md5 | 2681 | 2.6 years | 2625 years |
| crypt-des | 362837 | 7 days | 19 years |
| sha1 | 590223 | 4 days | 12 years |
| md5 | 2345086 | 1 day | 3 years |

Notes:

* The machine used is a 1.5GHz Pentium 4.
* crypt-des and crypt-md5 algorithm numbers are taken from John the Ripper v1.6.38 -test output.
* md5 numbers are from mdcrack 1.2.
* sha1 numbers are from lcrack-20031130-beta.
* crypt-bf numbers are taken using a simple program that loops over 1000 8-character passwords. That way I can show the speed with different numbers of iterations. For reference: john -test shows 213 loops/sec for crypt-bf/5. (The very small difference in results is in accordance with the fact that the crypt-bf implementation in pgcrypto is the same one used in John the Ripper.)

Note that "try all combinations" is not a realistic exercise. Usually password cracking is done with the help of dictionaries, which contain both regular words and various mutations of them. So, even somewhat word-like passwords could be cracked much faster than the above numbers suggest, while a 6-character non-word-like password may escape cracking. Or not.

## PGP Encryption Functions

The functions here implement the encryption part of the OpenPGP (RFC 4880) standard. Supported are both symmetric-key and public-key encryption.

An encrypted PGP message consists of 2 parts, or *packets*:

* Packet containing a session key — either symmetric-key or public-key encrypted.
* Packet containing data encrypted with the session key.

When encrypting with a symmetric key (i.e., a password):

1. The given password is hashed using a String2Key (S2K) algorithm. This is rather similar to crypt() algorithms — purposefully slow and with random salt — but it produces a full-length binary key.
2. If a separate session key is requested, a new random key will be generated. Otherwise the S2K key will be used directly as the session key.
3. If the S2K key is to be used directly, then only S2K settings will be put into the session key packet. Otherwise the session key will be encrypted with the S2K key and put into the session key packet.

When encrypting with a public key:

1. A new random session key is generated.
2. It is encrypted using the public key and put into the session key packet.

In either case the data to be encrypted is processed as follows:

1. Optional data-manipulation: compression, conversion to UTF-8, and/or conversion of line-endings.
2. The data is prefixed with a block of random bytes. This is equivalent to using a random IV.
3. An SHA1 hash of the random prefix and data is appended.
4. All this is encrypted with the session key and placed in the data packet.

### pgp\_sym\_encrypt()

pgp\_sym\_encrypt(data text, psw text [, options text ]) returns bytea

pgp\_sym\_encrypt\_bytea(data bytea, psw text [, options text ]) returns bytea

Encrypt data with a symmetric PGP key psw. The options parameter can contain option settings, as described below.

### pgp\_sym\_decrypt()

pgp\_sym\_decrypt(msg bytea, psw text [, options text ]) returns text

pgp\_sym\_decrypt\_bytea(msg bytea, psw text [, options text ]) returns bytea

Decrypt a symmetric-key-encrypted PGP message.

Decrypting bytea data with pgp\_sym\_decrypt is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with pgp\_sym\_decrypt\_bytea is fine.

The options parameter can contain option settings, as described below.

### pgp\_pub\_encrypt()

pgp\_pub\_encrypt(data text, key bytea [, options text ]) returns bytea

pgp\_pub\_encrypt\_bytea(data bytea, key bytea [, options text ]) returns bytea

Encrypt data with a public PGP key key. Giving this function a secret key will produce an error.

The options parameter can contain option settings, as described below.

### pgp\_pub\_decrypt()

pgp\_pub\_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text

pgp\_pub\_decrypt\_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea

Decrypt a public-key-encrypted message. key must be the secret key corresponding to the public key that was used to encrypt. If the secret key is password-protected, you must give the password in psw. If there is no password, but you want to specify options, you need to give an empty password.

Decrypting bytea data with pgp\_pub\_decrypt is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with pgp\_pub\_decrypt\_bytea is fine.

The options parameter can contain option settings, as described below.

### pgp\_key\_id()

pgp\_key\_id(bytea) returns text

pgp\_key\_id extracts the key ID of a PGP public or secret key. Or it gives the key ID that was used for encrypting the data, if given an encrypted message.

It can return 2 special key IDs:

* SYMKEY

The message is encrypted with a symmetric key.

* ANYKEY

The message is public-key encrypted, but the key ID has been removed. That means you will need to try all your secret keys on it to see which one decrypts it. pgcrypto itself does not produce such messages.

Note that different keys may have the same ID. This is rare but a normal event. The client application should then try to decrypt with each one, to see which fits — like handling ANYKEY.

### armor(), dearmor()

armor(data bytea) returns text

dearmor(data text) returns bytea

These functions wrap/unwrap binary data into PGP ASCII-armor format, which is basically Base64 with CRC and additional formatting.

### Options for PGP Functions

Options are named to be similar to GnuPG. An option's value should be given after an equal sign; separate options from each other with commas. For example:

pgp\_sym\_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')

All of the options except convert-crlf apply only to encrypt functions. Decrypt functions get the parameters from the PGP data.

The most interesting options are probably compress-algo and unicode-mode. The rest should have reasonable defaults.

#### Cipher-Algo

Which cipher algorithm to use.

Values: bf, aes128, aes192, aes256 (OpenSSL-only: 3des, cast5)  
Default: aes128  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### Compress-Algo

Which compression algorithm to use. Only available if PostgreSQL was built with zlib.

Values:  
  0 - no compression  
  1 - ZIP compression  
  2 - ZLIB compression (= ZIP plus meta-data and block CRCs)  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### Compress-Level

How much to compress. Higher levels compress smaller but are slower. 0 disables compression.

Values: 0, 1-9  
Default: 6  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### Convert-Crlf

Whether to convert \n into \r\n when encrypting and \r\n to \n when decrypting. RFC 4880 specifies that text data should be stored using \r\n line-feeds. Use this to get fully RFC-compliant behavior.

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt, pgp\_sym\_decrypt, pgp\_pub\_decrypt

#### Disable-Mdc

Do not protect data with SHA-1. The only good reason to use this option is to achieve compatibility with ancient PGP products, predating the addition of SHA-1 protected packets to RFC 4880. Recent gnupg.org and pgp.com software supports it fine.

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

#### Sess-Key

Use separate session key. Public-key encryption always uses a separate session key; this option is for symmetric-key encryption, which by default uses the S2K key directly.

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt

#### S2k-Mode

Which S2K algorithm to use.

Values:  
  0 - Without salt.  Dangerous!  
  1 - With salt but with fixed iteration count.  
  3 - Variable iteration count.  
Default: 3  
Applies to: pgp\_sym\_encrypt

#### S2k-Digest-Algo

Which digest algorithm to use in S2K calculation.

Values: md5, sha1  
Default: sha1  
Applies to: pgp\_sym\_encrypt

#### S2k-Cipher-Algo

Which cipher to use for encrypting separate session key.

Values: bf, aes, aes128, aes192, aes256  
Default: use cipher-algo  
Applies to: pgp\_sym\_encrypt

#### Unicode-Mode

Whether to convert textual data from database internal encoding to UTF-8 and back. If your database already is UTF-8, no conversion will be done, but the message will be tagged as UTF-8. Without this option it will not be.

Values: 0, 1  
Default: 0  
Applies to: pgp\_sym\_encrypt, pgp\_pub\_encrypt

### Generating PGP Keys with GnuPG

To generate a new key:

gpg --gen-key

The preferred key type is "DSA and Elgamal".

For RSA encryption you must create either DSA or RSA sign-only key as master and then add an RSA encryption subkey with gpg --edit-key.

To list keys:

gpg --list-secret-keys

To export a public key in ASCII-armor format:

gpg -a --export KEYID > public.key

To export a secret key in ASCII-armor format:

gpg -a --export-secret-keys KEYID > secret.key

You need to use dearmor() on these keys before giving them to the PGP functions. Or if you can handle binary data, you can drop -a from the command.

For more details see man gpg, [**The GNU Privacy Handbook**](http://www.gnupg.org/gph/en/manual.html) and other documentation on [**http://www.gnupg.org**](http://www.gnupg.org/).

### Limitations of PGP Code

* No support for signing. That also means that it is not checked whether the encryption subkey belongs to the master key.
* No support for encryption key as master key. As such practice is generally discouraged, this should not be a problem.
* No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with pgcrypto, but create new ones, as the usage scenario is rather different.

## Raw Encryption Functions

These functions only run a cipher over data; they don't have any advanced features of PGP encryption. Therefore they have some major problems:

1. They use user key directly as cipher key.
2. They don't provide any integrity checking, to see if the encrypted data was modified.
3. They expect that users manage all encryption parameters themselves, even IV.
4. They don't handle text.

So, with the introduction of PGP encryption, usage of raw encryption functions is discouraged.

encrypt(data bytea, key bytea, type text) returns bytea

decrypt(data bytea, key bytea, type text) returns bytea

encrypt\_iv(data bytea, key bytea, iv bytea, type text) returns bytea

decrypt\_iv(data bytea, key bytea, iv bytea, type text) returns bytea

Encrypt/decrypt data using the cipher method specified by type. The syntax of the type string is:

***algorithm*** [ - ***mode*** ] [ /pad: ***padding*** ]

where ***algorithm*** is one of:

* bf — Blowfish
* aes — AES (Rijndael-128)

and ***mode*** is one of:

* cbc — next block depends on previous (default)
* ecb — each block is encrypted separately (for testing only)

and ***padding*** is one of:

* pkcs — data may be any length (default)
* none — data must be multiple of cipher block size

So, for example, these are equivalent:

encrypt(data, 'fooz', 'bf')

encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')

In encrypt\_iv and decrypt\_iv, the iv parameter is the initial value for the CBC mode; it is ignored for ECB. It is clipped or padded with zeroes if not exactly block size. It defaults to all zeroes in the functions without this parameter.

## Random-Data Functions

gen\_random\_bytes(count integer) returns bytea

Returns count cryptographically strong random bytes. At most 1024 bytes can be extracted at a time. This is to avoid draining the randomness generator pool.

## Notes

### Configuration

pgcrypto configures itself according to the findings of the main PostgreSQL configure script. The options that affect it are --with-zlib and --with-openssl.

When compiled with zlib, PGP encryption functions are able to compress data before encrypting.

When compiled with OpenSSL, there will be more algorithms available. Also public-key encryption functions will be faster as OpenSSL has more optimized BIGNUM functions.

**Table F-20. Summary of Functionality with and without OpenSSL**

| **Functionality** | **Built-in** | **With OpenSSL** |
| --- | --- | --- |
| MD5 | yes | yes |
| SHA1 | yes | yes |
| SHA224/256/384/512 | yes | yes (Note 1) |
| Other digest algorithms | no | yes (Note 2) |
| Blowfish | yes | yes |
| AES | yes | yes (Note 3) |
| DES/3DES/CAST5 | no | yes |
| Raw encryption | yes | yes |
| PGP Symmetric encryption | yes | yes |
| PGP Public-Key encryption | yes | yes |

Notes:

1. SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, pgcrypto will use built-in code.
2. Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.
3. AES is included in OpenSSL since version 0.9.7. For older versions, pgcrypto will use built-in code.

### NULL Handling

As is standard in SQL, all functions return NULL, if any of the arguments are NULL. This may create security risks on careless usage.

***PRACTICE***

CREATE TABLE testusers**(**username varchar**(**100**)** PRIMARY KEY, cryptpwd text, md5pwd text**)**;

INSERT INTO testusers**(**username, cryptpwd, md5pwd**)**

VALUES **(***'robby'*, crypt**(***'test'*, gen\_salt**(***'md5'***))**, md5**(***'test'***))**,

**(***'artoo'*, crypt**(***'test'*,gen\_salt**(***'md5'***))**, md5**(***'test'***))**;

SELECT username, cryptpwd, md5pwd

FROM testusers;

username **|** cryptpwd **|** md5pwd

robby **|** $1$IOchfG**/**z$bZW1pRFA3wuvn6pAuD.Du**/** **|** 098f6bcd4621d373cade4e832627b4f6

artoo **|** $1$84oZTXI**/**$yZ6wV5jhJo6aQYrTciMQR**/** **|** 098f6bcd4621d373cade4e832627b4f6

SELECT username

FROM testusers

WHERE username **=** *'robby'* AND cryptpwd **=** crypt**(***'test'*, cryptpwd**)**;

SELECT username

FROM testusers

WHERE username **=** *'artoo'* AND cryptpwd **=** crypt**(***'test'*, cryptpwd**)**;

SELECT username

FROM testusers

WHERE username **=** *'artoo'* AND cryptpwd **=** crypt**(***'artoo'*, cryptpwd**)**;

SELECT username

FROM testusers

WHERE username **=** *'robby'* and md5pwd **=** md5**(***'test'***)**;

CREATE TABLE testuserscards**(**card\_id SERIAL PRIMARY KEY, username varchar**(**100**)**, cc bytea**)**;

INSERT INTO testuserscards**(**username, cc**)**

SELECT robotccs.username, pgp\_pub\_encrypt**(**robotccs.cc, keys.pubkey**)** As cc

FROM **(**VALUES **(***'robby'*, *'41111111111111111'***)**,

**(***'artoo'*, *'41111111111111112'***)** **)** As robotccs**(**username, cc**)**

CROSS JOIN **(**SELECT dearmor**(***'-----BEGIN PGP PUBLIC KEY BLOCK-----*

*super publickey goobly gook goes here*

*-----END PGP PUBLIC KEY BLOCK-----'***)** As pubkey**)** As keys;

SELECT username, cc

FROM testuserscards;

SELECT pgp\_key\_id**(**dearmor**(***'-----BEGIN PGP PUBLIC KEY BLOCK-----*

*super publickey goobly gook goes here*

*-----END PGP PUBLIC KEY BLOCK-----'***))**;

E0B086C2999DEFG

SELECT username, pgp\_key\_id**(**cc**)** As keyweused

FROM testuserscards;

username **|** keyweused

*----------+------------------*

robby **|** E0B086C2999DEFG

artoo **|** E0B086C2999DEFG

*-- To decrpt the data*

SELECT username, pgp\_pub\_decrypt**(**cc, keys.privkey**)** As ccdecrypt

FROM testuserscards

CROSS JOIN

**(**SELECT dearmor**(***'-----BEGIN PGP PRIVATE KEY BLOCK-----*

*super private key gobbly gook goes here*

*-----END PGP PRIVATE KEY BLOCK-----'***)** As privkey**)** As keys;

username **|** ccdecrypt

robby **|** 41111111111111111

artoo **|** 41111111111111112