The pgcrypto extension

Secure your data with cryptographic functions in Postgres

- The pgcrypto extension offers a range of cryptographic functions within Postgres. These functions enable encryption, decryption, and hashing operations through standard SQL queries. This can reduce reliance on external cryptographic tools for data security tasks in a Postgres environment.

- In this guide, you'll learn how to enable the pgcrypto extension on Neon, use its core cryptographic functions, explore practical applications for data security, and follow best practices for managing security considerations.

❖ Enable the pgcrypto extension

❖ You can enable the extension by running the following CREATE EXTENSION statement in the Neon SQL Editor or from a client such as psql that is connected to your Neon database.

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

Version availability:

❖ Please refer to the list of all extensions available in Neon for up-to-date extension version information.

Cryptographic functions

❖ The pgcrypto extension provides a wide range of cryptographic functions that can be used directly within SQL queries. These functions can be broadly categorized into the following groups:

General hashing functions

- pgcrypto provides functions for generating one-way hashes, crucial for verifying data integrity and securely comparing data without revealing the original content.

- **digest(data, type):**

The digest function computes a binary hash of the input data using the algorithm specified by type. This function supports a wide range of algorithms, including
md5,sha1, and the SHA-2 family (sha224, sha256, sha384, sha512), as well as any other digest algorithm supported by the underlying OpenSSL library.

```
SELECT digest('Sensitive Information', 'sha256');
-- \x7daa83aa2e4618c8de40eb6642dbde3bceead971c322c66ed47676897a1b31c1 (binary output)
hmac(data, key, type):
```

- The hmac function calculates a keyed hash, also known as a Hash-based Message Authentication Code. It incorporates a secret key into the hashing process, ensuring that only parties with knowledge of the key can verify the hash. This provides both data integrity and authenticity.

```
SELECT hmac('Data to Authenticate', 'shared_secret_key', 'sha256');
-- \x261415730795bccaedb60061af12bf8fdb0833b4bad7735214dc78789e233257 (binary output)
```

Password hashing functions

- pgcrypto includes specialized functions designed for securely hashing passwords, essential for protecting user credentials.

crypt(password text, salt text):

- The crypt function implements a crypt(3)-style hashing algorithm, specifically tailored for password security. It takes the password to be hashed and a salt value as input.

```
SELECT crypt('user_password', gen_salt('md5'));
-- $1$bPYjhtip$NT.UC/6xLeoj8leDs7Neh0 (example hashed password output)
gen_salt(type text [, iter_count integer ]):
```

- The gen_salt function generates new, random salt values for use with the crypt() function. The type parameter specifies the hashing algorithm (e.g., bf for Blowfish, md5, xdes, des). For algorithms like Blowfish and Extended DES (xdes), you can specify iter_count to control the number of iterations, increasing the computational cost and security.

```
SELECT gen_salt('bf'); -- Generate a Blowfish salt
-- $2a$06$KlIoNEoix2oKbLwMimhQpu (example output)
SELECT gen_salt('bf', 10); -- Generate Blowfish salt with 2^10 iterations
-- $2a$10$nnHUvyZckh1VBh5zWNEFKO (example output)
```

PGP encryption functions

- For general-purpose encryption needs, pgcrypto implements the encryption part of the OpenPGP standard, providing functions for both symmetric-key and public-key encryption.

pgp_sym_encrypt(data, psw [, options]):

- The pgp_sym_encrypt function encrypts data using symmetric-key encryption with a provided password psw. Symmetric encryption uses the same key for both encryption and decryption.

```
SELECT pgp_sym_encrypt('Confidential Data', 'encryption_password');
--
\xc30d040703029c3eba2b3565a3937bd2420120ae6792f663bd35977d21a5a8e13de9a8a8e5a9212ef
06f8b056dcc31e0b48096915ddac66f14ab403ea671a8b4c740a198d32bcc5b804a30ef7e9aeacd7c124
6 (binary output)
```

pgp_sym_decrypt(msg, psw [, options]):

- The pgp_sym_decrypt function decrypts a message msg that was encrypted using symmetric-key encryption with the password psw.

```
SELECT pgp_sym_decrypt(encrypted_message, 'encryption_password');
-- SELECT
pgp_sym_decrypt('\xc30d040703029c3eba2b3565a3937bd2420120ae6792f663bd35977d21a5a8e13
de9a8a8e5a9212ef06f8b056dcc31e0b48096915ddac66f14ab403ea671a8b4c740a198d32bcc5b804a3
0ef7e9aeacd7c1246', 'encryption_password');
-- Confidential Data (plaintext output)
```

pgp_pub_encrypt(data, key [, options]):

- The pgp_pub_encrypt function encrypts data using public-key encryption with a provided public key. Public-key encryption uses separate keys for encryption (public key) and decryption (private key).

```
SELECT pgp_pub_encrypt('Secret Message', 'public_key_here');
-- encrypted_message (binary output)
```

pgp_pub_decrypt(msg, key [, psw [, options]]):

- The pgp_pub_decrypt function decrypts a message msg that was encrypted using public-key encryption. It requires the private key corresponding to the public key used for encryption. If the private key is password-protected, the psw is also required.

```
SELECT pgp_pub_decrypt(encrypted_message, 'private_key_here', 'private_key_password');
-- Secret Message (plaintext output)
```

Random data functions

- pgcrypto provides functions for generating cryptographically secure random data, essential for various security operations.

gen_random_bytes(count integer):

- The gen_random_bytes function generates a specified number of cryptographically strong random bytes. These bytes can be used as salts, initialization vectors, or for other security-sensitive purposes.

```
SELECT gen_random_bytes(16); -- Generate 16 random bytes
-- \xc9259a991537e3d730db78133f208e94 (example binary output)
```

gen_random_uuid():

The gen_random_uuid() function generates a version 4 universally unique identifier (UUID) based on random numbers. This is functionally equivalent to PostgreSQL's built-in gen_random_uuid().

```
SELECT gen_random_uuid();
-- 90d18ac7-4af7-458d-8f7a-a7211b5d3eee (example output)
```

Practical applications

- pgcrypto offers a wide range of practical applications for enhancing data security within your Postgres environment:

1. **Secure password storage:** Use crypt() and gen_salt() to securely store user passwords as hashes, protecting them from exposure in case of a data breach.

2. **Data encryption at rest (Column-Level):** Employ pgp_sym_encrypt() or pgp_pub_encrypt() to encrypt sensitive data columns within your tables, ensuring data confidentiality even if the database is compromised.

3. **Data anonymization:** Leverage encryption functions to pseudonymize or anonymize sensitive data for non-production environments or for compliance purposes.

Example: Secure password storage

- Let's walk through a practical example of using pgcrypto to securely store and verify user passwords in a Postgres database.

1. Hash and salt a password using crypt() and gen_salt()

Suppose you want to hash the password "mypassword". You'll use gen_salt() to generate a salt and crypt() to hash the password with the salt. For this example, we'll use the Blowfish algorithm with 4 rounds (iterations):

```
SELECT crypt('mypassword', gen_salt('bf', 4));
-- $2a$04$vVVrQ777SjxyQKuFp7z6ue (example hashed password output)
```

The output is the hashed password, which includes the salt and algorithm identifier. You should store this entire hashed password string in your database, not the original password.

2. Store the hashed password:

Create a table to store usernames and their hashed passwords.

```
CREATE TABLE users (
   username VARCHAR(50) PRIMARY KEY,
   password_hash TEXT NOT NULL
);
INSERT INTO users (username, password_hash) VALUES
('testuser', '$2a$04$vVVrQ777SjxyQKuFp7z6ue'); -- Replace with the hash from the previous step
```

3. Verify a password during login:

When a user attempts to log in, you'll receive the password they entered (e.g., "mypassword" again). To verify it, you'll use crypt() again, passing the entered password and the stored password_hash from the database.

```
SELECT password_hash = crypt('mypassword', password_hash) AS password_match
FROM users
WHERE username = 'testuser';
-- password_match
-- --------------
-- t
-- (1 row)
```

If the passwords match, crypt() will return the same stored hash (or a hash that compares as equal), and the query will return t (true).

4. Incorrect password attempt:

If the user enters an incorrect password (e.g., "wrongpassword"), the verification will fail:

```
SELECT password_hash = crypt('wrongpassword', password_hash) AS password_match
FROM users
WHERE username = 'testuser';
-- password_match
-- --------------
-- f
-- (1 row)
```

In this case, the query returns f (false), indicating an incorrect password.

By following these steps, you can securely store and verify user passwords using pgcrypto in your Postgres database.

Performance Implications

- While pgcrypto provides robust security features, it's important to consider the performance implications of cryptographic operations:

- **Computational overhead:** Encryption, decryption, and hashing operations inherently require computational resources. The extent of the overhead depends on the chosen algorithms, data size, and frequency of operations.

- **Password hashing:** Password hashing algorithms, like those used in crypt(), are intentionally designed to be slow to resist brute-force attacks. This can introduce a slight delay during user authentication processes.

Security Considerations

When using pgcrypto, it's crucial to adhere to security best practices:

- **Key management:** Securely manage encryption keys. Store them outside the database if possible, and implement key rotation policies. Never store keys in plaintext within the database as that would defeat the purpose of encryption.

- **Algorithm selection:** Choose appropriate cryptographic algorithms based on your security requirements. For password hashing, use strong algorithms like Blowfish with sufficient iteration counts. For data encryption, select robust and widely-vetted algorithms like AES.

Conclusion

The pgcrypto extension is a powerful and versatile tool for enhancing data security in Postgres. By providing a rich set of cryptographic functions, it enables you to implement robust security measures directly within your database environment. From secure password storage to data encryption and hashing, pgcrypto offers a wide range of applications to protect your data.