Workshop pgsodium

iffrer vos données dans PostgreSQL avec pgsodi

23.02

Contents

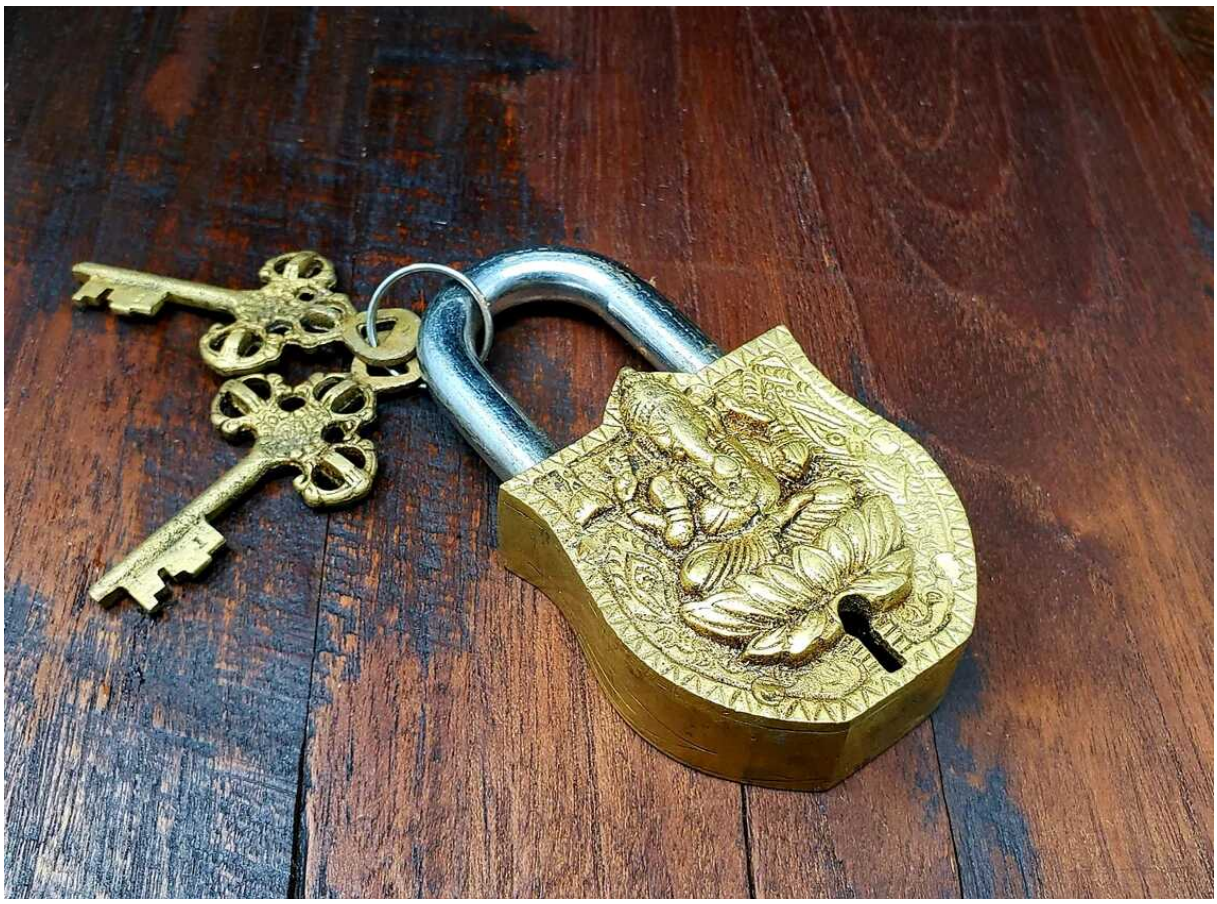

Figure 1: PostgreSQL et chiffrement

1/ Introduction

- Présentation
- Principes
- Installation et configuration
- Mise en œuvre
- Sécurité

Le but de cet atelier est de découvrir et faire connaitre une extension peu connue : pgsodium. Cette extension permet d'utiliser des fonctions cryptographiques en SQL.

En plus de ces fonctions, elle propose une sur-couche permettant la mise œuvre de colonne chiffrées de façon transparente du point de vue applicatif. Bien que très récente et pas tout à fait mature, cette fonctionnalité mérite déjà d'être étudiée comme source d'inspiration, solution pour un projet en cours ou futur.

1.1 PRÉ-REQUIS DE CET ATELIER

- instance PostgreSQL 13 ou supérieure
- Redhat (ou dérivé) + dépôts PGDG
  - ou Redhat + outils classiques de compilation
  - ou Debian + outils classiques de compilation

Certaines fonctionnalités de pgsodium sont compatibles avec PostgreSQL 12, mais l'implémentation TCE n'est compatible qu'à partir de PostgreSQL 13.

1.2 PRÉSENTATION LIBSODIUM

- « pgsodium » car basé sur « libsodium »
- lib de plus haut niveau par rapport à openssl
- force à utiliser les bonnes pratiques cryptographiques
- simplicité d'utilisation
- performances
- quelques références d'utilisateurs

libsodium est un *fork* de NaCl, qui ne s'écrit **pas** comme ça se prononce (« salt »).

NaCl est une bibliothèque cryptographique développée entre 2009 et 2011, principalement par Daniel J. Bernstein, une personnalité importante du monde de la cryptographie. Les principes de cette bibliothèque sont exposés dans cet article universitaire : principes de NaCl[1] Le besoin principal était de créer une bibliothèque de plus haut niveau par rapport à OpenSSL, avec une API simplifiée au maximum :

- regroupement de plusieurs opérations en une seule. Par exemple, la fonction `crypto_box(m,n,pk,sk)` permet de chiffrer et signer un message ;
- sélection par défaut des meilleures primitives cryptographiques.

Le développement de cette bibliothèque semble s'être arrêté en 2011, seule une archive du code est disponible sur internet.

À partir de 2013, Franck Denis a débuté le développement de libsodium, à partir des sources de NaCl, avec un mode de développement plus ouvert (sources[2] disponibles sur GitHub), une attention portée sur la portabilité du code et la mise à disposition de *bindings* pour les principaux langages de programmation. De plus, l'API a été étendue, et de nombreuses primitives cryptographiques ont été ajoutées.

Cette bibliothèque est toujours activement maintenue, et utilisée par un grand nombre d'entreprises et organisations[3], dont certains grands noms comme Facebook ou OVH.

---

[1] https://cr.yp.to/highspeed/coolnacl-20120725.pdf
[2] https://github.com/jedisct1/libsodium
[3] https://doc.libsodium.org/libsodium_users

1.3 PRÉSENTATION PGSODIUM

- créé en 2017 par Michel Pelletier
- actuellement maintenu et développé au sein de Supabase
- expose au niveau SQL de nombreuses fonctions de la libsodium
- apporte la possibilité de gérer les clés coté serveur
- propose une implémentation de TCE

Le projet pgsodium a été créé en 2017 par Michel Pelletier.

Auteur de pgJWT l'année précédente, il découvre alors qu'aucune solution n'existe dans l'écosystème PostgreSQL pour signer des données, fonctionnalité nécessaire à la spécification JWT (JSON Web Tokens). Ce fut l'élément déclencheur de l'implémentation de pgsodium à partir de la libsodium, afin d'exposer une telle fonctionnalité au niveau SQL.

L'auteur n'est depuis pas revenu sur pgJWT et poursuit son travail sur pgsodium au sein de Supabase, travaillant toujours sur les fonctionnalités de chiffrement et de signature.

pgSodium est en premier lieu un wrapper autour de la libsodium, exposant une grande partie des fonctionnalités de cette librairie au niveau SQL. Il permet ainsi de générérer des clés, hash ou nonce de qualité cryptographique et dans différents algorithmes, ou encore de chiffrer ou signer les données directement depuis l'instance PostgreSQL.

Grâce aux fonctionnalités de la libsodium et aux nombreux langages pouvant l'utiliser, il est par exemple possible de développer une application assurant le chiffrement des données de bout en bout, sans que la base de donnée ne soit capable de les déchiffrer, mais puisse malgré tout en valider l'authenticité ou en autoriser l'accès en fonction des certificats utilisateur.

Au delà des fonctionnalités de la libsodium, pgsodium ajoute une gestion des clés et des droits sur les différents objets créés. Notamment, pgsodium est capable d'exécuter les différentes fonctions cryptographiques sans jamais stocker en base une clé de chiffrement privée, ni la faire apparaitre dans une requête. Nous détaillerons ce point technique dans un prochain chapitre.

Aussi, en assemblant ces différentes pièces et en utilisant l'extensibilité de PostgreSQL, il propose une implémentation du chiffrement de colonne transparent, appelé *TCE*. Cette implémentation permet la mise en œuvre d'une base de donnée supportant le *data encryption-at-rest* pour les colonnes choisies.

Cet atelier met en œuvre et explique en détail le fonctionnement de cette implémentation de TCE, ses fonctionnalités et son administration.

Un travail en cours dans le core de PostgreSQL permettant de chiffrer les données d'une colonne de façon transparente a aussi adopté le nom *TCE*. Dans la suite de ce document, nous continuerons à utiliser *TCE* pour désigner l'implémentation proposée par pgsodium.

2/ Installation et configuration

- installation par paquet
- installation par les sources
- utilisation dans une base
- configuration du module

2.1 INSTALLATION PAR PAQUETS

- Debian : seule la libsodium est empaquetée, compilation nécessaire
- EL : paquets `pgsodium_XX` dans les dépôts PGDG seulement
  - compilation nécessaire sans les dépôts PGDG

Il n'existe pas de paquet pour pgsodium sur les systèmes d'exploitation Debian et dérivés. Seule la libsodium y est disponible, ce qui facilite néanmoins grandement l'installation par les sources (voir le chapitre suivant).

Pour les systèmes Entreprise Linux et dérivés, les paquets `pgsodium_XX` sont disponibles dans les dépôts PGDG. Si vous n'utilisez pas ces dépôts, voir le chapitre suivant.

Les dépôts PGDG supportent plusieurs versions majeures de PostgreSQL. Le nom du paquet pgsodium est suffixé par la version majeure pour laquelle il a été empaqueté. Il faut par exemple choisir le paquet `pgsodium_15` si vous avez installé `postgresql15-server`. La dernière version de pgsodium est disponible pour les versions 13 à 15 de PostgreSQL[1].

Installez le dépôt PGDG, puis PostgreSQL et pgsodium.

```
# prérequis : dépot EPEL
dnf install epel-release
dnf install https://download.postgresql.org/pub/repos/yum/\
  reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
dnf -qy module disable postgresql
dnf install pgsodium_15 postgresql15-server
```

[1] et par erreur pour la 12 aussi au moment où nous rédigeons ce document.

2.2 INSTALLATION PAR LES SOURCES

- nécessite libsodium ainsi que ses fichiers d'entête
- sources de pgsodium disponibles sur github
- `make install`
    - variable d'environnement `PG_CONFIG`

L'installation par les sources est relativement simple. Elle nécessite:

- d'avoir un environnement de compilation prêt à l'emploi
- que la libsodium et ses fichiers de développement soient installés
- que les fichiers de développement de PostgreSQL soient installés

Une fois ces pré-requis validés, la compilation et l'installation de pgsodium se fait avec la seule commande `make install` exécutée à la racine de son code source.

Si votre instance PostgreSQL n'est pas installée de façon traditionnelle, ou si vous avez plusieurs versions majeures disponibles sur le serveur, ou encore si l'outil `pg_config` n'est pas dans le PATH, il peut être nécessaire de positionner la variable d'environnement `PG_CONFIG` pointant sur la version de `pg_config` désirée. Par exemple:

```
PG_CONFIG=/usr/lib/postgresql/15/bin/pg_config
export PG_CONFIG
make install
```

ou plus simplement:

```
PG_CONFIG=/usr/lib/postgresql/15/bin/pg_config make install
```

Ci-après un exemple de compilation et d'installation pour Debian 11 et ses dérivés, sans utiliser les dépôts PGDG:

```
apt install build-essential libsodium-dev postgresql postgresql-server-dev-all
cd /usr/local/src
wget https://github.com/michelp/pgsodium/archive/refs/tags/v3.1.5.tar.gz
tar zxf v3.1.5.tar.gz
cd pgsodium-3.1.5/
make install
```

À titre d'information, les paquets nécessaires sur Entreprise Linux 8 et dérivés sont: `libsodium-devel`, `postgresql-server-devel` et `libpq-devel`. Attention à bien choisir la version 13 ou supérieur de PostgreSQL. Voici une procédure minimale:

```
dnf module enable -y postgresql:13
# activation redhat-rpm-config ( si "gcc: error:
 ↪  /usr/lib/rpm/redhat/redhat-hardened-cc1: No such file or directory" ):
dnf install dnf-plugins-core
dnf config-manager --set-enabled powertools

dnf install postgresql-server postgresql-server-devel libpq-devel \
  libsodium-devel redhat-rpm-config

cd /usr/local/src
wget https://github.com/michelp/pgsodium/archive/refs/tags/v3.1.5.tar.gz
tar zxf v3.1.5.tar.gz
cd pgsodium-3.1.5/
make install
```

2.3 VÉRIFICATIONS POST-INSTALLATION

Vérifier la bonne installation des éléments de pgsodium :

- un module
- une extension

Le projet pgsodium consiste en deux éléments :

- un module, sous forme de librairie partagée
- une extension postgreSQL, sous forme d'un fichier *control* et de scripts SQL

Vérifiez que ces deux éléments sont bien installés pour votre version de PostgreSQL. Leur emplacement dépend du système d'exploitation et de la méthode d'installation. Si l'outil `pg_config` est installé, il est possible de retrouver ces fichiers grâce aux commandes suivantes:

```
ls $(pg_config --pkglibdir)/pgsodium.so
ls $(pg_config --sharedir)/extension/pgsodium.control
```

Sinon, le module `pgsodium.so` est installé usuellement dans `/usr/lib64/pgsql/`, `/usr/pgsql-<PG_VERSION>/lib/` ou `/usr/lib/postgresql/<PG_VERSION>/lib/pg`.

L'extension est quant à elle installée usuellement dans `/usr/share/pgsql/extension/`, `/usr/pgsql-<PG_VERSION>/share/extension/pgsodium.control`, ou `/usr/share/postgresql/<PG_VERSION>/extension/pgsodium.control`.

Observez que le module est bien lié à la librairie libsodium:

```
ldd $(pg_config --pkglibdir)/pgsodium.so | grep sodium
```

2.4 INSTALLATION DANS UNE BASE DE DONNÉE

- création de la base `nacl`
- installation de l'extension dans la base
- rapide découverte des fonctions de pgsodium

Une fois pgsodium installé, il est possible d'installer l'extension dans n'importe quelle base de données. Cette dernière vous permet d'utiliser toutes les fonctions cryptographiques exportées de la libsodium directement en SQL : générer des clés, chiffrer, signer, etc.

Pour la suite de l'atelier, créez une base nommée `nacl` et chargez-y l'extension `pgsodium`.

```
sudo -iu postgres
createdb nacl
psql -c 'create extension pgsodium' nacl
```

Il est possible dans `psql` de lister toutes les fonctions installées par l'extension grâce à la commande `\dx+ pgsodium`. On retrouve par exemple ces quelques fonctions :

```
pgsodium.crypto_aead_det_encrypt(bytea,bytea,bytea,bytea)
pgsodium.crypto_aead_det_decrypt(bytea,bytea,bigint,bytea,bytea)
pgsodium.crypto_aead_det_noncegen()

pgsodium.crypto_aead_ietf_encrypt(bytea,bytea,bytea,bigint,bytea)
pgsodium.crypto_aead_ietf_decrypt(bytea,bytea,bytea,bigint,bytea)
pgsodium.crypto_aead_ietf_noncegen()

pgsodium.crypto_box_seal(bytea,bytea)
pgsodium.crypto_box_open(bytea,bytea,bytea,bytea)

pgsodium.crypto_sign(bytea,bytea)

pgsodium.crypto_hash_sha512(bytea)
```

Voici un exemple de création d'une clé symétrique et de son utilisation, similaire à ce qu'il est possible de faire avec l'extension `pgcrypto`:

```
nacl=# -- création de la clé
nacl=# SELECT pgsodium.crypto_aead_ietf_keygen() AS privk \gset
```

```
nacl=# SELECT :'privk' AS "secret key";
                              secret key
----------------------------------------------------------------------
 \x705a47eaf845ee4ffdb2fce04234cfa2dbbcdda03eaae4582889faf2bde85a90

nacl=# -- création d'un nonce, nécessaire pour le chiffrement
nacl=# SELECT pgsodium.crypto_aead_ietf_noncegen() AS msg_nonce \gset
nacl=# SELECT :'msg_nonce' AS nonce;
           nonce
----------------------------
 \xd2436ae614a1d7a133864f96

nacl=# -- chiffrement en utilisant la clé et le nonce associé à la donnée
nacl=# SELECT pgsodium.crypto_aead_ietf_encrypt(
    'secret'::bytea,
    NULL,
    :'msg_nonce'::bytea,
    :'privk'::bytea
) AS encrypted_msg \gset
nacl=# SELECT :'encrypted_msg' AS "encrypted message";
               encrypted message
------------------------------------------------
 \x58ba578bc099d0350c30d1fa9e5874d9c278e4cf782f

nacl=# -- déchiffrement grâce à la clé et au nonce
nacl=# SELECT pgsodium.crypto_aead_ietf_decrypt(
    :'encrypted_msg'::bytea,
    NULL,
    :'msg_nonce',
    :'privk'
) AS "decrypted message";
 decrypted message
-------------------
 \x736563726574

nacl=# -- déchiffrement grâce à la clé et au nonce et conversion en texte
nacl=# SELECT convert_from(pgsodium.crypto_aead_ietf_decrypt(
        :'encrypted_msg'::bytea,
        NULL,
        :'msg_nonce',
        :'privk'
    ),
'utf-8') AS "decrypted message";
 decrypted message
-------------------
 secret
```

Un autre exemple, utilisant des clé asymétrique, est disponible dans le fichier README du projet: https://github.com/michelp/pgsodium#simple-public-key-encryption-with-crypto_box

---

2.5 CONFIGURATION DU MODULE

- le module `pgsodium` est optionnel
- à positionner dans le paramètre `shared_preload_libraries`
- permet de créer des `SECURITY LABEL` utiles à TCE
- permet de récupérer et conserver en mémoire la *primary server secret key*
  - la mère de toutes les clés
  - un paramètre de configuration: `pgsodium.getkey_script`

Le module `pgsodium` est optionnel. Il n'est nécessaire que si vous souhaitez utiliser TCE ou administrer vos clés depuis votre instance. Son rôle est double.

Le premier est de se déclarer auprès de l'instance comme fournisseur de label de sécurités. Cette démarche ne se fait qu'en C depuis un module et permet ensuite d'utiliser les ordres SQL `SECURITY LABEL FOR pgsodium ON [...]`. Nous découvrirons ces ordres dans les chapitres suivants pour différentes tâches d'administration de TCE.

La seconde fonction de ce module est de charger en mémoire une *primary server secret key* lors du démarrage de l'instance. En cas d'échec lors de la récupération de cette clé, l'instance refusera de démarrer. Cette clé permet de ne plus avoir à fournir de clé de chiffrement explicitement aux fonctions cryptographiques, ni de les stocker en base, évitant ainsi tous les risques de fuites de ces deux pratiques. Nous revenons précisément sur cette mécanique plus loin dans cet atelier.

Le paramètre `pgsodium.getkey_script` permet d'indiquer le chemin vers le script à exécuter pour récupérer cette clé et la retourner au format hexadécimal. Plusieurs exemples de scripts sont disponibles dans le dépôt de pgsodium: https://github.com/michelp/pgsodium/tree/main/getkey_scripts. Par défaut, le script est recherché à l'emplacement `$SHAREDIR/extension/pgsodium_getkey`.

Il est recommandé de ne jamais stocker cette clé sur le serveur lui même. `getkey_script` permet par exemple de récupérer la clé depuis un KMS sans jamais que celle-ci ne touche le disque. Ainsi, elle est strictement conservée en mémoire, jamais écrite sur disque et l'extension s'assure qu'elle n'est pas accessible en SQL. Elle ne peut être utilisée que par du code C installé sur le serveur, donc `untrusted`, et ne peut être chargée que par un super-utilisateur, typiquement via l'extension `pgsodium` donc.

Pour activer le module, il est nécessaire de:

1. installer un script de chargement pour la *primary server secret key*

2. ajouter `pgsodium` au paramètre `shared_preload_libraries`
3. redémarrer votre instance.

Pour les besoins de l'atelier, nous utilisons pour la première étape un script trivial fourni dans le dépôt de pgsodium: `pgsodium_getkey_urandom.sh`. Ce script crée la clé et la conserve dans un fichier non chiffré à l'intérieur du PGDATA. Il est **très fortement** déconseillé de l'utiliser en production.

Pour plus de facilité, placez ce script à l'emplacement par défaut pointé par `pgsodium.getkey_script`.

```
cd $(pg_config --sharedir)/extension/
wget -O pgsodium_getkey https://raw.githubusercontent.com/michelp/pgsodium/main/\
  getkey_scripts/pgsodium_getkey_urandom.sh
chmod +x pgsodium_getkey
```

Pour rappel, en production, il est recommandé de ne **PAS** stocker cette clé localement et d'utiliser un script capable de la récupérer depuis (par exemple) un KMS.

Pensez à faire charger le module au démarrage et à redémarrer l'instance. Par exemple, avec une installation sous Rocky 8:

```
sudo -iu postgres psql -c 'ALTER SYSTEM SET shared_preload_libraries TO pgsodium'
systemctl restart postgresql-13
```

Si l'ensemble est configuré correctement vous devriez trouver le fichier contenant de la clé à l'emplacement `$PGDATA/pgsodium_root.key`:

```
cat "$PGDATA"/pgsodium_root.key
```

De plus, un message de confirmation apparaît au démarrage dans les journaux d'activité:

```
~# grep pgsodium postgresql.log
LOG:  pgsodium primary server secret key loaded
```

3/ TCE

Présentation et utilisation de la fonctionnalité *TCE* de pgsodium.

3.1 PRÉSENTATION DE LA MAQUETTE

Une table utilisateur avec comme colonnes:

- un identifiant
- le nom
- le téléphone
- le numéro de carte de paiement chiffré
- le cryptogramme visuel à 3 chiffres chiffré
- la date d'expiration de la carte de paiement

Dans la base `nacl`, créez un schéma `workshop` et la table `encrypted_credit_cards` :

```
CREATE SCHEMA workshop;

SET search_path TO workshop, pg_catalog, pgsodium;

CREATE TABLE workshop.encrypted_credit_cards (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name TEXT,
    phone_number TEXT,
    number TEXT,
    cryptogram TEXT,
    expiration_date DATE
);
```

3.2 CRÉATION DES CLÉS

- la fonction `pgsodium.create_key()` crée une nouvelle clé
- appelée sans arguments, la clé est dérivée de la *primary server secret key*
- possibilité de préciser le type, des propriétés et/ou d'importer une clé existante
- pour notre atelier, on crée trois clés avec une date d'expiration

Une fois l'extension installée dans votre base, cette dernière crée la table `pgsodium.key` où sont répertoriées les clés qui pourront être utilisées par les fonctions de pgsodium. La fonction `pgsodium.create_key()` permet d'ajouter des clés à cette table. Elles peuvent être de deux natures: dérivées ou externes.

Les clés dérivées sont créées par dérivation de la *primary server secret key*. À partir de cette seule clé chargée en mémoire, pgsodium est capable de dériver autant de clés que nécessaire. Chaque génération de clé nécessite quatre informations:

- la *primary server secret key*, accessible en mémoire donc
- un identifiant de clé sur 64 bits
- la taille de clé désirée (entre 128 et 512 bits, 16 et 64 octets)
- le *contexte* associé à la clé, une simple chaîne de caractère à votre discrétion.

La table conserve l'identifiant de chaque clé, sa taille et son contexte, mais jamais la clé elle même, ni la *primary server secret key*. Par la suite, en fournissant un simple identifiant de clé, pgsodium est capable de re-dériver la clé et ainsi chiffrer / déchiffrer vos données sans jamais faire apparaître la clé réelle de chiffrement. Aucune interception de requête ou lecture dans les tables ne permet de déchiffrer la donnée sans la *primary server secret key*.

Contrairement aux clés dérivées, les clés externes sont bien importées dans la table, mais elles sont automatiquement chiffrées à partir d'une clé dérivée. Cette fonctionnalité n'est pas utilisée par TCE, elle n'existe que pour d'autres besoins cryptographiques.

La fonction `pgsodium.create_key()` permet donc de créer ou d'importer des clés dans cette table:

- le 1er argument `key_type` spécifie le type de clé utilisé parmi les suivants: `aead-ietf` `aead-det`, `hmacsha512`, `hmacsha256`, `auth`, `shorthash`, `generichash`, `kdf`, `secretbox`, `secretstream` et `stream_xchacha20`. Un seul choix est supporté pour TCE: `aead-det` ;
- le 2nd argument `name` permet de donner un nom à la clé, plus facile à à retenir que son ID ;

- les arguments suivants, `raw_key`, `raw_key_nonce` et `parent_key`, permettent d'importer des clés externes. Elles sont alors chiffrées avec la clé désignée par `parent_key` ;
- le 6e argument `key_context` correspond au contexte, par défaut `pgsodium`
- le 7e argument permet de spécifier une date d'expiration

Nota Bene : L'utilisation de clés externes (*raw keys*) ne permet pas de se protéger d'un super-utilisateur malveillant, puisque celles-ci sont déchiffrées côté serveur.

Dans le cadre de notre atelier, nous créons trois clés, chacune avec une date d'expiration, dont une déjà expirée. Nous reviendrons sur ces dates dans le chapitre consacré à la rotation des clés.

```
SELECT * FROM create_key(expires => date_trunc('month', now()));
SELECT * FROM create_key(expires => date_trunc('month', now()) + '1 month');
SELECT * FROM create_key(expires => date_trunc('month', now()) + '2 month');
```

Nous laissons l'importation d'une clé externe à l'exercice du lecteur.

3.3 CHIFFRER LE NUMÉRO DE CARTE DE PAIEMENT

- utilisation de `SECURITY LABEL` pour déclarer une colonne chiffrée
- `SECURITY LABEL FOR pgsodium ON COLUMN ... IS ENCRYPT WITH ...`
- une seule clé pour toute la table…
- … ou une clé par ligne
- création automatique d'un **trigger** pour le chiffrement à l'écriture
- création automatique d'une **vue** pour le déchiffrement à la lecture

Lisez attentivement ce chapitre avant de le mettre en application, plusieurs commandes ne sont utiles qu'aux explications et ne doivent pas être reproduites !

Afin de chiffrer le numéro de la carte bancaire, pgsodium nécessite de marquer la colonne `number` avec un label de sécurité, par exemple comme ceci :

```
SECURITY LABEL FOR pgsodium ON COLUMN workshop.encrypted_credit_cards.number IS
  'ENCRYPT WITH KEY ID 1552b635-82d1-427d-a469-e40413c42eb2';
```

Dans ce cas, tous les numéros de carte bancaire de la table sont chiffrés avec la même clé.

Pour notre atelier, nous souhaitons utiliser une nouvelle clé chaque mois. Nous ajoutons donc une colonne `key_id` à la table, qui contient pour chaque ligne l'identifiant de la clé de chiffrement utilisée pour chiffrer le champs `number` :

```
ALTER TABLE workshop.encrypted_credit_cards ADD COLUMN key_id uuid;
```

On peut alors utiliser la syntaxe suivante pour le label de sécurité :

```
SECURITY LABEL FOR pgsodium ON COLUMN workshop.encrypted_credit_cards.number IS
  'ENCRYPT WITH KEY COLUMN key_id';
```

Mais nous pouvons aller encore un peu plus loin. Un attaquant ayant les droits en écriture sur cette table pourrait s'approprier la carte de quelqu'un d'autre en recopiant la valeur chiffrée vers une autre ligne, sans avoir à la déchiffrer.

Afin de s'en prémunir, il est possible d'associer au chiffrement une ou plusieurs colonnes non chiffrées. Une liaison forte est alors créée entre la valeur chiffrée et ces données associées. La valeur chiffrée ne peut plus être déchiffrée si les valeurs associées ne sont pas les mêmes.

Voici alors le label de sécurité complet que nous utilisons:

```
SECURITY LABEL FOR pgsodium ON COLUMN workshop.encrypted_credit_cards.number IS
  'ENCRYPT WITH KEY COLUMN key_id ASSOCIATED (id, name)';
```

Notez que la conversion vers le type TEXT d'une colonne utilisée pour les données authentifiées supplémentaires (*associated data*) doit être **déterministe**. Par exemple, la conversion de la date d'expiration n'est pas déterministe car dépend de la *locale* et du format de date utilisé dans la session. Elle ne peut donc pas être utilisée.

C'est aussi un exemple de fonctionnalité qui n'est pas supportée par la vénérable extension pgcrypto.

La création d'un SECURITY LABEL déclenche automatiquement un EVENT TRIGGER qui s'occupe notamment de créer un trigger BEFORE INSERT OR UPDATE sur la table pour cette colonne. En voici le code légèrement modifié pour notre exemple:

```
CREATE OR REPLACE
FUNCTION workshop.encrypted_credit_cards_encrypt_secret_number()
RETURNS trigger
LANGUAGE plpgsql
AS $function$
BEGIN
  new.number =
    CASE WHEN new.number IS NULL THEN NULL
    ELSE
      CASE WHEN new.key_id IS NULL THEN NULL
      ELSE pg_catalog.encode(                                          -- (4)
        pgsodium.crypto_aead_det_encrypt(                              -- (3)
          pg_catalog.convert_to(new.number, 'utf8'),                   -- (1)
          pg_catalog.convert_to(new.id::text || new.name::text, 'utf8'), -- (2)
          new.key_id::uuid,                                            -- (3)
          NULL -- pas de nonce configuré
        ),
        'base64'
      )
      END
    END;
RETURN new;
END
$function$
```

Son fonctionnement est simple:

1. la valeur de la colonne number est convertie en bytea
2. les colonnes associées sont converties en texte, concaténées, et le résultat est converti en bytea
3. le tout est chiffré avec la clé contenue dans la colonne key_id

4. le résultat est encodée en `base64` afin de le stocker dans une colonne de type `TEXT`.

Des travaux sont actuellement en cours pour simplifier la génération de ce trigger et de son code.

L'algorithme de chiffrement utilisé est XChaCha20-Poly1305[1], conçu par Daniel J. Bernstein, principal auteur de NaCl, comme vu précédemment. Cet algorithme est robuste, performant et non breveté.

Il nous reste un peu de travail avant de pouvoir insérer nos premières lignes : le chiffrement du cryptogramme visuel de la carte de paiement.

---

[1] https://en.wikipedia.org/wiki/ChaCha20-Poly1305

3.4 CHIFFRER LE CRYPTOGRAMME VISUEL

- le cryptogramme visuel est composé de trois chiffres
- peu de combinaisons possibles !
- nécessite un `nonce` pour que deux mêmes cryptogrammes soient chiffrés différemment
- le `nonce` doit être lui-même de qualité cryptographique

Pour chiffrer le cryptogramme visuel, nous devons utiliser un *nonce* afin d'augmenter la sécurité. En effet, le nombre de cryptogrammes sur trois chiffres est petit, la probabilité de collision est donc importante. Sans nonce, tous les cryptogrammes identiques auraient la même valeur chiffrée en base, ce qui est une information et une faiblesse trop importante pour ce type de donnée.

Pour ce faire, ajoutez une colonne `nonce` à la table, avec comme valeur par défaut un nonce `aead_det` généré par pgsodium.

```
ALTER TABLE workshop.encrypted_credit_cards ADD COLUMN
  nonce BYTEA DEFAULT pgsodium.crypto_aead_det_noncegen();
```

La fonction `crypto_aead_det_noncegen()` permet de générer un *nonce* aléatoire de qualité cryptographique en utilisant la fonction `randombytes_buf()` de *libsodium* qui elle-même repose sur l'appel système `getrandom()` (ou équivalent). Cette fonction, comme la plupart des autres fonctions de pgsodium, utilise par ailleurs un allocateur mémoire spécifique qui permet d'effacer la zone mémoire utilisée au plus tôt.

Vous pouvez désormais déclarer la colonne comme chiffrée avec un label de sécurité. Déclarez y la colonne porteuse du *nonce* avec la syntaxe `NONCE [...]`:

```
SECURITY LABEL FOR pgsodium ON COLUMN workshop.encrypted_credit_cards.cryptogram IS
  'ENCRYPT WITH KEY COLUMN key_id NONCE nonce';
```

Observez le nouveau trigger créé :

```
# \d workshop.encrypted_credit_cards
[...]
Triggers:
   encrypted_credit_cards_encrypt_secret_trigger_cryptogram BEFORE INSERT OR UPDATE
        OF cryptogram ON workshop.encrypted_credit_cards FOR EACH ROW
      EXECUTE FUNCTION workshop.encrypted_credit_cards_encrypt_secret_cryptogram()
   encrypted_credit_cards_encrypt_secret_trigger_number BEFORE INSERT OR UPDATE
        OF number ON encrypted_credit_cards FOR EACH ROW
```

```
EXECUTE FUNCTION encrypted_credit_cards_encrypt_secret_number()
```

3.5 GESTION DE LA VUE DÉCHIFFRÉE

- l'event trigger sur les `SECURITY LABEL` crée aussi une vue
- permet de lire la donnée déchiffrée en conservant des requêtes simples
- le nom de la vue est configurable

Le même `EVENT TRIGGER` précédent est aussi responsable de la création ou de la mise à jour d'une vue associée à la table. Par rapport à la table, la vue ajoute pour chaque colonne chiffrée son équivalent déchiffré.

L'intérêt de cette vue est de pouvoir consulter les données déchiffrées sans avoir à écrire des requêtes manipulant des fonctions complexes et ainsi garder une certaine lisibilité. Nous abordons aussi un peu plus loin comment sont gérés les droits dessus.

Par défaut, la vue porte le même nom que la table, préfixé par `decrypted_`. Dans notre exemple, la vue créée se nomme donc `decrypted_encrypted_credit_cards`.

Il est possible de modifier le nom de cette vue grâce à un autre label de sécurité, cette fois-ci déposé sur la table elle-même, avec la syntaxe suivante: `SECURITY LABEL FOR pgsodium ON TABLE ... IS 'DECRYPT WITH VIEW...'`. Notez cependant que l'event trigger ne prends pas l'initiative de supprimer l'ancienne vue à votre place.

Configurez TCE pour que la vue déchiffrée se nomme `workshop.credit_cards`.

```
DROP VIEW workshop.decrypted_encrypted_credit_cards;
SECURITY LABEL FOR pgsodium ON TABLE workshop.encrypted_credit_cards IS
  'DECRYPT WITH VIEW workshop.credit_cards';
```

Observez le code de la vue générée (ici modifié pour notre exemple):

```
CREATE OR REPLACE VIEW workshop.credit_cards AS
SELECT id, name, phone_number, number, cryptogram, expiration_date, key_id,
  -- [...]
  CASE WHEN [...]
  END AS decrypted_number,
  -- [...]
  CASE WHEN cryptogram IS NULL THEN NULL::text
  ELSE
    CASE WHEN key_id IS NULL THEN NULL::text
    ELSE convert_from(                                    -- (4)
      pgsodium.crypto_aead_det_decrypt(                   -- (3)
        decode(cryptogram, 'base64'::text),               -- (1)
        convert_to(id::text || name::text, 'utf8'::name), -- (2)
```

```
        key_id,                                          -- (3)
        encrypted_credit_cards.nonce                     -- (3)
      ),
      'utf8'::name
    )
    END
  END AS decrypted_cryptogram,
  -- [...]
FROM encrypted_credit_cards
```

Nous retrouvons ici le pendant de la fonction précédente pour le déchiffrement :

1. décodage `base64` de la donnée chiffrée
2. les colonnes associés sont converties en texte et concaténées
3. le tout est déchiffré en utilisant la clé contenue dans la colonne `key_id`
4. l'ensemble est converti en type `TEXT`

3.6 ÉCRITURE ET LECTURE DES DONNÉES

- chiffrement transparent à l'insertion des données
- données chiffrées lisibles en table
- données déchiffrées lisibles depuis la vue
- vue peu performante pour les traitements en masse

Nous pouvons désormais insérer des lignes dans notre table. Voici une requête générant 300.000 lignes pour les besoins de l'atelier:

```
INSERT INTO workshop.encrypted_credit_cards
  (name, phone_number, number, cryptogram, expiration_date, key_id)
WITH keys AS (
  SELECT row_number() OVER () - 1 AS n, id
  FROM pgsodium.key
  LIMIT 3
)
SELECT 'nom '||i::text,                 -- nom
  '0' || (607000000+i)::text,           -- tel
  '0123 4567 89'
    || lpad((i/10000)::text, 2, '0') || ' '
    || lpad((i%10000)::text, 4, '0'),   -- numéro de carte
  (random()*999+1)::int,                -- cryptogramme
  current_date + (random()*365*2)::int, -- date d'expiration
  keys.id                               -- clé de chiffrement
FROM generate_series(0,300000) AS s(i)
JOIN keys on keys.n = s.i/100000
```

À la lecture, nous constatons que les données sont bien chiffrées dans la table:

```
nacl=# SELECT * FROM workshop.encrypted_credit_cards ORDER BY id LIMIT 2 \gx
-[ RECORD 1
 ↪  ]---+------------------------------------------------------------------------
id              | 1
name            | nom 0
phone_number    | 0607000000
number          |
 ↪  uHMB4oIHF0uBSuIcPH6fPTI8QhA5lR28ZEkcbm/Hek5BxPH72ETTFYOB4fgjEy1j4aaH
cryptogram      | ZNubwcONJ3YVhzmMW3A5tj7b24U3k3EtnBSTh7DdabEuOg4=
expiration_date | 2024-10-05
key_id          | dfae7e6c-d350-4b33-8603-4d6956b0b0ba
nonce           | \x2a6723ef9f5d9990b53a2d735d3393e0
```

```
-[ RECORD 2
 ↪  ]---+------------------------------------------------------------------------
id              | 2
name            | nom 1
phone_number    | 0607000001
number          |
 ↪  wjbF0v6Qwsf0A4wRMan1nKoIQgOXw0hlPF39NCTx/kfYdpzJjWi4yhiZK6ZpvaaynOst
cryptogram      | WLtAI6xTbvbO78dM0VSpYz+6S1kzLJWfBGR7npATIYNbTMw=
expiration_date | 2024-10-04
key_id          | dfae7e6c-d350-4b33-8603-4d6956b0b0ba
nonce           | \xbeb065bd3a15c733edf39a4e1c7d32ba
```

Et la lecture de la vue affiche correctement les données chiffrées:

```
nacl=# SELECT * FROM workshop.credit_cards ORDER BY id LIMIT 2 \gx
-[ RECORD 1
 ↪  ]--------+------------------------------------------------------------------
id                   | 1
name                 | nom 0
phone_number         | 0607000000
number               |
 ↪  uHMB4oIHF0uBSuIcPH6fPTI8QhA5lR28ZEkcbm/Hek5BxPH72ETTFYOB4fgjEy1j4aaH
decrypted_number     | 0123 4567 8900 0000
cryptogram           | ZNubwcONJ3YVhzmMW3A5tj7b24U3k3EtnBSTh7DdabEuOg4=
decrypted_cryptogram | 423
expiration_date      | 2024-10-05
key_id               | dfae7e6c-d350-4b33-8603-4d6956b0b0ba
nonce                | \x2a6723ef9f5d9990b53a2d735d3393e0
-[ RECORD 2
 ↪  ]--------+------------------------------------------------------------------
id                   | 2
name                 | nom 1
phone_number         | 0607000001
number               |
 ↪  wjbF0v6Qwsf0A4wRMan1nKoIQgOXw0hlPF39NCTx/kfYdpzJjWi4yhiZK6ZpvaaynOst
decrypted_number     | 0123 4567 8900 0001
cryptogram           | WLtAI6xTbvbO78dM0VSpYz+6S1kzLJWfBGR7npATIYNbTMw=
decrypted_cryptogram | 983
expiration_date      | 2024-10-04
key_id               | dfae7e6c-d350-4b33-8603-4d6956b0b0ba
nonce                | \xbeb065bd3a15c733edf39a4e1c7d32ba
```

Ci-après quelques statistiques sur ces données afin de bien comprendre notre ensemble de test et sa répartition:

```
nacl=# SELECT key_id, count(*) FROM workshop.credit_cards GROUP BY key_id;
               key_id                | count
```

```
--------------------------------------+--------
 068eb8af-4576-4625-9c5f-126795a93cde | 100000
 dfae7e6c-d350-4b33-8603-4d6956b0b0ba | 100000
 aef026e9-226b-4583-aa4f-e12cd0f8d67c | 100000
(3 rows)
```

Nous avons bien trois clés utilisées avec une distribution équitable des 300 000 lignes. Étudions la répartition des valeurs chiffrées et déchiffrées. Nous en profitons aussi pour mesurer les performances des fonctions de déchiffrement:

```
nacl=# ALTER SYSTEM SET track_functions TO 'all';

nacl=# SELECT pg_reload_conf();

nacl=# SELECT * FROM pg_catalog.pg_stat_user_functions ;
 funcid | schemaname | funcname | calls | total_time | self_time
--------+------------+----------+-------+------------+-----------
(0 rows)

nacl=# SELECT
  count(DISTINCT decrypted_number)     AS "#_cc",
  count(DISTINCT number)               AS "#_enc_cc",
  count(DISTINCT decrypted_cryptogram) AS "#_crypto",
  count(DISTINCT cryptogram)           AS "#_enc_crypto"
FROM workshop.credit_cards;
  #_cc  | #_enc_cc | #_crypto | #_enc_crypto
--------+----------+----------+--------------
 300000 |   300000 |     1000 |       300000
(1 row)

Time: 17635.791 ms (00:17.636)
```

Nous observons:

- 300 000 numéros de cartes distincts, chiffrés et non chiffrés, ce qui est attendu
- seulement 1000 cryptogrammes visuels distincts possibles, mais 300 000 chiffrés différents (grâce au *nonce*), ce qui est voulu
- l'utilisation de la vue pour déchiffrer en masse est lente !

Cette lenteur est due à l'appel de fonction qui va rechercher la clé correspondante et ses méta données individuellement pour chaque ligne, comportement similaire à une jointure *Nested Loop* sur 300 000 lignes. Attardons-nous sur cette mauvaise performance:

```
nacl=# SELECT l.lanname, s.*
FROM pg_catalog.pg_stat_user_functions s
JOIN pg_catalog.pg_proc p ON s.funcid = p.oid
```

```
JOIN pg_catalog.pg_language l ON l.oid = p.prolang;

 lanname | funcid | schemaname |        funcname         | calls  | total_time |
 ↪  self_time
---------+--------+------------+-------------------------+--------+------------+----
 ↪  -------
 c       |  24684 | pgsodium   | crypto_aead_det_decrypt | 600000 |   1310.196 |
 ↪  1310.196
 plpgsql |  24750 | pgsodium   | crypto_aead_det_decrypt | 600000 |  14336.636 |
 ↪  13026.439
(2 rows)
```

Nous retrouvons bien 600 000 appels aux fonctions de pgsodium, soit deux appels par lignes, pour les colonnes `number` et `cryptogram`.

Nous vérifions bien que sur ces grosses 17 secondes, plus de 14 sont perdues dans ces fonctions, dont 13 dans la seule fonction PL/PgSQL responsable de récupérer les meta données des clés avant de faire appel à la fonction de déchiffrement implémentée en C.

Une ré-écriture de cette requête sans passer par la vue permet de court-circuiter l'appel de la fonction PL/PgSQL, et fait tomber son temps d'exécution total à 2 secondes, dont une seconde passée dans la fonction C `crypto_aead_det_decrypt`. L'exercice est laissé à la discrétion du lecteur.

Notez que ce problème de performance était aussi présent lors de l'insertion en masse des données, cette fois-ci au niveau du trigger.

Des discussions et réflexions sur le sujet sont en cours avec l'auteur de pgsodium. Ce point pourrait faire l'office d'une amélioration future de l'extension.

4/ Maintenances

- gestion des droits
- rotation des clés

4.1 GESTION DES DROITS SUR LES DONNÉES

- seul le propriétaire ou un super-utilisateur accèdent à une table ou un vue
- rôles spécifiques à pgsodium
- SECURITY LABEL sur un rôle

Par défaut, PostgreSQL n'autorise que le propriétaire et les super-utilisateurs à accéder à une table. Il est nécessaire d'accorder spécifiquement des droits pour que d'autres rôles puissent y accéder.

Créez par exemple un nouveau rôle `anonymous` et tentez de lire nos tables, vues précédentes, clés ou d'exécuter des fonctions cryptographiques.

```
nacl=# CREATE ROLE anonymous LOGIN;
CREATE ROLE

nacl=# SET SESSION AUTHORIZATION anonymous;

nacl=> SELECT * FROM workshop.encrypted_credit_cards LIMIT 0;
ERROR:  permission denied for schema workshop

nacl=> RESET SESSION AUTHORIZATION;

nacl=# GRANT ALL ON SCHEMA workshop TO anonymous;

nacl=# SET SESSION AUTHORIZATION anonymous;

nacl=> SELECT * FROM workshop.encrypted_credit_cards LIMIT 0;
ERROR:  permission denied for table encrypted_credit_cards

nacl=> SELECT * FROM workshop.credit_cards LIMIT 0;
ERROR:  permission denied for view credit_cards
```

Piochez une valeur et sa clé au hasard dans la table `workshop.encrypted_credit_cards` et tentez de la déchiffrer avec l'utilisateur `anonymous` :

```
nacl=# SELECT decode(cryptogram, 'base64') AS enc_crypto,
    cc.nonce AS nonce, k.key_id AS key_id
FROM workshop.encrypted_credit_cards cc
JOIN pgsodium.key k ON (cc.key_id = k.id)
LIMIT 1 \gset

nacl=# SET SESSION AUTHORIZATION anonymous;
```

```
nacl=> SELECT pgsodium.crypto_aead_det_decrypt(
        :'enc_crypto'::bytea,
        ''::bytea,
        :key_id, 'pgsodium'::bytea, :'nonce'::bytea
      );
ERROR:  permission denied for function crypto_aead_det_decrypt
```

Par défaut, pgsodium crée trois rôles qui peuvent être utilisés comme des groupes, permettant ainsi d'accorder ou retirer ces rôles et leurs droits en utilisant GRANT ou REVOKE:

- `pgsodium_keyiduser` peut :
  - générer de nouvelles clés, uniquement par dérivation
  - utiliser les clés générées par dérivation, uniquement via l'API
- `pgsodium_keyholder` peut :
  - faire tout ce que peut faire `pgsodium_keyiduser`
  - accéder en lecture seule à la table `pgsodium.key`
- `pgsodium_keymaker` peut :
  - générer n'importe quel type de clés
  - importer des clés (non dérivées) dans la table `pgsodium.key`
  - modifier ou supprimer des clés dans la table `pgsodium.key`

Le rôle `pgsodium_keyholder` est déprécié, et seuls les deux autres sont réellement utiles d'un point de vue utilisateur.

Pgsodium propose également de déposer un label de sécurité sur les rôles, permettant ainsi de désigner qui peut avoir accès à quelles tables. Le format est le suivant:

```
SECURITY LABEL FOR pgsodium ON ROLE ... IS
  'ACCESS <fully qualified table 1>[, ...]';
```

Il est possible de préciser plusieurs tables dans le label et chaque table **doit** être désignée avec son schéma, même si ce dernier est `public`.

Lors des prochaines modifications sur l'objet, l'extension s'occupe alors via son `EVENT TRIGGER` de positionner les bons droits sur les bon objets, permettant ainsi au rôle concerné de lire ou écrire les données chiffrées de façon transparente.

Créez un utilisateur `cashier` et accordez-lui l'accès à la table `workshop.encrypted_credit_cards`.

```
nacl=# CREATE ROLE cashier LOGIN;
nacl=# GRANT ALL ON SCHEMA workshop TO cashier;
```

```
nacl=# SECURITY LABEL FOR pgsodium ON ROLE cashier IS
  'ACCESS workshop.encrypted_credit_cards';
```

L'objet existant déjà et aucune modification y étant faite, il est nécessaire de demander à pgsodium de réviser l'ensemble de ses vues et des droits y afférant :

```
nacl=# SELECT pgsodium.update_masks();
[...]
```

Cette précédente étape pourrait faire l'office d'une amélioration future de l'extension afin d'automatiser cette étape.

Nous constatons que le rôle `cashier` est désormais membre des rôles `pgsodium_keyiduser` et `pgsodium_keyholder`:

```
nacl=> \du
                         List of roles
     Role name      |     Attributes     |       Member of
--------------------+--------------------+-----------------------
 anonymous          |                    | {}
 cashier            |                    | {pgsodium_keyiduser,pgsodium_keyholder}
[...]
```

Re-piochez une valeur et sa clé au hasard dans votre table et tentez de la déchiffrer:

```
nacl=# SELECT decode(cryptogram, 'base64') AS enc_crypto,
    cc.nonce AS nonce, k.key_id AS key_id
FROM workshop.encrypted_credit_cards cc
JOIN pgsodium.key k ON (cc.key_id = k.id)
LIMIT 1 \gset

nacl=# SET SESSION AUTHORIZATION cashier;

nacl=> SELECT convert_from(pgsodium.crypto_aead_det_decrypt(
        :'enc_crypto'::bytea,
        ''::bytea,
        :key_id, 'pgsodium'::bytea, :'nonce'::bytea
      ), 'utf-8') AS crypto;
 crypto
--------
 710
```

L'utilisateur `cashier` est bien capable de déchiffrer la donnée.

Testez désormais les droits en lecture et écriture du rôle `cashier`.

```
nacl=# \z workshop.credit_cards|encrypted_credit_cards
                                        Access privileges
  Schema  |          Name          | Type  |     Access privileges
----------+------------------------+-------+---------------------------
 workshop | credit_cards           | view  | postgres=arwdDxt/postgres+
          |                        |       | cashier=arwdDxt/postgres
 workshop | encrypted_credit_cards | table |
(2 rows)

nacl=> SELECT * FROM workshop.encrypted_credit_cards LIMIT 1;
ERROR:  permission denied for table encrypted_credit_cards

nacl=> SELECT id AS latest_key_id
FROM pgsodium.valid_key
ORDER BY expires DESC, created DESC
LIMIT 1 \gset

nacl=> INSERT INTO workshop.encrypted_credit_cards
  (name, phone_number, number, cryptogram, expiration_date, key_id)
VALUES ('Léon Musc', '0607080910', '0123 4567 8910 1234', '123', '2050-12-31',
        :'latest_key_id');
ERROR:  permission denied for table encrypted_credit_cards

nacl=> SELECT * FROM workshop.credit_cards LIMIT 1 \gx
-[ RECORD 1
 ↪  ]--------+------------------------------------------------------------------------
id                   | 43777
name                 | nom 43776
phone_number         | 0607043776
number               |
 ↪  nWPwRfe4ywv4u1TpHzpJQnMlSL1PDffGKF4Yr62toYnWg1/FDacBmkQGo5883FAottmh
decrypted_number     | 0123 4567 8904 3776
cryptogram           | TmyQM6B6Jfp0hH6pycfxvYinM6mzIQi6O8nyLhQnML4K3T0=
decrypted_cryptogram | 753
expiration_date      | 2024-08-25
key_id               | dfae7e6c-d350-4b33-8603-4d6956b0b0ba
nonce                | \x299055c96de6ba2422061db9bbd87f1b
```

Nous observons que `cashier` a bien le droit d'exécuter les fonctions de chiffrement, mais n'a aucun droit sur la table, ni en lecture ni en écriture, seulement sur la vue. Or, notez que PostgreSQL accepte les écritures sur les vues simples (sans agrégats, jointures, etc). Testez donc une écriture au travers de la vue.

```
nacl=> INSERT INTO workshop.credit_cards
  (name, phone_number, number, cryptogram, expiration_date, key_id)
VALUES ('Léon Musc', '0607080910', '0123 4567 8910 1234', '123', '2050-12-31',
```

```
        :'latest_key_id');
INSERT 0 1

nacl=> SELECT name, number, decrypted_number, cryptogram, decrypted_cryptogram
       FROM workshop.credit_cards
       WHERE name ~* 'léon' \gx
-[ RECORD 1
 ↪  ]--------+-----------------------------------------------------------------
name                 | Léon Musc
number               |
 ↪  56YRJXEBRGouhs337Nqkp03fK8VJt35FUpo16pgnol70T0rpY5qz6xvQtpaJ0Yh6d4gM
decrypted_number     | 0123 4567 8910 1234
cryptogram           | XhXUqvh06aCmghR6aaquia/mo70qkwUdupxOJxVJAWcwV/c=
decrypted_cryptogram | 123
```

Le rôle `cashier` a donc bien les droits suffisants pour lire et écrire de façon transparente dans la table.

4.2 ROTATION DES CLÉS ?

- peut être nécessaire dans le cadre de certains standards (p. ex. PCI DSS?)
- le besoin réel dépend de l'algorithme de chiffrement et de l'utilisation
- procédure d'exemple très exagérée:
  - création et utilisation d'une nouvelle clé tous les mois
  - chaque clé est considérée valide pendant 3 mois
  - re-chiffrement des données pour toute clé invalide

La rotation des clés de chiffrement est un sujet en dehors de notre expertise. Néanmoins, plusieurs facteurs peuvent affecter la durée de vie raisonnable d'une clé :

- l'environnement où sont stockées les données
- le changement de personnel
- son implémentation logicielle ou hardware (HSM)
- la résistance/force de l'algorithme de chiffrement
- ses limitations d'utilisation (p. ex. le volume de *nonce* utilisable)
- le volume de données stocké **et/ou** le nombre de données chiffrées
- le volume de données concerné en cas de compromission d'une clé
- la durée de vie de l'information à protéger
- la charge induite par la procédure de rotation
- …

Mais surtout, l'intérêt principal de la mise en œuvre d'une politique de rotation de clé est d'éprouver et maintenir cette procédure très importante et requise en cas de compromission d'une clé.

L'exemple de rotation proposé ici est caricatural, mais permet de poursuivre l'étude de TCE et d'ébaucher quelques pistes:

- création d'une nouvelle clé valide pendant 3 mois
- utilisation de la clé comme valeur par défaut de la colonne `key_id`
- rotation des clés devenues invalides une fois par mois

Mettez en œuvre cette procédure.

1. création d'une nouvelle clé

```
SELECT id AS latest_key_id FROM pgsodium.create_key(
  expires => date_trunc('month', now()) + '3 months'
) \gset
```

2. mise à jour de la clé par défaut utilisée dans la table et la vue

```
ALTER TABLE workshop.encrypted_credit_cards
  ALTER COLUMN key_id SET DEFAULT :'latest_key_id';

ALTER VIEW workshop.credit_cards
  ALTER COLUMN key_id SET DEFAULT :'latest_key_id';
```

3. rotation des clés expirées:

```
UPDATE workshop.credit_cards c
SET
  number = c.decrypted_number,
  cryptogram = c.decrypted_cryptogram,
  key_id = DEFAULT
FROM pgsodium.key k
WHERE k.expires IS NOT NULL
  AND k.expires < current_timestamp
  AND k.id = c.key_id;
```

Cet exemple permet donc de ne conserver une clé de chiffrement que 3 mois maximum, un tiers de la table est ré-écrit chaque mois. Si une clé est compromise, un tiers des données sont concernées par le risque.

Encore une fois, cet exemple est très artificiel et inadapté à la plupart des situations. En premier lieu, le nombre de clés est trop faible. L'utilisation de la vue pour la mise à jour est aussi sous-performante, comme expliqué précédemment (l'écriture d'une requête plus performante est laissée comme exercice au lecteur). Aussi, notez que cet exemple ne tient pas compte de la rétention des données. Certains cas d'utilisation nécessitent la mise en œuvre d'une politique de purge des données. Enfin, la mise à jour chaque mois de la clé par défaut impose un verrou exclusif sur la table, certes court.

Pour ce dernier point, il est possible d'utiliser une fonction SQL `IMMUTABLE` qui retourne directement la clé comme un scalaire et peut être mise à jour une fois par mois sans impacter la table.

5/ Rappels de sécurité

- répond au besoin d'« encryption at rest »
- ne protège pas d'un super-utilisateur malveillant
- appliquer une gestion de rôle et de droit aussi fine que possible
- n'épargne pas le besoin de :
  - forcer le chiffrement des connexions
  - utiliser l'authentification SCRAM
  - mettre en œuvre une politique de sécurité robuste coté système
- attention aux fichiers temporaires sur disque

La fonctionnalité TCE de pgsodium ne répond qu'au besoin d'«encryption at rest» où la donnée doit être chiffrée avant d'être écrite sur disque. L'extension fait tout son possible pour que les clés ne puissent pas fuiter sur disque, ne restent qu'en mémoire et n'y résident que le moins longtemps possible.

Néanmoins, l'extension est impuissante face à un super-utilisateur ou un compte `root` (non confiné par SELinux). Il convient donc d'appliquer coté système toutes les mesures de sécurité nécessaires requises aux données que vous souhaitez protéger:

- attention aux *core dump* en cas de crash où la *primary server secret key* pourrait fuiter
- chiffrement de l'espace de *swap* pour les mêmes raisons [1]
- sécurisation de l'hyperviseur si le serveur est virtuel
- audit des accès
- *SELinux* pour les plus courageux
- …

Coté PostgreSQL, certains points sont à surveiller:

- ne pas utiliser de compte super-utilisateur dans vos applications
- conserver la *primary server secret key* sur un serveur tier (p. ex. KMS)
- forcer l'utilisation de connexions chiffrées en SSL avec le mode `verify-ca`
- gestion fine des droits sur les objets, jusqu'à la colonne, voir même aux lignes (RLS)
- utiliser l'authentification SCRAM

[1] A priori, l'utilisation de la fonction `sodium_malloc()` empêche la clé de se retrouver en *swap*, mais cela reste une bonne pratique de chiffrer celui-ci.

- lorsque c'est possible, pensez aux méthodes cryptographiques destructives dont la donnée ne peut être retrouvée (eg. la famille des SHA)
- les requêtes manipulant beaucoup de données peuvent créer des fichiers temporaires sur disque pour leurs tri, table de hachage, ou simplement la matérialisation d'un set de données. Attention à déchiffrer la donnée le plus tard possible dans votre requête

---

6/ Futur

- projet encore jeune, un seul mainteneur
- nécessite plus de tests automatisés
- manque de relecture et de documentation
- Dalibo débute sa participation au projet
  - accueil positif du PO
  - quelques correctifs et améliorations en cours
- quelques nouvelles fonctionnalités ?

Le projet pgsodium a plusieurs années d'existence, mais l'implémentation TCE proposée ne date que de l'été 2022. Cette partie n'est donc pas encore pleinement mature et nécessite encore un peu d'attention sur les finitions, documentation et performance.

Dalibo investi actuellement du temps de recherche et développement afin de contribuer à l'amélioration de ces différents points et plusieurs patch ont d'ores et déjà été intégrés.

Des discussions sont en cours à propos de la politique de gestion des versions, actuellement trop rapide et publiant des modifications quasi expérimentales. Aussi, il a été question d'une éventuelle séparation de TCE en dehors de pgsodium.

Parmi les travaux envisageables:

- support du chiffrement AES : questionnement autour des performance via l'accélération matériel, mais rotation des clés fortement conseillée [1] !
- optimiser les performances de la vue de déchiffrement
- appliquer les droits aux rôles automatiquement lors de l'application d'un label de sécurité
- exemple d'utilisation du chiffrement de bout en bout
- pouvoir utiliser le nom d'une clé plutôt que son uuid

---

[1] voir à ce propos: https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm

7/ Conclusion

- extension recommandée en remplacement de pgcrypto !
- fonctionnalité TCE naissante, à découvrir
- quelques aspérités à corriger
- peut-être une source d'inspiration pour une implémentation simplifiée

Le projet pgsodium est déjà recommandé en remplacement de pgcrypto:

- couvre le spectre des fonctionnalités (et leurs problèmes) de pgcrypto
- plus de fonctionnalités modernes
- des algorithmes de chiffrement modernes
- libsodium et pgsodium sont maintenus contrairement à pgcrypto

Concernant sa fonctionnalité TCE, cette dernière reste très récente et quelques aspérités restent à corriger pour la rendre un peu plus robuste et propre.

En attendant, il reste tout à fait envisageable d'implémenter TCE de façon plus simple, moins automatisée et peut-être plus performante en se séparant de certains choix techniques.

Notes

Notes

Notes

Nos autres publications

FORMATIONS

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

LIVRES BLANCS

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

8/ DALIBO, L’Expertise PostgreSQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l’animation de la communauté fran- cophone de PostgreSQL. La société est également à l’origine de nombreux outils libres de supervision, de migration, de sauvegarde et d’optimisation.

Le succès de PostgreSQL démontre que la transparence, l’ouverture et l’auto-gestion sont à la fois une source d’innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.