Практическая работа № 3.
Настройка шифрования базы данных

Цель работы: научиться использовать средства шифрования базы данных в PostgreSQL.

Теоретическая часть

Типы шифрования в PostgreSQL

PostgreSQL поддерживает различные подходы к шифрованию:

1. Шифрование при передаче данных (TLS):

Используется для защиты данных, передаваемых между клиентом и сервером PostgreSQL.

2. Шифрование на уровне файловой системы:

Данные шифруются на уровне операционной системы с использованием инструментов, таких как LUKS или BitLocker.

3. Шифрование на уровне приложения:

Данные шифруются перед отправкой в базу данных с помощью алгоритмов шифрования, встроенных в приложение.

4. Шифрование на уровне базы данных:

Используется расширение pgcrypto, которое позволяет шифровать и хэшировать данные непосредственно в PostgreSQL.

Криптографические алгоритмы

PostgreSQL через pgcrypto поддерживает различные алгоритмы шифрования:

- Симметричное шифрование:

Используется один ключ для шифрования и расшифровки данных (например, AES). Простой и эффективный метод для большинства сценариев.

- Асимметричное шифрование:

Используются два ключа (публичный и приватный), что повышает безопасность. Обычно используется для безопасной передачи данных или ключей.

- Хэширование:

Преобразование данных в фиксированную строку (хэш), который невозможно расшифровать обратно (например, SHA-256). Применяется для хранения паролей.

Практическая часть

1. Шифрование данных на уровне базы данных

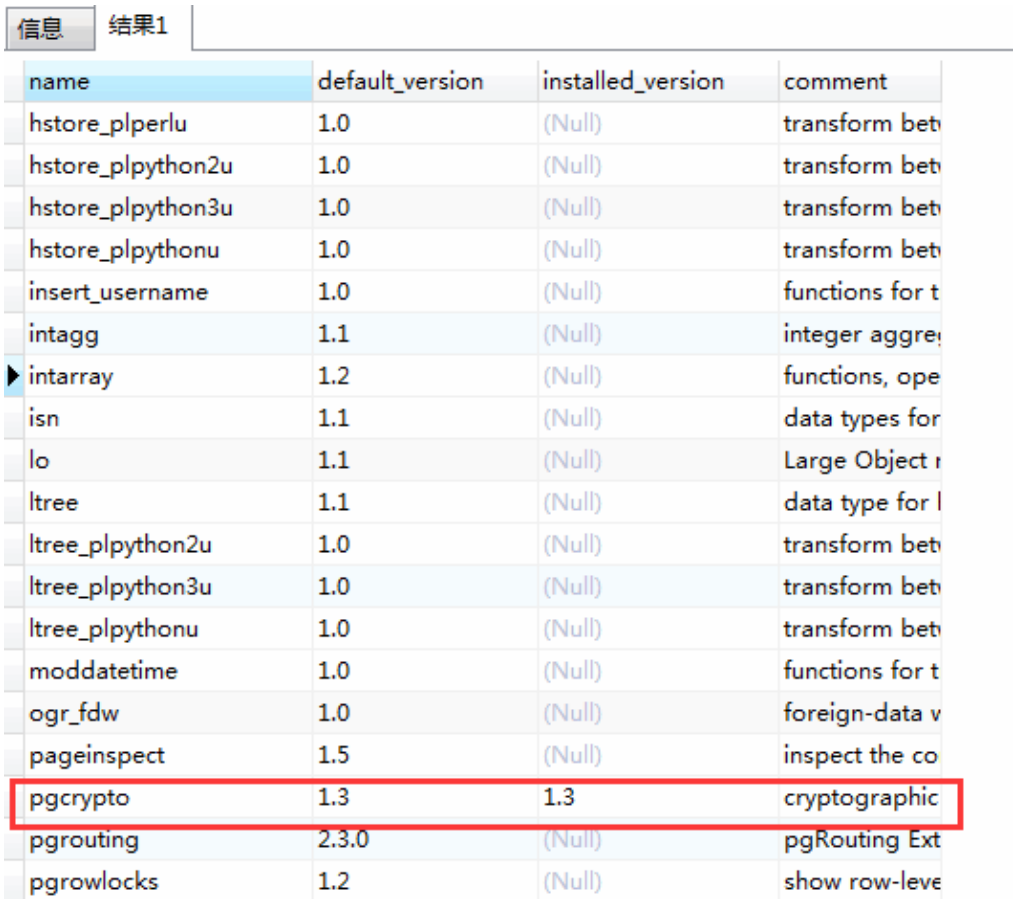
Для шифрования данных PostgreSQL поддерживает расширение pgcrypto, которое можно настроить через pgAdmin. Pgcrypto — это расширение PostgreSQL, которое предоставляет функции для симметричного и асимметричного шифрования данных, а также для хэширования.

1. Установка расширения pgcrypto

- Подключитесь к базе данных через pgAdmin.
- Откройте SQL Query Tool и выполните команду:
- `CREATE EXTENSION IF NOT EXISTS pgcrypto;`
- Выполните запрос:

```
select * from pg_available_extensions;
```

- чтобы убедиться, что расширение установлено



信息 结果1

name	default_version	installed_version	comment
hstore_plperlu	1.0	(Null)	transform bet
hstore_plpython2u	1.0	(Null)	transform bet
hstore_plpython3u	1.0	(Null)	transform bet
hstore_plpythonu	1.0	(Null)	transform bet
insert_username	1.0	(Null)	functions for t
intagg	1.1	(Null)	integer aggre
intarray	1.2	(Null)	functions, ope
isn	1.1	(Null)	data types for
lo	1.1	(Null)	Large Object
ltree	1.1	(Null)	data type for
ltree_plpython2u	1.0	(Null)	transform bet
ltree_plpython3u	1.0	(Null)	transform bet
ltree_plpythonu	1.0	(Null)	transform bet
moddatetime	1.0	(Null)	functions for t
ogr_fdw	1.0	(Null)	foreign-data v
pageinspect	1.5	(Null)	inspect the co
pgcrypto	1.3	1.3	cryptographic
pgrouting	2.3.0	(Null)	pgRouting Ext
pgrowlocks	1.2	(Null)	show row-leve

2. Симметричное шифрование данных

Создайте таблицу с полем для хранения зашифрованных данных:

Query | Query History

```
1  CREATE TABLE employees (
2      id SERIAL PRIMARY KEY,
3      name VARCHAR(100) NOT NULL,
4      email VARCHAR(100) UNIQUE NOT NULL,
5      phone VARCHAR(20),
6      position VARCHAR(50),
7      salary BYTEA, -- Зашифрованная колонка
8      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
9  );
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 33 msec.
```

Выполните заполнение таблицы и шифрование данных:

Query | Query History

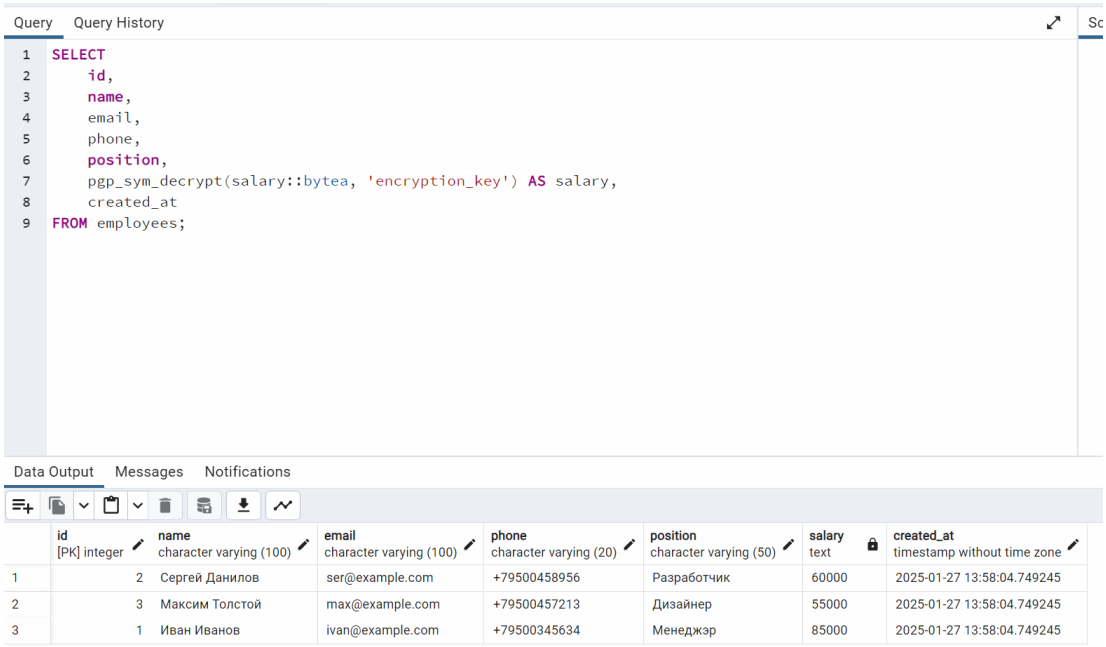
```
1  INSERT INTO employees (name, email, phone, position, salary)
2  VALUES
3  ('Иван Иванов', 'ivan@example.com', '+79500345634', 'Менеджэр', pgp_sym_encrypt('80000', 'encryption_key')),
4  ('Сергей Данилов', 'ser@example.com', '+79500458956', 'Разработчик', pgp_sym_encrypt('60000', 'encryption_key')),
5  ('Максим Толстой', 'max@example.com', '+79500457213', 'Дизайнер', pgp_sym_encrypt('55000', 'encryption_key'));
6
```

Data Output | Messages | Notifications

```
INSERT 0 3

Query returned successfully in 33 msec.
```

Для дешифрования данных выполните запрос:



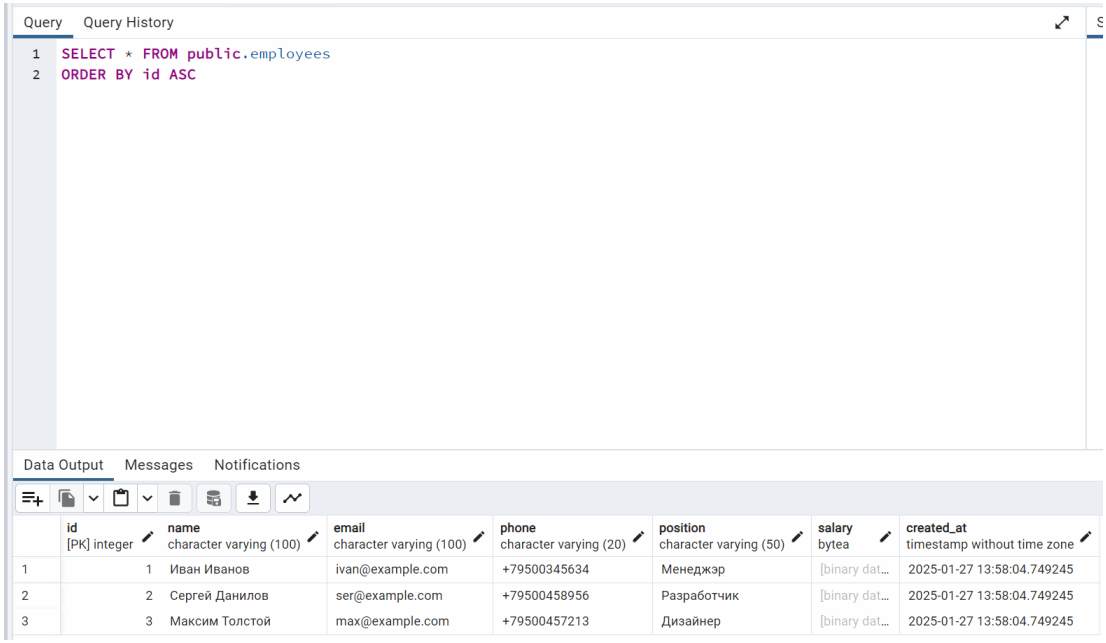
Query | Query History

```
SELECT
    id,
    name,
    email,
    phone,
    position,
    pgp_sym_decrypt(salary::bytea, 'encryption_key') AS salary,
    created_at
FROM employees;
```

Data Output | Messages | Notifications

	id [PK] integer	name character varying (100)	email character varying (100)	phone character varying (20)	position character varying (50)	salary text	created_at timestamp without time zone
1	2	Сергей Данилов	ser@example.com	+79500458956	Разработчик	60000	2025-01-27 13:58:04.749245
2	3	Максим Толстой	max@example.com	+79500457213	Дизайнер	55000	2025-01-27 13:58:04.749245
3	1	Иван Иванов	ivan@example.com	+79500345634	Менеджэр	85000	2025-01-27 13:58:04.749245

Обычное отображение таблицы без дешифрования:



Query | Query History

```
SELECT * FROM public.employees
ORDER BY id ASC
```

Data Output | Messages | Notifications

	id [PK] integer	name character varying (100)	email character varying (100)	phone character varying (20)	position character varying (50)	salary bytea	created_at timestamp without time zone
1	1	Иван Иванов	ivan@example.com	+79500345634	Менеджэр	[binary dat...	2025-01-27 13:58:04.749245
2	2	Сергей Данилов	ser@example.com	+79500458956	Разработчик	[binary dat...	2025-01-27 13:58:04.749245
3	3	Максим Толстой	max@example.com	+79500457213	Дизайнер	[binary dat...	2025-01-27 13:58:04.749245

3. Хэширование данных таблицы

В отличие от шифрования, хеширование работает только в одну сторону: из значения хеш-функции невозможно получить исходные данные. Это используется для аутентификации и подписывания документов.

Хеширование значения:

Query | Query History

```
1  INSERT INTO users (name, email, encrypted_data)
2  VALUES (
3      'Bob',
4      'bob@example.com',
5      digest('SuperSecurePassword', 'sha256')
6  );
```

Обычное отображение таблицы:

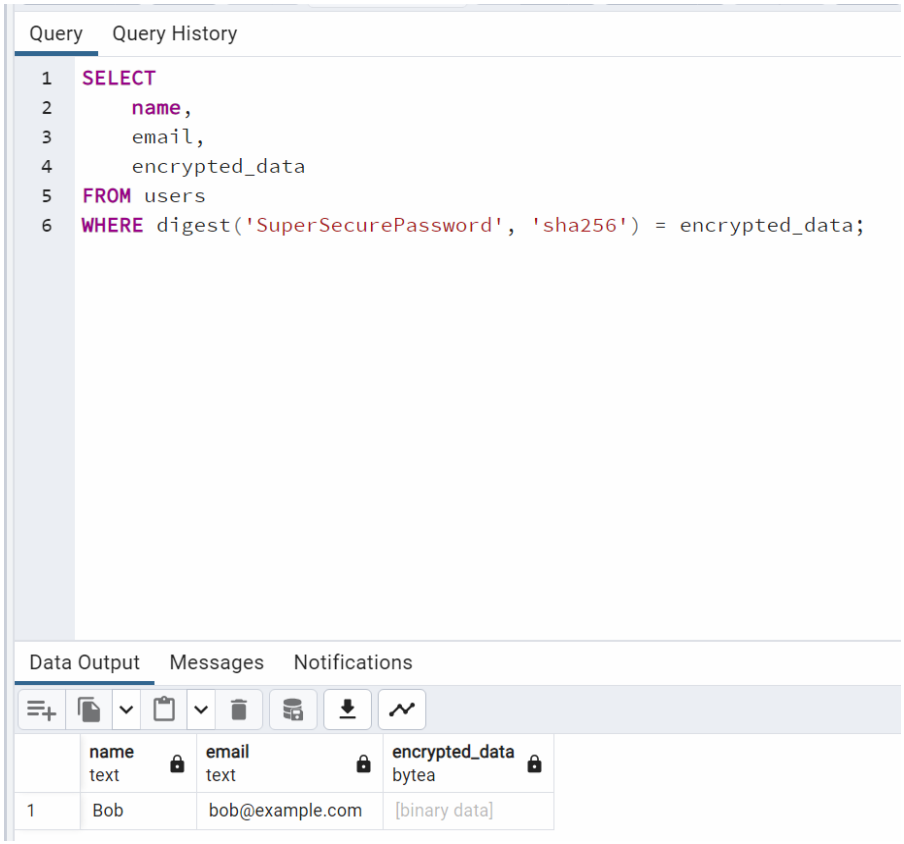
Query | Query History | Scratch Pad

```
1  SELECT * FROM public.users
2  ORDER BY id ASC
```

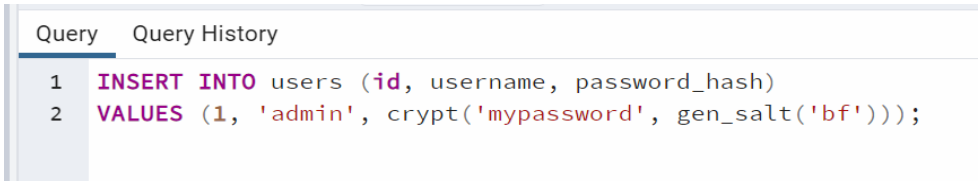
Data Output | Messages | Notifications

	id [PK] integer	name text	email text	encrypted_data bytea
1	1	John Doe	john.doe@example.com	[binary data]
2	2	Bob	bob@example.com	[binary data]

Поиск строки по совпадающему паролю:



Альтернативный способ хэширования:



Благодаря «соли», т.е. случайному набору данных, подмешиваемому к хешированным данным, этот метод обеспечивает большую безопасность, поскольку при этом одинаковые значения перестают иметь один и тот же хеш. Если злоумышленник знает пароль, и два пароля совпадают, это не позволит ему узнать другой пароль по совпадающему хешу. Однако платой за безопасность становится намного меньшая производительность. Рекомендуется использовать его только тогда, когда запросы с использованием хеширования редки, а требования к безопасности высоки.

4. Создание и настройка сертификатов SSL

Для обеспечения безопасного взаимодействия между серверами при использовании Foreign data wrapper требуются сертификаты. В работе предлагается использовать утилиту OpenSSL, доступную в режиме командной строки в любой операционной системе. Для выполнения работы достаточно использовать версию Light.

1. Сгенерируйте сертификаты с помощью OpenSSL:

```
openssl req -new -text -out server.req
openssl rsa -in privkey.pem -out server.key
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

В ходе выполнения команд программа запросит параметры создаваемых сертификатов. Особое внимание нужно обратить на параметр **Common Name**, значением которого должно быть имя хоста, к которому будет подключаться внешний сервер.

2. Переместите **server.crt** и **server.key** в каталог данных PostgreSQL (например, **C:\Program Files\PostgreSQL\data**).
3. В файле **postgresql.conf** укажите пути к сертификатам:

```
ssl = on
ssl_cert_file = 'C:\Program Files\PostgreSQL\data\server.crt'
ssl_key_file = 'C:\Program Files\PostgreSQL\data\server.key'
```

Если сертификаты находятся в директории, в которой развёрнут сервер, полный путь к ним указывать не обязательно.

4. В файле **pg_hba.conf** для нужного пользователя замените строку **host** на **hostssl** (можно скопировать её из другой строки и отредактировать).

При интеграции разных серверов необходимо сертификат **server.crt** переименовать в **root.crt** и поместить в директорию **C:\Users\[CURRENTUSER]\AppData\Roaming\postgresql**. На месте **CURRENTUSER** должен быть пользователь операционной системы, под которым работает развёрнутый экземпляр СУБД Postgres. Для установки защищённого соединения команда создания внешнего сервера должна будет выглядеть так:

```
CREATE SERVER foreign_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (
        host '<ip-адрес-источника>',
        dbname '<имя-базы-источника>',
        port '<порт>',
        sslmode 'require'
    );
```

Самый последний параметр включает защищённое соединение без проверки надёжности сертификата. Чтобы проверить подлинность сертификата, необходимо в параметр **sslmode** вместо **require** написать **verify-ca**. В принципе, если использовать значение **require**, помещать переименованный сертификат в директорию **C:\Users\[CURRENTUSER]** не обязательно. Однако этот подход является небезопасным, так как злоумышленник может подменить сертификат сервера и похитить конфиденциальные данные. Ещё более строгая проверка будет, если задать значение **verify-full**. Этот режим защиты соединения будет не только проверять наличие доверенного сертификата, но и то, что он выписан именно на тот хост, на котором развёрнут сервер, к которому идёт подключение. В этой ситуации проверяться будет соответствие параметра **host**, заданного при создании подключения к внешнему серверу, значению **Common Name**, введённому при создании сертификата.

Задание

1. Создайте два сервера, работающие на разных портах, и запустите их.
2. На первом из этих серверов в базе данных создайте таблицу, содержащую зашифрованное поле. Можно просто взять скрипт создания таблицы **employees** из примера.
3. На втором сервере в базе данных создайте таблицу, в которой хранятся ключи для шифрования данных в таблице, созданной в пункте 2. Каждый ключ имеет своё имя, но в этой таблице будут храниться не имена ключей, а их хеши.
4. Сгенерируйте сертификаты и установите их на втором сервере. Создайте пользователя, который может обращаться к таблице, созданной на шаге 3.
5. Подключите первый сервер ко второму с включенным SSL. Создайте на первом сервере внешнюю таблицу, ссылающуюся на таблицу, в которой хранятся ключи для шифрования.
6. Напишите хранимую процедуру для вставки данных в таблицу, созданную на шаге 2. Эта хранимая процедура должна сгенерировать ключ для шифрования и использовать значение естественного ключа этой таблицы в качестве имени ключа. Имя ключа должно быть захешировано с использованием функции **digest** и сохранено во внешней базе данных на втором сервере вместе с самим ключом, который можно сгенерировать любым способом. С помощью этого ключа конфиденциальная информация на первом сервере шифруется.
7. Создайте в таблице ключей ключ с пустым именем ключа и известным вам ключом-паролем, захешированным с использованием функции **crypt**. Пустое имя – это не NULL, а массив из 0 байт (или digest от пустой строки – по желанию).
8. Напишите функцию, которая принимает в качестве параметра пароль, придуманный на шаге 7, и возвращает таблицу с расшифрованными данными, если ей передан правильный пароль.

Оформление отчёта

Отчёт должен иметь титульный лист с указанием ФИО и группы студента, а так же темы практической работы. После этого должен быть скрипт создания таблицы с конфиденциальными данными. Необходимо привести скриншоты командной строки с генерацией сертификатов и с тем, что получилось в файловой системе на обоих серверах. Затем необходимо привести скрипт подключения одного сервера к другому и скриншот с результатами запроса к таблице ключей. Затем необходимо привести скрипты хранимой процедуры для вставки значений в таблицу с конфиденциальными данными и функции для их расшифровки, а также скрипты, в которых вызывается эта процедура и создаётся пароль для расшифровки. Необходимо привести скриншоты, на которых видно, что данные в таблице зашифрованы, что при неправильном пароле они не расшифровываются, а при правильном пароле они расшифровываются.