Secure Database

Dheeraj Kumar Sundar, 811230397
Vinay Tahiliani
Yash Kajavadara

Introduction

As we are all aware, with cloud computing becoming the norm in the industry for flexible and ease of implementation for running and hosting applications for the obvious benefits of dynamic scaling and pay for what you use model, it has become increasingly difficult to protect data. It is impossible to protect the data from content scanning of these cloud providers to feed their business models. Not only privacy protection from the cloud providers, these cloud providers can also be a target of security breaches by malicious outsiders and the security is completely in the hands of the cloud provider. To overcome the above said issues we discuss our approach towards the problem and also show an implementation of the solution in Django.

Existing solutions & their problems

The project aims at providing privacy and security in the modern day cloud based hosting. There are few solutions to this problem which only provide a partial solution:

1. The first is Encryption-in-Transit: Data is encrypted during transmission to the cloud provider which protects the data against man-in-the-middle attacks. But the data received on the cloud storage is in plain-text form.
2. Encryption-at-rest: A widely provided feature by many databases is Transparent Data Encryption or TDE, encrypts data at rest.

Now, it may look like when used together this should solve the problem that we have but it does not and has two major drawbacks:

1. Data, though stored in encrypted form can still be inferred when loaded into the memory either by the cloud provider or by a malicious outsider.
2. Keys are at the same location as the encrypted data, one breach could uncover all the data stored.

Our solution & Project Description

To overcome the drawbacks of existing solutions, we propose a solution where-in we have a middleware to intercept all queries, encrypt the data as configured by the client and store this encrypted data on the cloud provider. The same middleware will also perform search on the encrypted data and also retrieval of decrypted data for the application.

Middleware also uses a Key management service to keep track of keys and rotate them periodically or if and when needed.

Challenges

The first challenge here is to abstract the underlying encryption, decryption, Key-management and other functionalities of the middleware service. Application must treat the middleware just like any other database.

Not having a fixed key to encrypt and decrypt, symmetric encryption will be the obvious choice for performance but how to not restrict ourselves for one key.

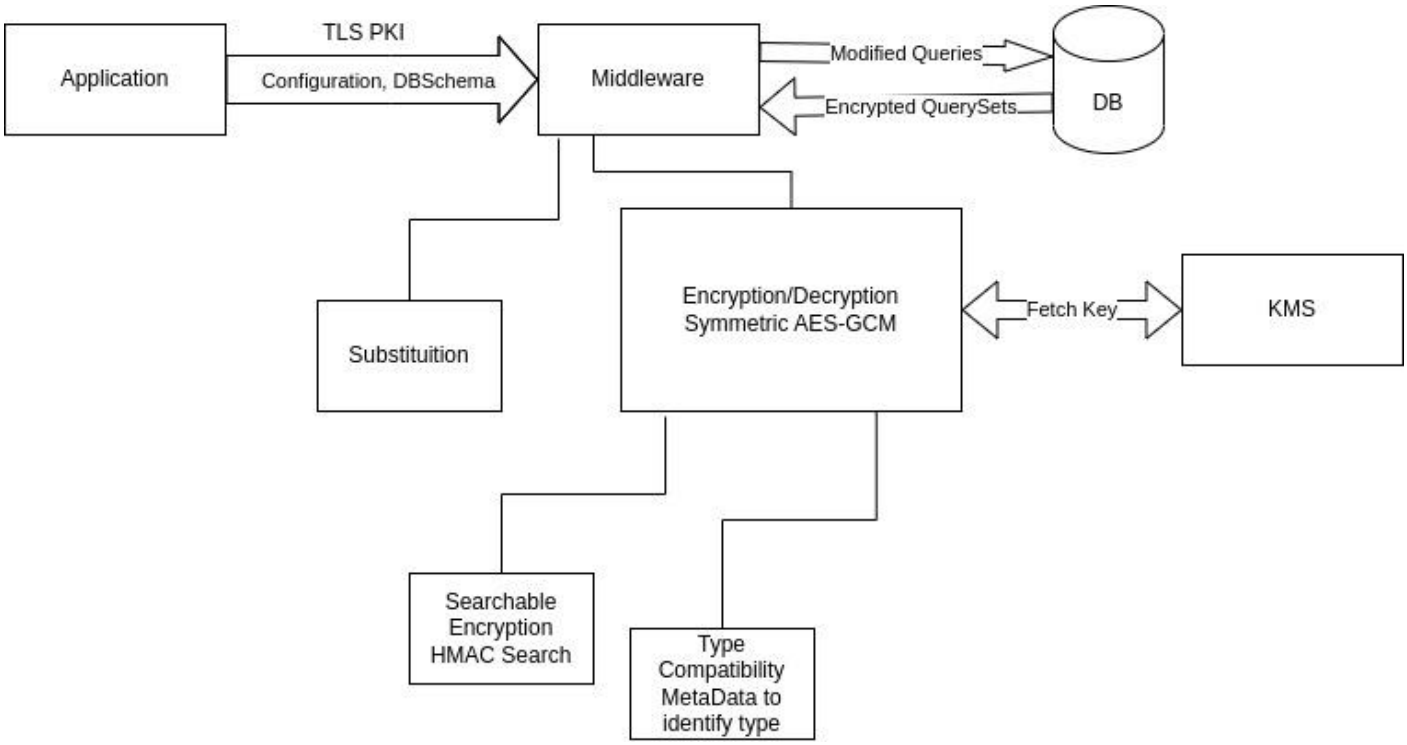
Second challenge was making the encrypted data query searchable, performing search and then decrypting the resultant queryset.

Third challenge was with respect to the datatype for a given database, firstly, any encrypted data may end up producing a ciphertext which is longer than the original data. Also how to respect the data types set, treating big int and encrypting as big int, int as int, text as text.

Authentication for the requests made to the middleware server, it must be strictly adhered to policy within our system that no-one other than the one with the right to access the encrypted data accesses the data.

Configuration, another aspect of using the middleware approach is being able to configure which table, and which attribute within this table needs to be encrypted, and what encryption strategy to use. Rather than providing a default, the developer must have the right to choose between security and performance, and assign sensitivity towards privacy and security towards the data must be their choice.

Architecture



The above diagram shows the design of our system, application provides configuration and DBSchema to our middleware on the basis of which the middleware decides to encrypt or substitute or leave the data as is.

To avoid unauthorized access, middleware performs authentication by preconfiguring client ID on middleware which needs to be matched against the TLS requests coming to the middleware.

Middleware acts as a proxy and modifies the queries for db in accordance with its encryption, substitution, searchable encryptions, and type compatibility components.

Substitution: If configured for a column, the middleware substitutes the actual value with some randomized value, a small storage on the middleware maps this random value to the actual value. This strategy, though may leak information, can be used for attributes which fall on a lower scale of privacy protection. This strategy protects information to an extent and with the advantage of high performance when compared to encryption strategy.

Encryption-Decryption: When configured for a column, the middleware encrypts the data and adds to the db along with some more meta-data. This metadata is useful for other components supported by the middleware, such as searchable encryption and type compatibility.

AES-GCM encryption strategy is used for the data encryption with nonce and tag appended to our final encryption block. This tag and nonce are recovered to verify and decrypt the data.

Searchable Encryption: To protect against the ciphertext attack we use AES-GCM which uses nonce to generate different cipher text for the same input, to be able to search the encrypted data we append keyed hash of the input to the encrypted data. This hash can be used to perform substr search (since nonce, tag, hash have fixed lengths and search for the hash) to fetch the appropriate data. But this comes with a limitation of being able to search with the exact value. This can be extended to perform a blind index search, essentially creating a hash(blind index) for every type of search query. For example, if name starts with Jon(saving hash of Jon and referencing this row to all the rows where name starts with Jon).

Type-Compatibility: To achieve true abstraction, middleware must handle typecasts, treat Big-int as Big-Int, varchar as varchar, etc. This means added extra data as metadata which indicates the type of the field that the application has set it to be.

Key Management Service: A Key management service will inject key into the environment variables of our middleware application.

Configuration: Configuration is the means through which middleware identifies what table to encrypt or substitute, the data type of the attribute.

Implementation

We've implemented the above solution in Django, mainly for two reasons, the obvious one being we are proficient in Django and second is that Django using ORM to communicate with database and each database has an adapter for these ORM conversions which more or less are similar to a proxy middleware(though they are on the same server). Creating middleware will require us to implement a lot of stuff from scratch. Rather than going through the hassle of implementing a sql parser and maintaining connections to DB ourselves we extend the existing functionality by injecting our middleware components onto the pyscopg2 adapter that is available for postgres in python.

We configure a sample django application with just the User table and configure the following fields, first_name, last_name, email, username to be encrypted. We use AES-GCM for encryption of data and HMAC is generated using SHA3-256. Additionally, nonce, tag and headers along with hmac are prefixed to the encrypted data.

Psycopg2 provides extension to the cursor class which is responsible for execution of queries and data retrieval, we override the execute, fetchone, fetchall, fetchmany methods to support the functionality of the middleware. Small portion of the code is shown below.

```
class CustomCursor(psycopg2.extensions.cursor):
    def execute(self, query, vars=None):
        if vars:
            vars = list(vars)
        if "WHERE" in query and "UPDATE" not in query:
            filters = query.split('%s')
            for index in range(len(filters)):
                match = [each for each in encryption_columns if each+" =" in filters[index]]
                if match:
                    vars[index] = hmac.new(password, bytes(vars[index].adapted, 'utf-8'), hashlib.sha3_256).digest()
                    query = query.replace(match[0], f"substring({match[0]}, 33, 32)")
        if "INSERT" in query:
            insertion_attributes = query.split(",")
            for index in range(len(insertion_attributes)):
                if [True for each in encryption_columns if each.split('.')[1] in insertion_attributes[index]]:
                    vars[index] = create_encryption_block(vars[index].adapted)
        if vars:
            vars = tuple(vars)
        super().execute(query, vars)
```

Sample code for encryption and decryption blocks is show below

```
def create_encryption_block(data):
    hmac_for_search = hmac.new(password, bytes(data, 'utf-8'), hashlib.sha3_256).digest()
    cipher = AES.new(key, AES.MODE_GCM)
    cipher.update(header)
    cipher_text, tag = cipher.encrypt_and_digest(bytes(data, 'utf-8'))
    nonce = cipher.nonce
    final_data = nonce + tag + hmac_for_search + cipher_text
    return final_data
```

Team Roles and Responsibilities

Dheeraj was responsible for the research for architecture, architecturing and coding in Django. Vinay was part of the research for architecture. Yash took up the responsibility for documentation and presentation of the project along with some research.