Securing Data in SQL Tables Using Master Key, Asymmetric Key, and Symmetric Key

In today's data-driven world, securing sensitive information in databases is of utmost importance. SQL Server provides various encryption mechanisms that can help protect data at rest. In this article, I will explain how to secure data in a SQL table using a master key with a password, asymmetric keys, and symmetric keys, drawing from my own experiences and implementations.

For the sake of simplicity, this document uses a hardcoded encryption password as an example. However, hardcoding passwords, particularly in SQL scripts, is not recommended, as it introduces security risks and potential vulnerabilities.

Depending on your specific preferences, requirements, and environment, some options for securely storing passwords can include using a secure password vault, a server environment variable, a user-defined function for password retrieval, windows authentication, an encrypted configuration file, or the SQL Server credential store.

Understanding Encryption Keys in SQL Server

SQL Server utilizes three primary types of keys for encryption: master keys, asymmetric keys, and symmetric keys. Each serves a unique purpose in the encryption hierarchy, allowing for secure management of sensitive data.

1. Master Key

A master key is a symmetric key that acts as the root of the encryption hierarchy in SQL Server. It is used to encrypt other keys and is typically protected by a password.

Creating a Master Key

```
use [YourDatabaseName]
go

if not exists (select 1 from sys.symmetric_keys where name = 'YourMasterKeyName')
begin
  create master key encryption by password = 'YourStrongPassword!';
end
go
```

2. Asymmetric Key

An asymmetric key uses a public key for encryption and a private key for decryption. This key type is useful for securely exchanging symmetric keys.

Creating an Asymmetric Key

```
use [YourDatabaseName]
go

if not exists (select 1 from sys.asymmetric_keys where name = 'YourAsymmetricKeyName')
begin
  create asymmetric key YourAsymmetricKeyName
  with algorithm = rsa_2048
  encryption by password = 'YourStrongPassword!';
end
go
```

2. Symmetric Key

A symmetric key uses the same key for both encryption and decryption, making it efficient for encrypting large amounts of data.

Creating a Symmetric Key

```
use [YourDatabaseName]
go

if not exists (select 1 from sys.symmetric_keys where name =
'YourSymmetricKeyName')
begin
  create symmetric key YourSymmetricKeyName
  with algorithm = aes_256
  encryption by password = 'YourStrongPassword!';
end
go
```

Securing Data in a SQL Table

An encrypted SQL table is a database table where the data is stored in an encrypted format, protecting sensitive information from unauthorized access. This encryption ensures that even if the database is compromised, the data remains unreadable without the proper decryption keys.

Here's an example of a table to store partner API information:

```
create table [dbo].[Partner_API]
(
  [pkid] uniqueidentifier not null, -- unique id/primary key
  [Partner_id] int not null identity,
  [Partner_Name] varchar(255) unique not null,
  [Partner_APIKey] varchar(255) not null,
```

```
  [Token_Endpoint] varchar(255) not null
  constraint [pk_settings] primary key clustered
  (
    [pkid] asc
  ) with (pad_index = off, statistics_norecompute = off, ignore_dup_key = off,
allow_row_locks = on, allow_page_locks = on) on [primary]
) on [primary]
go
```

Inserting and Encrypting Data

When inserting sensitive data, use the symmetric key to encrypt the API key before storing it. In this example, the [Partner_APIKey] column (field) is encrypted.

```
insert into [dbo].[Partner_API] (pkid, Partner_Name, Partner_APIKey,
Token_Endpoint)
values (newid(), 'PartnerName', 'https://api.endpoint.com');

open symmetric key YourSymmetricKeyName
decryption by asymmetric key YourAsymmetricKeyName with password =
'YourStrongPassword!';

  update [dbo].[Partner_API]
  set [Partner_APIKey] = encryptbykey(key_guid('YourSymmetricKeyName'),
[Partner_APIKey])
  where [Partner_Name] = 'PartnerName';

close symmetric key YourSymmetricKeyName;
```

Reading Encrypted Data

When selecting encrypted data, use the asymmetric and symmetric keys to decrypt the encrypted column (field). If you do not do this, then the [Partner_APIKey] column (field) will return as NULL in the result set.

```
open symmetric key YourSymmetricKeyName
decryption by YourAsymmetricKeyName key CP_ASymetricKey with password =
'YourStrongPassword'
  select
  [pkid],
  [Partner_id],
  [Partner_Name],
  convert(varchar(255), decryptbykey([Partner_APIKey])) as [Partner_APIKey], --
returns null if symetric key is not opened first
  [Token_Endpoint]
  from [dbo].[CP_Partner_API]
close symmetric key CP_SymetricKey
```

Conclusion

By using a master key with a password, along with asymmetric and symmetric keys, you can effectively secure sensitive data in SQL tables. This layered approach to encryption not only protects data at rest but also enhances overall data security.

Encrypted SQL tables enhance security by safeguarding critical information such as passwords, API credentials, and personal identifiable information (PII) while also supporting compliance with data protection regulations. Access to the encrypted data can be tightly controlled, and any changes or unauthorized attempts to access it can be monitored, providing an additional layer of security for sensitive information.

Storing sensitive information, such as API credentials, in an encrypted SQL table offers several benefits beyond enhanced security:

Data Protection: Encryption protects sensitive data from unauthorized access, even if the database is compromised. This is crucial for compliance with data protection regulations.

Compliance: Many industries have regulations (e.g., HIPAA, GDPR) that require the encryption of sensitive data. Using encrypted SQL tables helps ensure compliance with these legal standards.

Access Control: By storing credentials in a centralized location with strict access controls, you can manage who has access to sensitive information, reducing the risk of exposure.

Data Integrity: Encryption can help maintain data integrity by ensuring that any unauthorized changes to the data can be detected. If the data is altered, it may not decrypt correctly, alerting you to potential tampering.

Easier Key Rotation: When using an encrypted table, you can implement key rotation strategies more easily, allowing you to update encryption keys without significant changes to the application.

Separation of Concerns: Storing credentials in an encrypted SQL table can help separate sensitive data from application code, reducing the risk of accidental exposure in code repositories.

Backup Security: Encrypted data remains secure even in backups. If backups are accessed by unauthorized parties, the data remains unreadable without the appropriate decryption keys.

Auditing and Monitoring: With encrypted tables, you can implement logging and monitoring of access attempts to sensitive data, helping you to detect and respond to potential security breaches more effectively.

Using an encrypted SQL table for storing sensitive information is a best practice that contributes to a more secure application architecture.