Implementing a Hybrid Column Level Encryption in MySQL

Alexander Rubin
May 13, 2021
About me

Working with MySQL for ~15 years
• Started at MySQL AB 2006
  • Sun Microsystems, Oracle (MySQL Consulting)
  • Percona since 2014
• Joined the Amazon Relational Database Service (RDS) engineering team in 2020

Interests in:
• IoT / devices
• IT security
Agenda

1. Existing encryption methods
   - May not provide adequate encryption for sensitive data
2. Proposed method of column level encryption
Protecting Data In MySQL

Background information and existing solutions
Protecting Data In MySQL: encryption

- Encryption will protect sensitive data
- Required by HIPAA, PCI compliances, etc

Amazon RDS security features:

- Encryption of Data at Rest
- Encryption of Data in Transit

https://aws.amazon.com/rds/features/security/

Some sensitive data may require an additional protection
Protecting data in MySQL: 

**Types of encryption**

1. **Data in flight**
   - SSL/TLS

2. **Data at Rest**
   - Full disk encryption
   - Transparent DB Encryption
   - Field level encryption
Data at Rest Encryption: full disk encryption options

Data at Rest

Full disk encryption

- Amazon RDS or EC2: Encrypting disk with KMS
  
  Encryption

  - Enable encryption
    Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. Info

  - Master key Info
    (default) aws/rds

- Full disk encryption on Linux: LUKS / etc
- Shared storage encryption
Data at Rest Encryption: full disk encryption downsides

- Only protect from physical access to disk (or reusing images)
- If MySQL is running:
  - data in MySQL files *can be seen as unencrypted*
    - *It is encrypted only when volume(s) are not mounted*
Data at Rest Encryption: **TDE**

**Data at Rest**

**Transparent DB Encryption**

Transparent Database Encryption (TDE): encrypting db files

1. **InnoDB files**: tablespaces, redo logs, undo logs
2. **Binary logs, relay logs**: for MySQL replication
3. **Tmp files**
Data at Rest Encryption: **field/column level encryption**

**Data at Rest**

**Field level encryption**

**Application code** encrypts needed fields.

For example:
- PII information
- Medical (PHI) information
- Etc

**Issues:**
- Key rotation
- Searches in MySQL - range search does not work
- Order by searches
- Indexes
## Data at Rest Encryption options: comparison

<table>
<thead>
<tr>
<th>Encryption option</th>
<th>PROs</th>
<th>CONs</th>
</tr>
</thead>
</table>
| **Full disk**     | • No application changes needed  
|                   | • Lowest overhead             | • When **system** is running data is decrypted  
|                   |                               | • Does not protect from **copying database files**  |
| **TDE**           | • No application changes needed  
|                   | • Low overhead               | • When **MySQL** is running data is decrypted  
|                   |                               | • Does not protect from **exporting data** (i.e. mysqldump)  |
| **Field / column**| • Best protection  
|                   | • Can be combined with other options | • Require some application changes  
|                   |                               | • Overhead can be significant  |
Field Level Encryption

Implementation options
Field/column level encryption: common approach

Application encrypt/decrypt data (code)

• Data is already encrypted
  • In flight (from app to MySQL)
  • At rest (when MySQL stores it on disk)
Field/column level encryption: problems

1. Application needs to implement encryption
   • What if we have a legacy application?
2. Need to store and rotate encryption key
3. Complicate MySQL queries:
   • Range scan / order by will not work inside MySQL
Field Level Encryption

Possible approach for MySQL
Field/column level encryption for legacy application

Our Plan

1. Identify fields that require field/level encryption
2. Create encrypt /decrypt function (stored function or UDF)
3. Use MySQL rewrite plugin (+ triggers) to “inject” encryption
4. Use application encryption keys
Implementation diagram

```sql
set @app_key='secret';

insert into employees (... ssn) values (... my_encrypt(...))

select my_decrypt(ssn) from employees where ...
```

my_encrypt() my_decrypt(): stored functions we will create
Field/column level encryption for legacy application

Pre-requisites
1. Data in flight encryption:
   • application connects to MySQL with SSL/TLS
2. Application stores its key securely
Field/column level encryption for legacy application

Implementation 1:
create MySQL stored functions to encrypt / decrypt

```sql
use secret_schema;
create function my_encrypt(str text character set utf8) returns text binary
return aes_encrypt(str, @app_key);

create function my_decrypt(str text binary) returns text character set utf8
return aes_decrypt(str, @app_key);
```
Field/column level encryption for legacy application

mysql> set @app_key='secret';
Query OK, 0 rows affected (0.00 sec)

mysql> select hex(my_encrypt('test'));
+----------------------------------+
| hex(my_encrypt('test'))          |
| 1356CAC477BA0A814054243FDBE2398F |
+----------------------------------+

mysql> select my_decrypt(my_encrypt('test'));
+----------------------------------+
| my_decrypt(my_encrypt('test'))   |
| test                             |
+----------------------------------+

Application sets this key
Security

App: Attacker
Need key to decrypt data
Vulnerable?

MySQL DBA
can’t decrypt exported data
OK

Man-in-the-middle
OK: can’t decrypt SSL

© 2021, Amazon Web Services, Inc. or its Affiliates.
Vulnerable points

Application level: key needs to be protected
  On disk - store in secure storage. I.e.:
    • AWS Secrets Manager: https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html
    • Vault
  Load key in Application memory on start

MySQL level
  key can be recorded in logs (i.e. slow log, audit log etc)
Dealing with legacy applications

```sql
set @app_key='secret';

insert into employees (… ssn) values (… '111-11')

select ssn from employees...

MySQL trigger: 
'111-11' -> 
my_encrypt(…)

Query rewrite:  
ssn ->  
my_decrypt(ssn)
```
Dealing with legacy applications: implementation

```sql
mysql> CREATE TABLE `employees` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  ...  
  `ssn` blob,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB;
```

```sql
mysql> CREATE TRIGGER encr  
BEFORE INSERT on employees  
FOR EACH ROW  
set NEW.ssn=my_encrypt(NEW.ssn);
Query OK, 0 rows affected (0.01 sec)
```

Trigger will take care of inserts / updates

```sql
mysql> insert into employees(ssn)  
values('111-11-111');  
Query OK, 1 row affected (0.00 sec)
```

```sql
mysql> select * from employees;
+-----------------+
| id | ssn            |
+-----------------+
| 1   | !=9Ll..V.      |
1 row in set (0.00 sec)
```

```sql
mysql> select my_decrypt(ssn)  
from employees;
+-----------------+
| my_decrypt(ssn) |
+-----------------+
| 111-11-111      |
1 row in set (0.00 sec)
```
Dealing with legacy applications: implementation

Install re-write plugin:  
* Query rewrite plugin is not available currently on Amazon RDS


```sql
mysql> INSERT INTO query_rewrite.rewrite_rules (pattern, replacement) 
VALUES('SELECT ssn from app.employees', 
       'SELECT secret_schema.my_decrypt(ssn) from app.employees');
Query OK, 1 row affected (0.00 sec)

mysql> CALL query_rewrite.flush_rewrite_rules();
Query OK, 0 rows affected (0.00 sec)

mysql> select ssn from app.employees;
+-------------------------------+
| secret_schema.my_decrypt(ssn) |
+-------------------------------+
| 111-11-111                   |
+-------------------------------+
1 row in set, 1 warning (0.00 sec)
```

mysql> show warnings\G

```
Level: Note
Code: 1105
Message: Query 'select ssn from app.employees' rewritten to 'SELECT secret_schema.my_decrypt(ssn) from app.employees' by a query rewrite plugin
```

Re-write will take care of selects
Other operations on encrypted columns

mysql> alter table employees change ssn ssn varbinary(255),
     
     add key (ssn);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> set @app_key='secret';
Query OK, 0 rows affected (0.00 sec)

mysql> set @app_key='secret';
   
   select id, secret_schema.my_decrypt(ssn)
   from employees
   where ssn=secret_schema.my_encrypt('111-11-111');

<table>
<thead>
<tr>
<th>id</th>
<th>secret_schema.my_decrypt(ssn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>111-11-111</td>
</tr>
</tbody>
</table>

Re-write CAN take care of this as well
Other operations on encrypted columns: order by

```sql
mysql> select id, secret_schema.my_decrypt(ssn), secret_schema.my_decrypt(last_name) from employees order by secret_schema.my_decrypt(last_name) limit 10;
...
10 rows in set (29.79 sec)
```

```sql
mysql> alter table employees add key (last_name);
```

```sql
mysql> explain select id, secret_schema.my_decrypt(ssn), secret_schema.my_decrypt(last_name)
from employees
order by secret_schema.my_decrypt(last_name) limit 10\G
```

```
table: employees
type: ALL
possible_keys: NULL
  key: NULL
key_len: NULL
  rows: 1045875
filtered: 100.00
Extra: Using temporary; Using filesort
```

Order by queries will not use indexes and will be slow
Performance considerations: benchmark

Without trigger:

```sql
mysql> insert into employees_copy
    select * from employees;
Query OK, 300024 rows affected (3.73 sec)
Records: 300024  Duplicates: 0  Warnings: 0
```

With trigger:

```sql
mysql> insert into employees_copy_with_trigger
    select * from employees;
Query OK, 300024 rows affected (7.15 sec)
Records: 300024  Duplicates: 0  Warnings: 0
```

- Negligible / hard to measure for single inserts
- ~2x slower for bulk inserts
### Field Level Encryption Implementation: Pros/Cons

<table>
<thead>
<tr>
<th>PROs</th>
<th>CONs</th>
</tr>
</thead>
</table>
| • Higher level of security for sensitive data  
   • Data is encrypted even when database is running  
• Compatible with other encryption methods  
• No application changes needed  
   • Need to set the app key on connection | • Limits on SELECT queries:  
   • Slow ORDER BY  
   • Slow range queries  
• Performance for bulk inserts/updates can be 2x worse  
• Key rotation will require re-encryption  
• Query text can be recorded in logs (i.e. slow log general_log, performance_schema, etc) |
Other options

Using MySQL keyring SECRET and Asymmetric Encryption

This is a different setup with 2 database users
Summary

1. Discussed field level data encryption implementation
2. PRO: Transparent to the application
   • Better security
   • No/minimal application changes needed
3. CON: some queries will be slower, need to test/benchmark
Thank you!