Computing Arbitrary Functions of Encrypted Data

Klev

September 21, 2015

1 Abstract

Two problems: Curious database managers that peaks into the information stored in the system, and malicious adversaries that obtains access to the system through bugs and security flaws. With physical access to the database, an attacker would be able to access all data, both in memory and what is stored on disk.

CryptDB: Provides practical and provable confidentiality when exploited to these types of attacks on applications containing an SQL database.

Does so by using a collection of efficient SQL-aware encryption schemes. Can bootstrap encryption keys to user passwords. Can only decrypt using the password of some of the users with access to the data.

Low overhead, reducing throughput by 14,5%. Which is modest.

2 Introduction

Approaches: Encrypt all data in the database/on the server. Let the client process the data. Kinda stupid. Puts a lot of competition on the user, and is useless for application websites/systems that process a lot of data. Difficult to convert to this kind of solution.

CryptDB: Intermediate design for applications using database management systems. Leverages the application server - database server model. Practical because of SQL uses a well-defined set of operators which can be supported efficiently.

Problem 2: Cannot guarantee security of users that are logged-in during an attack.

Two challenges:

1) Limit the amount of confidential information that is exposed to the application/system, and at the same time support a large amount of different types of queries to be performed.
If encrypting the data with a strong cryptosystem, say AES, you would defeat much of the purpose as you would have to give the system your decryption key for it to be able to locate and decrypt the information you need.

2) Minimize the information that is leaked if the system gets compromised. Ensure that the system can only obtain a limited amount of decrypted data. Giving each user a database encryption key does not work for shared data.

Three key ideas:

1) Execute SQL queries over encrypted data. Leverages that SQL is made of well-defined set of primitive operators. Encrypts the data in a way that makes it possible to perform these operations on the transformed data. Efficient because it uses symmetric key encryption, and avoids fully homomorphic encryption.
2) Adjustable query-based encryption. Uses an "onion" of encryptions to avoid leaking information. Adjusts the encryption layer in realtime based on the queries. Compactly store multiple ciphertexts within eachother.
3) Chain encryption keys to user passwords. Can only decrypt data using a chain of keys consisting of a user with access' password and an encryption key.

3 Security Overview

Intercepts all SQL queries using a Database Proxy server. the function of this proxy is to rewrite the query to be able to operate on the encrypted data. Encrypts/decrypts data.

The database never receives decryption keys, so it is not capable of decrypting any of the information stored.

Developers must annotate the SQL schema to define different principals. Their keys will be able to decrypt the different parts of the database.

Out of the scope: Deleting the database after getting access, SQL injection and XSS.

Goal of CryptDB: Confidentiality, not integrity or availability.

Attacker assumed to be passive: Does not change queries, results or the data in the database system.

Database uses a secret key to encrypt all data inserted in the DB + queries.

SQL-aware encryption that dynamically adapts to the queries asked from the user. Examples: Group by, Sort, Max and Min

Reveals the overlaying table structure of the database, but not column names or types, number of rows and size.

Properties provided:

- Sensitive data is never in plaintext on database server
- Data exposed to the database server depends on what the application is allowed to query, and on the developers schema.
- The database server can not compute (encrypted) results for queries that arent requested by the application.

Encrypt data that belongs to different users with different encryption keys. This is specified in the database schema using annotations. CryptDB leaks at most the data of the users that are logged in during the server compromise.

4 Queries over Encrypted Data

No need to change existing applications to be able to support SQL queries over encrypted data.

CryptDB's proxy server stores a secret masterkey MK, the database schema and the current encryption layer of all columns. The database server sees an anonymized database schema.

Equips the server with CryptDB's own user-defined functions (UDF). Basically enables the server to perform certain operators on the ciphertexts.

How to CryptDB:
The user performs an action in the application that triggers a query. The query is intercepted by the CryptDB proxy server and rewritten. This involves renaming all column names in the query so it is anonymized. It then encrypts (using the masterkey MK) each constant in the query with the encryption schema that is the best fit for the given query and operation.

If necessary, the proxy sends the keys for adjusting the encryption layer to the database server. If so, it sends an UPDATE that adjust the encryption layer for the appropriate columns.

The proxy forwards the encrypted query to the database, which executes and returns the encrypted results to the proxy. The proxy then decrypts the returned data and ships it to the application.

4.1 SQL-aware Encryption

Different encryption types:
Random (RND): Highest level of security. IND-CPA. Does not allow any computation. Uses AES or Blowfish in CBC mode with an IV.

Deterministic (DET): Leaks which encrypted values that decrypts to the same value. Allows select with equality check, joins, group by, count and distinct. Should be a pseudo-random permutation (PRP). Uses AES with CMC mode.

Order-preserving encryption (OPE): Allows order relations between values based on their encrypted values. Enables the server to perform Order by, Max, Min, Sort. Weaker, because it reveals the order. Only used if the user request order-queries on the columns.

Homomorphic Encryption (HOM): IND-CPA secure. Using Pailler: $HOM_k(x) * HOM_k(y) = HOM_k(x + y)$. Examples: SUM, then the proxy replaces the a call to a UDF that performs Pailler multiplication.

Join (JOIN and OPE-JOIN): To enable equality joins between two columns. OPE-JOIN enables to joins based on order relations.

Word search (SEARCH): Perform search on encrypted text. "Like" operator from MySQL. If column needs SEARCH, split text on delimiter. Nearly as secure as RND. Gets an encrypted token from the proxy. Searches for words matching the token. Can only search on whole words. Not regular expressions.

4.2 Adjustable Query-based Encryption

Idea: Dynamically adjust the encryption layer based on the queries that are observed in real-time. Always use the highest possible layer while still be able to execute the query. Encrypt data items in one or multiple layers. Encrypted in layers, where each layer enable certain functionalities. Multiple onions needed because of performance etc. Do not need all onions for every column.

"For each layer of each onion, the proxy uses the same key for encrypting values in the same column, and different keys across tables, columns, onions and onion layers." All keys are derived from the master key MK.

$K_{t;c;o;l} = PRP_{MK}(tablet; columnc; oniono; layerl)$

Proxy server adjustes encryption layer of onion based on query received. Uses DecryptUDF in UPDATE querys to adjust onion layer.

4.3 Executing over Encrypted Data

When the onion layer is adjusted properly, the proxy server transform into a query that can operate on the transformed onion. Changing names of column,

tables and variables, etc. Send to database, and gets the result in return.

Example:

```
SELECT ID FROM EMPLOYEES WHERE NAME = 'ALICE';
```

triggers

```
UPDATE TABLE_1 SET C2-Eq = DECRYPT_RND(Key, C2-Eq, C2-IV)
```

proxy then queries

```
SELECT C1-Eq, C1-IV FROM TABLE_1 WHERE C2-Eq = x7..d
```

Exposes NULL-values to the database without any encryption.

4.4 Computing Joins

Meh.

4.5 Security Improvements

Application developer specifies the lowest onion level for each column.
Evaluating in proxy can improve security.
Training mode: CryptDB suggests encryption level based on a training set of queries.
Onion re-encryption: Re-encrypt to a higher level after an infrequent query.

4.6 Performance Optimizations

Using annotations if fields are not sensitive to avoid using encryption.
If the developer knows most of the queries that the users will perform, possible to compute a prior.

5 Multiple Principals

In cryptDB: Each user has a key(The application level password) that gives access to the user's data. Encrypts different data items with different keys. Onion keys to decrypt data from different fields in the database are derived from the password of the logged-in user. The keys are deleted when user logs out.

Access control policy determines the data that a user has access to.

Developer specifies in the schema using annotations which user entities that should be granted access to different data items.

Step 1:

Define principal types using PRINCTYPE. Examples: Users, Groups, Messages. To types: External and internal. External users are end-users with

passwords, internal can only be acquired through delegation inside the application.

Step 2: Developer specifies which columns contains sensitive data plus the principal entities that should have access to it.

Step 3: Using "speaks for", programmer can delegate privileges of one principal to other principals.

5.1 Key chaining

Each principal is associated with a randomly chosen key which is stored in the special *access_keys* table.

A **SPEAKS FOR** B: B's key is encrypted using A's key.

Message example: Message between A and B are encrypted individually using A and B's keys.

ENC FOR: Sensitive field encrypted using the key of the specified principal. Onion keys are derived from a principal's key.

The key of each principal is a combination of a symmetric key and a public-key-pair. Uses primarily the symmetric key. If user is not logged in, use public-key.

For external principals (users): Generate random key as for internal principals. Stores the key of each external in a table *external_keys*. Allows CryptDB to obtain the user's key given its password.