CryptDB

- Pessimistic - Data gets stolen all the time
  - 1: DBAs can look but should not ("curious DBA")
    - "Malicious DBAs are more like to read data… than to change data or query results"
  - 2: Adversary gains control (General theft)
  - Not trusted: DBMS machines, administrators
  - Trusted: application and proxy
- Application <-> Proxy <-> Server
  - Good at doing predefined queries. Good? Bad
  - Proxy intercepts query from Application, encrypts, and sends to server
  - Server does its thing, sends to proxy, proxy decrypts, sends to app
  - Master key in the proxy
  - All on the same machine according to MR
  - DBMS sees anonymized schema (puzzle pieces with no picture)
- **Idea: execute SQL queries over encrypted data**
- We could use randomized encryption
  - Q: What happens if we ask for people with Employee ID of 1234?
    - A: Could map to different values
    - S: Use non-deterministic encryption for equality checks
    - Server can perform equality checks by checking the encrypted data
- Good news: most of SQL is just a handful of operations
- Equality
- Note that the ORDER IS NOT PRESERVED
  - Use an order-preserving encryption (OPE) scheme
- Encryption schemes (decreasing security, increasing functionality):
  - Randomized (RAND)
    - strongest, leaks no info about the data
    - No SQL use
  - Homomorphic (HOM)
    - SUM operations
      - HOM(x) • HOM(y) = HOM(x+y)
    - pretty secure (almost RAND)
    - Stan Zdonik
  - Search (SEARCH)
    - LIKE operations
    - Note: only full words supported
  - Deterministic (DET)
    - Equality operations (=, in, groups)
  - Join (JOIN)
    - Finding equality matches between two columns
    - Q: Each column has a different key!
  - Order Preserving (OPE)
    - x < y => OPE(x) < OPE(y)
- **Idea: use "onions" to encrypt in layers ("adjustable query-based encryption")**
  - We NEVER fully decrypt an onion; there is ALWAYS at least one layer
  - Eq
  - Ord
  - Search
  - Add
  - Note that the RAND is on the outside!
  - We encrypt from weakest on inside to strongest on outside
  - Q: Why multiple onions?
    - A: Computations supported by different enc schemes not necc. ord'd
    - A: Performance considerations
    - Problems? Space concerns? How much concern?
  - Where does a specific onion make sense?
    - Search on integers? NO
    - Search on username? YES
    - Add on strings? NO
- ALL keys are derived from the MK
  - Each key is specific to a single COL

  - User keys are generated at each login, deleted at log off
- Onion decryption is performed entirely by the DBMS server. Why?
  - Only done when needed / DBMS is secure
- When you are offline, your key is not available
  - What if I send a message to Alice while she's offline?
  - Encrypts message with Alice's public key, Alice logs on and automatically encrypts with her private key
- NO change to the DBMS, all through UDFs. Good? Bad?
- Not all operators supported
  - Trigonometry
  - Combining encryption schemes (T1.a + T1.b > T2.c)
    - HOM does not preserve order
  - Is this a problem?
- How do we set this up:
  - Step 1: Define principle types (external: authenticated users, internal: delegated users)
  - Step 2: Specify sensitive data columns ENC_FOR
- Problems:
  - does NOT ensure integrity, freshness, or completeness of results
  - "Equality checks for a few rows… require revealing that class of computation for an entire column"
- How do we attack this?
  - Sorting attack: Attack OPE columns, every value sorted in the column, map to plain text
    - DBMS uses sorted intermediate results for answering queries with range joins
- What if we attack users?
  - Only logged in users are vulnerable
    - How many people stay logged into an application?
  - Case of Marriott -> most people would not be affected
  - Task: get somebody online
    - Who do we target?
- Attack: this is easily crackable because OPE is bad!
  - Response: you're not using it properly!

<u>EX:</u>

| GuestId | Name | Username | Email | Password | CreditCardNo

Connor stayed at the Westin and wants to pay:
SELECT CreditCardNo FROM Guests WHERE Name='Connor Luckett';
This is targeting an Eq onion. We use:
    K_{Guests, CreditCardNo, Eq, RND}
    K_{Guests, CreditCardNo, Eq, DET}
    K_{Guests, CreditCardNo, Eq, JOIN}

Connor moved and needs to check his bill:
CREATE TABLE bills (...
     payerId int, *total* int
    (*payerId* payer) int SPEAKS_FOR (*total* bill)
)