Security

- SQL Injection
- Encryption

SQL injection attacks

- Affect applications that use untrusted input as part of an SQL query to a back-end database

- Specific case of a more general problem: using untrusted input in commands
SQL injection: example

- Consider a browser form, e.g.:

  ![Webpage Screenshot](image)

  - When the user enters a number and clicks the button, this generates an http request like

Example continued...

- Upon receiving the request, a script might generate an SQL query as follows:

  ```java
  sql_query = "SELECT pizza, quantity, order_day " + "FROM orders " + "WHERE userid=" + session.getCurrentUserId() + " AND order_month=" + request.getParameter("month");
  ```

- A normal query would look like:

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month=10
  ```
Example continued...

- What if the user makes a modified http request: https://www.pizza.com/show_orders?month=0%20OR%201=1
- (Parameters transferred in URL-encoded form, where meta-characters are encoded in ASCII)
- This has the effect of setting request.getParameter("month") equal to the string 0 OR 1=1

Example continued

- So the script generates the following SQL query:

```sql
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=1
```
Example continued

- So the script generates the following SQL query:
  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE (userid=4123
  AND order_month=0) OR 1=1
  ```

- Always evaluates to TRUE
  - The attacker gets every entry in the table!

Even worse...

- Craft an http request that generates an SQL query like the following:
  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month=0 OR 1=0
  UNION SELECT cardholder, number, exp_date
  FROM creditcards
  ```

- Attacker gets the entire credit-card table instead!
More damage…

- SQL queries can encode multiple commands, separated by ‘;’
- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
DROP TABLE creditcards
```

- Credit-card table deleted!

XKCD…

```
HI, THIS IS YOUR SON’S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR — DID HE BREAK SOMETHING? IN A WAY—

DID YOU REALLY NAME YOUR SON Robert?); DROP TABLE Students;-- ?

OH, YES LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
```
More damage…

* Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0;
INSERT INTO admin VALUES ('hacker', ...)
```

* User (with chosen password) entered as an administrator!
  * Database owned!

May need to be more clever…

* Consider the following script for text queries:

```java
sql_query
    = "SELECT pizza, quantity, order_day "
      + "FROM orders "
      + "WHERE userid=" + session.getCurrentUserId()
      + " AND topping=’" 
      + request.getParameter("topping") + “’"
```

* Previous attacks will not work directly, since the commands will be quoted

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=’0; 
INSERT INTO admin VALUES (‘hacker’, ...)
```
Example continued…

- Craft an http request where
  ```java
  request.getParameter("topping")
  ```
  is set to:
  ```sql
  abc’; DROP TABLE creditcards; --
  ```

- The effect is to generate the SQL query:
  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND toppings='abc';
  DROP TABLE creditcards ; --
  ```

- ‘--’ represents an SQL comment

Second-order SQL injection

- Use a previously stored value to do SQL injection
- Let’s say we have the following query:
  ```sql
  SELECT ssn FROM users
  WHERE uname="’"+$uname+"’"
  ```

- Later, can change own password:
  - set uname to “XXX’ OR uname= ’JANE”
Solutions?

- Defense-in-depth…
  - Use several solutions, as appropriate

- Blacklisting
- Whitelisting
- Sanitization
- Prepared statements/bind variables
- Mitigate the impact of SQL injections

Blacklisting?

- I.e., searching for/preventing ‘bad’ inputs
- E.g., for previous example:

```java
sql_query
    = "SELECT pizza, quantity, order_day "
    + "FROM orders "
    + "WHERE userid=" + session.getCurrentUserId()
    + " AND topping='""
    + kill_chars(request.getParameter("topping"))
    + "'"
```

- …where kill_chars() deletes, e.g., quotes and semicolons
Drawbacks of blacklisting

- How do you know if/when you’ve eliminated all possible ‘bad’ strings?
  - If you miss one, could allow successful attack
- Does not prevent first set of attacks (numeric values)
  - Although similar approach could be used, starts to get complex!
- May conflict with functionality of the database
  - E.g., user with name O’Brien

Whitelisting

- Check that user-provided input is in some set of values known to be safe
  - E.g., check that month is an integer in the right range
- If invalid input detected, better to reject it than to try to fix it
  - Fixes may introduce vulnerabilities
  - Principle of fail-safe defaults
Prepared statements/bind variables

- **Bind variables**: placeholders guaranteed to be data (not control), in correct format
  
  ```java
db.exec("SELECT ssn FROM users WHERE uname = ?", name)
  ```

- **Prepared statements**: allow creation of queries with bind variables
  
  ```java
  q = db.query("SELECT ssn FROM users WHERE uname = ?")
  q.execute(id)
  ```

- Parameters not involved in query parsing

**Another Example (Java)**

```java
PreparedStatement ps =
    db.prepareStatement("SELECT pizza, quantity, order_day " + "FROM orders WHERE userid=? AND order_month=?");

ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();

• Query parsed w/o parameters
• Bind variables are typed
```
Mitigating the impact

- **Limit privileges**
  - I.e., allow SELECT queries on the *orders* database, but no queries on *creditcards* database
  - Can limit commands, or tables to which access is given (or both)
  - *Principle of least privilege*
    - Not a complete fix, but it helps

- **Encrypt sensitive data stored in database**
  - E.g., orders in the clear but credit card numbers encrypted

Encryption
CryptDB: Protecting Confidentiality with Encrypted Query Processing

Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan
MIT CSAIL

Problem

- Confidential data leaks from databases
  - E.g., Sony Playstation Network, impacted 77 million personal information profiles

Threat 1: passive DB server attacks
Threat 2: any attacks on all servers
CryptDB in a nutshell

- Goal: protect confidentiality of data

1. Process SQL queries on encrypted data
2. Use fine-grained keys; chain these keys to user passwords based on access control

Contributions

1. First practical DBMS to process most SQL queries on encrypted data
   - Hide DB from sys. admins., outsource DB

2. Protects data of users logged out during attack, even when all servers are compromised
   - Limit leakage from compromised applications

3. Modest overhead: 26% throughput loss for TPC-C
4. No changes to DBMS (e.g., Postgres, MySQL)
Threat 1: Passive attacks to DB Server

- Application
  - plain query
  - decrypted results

- Proxy
  - transformed query
  - encrypted results

- DB Server
  - Under attack
  - Encrypted DB
  - Store schema, master key
  - No data storage
  - No query execution

- Process SQL queries on encrypted data

```
SELECT * FROM emp
WHERE salary = 100

SELECT * FROM table1
WHERE col3 = x5a8c34
```
Two techniques

1. Use SQL-aware set of encryption schemes
   - Most SQL uses a limited set of operations

2. Adjust encryption of database based on queries
### Encryption schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Construction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>AES in CBC</td>
<td>none</td>
</tr>
<tr>
<td>HOM</td>
<td>Paillier</td>
<td>+, *</td>
</tr>
<tr>
<td>SEARCH</td>
<td>Song et al., '00</td>
<td>word search</td>
</tr>
<tr>
<td>DET</td>
<td>AES in CMC</td>
<td>equality</td>
</tr>
<tr>
<td>JOIN</td>
<td>our new scheme</td>
<td>join</td>
</tr>
<tr>
<td>OPE</td>
<td>Boldyreva et al., '09</td>
<td>order</td>
</tr>
</tbody>
</table>

- **Highest Security**
- **Function**
  - e.g., sum
  - restricted ILIKE
  - e.g., =, !=, IN, COUNT, GROUP BY, DISTINCT
  - e.g., >, <, ORDER BY, SORT, MAX, MIN

**How to encrypt each data item?**

- Encryption schemes needed depend on queries
- May not know queries ahead of time

```
rank
'CEO' 'worker'

ALL?
```

Leaks order!
Confidentiality level

Queries encryption scheme exposed amount of leakage

- Encryption schemes exposed for each column are the most secure enabling queries
  - equality predicate on a column DET repeats
  - aggregation on a column HOM nothing
  - no filter on a column RND nothing

common in practice

Never reveals plaintext

Application protection

Arbitrary attacks on any servers

User password gives access to data allowed to user by access control policy

Protects data of logged out users during attack
Challenge: data sharing

1. How to enforce access control cryptographically? Key chains from user passwords
2. Capture read access policy of application at SQL level? Annotations
3. Process queries on encrypted data

### Implementation

- No change to the DBMS
- Portable: from Postgres to MySQL with 86 lines
- One-key: no change to applications
- Multi-user keys: annotations and login/logout
Queries not supported

- More complex operators, e.g., trigonometry
- Operations that require combining incompatible encryption schemes
  - e.g., $T1.a + T1.b > T2.c$

Extensions: split queries, precompute columns, or add new encryption schemes

Conclusions

CryptDB:
1. The first practical DBMS for running most standard queries on encrypted data
2. Protects data of users logged out during attack even when all servers are compromised
3. Modest overhead and no changes to DBMS

Website: http://css.csail.mit.edu/cryptdb/