Outline

SQL Security – 8.7

Two famous attacks

Two new trends

Optional material; May not have time to cover in class

Discretionary Access Control in SQL

GRANT privileges
ON object
TO users
[WITH GRANT OPTIONS]

privileges = SELECT | INSERT(column-name) | UPDATE(column-name) | DELETE | REFERENCES(column-name)
object = table | attribute

Examples

GRANT INSERT, DELETE ON Customers TO Yuppy WITH GRANT OPTIONS

Queries allowed to Yuppy:

INSERT INTO Customers(cid, name, address) VALUES(32940, ‘Joe Blow’, ‘Seattle’)
DELETE Customers
WHERE LastPurchaseDate < 1995

Queries denied to Yuppy:

SELECT Customer.address
FROM Customer
WHERE name = ‘Joe Blow’

Examples

GRANT SELECT ON Customers TO Michael

Now Michael can SELECT, but not INSERT or DELETE

Examples

GRANT SELECT ON Customers TO Michael WITH GRANT OPTIONS

Michael can say this:

GRANT SELECT ON Customers TO Yuppy

Now Yuppy can SELECT on Customers
Examples

GRANT UPDATE (price) ON Product TO Leah

Leah can update, but only Product.price, but not Product.name

Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW PublicCustomers
SELECT Name, Address
FROM Customers
GRANT SELECT ON PublicCustomers TO Fred

Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW BadCreditCustomers
SELECT * FROM Customers
WHERE Balance < 0
GRANT SELECT ON BadCreditCustomers TO John

Views and Security

- Each customer should see only her/his record

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW CustomerMary
SELECT * FROM Customers
WHERE name = 'Mary'
GRANT SELECT
ON CustomerMary TO Mary

Doesn’t scale.
Need row-level access control!

Recovery

REVOKE [GRANT OPTION FOR] privileges
ON object FROM users { RESTRICT | CASCADE }

Administrator says:

REVOKE SELECT ON Customers FROM David CASCADE

John loses SELECT privileges on BadCreditCustomers
Revocation

Joe: GRANT [...] TO Art ...
Art: GRANT [...] TO Bob ...
Bob: GRANT [...] TO Art ...
Joe: GRANT [...] TO Cal ...
Joe: REVOKE [...] FROM Art CASCADE

Same privilege, same object, GRANT OPTION

What happens ??

Summary of SQL Security

Limitations:
• No row level access control
• Table creator owns the data: that’s unfair !

Access control = great success story of the DB community...

… or spectacular failure:
• Only 30% assign privileges to users/roles
  – And then to protect entire tables, not columns

Summary (cont)

• Most policies in middleware: slow, error prone:
  – SAP has $10^4$ tables
  – GTE over $10^5$ attributes
  – A brokerage house has 80,000 applications
  – A US government entity thinks that it has 350K

• Today the database is not at the center of the policy administration universe

[Universe of the Policy Administration]

Two Famous Attacks

• SQL injection
• Sweeney’s example

SQL Injection

Your health insurance company lets you see the claims online:

First login:
User: fred
Password: ********

Now search through the claims :

Search claims by:
Dr. Lee

SELECT…FROM…WHERE doctor='Dr. Lee' and patientID='fred'
**SQL Injection**

Now try this:

Search claims by: `Dr. Lee' OR patientID = 'suciu'; --`

```
...WHERE doctor='Dr. Lee' OR patientID='suciu'; -- and patientID='fred'
```

Better:

Search claims by: `Dr. Lee' OR 1 = 1; --`

---

**SQL Injection**

The DBMS works perfectly. So why is SQL injection possible so often?

- Quick answer:
  - Poor programming: use stored procedures!
- Deeper answer:
  - Move policy implementation from apps to DB

---

**Latanya Sweeney’s Finding**

- In Massachusetts, the Group Insurance Commission (GIC) is responsible for purchasing health insurance for state employees
- GIC has to publish the data:

```
GIC(zip, dob, sex, diagnosis, procedure, ...)
```

---

**Latanya Sweeney’s Finding**

Sweeney paid $20 and bought the voter registration list for Cambridge Massachusetts:

```
GIC(zip, dob, sex, diagnosis, procedure, ...)
VOTER(name, party, ..., zip, dob, sex)
```

- William Weld (former governor) lives in Cambridge, hence is in VOTER
- 6 people in VOTER share his dob
- only 3 of them were man (same sex)
- Weld was the only one in that zip
- Sweeney learned Weld’s medical records!
Latanya Sweeney’s Finding

• All systems worked as specified, yet an important data has leaked

• How do we protect against that?

Some of today’s research in data security address breaches that happen even if all systems work correctly

Summary on Attacks

SQL injection:
• A correctness problem:
  – Security policy implemented poorly in the application

Sweeney’s finding:
• Beyond correctness:
  – Leakage occurred when all systems work as specified

Two Novel Techniques

• K-anonymity, information leakage
• Row-level access control

Information Leakage: k-Anonymity

Definition: each tuple is equal to at least k-1 others

Anonymizing: through suppression and generalization

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Age</th>
<th>Race</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stone</td>
<td>John</td>
<td></td>
<td>Afr-Am</td>
<td>Flue</td>
</tr>
<tr>
<td>R*</td>
<td>*</td>
<td></td>
<td></td>
<td>Measels</td>
</tr>
<tr>
<td>Stone</td>
<td>John</td>
<td></td>
<td>Afr-am</td>
<td>Pain</td>
</tr>
<tr>
<td>R*</td>
<td>*</td>
<td></td>
<td></td>
<td>Fever</td>
</tr>
</tbody>
</table>

Hard: NP-complete for suppression only
Approximations exists; but work poorly in practice

Information Leakage: Query-view Security

TABLE Employee(name, dept, phone)

<table>
<thead>
<tr>
<th>Secret Query</th>
<th>View(s)</th>
<th>Disclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(name)</td>
<td>V(name)</td>
<td>total</td>
</tr>
<tr>
<td>S(name, phone)</td>
<td>V1(name,dept)</td>
<td>big</td>
</tr>
<tr>
<td>S(name)</td>
<td>V(dept)</td>
<td>tiny</td>
</tr>
<tr>
<td>S(name)</td>
<td>V(name)</td>
<td>none</td>
</tr>
<tr>
<td>where dept=’HR’</td>
<td>V(dept)</td>
<td></td>
</tr>
<tr>
<td>where dept=’RD’</td>
<td>V(phone)</td>
<td></td>
</tr>
</tbody>
</table>

Fine-grained Access Control

Control access at the tuple level.

• Policy specification languages
• Implementation
Policy Specification Language

No standard, but usually based on parameterized views.

```
CREATE AUTHORIZATION VIEW PatientsForDoctors AS
SELECT Patient.*
FROM Patient, Doctor
WHERE Patient.doctorID = Doctor.ID
and Doctor.login = %currentUser
```

Two Semantics

- The Truman Model = filter semantics
  - transform reality
  - ACCEPT all queries
  - REWRITE queries
  - Sometimes misleading results

- The non-Truman model = deny semantics
  - reject queries
  - ACCEPT or REJECT queries
  - Execute query UNCHANGED
  - May define multiple security views for a user

Summary on Information Disclosure

- The theoretical research:
  - Exciting new connections between databases and information theory, probability theory, cryptography
  - [Abadi&Warinschi'05]

- The applications:
  - many years away

Summary of Fine Grained Access Control

- Trend in industry: label-based security
- Killer app: application hosting
  - Independent franchises share a single table at headquarters (e.g., Holiday Inn)
  - Application runs under requester’s label, cannot see other labels
  - Headquarters runs Read queries over them
- Oracle’s Virtual Private Database

[Abadi&Warinschi'05]