Database Security MidSem 2024

Q1 — Problems in Bell–LaPadula if Object Hierarchy is Not a Tree

Ans:

🔷 Background

In the Bell–LaPadula (BLP) model:

- Objects are arranged in a hierarchy.
- Security labels form a **lattice (tree-like structure)**.
- Information flows follow:
    - **No Read Up**
    - **No Write Down**

The hierarchy is expected to behave cleanly like a **tree** so dominance is clear.

If the object hierarchy becomes a **general graph (DAG or cyclic graph)**, problems arise.

🔷 Case 1: Directed Acyclic Graph (Multiple Parents)

Example

Suppose we have classifications:

- Top Secret (TS)
- Secret (S)
- Nuclear (N)
- Military (M)

Assume:

- TS dominates both N and M.
- N and M are incomparable (different compartments).

Now suppose an object O_1 belongs to **both Nuclear and Military compartments**.

The structure now becomes a DAG because O_1 has two parent security paths.

🔷 Problem 1: Ambiguous Dominance

If a subject has:

- Clearance = (Secret, Nuclear)

Can they read O_1?

- They dominate Nuclear.
- But do not dominate Military.

Since O_1 belongs to both compartments, dominance becomes ambiguous.

Thus:

- Access decision becomes unclear.
- Security policy becomes inconsistent.

In a tree, each node has one parent → dominance is clear.

🔷 Case 2: Cyclic Graph

Suppose hierarchy accidentally forms a cycle:

$$A > B$$

$$B > C$$

$$C > A$$

Now:

- A dominates B
- B dominates C
- C dominates A

By transitivity:

$$A > A$$

This breaks antisymmetry of partial order.

🔷 Problem 2: Security Collapse

If cycles exist:

- Every node eventually dominates every other.

- All security levels become equivalent.
- Confidentiality guarantees fail.

Thus:

- No meaningful separation remains.
- “No Write Down” and “No Read Up” lose meaning.

🔷 Problem 3: Violation of Lattice Properties

BLP requires:

- Reflexivity
- Transitivity
- Antisymmetry
- Unique least upper bound (LUB)

In general graphs:

- LUB may not exist.
- Two nodes may not have a unique join.
- Conflicting classifications arise.

This breaks formal lattice structure required for BLP.

🔷 Why Tree Structure Is Preferred

A tree ensures:

- Clear dominance chain
- No ambiguity in access decision
- Unique parent
- No cycles
- Well-defined lattice joins

🔷 Final Conclusion (Exam Ready)

If the object hierarchy forms a DAG or cyclic graph instead of a tree:

1. Access decisions become ambiguous (multiple parents).

2. Cycles collapse security levels.
3. Partial order properties may be violated.
4. Lattice structure required for Bell–LaPadula breaks.
5. Confidentiality guarantees fail.

Hence, Bell–LaPadula requires a properly defined lattice (tree-like hierarchy) to function correctly.

Q2 — Revocation Problem in ACLs and Capabilities

🔷 What is the Revocation Problem?

Revocation means **removing previously granted access rights** from a user.

The difficulty depends on how access control is implemented:

- **Access Control Lists (ACLs)** → permissions stored with objects.
- **Capabilities** → permissions stored with users.

🔷 1️⃣ Revocation in Access Control Lists (ACLs)

In ACLs:

- Each object maintains a list of users and their permissions.
- To revoke access of user U to object O:

→ Simply remove U's entry from O's ACL.

Advantage:

✓ Easy and direct
✓ Revocation is localized to one object

Problem:

If U has copied data or further granted rights (in discretionary systems), additional cleanup may be required.

🔷 2️⃣ Revocation in Capability Systems

In capabilities:

- Users possess unforgeable tokens (capabilities) granting access to objects.

- These tokens may be copied and distributed.

Problem:

To revoke access:

- System must find and invalidate all capabilities issued to that user.
- If capabilities are distributed or copied, revocation becomes difficult.
- System may not know where all copies exist.

Thus:
Revocation is harder in capability-based systems.

🔷 Efficient Way to Implement Revocation

Solution 1: For ACL Systems

Remove the user's entry from the object's ACL:

Command: Delete (User, Permission) from Object's ACL

This immediately blocks access.

Solution 2: For Capability Systems (Efficient Method)

Use one of the following techniques:

1. **Indirection (Pointer technique)**
    - Capability points to an entry in a table.
    - On revocation, invalidate that table entry.
    - All capabilities referencing it stop working.
2. **Key-based revocation**
    - Associate object with a secret key.
    - Change the key on revocation.
    - Reissue capabilities to authorized users only.

These allow efficient centralized revocation.

🔷 Final Summary

- ACLs: Revocation is simple — remove user from object's list.
- Capabilities: Revocation is difficult because tokens may be distributed.
- Efficient solution: Use indirection or key rotation so revocation can be done centrally without tracking all copies.

Q4 (a) — Why " ' OR 1=1 -- " Causes Successful Login

🔷 Given PHP Code

The SQL query constructed is:

```
SELECT *
FROM usertable
WHERE username = '$username'
AND password = '$password';
```

User input is directly concatenated into the query (no sanitization).

🔷 Malicious Input

Attacker sets:

```
user = ' OR 1=1 --
password = anything
```

So:

```
$username = ' OR 1=1 --
```

🔷 Resulting SQL Query

Substituting into query:

```
SELECT *
FROM usertable
WHERE username = '' OR 1=1 -- '
AND password = 'anything';
```

🔷 What Happens Now?

1. username = '' → may be false

2. OR 1=1 → always TRUE
3. -- → SQL comment operator
   Everything after -- is ignored.

So database actually sees:

SELECT *

FROM usertable

WHERE username = '' OR 1=1

The password condition is commented out.

🔷 Why This Logs In Successfully

Since:

1=11=11=1

is always true,

The WHERE condition becomes TRUE for all rows.

So:

- Query returns all users.
- $result->num_rows > 0
- Condition becomes TRUE → Login success.

🔷 Root Cause

The application:

- Directly concatenates user input into SQL.
- Does not sanitize or parameterize input.
- Allows user input to change query logic.

This is a classic **SQL Injection attack**.

Q4 (b) — Why addslashes() Prevents the Attack from (a)

🔷 Change Made in Code

Original code:

```
$username = $_GET[user];
$password = $_GET[pwd];
```

Modified code:

```
$username = addslashes($_GET[user]);
$password = addslashes($_GET[pwd]);
```

🔷 What Does addslashes() Do?

It adds a backslash (\) before special characters like:

- ' (single quote)
- " (double quote)
- \ (backslash)
- NULL

Example:

```
addslashes("' OR 1=1 --")
```

becomes

```
\' OR 1=1 --
```

🔷 What Happens Now to the Attack Input?

Attacker submits:

```
' OR 1=1 --
```

After addslashes(), it becomes:

```
\' OR 1=1 --
```

🔷 Resulting SQL Query

Original injection produced:

```
WHERE username = '' OR 1=1 -- '
```

But now query becomes:

```
WHERE username = '\' OR 1=1 -- '
AND password = '...';
```

Important point:

The ' is now escaped as \'.

That means:

- The quote does NOT terminate the string.
- It is treated as a literal character inside the string.
- The attacker cannot break out of the username string.

🔷 Why Injection Fails

Because:

- The attacker can no longer close the quote.
- OR 1=1 stays inside the string.
- Database searches for username literally equal to:

' OR 1=1 --

This username does not exist.

So:

- Query returns 0 rows.
- Login fails.

🔷 Root Idea

SQL injection works by:

- Breaking out of the string literal.

addslashes() prevents this by escaping quotes, so:

- The structure of the SQL query cannot be modified.

Q4 (c) — Does addslashes() Completely Solve the Problem?

🔷 Short Answer

No.
addslashes() does NOT completely prevent SQL injection when certain multi-byte character encodings (like GBK) are used.

🔷 Why the Problem Occurs

In the **GBK character set**:

- Some characters are **2 bytes**
- Some are **1 byte**

Given:

0x5c = \
0x27 = '
0xbf 0x27 = two characters (¿ and ')
0xbf 0x5c = one single Chinese character

Important part:

- 0xbf5c is interpreted by the database as **one character**.
- But 0xbf27 is interpreted as **two separate characters**.

🔷 What addslashes() Does

addslashes() escapes ' by inserting a backslash before it.

If attacker sends:

\xbf

(0xbf followed by 0x27)

Then addslashes() converts it to:

0xbf 0x5c 0x27

Because it inserts \ before '.

So resulting bytes become:

bf 5c 27

🔷 How GBK Interprets This

The database sees:

0xbf 0x5c

as ONE Chinese character
and then

0x27

as an unescaped single quote.

So the backslash gets "absorbed" into the multi-byte character!

The quote (') is no longer escaped.

🔷 Result

The attacker successfully injects:

' OR 1=1 --

because the quote is no longer escaped.

🔷 Malicious Input That Causes Successful Login

User value (in hex form):

0xbf 0x27 OR 1=1 --

In visible form:

¿' OR 1=1 --

After addslashes():

bf 5c 27 OR 1=1 --

Database interprets:

(ChineseChar) ' OR 1=1 --

So injection succeeds.

🔷 Final Conclusion

addslashes() fails in multi-byte encodings like GBK because:

- The inserted backslash may become part of a multi-byte character.
- The quote remains unescaped.
- SQL injection is still possible.

Proper solution is:

- Use prepared statements.
- Use parameterized queries.

- Use proper escaping functions based on database API.