Vertical propagation is more complicated; it limits the depth of the granting of privileges. Granting a privilege with a vertical propagation of zero is equivalent to granting the privilege with *no* GRANT OPTION. If account *A* grants a privilege to account *B* with the vertical propagation set to an integer number $j > 0$, this means that the account *B* has the GRANT OPTION on that privilege, but *B* can grant the privilege to other accounts only with a vertical propagation *less than j*. In effect, vertical propagation limits the sequence of GRANT OPTIONS that can be given from one account to the next based on a single original grant of the privilege.

We briefly illustrate horizontal and vertical propagation limits—which are *not available* currently in SQL or other relational systems—with an example. Suppose that A1 grants SELECT to A2 on the EMPLOYEE relation with horizontal propagation equal to 1 and vertical propagation equal to 2. A2 can then grant SELECT to at most one account because the horizontal propagation limitation is set to 1. Additionally, A2 cannot grant the privilege to another account except with vertical propagation set to 0 (no GRANT OPTION) or 1; this is because A2 must reduce the vertical propagation by at least 1 when passing the privilege to others. In addition, the horizontal propagation must be less than or equal to the originally granted horizontal propagation. For example, if account *A* grants a privilege to account *B* with the horizontal propagation set to an integer number $j > 0$, this means that *B* can grant the privilege to other accounts only with a horizontal propagation *less than or equal to j*. As this example shows, horizontal and vertical propagation techniques are designed to limit the depth and breadth of propagation of privileges.

24.3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: A user either has or does not have a certain privilege. In many applications, an *additional security policy* is needed that classifies data and users based on security classes. This approach, known as **mandatory access control (MAC)**, would typically be *combined* with the discretionary access control mechanisms described in Section 24.2. It is important to note that most commercial DBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications. Some DBMS vendors—for example, Oracle—have released special versions of their RDBMSs that incorporate mandatory access control for government use.

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where TS ≥ S ≥ C ≥ U, to illustrate our discussion. The commonly used model for multilevel security, known as the *Bell-LaPadula model*, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the **clearance** (classification) of a subject S as **class(*S*)** and to the **classification** of an object O as **class(*O*)**. Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless class(S) $\geq$ class(O). This is known as the **simple security property**.
2. A subject S is not allowed to write an object O unless class(S) $\leq$ class(O). This is known as the **star property** (or *-property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a **classification attribute** C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. The model we describe here is known as the *multilevel model*, because it allows classifications at multiple security levels. A **multilevel relation** schema R with n attributes would be represented as:

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

where each C_i represents the *classification attribute* associated with attribute A_i.

The value of the tuple classification attribute TC in each tuple t—which is the *highest* of all attribute classification values within t—provides a general classification for the tuple itself. Each attribute classification C_i provides a finer security classification for each attribute value within the tuple. The value of TC in each tuple t is the *highest* of all attribute classification values C_i within t.

The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering**. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the *apparent key*.

This leads to the concept of **polyinstantiation**,[4] where several tuples can have the same apparent key value but have different attribute values for users at different clearance levels.

We illustrate these concepts with the simple example of a multilevel relation shown in Figure 24.2(a), where we display the classification attribute values next to each attribute's value. Assume that the Name attribute is the apparent key, and consider the query **SELECT** * **FROM** EMPLOYEE. A user with security clearance *S* would see the same relation shown in Figure 24.2(a), since all tuple classifications are less than or equal to *S*. However, a user with security clearance *C* would not be allowed to see the values for Salary of 'Brown' and Job_performance of 'Smith', since they have higher classification. The tuples would be *filtered* to appear as shown in Figure 24.2(b), with Salary and Job_performance *appearing as null*. For a user with security clearance *U*, the filtering allows only the Name attribute of 'Smith' to appear, with all the other

(a) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

(b) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

(c) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

(d) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

Figure 24.2
A multilevel relation to illustrate multilevel security. (a) The original EMPLOYEE tuples. (b) Appearance of EMPLOYEE after filtering for classification C users. (c) Appearance of EMPLOYEE after filtering for classification U users. (d) Polyinstantiation of the Smith tuple.

[4]This is similar to the notion of having multiple versions in the database that represent the same real-world object.

attributes appearing as null (Figure 24.2(c)). Thus, filtering introduces null values for attribute values whose security classification is higher than the user's security clearance.

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the *same* security classification within each individual tuple. Additionally, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user can see the key if the user is permitted to see any part of the tuple. Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

To illustrate polyinstantiation further, suppose that a user with security clearance *C* tries to update the value of Job_performance of 'Smith' in Figure 24.2 to 'Excellent'; this corresponds to the following SQL update being submitted by that user:

```
UPDATE   EMPLOYEE
SET      Job_performance = 'Excellent'
WHERE    Name = 'Smith';
```

Since the view provided to users with security clearance *C* (see Figure 24.2(b)) permits such an update, the system should not reject it; otherwise, the user could *infer* that some nonnull value exists for the Job_performance attribute of 'Smith' rather than the null value that appears. This is an example of inferring information through what is known as a **covert channel**, which should not be permitted in highly secure systems (see Section 24.6.1). However, the user should not be allowed to overwrite the existing value of Job_performance at the higher classification level. The solution is to create a **polyinstantiation** for the 'Smith' tuple at the lower classification level *C*, as shown in Figure 24.2(d). This is necessary since the new tuple cannot be filtered from the existing tuple at classification *S*.

The basic update operations of the relational model (INSERT, DELETE, UPDATE) must be modified to handle this and similar situations, but this aspect of the problem is outside the scope of our presentation. We refer the interested reader to the Selected Bibliography at the end of this chapter for further details.

24.3.1 Comparing Discretionary Access Control and Mandatory Access Control

Discretionary access control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains. The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs. The reason is that discretionary authorization models do not impose any control on how information is propagated and used once it has been accessed by users authorized to do so. By contrast, mandatory policies ensure a high degree of protection—in a way, they prevent any illegal flow of information. Therefore, they are suitable for military and high security types of applications, which require a higher degree of protection. However, mandatory policies have the drawback of being too rigid in that they require a strict classification of subjects and objects into security levels, and therefore they are applicable to few environments. In many practical situations, discretionary policies are preferred because they offer a better tradeoff between security and applicability.

24.3.2 Role-Based Access Control

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems. Its basic notion is that privileges and other permissions are associated with organizational **roles**, rather than individual users. Individual users are then assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands discussed in Section 24.2 can then be used to assign and revoke privileges from roles, as well as for individual users when needed. For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, department manager, and so on. Multiple individuals can be assigned to each role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

RBAC can be used with traditional discretionary and mandatory access controls; it ensures that only authorized users in their specified roles are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to several roles, but it maps to one user or a single subject only. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.

Separation of duties is another important requirement in various commercial DBMSs. It is needed to prevent one user from doing work that requires the involvement of two or more people, thus preventing collusion. One method in which separation of duties can be successfully implemented is with mutual exclusion of roles. Two roles are said to be **mutually exclusive** if both the roles cannot be used simultaneously by the user. **Mutual exclusion of roles** can be categorized into two types, namely *authorization time exclusion (static)* and *runtime exclusion (dynamic)*. In authorization time exclusion, two roles that have been specified as mutually exclusive cannot be part of a user's authorization at the same time. In runtime exclusion, both these roles can be authorized to one user but cannot be activated by the user at the same time. Another variation in mutual exclusion of roles is that of complete and partial exclusion.

The **role hierarchy** in RBAC is a natural way to organize roles to reflect the organization's lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and antisymmetric. In other words, if a user has one role, the user automatically has roles lower in the hierarchy. Defining a role hierarchy involves choosing the type of hierarchy and the roles, and then implementing the hierarchy by granting roles to other roles. Role hierarchy can be implemented in the following manner:

```
GRANT ROLE full_time TO employee_type1
GRANT ROLE intern TO employee_type2
```

The above are examples of granting the roles *full_time* and *intern* to two types of employees.

Another issue related to security is *identity management*. **Identity** refers to a unique name of an individual person. Since the legal names of persons are not necessarily unique, the identity of a person must include sufficient additional information to make the complete name unique. Authorizing this identity and managing the schema of these identities is called **Identity Management.** Identity Management addresses how organizations can effectively authenticate people and manage their access to confidential information. It has become more visible as a business requirement across all industries affecting organizations of all sizes. Identity Management administrators constantly need to satisfy application owners while keeping expenditures under control and increasing IT efficiency.

Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as the time and duration of role activations, and timed triggering of a role by an activation of another role. Using an RBAC model is a highly desirable goal for addressing the key security requirements of Web-based applications. Roles can be assigned to workflow tasks so that a user with any of the roles related to a task may be authorized to execute it and may play a certain role only for a certain duration.

RBAC models have several desirable features, such as flexibility, policy neutrality, better support for security management and administration, and other aspects that make them attractive candidates for developing secure Web-based applications. These features are lacking in DAC and MAC models. In addition, RBAC models include the capabilities available in traditional DAC and MAC policies. Furthermore, an RBAC model provides mechanisms for addressing the security issues related to the execution of tasks and workflows, and for specifying user-defined and organization-specific policies. Easier deployment over the Internet has been another reason for the success of RBAC models.

24.3.3 Label-Based Security and Row-Level Access Control

Many commercial DBMSs currently use the concept of row-level access control, where sophisticated access control rules can be implemented by considering the data row by row. In row-level access control, each data row is given a label, which is used to store information about data sensitivity. Row-level access control provides finer granularity of data security by allowing the permissions to be set for each row and not just for the table or column. Initially the user is given a default session label by the database administrator. Levels correspond to a hierarchy of data-sensitivity levels to exposure or corruption, with the goal of maintaining privacy or security. Labels are used to prevent unauthorized users from viewing or altering certain data. A user having a low authorization level, usually represented by a low number, is denied access to data having a higher-level number. If no such label is given to a row, a row label is automatically assigned to it depending upon the user's session label.

A policy defined by an administrator is called a **Label Security policy.** Whenever data affected by the policy is accessed or queried through an application, the policy is automatically invoked. When a policy is implemented, a new column is added to each row in the schema. The added column contains the label for each row that reflects the sensitivity of the row as per the policy. Similar to MAC, where each user has a security clearance, each user has an identity in label-based security. This user's identity is compared to the label assigned to each row to determine whether the user has access to view the contents of that row. However, the user can write the label value himself, within certain restrictions and guidelines for that specific row. This label can be set to a value that is between the user's current session label and the user's minimum level. The DBA has the privilege to set an initial default row label.

The Label Security requirements are applied on top of the DAC requirements for each user. Hence, the user must satisfy the DAC requirements and then the label security requirements to access a row. The DAC requirements make sure that the user is legally authorized to carry on that operation on the schema. In most applications, only some of the tables need label-based security. For the majority of the application tables, the protection provided by DAC is sufficient.

Security policies are generally created by managers and human resources personnel. The policies are high-level, technology neutral, and relate to risks. Policies are a result of management instructions to specify organizational procedures, guiding principles, and courses of action that are considered to be expedient, prudent, or advantageous. Policies are typically accompanied by a definition of penalties and countermeasures if the policy is transgressed. These policies are then interpreted and converted to a set of label-oriented policies by the **Label Security administrator**, who defines the security labels for data and authorizations for users; these labels and authorizations govern access to specified protected objects.

Suppose a user has SELECT privileges on a table. When the user executes a SELECT statement on that table, Label Security will automatically evaluate each row returned by the query to determine whether the user has rights to view the data. For example, if the user has a sensitivity of 20, then the user can view all rows having a security level of 20 or lower. The level determines the sensitivity of the information contained in a row; the more sensitive the row, the higher its security label value. Such Label Security can be configured to perform security checks on UPDATE, DELETE, and INSERT statements as well.

24.3.4 XML Access Control

With the worldwide use of XML in commercial and scientific applications, efforts are under way to develop security standards. Among these efforts are digital signatures and encryption standards for XML. The XML Signature Syntax and Processing specification describes an XML syntax for representing the associations between cryptographic signatures and XML documents or other electronic resources. The specification also includes procedures for computing and verifying XML signatures. An XML digital signature differs from other protocols for message signing, such as **PGP** (**Pretty Good Privacy**—a confidentiality and authentication service that can be used for electronic mail and file storage application), in its support for signing only specific portions of the XML tree (see Chapter 12) rather than the complete document. Additionally, the XML signature specification defines mechanisms for countersigning and transformations—so-called *canonicalization* to ensure that two instances of the same text produce the same digest for signing even if their representations differ slightly, for example, in typographic white space.

The XML Encryption Syntax and Processing specification defines XML vocabulary and processing rules for protecting confidentiality of XML documents in whole or in part and of non-XML data as well. The encrypted content and additional processing information for the recipient are represented in well-formed XML so that the result can be further processed using XML tools. In contrast to other commonly used technologies for confidentiality such as SSL (Secure Sockets Layer—a leading Internet security protocol), and virtual private networks, XML encryption also applies to parts of documents and to documents in persistent storage.

24.3.5 Access Control Policies for E-Commerce and the Web

Electronic commerce (**e-commerce**) environments are characterized by any transactions that are done electronically. They require elaborate access control policies that go beyond traditional DBMSs. In conventional database environments, access control is usually performed using a set of authorizations stated by security officers or users according to some security policies. Such a simple paradigm is not well suited for a dynamic environment like e-commerce. Furthermore, in an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience. Such peculiarities call for more flexibility in specifying access control policies. The access control mechanism must be flexible enough to support a wide spectrum of heterogeneous protection objects.

A second related requirement is the support for content-based access control. **Content-based access control** allows one to express access control policies that take the protection object content into account. In order to support content-based access control, access control policies must allow inclusion of conditions based on the object content.

A third requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications rather than on specific and individual characteristics (for example, user IDs). A possible solution, to better take into account user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age or position or role within an organization). For instance, by using credentials, one can simply formulate policies such as *Only permanent staff with five or more years of service can access documents related to the internals of the system.*

It is believed that the XML is expected to play a key role in access control for e-commerce applications[5] because XML is becoming the common representation language for document interchange over the Web, and is also becoming the language for e-commerce. Thus, on the one hand there is the need to make XML representations secure, by providing access control mechanisms specifically tailored to the protection of XML documents. On the other hand, access control information (that is, access control policies and user credentials) can be expressed using XML itself. The **Directory Services Markup Language** (DSML) is a representation of directory service information in XML syntax. It provides a foundation for a standard for communicating with the directory services that will be responsible for providing and authenticating user credentials. The uniform presentation of both protection objects and access control policies can be applied to policies and credentials themselves. For instance, some credential properties (such as the user name) may be accessible to everyone, whereas other properties may be visible only to a restricted class of users. Additionally, the use of an XML-based language for specifying credentials and access control policies facilitates secure credential submission and export of access control policies.

24.4 SQL Injection

SQL Injection is one of the most common threats to a database system. We will discuss it in detail later in this section. Some of the other attacks on databases that are quite frequent are:

- **Unauthorized privilege escalation.** This attack is characterized by an individual attempting to elevate his or her privilege by attacking vulnerable points in the database systems.
- **Privilege abuse.** While the previous attack is done by an unauthorized user, this attack is performed by a privileged user. For example, an administrator who is allowed to change student information can use this privilege to update student grades without the instructor's permission.
- **Denial of service.** A **Denial of Service (DOS) attack** is an attempt to make resources unavailable to its intended users. It is a general attack category in which access to network applications or data is denied to intended users by overflowing the buffer or consuming resources.
- **Weak Authentication.** If the user authentication scheme is weak, an attacker can impersonate the identity of a legitimate user by obtaining their login credentials.

[5]See Thuraisingham et al. (2001).

24.4.1 SQL Injection Methods

As we discussed in Chapter 14, Web programs and applications that access a database can send commands and data to the database, as well as display data retrieved from the database through the Web browser. In an **SQL Injection attack**, the attacker injects a string input through the application, which changes or manipulates the SQL statement to the attacker's advantage. An SQL Injection attack can harm the database in various ways, such as unauthorized manipulation of the database, or retrieval of sensitive data. It can also be used to execute system level commands that may cause the system to deny service to the application. This section describes types of injection attacks.

SQL Manipulation. A manipulation attack, which is the most common type of injection attack, changes an SQL command in the application—for example, by adding conditions to the WHERE-clause of a query, or by expanding a query with additional query components using set operations such as UNION, INTERSECT, or MINUS. Other types of manipulation attacks are also possible. A typical manipulation attack occurs during database login. For example, suppose that a simplistic authentication procedure issues the following query and checks to see if any rows were returned:

```
SELECT * FROM users WHERE username = 'jake' and PASSWORD =
    'jakespasswd'.
```

The attacker can try to change (or manipulate) the SQL statement, by changing it as follows:

```
SELECT * FROM users WHERE username = 'jake' and (PASSWORD =
    'jakespasswd' or 'x' = 'x')
```

As a result, the attacker who knows that 'jake' is a valid login of some user is able to log into the database system as 'jake' without knowing his password and is able to do everything that 'jake' may be authorized to do to the database system.

Code Injection. This type of attack attempts to add additional SQL statements or commands to the existing SQL statement by exploiting a computer bug, which is caused by processing invalid data. The attacker can inject or introduce code into a computer program to change the course of execution. Code injection is a popular technique for system hacking or cracking to gain information.

Function Call Injection. In this kind of attack, a database function or operating system function call is inserted into a vulnerable SQL statement to manipulate the data or make a privileged system call. For example, it is possible to exploit a function that performs some aspect related to network communication. In addition, functions that are contained in a customized database package, or any custom database function, can be executed as part of an SQL query. In particular, dynamically created SQL queries (see Chapter 13) can be exploited since they are constructed at run time.

For example, the *dual* table is used in the FROM clause of SQL in Oracle when a user needs to run SQL that does not logically have a table name. To get today's date, we can use:

```
SELECT SYSDATE FROM dual;
```

The following example demonstrates that even the simplest SQL statements can be vulnerable.

```
SELECT TRANSLATE ('user input', 'from_string', 'to_string') FROM dual;
```

Here, TRANSLATE is used to replace a string of characters with another string of characters. The TRANSLATE function above will replace the characters of the 'from_string' with the characters in the 'to_string' one by one. This means that the *f* will be replaced with the *t*, the *r* with the *o*, the *o* with the _, and so on.

This type of SQL statement can be subjected to a function injection attack. Consider the following example:

```
SELECT TRANSLATE (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || ",
    '98765432', '9876') FROM dual;
```

The user can input the string (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || "), where || is the concatenate operator, thus requesting a page from a Web server. UTL_HTTP makes Hypertext Transfer Protocol (HTTP) callouts from SQL. The REQUEST object takes a URL ('http://129.107.2.1/' in this example) as a parameter, contacts that site, and returns the data (typically HTML) obtained from that site. The attacker could manipulate the string he inputs, as well as the URL, to include other functions and do other illegal operations. We just used a dummy example to show conversion of '98765432' to '9876', but the user's intent would be to access the URL and get sensitive information. The attacker can then retrieve useful information from the database server—located at the URL that is passed as a parameter—and send it to the Web server (that calls the TRANSLATE function).

24.4.2 Risks Associated with SQL Injection

SQL injection is harmful and the risks associated with it provide motivation for attackers. Some of the risks associated with SQL injection attacks are explained below.

- **Database Fingerprinting.** The attacker can determine the type of database being used in the backend so that he can use database-specific attacks that correspond to weaknesses in a particular DBMS.
- **Denial of Service.** The attacker can flood the server with requests, thus denying service to valid users, or they can delete some data.
- **Bypassing Authentication.** This is one of the most common risks, in which the attacker can gain access to the database as an authorized user and perform all the desired tasks.

- **Identifying Injectable Parameters.** In this type of attack, the attacker gathers important information about the type and structure of the back-end database of a Web application. This attack is made possible by the fact that the default error page returned by application servers is often overly descriptive.
- **Executing Remote Commands.** This provides attackers with a tool to execute arbitrary commands on the database. For example, a remote user can execute stored database procedures and functions from a remote SQL interactive interface.
- **Performing Privilege Escalation.** This type of attack takes advantage of logical flaws within the database to upgrade the access level.

24.4.3 Protection Techniques against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web-accessible procedures and functions. This section describes some of these techniques.

Bind Variables (Using Parameterized Statements). The use of bind variables (also known as *parameters*; see Chapter 13) protects against injection attacks and also improves performance.

Consider the following example using Java and JDBC:

```
PreparedStatement stmt = conn.prepareStatement( "SELECT * FROM
    EMPLOYEE WHERE EMPLOYEE_ID=? AND PASSWORD=?");
stmt.setString(1, employee_id);
stmt.setString(2, password);
```

Instead of embedding the user input into the statement, the input should be bound to a parameter. In this example, the input '1' is assigned (bound) to a bind variable 'employee_id' and input '2' to the bind variable 'password' instead of directly passing string parameters.

Filtering Input (Input Validation). This technique can be used to remove escape characters from input strings by using the SQL `Replace` function. For example, the delimiter single quote (') can be replaced by two single quotes (''). Some SQL Manipulation attacks can be prevented by using this technique, since escape characters can be used to inject manipulation attacks. However, because there can be a large number of escape characters, this technique is not reliable.

Function Security. Database functions, both standard and custom, should be restricted, as they can be exploited in the SQL function injection attacks.