CHAPTER 5 DATABASE AND DATA CENTER SECURITY

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the unique need for database security, separate from ordinary computer security measures.
- Present an overview of the basic elements of a database management system.
- Present an overview of the basic elements of a relational database system.
- Define and explain SQL injection attacks.
- Compare and contrast different approaches to database access control.
- Explain how inference poses a security threat in database systems.
- Discuss the use of encryption in a database system.
- Discuss security issues related to data centers.

This chapter looks at the unique security issues that relate to databases. The focus of this chapter is on relational database management systems (RDBMS). The relational approach dominates industry, government, and research sectors, and is likely to do so for the foreseeable future. We begin with an overview of the need for database-specific security techniques. Then we provide a brief introduction to database management systems, followed by an overview of relational databases. Next, we look at the issue of database access control, followed by a discussion of the inference threat. Then, we examine database encryption. Finally, we will examine the security issues related to the deployment of large data centers.

5.1 THE NEED FOR DATABASE SECURITY

Organizational databases tend to concentrate sensitive information in a single logical system. Examples include:

- Corporate financial data
- Confidential phone records
- Customer and employee information, such as name, Social Security number, bank account information, and credit card information
- Proprietary product information
- Health care information and medical records

For many businesses and other organizations, it is important to be able to provide customers, partners, and employees with access to this information. But such information can be targeted by internal and external threats of misuse or unauthorized change. Accordingly, security specifically tailored to databases is an increasingly important component of an overall organizational security strategy.

[BENN06] cites the following reasons why database security has not kept pace with the increased reliance on databases:

1. There is a dramatic imbalance between the complexity of modern database management systems (DBMS) and the security techniques used to protect these critical systems. A DBMS is a very complex, large piece of software, providing many options, all of which need to be well understood and then secured to avoid data breaches. Although security techniques have advanced, the increasing complexity of the DBMS—with many new features and services—has brought a number of new vulnerabilities and the potential for misuse.
2. Databases have a sophisticated interaction protocol called the Structured Query Language (SQL), which is far more complex, than for example, the Hypertext Transfer Protocol (HTTP) used to interact with a Web service. Effective database security requires a strategy based on a full understanding of the security vulnerabilities of SQL.
3. The typical organization lacks full-time database security personnel. The result is a mismatch between requirements and capabilities. Most organizations have a staff of database administrators, whose job is to manage the database to ensure availability, performance, correctness, and ease of use. Such administrators may have limited knowledge of security and little available time to master and apply security techniques. On

the other hand, those responsible for security within an organization may have very limited understanding of database and DBMS technology.

4. Most enterprise environments consist of a heterogeneous mixture of database platforms (Oracle, IBM DB2 and Informix, Microsoft, Sybase, etc.), enterprise platforms (Oracle E-Business Suite, PeopleSoft, SAP, Siebel, etc.), and OS platforms (UNIX, Linux, z/OS, and Windows, etc.). This creates an additional complexity hurdle for security personnel.

An additional recent challenge for organizations is their increasing reliance on cloud technology to host part or all of the corporate database. This adds an additional burden to the security staff.

5.2 DATABASE MANAGEMENT SYSTEMS

In some cases, an organization can function with a relatively simple collection of files of data. Each file may contain text (e.g., copies of memos and reports) or numerical data (e.g., spreadsheets). A more elaborate file consists of a set of records. However, for an organization of any appreciable size, a more complex structure known as a database is required. A **database** is a structured collection of data stored for use by one or more applications. In addition to data, a database contains the relationships between data items and groups of data items. As an example of the distinction between data files and a database, consider the following: A simple personnel file might consist of a set of records, one for each employee. Each record gives the employee's name, address, date of birth, position, salary, and other details needed by the personnel department. A personnel database includes a personnel file, as just described. It may also include a time and attendance file, showing for each week the hours worked by each employee. With a database organization, these two files are tied together so a payroll program can extract the information about time worked and salary for each employee to generate paychecks.

Accompanying the database is a **database management system (DBMS)**, which is a suite of programs for constructing and maintaining the database and for offering ad hoc query facilities to multiple users and applications. A **query language** provides a uniform interface to the database for users and applications.

Figure 5.1 provides a simplified block diagram of a DBMS architecture. Database designers and administrators make use of a data definition language (DDL) to define the database logical structure and procedural properties, which are represented by a set of database description tables. A data manipulation language (DML) provides a powerful set of tools for application developers. Query languages are declarative languages designed to support end users. The database management system makes use of the database description tables to manage the physical database. The interface to the database is through a file manager module and a transaction manager module. In addition to the database description table, two other tables support the DBMS. The DBMS uses authorization tables to ensure the user has permission to execute the query language statement on the database. The concurrent access table prevents conflicts when simultaneous conflicting commands are executed.

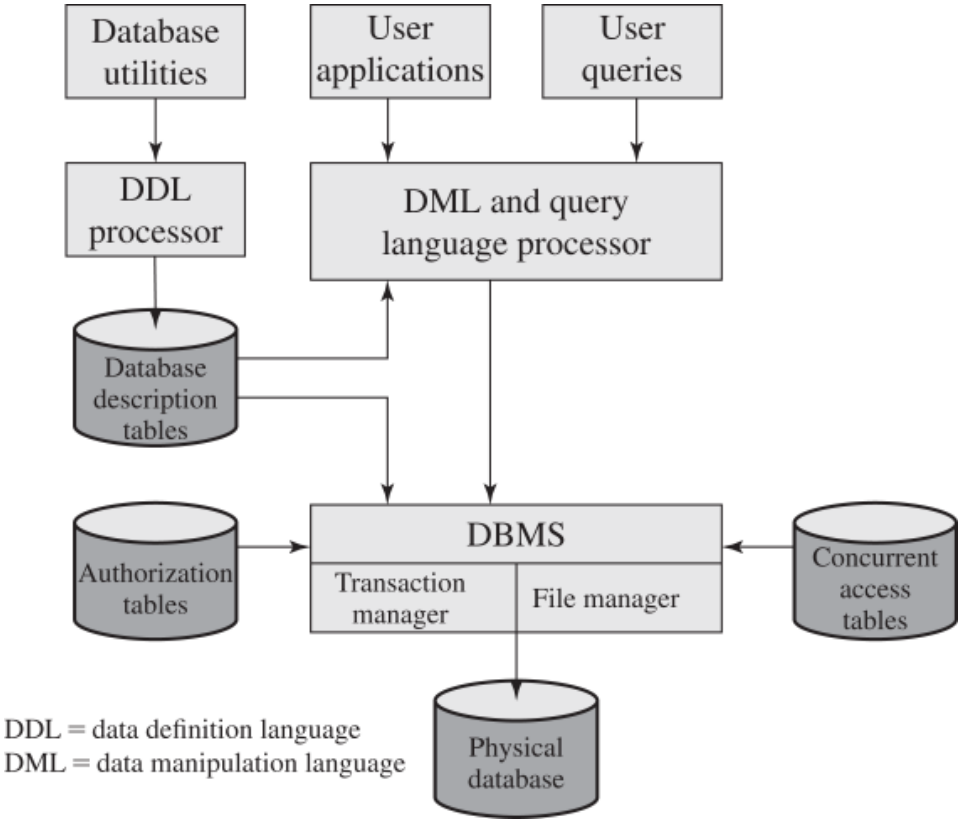


Figure 5.1 DBMS Architecture

Database systems provide efficient access to large volumes of data and are vital to the operation of many organizations. Because of their complexity and criticality, database systems generate security requirements that are beyond the capability of typical OS-based security mechanisms or stand-alone security packages.

Operating system security mechanisms typically control read and write access to entire files. So, they could be used to allow a user to read or to write any information in, for example, a personnel file. But they could not be used to limit access to specific records or fields in that file. A DBMS typically does allow this type of more detailed access control to be specified. It also usually enables access controls to be specified over a wider range of commands, such as to select, insert, update, or delete specified items in the database. Thus, security services and mechanisms are needed that are designed specifically for, and integrated with, database systems.

5.3 RELATIONAL DATABASES

The basic building block of a relational database is a table of data, consisting of rows and columns, similar to a spreadsheet. Each column holds a particular type of data, while each row contains a specific value for each column. Ideally, the table has at least one column in which each value is unique, thus serving as an identifier for a given entry. For example, a typical telephone directory contains one entry for each subscriber, with columns for name, telephone number, and address. Such a table is called a flat file because it is a single two-dimensional (rows and columns) file. In a flat file, all of the data are stored in a single table. For the telephone directory, there might be a number of subscribers with the same name, but the telephone numbers should be unique, so the telephone number serves as a unique identifier for a row. However, two or more people sharing the same phone number might each be listed in the directory. To continue to hold all of the data for the telephone directory in a single table and to provide for a unique identifier for each row, we could require a separate column for secondary subscriber, tertiary subscriber, and so on. The result would be that for each telephone number in use, there is a single entry in the table.

The drawback of using a single table is that some of the column positions for a given row may be blank (not used). In addition, any time a new service or new type of information is incorporated in the database, more columns must be added and the database and accompanying software must be redesigned and rebuilt.

The relational database structure enables the creation of multiple tables tied together by a unique identifier that is present in all tables. **Figure 5.2** shows how new services and features can be added to the telephone database without reconstructing the main table. In this example, there is a primary table with basic information for each telephone number. The telephone number serves as a primary key. The database administrator can then define a new table with a column for the primary key and other columns for other information.

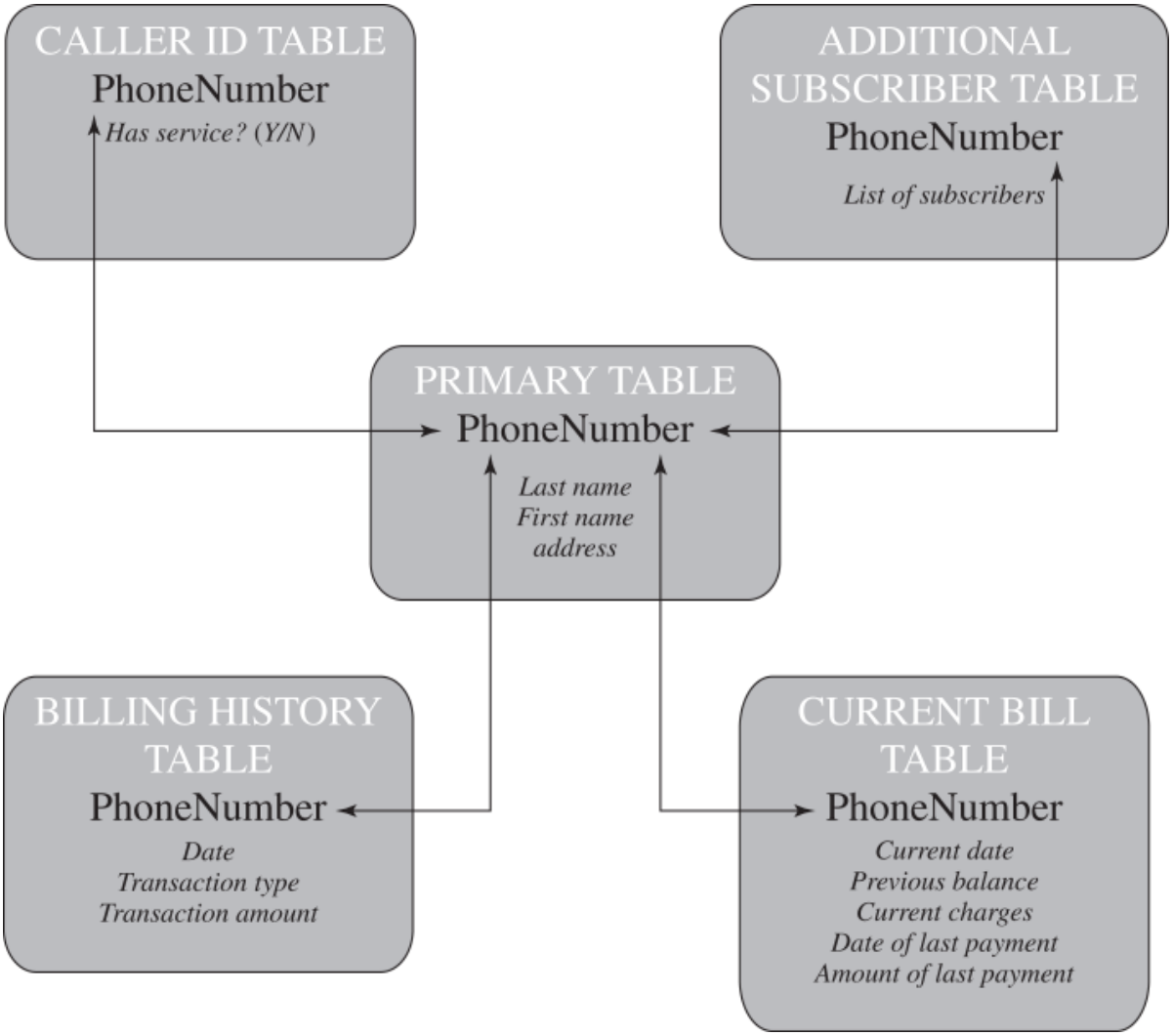


Figure 5.2 Example Relational Database Model

A relational database uses multiple tables related to one another by a designated key; in this case the key is the Phone-Number field.

Users and applications use a relational query language to access the database. The query language uses declarative statements rather than the procedural instructions of a programming language. In essence, the query language allows the user to request selected items of data from all records that fit a given set of criteria. The software then figures out how to extract the requested data from one or more tables. For example, a telephone company representative could retrieve a subscriber's billing information as well as the status of special services or the latest payment received, all displayed on one screen.

Elements of a Relational Database System

In relational database parlance, the basic building block is a **relation**, which is a flat table. Rows

are referred to as **tuples**, and columns are referred to as **attributes** (see **Table 5.1**). A **primary key** is defined to be a portion of a row used to uniquely identify a row in a table; the primary key consists of one or more column names. In the example of **Figure 5.2**, a single attribute, PhoneNumber, is sufficient to uniquely identify a row in a particular table. An abstract model of a relational database table is shown as **Figure 5.3**. There are *N* individuals, or entities, in the table and *M* attributes. Each attribute Aj has | Aj | possible values, with xij denoting the value of attribute *j* for entity *i*.

Table 5.1 Basic Terminology for Relational Databases

Formal Name	Common Name	Also Known As
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

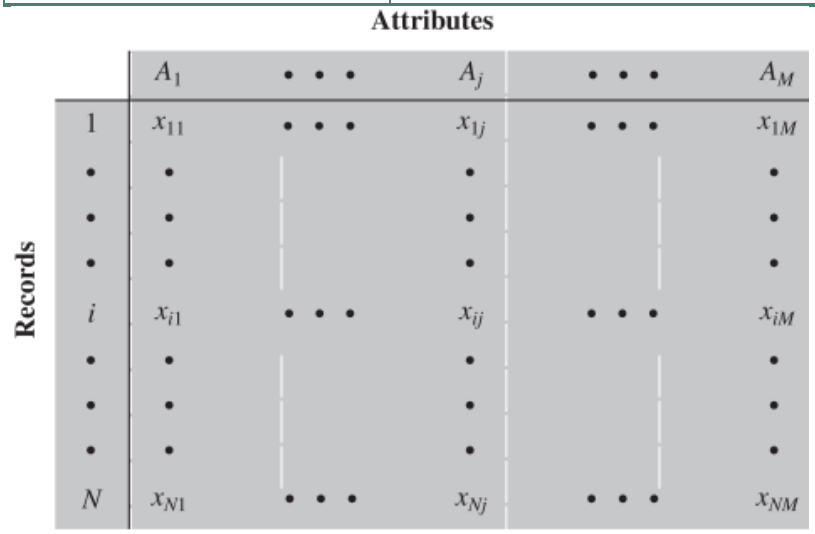


Figure 5.3 Abstract Model of a Relational Database

To create a relationship between two tables, the attributes that define the primary key in one table must appear as attributes in another table, where they are referred to as a **foreign key**. Whereas the value of a primary key must be unique for each tuple (row) of its table, a foreign key value can appear multiple times in a table, so there is a one-to-many relationship between a row in the table with the primary key and rows in the table with the foreign key. **Figure 5.4a** provides an example. In the Department table, the department ID (*Did*) is the primary key; each value is unique. This table gives the ID, name, and account number for each department. The Employee table contains the name, salary code, employee ID, and phone number of each employee. The Employee table

also indicates the department to which each employee is assigned by including *Did*. *Did* is identified as a foreign key and provides the relationship between the Employee table and the Department table.

Department Table

Did	Dname	Dacctno
4	human resources	528221
8	education	202035
9	accounts	709257
13	public relations	755827
15	services	223945

Primary key

Employee Table

Ename	Did	Salarycode	Eid	Ephone
Robin	15	23	2345	6127092485
Neil	13	12	5088	6127092246
Jasmine	4	26	7712	6127099348
Cody	15	22	9664	6127093148
Holly	8	23	3054	6127092729
Robin	8	24	2976	6127091945
Smith	9	21	4490	6127099380

Foreign key — Primary key

(a) Two tables in a relational database

Dname	Ename	Eid	Ephone
human resources	Jasmine	7712	6127099348
education	Holly	3054	6127092729
education	Robin	2976	6127091945
accounts	Smith	4490	6127099380
public relations	Neil	5088	6127092246
services	Robin	2345	6127092485
services	Cody	9664	6127093148

(b) A view derived from the database

Figure 5.4 Relational Database Example

A **view** is a virtual table. In essence, a view is the result of a query that returns selected rows and columns from one or more tables. **Figure 5.4b** is a view that includes the employee name, ID, and phone number from the Employee table and the corresponding department name from the Department table. The linkage is the *Did*, so the view table includes data from each row of the Employee table, with additional data from the Department table. It is also possible to construct a view from a single table. For example, one view of the Employee table consists of all rows, with the salary code column deleted. A view can be qualified to include only some rows and/or some columns. For example, a view can be defined consisting of all rows in the Employee table for which the Did=15.

Views are often used for security purposes. A view can provide restricted access to a relational database so a user or application only has access to certain rows or columns.

Structured Query Language

Structured Query Language (SQL) is a standardized language that can be used to define schema, manipulate, and query data in a relational database. There are several versions of the ANSI/ISO standard and a variety of different implementations, but all follow the same basic syntax and semantics.

For example, the two tables in **Figure 5.4a** are defined as follows:

```
CREATE TABLE department (
   Did INTEGER PRIMARY KEY,
   Dname CHAR (30),
   Dacctno CHAR (6) )
CREATE TABLE employee (
   Ename CHAR (30),
   Did INTEGER,
   SalaryCode INTEGER,
   Eid INTEGER PRIMARY KEY,
   Ephone CHAR (10),
   FOREIGN KEY (Did) REFERENCES department (Did) )
```

The basic command for retrieving information is the SELECT statement. Consider this example:

```
SELECT Ename, Eid, Ephone
   FROM Employee
   WHERE Did = 15
```

This query returns the Ename, Eid, and Ephone fields from the Employee table for all employees assigned to department 15.

The view in **Figure 5.4b** is created using the following SQL statement:

```
CREATE VIEW newtable (Dname, Ename, Eid, Ephone)
AS SELECT D.Dname E.Ename, E.Eid, E.Ephone
FROM Department D Employee E
WHERE E.Did = D.Did
```

The preceding are just a few examples of SQL functionality. SQL statements can be used to

create tables, insert and delete data in tables, create views, and retrieve data with query statements.

5.4 SQL INJECTION ATTACKS

The SQL injection (SQLi) attack is one of the most prevalent and dangerous network-based security threats. Consider the following reports:

1. The July 2013 Imperva Web Application Attack Report [IMPE13] surveyed a cross section of Web application servers in industry and monitored eight different types of common attacks. The report found that SQLi attacks ranked first or second in total number of attack incidents, the number of attack requests per attack incident, and average number of days per month that an application experienced at least one attack incident. Imperva observed a single website that received 94,057 SQL injection attack requests in one day.
2. The Open Web Application Security Project's 2013 report [OWAS13] on the 10 most critical Web application security risks listed injection attacks, especially SQLi attacks, as the top risk. This ranking is unchanged from its 2010 report.
3. The Veracode 2016 State of Software Security Report [VERA16] found that percentage of applications affected by SQLi attacks is around 35%.
4. The Trustwave 2016 Global Security Report [TRUS16] lists SQLi attacks as one of the top two intrusion techniques. The report notes that SQLi can pose a significant threat to sensitive data such as personally identifiable information (PII) and credit card data, and it can be hard to prevent and relatively easy to exploit these attacks.

In general terms, an SQLi attack is designed to exploit the nature of Web application pages. In contrast to the static webpages of years gone by, most current websites have dynamic components and content. Many such pages ask for information, such as location, personal identity information, and credit card information. This dynamic content is usually transferred to and from back-end databases that contain volumes of information—anything from cardholder data to which type of running shoes is most purchased. An application server webpage will make SQL queries to databases to send and receive information critical to making a positive user experience.

In such an environment, an SQLi attack is designed to send malicious SQL commands to the database server. The most common attack goal is bulk extraction of data. Attackers can dump database tables with hundreds of thousands of customer records. Depending on the environment, SQL injection can also be exploited to modify or delete data, execute arbitrary operating system commands, or launch denial-of-service (DoS) attacks. SQL injection is one of several forms of injection attacks that we discuss more generally in **Chapter 11.2**.

A Typical SQLi Attack

SQLi is an attack that exploits a security vulnerability occurring in the database layer of an application (such as queries). Using SQL injection, the attacker can extract or manipulate the Web application's data. The attack is viable when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed, and thereby unexpectedly executed.

Figure 5.5, from [ACUN13], is a typical example of an SQLi attack. The steps involved are as follows:

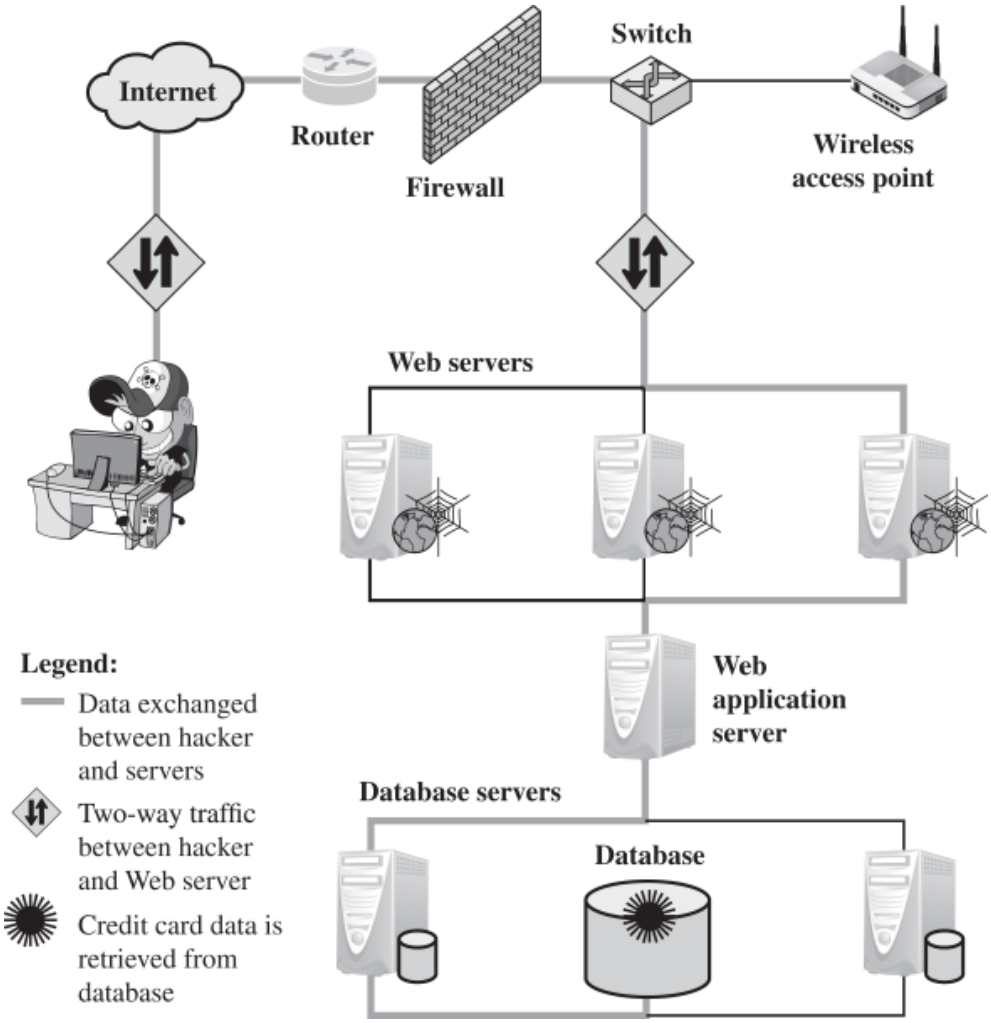


Figure 5.5 Typical SQL Injection Attack

1. Hacker finds a vulnerability in a custom Web application and injects an SQL command to a database by sending the command to the Web server. The command is injected into traffic that will be accepted by the firewall.
2. The Web server receives the malicious code and sends it to the Web application server.
3. The Web application server receives the malicious code from the Web server and sends it to the database server.
4. The database server executes the malicious code on the database. The database returns data from credit cards table.
5. The Web application server dynamically generates a page with data including credit card

details from the database.
6. The Web server sends the credit card details to the hacker.

The Injection Technique

The SQLi attack typically works by prematurely terminating a text string and appending a new command. Because the inserted command may have additional strings appended to it before it is executed, the attacker terminates the injected string with a comment mark “--”. Subsequent text is ignored at execution time.

As a simple example, consider a script that build an SQL query by combining predefined strings with text entered by a user:

```
var Shipcity;
ShipCity = Request.form (“ShipCity”);
var sql = “select * from OrdersTable where ShipCity = ‘” +
ShipCity + “‘ ”;
```

The intention of the script’s designer is that a user will enter the name of a city. For example, when the script is executed, the user is prompted to enter a city, and if the user enters Redmond, then the following SQL query is generated:

```
SELECT * FROM OrdersTable WHERE ShipCity = ‘Redmond’
```

Suppose, however, the user enters the following:

```
Boston’; DROP table OrdersTable--
```

This results in the following SQL query:

```
SELECT * FROM OrdersTable WHERE ShipCity =
‘Redmond’; DROP table OrdersTable--
```

The semicolon is an indicator that separates two commands, and the double dash is an indicator that the remaining text of the current line is a comment and not to be executed. When the SQL server processes this statement, it will first select all records in `OrdersTable where ShipCity is`

`Redmond`. Then, it executes the DROP request, which deletes the table.

SQLi Attack Avenues and Types

We can characterize SQLi attacks in terms of the avenue of attack and the type of attack [CHAN11, HALF06]. The main avenues of attack are as follows:

- **User input:** In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLi attacks that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.
- **Server variables:** Server variables are a collection of variables that contain HTTP headers, network protocol headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create an SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing data directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.
- **Second-order injection:** Second-order injection occurs when incomplete prevention mechanisms against SQL injection attacks are in place. In second-order injection, a malicious user could rely on data already present in the system or database to trigger an SQL injection attack, so when the attack occurs, the input that modifies the query to cause an attack does not come from the user, but from within the system itself.
- **Cookies:** When a client returns to a Web application, cookies can be used to restore the client's state information. Because the client has control over cookies, an attacker could alter cookies such that when the application server builds an SQL query based on the cookie's content, the structure and function of the query is modified.
- **Physical user input:** SQL injection is possible by supplying user input that constructs an attack outside the realm of Web requests. This user-input could take the form of conventional barcodes, RFID tags, or even paper forms which are scanned using optical character recognition and passed to a database management system.

Attack types can be grouped into three main categories: inband, inferential, and out-of-band. An **inband attack** uses the same communication channel for injecting SQL code and retrieving results. The retrieved data are presented directly in the application webpage. Inband attack types include the following:

- **Tautology:** This form of attack injects code in one or more conditional statements so they always evaluate to true. For example, consider this script, whose intent is to require the user

to enter a valid name and password:

```
$query = "SELECT info FROM user WHERE name =
'$_GET["name"]' AND pwd = '$_GET["pwd"]'";
```

Suppose the attacker submits `" ' OR 1=1 --"` for the name field. The resulting query would look like this:

```
SELECT info FROM users WHERE name = ' ' OR 1=1 -- AND pwpd = ' '
```

The injected code effectively disables the password check (because of the comment indicator --) and turns the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

- **End-of-line comment:** After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments. An example would be to add "- -" after inputs so that remaining queries are not treated as executable code, but comments. The preceding tautology example is also of this form.
- **Piggybacked queries:** The attacker adds additional queries beyond the intended query, piggy-backing the attack on top of a legitimate request. This technique relies on server configurations that allow several different queries within a single string of code. The example in the preceding section is of this form.

With an **inferential attack**, there is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the website/database server. Inferential attack types include the following:

- **Illegal/logically incorrect queries:** This attack lets an attacker gather important information about the type and structure of the backend database of a Web application. The attack is considered a preliminary, information-gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker.
- **Blind SQL injection:** Blind SQL injection allows attackers to infer the data present in a database system even when the system is sufficiently secure to not display any erroneous information back to the attacker. The attacker asks the server true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally functioning page.

In an **out-of-band attack**, data are retrieved using a different channel (e.g., an e-mail with the

results of the query is generated and sent to the tester). This can be used when there are limitations on information retrieval, but outbound connectivity from the database server is lax.

SQLi Countermeasures

Because SQLi attacks are so prevalent, damaging, and varied both by attack avenue and type, a single countermeasure is insufficient. Rather an integrated set of techniques is necessary. In this section, we provide a brief overview of the types of countermeasures that are in use or being researched, using the classification in [SHAR13]. These countermeasures can be classified into three types: defensive coding, detection, and run-time prevention.

Many SQLi attacks succeed because developers have used insecure coding practices, as we discuss in **Chapter 11**. Thus, defensive coding is an effective way to dramatically reduce the threat from SQLi. Examples of **defensive coding** include the following:

- **Manual defensive coding practices:** A common vulnerability exploited by SQLi attacks is insufficient input validation. The straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practices. An example is input type checking, to check that inputs that are supposed to be numeric contain no characters other than digits. This type of technique can avoid attacks based on forcing errors in the database management system. Another type of coding practice is one that performs pattern matching to try to distinguish normal input from abnormal input.
- **Parameterized query insertion:** This approach attempts to prevent SQLi by allowing the application developer to more accurately specify the structure of an SQL query, and pass the value parameters to it separately such that any unsanitary user input is not allowed to modify the query structure.
- **SQL DOM:** SQL DOM is a set of classes that enables automated data type validation and escaping [MCCL05]. This approach uses encapsulation of database queries to provide a safe and reliable way to access databases. This changes the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within the API, developers are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input.

A variety of **detection** methods have been developed, including the following:

- **Signature-based:** This technique attempts to match specific attack patterns. Such an approach must be constantly updated and may not work against self-modifying attacks.
- **Anomaly-based:** This approach attempts to define normal behavior then detect behavior patterns outside the normal range. A number of approaches have been used. In general terms, there is a training phase, in which the system learns the range of normal behavior, followed by the actual detection phase.
- **Code analysis:** Code analysis techniques involve the use of a test suite to detect SQLi

vulnerabilities. The test suite is designed to generate a wide range of SQLi attacks and assess the response of the system.

Finally, a number of **run-time prevention** techniques have been developed as SQLi countermeasures. These techniques check queries at runtime to see if they conform to a model of expected queries. Various automated tools are available for this purpose [CHAN11, SHAR13].

5.5 DATABASE ACCESS CONTROL

Commercial and open-source DBMSs typically provide an access control capability for the database. The DBMS operates on the assumption that the computer system has authenticated each user. As an additional line of defense, the computer system may use the overall access control system described in **Chapter 4** to determine whether a user may have access to the database as a whole. For users who are authenticated and granted access to the database, a database access control system provides a specific capability that controls access to portions of the database.

Commercial and open-source DBMSs provide discretionary or role-based access control. We defer a discussion of mandatory access control considerations to **Chapter 27**. Typically, a DBMS can support a range of administrative policies, including the following:

- **Centralized administration:** A small number of privileged users may grant and revoke access rights.
- **Ownership-based administration:** The owner (creator) of a table may grant and revoke access rights to the table.
- **Decentralized administration:** In addition to granting and revoking access rights to a table, the owner of the table may grant and revoke authorization rights to other users, allowing them to grant and revoke access rights to the table.

As with any access control system, a database access control system distinguishes different access rights, including create, insert, delete, update, read, and write. Some DBMSs provide considerable control over the granularity of access rights. Access rights can be to the entire database, to individual tables, or to selected rows or columns within a table. Access rights can be determined based on the contents of a table entry. For example, in a personnel database, some users may be limited to seeing salary information only up to a certain maximum value. And a department manager may only be allowed to view salary information for employees in his or her department.

SQL-Based Access Definition

SQL provides two commands for managing access rights, GRANT and REVOKE. For different versions of SQL, the syntax is slightly different. In general terms, the GRANT command has the following syntax:[1]

1The following syntax definition conventions are used. Elements separated by a vertical line are alternatives. A

list of alternatives is grouped in curly brackets. Square brackets enclose optional elements. That is, the elements inside the square brackets may or may not be present.

GRANT	{ privileges | role }
[ON	table]
TO	{ user | role | PUBLIC }
[IDENTIFIED BY	password]
[WITH	GRANT OPTION]

This command can be used to grant one or more access rights or can be used to assign a user to a role. For access rights, the command can optionally specify that it applies only to a specified table. The TO clause specifies the user or role to which the rights are granted. A PUBLIC value indicates that any user has the specified access rights. The optional IDENTIFIED BY clause specifies a password that must be used to revoke the access rights of this GRANT command. The GRANT OPTION indicates that the grantee can grant this access right to other users, with or without the grant option.

As a simple example, consider the following statement:

GRANT SELECT ON ANY TABLE TO ricflair

This statement enables the user ricflair to query any table in the database.

Different implementations of SQL provide different ranges of access rights. The following is a typical list:

- Select: Grantee may read entire database; individual tables; or specific columns in a table.
- Insert: Grantee may insert rows in a table; or insert rows with values for specific columns in a table.
- Update: Semantics is similar to INSERT.
- Delete: Grantee may delete rows from a table.
- References: Grantee is allowed to define foreign keys in another table that refer to the specified columns.

The REVOKE command has the following syntax:

REVOKE	{ privileges | role }
[ON	table]
FROM	{ user | role | PUBLIC }

Thus, the following statement revokes the access rights of the preceding example:

REVOKE SELECT ON ANY TABLE FROM ricflair

Cascading Authorizations

The grant option enables an access right to cascade through a number of users. We consider a specific access right and illustrate the cascade phenomenon in **Figure 5.6**. The figure indicates that Ann grants the access right to Bob at time t=10 and to Chris at time t=20. Assume the grant option is always used. Thus, Bob is able to grant the access right to David at t=30. Chris redundantly grants the access right to David at t=50. Meanwhile, David grants the right to Ellen, who in turn grants it to Jim; and subsequently David grants the right to Frank.

Just as the granting of privileges cascades from one user to another using the grant option, the revocation of privileges also cascaded. Thus, if Ann revokes the access right to Bob and Chris, then the access right is also revoked to David, Ellen, Jim, and Frank. A complication arises when a user receives the same access right multiple times, as happens in the case of David. Suppose Bob revokes the privilege from David. David still has the access right because it was granted by Chris at t=50. However, David granted the access right to Ellen after receiving the right, with grant option, from Bob but prior to receiving it from Chris. Most implementations dictate that in this circumstance, the access right to Ellen and therefore Jim is revoked when Bob revokes the access right to David. This is because at t=40, when David granted the access right to Ellen, David only had the grant option to do this from Bob. When Bob revokes the right, this causes all subsequent cascaded grants that are traceable solely to Bob via David to be revoked. Because David granted the access right to Frank after David was granted the access right with grant option from Chris, the access right to Frank remains. These effects are shown in the lower portion of **Figure 5.6**.

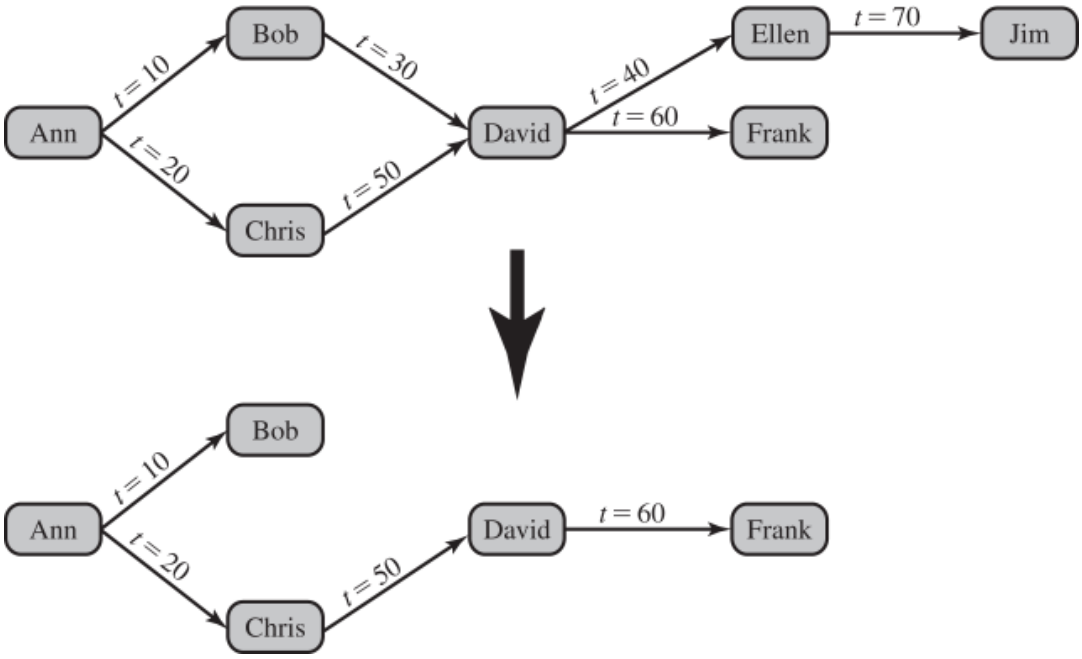


Figure 5.6 Bob Revokes Privilege from David

To generalize, the convention followed by most implementations is as follows. When user A revokes an access right, any cascaded access right is also revoked, unless that access right would exist even if the original grant from A had never occurred. This convention was first proposed in [GRIF76].

Role-Based Access Control

A role-based access control (RBAC) scheme is a natural fit for database access control. Unlike a file system associated with a single or a few applications, a database system often supports dozens of applications. In such an environment, an individual user may use a variety of applications to perform a variety of tasks, each of which requires its own set of privileges. It would be poor administrative practice to simply grant users all of the access rights they require for all the tasks they perform. RBAC provides a means of easing the administrative burden and improving security.

In a discretionary access control environment, we can classify database users in to three broad categories:

- **Application owner:** An end user who owns database objects (tables, columns, and rows) as part of an application. That is, the database objects are generated by the application or are prepared for use by the application.
- **End user other than application owner:** An end user who operates on database objects via a particular application but does not own any of the database objects.
- **Administrator:** User who has administrative responsibility for part or all of the database.

We can make some general statements about RBAC concerning these three types of users. An application has associated with it a number of tasks, with each task requiring specific access rights to portions of the database. For each task, one or more roles can be defined that specify the needed access rights. The application owner may assign roles to end users. Administrators are responsible for more sensitive or general roles, including those having to do with managing physical and logical database components, such as data files, users, and security mechanisms. The system needs to be set up to give certain administrators certain privileges. Administrators in turn can assign users to administrative-related roles.

A database RBAC facility needs to provide the following capabilities:

- Create and delete roles.
- Define permissions for a role.
- Assign and cancel assignment of users to roles.

A good example of the use of roles in database security is the RBAC facility provided by Microsoft SQL Server. SQL Server supports three types of roles: Server roles, database roles, and user-defined roles. The first two types of roles are referred to as fixed roles (see **Table 5.2**); these are preconfigured for a system with specific access rights. The administrator or user cannot add, delete, or modify fixed roles; it is only possible to add and remove users as members of a fixed role.

Table 5.2 Fixed Roles in Microsoft SQL Server

Role	Permissions
Fixed Server Roles	
sysadmin	Can perform any activity in SQL Server and have complete control over all database functions
serveradmin	Can set server-wide configuration options and shut down the server
setupadmin	Can manage linked servers and startup procedures
securityadmin	Can manage logins and CREATE DATABASE permissions, also read error logs and change passwords

processadmin	Can manage processes running in SQL Server
Dbcreator	Can create, alter, and drop databases
diskadmin	Can manage disk files
bulkadmin	Can execute BULK INSERT statements
Fixed Database Roles	
db_owner	Has all permissions in the database
db_accessadmin	Can add or remove user IDs
db_datareader	Can select all data from any user table in the database
db_datawriter	Can modify any data in any user table in the database
db_ddladmin	Can issue all data definition language statements
db_securityadmin	Can manage all permissions, object ownerships, roles and role memberships
db_backupoperator	Can issue DBCC, CHECKPOINT, and BACKUP statements
db_denydatareader	Can deny permission to select data in the database
db_denydatawriter	Can deny permission to change data in the database

Fixed server roles are defined at the server level and exist independently of any user database. They are designed to ease the administrative task. These roles have different permissions and are intended to provide the ability to spread the administrative responsibilities without having to give out complete control. Database administrators can use these fixed server roles to assign

different administrative tasks to personnel and give them only the rights they absolutely need.

Fixed database roles operate at the level of an individual database. As with fixed server roles, some of the fixed database roles, such as db_accessadmin and db_securityadmin, are designed to assist a DBA with delegating administrative responsibilities. Others, such as db_datareader and db_datawriter, are designed to provide blanket permissions for an end user.

SQL Server allows users to create roles. These **user-defined roles** can then be assigned access rights to portions of the database. A user with proper authorization (typically, a user assigned to the db_securityadmin role) may define a new role and associate access rights with the role. There are two types of user-defined roles: Standard and application. For a standard role, an authorized user can assign other users to the role. An application role is associated with an application rather than with a group of users and requires a password. The role is activated when an application executes the appropriate code. A user who has access to the application can use the application role for database access. Often, database applications enforce their own security based on the application logic. For example, you can use an application role with its own password to allow the particular user to obtain and modify any data only during specific hours. Thus, you can realize more complex security management within the application logic.

5.6 INFERENCE

Inference, as it relates to database security, is the process of performing authorized queries and deducing unauthorized information from the legitimate responses received. The inference problem arises when the combination of a number of data items is more sensitive than the individual items, or when a combination of data items can be used to infer data of higher sensitivity. **Figure 5.7** illustrates the process. The attacker may make use of nonsensitive data as well as metadata. Metadata refers to knowledge about correlations or dependencies among data items that can be used to deduce information not otherwise available to a particular user. The information transfer path by which unauthorized data is obtained is referred to as an **inference channel**.

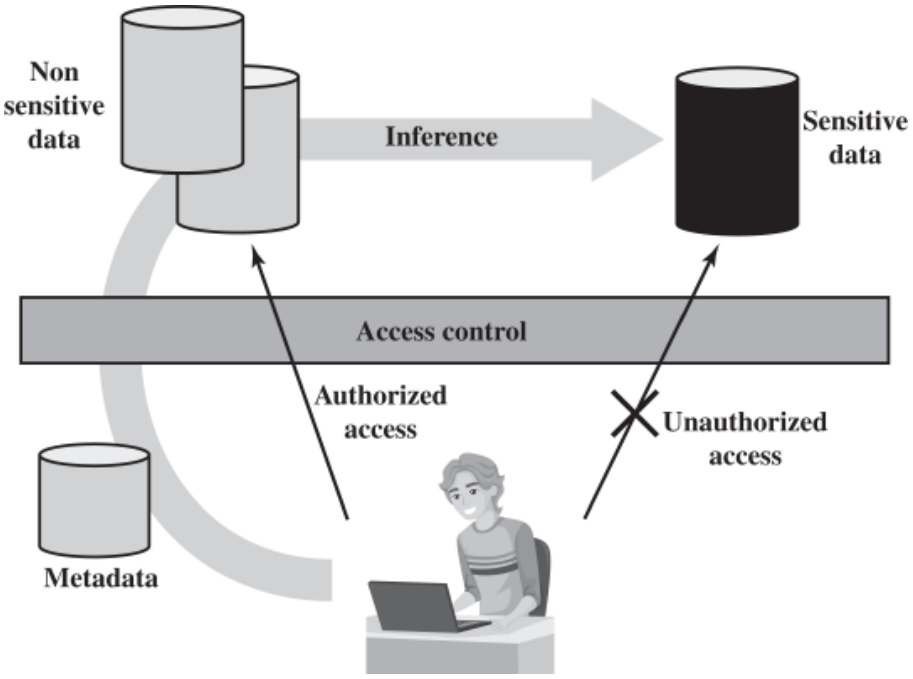


Figure 5.7 Indirect Information Access via Inference Channel

In general terms, two inference techniques can be used to derive additional information: Analyzing functional dependencies between attributes within a table or across tables, and merging views with the same constraints.

An example of the latter, shown in **Figure 5.8**, illustrates the inference problem. **Figure 5.8a** shows an Inventory table with four columns. **Figure 5.8b** shows two views, defined in SQL as follows:

Item	Availability	Cost ($)	Department
Shelf support	in-store/online	7.99	hardware
Lid support	online only	5.49	hardware
Decorative chain	in-store/online	104.99	hardware
Cake pan	online only	12.99	housewares
Shower/tub cleaner	in-store/online	11.99	housewares
Rolling pin	in-store/online	10.99	housewares

(a) Inventory table

Availability	Cost ($)
in-store/online	7.99
online only	5.49
in-store/online	104.99

Item	Department
Shelf support	hardware
Lid support	hardware
Decorative chain	hardware

(b) Two views

Item	Availability	Cost ($)	Department
Shelf support	in-store/online	7.99	hardware
Lid support	online only	5.49	hardware
Decorative chain	in-store/online	104.99	hardware

(c) Table derived from combining query answers

Figure 5.8 Inference Example

`CREATE view V1 AS`	`CREATE view V2 AS`
`SELECT Availability, Cost`	`SELECT Item, Department`
`FROM Inventory`	`FROM Inventory`
`WHERE Department = "hardware"`	`WHERE Department = "hardware"`

Users of these views are not authorized to access the relationship between Item and Cost. A user who has access to either or both views cannot infer the relationship by functional dependencies. That is, there is not a functional relationship between Item and Cost such that knowing Item and perhaps other information is sufficient to deduce Cost. However, suppose the two views are created with the access constraint that Item and Cost cannot be accessed together. A user who knows the structure of the Inventory table and who knows that the view tables maintain the same row order as the Inventory table is then able to merge the two views to construct the table shown in **Figure 5.8c**. This violates the access control policy that the relationship of attributes Item and

Cost must not be disclosed.

In general terms, there are two approaches to dealing with the threat of disclosure by inference:

- **Inference detection during database design:** This approach removes an inference channel by altering the database structure or by changing the access control regime to prevent inference. Examples include removing data dependencies by splitting a table into multiple tables or using more fine-grained access control roles in an RBAC scheme. Techniques in this category often result in unnecessarily stricter access controls that reduce availability.
- **Inference detection at query time:** This approach seeks to eliminate an inference channel violation during a query or series of queries. If an inference channel is detected, the query is denied or altered.

For either of the preceding approaches, some inference detection algorithm is needed. This is a difficult problem and the subject of ongoing research. To give some appreciation of the difficulty, we present an example taken from [LUNT89]. Consider a database containing personnel information, including names, addresses, and salaries of employees. Individually, the name, address, and salary information is available to a subordinate role, such as Clerk, but the association of names and salaries is restricted to a superior role, such as Administrator. This is similar to the problem illustrated in **Figure 5.8**. One solution to this problem is to construct three tables, which include the following information:

Employees (Emp#, Name, Address)

Salaries (S#, Salary)

Emp-Salary (Emp#, S#)

where each line consists of the table name followed by a list of column names for that table. In this case, each employee is assigned a unique employee number (Emp#) and a unique salary number (S#). The Employees table and the Salaries table are accessible to the Clerk role, but the Emp-Salary table is only available to the Administrator role. In this structure, the sensitive relationship between employees and salaries is protected from users assigned the Clerk role. Now, suppose we want to add a new attribute, employee start date, which is not sensitive. This could be added to the Salaries table as follows:

Employees (Emp#, Name, Address)

Salaries (S#, Salary, Start-Date)

Emp-Salary (Emp#, S#)

However, an employee's start date is an easily observable or discoverable attribute of an employee. Thus, a user in the Clerk role should be able to infer (or partially infer) the employee's name. This would compromise the relationship between employee and salary. A straightforward way to remove the inference channel is to add the start-date column to the Employees table

rather than to the Salaries table.

The first security problem indicated in this sample, that it was possible to infer the relationship between employee and salary, can be detected through analysis of the data structures and security constraints that are available to the DBMS. However, the second security problem, in which the start-date column was added to the Salaries table, cannot be detected using only the information stored in the database. In particular, the database does not indicate that the employee name can be inferred from the start date.

In the general case of a relational database, inference detection is a complex and difficult problem. For multilevel secure databases, to be discussed in **Chapter 27**, and statistical databases, to be discussed in the next section, progress has been made in devising specific inference detection techniques.

5.7 DATABASE ENCRYPTION

The database is typically the most valuable information resource for any organization and is therefore protected by multiple layers of security, including firewalls, authentication mechanisms, general access control systems, and database access control systems. In addition, for particularly sensitive data, database encryption is warranted and often implemented. Encryption becomes the last line of defense in database security.

There are two disadvantages to database encryption:

- **Key management:** Authorized users must have access to the decryption key for the data for which they have access. Because a database is typically accessible to a wide range of users and a number of applications, providing secure keys to selected parts of the database to authorized users and applications is a complex task.
- **Inflexibility:** When part or all of the database is encrypted, it becomes more difficult to perform record searching.

Encryption can be applied to the entire database, at the record level (encrypt selected records), at the attribute level (encrypt selected columns), or at the level of the individual field.

A number of approaches have been taken to database encryption. In this section, we look at a representative approach for a multiuser database.

A DBMS is a complex collection of hardware and software. It requires a large storage capacity and requires skilled personnel to perform maintenance, disaster protection, update, and security. For many small and medium-sized organizations, an attractive solution is to outsource the DBMS and the database to a service provider. The service provider maintains the database off-site and can provide high availability, disaster prevention, and efficient access and update. The main concern with such a solution is the confidentiality of the data.

A straightforward solution to the security problem in this context is to encrypt the entire database and not provide the encryption/decryption keys to the service provider. This solution by itself is inflexible. The user has little ability to access individual data items based on searches or indexing on key parameters, but rather would have to download entire tables from the database, decrypt the tables, and work with the results. To provide more flexibility, it must be possible to work with the database in its encrypted form.

An example of such an approach, depicted in **Figure 5.9**, is reported in [DAMI05] and [DAMI03]. A similar approach is described in [HACI02]. Four entities are involved:

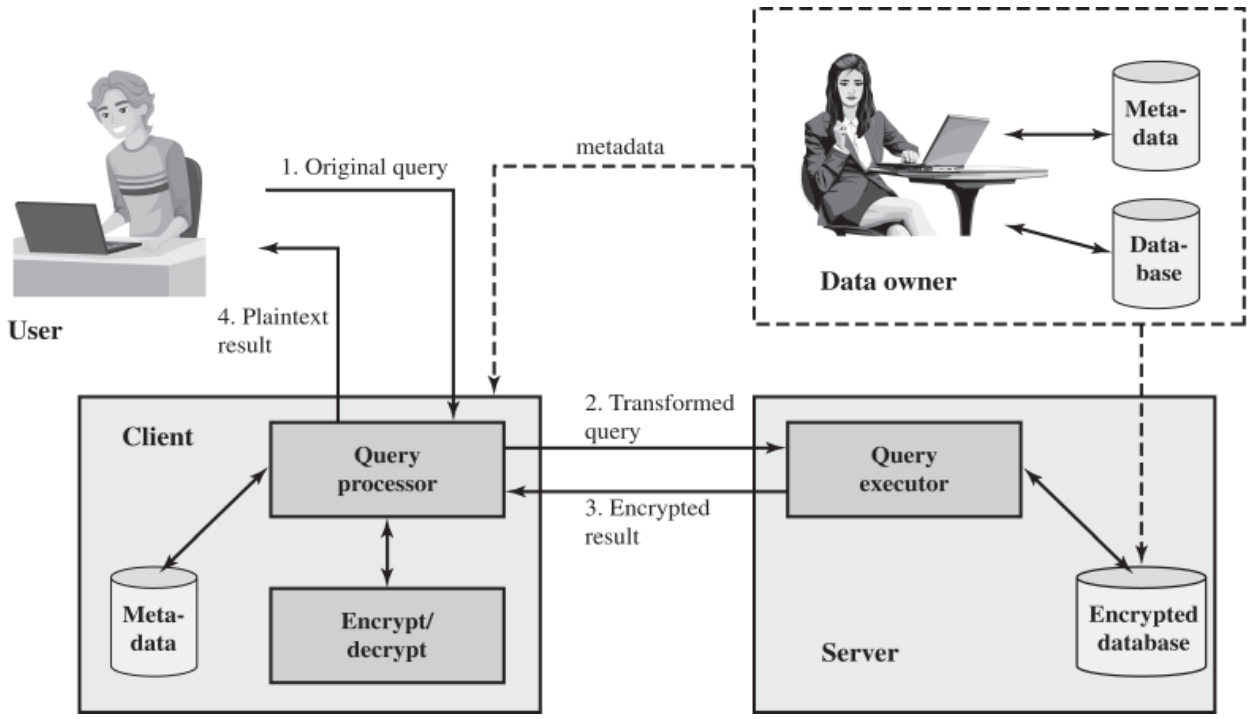


Figure 5.9 A Database Encryption Scheme

- **Data owner:** An organization that produces data to be made available for controlled release, either within the organization or to external users.
- **User:** Human entity that presents requests (queries) to the system. The user could be an employee of the organization who is granted access to the database via the server, or a user external to the organization who, after authentication, is granted access.
- **Client:** Front end that transforms user queries into queries on the encrypted data stored on the server.
- **Server:** An organization that receives the encrypted data from a data owner and makes them available for distribution to clients. The server could in fact be owned by the data owner but, more typically, is a facility owned and maintained by an external provider.

Let us first examine the simplest possible arrangement based on this scenario. Suppose each individual item in the database is encrypted separately, all using the same encryption key. The encrypted database is stored at the server, but the server does not have the key, so the data are secure at the server. Even if someone were able to hack into the server's system, all he or she would have access to is encrypted data. The client system does have a copy of the encryption key. A user at the client can retrieve a record from the database with the following sequence:

1. The user issues an SQL query for fields from one or more records with a specific value of the primary key.
2. The query processor at the client encrypts the primary key, modifies the SQL query accordingly, and transmits the query to the server.
3. The server processes the query using the encrypted value of the primary key and returns the appropriate record or records.
4. The query processor decrypts the data and returns the results.

For example, consider this query, which was introduced in **Section 5.1**, on the database of **Figure 5.4a**:

```
SELECT Ename, Eid, Ephone
   FROM Employee
   WHERE Did = 15
```

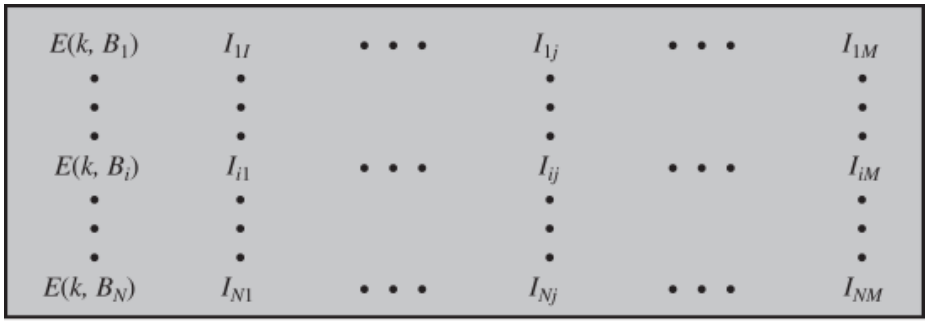
Assume the encryption key *k* is used and the encrypted value of the department id 15 is E(k, 15)=1000110111001110. Then, the query processor at the client could transform the preceding query into

```
SELECT Ename, Eid, Ephone
   FROM Employee
   WHERE Did = 1000110111001110
```

This method is certainly straightforward but, as was mentioned, lacks flexibility. For example, suppose the Employee table contains a salary attribute and the user wishes to retrieve all records for salaries less than $70K. There is no obvious way to do this, because the attribute value for salary in each record is encrypted. The set of encrypted values do not preserve the ordering of values in the original attribute.

To provide more flexibility, the following approach is taken. Each record (row) of a table in the database is encrypted as a block. Referring to the abstract model of a relational database in **Figure 5.3**, each row Ri is treated as a contiguous block Bi=(xi xi2 ... xiM) Thus, each attribute value in Ri, regardless of whether it is text or numeric, is treated as a sequence of bits, and all of the attribute values for that row are concatenated together to form a single binary block. The entire row is encrypted, expressed as E(k,B)=E(k,(xi xi2 ... xiM)) To assist in data retrieval, attribute indexes are associated with each table. For some or all of the attributes an index value is created. For each row Ri of the unencrypted database, the mapping is as follows (see **Figure 5.10**):

(xi1, xi2, …, xiM)→[E(k, Bi), Ii1, Ii2, …, IiM]



$B_i = (x_{i1} \| x_{i2} \| \ldots \| x_{iM})$

Figure 5.10 Encryption Scheme for Database of Figure 5.3

For each row in the original database, there is one row in the encrypted database. The index values are provided to assist in data retrieval. We can proceed as follows. For any attribute, the range of attribute values is divided into a set of non-overlapping partitions that encompass all possible values, and an index value is assigned to each partition.

Table 5.3 provides an example of this mapping. Suppose employee ID (*eid*) values lie in the range [1, 1000]. We can divide these values into five partitions: [1, 200], [201, 400], [401, 600], [601, 800], and [801, 1000]; then assign index values 1, 2, 3, 4, and 5, respectively. For a text field, we can derive an index from the first letter of the attribute value. For the attribute *ename*, let us assign index 1 to values starting with A or B, index 2 to values starting with C or D, and so on. Similar partitioning schemes can be used for each of the attributes. Table 5.3b shows the resulting table. The values in the first column represent the encrypted values for each row. The actual values depend on the encryption algorithm and the encryption key. The remaining columns show index values for the corresponding attribute values. The mapping functions between attribute values and index values constitute metadata that are stored at the client and data owner locations but not at the server.

Table 5.3 Encrypted Database Example

(a) Employee Table

eid	ename	salary	addr	did
23	Tom	70K	Maple	45
860	Mary	60K	Main	83
320	John	50K	River	50

875	Jerry	55K	Hopewell	92

(b) Encrypted Employee Table with Indexes

E(*k*, *B*)	*I*(*eid*)	*I*(*ename*)	*I*(*salary*)	*I*(*addr*)	*I*(*did*)
1100110011001011 . . .	1	10	3	7	4
0111000111001010 . . .	5	7	2	7	8
1100010010001101 . . .	2	5	1	9	5
0011010011111101 . . .	5	5	2	4	9

This arrangement provides for more efficient data retrieval. Suppose, for example, a user requests records for all employees with eid<300. The query processor requests all records with I(eid)=2. These are returned by the server. The query processor decrypts all rows returned, discards those that do not match the original query, and returns the requested unencrypted data to the user.

The indexing scheme just described does provide a certain amount of information to an attacker, namely a rough relative ordering of rows by a given attribute. To obscure such information, the ordering of indexes can be randomized. For example, the *eid* values could be partitioned by mapping [1, 200], [201, 400], [401, 600], [601, 800], and [801, 1000] into 2, 3, 5, 1, and 4, respectively. Because the metadata are not stored at the server, an attacker could not gain this information from the server.

Other features may be added to this scheme. To increase the efficiency of accessing records by means of the primary key, the system could use the encrypted value of the primary key attribute values, or a hash value. In either case, the row corresponding to the primary key value could be retrieved individually. Different portions of the database could be encrypted with different keys, so users would only have access to that portion of the database for which they had the decryption key. This latter scheme could be incorporated into a role-based access control system.

5.8 DATA CENTER SECURITY

A data center is an enterprise facility that houses a large number of servers, storage devices, and network switches and equipment. The number of servers and storage devices can run into the tens of thousands in a single facility. Examples of uses for these large data centers include cloud service providers, search engines, large scientific research facilities, and IT facilities for large enterprises. A data center generally includes redundant or backup power supplies, redundant network connections, environmental controls (e.g., air conditioning and fire suppression), and various security devices. Large data centers are industrial scale operations using as much electricity as a small town. A data center can occupy one room of a building, one or more floors, or an entire building.

Data Center Elements

Figure 5.11 illustrates key elements of a large data center configuration. Most of the equipment in a large data center is in the form of stacks of servers and storage modules mounted in open racks or closed cabinets, which are usually placed in single rows forming corridors between them. This allows access to the front and rear of each rack or cabinet. Typically, the individual modules are equipped with 10-Gbps or 40-Gbps Ethernet ports to handle the massive traffic to and from these servers. Also typically, each rack has one or two 10, 40 or 100-Gbps Ethernet switches to interconnect all the servers and provide connectivity to the rest of the facility. The switches are often mounted in the rack and referred to as top-of-rack (ToR) switches. The term *ToR* has become synonymous with server access switch, even if it is not located “top of rack.” Very large data centers, such as cloud providers, require switches operating at 100 Gbps to support the interconnection of server racks and to provide adequate capacity for connecting off-site through network interface controllers (NICs) on routers or firewalls.

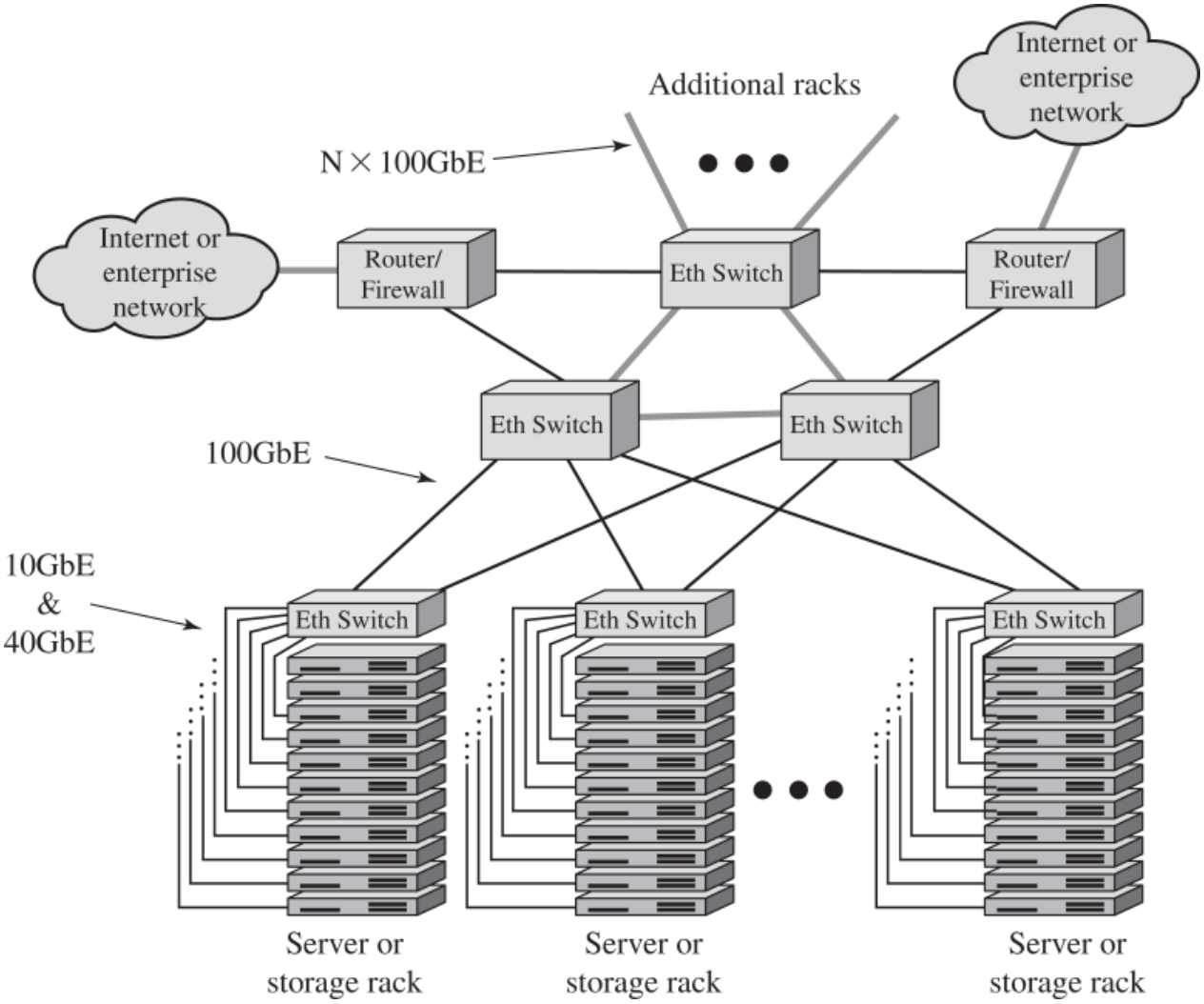


Figure 5.11 Key Data Center Elements

Key elements not shown in **Figure 5.11** are cabling and cross connects, which we can list as follows:

- **Cross connect:** A facility enabling the termination of cables, as well as their interconnection with other cabling or equipment.
- **Horizontal cabling:** Any cabling that is used to connect a floor's wiring closet to wall plates in the work areas to provide local area network (LAN) drops for connecting servers and other digital equipment to the network. The term *horizontal* is used because such cabling is typically run along the ceiling or floor.
- **Backbone cabling:** Run between data center rooms or enclosures and the main cross-connect point of a building.

Data Center Security Considerations

All of the security threats and countermeasures discussed in this text are relevant in the context

of large data centers, and indeed it is in this context that the risks are most acute. Consider that the data center houses massive amounts of data that are:

- located in a confined physical space.
- interconnected with direct-connect cabling.
- accessible through external network connections, so once past the boundary, a threat is posed to the entire complex.
- typically representative of the greatest single asset of the enterprise.

Thus, data center security is a top priority for any enterprise with a large data center. Some of the important threats to consider include the following:

- Denial of service
- Advanced persistent threats from targeted attacks
- Privacy breaches
- Application exploits such as SQL injection
- Malware
- Physical security threats

Figure 5.12 highlights important aspects of data center security, represented as a four-layer model. Site security refers primarily to the physical security of the entire site including the building that houses the data center, as well as the use of redundant utilities. Physical security of the data center itself includes barriers to entry, such as a mantrap (a double-door single-person access control space) coupled with authentication techniques for gaining physical access. Physical security can also include security personnel, surveillance systems, and other measures which will be discussed in **Chapter 16**. Network security is extremely important in a facility in which such a large collection of assets are concentrated in a single place and accessible by external network connections. Typically, a large data center will employ all of the network security techniques discussed in this text. Finally, security of the data itself, as opposed to the systems they reside on, involves techniques discussed in the remainder of this chapter.

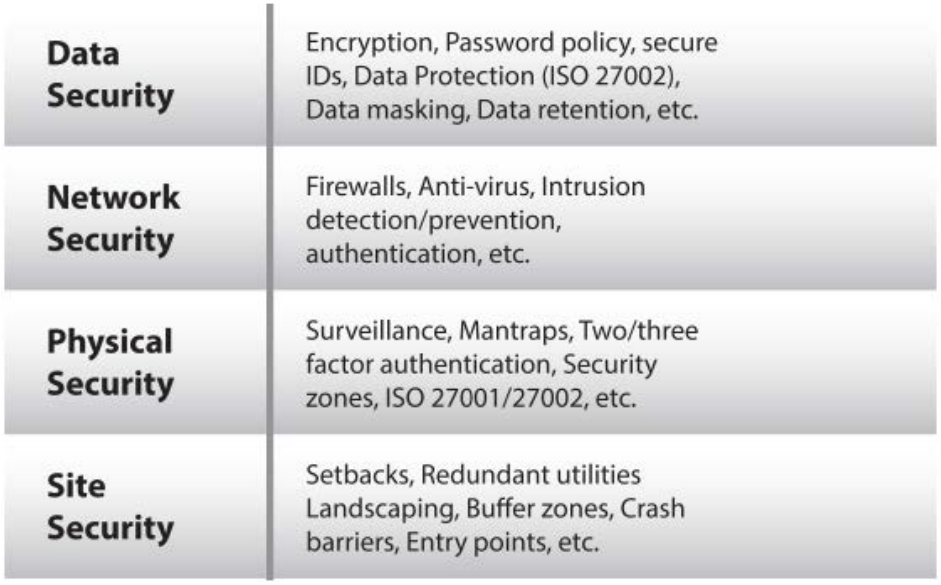


Figure 5.12 Data Center Security Model

TIA-492

The Telecommunications Industry Association (TIA) standard TIA-492 (*Telecommunications Infrastructure Standard for Data Centers*) specifies the minimum requirements for telecommunications infrastructure of data centers. Topics covered include the following:

- Network architecture
- Electrical design
- File storage, backup, and archiving
- System redundancy
- Network access control and security
- Database management
- Web hosting
- Application hosting
- Content distribution
- Environmental control
- Protection against physical hazards (fire, flood, and windstorm)
- Power management

The standard specifies function areas, which helps to define equipment placement based on the standard hierarchical design for regular commercial spaces. This architecture anticipates growth and helps create an environment where applications and servers can be added and upgraded with minimal downtime. This standardized approach supports high availability and a uniform environment for implementing security measures. TIA-942 specifies that a data center should include the following functional areas (see **Figure 5.13**):

- **Computer room:** Portion of the data center that houses date processing equipment.
- **Entrance room:** One or more entrance rooms house external network access provider equipment, plus provide the interface between the computer room equipment and the enterprise cabling systems. Physical separation of the entrance room from the computer room provides better security.
- **Main distribution area:** A centrally located area that houses the main cross-connect as well as core routers and switches for LAN and SAN (storage area network) infrastructures.
- **Horizontal distribution area (HDA):** Serves as the distribution point for horizontal cabling and houses cross-connects and active equipment for distributing cable to the equipment distribution area.
- **Equipment distribution area (EDA):** The location of equipment cabinets and racks, with horizontal cables terminating with patch panels.
- **Zone distribution area (ZDA):** An optional interconnection point in the horizontal cabling between the HDA and EDA. The ZDA can act as a consolidation point for reconfiguration flexibility or for housing freestanding equipment such as mainframes.

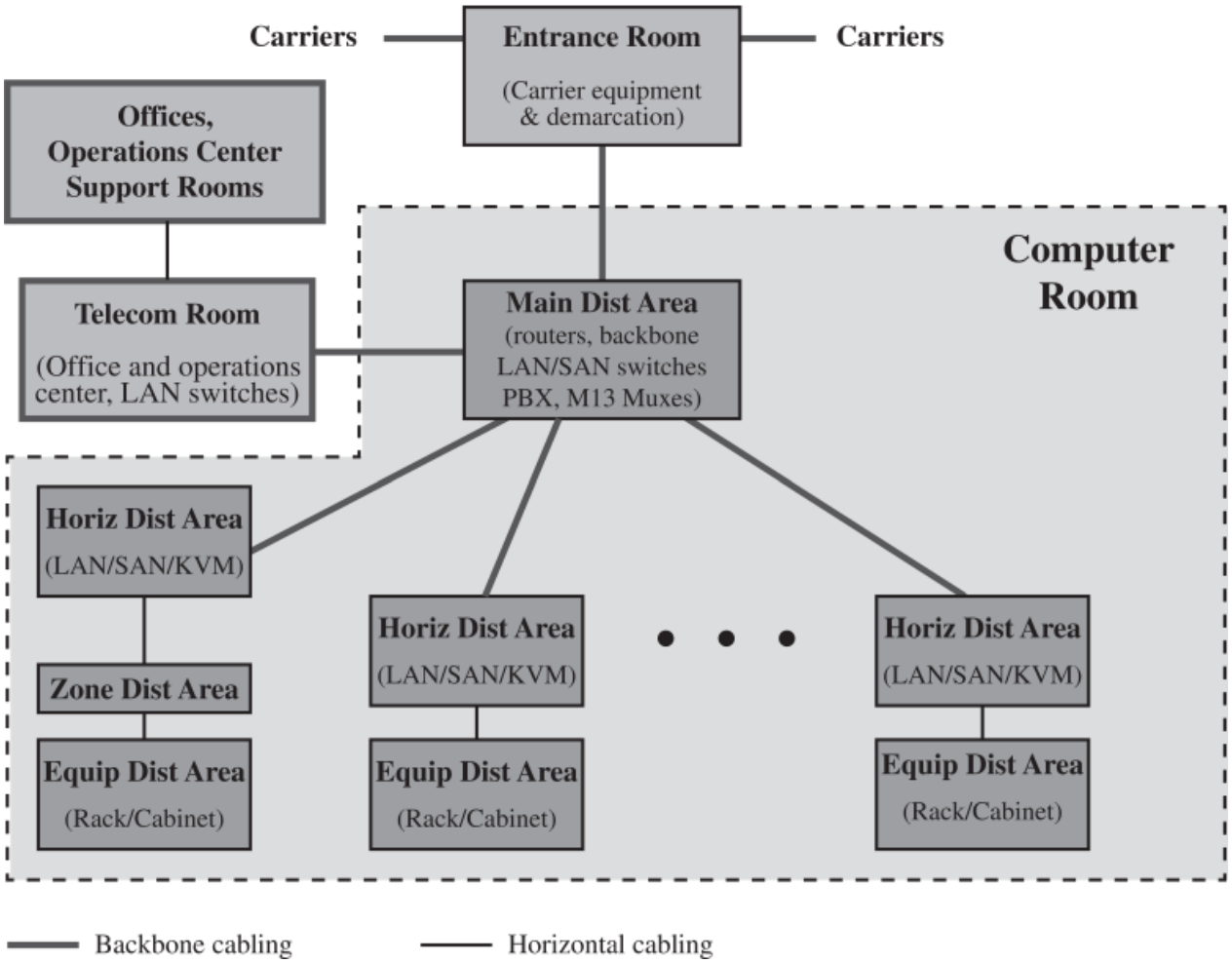


Figure 5.13 TIA-942 Compliant Data Center Showing Key Functional Areas

An important part of TIA-942, especially relevant for computer security, is the concept of tiered reliability. The standard defines four tiers, as shown in **Table 5.4**. For each of the four tiers, TIA-

942 describes detailed architectural, security, electrical, mechanical, and telecommunications recommendations such that the higher the tier is, the higher will be the availability.

Table 5.4 Data Center Tiers Defined in TIA-942

Tier	System Design	Availability/Annual Downtime
1	• Susceptible to disruptions from both planned and unplanned activity • Single path for power and cooling distribution, no redundant components • May or may not have raised floor, UPS, or generator • Takes 3 months to implement • Must be shut down completely to perform preventive maintenance	99.671%/28.8 hours
2	• Less susceptible to disruptions from both planned and unplanned activity • Single path for power and cooling distribution, includes redundant components • Includes raised floor, UPS, and generator • Takes 3 to 6 months to implement • Maintenance of power path and other parts of the infrastructure require a processing shutdown	99.741%/22.0 hours
3	• Enables planned activity without disrupting computer hardware operation but unplanned events will still cause disruption • Multiple power and cooling distribution paths but with only one path active, includes redundant components • Takes 15 to 20 months to implement • Includes raised floor and sufficient capacity and distribution to carry load on one path while performing maintenance on the other	99.982%/1.6 hours
4	• Planned activity does not disrupt critical load and data center can sustain at least one worst-case unplanned event with no critical load impact • Multiple active power and cooling distribution paths, includes redundant components • Takes 15 to 20 months to implement	99.995%/0.4 hours

5.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

attribute
blind SQL injection
cascading authorizations
compromise
data center
data swapping
database
database access control
database encryption
database management system (DBMS)
defensive coding
detection
end-of-line comment
foreign key
inband attack
inference
inference channel
inferential attack
out-of-band attack
parameterized query insertion
partitioning
piggybacked queries
primary key
query language
query set
relation
relational database
relational database management system (RDBMS)
run-time prevention
Structured Query Language (SQL)
SQL injection (SQLi) attack
tautology

tuple
view

Review Questions

5.1 Define the terms *database*, *database management system*, and *query language*.
5.2 What is a relational database and what are its principal ingredients?
5.3 How many primary keys and how many foreign keys may a table have in a relational database?
5.4 List and briefly describe some administrative policies that can be used with a RDBMS.
5.5 Explain the concept of cascading authorizations.
5.6 Explain the nature of the inference threat to an RDBMS.
5.7 What are the disadvantages of database encryption?
5.8 List and briefly define four data center availability tiers.

Problems

5.1 Consider a simplified university database that includes information on courses (name, number, day, time, room number, and max enrollment) and on faculty teaching courses and students attending courses. Suggest a relational database for efficiently managing this information.
5.2 The following table provides information on members of a mountain climbing club:

Climber-ID	Name	Skill Level	Age
123	Edmund	Experienced	80
214	Arnold	Beginner	25
313	Bridget	Experienced	33
212	James	Medium	27

The primary key is *Climber-ID*. Explain whether or not each of the following rows can be added to the table.

Climber-ID	Name	Skill Level	Age
214	Abbot	Medium	40
	John	Experienced	19

15	Jeff	Medium	42

5.3 The following table shows a list of pets and their owners that is used by a veterinarian service.

P_Name	Type	Breed	DOB	Owner	O_Phone	O_E-mail
Kino	Dog	Std. Poodle	3/27/97	M. Downs	5551236	md@abc.com
Teddy	Cat	Chartreaux	4/2/98	M. Downs	1232343	md@abc.com
Filo	Dog	Std. Poodle	2/24/02	R. James	2343454	rj@abc.com
AJ	Dog	Collie Mix	11/12/95	Liz Frier	3456567	liz@abc.com
Cedro	Cat	Unknown	12/10/96	R. James	7865432	rj@abc.com
Woolley	Cat	Unknown	10/2/00	M. Trent	9870678	mt@abc.com
Buster	Dog	Collie	4/4/01	Ronny	4565433	ron@abc.com

a. Describe four problems that are likely to occur when using this table.
b. Break the table into two tables in a way that fixes the four problems.

5.4 We wish to create a student table containing the student's ID number, name, and telephone number. Write an SQL statement to accomplish this.

5.5 Consider an SQL statement:

SELECT id, forename, surname FROM authors WHERE forename = 'john' AND surname = 'smith'

a. What is this statement intended to do?
b. Assume the forename and surname fields are being gathered from user-supplied input, and suppose the user responds with:
   Forename: jo'hn

   Surname: smith

   What will be the effect?

c. Now suppose the user responds with:
   Forename: jo'; drop table authors--

   Surname: smith

   What will be the effect?

5.6 **Figure 5.14** shows a fragment of code that implements the login functionality for a

database application. The code dynamically builds an SQL query and submits it to a database.

```
1. String login, password, pin, query
2. login = getParameter("login");
3. password = getParameter("pass");
3. pin = getParameter("pin");
4. Connection conn.createConnection("MyDataBase");
5. query = "SELECT accounts FROM users WHERE login='" +
6.       login + "'AND pass = '" + password +
7.       "'AND pin=" + pin;
8. ResultSet result = conn.executeQuery(query);
9. if (result!=NULL)
10       displayAccounts(result);
11 else
12       displayAuthFailed();
```

Figure 5.14 Code for Generating an SQL Query

a. Suppose a user submits login, password, and pin as doe, secret, and 123. Show the SQL query that is generated.
b. Instead, the user submits for the login field the following:
   ' or 1=1 - -

   What is the effect?

5.7 The SQL command word UNION is used to combine the result sets of 2 or more SQL SELECT statements. For the login code of **Figure 5.14** , suppose a user enters the following into the login field:
'UNION SELECT cardNo from CreditCards where acctNo = 10032 - -

What is the effect?

5.8 Assume A, B, and C grant certain privileges on the employee table to X, who in turn grants them to Y, as shown in the following table, with the numerical entries indicating the time of granting:

UserID	Table	Grantor	READ	INSERT	DELETE
X	Employee	A	15	15	—
X	Employee	B	20	—	20
Y	Employee	X	25	25	25

X	Employee	C	30	—	30

At time t=35, B issues the command REVOKE ALL RIGHTS ON Employee FROM X. Which access rights, if any, of Y must be revoked, using the conventions defined in Section 5.2 ?

5.9 Figure 5.15 shows a sequence of grant operations for a specific access right on a table. Assume at t=70, B revokes the access right from C. Using the conventions defined in Section 5.2 , show the resulting diagram of access right dependencies.

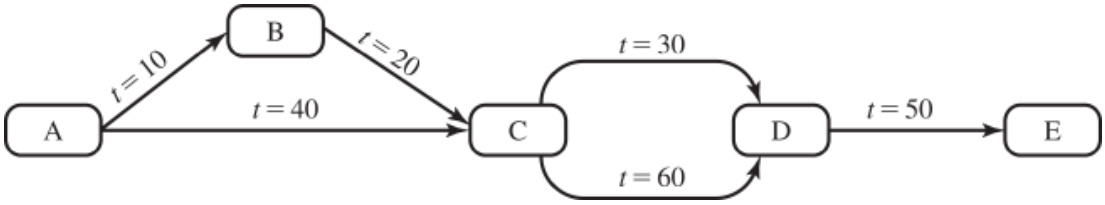


Figure 5.15 Cascaded Privileges

5.10 Figure 5.16 shows an alternative convention for handling revocations of the type illustrated in Figure 5.6 .

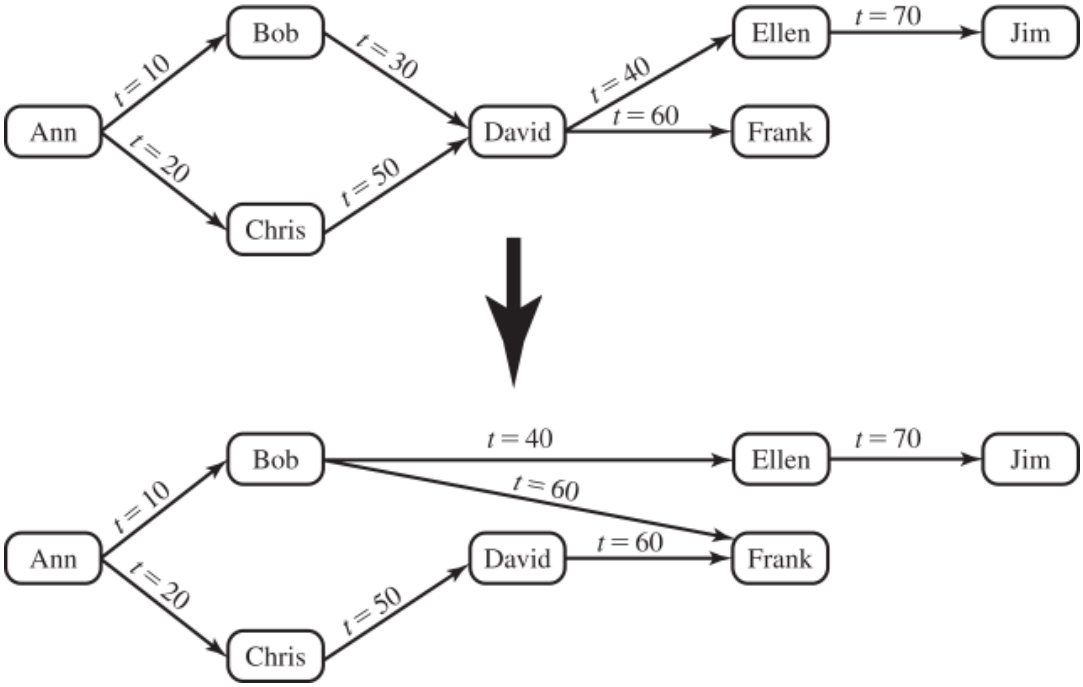


Figure 5.16 Bob Revokes Privilege from David, Second Version

a. Describe an algorithm for revocation that fits this figure.
b. Compare the relative advantages and disadvantages of this method to the original method, illustrated in Figure 5.6 .

5.11 Consider the parts department of a plumbing contractor. The department maintains an

inventory database that includes parts information (part number, description, color, size, number in stock, etc.) and information on vendors from whom parts are obtained (name, address, pending purchase orders, closed purchase orders, etc.). In an RBAC system, suppose roles are defined for accounts payable clerk, an installation foreman, and a receiving clerk. For each role, indicate which items should be accessible for read-only and read-write access.

5.12 Imagine you are the database administrator for a military transportation system. You have a table named cargo in your database that contains information on the various cargo holds available on each outbound airplane. Each row in the table represents a single shipment and lists the contents of that shipment and the flight identification number. Only one shipment per hold is allowed. The flight identification number may be cross-referenced with other tables to determine the origin, destination, flight time, and similar data. The cargo table appears as follows:

Flight ID	Cargo Hold	Contents	Classification
1254	A	Boots	Unclassified
1254	B	Guns	Unclassified
1254	C	Atomic bomb	Top Secret
1254	D	Butter	Unclassified

Suppose two roles are defined: Role 1 has full access rights to the cargo table. Role 2 has full access rights only to rows of the table in which the Classification field has the value Unclassified. Describe a scenario in which a user assigned to role 2 uses one or more queries to determine that there is a classified shipment on board the aircraft.

5.13 Users hulkhogan and undertaker do not have the SELECT access right to the Inventory table and the Item table. These tables were created by and are owned by user bruno-s. Write the SQL commands that would enable bruno-s to grant SELECT access to these tables to hulkhogan and undertaker.

5.14 In the example of **Section 5.6** involving the addition of a start-date column to a set of tables defining employee information, it was stated that a straightforward way to remove the inference channel is to add the start-date column to the employees table. Suggest another way.

5.15 Consider a database table that includes a salary attribute. Suppose the three queries **sum**, **count**, and **max** (in that order) are made on the salary attribute, all conditioned on the same predicate involving other attributes. That is, a specific subset of records is selected and the three queries are performed on that subset. Suppose the first two queries are answered, and the third query is denied. Is any information leaked?