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 2021 report [OWAS21] on the 10 most critical Web application security risks listed injection attacks, including SQLi attacks, as the third highest risk, down from top place in the previous reports.
3. The Veracode 2016 State of Software Security Report [VERA16] found that the 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 creating 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 Section 11.2.

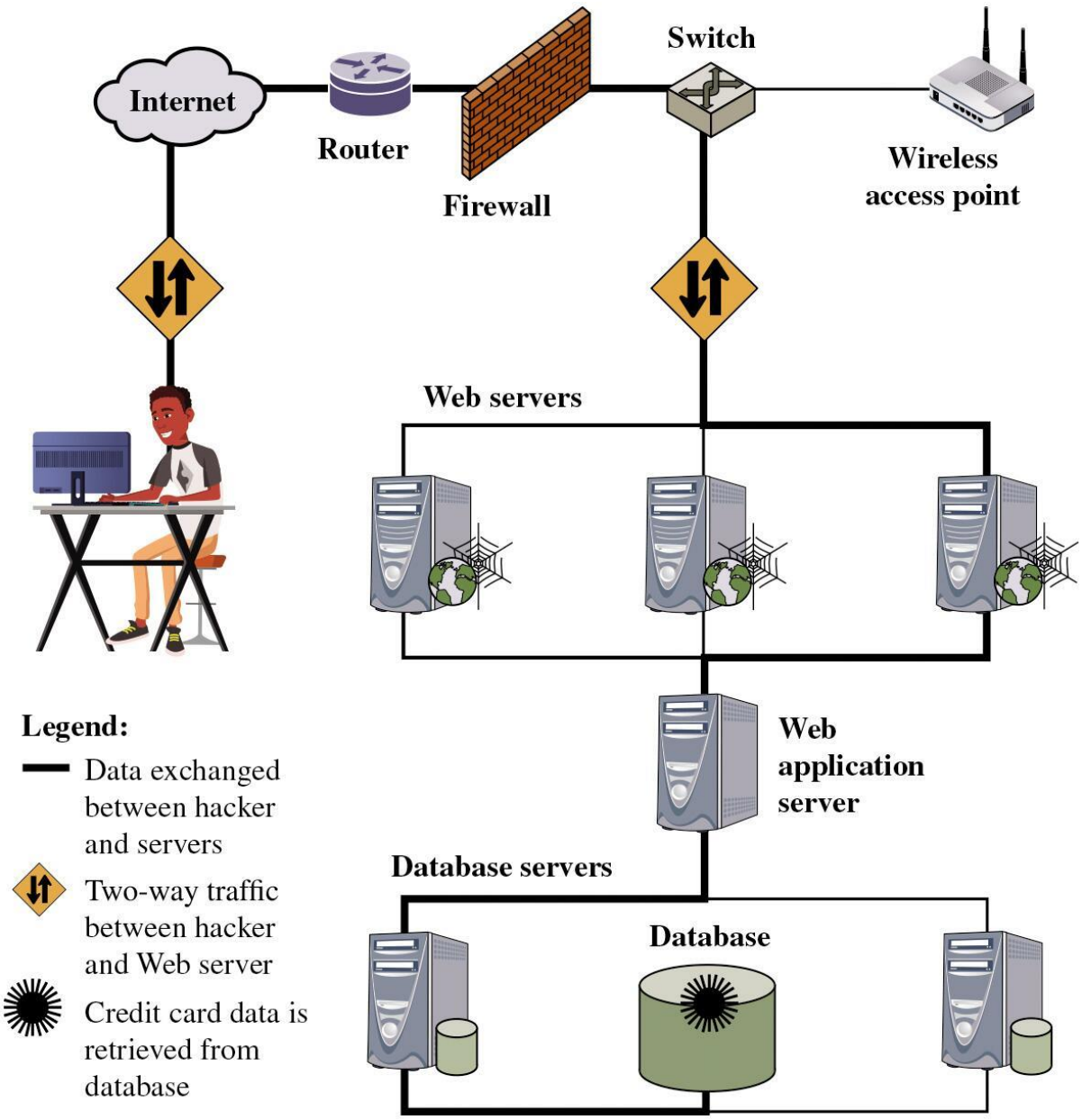
A Typical SQLi Attack

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 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:

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 the 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.

Figure 5.5 **Typical SQL Injection Attack**


Internet
Router
Firewall
Switch
Wireless access point
Web servers
Web application server
Database servers
Database
Legend:
Data exchanged between hacker and servers
Two-way traffic between hacker and Web server
Credit card data is retrieved from database

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 builds 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 that 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 pwd = ‘ ‘

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 is nullified through the use 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, piggybacking 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 message 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 and 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. Some specialized DBMSs also provide mandatory access control. 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 their 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:[17]

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: