Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

SQL Injection

Agenda

- Statistics
- Background
- SQL injections
- SQLi Mitigations

OWASP Top 10 Security Risks

- OWASP Top-10 Web App Vulnerabilities
- A3 – Injection (A1 – 2017)
  - **SQL Injection**
  - Command Injection
  - XXE Injection
  - XPath Injection
  - LDAP Injection
  - Log Injection
  - Http Parameter pollution
  - etc

https://owasp.org/www-project-top-ten/

Statistics

Background

SQL injection

SQLi Mitigation

SQL injection statistics

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2015	343	1093	216	773	146	3	248	49	8	46	0
2016	418	1096	85	476	90	4	85	39	15	28	0
2017	2470	1539	505	1500	281	154	334	109	57	97	931
2018	2078	1729	503	2039	569	112	479	188	118	85	1229
2019	1202	2006	544	2387	485	126	559	136	103	121	895
2020	1216	1847	464	2201	436	108	414	119	130	100	808
2021	1658	2516	742	2724	547	90	520	126	188	133	671
2022	1793	2886	1762	3378	690	87	766	123	230	137	671
2023	1607	2105	2116	5102	742	111	1392	124	240	168	522
2024	1739	2380	2646	7441	923	249	1433	110	372	113	101
2025	1852	2301	3326	7120	845	426	1612	93	449	131	0
Total	16376	21498	12909	35141	5754	1470	7842	1216	1910	1159	5828

Statistics

Background

SQL injection

SQLi Mitigation

SQL injection statistics

SQL injection vulnerability in MOVEit Transfer leads to data breaches worldwide

Zbigniew Banach - Thu, 08 Jun 2023 -

SQL injection vulnerability found in Trusted Tools Free Music v.2.1.0.47, v.2.0.0.46, v.1.9.1.45, v.1.8.2.43 allows a remote attacker to cause a denial of service via the search history table

CVE-2024-49691

Improper Neutralization of Special Elements used in a Product Filter by WBW allows SQL Injection.This issu

Source: Patchstack

CVE-2024-49681

Improper Neutralization of Special Elements used in a Time Monitoring Full Automatic allows SQL Injection. n/a through 1.0.9.

Source: Patchstack

CVE-2024-49623

Improper Neutralization of Special Elements used in a Duplicate Title Validate allows Blind SQL Injection.Thi

Source: Patchstack

Music Player, MP3 Player

Trusted Tools

Contains ads · In-app purchases

4.5★ 161K reviews | 10M+ Downloads | PEGI 3

Install | Add to wishlist

Statistics

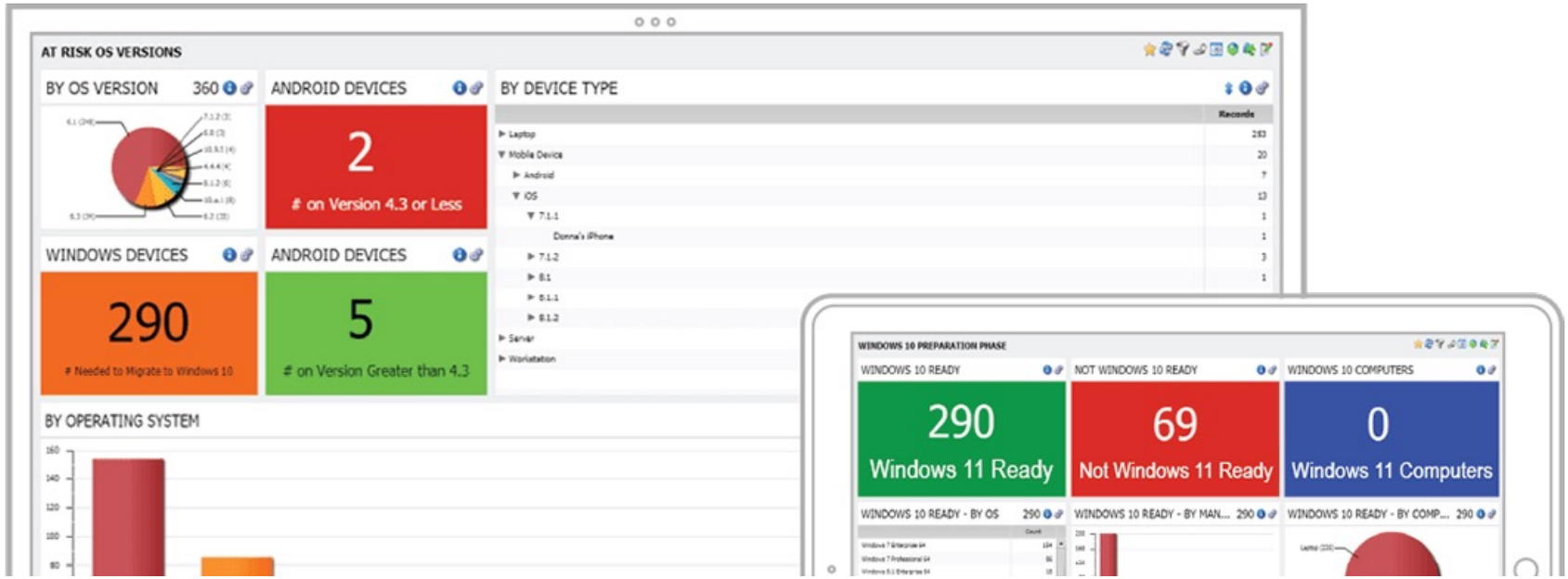
Background

SQL injection

SQLi Mitigation

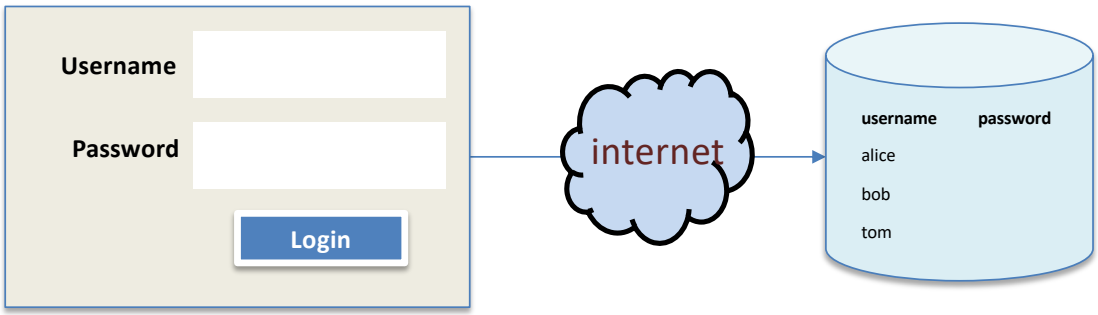
SQL injection statistics

- Ivanti Endpoint Manager SQL Injection Vulnerability Allows Remote Authenticated Data Disclosure (**CVE-2025-62392**)



SQL Interfaces

- Web applications are mostly glorified database applications
- Programmers interface the web app to database via e.g., SQL (Structured Query Language)



SQL interfaces & Untrusted data

- **Source and Sink**
  - **Source**: Entry point where external data enters the application
  - **Sink**: Exit point where such data is consumed/processed by the application
- If the source of the input data is untrustworthy, then data is said to be tainted
  - Web parameters and cookies
  - Data from files, Data from databases
  - Data from web services
  - Environment variables
  - Open ports
- If tainted data reaches a sensitive sink, a security issue may exist
  - e.g., a SQL engine

SQL - Attack surface (Web App)

```
GET http://localhost:9090 HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:68.0) Gecko/20100101
Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:9090/OWASPTopVulnerability/CommandInjection
Connection: keep-alive
Cookie: username=test1; role=user; JSESSIONID=BC48897440437F27C7508892394414C4
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
Host: localhost:9090
```

```
<form action="" method=post>
        <select name='filename'>
                <option value='page1' selected>page1</option>
                <option value='page2'>page2</option> </select>
        <input type=submit name=View>
</form>
```

- HTTP header info
  - e.g. Referrers, etc.
- Form fields
  - Textarea, buttons, password, text, check boxes, etc
- Cookies, files, databases, image, url, css,
- Environment variables, Open ports

SQL Commands

- Data Control Language (DCL)
  - GRANT, REVOKE
- Data Definition Language (DDL)
  - CREATE, ALTER, DROP, TRUNCATE
- Data Query Language (DQL)
  - SELECT
- Data Manipulation Language (DML)
  - INSERT, UPDATE, DELETE

SQL Injection

- SQL Injection occurs when untrusted data are added to database queries to change the application's design
- Dynamic string building when programmers mix query language fragments and untrusted data

```
String query = "SELECT id, name, short_name FROM company WHERE id = " + request.getParameter("company_id");
```

The injection vector here is from the ***request.getParameter()*** which is determined at runtime!!!

See: https://www.websec.ca/kb/sql_injection

SQL Injection

- First-Order SQLi
  - Adds unsafe user input and processes it immediately
- Second-Order SQLi
  - Stores unsafe user input for future use and later uses this unsafe input with SQL query
  - Also known as stored SQL injection

SQLi - Impact

A successful SQL injection exploit can:

- Read and modify sensitive data from the database
- Execute administrative operations on the database
  - Shutdown auditing or the DBMS
  - Truncate/drop tables and logs
  - Add/delete users
- Recover the content of a given file present on the DBMS file system
- Issue commands to the operating system

Statistics

Background

SQL injection

SQLi Mitigation

SQLi - Impact

- Confidentiality, Integrity, Availability, Authentication, Authorization, Auditing (Non-Repudiation)
- Allows attackers to
  - Spoof identity
  - Tamper with existing data
  - Cause repudiation issues such as voiding transactions or changing balances
  - Allow the complete disclosure of all data on the system
  - Destroy the data or make it otherwise unavailable
  - Become administrator of the database server

SQLi – Common Attacks

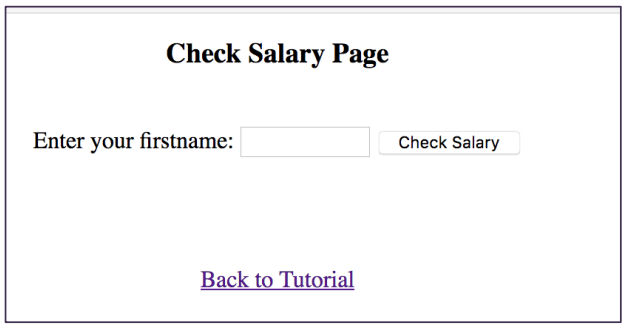
Retrieving hidden data	can modify an SQL query to return additional results.
Subverting application logic	can change a query to interfere with the application's logic.
UNION attacks	can retrieve data from different database tables.
Examining the database	can extract information about the version and structure of the database.
Blind SQL injection	results of a query you control are not returned in the application's responses.

SQLi – Retrieving hidden data

An employee can only see his record stored in a MySQL db when logged in to the system

A ‘curious’ employee wants to see other people’s salary.

- The system has an interface where he can query the record with his userid




```
public void getUserSalary (String user) {
     String query = "SELECT * FROM sal_table WHERE
     FirstName='"+user+"'";

     Statement stmt = conn.createStatement();
     stmt.execute(query);
     ResultSet rs = stmt.getResultSet();

}
```

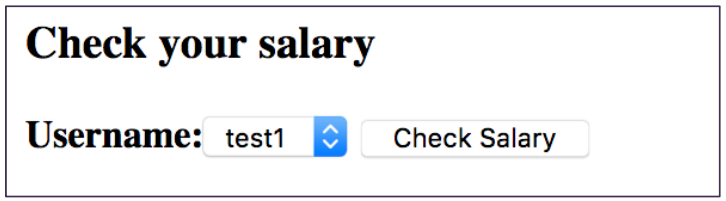
with a payload: **user =** “test’ or ‘1’=‘1’– “

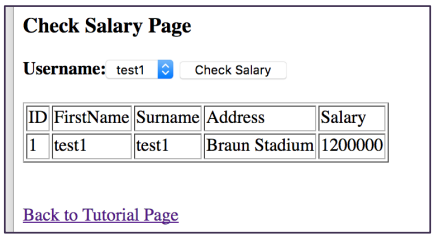
SQLi - Examples

An employee can only see his record stored in a MySQL db when logged in to the system

A curious employee wants to see other people's salary.

- Once logged in, he can only see his userid from where he can request for his salary record.



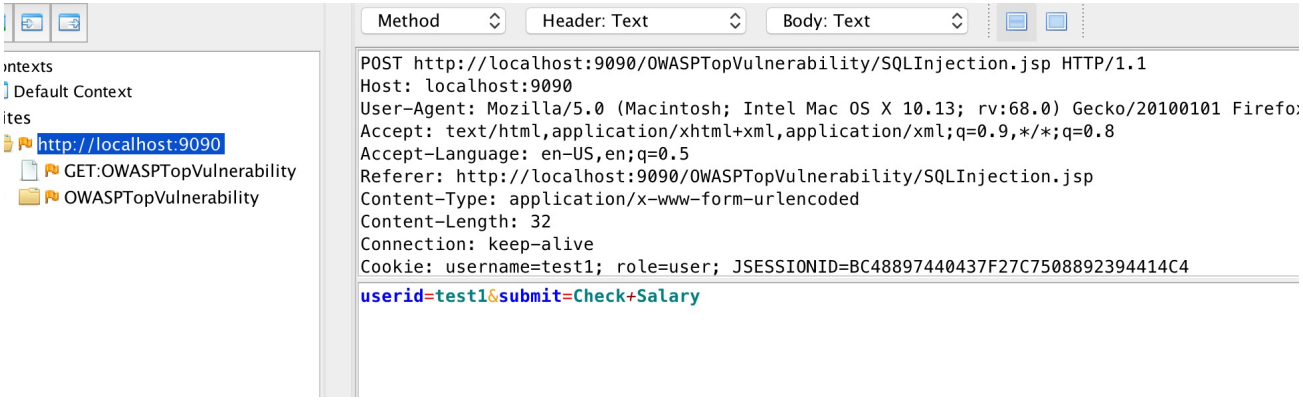


How can we bypass this?

SQLi - Examples

How can we bypass this?

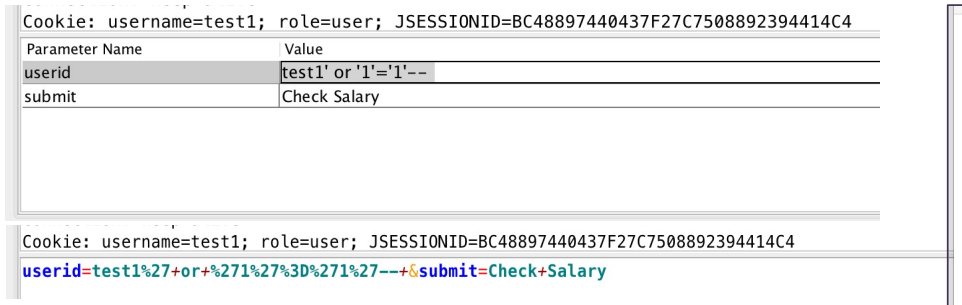
- One way is it to use a proxy between the web client and the web server
- The proxy accepts request packets from the client and forwards to the server, get the response from the server and forwards to the client

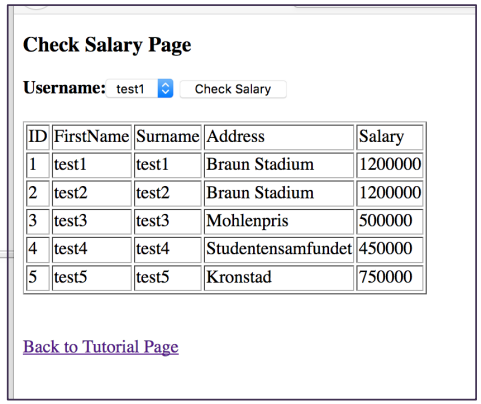


SQLi - Examples

How can we bypass this?

- Use a proxy between the web client and the web server
- The proxy accepts request packets from the client and forwards to the server, get the response from the server and forwards to the client

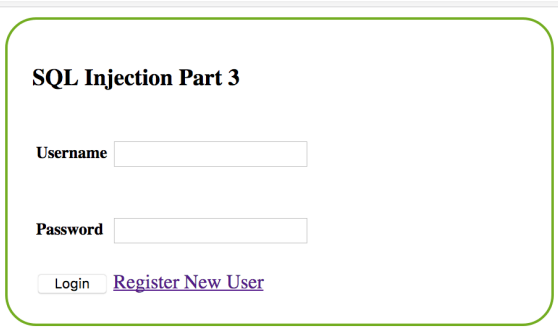




Lesson #0: Validate at the server side: **NEVER** rely on client side validation for critical functions

SQLi – Subverting application logic

- An attacker wants to bypass an authentication system

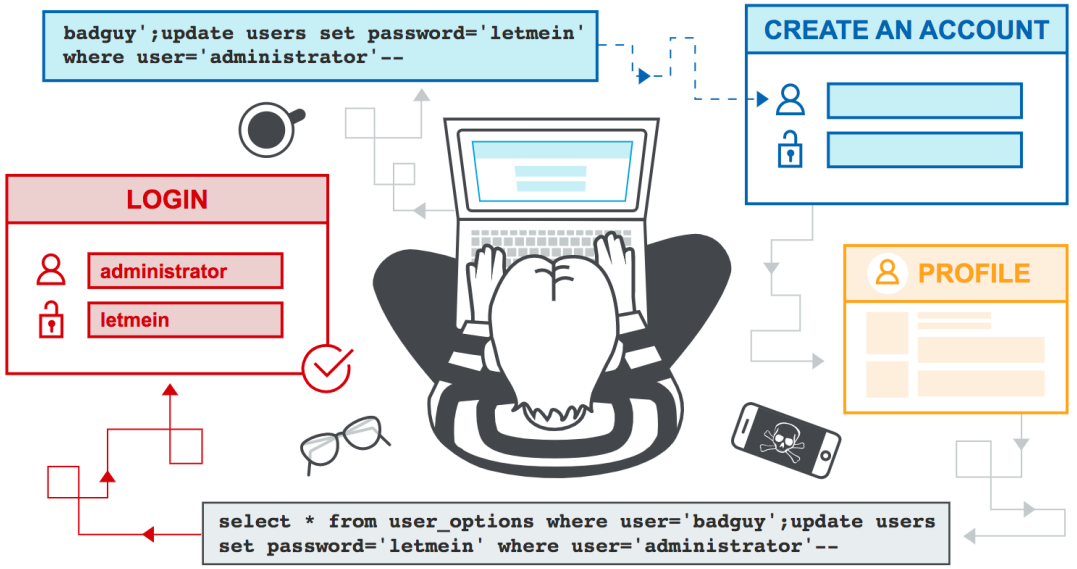



```
public String authenticateWithSalt(String userid, String password) {
     String query0 = "SELECT salt FROM owasp_users WHERE username = '"+userid+"'";
     String salt = getSalt(query0);
     String passhash = null;
     try {
          byte[] saltbytes = DatatypeConverter.parseHexBinary(salt);
          PasswordHash ph = new PasswordHash(PasswordHash.SHA256);
          passhash = ph.generateHashWithSalt(password, saltbytes);
     } catch (NoSuchAlgorithmException e) {
          //e.printStackTrace();
     }
     String query = "SELECT * FROM owasp_users WHERE username = '"+userid+"' AND
                    password = '"+passhash+"'";
```

username = *"test1' or 1 = 2-- "*

SQLi – Subverting application logic

- An attacker registers as a new user but changes the administrator's password



Statistics

Background

SQL injection

SQLi Mitigation

SQLi – Union Attack

- Example: Probe the user's database to retrieve other users login credentials

```
SELECT * FROM sal_table WHERE FirstName = '" + untrusteddata
```

untrusteddata = test1' UNION SELECT id, username, password, salt, 1 FROM owasp_users– '

Check Salary Page

Enter your firstname: test1' UNION SE Check Salary

ID	FirstName	Surname	Address	Salary
1	test1	test1	Braun Stadium	1200000
1	test1	3B58DF45DAC1057E9124828E7863D17F2D79C6DC	9F62DA43ED27A784304E167C36D17DA1	1
4	test2	230E0910D7F8A8F6FE95ED60D3021A705292F178	9FFBA66B74FCFDF9F5767ADD5CE49235	1
5	admin	F0DC57086469B80897AB2C71B30CA981CBA555CA	0384B950C5AB9C3B92B50F5332AA4804	1
6	test3	DC004FBE189EF1B5B95C1EDD1807B6A883C01BE1	88F455A33E4FE51F5B49393DA33BBF14	1
7	test4	72D5D23FECC9056E6D0AC3F3A3879C135293869D	7F33CC89669661BCB08482D0D32C42B1	1
8	test5	0B840D2E1A5756A9E732054270F2D26530A55605	405FCB8184162DAE6C72D17FEA35B9C0	1

Back to Tutorial

SQLi – Examining the database

- Purpose: Gather some info about the db
  - Type of DB
  - Version of DB engine
  - Tables in the DB
  - Columns in the tables

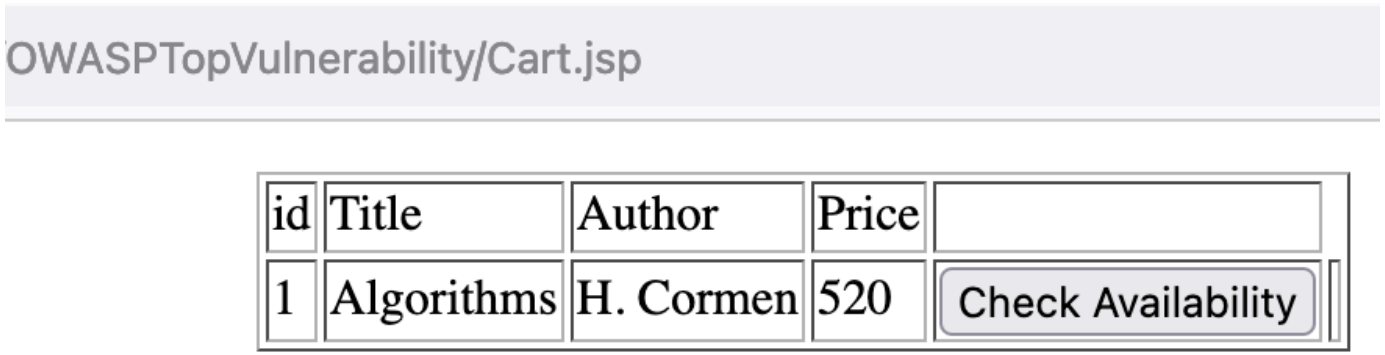
```
SELECT @@version–
SELECT * FROM information_schema.tables
SELECT * FROM information_schema.columns WHERE table_name = 'Users'
```

Blind SQLi

- Blind SQL injection arises when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors.
- Use case: Retrieving information (e.g., password)

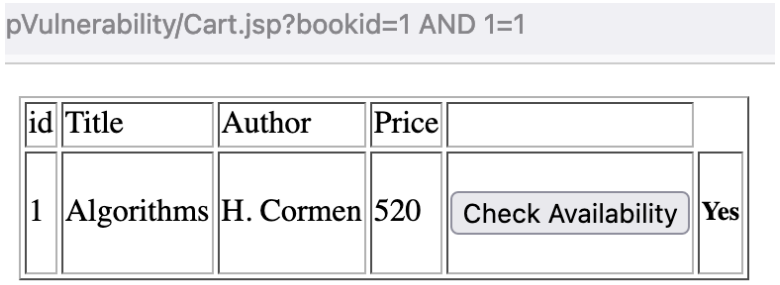
Blind SQLi

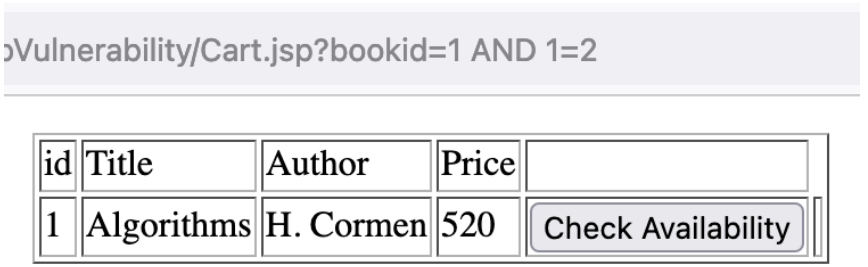
- Assume a book cart where you can check whether a displayed book is available or not
  - If available, a “Yes” is displayed
  - If unavailable. Nothing is shown



Blind SQLi

- Payloads
  - Cart.jsp?bookid=1 AND 1=1
    - Will evaluate to true (Yes is displayed)
  - Cart.jsp?bookid=1 AND 1=2
    - Will evaluate to false (Nothing is displayed)





Blind SQLi

- We can use this behaviour to exploit blind SQLi and retrieve admin or a user's password
- Assumption:
  - We know the database table we want to attack
- First, we may determine the length of the password for a given username that we guess
- e.g.,
  - "1 AND 1=(SELECT 1 FROM owasp_users WHERE username='admin' AND length(password) > 40)-- " = false (nothing is displayed)
  - "1 AND 1=(SELECT 1 FROM owasp_users WHERE username='admin' AND length(password) < 40)-- " = false(nothing is displayed)
  - "1 AND 1=(SELECT 1 FROM owasp_users WHERE username='admin' AND length(password) = 40)-- " = **true** (Yes is displayed)
  - Or
  - "1 AND 1=(SELECT 1 FROM owasp_users WHERE username='admin' AND length(password) < 41)-- " = **true** (Yes is displayed)

Blind SQLi

- for (i=1 to 40){
  - for(j=0 to alphanum.length-1){
    - char = alphanum[j]
    - "1 AND 1=(SELECT 1 FROM owasp_users WHERE username='admin' AND substr(password, i,1) = char')-- "
    - if(result.query == true){
      - save(char)
      - break;
    - }
  - }
- }
- String[] alphanum = {"A", "B", "C", "D", "E", "F","0","1","2","3","4","5","6","7","8","9"};

Blind SQLi

Suppose the application performs the same query but does not behave differently (no visible response can be inferred)

- Can induce conditional responses by triggering SQL errors
- Can trigger time delays

```
xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END)='a
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a
```

```
xyz' AND (SELECT CASE WHEN (Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') THEN 1/0 ELSE 'a' END FROM Users)='a
```

```
'; IF (1=2) SLEEP(10)--
'; IF (1=1) SLEEP(10)--
```

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') = 1 SLEEP(10)'--
```

SQLi - Mitigations

Primary defenses

- Option 1: Use of Prepared Statements (with Parameterized Queries)
- Option 2: Use of Properly Constructed Stored Procedures
- Option 3: Allow-list Input Validation
- Option 4: Escaping All User Supplied Input

Additional defenses

- Enforcing Least Privilege
- Performing Allow-list Input Validation as a Secondary Defense

➢ https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

SQLi - Mitigations

Option 1: Parameterized query – Most reliable technique

- Parameterization is a mechanism where query data is separated from query structure
- Separates command/code from data
- In Java, this is provided via the: java.sql.PreparedStatement

```
String query = "SELECT * FROM sal_table WHERE FirstName=?";
```

```
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setString(1, userInput);
ResultSet rs = pstmt.executeQuery();
```

Query is prepared first and binds data later

```
Hibernate: insert into author (firstname,lastname,author_id) values (?,?,?)
Hibernate: select a1_0.author_id,a1_0.firstname,a1_0.lastname from author a1_0 where a1_0.author_id=?
```

SQLi - Mitigations

Option 2: Stored Procedures

```
// This should REALLY be validated
String custname = request.getParameter("customerName");
try {
  CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
  cs.setString(1, custname);
  ResultSet results = cs.executeQuery();
  // … result set handling
} catch (SQLException se) {
  // … logging and error handling
}
```

When used properly = parameterized query

SQLi - Mitigations

Option 3: Allow-list Input Validation

Bind variables cannot be used in some parts of SQL queries
e.g., Table or column names

```
String tableName;
switch(PARAM):
  case "Value1": tableName = "fooTable";
                 break;
  case "Value2": tableName = "barTable";
                 break;
  ...
  default      : throw new InputValidationException("unexpected value provided"
                                                    + " for table name");
```

SQLi - Mitigations

Option 4: Escaping All User-Supplied Input (Highly Discouraged!)

- Escape all user supplied input using the proper escaping scheme for the database you are using

```
String username = "victim' OR 1=1 -- ";
```

```
String sql = "SELECT * FROM SecOblig.AppUser WHERE username = '"
+ username + "' AND passhash = '" + hashedPassword + "'";
```

```
SELECT * FROM SecOblig.AppUser WHERE username = 'victim' OR
1=1 -- ' AND passhash = '';
```

```
escape(username) = "victim\' OR 1=1 - ";
```

Query will fail

```
SELECT * FROM SecOblig.AppUser WHERE username = 'victim '' OR
1=1 -- ' AND passhash = '';
```

SQLi – Mitigations (Additional)

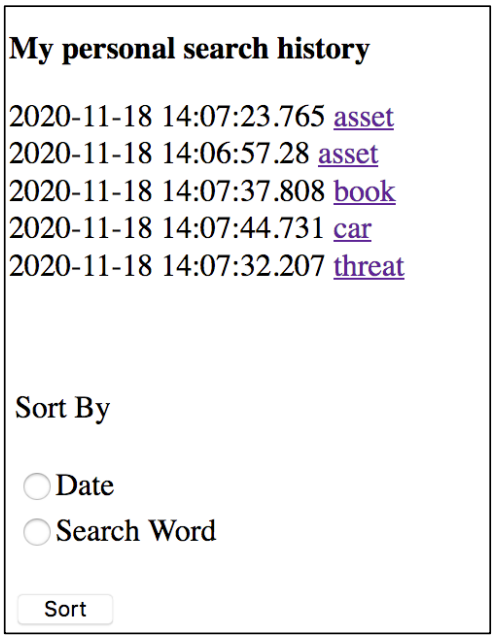
- **Least privilege**: minimize the privileges assigned to different database accounts (avoid DBA/admin access rights to all accounts)
- Verify the number of actual results to expected results
- **Type safely**: If you're expecting a date object, force the result into a date object

SQLi – Mitigations (Extra)

- Question: Does a prepared statement always prevent against an SQL injection?
  - Answer: **No**
  - e.g., If user input is used to provide sorting ("**order by**" clause)

SQLi – Mitigations (Extra)

- Order by



SecOblig.History table has the following columns and datatypes.

- datetime TIMESTAMP
- username VARCHAR(50)
- searchkey VARCHAR(50)

```
public List<SearchItem> getSearchHistoryForUser(String username, String sortkey){
    String sql = "SELECT * FROM SecOblig.History WHERE username = '" + username
        + "' ORDER BY "+sortkey+" ASC";
    return getSearchItemList(sql,50);
}
```

SQLi – Mitigations (Extra)

Answer:
1. perform input validation on the **sortkey** to make sure it can only be either **datetime** or **searchkey** column
2. Rewrite the query such that the username is parameterized and the validated column from (1) is included in the query.

1. Input validation (example) – Note that you can also store the expected columns in a "Allow list" and check sortkey against this list.

```
String sql = "String sql = "SELECT * FROM SecOblig.History WHERE username = '" + username";
If(sortkey.equals("Date"){
     sql = "SELECT * FROM SecOblig.History WHERE username = ? ORDER BY datetime ASC";
} else if(sortkey.equals("Search Word"){
   sql = "SELECT * FROM SecOblig.History WHERE username = ? ORDER BY searchkey ASC";
} else {
  // log/throw exception – invalid input
}
```

2. Then use **preparedstatement**

```
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, username);
ResultSet rs = pstmt.executeQuery();
```

2mins exercise

Question: Is the code below vulnerable to SQLi?

```
public void doPrivilegedAction(
  String username, char[] password
) throws SQLException {
  Connection connection = getConnection();
  if (connection == null) {
    // Handle error
  }
  try {
    String pwd = hashPassword(password);
    String sqlString = "select * from db_user where username=" +
      username + " and password =" + pwd;
    PreparedStatement stmt = connection.prepareStatement(sqlString);

    ResultSet rs = stmt.executeQuery();
    if (!rs.next()) {
      throw new SecurityException("User name or password incorrect");
    }

    // Authenticated; proceed
  } finally {
    try {
```

Source: wiki.sei.cmu.edu

Next Lecture
Security Testing - SAST and DAST