Vulnerability Assessment Report

SQL Injection

Application: WebGoat

Chloe Tremmel
26 Apr 2023

Description:

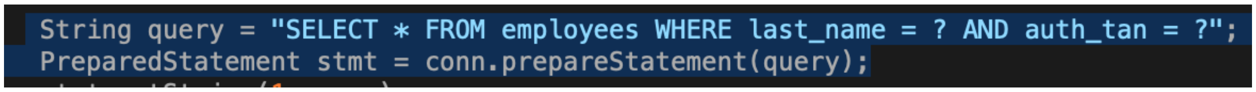
SQL (structured query language) Injection is a widespread vulnerability that affects many industrial database applications. It occurs when an attacker inserts malicious SQL code into areas of the application that accept user input and interact with the database. This enables the attacker to manipulate the database to gain unauthorized access, potentially resulting in the theft of sensitive data, data modification, or straight-up data deletion. Due to the severity of the consequences, SQL injection is considered a **high-risk** vulnerability that requires prompt attention and mitigation. The vulnerable web application used for demonstrating the SQL injection vulnerability is called Web Goat. It is an intentionally-vulnerable web application designed for learning penetration testing. In particular, Module 8 of Web Goat features a high-risk SQL injection vulnerability, which this report aims to address by demonstrating effective mitigation techniques. By understanding and mitigating this vulnerability, readers can improve their understanding of SQL injection and better protect their own web applications against this common and dangerous attack.

Impact:

The impact of an SQL injection vulnerability can vary significantly, depending on the security measures in place. In the absence of source code fortification, input validation, or effective web application firewall rules, attackers can exploit the vulnerability to gain unauthorized access to sensitive data, modify or delete data, and even take control of the entire application. The consequences of such an attack can be severe, ranging from reputational damage to financial loss, legal liability, and compliance issues. Therefore, it is *crucial* to implement appropriate security measures, such as input validation and web application firewall rules, to mitigate the risk of SQL injection attacks.

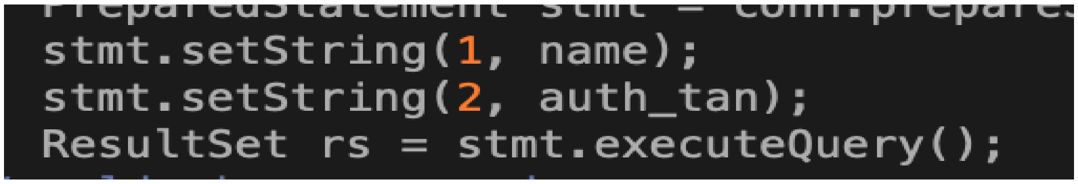
Recommendations:

1. Use parameterized queries or prepared statements, which can help prevent SQL injection by separating user input from the query itself.
   Example:
   Prepared statement:



```
String query = "SELECT * FROM employees WHERE last_name = ? AND auth_tan = ?";
PreparedStatement stmt = conn.prepareStatement(query);
```

   Parameterized query:



```
stmt.setString(1, name);
stmt.setString(2, auth_tan);
ResultSet rs = stmt.executeQuery();
```

2. Implement input validation and sanitization techniques to ensure that user input is safe and does not contain malicious SQL code.
   Example:

```
if (rs.next()) {
        User user = new User(rs.getString("name"), rs.getInt("auth_tan"));
        return user;
      } else {
        return null;
      }
```

   This prevents SQL injection attacks because any malicious code entered by the user will be treated as pure data (in this case either string or int), rather than executable code.

3. Perform regular security assessments and penetration testing to identify and address vulnerabilities, including SQL injection and develop and release patched versions of the application whenever necessary.

4. Use a web application firewall (WAF) to help protect against SQL injection and other types of attacks.
   Example: If you're running an application from a Linux server, you can use the 'iptables' command line utility to identify and block SQL injection attempts via the firewall.

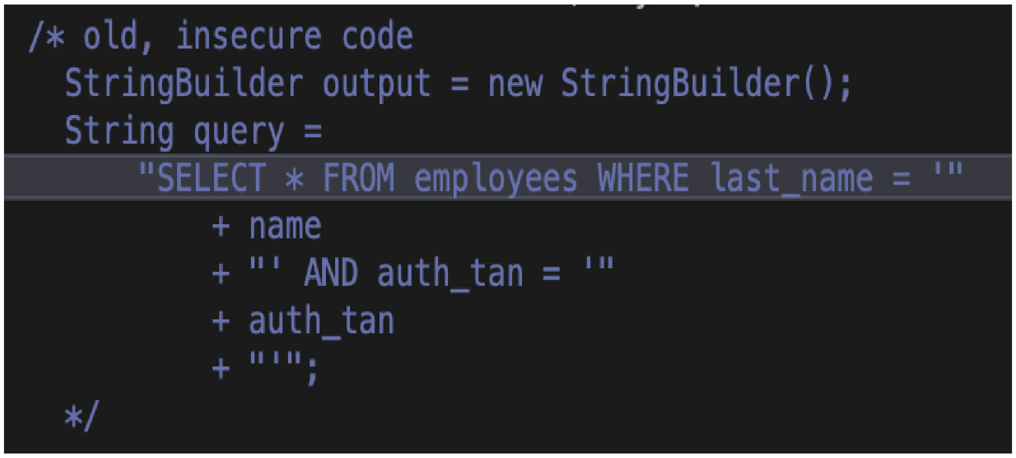
```
iptables -A INPUT -p tcp --dport 3306 -m string --string "union" --algo bm -j DROP
```

   Windows servers should do the same thing but with other firewall configuration utilities.

Code Analysis:

Initial, insecure code:

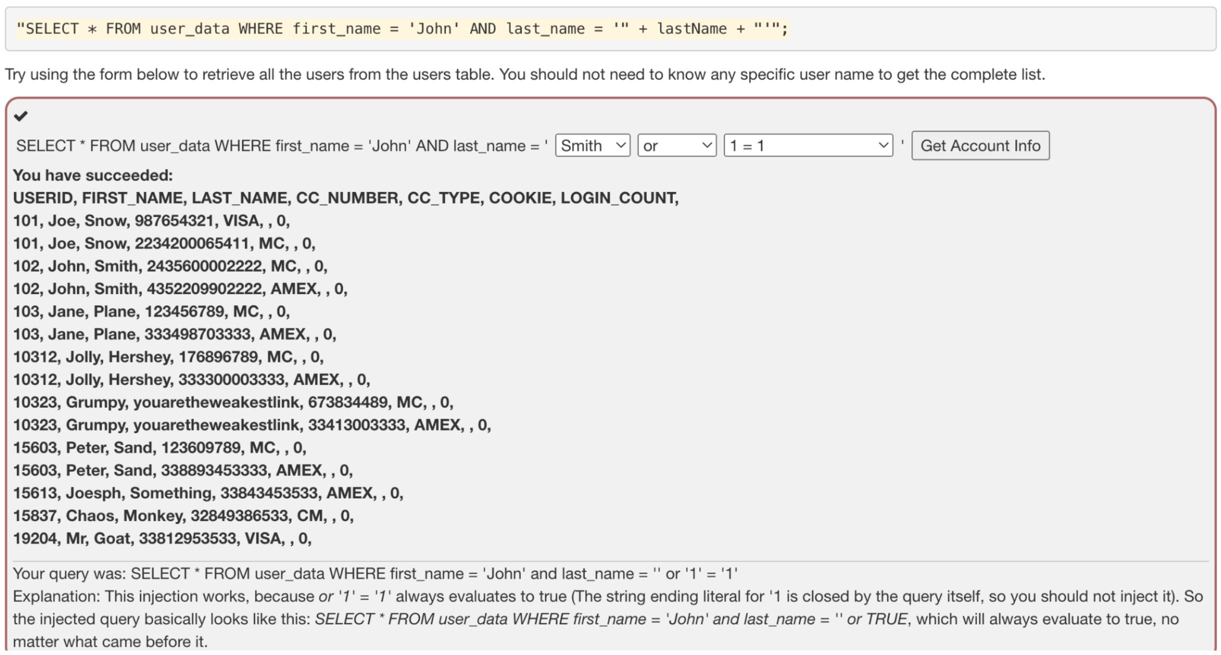
Figure 1



```
/* old, insecure code
  StringBuilder output = new StringBuilder();
  String query =
      "SELECT * FROM employees WHERE last_name = '"
          + name
          + "' AND auth_tan = '"
          + auth_tan
          + "'";
  */
```

Result produced (when uncommented):

Figure 1.2



```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + lastName + "'";
```

Try using the form below to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.

✓

SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ' Smith or 1 = 1 ' Get Account Info

You have succeeded:

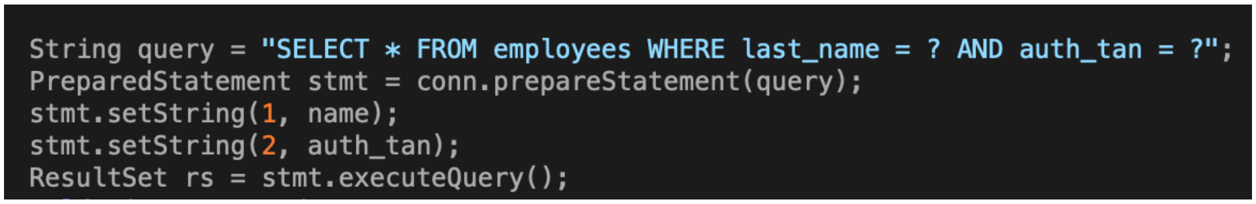
```
USERID, FIRST_NAME, LAST_NAME, CC_NUMBER, CC_TYPE, COOKIE, LOGIN_COUNT,
101, Joe, Snow, 987654321, VISA, , 0,
101, Joe, Snow, 2234200065411, MC, , 0,
102, John, Smith, 2435600002222, MC, , 0,
102, John, Smith, 4352209902222, AMEX, , 0,
103, Jane, Plane, 123456789, MC, , 0,
103, Jane, Plane, 333498703333, AMEX, , 0,
10312, Jolly, Hershey, 176896789, MC, , 0,
10312, Jolly, Hershey, 333300003333, AMEX, , 0,
10323, Grumpy, youaretheweakestlink, 673834489, MC, , 0,
10323, Grumpy, youaretheweakestlink, 33413003333, AMEX, , 0,
15603, Peter, Sand, 123609789, MC, , 0,
15603, Peter, Sand, 338893453333, AMEX, , 0,
15613, Joesph, Something, 33843453533, AMEX, , 0,
15837, Chaos, Monkey, 32849386533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,
```

Your query was: SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or '1' = '1'

Explanation: This injection works, because *or '1' = '1'* always evaluates to true (The string ending literal for '1 is closed by the query itself, so you should not inject it). So the injected query basically looks like this: *SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or TRUE*, which will always evaluate to true, no matter what came before it.

Fixed, secure code:

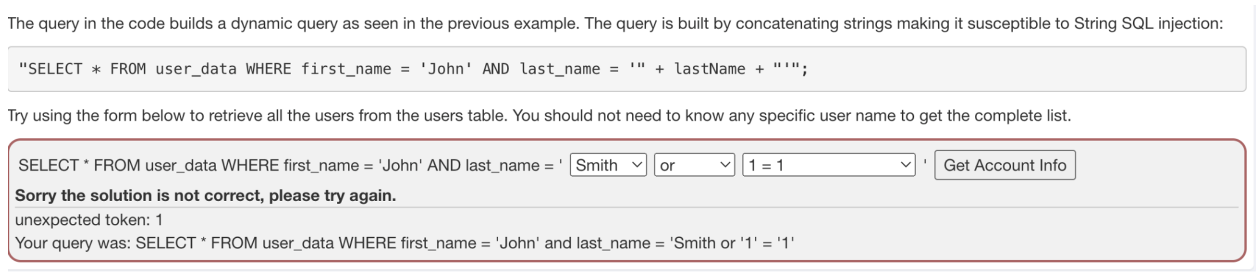
Figure 2



```
String query = "SELECT * FROM employees WHERE last_name = ? AND auth_tan = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, name);
stmt.setString(2, auth_tan);
ResultSet rs = stmt.executeQuery();
```

Result produced:

Figure 2.2


The query in the code builds a dynamic query as seen in the previous example. The query is built by concatenating strings making it susceptible to String SQL injection:

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + lastName + "'";
```

Try using the form below to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.

SELECT * FROM user_data WHERE first_name = 'John' AND last_name = ' Smith or 1 = 1 '

Sorry the solution is not correct, please try again.

```
unexpected token: 1
Your query was: SELECT * FROM user_data WHERE first_name = 'John' and last_name = 'Smith or '1' = '1'
```

Explanation:
Figure 1 demonstrates the creation of a StringBuilder object to construct an SQL query, which retrieves all data from the 'employees' database based on the user's input. However, this approach uses direct user input rather than placeholder tokens, which can pose a security risk because it'll just concatenate and validate anything the user enters immediately. On the other hand, the code in Figure 2 implements a prepared statement to execute the same query. Unlike Figure 1, prepared statements maintain a clear separation between the SQL query and the input values by using placeholders instead of directly concatenating them. This approach is considered more secure because it prevents SQL injection attacks.

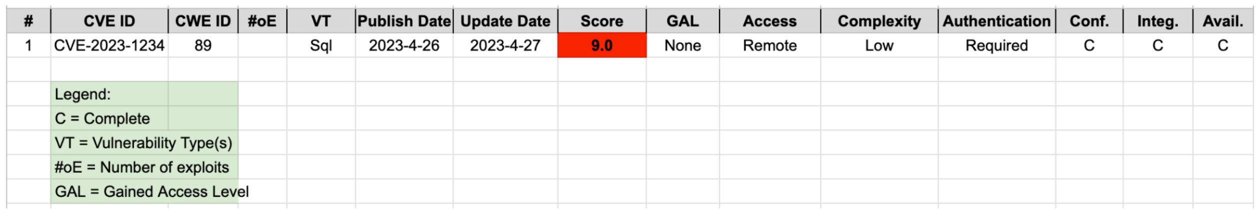
Severity:

As stated before, the severity of a SQL injection vulnerability can vary depending on the specific application and the level of access that an attacker is able to gain. However, given the potential impact of this type of vulnerability, it is generally considered to be a high or critical severity issue.

Potential Financial Impact:

As per leading cybersecurity corporation Imperva, "[the] cost of a minor SQL injection attack can exceed $196,000". In another instance, the US Navy reportedly spent *$500,000* dealing with the consequences of an SQL injection attack. Cyber attacks can be astronomically expensive, but fortunately, they are very preventable. Investing a few thousand dollars in security measures is more cost-effective than paying millions of dollars in security-related damages. The cost of prevention is far less than the cost of recovery, not to mention the potential damage to an organization's reputation, customer trust, and legal liabilities. Therefore, it is essential to prioritize cybersecurity investments and take proactive measures to protect against SQL injection and other types of cyber threats.

CVE Details



#	CVE ID	CWE ID	#oE	VT	Publish Date	Update Date	Score	GAL	Access	Complexity	Authentication	Conf.	Integ.	Avail.
1	CVE-2023-1234	89		Sql	2023-4-26	2023-4-27	9.0	None	Remote	Low	Required	C	C	C

Legend:
C = Complete
VT = Vulnerability Type(s)
#oE = Number of exploits
GAL = Gained Access Level

In WebGoat2023.2 Lesson 8, both the employee id and last name parameter suffer from SQL Injection Vulnerability allowing remote attackers to dump all database credentials and potentially gain admin access (privilege escalation).

CVSS Score: Approx. 8.4

Score interpretation:

Metric	Value	Comments
Attack Vector	Server and Company	The company's MySQL server can be compromised as well as most or all employee accounts and sensitive data.
Attack Complexity	Low	The only required conditions for this attack is for the attacker to have access to a browser to run the web app and an account.
Privileges Required	Low	Attackers must have an account (user privileges).
User Interaction	None	User is required to authenticate successfully to access the vulnerable module.
Scope	Changed	The vulnerable component is the MySQL server database and the impacted component is a remote MySQL server database (or databases).
Confidentially Impact	High	Full compromise of host OS via remote code execution.
Integrity Impact	High	Full compromise of host OS

		via remote code execution.
Availability Impact	Medium	Although injected code is run with high privilege, the nature of this attack prevents arbitrary SQL statements being run that could affect the availability of MySQL databases.