A Study on SQL Attack and Defence Techniques

Vaishnavi S
18MIS1029 (SCOPE)
Vellore Institute of Technology
Chennai

Balaji Chandramouli
18MIS1059 (SCOPE)
Vellore Institute of Technology
Chennai

Prarthana K
18MIS1085 (SCOPE)
Vellore Institute of Technology
Chennai

***Abstract*—SQL, being one of the most widely used database query languages has security issues that need to be addressed. The most common risk is SQL Injection, where entering certain combination of strings in input fields may trigger chaotic responses that give away sensitive information about the back-end of the application. However, there are methods to safeguard the application against such threats. In this paper, we try to expose SQL Injection risks in dummy websites and offer solutions to prevent such attacks.**

***Index Terms*—SQL, injection, defence, attack, query**

I. Introduction

A SQL Query is one of the ways in which an application communicates with a database in order to retrieve or store information or perform similar operations. A SQL injection attack occurs when an unauthorized person tries to retrieve data using an SQL query. It is a threat to web applications and a successful attack can read and modify sensitive data and execute admin operations. During the SQL injection process the commands are inserted into data-plane input in order to affect the execution of predefined SQL commands.In this project we aim to analyze the most common methods of SQL injection and defence methods suitable for them.

II. Overview

Attack There are 2 major ways to implement a SQL attack : the direct and Indirect attack method. In Direct attack method the code directly inserted into the user input variables of the SQL command and executed whereas in Indirect attack method malicious codes are converted to strings that are to be stored in tables and are usually connected to a dynamic SQL command Types

- The escape character is not filtered correctly
- Incorrect type handling
- Vulnerabilities in database servers
- Blind SQL Injection Attack
- Conditional response
- Conditional errors
- Time Delay

Defence SQL injection attacks can be prevented using secure software design practices and by following basic security practices. The defense techniques can be classified into 3 types based on their nature : Prevention, Detection and Deflection. Some of the ways using which the attacks can be prevented are listed below.

- Parametrization of all Queries
- Using stored procedures
- Limiting Permissions
- Validating input
- Concealing error messages
- Using encryption/hash functions when needed
- Segregate data
- Auditing and logging

III. Overall Architecture

The one common thing that every web application have in common, regardless of the programming language in which they were written, is that they are database driven.They simply accept HTTP requests as inputs and create SQL queries as outputs.The parameter values from HTTP requests are used to generate SQL queries. A back-end database accepts that SQL queries and sends back specific information to the web applications as required by web client in the front end. A diagrammatic representation of how a web application and a SQL database communicates is shown in *figure 1*. An example of SQL injection attack and defence is shown in *figure 2* and *figure 3* in the sections below.

Fig. 1. Working of a Web Application with SQL database

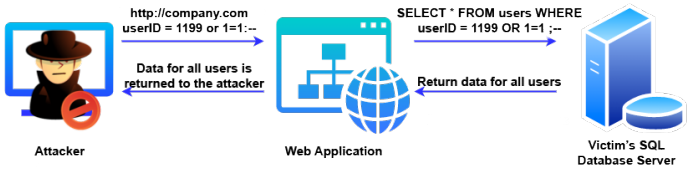

Fig. 2. Example of an SQL injection attack

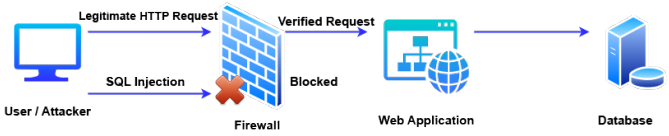

Fig. 3. Example of a SQL injection defence

IV. Methodology

A. Attack

- **SQL Map**
  SQLMap is an open source penetration testing tool that is used to detect and exploit SQL injection flaws of database servers. Using sqlmap, it is possible to view, manipulate or delete data from the databases that use Structured Query Language. A simple procedure to detect and penetrate into the database of a dummy website has been performed. A particular type of vulnerable web pages can be identified by looking at the URL. A good indicator is the presence of '/xxx.php?' after the domain name of the page. Open sqlmap and use the following command to get the names of all the databases present.

  ```
  -u http://testphp.vulnweb.com/
  listproducts.php?cat=1 --dbs
  ```

  In a similar manner, we can hierarchically obtain the names of the tables present in a database, columns present in a table and data present in a column.

  ```
  sqlmap -u http://testphp.vulnweb.com/
  listproducts.php?cat=1 -D
  acuart -T users --columns

  sqlmap -u http://testphp.vulnweb.com/
  listproducts.php?cat=1 -D
  acuart -T users -C uname,pass --dump
  ```

  Since sensitive information such as passwords and login IDs can be obtained from the metadata, the hacker can obtain the administrator's credentials and proceed to use, change or delete the data.

- **Blind SQL Injection using Python Script Automation and Burpsuite**
  Sometimes, when the backend of the database is built with insecure concatenations, SQL Injection can be easily be easily performed by entering the value " ' or 1 =1 " or a similar term that would end the query after the first single quote and execute the subquery 1=1 which would result in the condition being automatically true. However, this will not succeed all the time. For better protected websites, another attack type called Blind SQL Injection. Blind SQL injection is a type of SQL injection attack that tries to get responses from the database in the form of Yes/No questions and determines the answer based on the output. It is generally used when the web page is safeguarded to show only generic error messages. For example, a login form has been brute forced in a dummy penetration testing website called Webgoat. In the form, the following query was provided to get a true or false answer from the database:

  ```
  tom' AND substring(password,1,1) ='a
  ```

  In the register option, we use the query to check if the first letter of the password is a. If it is incorrect, the register option will be accepted. Using brute force, the letter t gave the message that the substring already exists. Therefore, it can be inferred that the first character of the password is t. Similarly we can brute force all the characters manually, but this is not feasible. Hence, a python script that automates the process using the HTTP packets collected using Burpsuite. A snippet of the script is as follows:

  ```
  #!/user/bin/env python3
  import requests
  url = 'http://localhost:8080/WebGoat/
  SqlInjection/challenge'
  headers = {
  'Cookie':
  'JSESSIONID= '...'
  }
  ```

  while running:

  ```
  for c in charset:
       current_char = c
       payload = f"tom' AND
       substring(password,{index},1) =
       '{current_char}"
       send_request(payload, c)
  ```

  Therefore, by appending the character of each index of the password, it is possible to gain the full password easily and access the account of a particular user.

B. Defence

- **Using escape characters in SQL queries**
  If the PHP file for verifying a login page is written as follows it is prone to SQL injection attacks.

  ```
  $uname = $_POST['txt_uname'];
  $password = $_POST['txt_pwd'];
  ```

  A better practice would be to use the

  ```
  mysqli_real_escape_string()
  ```

  function which escapes special characters in a string and takes only the current character set of the connection thereby creating a legal SQL string that can be used in an SQL statement. This is one of the most easiest and best ways to prevent String injections which was discussed earlier. Therefore the above lines of code can be rewritten as

  ```
  $uname = mysqli_real_escape_string
       ($con,$_POST['txt_uname']);
  $password = mysqli_real_escape_string
       ($con,$_POST['txt_pwd']);
  ```

- **Stored Procedures**
  SQL injections usually take place in sites where a user-controlled input is required. The input provided by the user will be inserted in a query directly. The query is then passed and the required information is retrieved. In situations where a query has to be passed

over and over again for example login or search pages, stored procedures can be used. Any frequently used SQL query can be saved as a stored procedure and can be simply called using execution. An example of the stored procedure

```
Create Procedure spSearch
@FirstName nvarchar(100),
@LastName nvarchar(100),
@Gender nvarchar(50),
@Salary int
As
Begin
    Select * from Employeedetails where
    (FirstName = @FirstName OR @FirstName IS NULL AND
    (LastName = @LastName OR @LastName IS NULL) AND
    (Gender = @Gender OR @Gender IS NULL)AND
    (Salary = @Salary OR @Salary IS NULL)
End
Go
```

A stored procedure takes two pre-defined parameters and any number of user-defined parameters.
@statement - The is the mandatory first parameter which should contain the SQL statements to execute
@params - This is an optional parameter which is used to . declare parameters specified in @statement. An example is given below.

```
Declare @statement nvarchar(1000)
Declare @params nvarchar(1000)

Set @statement = 'Select * from Employees where FirstName=@FirstName and LastName=@LastName'

Set @params = '@FirstName nvarchar(100), @LastName nvarchar(100)'

Execute spSearch @statement, @params, @FirstName='',@LastName=''
```

The Database connection for stored procedure can be given using CommandType as follows.

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = con;
cmd.CommandText = "spSearchEmployees";
cmd.CommandType = CommandType.StoredProcedure;
```

The stored procedure can get complex if the number of user input fields increases. Moreover conditions like AND, OR, WHERE etc can make a stored procedure difficult to maintain. These WHERE clauses can however be built dynamically at run time which can reduce the complexity.

- **Dynamic SQL**
  In Dynamic SQL, the SQL statements are constructed dynamically during the run time. Dynamic SQL can be executed by calling the stored procedure. Stored Procedures usually accept the dynamic SQL statement as a Unicode string, hence it has to be prefixed with a char usually 'N'. Dynamic SQL uses StringBuilder class instead of CommandType. An example of dynamic SQL with the stored procedure created previously is shown here.

  ```
  SqlCommand cmd = new SqlCommand();
  cmd.Connection = con;
  StringBuilder sbCommand = new StringBuilder("Select * from Employees where 1 = 1");
  ```

  The executed statement captured using SQL Server Profiler is

  ```
  exec spSearch N'Select * from Employeedetails where 1 = 1 AND Gender=@Gender',N'@Gender nvarchar(4)',@Gender=N''
  ```

  Dynamic SQL is also prone to SQL injections. For example, we have 2 databases Employeedetails and SalaryDetails. The user input is passed as follows.

  ```
  sbCommand.Append(" AND FirstName=' " + inputFirstname.Value + "'");
  sbCommand.Append(" AND LastName=' " + inputLastname.Value + "'");
  sbCommand.Append(" AND Gender =' "+ inputGender.Value + "'");
  sbCommand.Append(" AND Salary= " + inputSalary.Value);
  ```

  These lines directly concatenate the string input given by the user. When we try to drop the SalaryDetails database by entering

  ```
  ' Drop database SalaryDetails–
  ```

  in the any field of the web page. In the SQL Server Profiler we can see that the SQL query passed to be where –' here becomes a comment and as a result the database is dropped

  ```
  Select * from Employees where 1 = 1" AND FirstName=' ' Drop database Dummy–'
  ```

  In order to prevent this user input must be passed as parameters instead of string concatenation. A code snippet for using SQL parameters is given below.

  ```
  sbCommand.Append(" AND FirstName=@FirstName");
  SqlParameter param = new SqlParameter("@FirstName",
  ```

inputFirstname.Value); cmd.Parameters.Add(param);

If we were to use the above instead of concatenating strings the SQL injection doesn't work, the SQL query captured here is

exec spSearch N'Select * from Employees where 1 = 1 AND FirstName=@FirstName',N'@FirstName nvarchar(24)',@FirstName=N''' Drop Database SalaryDetails–'

where the input string is passed as the parameter thereby preventing string concatenation and the attempted SQLinjection.Thus Dynamic SQL statements are also prone to SQL injections but if a web application is created where parameters are used to create dynamic SQL statements, SQL injections can be prevented. Parameterized queries provide better security than concatenating strings. Using parameters to build dynamic SQL statements not only prevents SQL injection, but also increases performance by reusing the cached query plans.

- **Cloud Services to prevent SQLi attacks**
  Cloud computing is one of the potential solutions which enables consumers to acquire cloud resources and services on a self-service and on-demand model. It is really inspired by grid, parallel, and distributed computing via the internet, which uses highly efficient data setters to offer the hardware, software, data, and platform that each application requires.
  Because the cloud stores a large quantity of data, including personal and private information, data security has become a big problem. The internet's accomplishments have made it more powerful and efficient than ever before, making it more widely available than ever before.

  1. Amazon Web Services- Web Application Firewall
  AWS WAF is a web application firewall which will protect our web applications from web exploits or bots that may compromise with security, consume excessive resources or affect availability. AWS WAF gives us control by enabling us to create security rules that control the traffic and attacks caused by bots. It also protects the application from SQLi attacks and cross-site scripting.The rule propagation and updates takes less than a minute to enable security all across our environment when issues arise. Web applications are protected by filtering traffic, which is done using the filters we create. For example, any part of the web request, such as IP addresses, URI strings, HTTP headers can be filtered. This allows us to block common attack patterns like SQL injections.

  2. Google Cloud Armour- Web Application Firewall
  Google Cloud Armor helps protect our infrastructure and applications from protocol-based Volumetric DDoS attacks, Layer 3/Layer 4 DDoS attacks, SQLi attacks, etc. It makes use of Google's global network and Distributed infrastructure to detect the attacks and absorbs them through user-configurable security policies . Cloud Armour has a set of predefined rules for defending against Cross-site scripting and SQL injection attacks. These rules are complex Web Application Firewall rules with multiple signatures which are compiled from Open Source Industry Standards. The rules allow Google Cloud Armor to evaluate distinct traffic signatures by referring to conveniently-named rules, rather than requiring you to define each signature manually.

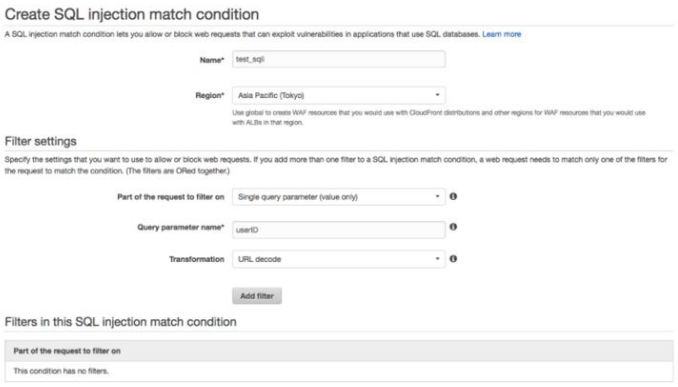

Fig. 4. SQL injection match condition in AWS WAF

Preconfigured ModSecurity rules

SQL injection (SQLi)

Signature ID (CRS Rule ID)	Sensitivity level	Description
owasp-crs-v030001-id942140-sqli	1	SQL injection attack: Common DB Names Detected
owasp-crs-v030001-id942160-sqli	1	Detects blind SQLi tests using sleep() or benchmark()
owasp-crs-v030001-id942170-sqli	1	Detects SQL benchmark and sleep injection attempts including conditional queries
owasp-crs-v030001-id942190-sqli	1	Detects MSSQL code execution and information gathering attempts
owasp-crs-v030001-id942220-sqli	1	Looks for integer overflow attacks

Google Cloud Armor rule name	ModSecurity rule name	Current status
sqli-stable	SQL injection	In sync with sqli-canary
sqli-canary	SQL injection	Latest

Fig. 5. Preconfigured ModSecurity rules in Cloud Armour

V. Conclusion

SQL injections mainly occur in web applications that use SQL databases as the backend for storing and retrieving information. Escaping characters, parameterized queries, whitelisting, typecasting and other forms of prevention

techniques can be breached by creative attackers. To prevent SQL injections completely NoSQL databases can be used which store information as objects. Examples include Firebase, Firestore, etc. Apart from this SQL injection attacks can be avoided by using cloud protection services like Google Cloud armor, AWS Shield, etc which also allow customization of firewalls and security rules as per the need.

REFERENCES

[1] L. Ma, D. Zhao, Y. Gao and C. Zhao, "Research on SQL Injection Attack and Prevention Technology Based on Web," 2019 International Conference on Computer Network, Electronic and Automation (ICCNEA), Xi'an, China, 2019, pp. 176-179, doi: 10.1109/ICCNEA.2019.00042.

[2] C. Ping, "A second-order SQL injection detection method," 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, China, 2017, pp. 1792-1796, doi: 10.1109/ITNEC.2017.8285104.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] İ. KARA and M. AYDOS, "Detection and Analysis of Attacks Against Web Services by the SQL Injection Method," 2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Ankara, Turkey, 2019, pp. 1-4, doi: 10.1109/ISMSIT.2019.8932755.

[5] H. Alsobhi and R. Alshareef, "SQL Injection Countermeasures Methods," 2020 International Conference on Computing and Information Technology (ICCIT-1441), Tabuk, Saudi Arabia, 2020, pp. 1-4, doi: 10.1109/ICCIT-144147971.2020.9213748.