Injection Attacks

Instructor
Davide Maiorca

Web Security and Malware Analysis

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence

University of Cagliari, Italy

Injection Techniques - OWASP

A1:2017-Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

Defining Injection

- Injection is the process of making an application process *user-supplied data as code*
- In particular, injection typically exploits a *server-side* technology (e.g., a database)
- The exploited technology adopts *interpreted* languages
  - Goal: injecting instructions *that can be processed by the interpreter*
  - In this way, it is either possible to exploit the authentication or to obtain sensitive information
- It may also be possible to to compromise a database completely
- Two major types of injections
  - Database (SQL) Injection
  - Command Injection

Databases

- Collections of data that represent parts of the real world
- Many types of Databases
  - Relational (based on relational algebra)
  - Hierarchical
  - Object-based
- Organized in *tables*
  - Each table is made of *records (rows)* and *fields (columns)*
- The ensemble of fields that constitute a record in a table is called *primary key*

SQL

- Acronym for *Structured Query Language*
  - Used in relational databases
  - We will use the *MySQL* implementation

- Data Definition Language Instructions (DDL)
  - **CREATE** Table
  - **ALTER** Table
  - **DROP** Table (delete table)

- Data Manipulation Language (DML)
  - **INSERT INTO** Table (col1, col2) **VALUES** (value1, value2)
  - **DELETE FROM** Table **WHERE** col_name = col_value (deletes record according to condition)
  - **UPDATE** Table **SET** col_name = new_value **WHERE** col_name=col_value

- Queries
  - **SELECT** col1, col2 … **FROM** Table
  - If you use **SELECT DISTINCT,** only the distinct values will be returned

The SELECT Statement

- Extremely important for accessing data in a Database
- To select all columns from a table
  - **SELECT** * **FROM** Table
- To select only specific columns
  - **SELECT** col1, col2 **FROM** Table
- Conditions are expressed with the WHERE statement
  - SELECT * From Table **WHERE** col1_name = value
  - You can combine multiple conditions with **AND** or **OR**
  - Strings are expressed with '… ', numbers without quotes
- Values with **NULL** are empty

SQL-Based Authentication

- Users credentials are generally stored in databases
- A typical way to check if the credentials of the user are correct is by checking if *a database entry exists*
- As a first example, consider that the user *markus* wants to login with the password *secret*
  - The user can supply username and password from a form
  - `SELECT * FROM users WHERE username = 'marcus' and password = 'secret'`
  - If the query returns a result, then the user is authenticated
- However, this method of authentication is extremely vulnerable
  - Can you imagine why?

SQL-Injection

- The problem of this query is that it is real, *interpreted* code
- This aspect means that any modification of the code will be interpreted as *real code*
- What happens if, *username = admin'--?*
  - Starting query: `SELECT * FROM users WHERE username = 'marcus' and password = 'secret'`
  - Resulting query: `SELECT * FROM users WHERE username = 'admin'--' and password = 'secret'`
- Considering that in MySQL «--» represents a comment, the query is interpreted in this way
  - `SELECT * FROM users WHERE username = 'admin'`
- The password check gets completely bypassed

Error-Based SQL Injection

- In many cases, it is possible to test if there is a vulnerability in SQL
- For example, give ‘ as input. If there is a vulnerability, you typically get an error message
- Suppose you have this query: `SELECT author,title,year FROM books WHERE publisher = ‘bla’ and published=1`
  - Suppose you can choose the publisher
  - If you send Mi’ao as a publisher, you may get a SQL error
  - `Incorrect syntax near ‘ao’. Server: Msg 105, Level 15, State 1, Line 1 Unclosed quotation mark before the character string`
- That error shows that a SQL Injection is possible

Data Extraction from Databases

- When exploiting a SQL Injection vulnerability, one possible goal is to obtain *information* from the Database
  - Database organization
  - Tables where *credentials* are stored
  - Financial information

- One common trick is using the OR 1=1 statement

- Let's consider the previous query
  - `SELECT author,title,year FROM books WHERE publisher = 'bla' and published=1`
  - We could send something like this: `wof' OR 1=1--`
  - The result would be: `SELECT author,title,year FROM books WHERE publisher = 'wof' OR 1=1 --' and published=1`

- The result is obtaining all the columns related to author, title and year
  - Another possibility would be: `Wiley' OR 'a' = 'a`

Injection With INSERT and DELETE

- Another way to bypass login would be by inserting fake credentials
- For example, it is possible to use an **INSERT** statement
  - `INSERT INTO users (username, password, ID, privs) VALUES (‘daf’, ‘secret’, 2248, 1)`
  - Careful with the data types though (the SQL query will fail if values of the wrong type are added)
- In many cases, you do not know how many columns the target table has
  - Trial and Error
  - `foo’)--`
  - `foo’, 1)--`
  - …
- **UPDATE** statements are similar to **INSERT**, but you are required to use **WHERE**
- An injection with **DELETE** may also compromise the whole database

Using UNION

- The UNION statement allows appending the results of two queries
- For example: `SELECT Users FROM Credentials UNION SELECT Surnames FROM Users`
  - Returns the column Users extracted from the table Credentials and appends the column Surnames extracted from the table Users
- It is possible to use **UNION SELECT** to dump the contents of a database with an injection
- Consider again the following query: `SELECT author,title,year FROM books WHERE publisher = 'bla' and published=1`
  - Injection on publisher: `bla' UNION SELECT user, password, uid FROM Credentials --`
  - Final query: `SELECT author,title,year FROM books WHERE publisher = 'bla' UNION SELECT user, password, uid FROM Credentials --' and published=1`
  - Careful here: the *number of columns* of the injected query must be the *same* as the original query
  - The data types must be the same as well – but if you do not know it, you can replace it will **NULL**
  - `SELECT author,title,year FROM books WHERE publisher = 'bla' UNION SELECT user, password, NULL FROM Credentials --' and published=1`

Analyzing a Database – 1: Finding Columns

- Suppose you have a page that performs a query on a database
- Also, suppose that you can perform an error-based SQL injection
- The goal to dump information is to use **UNION** by employing the *same number of columns of the original query*
  - You can use **NULL** to probe the correct values until you do not get any error
- Example: Suppose you do not know the table you are injecting data into, but you know that you are injecting into a string parameter.
  - You can start by checking this: `bla' UNION SELECT NULL --`
  - If you get an error, then you can try `bla' UNION SELECT NULL, NULL --`
  - And so on, till you get the right number of columns
- Once found the right number, you can execute *special commands* to start having more information
  - E.g. (with two columns): `bla' UNION SELECT @@version, NULL --`

Analyzing a Database – 2: Finding Databases and Tables

- Once the SQL query completely works, we have to start digging into the database information

- To obtain information about the *databases* contained you can craft the query in this way
  - One parameter of the **SELECT** must be `schema_name`
  - The corresponding **FROM** table must be `information_schema.schemata`
  - E.g `bla' UNION SELECT NULL, schema_name FROM information_schema.schemata`

- Once we know the target database, we can get the corresponding *table names* from the target database
  - Suppose we found a database 'miao'
  - One parameter of the **SELECT** must be `table_name`
  - The corresponding **FROM** table must be `information_schema.tables`
  - **E.g.:** `bla' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema = 'miao'`

Analyzing a Database – 3: Finding Columns and Credentials

- Suppose that from the previous operation we got the table *users*

- The next goal is getting the *column names* contained in the table
  - One parameter of the **SELECT** must be `column_name`
  - The corresponding **FROM** table must be `information_schema.columns`
  - **E.g.:** `bla' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name = 'users'`

- Suppose that we found the columns *user* and *password*
  - The goal is now retrieving the two elements
  - We can also replace the NULL with one of the elements of interest
  - **E.g.:** `bla' UNION SELECT user, password FROM users`

- In this way, you have enumerated all *users* and their corresponding *passwords*

Second Order SQL Injection

- It is possible to protect against SQL Injection by *doubling* the single quotes
- Suppose we give `bla'` as a parameter
  - The parameter is transformed in `bla'',` which means that it becomes 'bla''' in the query
  - This is concatenation in My SQL between 'bla' and ''
  - The instruction is legal
- However, what happens when this validation is performed only *once?*
  - The parameter is saved as `bla'`
  - This means that another query that uses this parameter may fail!
- For example, think about *password recovery*
- *Second Order SQL Injection:* the attack parameter is stored with one query and exploits the vulnerability with a second query that accesses it

Blind SQL Injection

- Also called as *inferential SQL Injection*
- What happens when you do not see *any error message* related to the behavior of the query?
- You can watch how the application behaves and try to *infer information*
  - For example, suppose that `‘ OR 1=1--` makes you log in as admin and `‘ OR 1=2--` does not
  - Suppose that we want to find what the password is, and not just to bypass the mechanism
- You can test this condition to brute force each letter of the password
  - Suppose you can access the value *user_password* with a query
  - You can add a check like this: `‘ OR SUBSTRING(user_password,1,1)= TARGET_VALUE`
  - 1, 1 takes the first letter of user_password
  - The condition is valid only when TARGET_VALUE matches the result of SUBSTRING
  - E.g. `‘ OR SUBSTRING(«Admin»,1,1)= 65 --` makes you in
  - Repeat for each character

Defending Against SQL Injection

- The best way to defend against SQL Injection is by using *parametrized queries*
  - Also known as *prepared statements*

- The structure of the query and the user-supplied input are separated and separately pre-processed.
  - The pre-processed query is handled through an API

- The API manages the parameter so that it does not interfere with the query structure

```
//define the query structure
String queryText = “SELECT ename,sal FROM EMP WHERE
ename = ?”;
//prepare the statement through DB connection “con”
stmt = con.prepareStatement(queryText);
//add the user input to variable 1 (at the first ?
placeholder)
stmt.setString(1, request.getParameter(“name”));
// execute the query
rs = stmt.executeQuery();
```

Command Injection

- SQL Injection is not the only injection possible
- Many web applications employ operating system-related APIs or system calls to handle specific inputs
- If a user injects an arbitrary command and the web application employs specific APIs, it is likely that the command gets executed
- Arbitrary command execution occurs in languages like PHP, with functions such as *eval*
  - `Eval` transforms a string input in a command that can be evaluated/executed

Example of Command Injection

- Suppose that you have a URL with a search function:
  `/search.php?storedsearch=\$mysearch%3dwahh`

- Suppose that, server-side, the parameter is evaluated as the following:

```
$storedsearch = $_GET['storedsearch'];
eval("$storedsearch;");
```

- It is possible to use the marker ; to add further commands that will be parsed by eval

- For example, to visualize the file /etc/passwd, the following can be performed:
  `/search.php?storedsearch=\$mysearch%3dwahh;%20system('cat%20/etc/passwd')`

- You can test the web application for special characters such as ; or & to see if it is possible to trigger a command-injection flaw

Path Traversal Vulnerability

- A variant of the command injection
  - This attack is typically doable when the server performs operations related to the file opening
- For example, consider the following URL
  `http://mdsec.net/filestore/8/GetFile.ashx?filename=keira.jpg`
  - The server opens the URL and reads the file contents
- However, what happens if the path check is not sanitized?
  - `http://mdsec.net/filestore/8/GetFile.ashx?filename=..\..\keira.jpg`
  - We are telling the url to open the keira.jpg file two levels above where it should be
  - The file will not be found
- However, if the vulnerability is present, we can access even the root of the target file system and read any file we want
  - `http://mdsec.net/filestore/8/GetFile.ashx?filename=..\..\..\..\..\etc\passwd`

References

- D. Stuttard and M. Pinto. «The Web Application Hacker's Handbook», Wiley Inc.
  - Chapters 9 and 10
- https://www.w3schools.com/sql/sql_intro.asp
- https://www.owasp.org/index.php/Testing_for_Command_Injection_(OTG-INPVAL-013)