SQL Injection Attacks (SQLi)

Learn what SQLi attacks are, who they target, how they differ from other types of cyberattacks.

INSIGHTIDR PRODUCT

TOPIC OVERVIEW

RELATED TOPICS

Common Cyberattacks

- Types of Cybersecurity Attacks
- Cross-Site Scripting
- Cross-Site Request Forgery
- Denial-of-Service Attacks
- Brute-Force and Dictionary Attacks
- Phishing Attacks
- Whaling Phishing Attacks
- Spear Phishing Attacks
- Spoofing Attacks
- Man-in-the-Middle (MITM) Attacks
- SQL Injection Attacks (SQLi)
- Malware Attacks
- Cybersecurity Attack

What is a SQL Injection Attack?

Structured Query Language (SQL) is a language designed to manipulate and manage data in a database. Since its inception, SQL has steadily found its way into many commercial and open source databases. SQL injection (SQLi) is a type of cybersecurity attack that targets these databases using specifically crafted SQL statements to trick the systems into doing unexpected and undesired things.

If you have less than five minutes, learn about SQL Injection Attacks in this video:

Actions a successful attacker may take on a compromised target include:

- Bypassing authentication
- Exfiltrating/stealing data
- Modifying or corrupting data
- Deleting data
- Running arbitrary code
- Gaining root access to the system itself

How Dangerous are SQL Injections?

If completed successfully, SQL injections have the potential to be incredibly detrimental to any business or individual. Once sensitive data is compromised in an attack, it can be difficult to ever fully recover.

Databases are commonly targeted for injection through an application (such as a website, which requests user input and then does a lookup in a database based on that input), but they can also be targeted directly. SQL injection attacks are listed on the OWASP Top 10 list of application security risks that companies wrestle with.

Types of SQL Injection Attacks

SQL injection attacks can be carried out in a number of ways. Attackers may observe a system's behavior before selecting a particular attack vector/method.

Unsanitized Input

Unsanitized input is a common type of SQLi attack in which the attacker provides user input that isn't properly sanitized for characters that should be escaped, and/or the input isn't validated to be the type that is correct/expected.

For example, a website used to pay bills online might request the user's account number in a web form and then send that to the database to pull up the associated account information. If the web application is building a SQL query string dynamically with the account number the user provided, it might look something like this:

*"SELECT * FROM customers WHERE account = " + userProvidedAccountNumber +";"*

While this works for users who are properly entering their account number, it leaves the door open for attackers. For example, if someone decided to provide an account number of ' or '1' = '1', that would result in a query string of:

*"SELECT * FROM customers WHERE account = " or '1' = '1';"*

Due to the '1' = '1' always evaluating to TRUE, sending this statement to the database will result in the data for *all* customers being returned instead of just a single customer.

Blind SQL Injection

Also referred to as Inferential SQL Injection, a Blind SQL injection attack doesn't reveal data directly from the database being targeted. Rather, the attacker closely examines indirect clues in behavior. Details within HTTP responses, blank web pages for certain user input, and how long it takes the database to respond to certain user input are all things that can be clues depending on the goal of the attacker. They could also point to another SQLi attack avenue for the attacker to try.

Out-of-Band Injection

This attack is a bit more complex and may be used by an attacker when they cannot achieve their goal in a single, direct query-response attack. Typically, an attacker will craft SQL statements that, when presented to the database, will trigger the database system to create a connection to an external server the attacker controls. In this fashion, the attacker can harvest data or potentially control behavior of the database.

A Second Order Injection is a type of Out-of-Band Injection attack. In this case, the attacker will provide an SQL injection that will get stored and executed by a separate behavior of the database system. When the secondary system behavior occurs (it could be something like a time-based job or something triggered by other typical admin or user use of the database) and the attacker's SQL injection is executed, that's when the "reach out" to a system the attacker controls happens.

SQL Injection Example

For this SQL injection example, let's use two database tables, Users and Contacts. The Users table may be as simple as having just three fields: ID, username, and password. The Contacts table has more information about the users, such as UserID, FirstName, LastName, Address1, Email, credit card number, and security code.

The Users table has information used for logins like:

1. jsmith,P@$$w0rd
2. sbrown,WinterIsComing!
3. kcharles,Sup3rSecur3Password$

Note: Passwords should always be hashed and salted when stored in a database and never in cleartext.

When someone wants to log in, they'll go to the login page and enter their username and password. This information is then sent to the webserver, which will construct a SQL query and send that query to the database server. An example of what that query looks like might be:

Select ID from Users where username='jsmith' and password='P@$$w0rd'

The way SQL works is that it will then perform a true or false comparison for each row that the query requests. In our example, the query says to check the Users table and give back the ID value for every row where the username is jsmith and the password is P@$$w0rd. Often, the webserver will then see what is returned by the database server and if it is a number. In our case, the webserver would receive back a 1 and let the user past the login page.

But, what if we want to get malicious with this? Because the database server performs that true-or-false check, we can trick it into believing that we have successfully authenticated. We can do this by adding an OR to the password. If we log in with x' or 1=1 as our password, that will create a new SQL query that looks like:

Select ID from Users where username='jsmith' and password='x' or 1=1

This will work for us, because while x is not jsmith's password, the database server will then check the second condition. If x isn't jsmith's password, then does 1 equal 1? It does! The ID will be sent back to the application and the user will be successfully authenticated.

This doesn't have to be a 1=1 condition. Any two equal values will work, 2=2, 4726=4726 or even a=a.

If a web page is capable of displaying data, it may also be possible to print additional data to the screen. To access the data, we can try to chain together two SQL requests. In addition to our ' or 1=1, we can add on to that a second statement like UNION SELECT LastName, credit card number, security code from Contacts. Extra clauses like this may take some extra work, but getting access to data is the ultimate goal of a SQL injection attack.

Another technique we can use for blind SQL injection, the one where no data is sent back to the screen is to inject other hints. Similar to our ' or 1=1 condition, we can tell the server to sleep. We could add ' " or sleep(10) " and this will do what it seems like. It will tell the database server to take a 10-second nap and all responses will be delayed.

How to Prevent SQL Injection Attacks

The following suggestions can help prevent an SQL injection attack from succeeding:

Don't use dynamic SQL

- Avoid placing user-provided input directly into SQL statements.
- Prefer prepared statements and parameterized queries, which are much safer.
- Stored procedures are also usually safer than dynamic SQL.

Sanitize user-provided inputs

- Properly escape those characters which should be escaped.
- Verify that the type of data submitted matches the type expected.

Don't leave sensitive data in plaintext

- Encrypt private/confidential data being stored in the database.
- Salt the encrypted hashes.
- This also provides another level of protection just in case an attacker successfully exfiltrates sensitive data.

Limit database permissions and privileges

- Set the capabilities of the database user to the bare minimum required.
- This will limit what an attacker can do if they manage to gain access.

Avoid displaying database errors directly to the user. Attackers can use these error messages to gain information about the database.

Use a Web Application Firewall (WAF) for web applications that access databases

- This provides protection to web-facing applications.
- It can help identify SQL injection attempts.
- Based on the setup, it can also help prevent SQL injection attempts from reaching the application (and, therefore, the database).

Use web application security testing to routinely test web apps that interact with databases. Doing so can help catch new bugs or regressions that could allow SQL injection.

Keep databases updated to the latest available patches. This prevents attackers from exploiting known weaknesses/bugs present in older versions.

SQL injection is a popular attack method for adversaries, but by taking the proper precautions such as ensuring data is encrypted, that you protect and test your web applications, and that you're up to date with patches, you can take meaningful steps toward keeping your data secure.

Related Topics

Spoofing Attacks
COMMON CYBERATTACKS
READ TOPIC

Denial-of-Service Attacks
COMMON CYBERATTACKS
READ TOPIC

Spear Phishing Attacks
COMMON CYBERATTACKS
READ TOPIC

VIEW MORE TOPICS