# حمله تزریق SQL چیست؟

حمله تزریق SQL نوعی از حملات وب است  که هکر با اجرای یک سری دستورات مخرب، عملیاتی را روی پایگاه داده آسیب پذیر انجام می‌دهد. در این نوع حمله، نفوذگر از نقص‌های برنامه نویسی و امنیتی که در سایت یا نرم افزار وجود دارد، سوء استفاده می‌کند. در واقع این نوع سایت‌ها و نرم افزارها هدف بسیاری از حمله کننده‌ها برای دستیابی به اطلاعات یک پایگاه داده خواهند بود. پرس و جوهای معمول دارای بخش‌های مختلفی هستند که حملات تزریق SQL فقط در بخش شرطی Where اتفاق می‌افتد. حمله تزریق SQL به راحتی داده‌های مهم موجود در پایگاه داده را افشاء کرده و هم چنین می‌تواند موارد دلخواه خود را به پایگاه داده اضافه کند و یا از آن حذف کند.

این حمله زمانی اتفاق می‌افتد که اپلیکیشن، داده‌هایی که توسط هکر دریافت می‌کند را بدون اعتبارسنجی و پاک سازی، پردازش می‌کند. در بدترین حالت هکر می‌تواند کنترل سروری که پایگاه داده را میزبانی می‌کند به دست بگیرد. اثر این حمله به عواملی مانند اینکه آسیب پذیری در کجای کد رخ داده، آسیب پذیری چقدر کار هکر را آسان کرده و اینکه تا چه سطحی دسترسی به پایگاه داده را به دست گرفته، بستگی دارد. هدف اصلی این حملات، داده‌های ساکن در یک پایگاه داده است که از طریق Firewall محافظت می‌شوند.

به طور کلی به یک مهاجم اجازه می‌دهد تا داده‌هایی را که معمولاً قادر به بازیابی نیستند مشاهده کند. این موضوع ممکن است شامل داده های کاربران یا هر داده دیگری باشد که برنامه به آن دسترسی دارد.

در بسیاری موارد، یک مهاجم می‌تواند این داده‌ها را تغییر داده یا حذف کند و باعث ایجاد تغییرات مداوم در محتوا یا رفتار برنامه شود. همچنین یک مهاجم می‌تواند با حمله تزریق SQL، سرور اصلی یا سایر زیرساخت های back-end را به خطر بیاندازد یا یک حمله denial-of-service را انجام دهد.

## تأثیرات حمله تزریق SQL

حمله تزریق SQL می‌تواند منجر به دسترسی غیر مجاز به داده‌های حساس مانند گذرواژه‌‌ها، اطلاعات کارت اعتباری یا اطلاعات شخصی کاربر شود. بسیاری از موارد نقض داده‌های مشهور در سال‌های اخیر نتیجه حملات حمله تزریق SQL بوده و منجر به صدمه به اعتبار آن‌ها شده است.

در بعضی موارد، یک مهاجم می‌تواند یک پشتیبان مداوم را در سیستم‌های سازمان بدست آورد و منجر به سازش طولانی مدت شود که می‌تواند برای مدت طولانی بدون اینکه شناسایی شود کار خود را ادامه دهد.

### انواع تزریق SQL

تعداد زیادی از آسیب پذیری‌ها، حملات و تکنیک‌های حمله تزریق SQL وجود دارد که در شرایط مختلف ایجاد می‌شوند. برخی از نمونه‌های معمول تزریق SQL عبارتند از:

* بازیابی داده‌های پنهان: جایی که می‌توانید یک درخواست SQL را برای بازگشت نتایج اضافی اصلاح کنید.
* برهم زدن منطق برنامه:جایی که می‌توانید یک سوال را برای تغییر در منطق برنامه تغییر دهید.
* حمله به UNION:  در آن می‌توانید داده‌ها را از جداول پایگاه داده‌های مختلف بازیابی کنید.
* بررسی بانک اطلاعاتی: جایی که می‌توانید اطلاعات مربوط به نسخه و ساختار بانک اطلاعات را استخراج کنید.
* تزریق کور SQL: در آن نتایج جستجوی شما کنترل شده و در پاسخ برنامه وجود ندارد.

1. **بازیابی داده‌های پنهان**

یک برنامه خرید را در نظر بگیرید که محصولات را در دسته های مختلف نمایش می‌دهد. وقتی کاربر بر روی دسته هدایا کلیک کند، مرورگر، URL را درخواست می‌کند:

این امر باعث می‌شود تا یک نرم افزار SQL برای بازگرداندن جزئیات مربوط به محصولات از پایگاه داده، از یک SQL Query استفاده کند:

SELECT \* FROM products WHERE category = 'Gifts' AND released = 1

این پرس و جو (SQL query) از database اطلاعات زیر را درخواست می‌کند:

* all details (\*)
* from the products table
* where the category is Gifts
* and released is 1.

“released = 1” برای مخفی کردن محصولاتی که عرضه نمی‌شوند، استفاده می‌شود. برای محصولات منتشر نشده “released = 0” استفاده می‌شود.

این برنامه هیچگونه دفاعی را در برابر حملات تزریق SQL پیاده سازی نمی‌کند، بنابراین یک مهاجم می‌تواند حمله‌ای مانند:

“–‘https://insecure-website.com/products?category=Gifts”

نتیجه جستجوی SQL به صورت زیر می‌باشد:

SELECT \* FROM products WHERE category = 'Gifts'--' AND released = 1

نکته اصلی در اینجا این است که توالی دو خط “–” یک شاخص در SQL است و به این معنی است که بقیه پرس و جو به عنوان یک تفسیر تعبیر می‌شود.

این موضوع به طور تاثیر گذاری باقی مانده پرس و جو را از بین می‌برد، بنابراین دیگر “1=AND released” منتشر نمی‌شود. این بدان معنی است که همه محصولات از جمله محصولات غیرمجاز نمایش داده می‌شوند.

2.برهم زدن منطق برنامه

برنامه‌ای را در نظر بگیرید که به کاربران اجازه می‌دهد با یک نام کاربری و رمز عبور وارد شوند. اگر کاربر نام کاربری و رمز عبور را ارسال کند، برنامه با انجام SQL زیر، اعتبار نامه را بررسی می‌کند:

'SELECT \* FROM users WHERE username = 'wiener' AND password = 'bluecheese

اگر پرس و جو، جزئیات یک کاربر را بازگرداند، ورود به سیستم موفقیت آمیز خواهد بود. در غیر این صورت درخواست پذیرفته نمی‌شود.

در این بخش، یک مهاجم می‌تواند مانند هر کاربر و بدون رمز ورود به سادگی با استفاده از”SQL comment--” وارد سیستم شود.

بخش چک کردن رمز عبور را از قسمت “WHERE” انجام میدهد و آن را حذف می‌کند. به عنوان مثال، ارسال نام کاربری ” –‘administrator “و یک رمز عبور خالی به صورت خط زیر:

''=SELECT \* FROM users WHERE username = 'administrator'--' AND password

می‌تواند کاربر را که نام کاربری آن administrator است از دیتابیس دریافت کند و هکر خود را به جای کاربر ثبت کند.

**3.حمله به UNION**

در مواردی که نتایج یک پرس و جو SQL در پاسخ برنامه‌ها برگردانده شود، مهاجم می‌تواند از آسیب پذیری SQL Injection برای بازیابی داده‌ها از جداول دیگر در پایگاه داده استفاده کند.

این کار با استفاده از کلمه کلیدی UNION انجام می‌شود و به شما امکان می‌دهد “Select query” اضافی را نیز انجام دهید و نتایج را به پرس و جو اصلی اضافه کنید.

به عنوان مثال، اگر برنامه‌ای درخواست زیر را که حاوی ورودی کاربر “gifts” است، اجرا کند:

'SELECT name, description FROM products WHERE category = 'Gifts

سپس هکر می‌تواند ورودی زیر را ارسال کند:

--UNION SELECT username, password FROM users

این امر باعث می‌شود برنامه تمام نام‌های کاربری و کلمات عبور را به همراه نام و توضیحات محصولات بازگرداند.

4.**بررسی بانک اطلاعاتی**

پس از شناسایی اولیه آسیب پذیری SQL Injection، به طور کلی دستیابی به برخی اطلاعات در مورد خود پایگاه داده مفید است. این اطلاعات اغلب می‌تواند راه را برای بهره برداری بیشتر هموار کند.

می‌توانید جزئیات نسخه مربوط به پایگاه داده را پرس و جو کنید. نحوه انجام این کار بستگی به نوع بانک اطلاعاتی دارد، بنابراین می‌توانید نوع پایگاه داده را از هر تکنیک دیگری بدست آورید. به عنوان مثال، در Oracle می توانید این کار را انجام دهید:

SELECT \* FROM v$version

همچنین می‌توانید تعیین کنید که جداول بانک اطلاعاتی وجود دارد و کدام ستون ها را شامل می‌شود. به عنوان مثال، در اکثر بانک‌های اطلاعاتی می‌توانید عبارت زیر را برای لیست کردن جدول‌ها اجرا کنید:

SELECT \* FROM information\_schema.tables

**5.تزریق کور SQL‌**

بسیاری از موارد تزریق SQL آسیب پذیری کور است. این بدان معناست که برنامه نتایج جستجوی SQL یا جزئیات هرگونه خطای پایگاه داده را در پاسخ های خود باز نمی‌گرداند.

از آسیب‌پذیری‌های کور هنوز هم می‌توان برای دسترسی به داده‌های غیر مجاز استفاده کرد، اما تکنیک‌های موجود عموماً پیچیده تر و دشوار هستند.

بسته به ماهیت آسیب‌پذیری و پایگاه داده، از تکنیک‌های زیر می‌توان برای سوءاستفاده از آسیب‌پذیری‌های تزریق کور SQL استفاده کرد:

* شما می‌توانید منطق پرس و جو را تغییر دهید تا بسته به حقیقت یک شرط واحد، اختلاف قابل تشخیص در پاسخ برنامه ایجاد شود. این ممکن است شامل تزریق یک شرایط جدید به برخی از منطق بولی و یا ایجاد مشروط مانند خطای تقسیم بر صفر باشد.
* شما می‌توانید به طور مشروط یک تأخیر زمانی در پردازش پرس و جو ایجاد کنید که به شما این امکان را می‌دهد تا واقعیت شرط را بر اساس زمانی که برنامه پاسخ می‌دهد استنباط کنید.
* با استفاده از تکنیک‌های OAST می‌توانید تعامل خارج از باند شبکه را آغاز کنید. این روش بسیار قدرتمند است و در شرایطی کار می‌کند که تکنیک‌های دیگر این کار را انجام ندهند. شما می‌توانید مستقیماً داده‌ها را از طریق کانال خارج از باند، مجزا کنید، به عنوان مثال با قرار دادن داده‌ها در جستجوی DNS برای دامنه‌ای که شما کنترل می‌کنید.

**نحوه تشخیص آسیب‌پذیری‌های تزریق SQL**

اکثر آسیب‌پذیری‌های تزریق SQL را می‌توان با استفاده از اسکنر آسیب پذیری وب “Burp Suite” یافت.

SQL Injection می‌تواند به صورت دستی و با استفاده از یک مجموعه سیستماتیک از تست در برابر هر نقطه ورود به برنامه تشخیص داده شود. به طور معمول شامل موارد زیر است:

* ارسال شخصیت نقل قول واحد و به دنبال خطا یا ناهنجاری‌های دیگر.
* ارسال برخی از نحوه‌های خاص SQL که به مقدار پایه (اصلی) از نقطه ورود و به یک مقدار دیگر ارزشیابی می‌کنند
* ارسال شرایط بولی مانند 1 = 1 و 1 = 2 و جستجوی تفاوت در پاسخ برنامه.
* ارسال بار‌های با هدف ایجاد تأخیر در هنگام اجرای درخواست در SQL و به دنبال تفاوت در زمان لازم برای پاسخگویی هستید.
* ارسال بار‌های OAST با هدف ایجاد تعامل شبکه خارج از باند هنگام اجرای یک پرس و جو SQL و نظارت بر هرگونه تعامل حاصل از آن، طراحی شده است.

**تزریق SQL در قسمت های مختلف پرس و جو**

بیشتر آسیب‌پذیری‌های تزریق SQL در بند “WHERE” از “SELECT Query” بوجود می‌آیند. این نوع تزریق SQL معمولاً توسط آزمایش کنندگان باتجربه قابل درک است.

آسیب‌پذیری‌های تزریق SQL می‌توانند در هر مکانی از پرس و جو  نمایش داده شود. رایج‌ترین مکان‌ها که در آن حمله تزریق SQL بوجود می آید یه شرح زیر می‌باشد:

* در بیانیه های UPDATE، در مقادیر به روز شده یا بند WHERE
* در عبارت INSERT، درون مقادیر درج شده
* در عبارت SELECT، در جدول یا نام ستون آمده است
* در بیانیه های SELEC، در بند ORDER BY

عوامل خاص در پایگاه داده

برخی از ویژگی‌های اصلی زبان SQL به همان روش در سراسر سیستم عامل‌های پایگاه داده محبوب اجرا می‌شود، بنابراین بسیاری از روش‌های شناسایی و بهره برداری از آسیب پذیری‌های تزریق SQL به طور یکسان در انواع مختلف بانک اطلاعاتی کار می‌کنند.

این بدان معنی است که تفاوت‌های زیادی بین بانک‌های اطلاعاتی مشترک نیز وجود دارد. و برخی از تکنیک‌های تشخیص و بهره برداری از حمله تزریق SQL بر روی سیستم عامل‌های مختلف کار می‌کنند. مثلا:

* ترکیب اتصال رشته‌ای
* نظرات
* پرسشهای دسته ای
* API های ویژه بستر های نرم افزاری.
* پیام خطا

**روش‌های جلوگیری از حمله تزریق SQL**

1. Parameterization queries (پارامتری کردن پرس و جوها): برای جلوگیری از به وجود آمدن باگ، برنامه‌نویس باید از Queryهای پارامتر شده استفاده کند. به آن معنی که در آن Queryها یکبار بدون داشتن متغیر به سرور پایگاه داده ارسال شوند و در مرحله بعد به صورت جداگانه  ارسال شوند. این موجب افزایش امنیت و سرعت کار با پایگاه داده می‌شود. زیرا متغیرهای ما  به صورت مستقیم در دستور SQL که می‌فرستیم قرار نگرفته و به صورت جداگانه ارسال می‌شوند. بنابراین هکر نمی‌تواند به Query ما دستوری اضافه کند.

2. Stored Procedure: تا حد امکان از روش‌های ذخیره شده استفاده کنیم. یک قسمتی از بانک اطلاعاتی SQL Server است که می‌توان در آنها از دستورات Select، Insert Delete و ... استفاده کرد و با استفاده از پارامترهای ورودی یا خروجی می‌توان اطلاعاتی از آنها دریافت کرد.

3. Escaping User Input: تمام اطلاعاتی که توسط کاربر وارد می‌شود را Scape کنیم، یعنی به طور مثال اگر کاربر <script> را وارد کرد، این کلمه تبدیل به &it;script&gt;  شود که به صورت string باشد.

4. Least privilege: به معنی کاهش اثر تزریق کد از طریق اعطای امتیاز کمتر به کاربران پایگاه داده است. به عنوان مثال به کاربران حق دسترسی read، view و ... بدهیم.

5. White List Input Validation: به معنای تعریف دقیق ورودی‌های معتبر، اعتبارسنجی آنها قبل از فرستادن به Query، ایجاد ساختار اعتبارسنجی برای ورودی‌هایی که فرمت خاصی دارند، مانند ایمیل و تاریخ، استفاده از توابع اعتبارسنجی ابزار ESAPI مانند getValidData()، getValidInput، getValidFileName و...

هنگام طراحی سایت بهتر است اطلاعات حیاتی مانند رمز عبور، شماره کارت اعتباری و... به صورت متن ساده یا Plain Text ذخیره نشوند. به صورت رمزنگاری در پایگاه داده ذخیره شوند. اگر به هر طریقی وب سایت مورد حمله قرار گرفت، هکر قادر به دستیابی به آن‌ها نباشد. برای این منظور بهتر است از پسوردهای قوی استفاده شود، آسیب‌پذیری‌ها بررسی شود، سایت‌ها با کدها و برنامه‌های جدید بروز رسانی شوند.

لطفا جهت مشاوره در [انتخاب آنتی ویروس مناسب](https://softyab.com/product-category/security/antivirus/) خود با این شماره 09224971053 تماس بگیرید.

 مارو در شبکه های [اجتماعی](https://www.instagram.com/softyab) خودتان به اشتراک بگذارید.

What is SQL injection (SQLi)?

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

What is the impact of a successful SQL injection attack?

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

SQL injection examples

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

* [Retrieving hidden data](https://portswigger.net/web-security/sql-injection#retrieving-hidden-data), where you can modify an SQL query to return additional results.
* [Subverting application logic](https://portswigger.net/web-security/sql-injection#subverting-application-logic), where you can change a query to interfere with the application's logic.
* [UNION attacks](https://portswigger.net/web-security/sql-injection/union-attacks), where you can retrieve data from different database tables.
* [Examining the database](https://portswigger.net/web-security/sql-injection/examining-the-database), where you can extract information about the version and structure of the database.
* [Blind SQL injection](https://portswigger.net/web-security/sql-injection/blind), where the results of a query you control are not returned in the application's responses.

Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

https://insecure-website.com/products?category=Gifts

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

SELECT \* FROM products WHERE category = 'Gifts' AND released = 1

This SQL query asks the database to return:

* all details (\*)
* from the products table
* where the category is Gifts
* and released is 1.

The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably released = 0.

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

https://insecure-website.com/products?category=Gifts'--

This results in the SQL query:

SELECT \* FROM products WHERE category = 'Gifts'--' AND released = 1

The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about:

https://insecure-website.com/products?category=Gifts'+OR+1=1--

This results in the SQL query:

SELECT \* FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

**LAB**[SQL injection vulnerability in WHERE clause allowing retrieval of hidden data](https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data)

Subverting application logic

Consider an application that lets users log in with a username and password. If a user submits the username wiener and the password bluecheese, the application checks the credentials by performing the following SQL query:

SELECT \* FROM users WHERE username = 'wiener' AND password = 'bluecheese'

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator'-- and a blank password results in the following query:

SELECT \* FROM users WHERE username = 'administrator'--' AND password = ''

This query returns the user whose username is administrator and successfully logs the attacker in as that user.

**LAB**[SQL injection vulnerability allowing login bypass](https://portswigger.net/web-security/sql-injection/lab-login-bypass)

Retrieving data from other database tables

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. This is done using the UNION keyword, which lets you execute an additional SELECT query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

SELECT name, description FROM products WHERE category = 'Gifts'

then an attacker can submit the input:

' UNION SELECT username, password FROM users--

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

**Read more**

[SQL injection UNION attacks](https://portswigger.net/web-security/sql-injection/union-attacks)

Examining the database

Following initial identification of an SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

SELECT \* FROM v$version

You can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

SELECT \* FROM information\_schema.tables

**Read more**

[Examining the database in SQL injection attacks](https://portswigger.net/web-security/sql-injection/examining-the-database)[SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet)

Blind SQL injection vulnerabilities

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

* You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.
* You can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.
* You can trigger an out-of-band network interaction, using [OAST](https://portswigger.net/burp/application-security-testing/oast) techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS lookup for a domain that you control.

How to detect SQL injection vulnerabilities

The majority of SQL injection vulnerabilities can be found quickly and reliably using Burp Suite's [web vulnerability scanner](https://portswigger.net/burp/vulnerability-scanner).

SQL injection can be detected manually by using a systematic set of tests against every entry point in the application. This typically involves:

* Submitting the single quote character ' and looking for errors or other anomalies.
* Submitting some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses.
* Submitting Boolean conditions such as OR 1=1 and OR 1=2, and looking for differences in the application's responses.
* Submitting payloads designed to trigger time delays when executed within an SQL query, and looking for differences in the time taken to respond.
* Submitting OAST payloads designed to trigger an out-of-band network interaction when executed within an SQL query, and monitoring for any resulting interactions.

SQL injection in different parts of the query

Most SQL injection vulnerabilities arise within the WHERE clause of a SELECT query. This type of SQL injection is generally well-understood by experienced testers.

But SQL injection vulnerabilities can in principle occur at any location within the query, and within different query types. The most common other locations where SQL injection arises are:

* In UPDATE statements, within the updated values or the WHERE clause.
* In INSERT statements, within the inserted values.
* In SELECT statements, within the table or column name.
* In SELECT statements, within the ORDER BY clause.

Second-order SQL injection

First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.

In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.

Second-order SQL injection often arises in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

Database-specific factors

Some core features of the SQL language are implemented in the same way across popular database platforms, and so many ways of detecting and exploiting SQL injection vulnerabilities work identically on different types of database.

However, there are also many differences between common databases. These mean that some techniques for detecting and exploiting SQL injection work differently on different platforms. For example:

* Syntax for string concatenation.
* Comments.
* Batched (or stacked) queries.
* Platform-specific APIs.
* Error messages.

How to prevent SQL injection

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

String query = "SELECT \* FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();

Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It is all too easy to make mistakes about the possible origin of data, or for changes in other code to violate assumptions about what data is tainted.