Every few years, the Open Web Application Security Project (OWASP) ranks the most critical web application security risks. Since the first report, injection risks have always been on top. Among all injection types, **SQL injection** is one of the most common attack vectors, and arguably the most dangerous. As Python is one of the most popular programming languages in the world, knowing how to protect against Python SQL injection is critical.

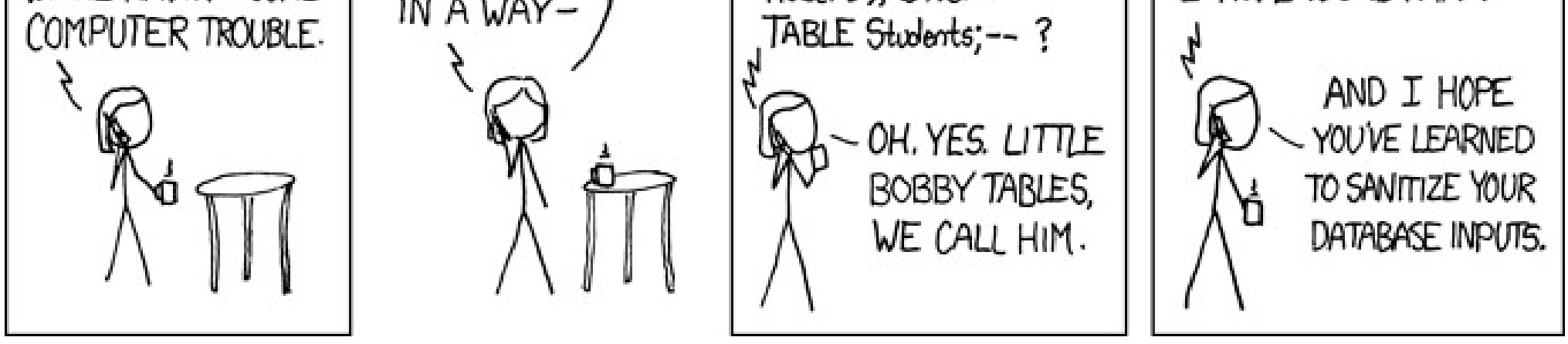
In this tutorial, you're going to learn:

- What **Python SQL injection** is and how to prevent it
- How to **compose queries** with both literals and identifiers as parameters
- How to **safely execute queries** in a database

This tutorial is suited for **users of all database engines**. The examples here use PostgreSQL, but the results can be reproduced in other database management systems (such as SQLite, MySQL, Microsoft SQL Server, Oracle, and so on).

Understanding Python SQL Injection

SQL Injection attacks are such a common security vulnerability that the legendary *xkcd* webcomic devoted a comic to it:



"Exploits of a Mom" (Image: xkcd)

Generating and executing SQL queries is a common task. However, companies around the world often make horrible mistakes when it comes to composing SQL statements. While the ORM layer usually composes SQL queries, sometimes you have to write your own.

When you use Python to execute these queries directly into a database, there's a chance you could make mistakes that might compromise your system. In this tutorial, you'll learn how to successfully implement functions that compose dynamic SQL queries *without* putting your system at risk for Python SQL injection.

Setting Up a Database

To get started, you're going to set up a fresh PostgreSQL database and populate it with data. Throughout the tutorial, you'll use this database to witness firsthand how Python SQL injection works.

Creating a Database

First, open your shell and create a new PostgreSQL database owned by the user `postgres`:

Shell

```
$ createdb -O postgres psycopgtest
```

Here you used the

user `postgres`. You also specified the name of the database, which is `psycopgtest`.

> **Note:** `postgres` is a **special user**, which you would normally reserve for administrative tasks, but for this tutorial, it's fine to use `postgres`. In a real system, however, you should create a separate user to be the owner of the database.

Your new database is ready to go! You can connect to it using `psql`:

Shell

```
$ psql -U postgres -d psycopgtest
psql (11.2, server 10.5)
Type "help" for help.
```

You're now connected to the database `psycopgtest` as the user `postgres`. This user is also the database owner, so you'll have read permissions on every table in the database.

Creating a Table With Data

Next, you need to create a table with some user information and add data to it:

PostgreSQL Console

```
psycopgtest=# CREATE TABLE users (
    username varchar(30),
    admin boolean
);
CREATE TABLE

psycopgtest=# INSERT INTO users
    (username, admin)
VALUES
    ('ran', true),
    ('haki', false);
INSERT 0 2

psycopgtest=# SELECT * FROM users;
 username | admin
----------+-------
 ran      | t
 haki     | f
(2 rows)
```

The table has two columns: `username` and `admin`. The `admin` column indicates whether or not a user has administrative privileges. Your goal is to target the `admin` field and try to abuse it.

Setting Up a Python Virtual Environment

Now that you have a database, it's time to set up your Python environment. For step-by-step instructions on how to do this, check out Python Virtual Environments: A Primer.

Create your virtual environment in a new directory:

Shell

```
(~/src) $ mkdir psycopgtest
(~/src) $ cd psycopgtest
(~/src/psycopgtest) $ python3 -m venv venv
```

After you run this command, a new directory called `venv` will be created. This directory will store

Connecting to the Database

To connect to a database in Python, you need a **database adapter**. Most database adapters follow version 2.0 of the Python Database API Specification PEP 249. Every major database engine has a leading adapter:

Database	Adapter
PostgreSQL	Psycopg
SQLite	sqlite3
Oracle	cx_oracle
MySql	MySQLdb

To connect to a PostgreSQL database, you'll need to install Psycopg, which is the most popular adapter for PostgreSQL in Python. Django ORM uses it by default, and it's also supported by SQLAlchemy.

In your terminal, activate the virtual environment and use `pip` to install `psycopg`:

Shell

```
(~/src/psycopgtest) $ source venv/bin/activate
(~/src/psycopgtest) $ python -m pip install psycopg2>=2.8.0
Collecting psycopg2
  Using cached https://....
  psycopg2-2.8.2.tar.gz
Installing collected packages: psycopg2
  Running setup.py install for psycopg2 ... done
Successfully installed psycopg2-2.8.2
```

Now you're ready to create a connection to your database. Here's the start of your Python script:

Python

```
import psycopg2

connection = psycopg2.connect(
    host="localhost",
    database="psycopgtest",
    user="postgres",
    password=None,
)
connection.set_session(autocommit=True)
```

You used `psycopg2.connect()` to create the connection. This function accepts the following arguments:

- **`host`** is the IP address or the DNS of the server where your database is located. In this case, the host is your local machine, or `localhost`.
- **`database`** is the name of the database to connect to. You want to connect to the database you created earlier, `psycopgtest`.
- **`user`** is a user with permissions for the database. In this case, you want to connect to the database as the owner, so you pass the user `postgres`.
- **`password`** is the password for whoever you specified in `user`. In most development environments, users can connect to the local database without a password.

After setting up the connection, you configured the session with `autocommit=True`. Activating `autocommit` means you won't have to manually manage transactions by issuing a `commit` or `rollback`. This is the default behavior in most ORMs. You use this behavior here as well so that you can focus on composing SQL queries instead of managing transactions.

Note: Django users can get the instance of the connection used by the ORM from `django.db.connection`:

Python

```
from django.db import connection
```

Executing a Query

Now that you have a connection to the database, you're ready to execute a query:

Python >>>

```
>>> with connection.cursor() as cursor:
...     cursor.execute('SELECT COUNT(*) FROM users')
...     result = cursor.fetchone()
... print(result)
(2,)
```

You used the `connection` object to create a `cursor`. Just like a file in Python, `cursor` is implemented as a context manager. When you create the context, a `cursor` is opened for you to use to send commands to the database. When the context exits, the `cursor` closes and you can no longer use it.

> **Note:** To learn more about context managers, check out Python Context Managers and the "with" Statement.

While inside the context, you used `cursor` to execute a query and fetch the results. In this case, you issued a query to count the rows in the `users` table. To fetch the result from the query, you executed `cursor.fetchone()` and received a tuple. Since the query can only return one result, you used `fetchone()`. If the query were to return more than one result, then you'd need to either iterate over `cursor` or use one of the other `fetch*` methods.

Using Query Parameters in SQL

In the previous section, you created a database, established a connection to it, and executed a query. The query you used was **static**. In other words, it had **no parameters**. Now you'll start to use parameters in your queries.

First, you're going to implement a function that checks whether or not a user is an admin. `is_admin()` accepts a username and returns that user's admin status:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = '%s'
        """ % username)
        result = cursor.fetchone()
    admin, = result
    return admin
```

This function executes a query to fetch the value of the `admin` column for a given username. You used `fetchone()` to return a tuple with a single result. Then, you unpacked this tuple into the variable `admin`. To test your function, check some usernames:

Python >>>

```
>>> is_admin('haki')
False
>>> is_admin('ran')
True
```

So far so good. The function returned the expected result for both users. But what about non-existing user? Take a look at this Python traceback:

Python >>>

```
>>> is_admin('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in is_admin
TypeError: cannot unpack non-iterable NoneType object
```

When the user does not exist, a `TypeError` is raised. This is because `.fetchone()` returns `None` when no results are found, and unpacking `None` raises a `TypeError`. The only place you can unpack a tuple is where you populate `admin` from `result`.

To handle non-existing users, create a special case for when `result` is `None`:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = '%s'
        """ % username)
        result = cursor.fetchone()

    if result is None:
        # User does not exist
        return False

    admin, = result
    return admin
```

Here, you've added a special case for handling `None`. If `username` does not exist, then the function should return `False`. Once again, test the function on some users:

Python >>>

```
>>> is_admin('haki')
False
>>> is_admin('ran')
True
>>> is_admin('foo')
False
```

Great! The function can now handle non-existing usernames as well.

Exploiting Query Parameters With Python SQL Injection

In the previous example, you used string interpolation to generate a query. Then, you executed the query and sent the resulting string directly to the database. However, there's something you may have overlooked during this process.

Think back to the `username` argument you passed to `is_admin()`. What exactly does this variable represent? You might assume that `username` is just a string that represents an actual user's name. As you're about to see, though, an intruder can easily exploit this kind of oversight and cause major harm by performing Python SQL injection.

Try to check if the following user is an admin or not:

Python >>>

```
>>> is_admin("'; select true; --")
True
```

Wait… What just happened?

Let's take another look at the implementation. Print out the actual query being executed in the database:

Python >>>

```
>>> print("select admin from users where username = '%s'" % "'; select true; -
select admin from users where username = ''; select true; --'
```

The resulting text contains three statements. To understand exactly how Python SQL injection works, you need to inspect each part individually. The first statement is as follows:

SQL

```
select admin from users where username = '';
```

This is your intended query. The semicolon (`;`) terminates the query, so the result of this query does not matter. Next up is the second statement:

SQL

```
select true;
```

This statement was constructed by the intruder. It's designed to always return `True`.

Lastly, you see this short bit of code:

SQL

```
--'
```

This snippet defuses anything that comes after it. The intruder added the comment symbol (`--`) to turn everything you might have put after the last placeholder into a comment.

When you execute the function with this argument, *it will always return `True`*. If, for example, you use this function in your login page, an intruder could log in with the username `'; select true; --`, and they'll be granted access.

If you think this is bad, it could get worse! Intruders with knowledge of your table structure can use Python SQL injection to cause permanent damage. For example, the intruder can inject an update statement to alter the information in the database:

Python >>>

```
>>> is_admin('haki')
False
>>> is_admin("'; update users set admin = 'true' where username = 'haki'; sele
True
>>> is_admin('haki')
True
```

Let's break it dowr

SQL

```
';
```

This snippet terminates the query, just like in the previous injection. The next statement is as follows:

SQL

```
update users set admin = 'true' where username = 'haki';
```

This section updates `admin` to `true` for user `haki`.

Finally, there's this code snippet:

SQL

```
select true; --
```

As in the previous example, this piece returns `true` and comments out everything that follows it.

Why is this worse? Well, if the intruder manages to execute the function with this input, then user `haki` will become an admin:

PostgreSQL Console

```
psycopgtest=# select * from users;
 username | admin
----------+-------
 ran      | t
 haki     | t
(2 rows)
```

The intruder no longer has to use the hack. They can just log in with the username `haki`. (If the intruder *really* wanted to cause harm, then they could even issue a `DROP DATABASE` command.)

Before you forget, restore `haki` back to its original state:

PostgreSQL Console

```
psycopgtest=# update users set admin = false where username = 'haki';
UPDATE 1
```

So, why is this happening? Well, what do you know about the `username` argument? You know it should be a string representing the username, but you don't actually check or enforce this assertion. This can be dangerous! It's exactly what attackers are looking for when they try to hack your system.

Crafting Safe Query Parameters

In the previous section, you saw how an intruder can exploit your system and gain admin permissions by using a carefully crafted string. The issue was that you allowed the value passed from the client to be executed directly to the database, without performing any sort of check or validation. SQL injections rely on this type of vulnerability.

Any time user input is used in a database query, there's a possible vulnerability for SQL injection. The key to preventing Python SQL injection is to make sure the value is being used as the developer intended. In the previous example, you intended for `username` to be used as a string. In reality, it was used as a raw SQL statement.

To make sure values are used as they're intended, you need to **escape** the value. For example, to prevent intruders from injecting raw SQL in the place of a string argument, you can escape quotation marks:

Python >>>

```
>>> # BAD EXAMPLE. DON'T DO THIS!
>>> username = username.replace("'", "''")
```

This is just one example. There are a lot of special characters and scenarios to think about when trying

adapters, come with built-in tools for preventing Python SQL injection by using **query parameters**. These are used instead of plain string interpolation to compose a query with parameters.

> **Note:** Different adapters, databases, and programming languages refer to query parameters by different names. Common names include **bind variables**, **replacement variables**, and **substitution variables**.

Now that you have a better understanding of the vulnerability, you're ready to rewrite the function using query parameters instead of string interpolation:

Python

```
1  def is_admin(username: str) -> bool:
2      with connection.cursor() as cursor:
3          cursor.execute("""
4              SELECT
5                  admin
6              FROM
7                  users
8              WHERE
9                  username = %(username)s
10         """, {
11             'username': username
12         })
13         result = cursor.fetchone()
14
15     if result is None:
16         # User does not exist
17         return False
18
19     admin, = result
20     return admin
```

Here's what's different in this example:

- **In line 9,** you used a named parameter `username` to indicate where the username should go. Notice how the parameter `username` is no longer surrounded by single quotation marks.

- **In line 11,** yo

`cursor.execute()`. The connection will use the type and value of `username` when executing the query in the database.

To test this function, try some valid and invalid values, including the dangerous string from before:

Python >>>

```
>>> is_admin('haki')
False
>>> is_admin('ran')
True
>>> is_admin('foo')
False
>>> is_admin("'; select true; --")
False
```

Amazing! The function returned the expected result for all values. What's more, the dangerous string no longer works. To understand why, you can inspect the query generated by `execute()`:

Python >>>

```
>>> with connection.cursor() as cursor:
...     cursor.execute("""
...         SELECT
...             admin
...         FROM
...             users
...         WHERE
...             username = %(username)s
...     """, {
...         'username': "'; select true; --"
...     })
...     print(cursor.query.decode('utf-8'))
SELECT
    admin
FROM
    users
WHERE
    username = '''; select true; --'
```

The connection tre

that might terminate the string and introduce Python SQL injection.

Passing Safe Query Parameters

Database adapters usually offer several ways to pass query parameters. **Named placeholders** are usually the best for readability, but some implementations might benefit from using other options.

Let's take a quick look at some of the right and wrong ways to use query parameters. The following code block shows the types of queries you'll want to avoid:

Python

```
# BAD EXAMPLES. DON'T DO THIS!
cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
cursor.execute("SELECT admin FROM users WHERE username = '{}'".format(username
cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

Each of these statements passes `username` from the client directly to the database, without performing any sort of check or validation. This sort of code is ripe for inviting Python SQL injection.

In contrast, these types of queries should be safe for you to execute:

Python

```
# SAFE EXAMPLES. DO THIS!
cursor.execute("SELECT admin FROM users WHERE username = %s'", (username, ));
cursor.execute("SELECT admin FROM users WHERE username = %(username)s", {'user
```

In these statements, `username` is passed as a named parameter. Now, the database will use the specified type and value of `username` when executing the query, offering protection from Python SQL injection.

Improve Your Python

Using SQL Composition

So far you've used parameters for literals. **Literals** are values such as numbers, strings, and dates. But what if you have a use case that requires composing a different query—one where the parameter is something else, like a table or column name?

Inspired by the previous example, let's implement a function that accepts the name of a table and returns the number of rows in that table:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def count_rows(table_name: str) -> int:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                count(*)
            FROM
                %(table_name)s
        """, {
            'table_name': table_name,
        })
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

Try to execute the function on your users table:

Python >>>

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in count_rows
psycopg2.errors.SyntaxError: syntax error at or near "'users'"
LINE 5:                 'users'
                        ^
```

The command failed to generate the SQL. As you've seen already, the database adapter treats the variable as a string or a literal. A table name, however, is not a plain string. This is

You already know it's not safe to use string interpolation to compose SQL. Luckily, Psycopg provides a module called `psycopg.sql` to help you safely compose SQL queries. Let's rewrite the function using `psycopg.sql.SQL()`:

Python

```
from psycopg2 import sql

def count_rows(table_name: str) -> int:
    with connection.cursor() as cursor:
        stmt = sql.SQL("""
            SELECT
                count(*)
            FROM
                {table_name}
        """).format(
            table_name = sql.Identifier(table_name),
        )
        cursor.execute(stmt)
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

There are two differences in this implementation. First, you used `sql.SQL()` to compose the query. Then, you used `sql.Identifier()` to annotate the argument value `table_name`. (An **identifier** is a column or table name.)

Note: Users of the popular package `django-debug-toolbar` might get an error in the SQL panel for queries composed with `psycopg.sql.SQL()`. A fix is expected for release in version 2.0.

Now, try executing the function on the `users` table:

Python >>>

```
>>> count_rows('users')
2
```

Great! Next, let's see what happens when the table does not exist:

Python >>>

```
>>> count_rows('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in count_rows
psycopg2.errors.UndefinedTable: relation "foo" does not exist
LINE 5:                     "foo"
                            ^
```

The function throws the `UndefinedTable` exception. In the following steps, you'll use this exception as an indication that your function is safe from a Python SQL injection attack.

> **Note:** The exception `UndefinedTable` was added in psycopg2 version 2.8. If you're working with an earlier version of Psycopg, then you'll get a different exception.

To put it all together, add an option to count rows in the table up to a certain limit. This feature might be useful for very large tables. To implement this, add a `LIMIT` clause to the query, along with query parameters for the limit's value:

Python

```
from psycopg2 import sql

def count_rows(table_name: str, limit: int) -> int:
    with connection.cursor() as cursor:
        stmt = sql.SQL("""
            SELECT
                COUNT(*)
            FROM (
                SELECT
                    1
                FROM
                    {table_name}
                LIMIT
                    {limit}
            ) AS limit_query
        """).format(
            table_name = sql.Identifier(table_name),
            limit = sql.Literal(limit),
        )
        cursor.execute(stmt)
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

In this code block, you annotated `limit` using `sql.Literal()`. As in the previous example, `psycopg` will bind all query parameters as literals when using the simple approach. However, when using `sql.SQL()`, you need to explicitly annotate each parameter using either `sql.Identifier()` or `sql.Literal()`.

> **Note:** Unfortunately, the Python API specification does not address the binding of identifiers, only literals. Psycopg is the only popular adapter that added the ability to safely compose SQL with both literals and identifiers. This fact makes it even more important to pay close attention when binding identifiers.

Execute the function to make sure that it works:

Python >>>

```
>>> count_rows('users', 1)
1
>>> count_rows('users', 10)
2
```

Now that you see the function is working, make sure it's also safe:

Python >>>

```
>>> count_rows("(select 1) as foo; update users set admin = true where name =
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 18, in count_rows
psycopg2.errors.UndefinedTable: relation "(select 1) as foo; update users set
LINE 8:                         "(select 1) as foo; update users set adm...
                                ^
```

This traceback shows that `psycopg` escaped the value, and the database treated it as a table name. Since a table with this name doesn't exist, an `UndefinedTable` exception was raised and you were not hacked!

Conclusion

You've successfully implemented a function that composes dynamic SQL *without* putting your system at risk for Python SQL injection! You've used both literals and identifiers in your query without compromising security.

You've learned:

- What **Python SQL injection** is and how it can be exploited
- How to **prevent Python SQL injection** using query parameters
- How to **safely compose SQL statements** that use literals and identifiers as parameters