Security Copilot for GitHub

Bringing real-time detection, learning, and remediation into your pull requests.

symbiotic
SECURITY

Summary

- >_ Our ambition
- >_ Our team
- >_ Our project

Code security should not be

- punitive
- a productivity blocker
- a ticket generator
- opaque
- after the fact
- one-size-fits-all

Our ambition

Create not only an AI reviewer but an AI companion that helps developers grow their security skill in their flow, in the tool they use every day

Symbiotic Security at a glance

Founded in 2024

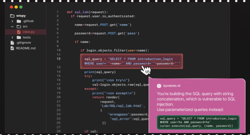
$ 3M raised

LERER HIPPEAU

axeleo capital

FACTORIAL CAPITAL

Developer-first
AI Security Platform



```
def sql_lab(request):
  if request.user.is_authenticated:

    name=request.POST.get('name')

    password=request.POST.get('pass')

    if name:

      if login.objects.filter(user=name):

        sql_query = 'SELECT * FROM introduction_login
        WHERE user='"+name+"'AND password='"+password+"'"

        print(sql_query)
        try:
            print("\nin try\n")
            val=login.objects.raw(sql_que
        except:
            print("\nin except\n")
            return render(
                request,
                'Lab/SQL/sql_lab.html',
                {
                    "wrongpass":password,
                     "sql_error":sql_quer
                 })

        if val:
```

Symbiotic AI

You're building the SQL query with string concatenation, which is vulnerable to SQL injection.
Use parameterized queries instead:

```
sql_query = "SELECT * FROM introduction_login
WHERE user=%s AND password=%s"
cursor.execute(sql_query, (name, password))
```

Our mission

Empower developers with AI-driven tools to instantly detect and fix vulnerabilities as they code - enhancing their skills in real time and making secure development a natural, frictionless experience.

In-IDE

>_ Instant detection

>_ Real-time & contextual AI remediation

>_ Just-in-Time AI-based learning experience

theodo.

MERCURY

trustpair

Heartbeat

upcell

16.86
Avoided vulns per dev per month

43.42%
Fewer issues after 1 month of JiT learning

$3,000+
Direct ROI per dev per month

Our team

Abir KHALLADI

↳ Full Stack Developer

Edouard VIOT

↳ CTO

Minh Thang Marc VU

↳ Lead Software Developer

Alexis COLONNA

↳ Product Manager

Anthony BONDU

↳ Cybersecurity Intern

Salah-Eddine ALABOUCH

↳ AI Engineer

symbiotic
SECURITY

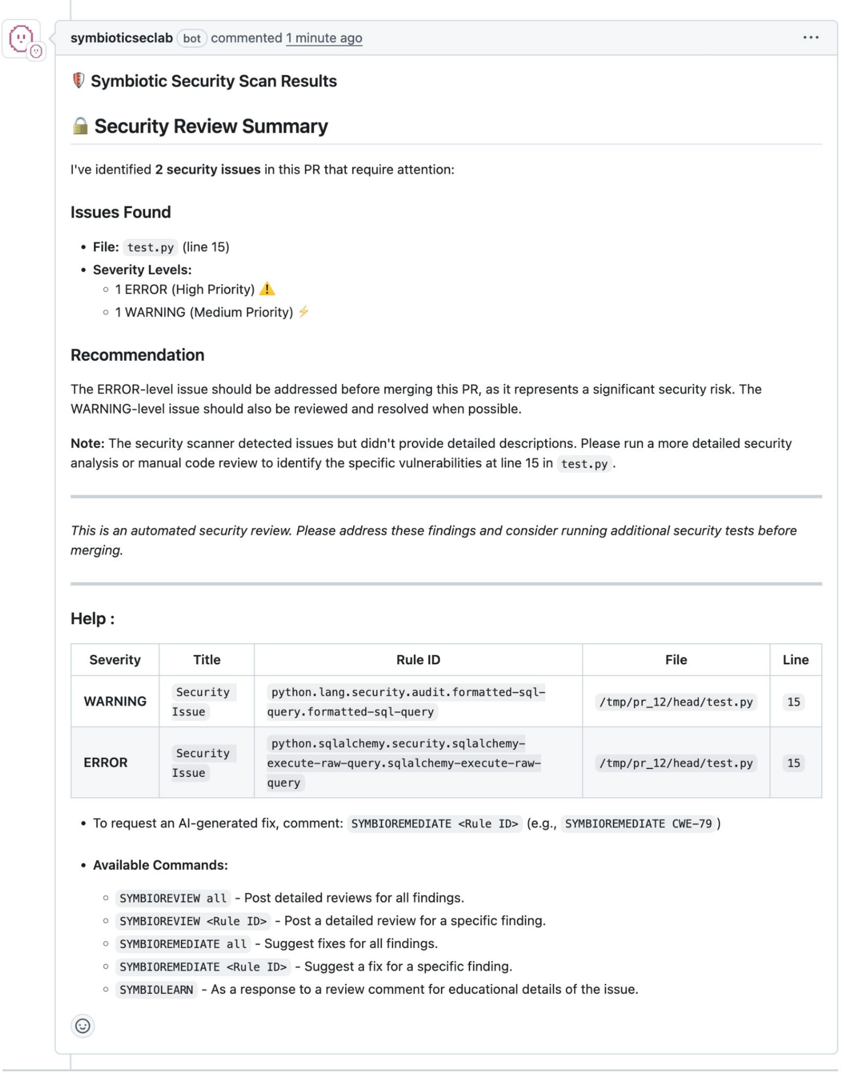
Developed with Blackbox.ai Agent

Our project

Using Claude Sonnet 4 via Blackbox.ai API

An automated security summary

>_ A security overview...
>_ ...automatically created
>_ ...each time a PR is opened



🛡️ Symbiotic Security Scan Results

🔒 Security Review Summary

I've identified **2 security issues** in this PR that require attention:

Issues Found

- **File:** `test.py` (line 15)
- **Severity Levels:**
  - 1 ERROR (High Priority) ⚠️
  - 1 WARNING (Medium Priority) ⚡

Recommendation

The ERROR-level issue should be addressed before merging this PR, as it represents a significant security risk. The WARNING-level issue should also be reviewed and resolved when possible.

Note: The security scanner detected issues but didn't provide detailed descriptions. Please run a more detailed security analysis or manual code review to identify the specific vulnerabilities at line 15 in `test.py`.

This is an automated security review. Please address these findings and consider running additional security tests before merging.

Help :

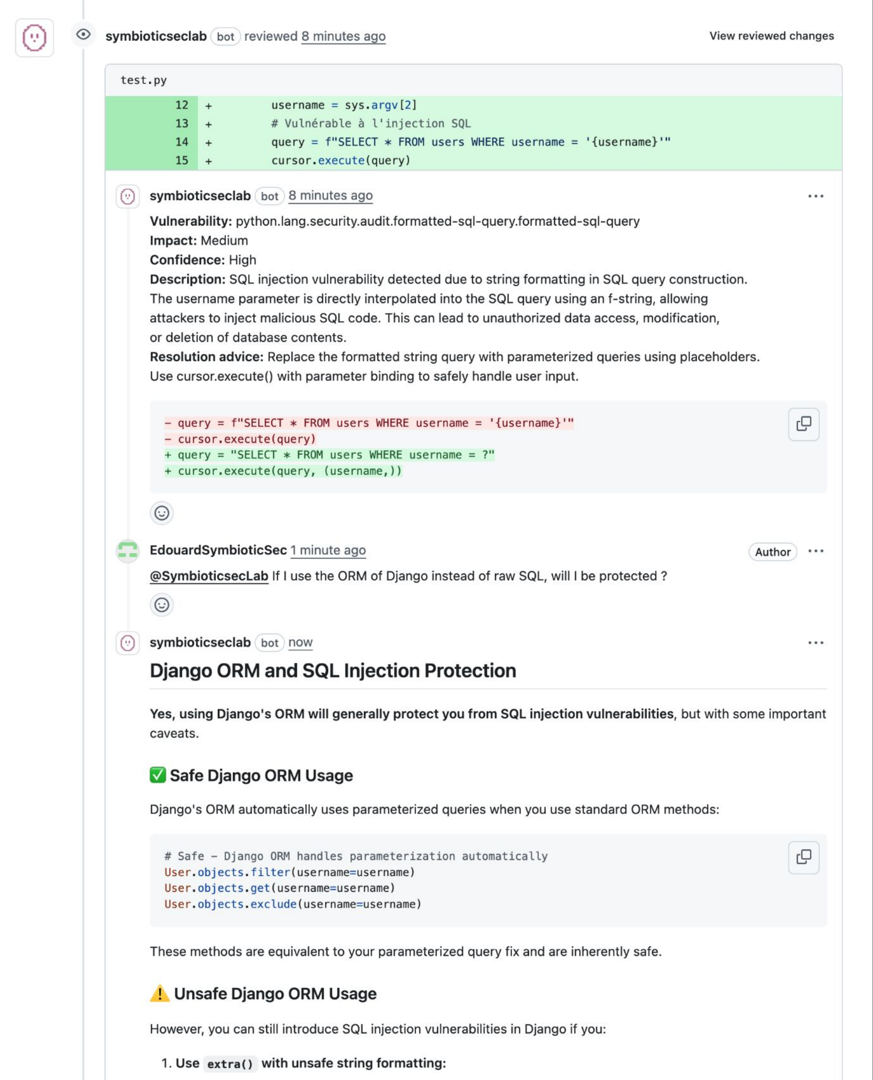
Severity	Title	Rule ID	File	Line
WARNING	`Security Issue`	`python.lang.security.audit.formatted-sql-query.formatted-sql-query`	`/tmp/pr_12/head/test.py`	`15`
ERROR	`Security Issue`	`python.sqlalchemy.security.sqlalchemy-execute-raw-query.sqlalchemy-execute-raw-query`	`/tmp/pr_12/head/test.py`	`15`

- To request an AI-generated fix, comment: `SYMBIOREMEDIATE <Rule ID>` (e.g., `SYMBIOREMEDIATE CWE-79`)
- **Available Commands:**
  - `SYMBIOREVIEW all` - Post detailed reviews for all findings.
  - `SYMBIOREVIEW <Rule ID>` - Post a detailed review for a specific finding.
  - `SYMBIOREMEDIATE all` - Suggest fixes for all findings.
  - `SYMBIOREMEDIATE <Rule ID>` - Suggest a fix for a specific finding.
  - `SYMBIOLEARN` - As a response to a review comment for educational details of the issue.

Ask for a detailed review

Want to know more?

>_ Request a detailed review
of a specific vulnerability
or all detected vulnerabilities



symbioticseclab bot reviewed 8 minutes ago

test.py

```
+        username = sys.argv[2]
+        # Vulnérable à l'injection SQL
+        query = f"SELECT * FROM users WHERE username = '{username}'"
+        cursor.execute(query)
```

symbioticseclab bot 8 minutes ago

Vulnerability: python.lang.security.audit.formatted-sql-query.formatted-sql-query
Impact: Medium
Confidence: High
Description: SQL injection vulnerability detected due to string formatting in SQL query construction. The username parameter is directly interpolated into the SQL query using an f-string, allowing attackers to inject malicious SQL code. This can lead to unauthorized data access, modification, or deletion of database contents.
Resolution advice: Replace the formatted string query with parameterized queries using placeholders. Use cursor.execute() with parameter binding to safely handle user input.

```
- query = f"SELECT * FROM users WHERE username = '{username}'"
- cursor.execute(query)
+ query = "SELECT * FROM users WHERE username = ?"
+ cursor.execute(query, (username,))
```

EdouardSymbioticSec 1 minute ago

@SymbioticsecLab If I use the ORM of Django instead of raw SQL, will I be protected ?

symbioticseclab bot now

Django ORM and SQL Injection Protection

Yes, using Django's ORM will generally protect you from SQL injection vulnerabilities, but with some important caveats.

✅ Safe Django ORM Usage

Django's ORM automatically uses parameterized queries when you use standard ORM methods:

```
# Safe - Django ORM handles parameterization automatically
User.objects.filter(username=username)
User.objects.get(username=username)
User.objects.exclude(username=username)
```

These methods are equivalent to your parameterized query fix and are inherently safe.

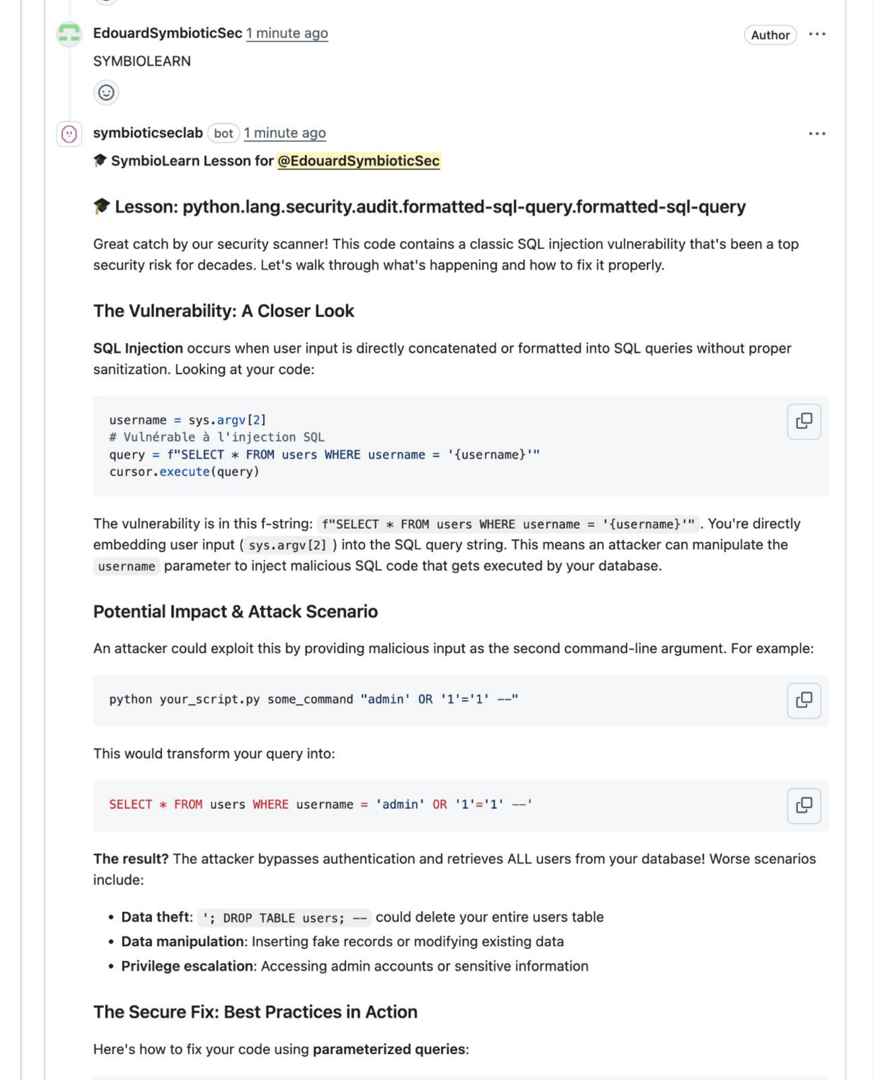
⚠️ Unsafe Django ORM Usage

However, you can still introduce SQL injection vulnerabilities in Django if you:

1. **Use `extra()` with unsafe string formatting:**

? Learn more about a vulnerability

>_ Ask for a detailed micro-training on a vulnerability to avoid doing it again



EdouardSymbioticSec 1 minute ago

SYMBIOLEARN

symbioticseclab 1 minute ago

🎓 **SymbioLearn Lesson for @EdouardSymbioticSec**

🎓 Lesson: python.lang.security.audit.formatted-sql-query.formatted-sql-query

Great catch by our security scanner! This code contains a classic SQL injection vulnerability that's been a top security risk for decades. Let's walk through what's happening and how to fix it properly.

The Vulnerability: A Closer Look

SQL Injection occurs when user input is directly concatenated or formatted into SQL queries without proper sanitization. Looking at your code:

```
username = sys.argv[2]
# Vulnérable à l'injection SQL
query = f"SELECT * FROM users WHERE username = '{username}'"
cursor.execute(query)
```

The vulnerability is in this f-string: `f"SELECT * FROM users WHERE username = '{username}'"`. You're directly embedding user input (`sys.argv[2]`) into the SQL query string. This means an attacker can manipulate the `username` parameter to inject malicious SQL code that gets executed by your database.

Potential Impact & Attack Scenario

An attacker could exploit this by providing malicious input as the second command-line argument. For example:

```
python your_script.py some_command "admin' OR '1'='1' --"
```

This would transform your query into:

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1' --'
```

The result? The attacker bypasses authentication and retrieves ALL users from your database! Worse scenarios include:

- **Data theft**: `'; DROP TABLE users; --` could delete your entire users table
- **Data manipulation**: Inserting fake records or modifying existing data
- **Privilege escalation**: Accessing admin accounts or sensitive information

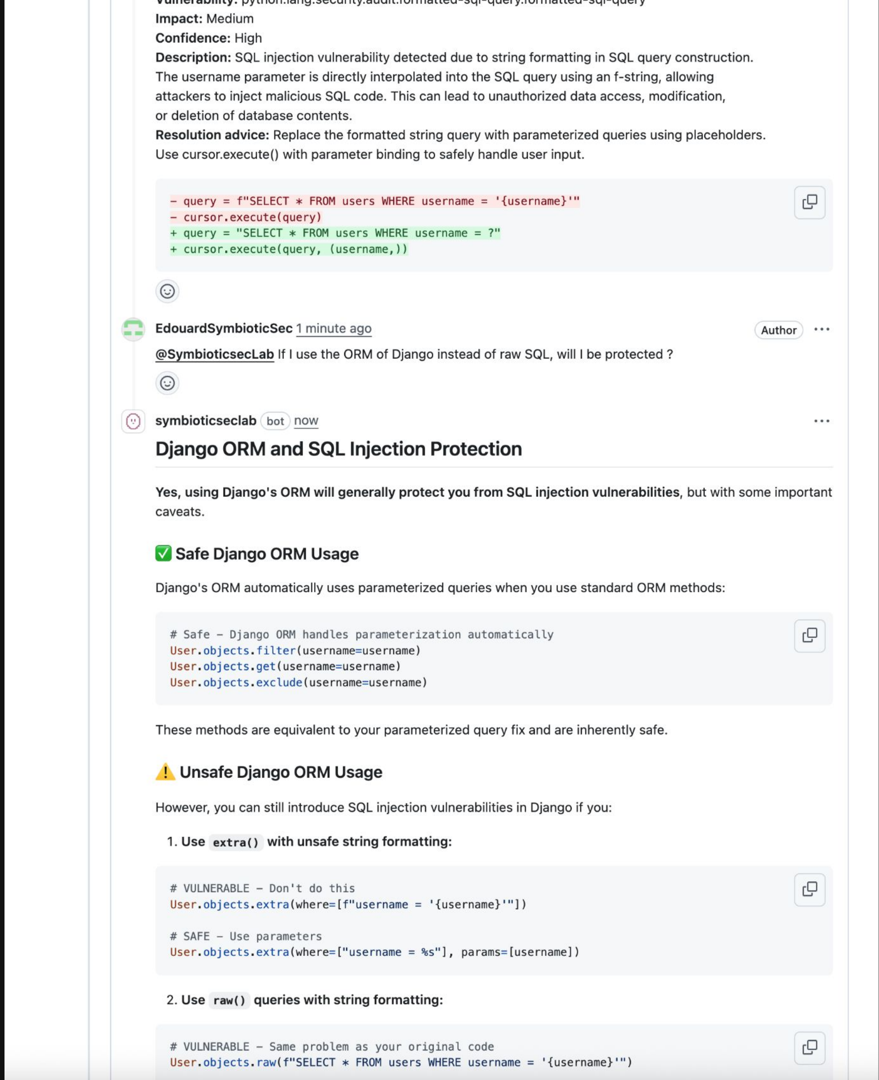
The Secure Fix: Best Practices in Action

Here's how to fix your code using **parameterized queries**:

Discuss with our AI

Want to dig deeper?

>_ Ask your question to our AI directly by commenting the generated review comment.



Vulnerability: python.lang.security.audit.formatted-sql-query.formatted-sql-query
Impact: Medium
Confidence: High
Description: SQL injection vulnerability detected due to string formatting in SQL query construction. The username parameter is directly interpolated into the SQL query using an f-string, allowing attackers to inject malicious SQL code. This can lead to unauthorized data access, modification, or deletion of database contents.
Resolution advice: Replace the formatted string query with parameterized queries using placeholders. Use cursor.execute() with parameter binding to safely handle user input.

```diff
- query = f"SELECT * FROM users WHERE username = '{username}'"
- cursor.execute(query)
+ query = "SELECT * FROM users WHERE username = ?"
+ cursor.execute(query, (username,))
```

EdouardSymbioticSec 1 minute ago

@SymbioticsecLab If I use the ORM of Django instead of raw SQL, will I be protected ?

symbioticseclab now

Django ORM and SQL Injection Protection

Yes, using Django's ORM will generally protect you from SQL injection vulnerabilities, but with some important caveats.

✅ Safe Django ORM Usage

Django's ORM automatically uses parameterized queries when you use standard ORM methods:

```python
# Safe - Django ORM handles parameterization automatically
User.objects.filter(username=username)
User.objects.get(username=username)
User.objects.exclude(username=username)
```

These methods are equivalent to your parameterized query fix and are inherently safe.

⚠️ Unsafe Django ORM Usage

However, you can still introduce SQL injection vulnerabilities in Django if you:

1. **Use `extra()` with unsafe string formatting:**

```python
# VULNERABLE - Don't do this
User.objects.extra(where=[f"username = '{username}'"])

# SAFE - Use parameters
User.objects.extra(where=["username = %s"], params=[username])
```

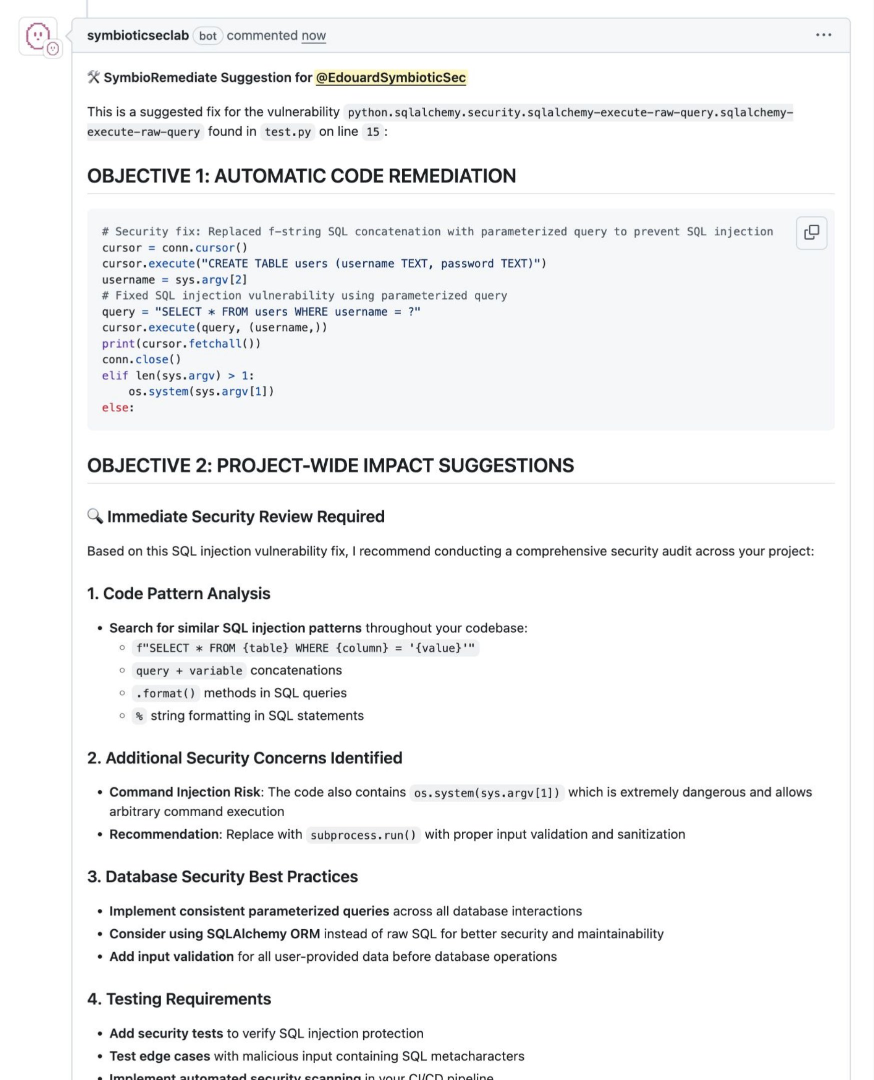
2. **Use `raw()` queries with string formatting:**

```python
# VULNERABLE - Same problem as your original code
User.objects.raw(f"SELECT * FROM users WHERE username = '{username}'")
```

Get a remediation suggestion

Want to dig deeper?

>_ Benefit our cutting edge AI remediation workflow to fix security issues within your PR



symbioticseclab bot commented now

🛠 SymbioRemediate Suggestion for @EdouardSymbioticSec

This is a suggested fix for the vulnerability `python.sqlalchemy.security.sqlalchemy-execute-raw-query.sqlalchemy-execute-raw-query` found in `test.py` on line `15`:

OBJECTIVE 1: AUTOMATIC CODE REMEDIATION

```
# Security fix: Replaced f-string SQL concatenation with parameterized query to prevent SQL injection
cursor = conn.cursor()
cursor.execute("CREATE TABLE users (username TEXT, password TEXT)")
username = sys.argv[2]
# Fixed SQL injection vulnerability using parameterized query
query = "SELECT * FROM users WHERE username = ?"
cursor.execute(query, (username,))
print(cursor.fetchall())
conn.close()
elif len(sys.argv) > 1:
    os.system(sys.argv[1])
else:
```

OBJECTIVE 2: PROJECT-WIDE IMPACT SUGGESTIONS

🔍 Immediate Security Review Required

Based on this SQL injection vulnerability fix, I recommend conducting a comprehensive security audit across your project:

1. Code Pattern Analysis

- **Search for similar SQL injection patterns** throughout your codebase:
  - `f"SELECT * FROM {table} WHERE {column} = '{value}'"`
  - `query + variable` concatenations
  - `.format()` methods in SQL queries
  - `%` string formatting in SQL statements

2. Additional Security Concerns Identified

- **Command Injection Risk**: The code also contains `os.system(sys.argv[1])` which is extremely dangerous and allows arbitrary command execution
- **Recommendation:** Replace with `subprocess.run()` with proper input validation and sanitization

3. Database Security Best Practices

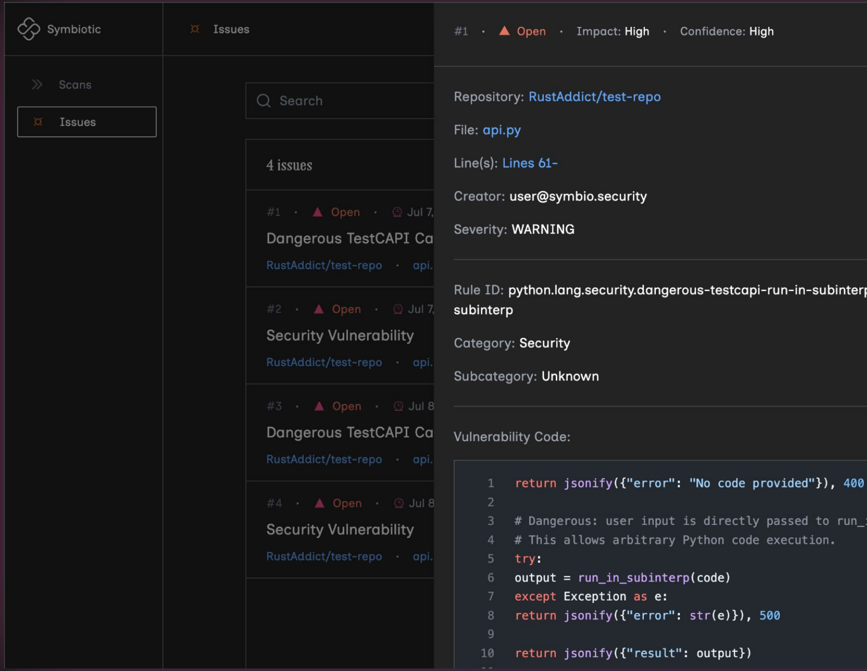
- **Implement consistent parameterized queries** across all database interactions
- **Consider using SQLAlchemy ORM** instead of raw SQL for better security and maintainability
- **Add input validation** for all user-provided data before database operations

4. Testing Requirements

- **Add security tests** to verify SQL injection protection
- **Test edge cases** with malicious input containing SQL metacharacters
- **Implement automated security scanning** in your CI/CD pipeline

Analyze insights in a dedicated portal

>_ Find all scanned repositories and scan results in our portal



Impact: High · Confidence: High

Repository: RustAddict/test-repo

File: api.py

Line(s): Lines 61-

Creator: user@symbio.security

Severity: WARNING

Rule ID: python.lang.security.dangerous-testcapi-run-in-subinterp subinterp

Category: Security

Subcategory: Unknown

Vulnerability Code:

```
return jsonify({"error": "No code provided"}), 400

# Dangerous: user input is directly passed to run_
# This allows arbitrary Python code execution.
try:
output = run_in_subinterp(code)
except Exception as e:
return jsonify({"error": str(e)}), 500

return jsonify({"result": output})
```

And if we win?

Let’s use earned credits to launch our free tier and do some co-marketing together

symbiotic
SECURITY

♡ *Thank you* ♡