SmartShop Code Documentation

This guide is part of the SmartShop training environment, designed to support hands-on learning in web application security.  
It aims to deepen your understanding of how real-world vulnerabilities occur, when and why they arise, how they can be exploited, and how they should be addressed through secure coding practices.

SmartShop is a deliberately vulnerable application that demonstrates a range of common security flaws, aligned with the OWASP Top 10.  
Each vulnerability is documented in detail, including where it appears in the code, the underlying cause, potential exploitation techniques, and recommended solutions.

The purpose of this guide is to provide practical insight into both offensive and defensive security, helping to build technical awareness and mindset needed to identify, exploit, and fix security issues in modern web applications.

Plaintext Passwords And Insecure Authentication

Name: Cryptographic Failures / Authentication Failure  
OWASP ID: A02:2021 / A07:2021  
Code refrence: main.py:

**Whats the issue?**

When passwords are stored in plaintext in the database, they can be stolen and used directly in the event of a data breach. This is a common beginner mistake and violates basic security principles. If another vulnerability, such as SQL injection, is present, an attacker could extract the passwords directly from the database without needing to crack or decrypt anything.

***Related authentication failure (A07:2021)***

This also introduces an authentication flaw, since the application checks passwords by comparing raw strings instead of using secure password verification. Combined with missing protections like brute-force prevention and token management, this leads to a weak and easily bypassed login mechanism.

**A screen shot of a computer code

AI-generated content may be incorrect.Code Refrence (insecure)**

**How do you exploit?**

* Exploit a different vulnerability like SQLi to dump the user-row.
* You can now see users and passwords in plain text.

**How do you prevent it?**

* Use hashing with salt for example, bcrypt to store passwords. bcrypt adds a unique salt and is intentionally slow, which helps defend against brute-force attacks.
* Never compare passords as plain strings, always use a password verification function as bcrypt.

**Code Reference (Secure):**

**A screen shot of a computer screen

AI-generated content may be incorrect.**

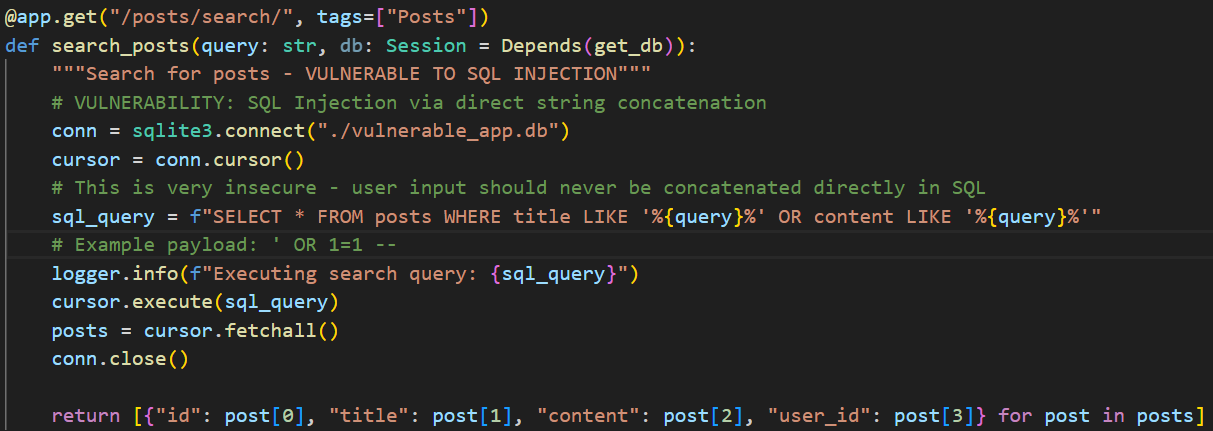
SQL – Injection in Post-search

Category: Injection  
OWASP ID: A03:2021  
Code Reference: main.py:

**Whats the issue?**

The SQL query is built using unsanitized user input, which allows attackers to inject and execute arbitrary SQL commands. This happens because the input is directly inserted into the query string, rather than being handled safely as a separate parameter.

**Code Reference (Insecure)**

****

**Here you provide the user with input**

****

**How do you exploit?**

1. If the input is ' OR 1=1 --, the resulting query becomes:

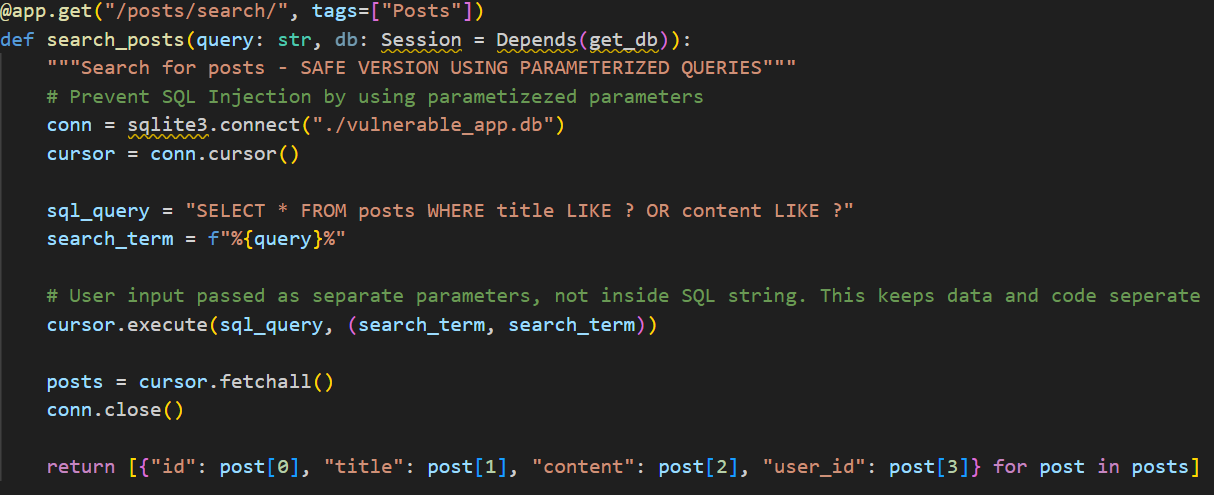


1. The query will return all posts from the table, or even other data if the payload is adjusted.
2. This allows for data extraction, credential dumping, or even corruption of database contents.

**ow do you prevent it?**

* Use parameterized queries, a way to write querys that lets the database treat user input as data, not code. Never sanitize input manually, always rely on the database engine’s built-in parameter handling.

**Code Reference (Secure)**

**

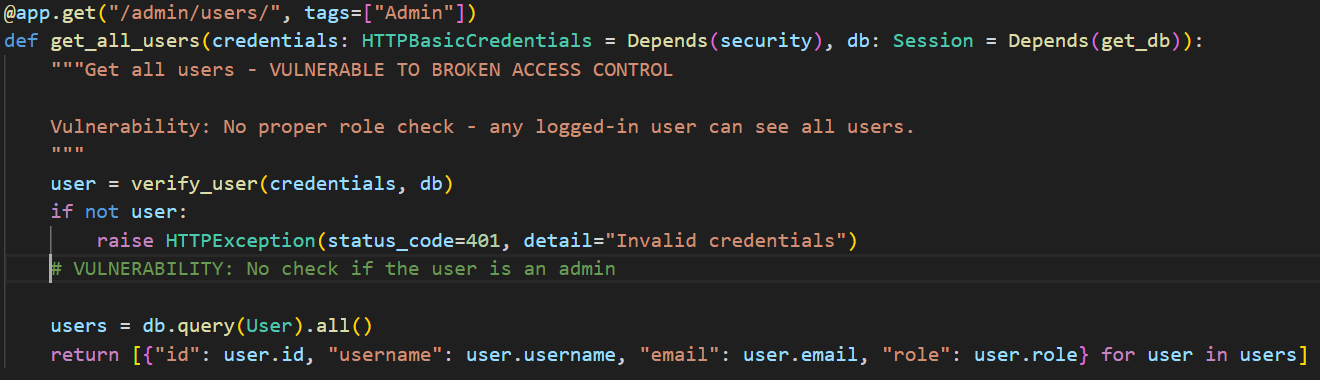
Broken Access Control

Category: Broken Access Control  
OWASP ID: A01:2021  
Code Reference: main.py:

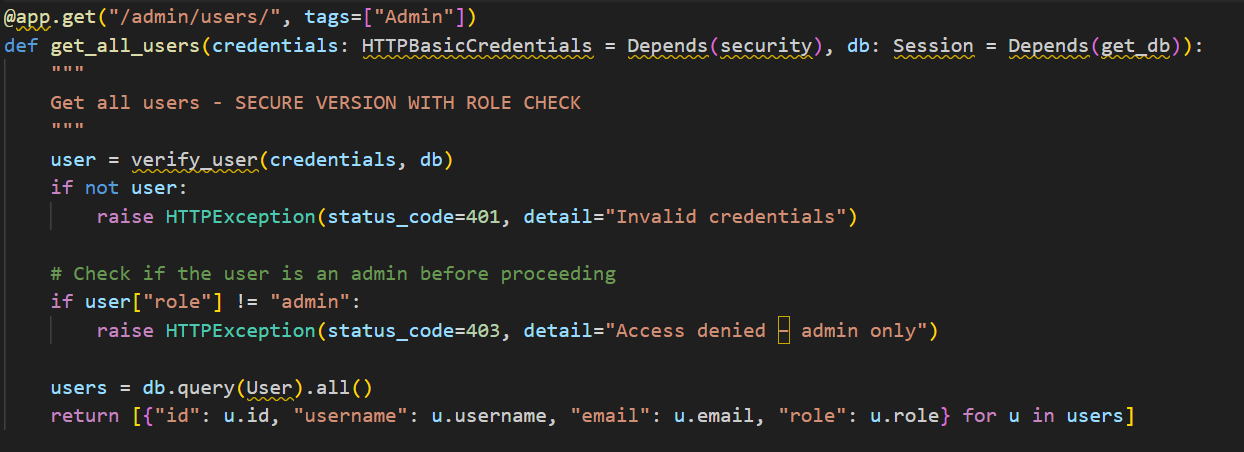
**Whats the issue?**

Funktionen returnerar all användardata utan att kontrollera användarens roll.

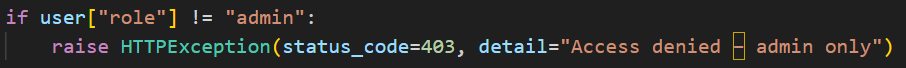
**Code Reference (Insecure)**

**

**Code Reference (Secure):**

**

**Server-side role validation**

****

**How do you exploit?**

1. Log in as regular user
2. Send GET-request to /admin/users/.
3. You access all user information, that is a serious information leak.

**How do you prevent it?**

* Always verify the user’s role on the backend before giving access
* Example with error handling: if user["role"] != "admin": return 403

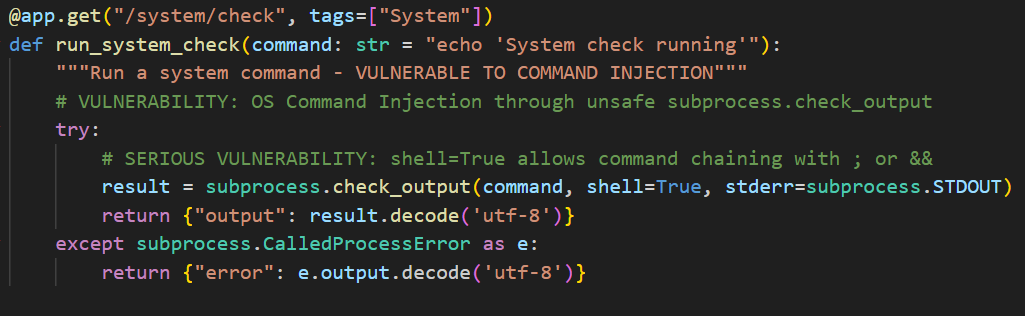
Command Injection through Shell Execution

Category: Vulnerable and Outdated Components  
OWASP ID: A06:2021  
Code Reference main.py:

**Whats the issue?**

The application takes user input and passes it directly to the system shell using subprocess.check\_output() with shell=True. This is dangerous because the shell will interpret the input as actual commands. An attacker can chain malicious commands using characters like ;, &&, or |.

**Code Reference (Insecure)**

******

**How do you exploit?**

* Call the endpoint with this input:  
  127.0.0.1 && cat /etc/passwd
* The application will run both ping and cat, leaking sensitive system files.
* In real-world deployments, this could lead to server compromise, lateral movement, or privilege escalation.

**How do you prevent it?**

* Never use shell=True when executing commands with user input.
* Use a list of arguments instead of a string. This disables shell interpretation.
* Whitelist allowed values (e.g., only specific hostnames or IPs).

**Kodexempel (säker):**

****

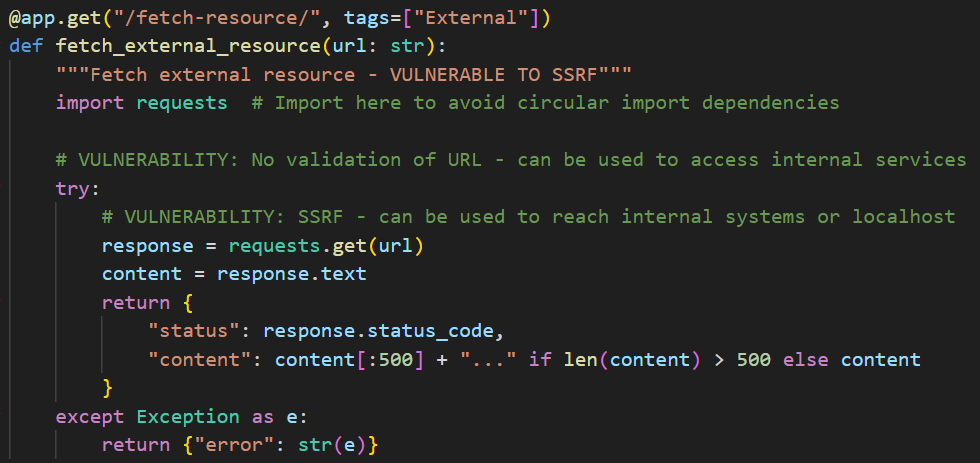
Server Side Request Forgery (SSRF)

Category: SSRF  
OWASP ID: A10:2021  
Code Reference: main.py:

**Whats the issue?**

The app lets users send in a URL, and the server fetches it without checking if it’s safe. This means an attacker can make the server call internal systems or cloud services that should not be publicly accessible.

**Code Refrence (Insecure)**

****

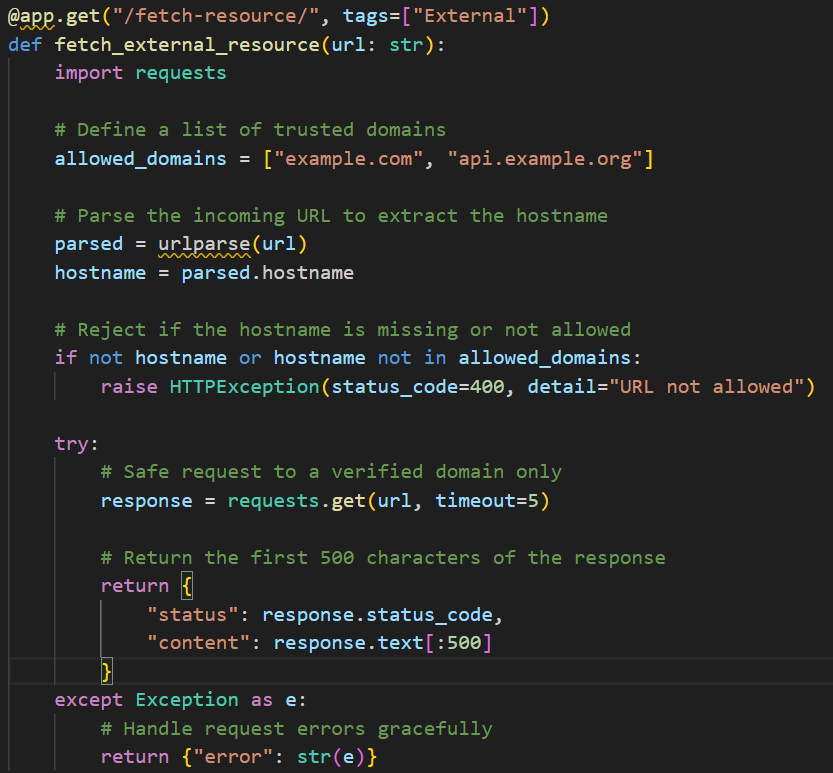
**How do you exploit?**

* Send a request with a local/internal address:  
  <http://localhost:8000/debug/config>
* The server requests its own config endpoint and leaks data like JWT secrets.

**How do you prevent it?**

* Validate and filter URLs before making requests.
* Use a DNS allowlist of trusted domains.
* Use a proxy or sandboxed microservice to make external requests safely.

**Code Reference (Secure)**

****

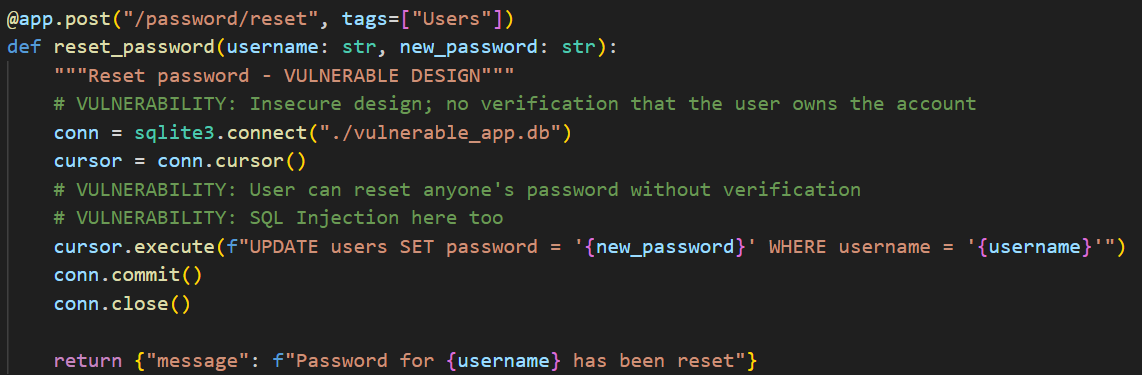
Insecure Design – Password Reset Without Verification

Category: Insecure Design  
OWASP ID: A04:2021  
Code Reference: main.py:

**What’s the issue?**

Anyone can reset a user's password by sending a POST request with a username. There is no verification step (e.g., token via email), allowing account takeover by design.

**Code Reference (Insecure)**

****

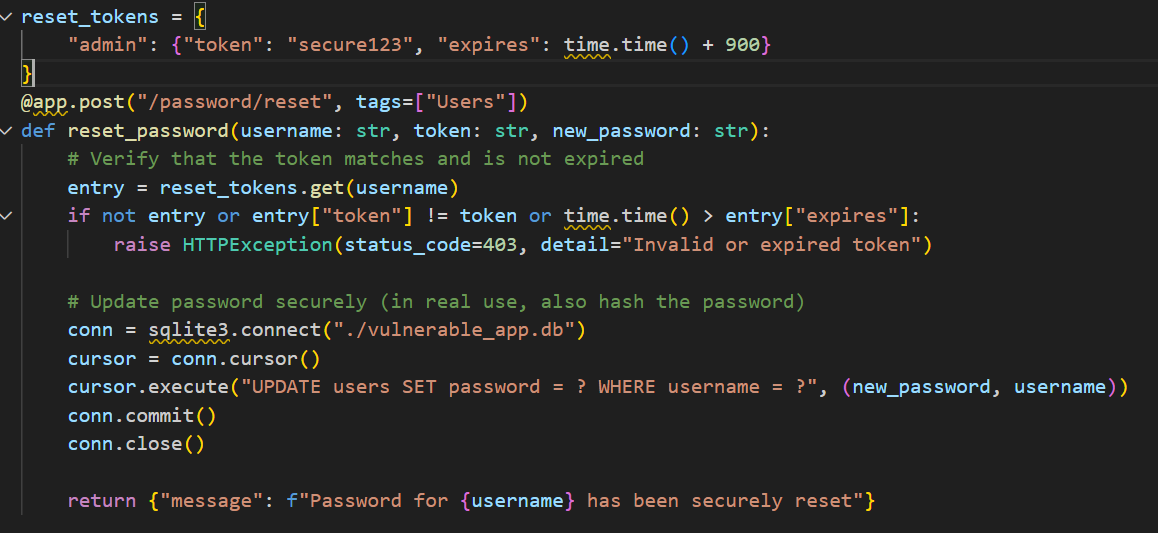
**How do you exploit?**

* Send a reset request for any user (username=admin).
* Provide a new password.
* You now control that user's account.

**How do you prevent it?**

* Require identity verification before allowing password resets.
* Use secure, time-limited tokens sent to the user's email.
* Never allow unauthenticated users to perform account-changing actions.

**Code Reference (Secure)**

****

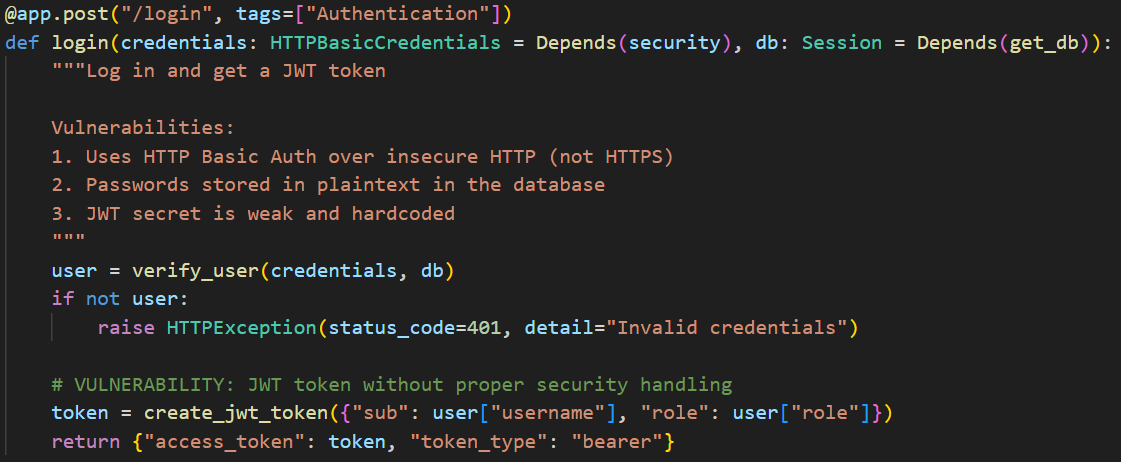
Brute Force-login No Rate limiting

Category: Identification and Authentication Failures / Cryptographic Failure  
OWASP ID: A07:2021 / A02:2021  
Code Reference: main.py:

**What’s the issue?**

The login system does not limit failed attempts. There are no delays, no lockouts, and no monitoring, allowing attackers to automate password guessing freely.

**Code Reference (Insecure)**



**How do you exploit?**

* Run an automated script to try common passwords.
* Eventually, a valid combination will succeed.
* There is no detection or protection in place.

**How do you prevent it?**

* Add login rate limiting (e.g. max 5 attempts).
* Use CAPTCHA or temporary lockouts on repeated failure.
* Log failed attempts and alert if thresholds are exceeded.

**Code Reference (Secure)**



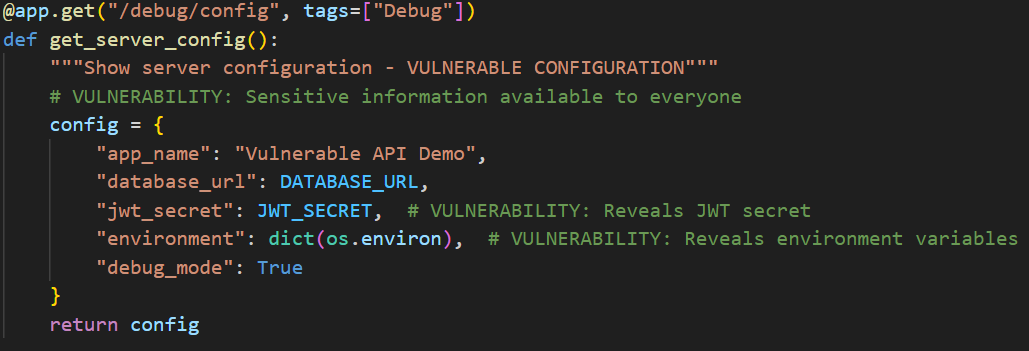
Exposed Sensitive Data – Debug Config Endpoint

Category: Security Misconfiguration  
OWASP ID: A05:2021  
Code Reference: main.py:

**What’s the issue?**

The /debug/config endpoint exposes internal environment variables, including secrets like JWT\_SECRET, database config, and debug flags. This gives attackers everything they need.

**Code Reference (Insecure)**

****

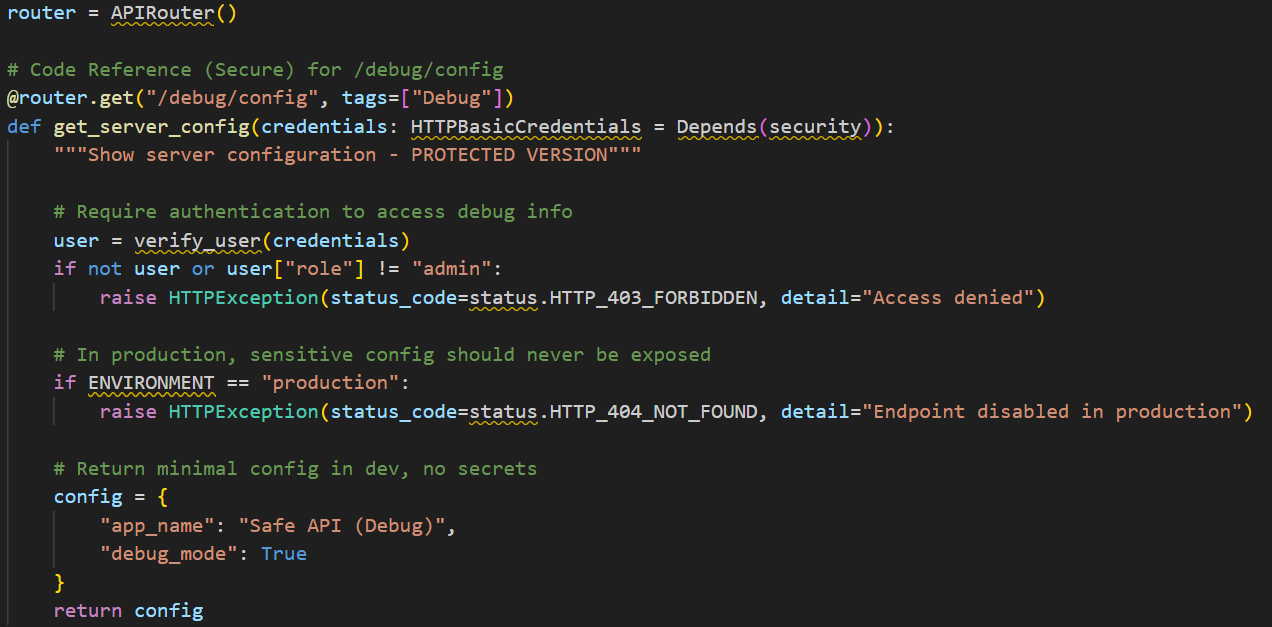
**How do you exploit?**

* Request /debug/config – no auth required.
* Receive full internal config including environment variables.
* Use secrets to forge tokens or escalate access.

**How do you prevent it?**

* Disable debug and diagnostic endpoints in production.
* Never expose secrets in frontend or APIs.
* Restrict internal debug tools behind auth or IP allowlists.

**Code Reference (Secure)**

****

Logging & Monitoring Failures - No Access Logging

Category: Security Logging and Monitoring Failures  
OWASP ID: A09:2021  
Code Reference: main.py

**What’s the issue?**

Sensitive user data is returned without requiring authentication, and no logging or alerts are in place. This allows silent data exfiltration

**Code Reference (Insecure)**

A screen shot of a computer code

AI-generated content may be incorrect.

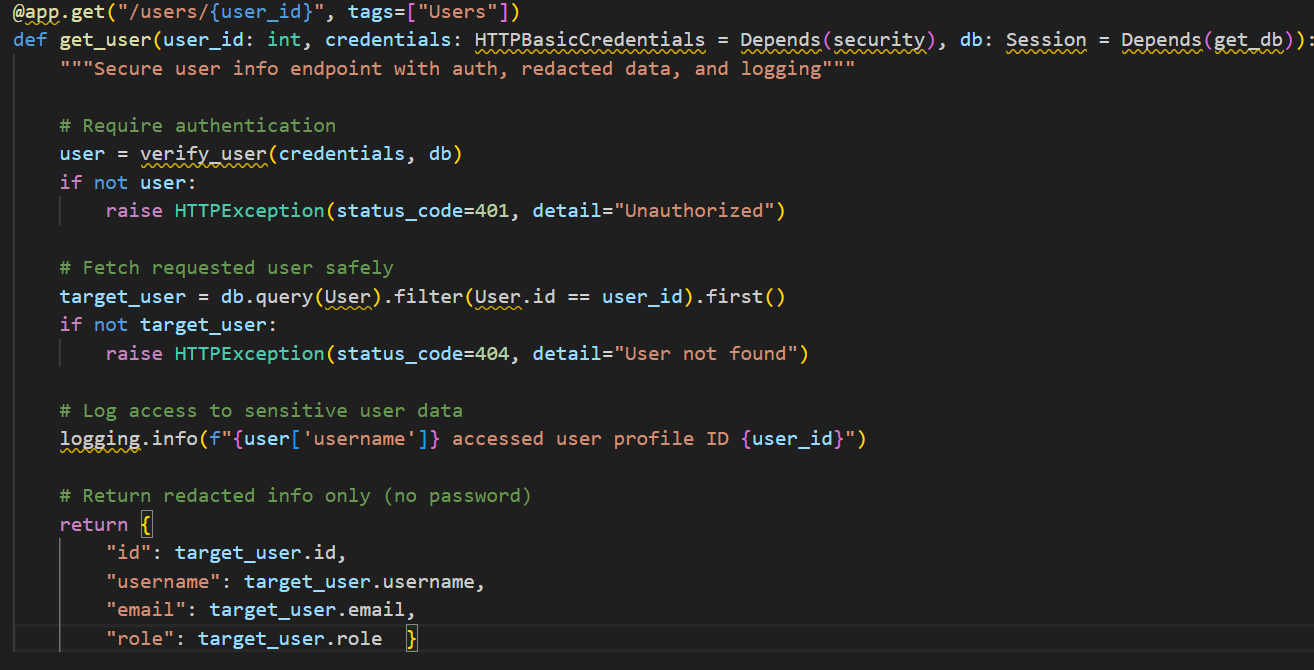
**How do you exploit?**

* Request /users/1, /users/2, etc.
* Receive sensitive data, including password fields.
* No logs are generated to detect misuse.

**How do you prevent it?**

* Require authentication and authorization for all user data endpoints.
* Log all access to sensitive resources.
* Monitor logs for patterns of abuse.

**Code Reference (Secure)**



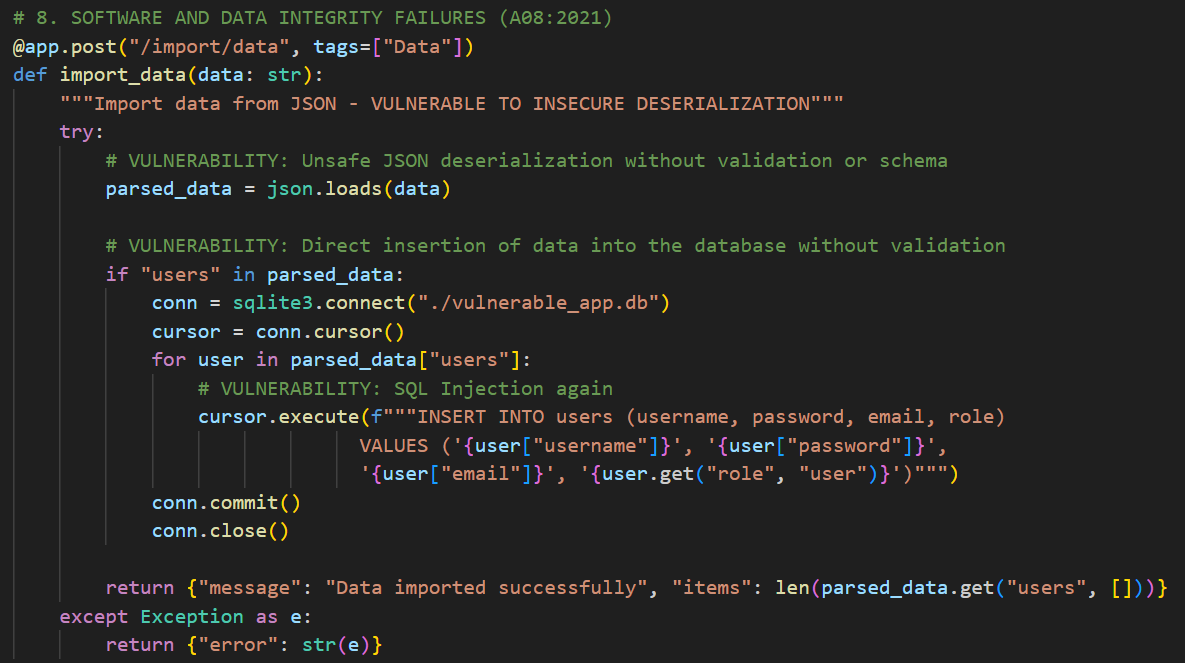
Insecure Deserialization – Unsafe JSON Import

Category: Software and Data Integrity Failures  
OWASP ID: A08:2021  
Code Reference: main.py

**What’s the issue?**

The application imports user data from a raw JSON string, which is parsed and inserted directly into the database without any validation. This opens for insecure deserialization, where attackers can craft malicious payloads that bypass integrity checks and potentially execute harmful actions, such as SQL injection or privilege escalation.

**Code Reference (Insecure)**

****

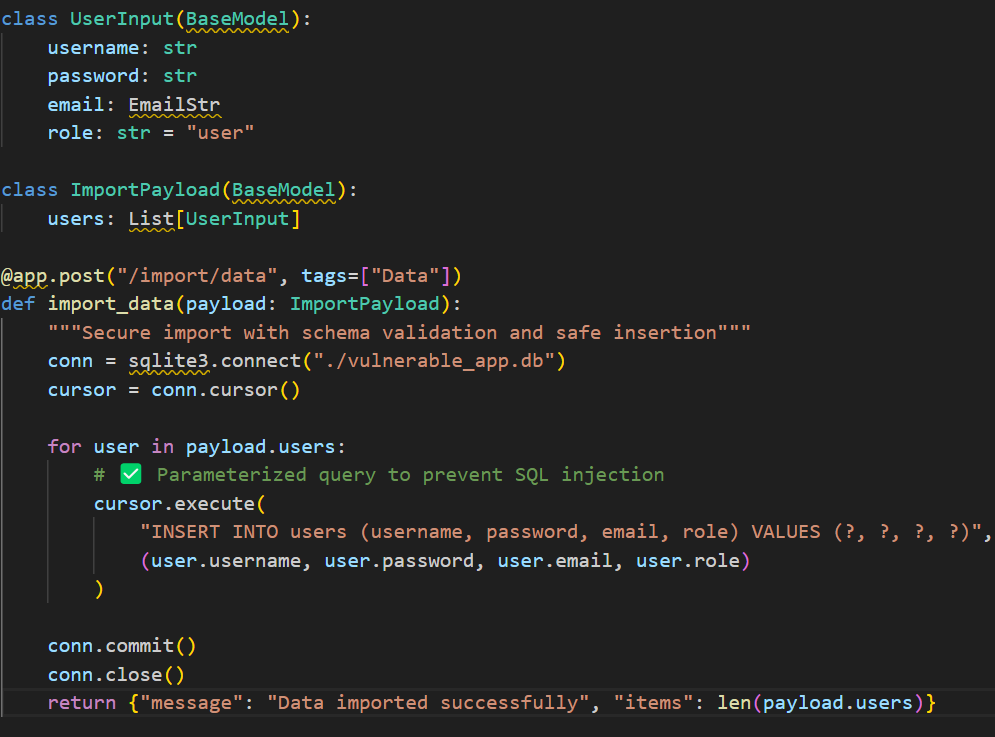
**How do you exploit?**

* Send a JSON payload containing crafted user objects.
* Use SQL injection in any field, such as "username": "admin'; DROP TABLE users;--".
* The payload is deserialized and inserted without validation or sanitization.

**How do you prevent it?**

* Use a proper data model (e.g., pydantic) to validate and sanitize input before using it.
* Avoid directly executing SQL queries with user data — use parameterized queries.
* Only accept trusted file formats and verify their structure/schema before processing.

**Code Reference (Secure)**

****