



nextwork.org

AI Security Scanner for Python



Abhinave P.B

```
PS C:\Users\abhin\OneDrive\Desktop\IronFleet\Archives\security-scanner> bash
Hello, security scanner!
(venv) abhinave32@DESKTOP-32A7P9:~/net/c/Users/abhin/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py

this single line of code password = "admin123" has **severe security implications** and is a textbook example of what **NOT** to do with sensitive information, especially passwords.

Here's a breakdown of the security issues:
1. **Hardcoded Password**
   * **Problem:** The password "admin123" is directly written into the source code. This means anyone with access to the code (source control, deployed artifacts, compiled binaries, logs if the code is ever printed) will immediately know the password.
   * **Risk:** If this code is ever compromised, the password is leaked instantly. It also makes password rotation impossible without modifying and redeploying the code.

2. **Plain-Text Storage**
   * **Problem:** Storing the password in plain text, even in memory, is extremely dangerous. If this variable is ever accidentally logged, saved to a file, database, or a memory dump occurs, the password is exposed.
   * **Risk:** Even if not hardcoded, storing it in plain text is a critical vulnerability. Attackers can often read process memory or intercept data in transit.

3. **Weak Password Choice ("admin123")**
   * **Problem:** "admin123" is a very common, easily guessable password. It's susceptible to dictionary attacks, brute-force attacks, and credential stuffing (where attackers try common username/password combinations across many sites).
   * **Risk:** An attacker wouldn't even need to compromise the code; they could likely guess this password quickly.

4. **Contextual Risks (Depending on where this password is used)**
   * **Problem:** User's Password (for a login system): This is catastrophic. User passwords must never be stored in plain text. They should be hashed using a strong, slow, and salted algorithm (like bcrypt, Argon2, etc.) and stored in the hash stored.
   * **Risk:** If it's an Application/Service Password (e.g., database credential, API key): This is also extremely dangerous. A compromise of this application would immediately grant access to the backend database, other services, or sensitive APIs.

**In Summary**
The line password = "admin123" represents a fundamental security flaw that can lead to:
* **Data Breaches:** Exposure of sensitive user data, application data, or system access.
* **System Compromise:** Attackers gaining unauthorized access to the application, database, or other connected services.
* **Reputational Damage:** Loss of user trust due to security incidents.

**Recommendations for Secure Password Handling:**
1. **For User Passwords (Login Systems):**
   * **Never store plain text passwords.** Instead, store only the hash and salt.
   * **Enforce Complexity:** Require strong passwords (length, mix of characters).
```



Abhinave P.B

NextWork Student

nextwork.org

Introducing Today's Project!

In this project, I'm going to build AI Security Scanner for python. This will help me learn how code vulnerabilities and how to manage them in a production environment. I'm interested in this because of to learn how to avoid security vulnerabilities in my code.

Key tools and concepts

Tools I used were Python, Gemini API, dotenv, and colorama for colored terminal output. Key concepts I learnt include API integration, environment variable management, prompt engineering for security analysis, and detecting common vulnerabilities like weak hashing and SQL injection. The most important skill was building an end-to-end workflow — reading files, sending them to an AI model, and presenting structured, meaningful security results.

Challenges and wins

This project took me approximately a few hours to design, test, and refine. The most challenging part was crafting an effective security prompt and handling API errors while ensuring structured, reliable output. It was most rewarding to see the scanner successfully detect real vulnerabilities and display them with clear, color-coded severity levels.



Abhinave P.B

NextWork Student

nextwork.org

Why I did this project

I did this project today because I wanted to strengthen my understanding of cybersecurity and learn how to integrate AI into real-world tools. This project met my goals by helping me build a working AI-powered vulnerability scanner that detects security issues and presents structured results. Next, I plan to improve it by adding more vulnerability categories, better report formatting (maybe JSON export), and possibly turning it into a CLI tool or web app.

Abhinave P.B
NextWork Student

nextwork.org

Connecting to Gemini API

In this step, I'm setting up the Gemini API connection. This involves configuring the Google Generative AI SDK, loading environment variables securely, authenticating using my API key, and initializing a Gemini model instance. I need to do this so my application can communicate with Google's AI infrastructure, send prompts, and receive model-generated responses in real time. Establishing this connection is a foundational step for integrating AI capabilities into my project.

```
File Edit Selection View Go Run Terminal Help ← → Q: security-scanner
EXPLORER SECURITY-SCANNER
resources
.gitignore
requirements.txt
scanner.py
scanner.py M .gitignore
scanner.py
You, 3 seconds ago | author (None)
1 import os
2 from dotenv import load_dotenv
3 from google import genai
4
5 # Load environment variables from .env file
6 load_dotenv()
7
8 # Configure with your API key from environment variable
9 api_key = os.getenv("GOOGLE_API_KEY")
10 client = genai.Client(api_key=api_key)
11
12 # YOUR CODE GOES BELOW HERE
13
14
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS GITLENS AZURE
PS C:\Users\abhin\OneDrive\Desktop\IronFleet\Archives\security-scanner> bash
abhin@DESKTOP-32W4TP9:~/ent/c/Users/abhin/OneDrive/Desktop/IronFleet/Archives/security-scanner$ source venv/bin/activate
(venv) abhin@DESKTOP-32W4TP9:~/ent/c/Users/abhin/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py
Hello, security scanner!
(venv) abhin@DESKTOP-32W4TP9:~/ent/c/Users/abhin/OneDrive/Desktop/IronFleet/Archives/security-scanner$
```

I verified the connection by running a test request to the Gemini API. Gemini responded with a valid output, which confirmed that the API key and client configuration were working correctly.

Abhinave P.B
NextWork Student

nextwork.org

The screenshot shows a terminal window titled "security-scanner" with the following content:

```
PS C:\Users\Abhinave\OneDrive\Desktop\IronFleet\Archives\security-scanner> bash
Hello, security scanner!
(venv) abhinave@DESKTOP-32W7P9:~$ /mnt/c/users/abhinave/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py
This single line of code `password = "admin123"` has **severe security implications** and is a textbook example of what **NOT** to do with sensitive information, especially passwords.
```

Here's a breakdown of the security issues:

1. **Hardcoded Password:**
 - * **Problem:** The password `admin123` is directly written into the source code. This means anyone with access to the code (source control, deployed artifacts, compiled binaries, logs if the code is ever printed) will immediately know the password.
 - * **Risk:** If this code is ever compromised, the password is leaked instantly. It also makes password rotation impossible without modifying and redeploying the code.
2. **Plain-text Storage:**
 - * **Problem:** Storing the password in plain text, even in memory, is extremely dangerous. If this variable is ever accidentally logged, saved to a file, database, or a memory dump occurs, the password is exposed.
 - * **Risk:** Even if not hardcoded, storing it in plain text is a critical vulnerability. Attackers can often read process memory or intercept data in transit.
3. **Weak Password Choice ('admin123'):
 - * **Problem:** `admin123` is a very common, easily guessable password. It's susceptible to dictionary attacks, brute-force attacks, and credential stuffing (where attackers try common usernames/password combinations across many sites).
 - * **Risk:** An attacker wouldn't even need to compromise the code; they could likely guess this password quickly.
4. **Contextual Risks (Depending on where this password is used):**
 - * **If it's a User's Password (for a login system):** This is catastrophic. User passwords **must** never be stored in plain text. They should be hashed using a strong, slow, and salted algorithm (like bcrypt, Argon2, or scrypt) and then the hash stored.
 - * **If it's an Application/Service Password (e.g., database credential, API key):** This is also extremely dangerous. A compromise of this application would immediately grant access to the backend database, other services, or sensitive APIs.

In Summary:

The line `password = "admin123"` represents a fundamental security flaw that can lead to:

- * **Data Breaches:** Exposure of sensitive user data, application data, or system access.
- * **System Compromise:** Attackers gaining unauthorized access to the application, database, or other connected services.
- * **Reputational Damage:** Loss of user trust due to security incidents.

Recommendations for Secure Password Handling:

1. **For User Passwords (Login Systems):**
 - * **Never store plain text passwords.**
 - * **Store only the hash and salt.**
 - * **Enforce Complexity:** Require strong passwords (length, mix of characters).

My scanner.py file works by sending a code snippet to the Gemini API using the generate_content method and requesting a security analysis. When I ran it, Gemini identified that the line password = "admin123" contains severe security vulnerabilities, including hardcoded credentials, plain-text storage risks, and weak password practices. This shows that the Gemini API can successfully analyze source code, detect common security flaws, and provide structured explanations along with best-practice recommendations.

A circular portrait of a young man with dark hair and a beard, wearing a blue polo shirt.

Abhinave P.B

NextWork Student

nextwork.org

Building the Vulnerability Scanner

In this step, I am designing a prompt-driven static vulnerability scanner that evaluates source code for common security misconfigurations and injection risks. The system will feed vulnerable samples into a large language model and assess detection quality, remediation accuracy, and severity classification. This forms the basis of an AI-assisted code auditing pipeline.

```
File Edit Selection View Go Run Terminal Help ↻ → Q security-scanner

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS GITLENS AZURE

0 PS C:\Users\abhin\OneDrive\Desktop\Ironfleet\Archives\security-scanner> bash
abhin@abhin-DESKTOP-32N7P9: /mnt/c/Users/abhin/OneDrive/Desktop/Ironfleet/Archives/security-scanner> source venv/bin/activate
(venv) abhin@abhin-DESKTOP-32N7P9: /mnt/c/Users/abhin/OneDrive/Desktop/Ironfleet/Archives/security-scanner> python3 scanner.py
Here's an analysis of the provided code for vulnerabilities:
...
...
### 1. SQL Injection

* **Vulnerability type:** SQL Injection
* **Why it's vulnerable:** User input ('username') is directly concatenated into the SQL query string without proper sanitization or parameterization.
* **Impact:** An attacker can manipulate the database query to bypass authentication, access sensitive data, or modify/delete data.
* **Secure code fix:** 
```python
def get_user(username):
 # Use parameterized queries to separate SQL logic from data
 query = "SELECT * FROM users WHERE username = %s"
 cursor.execute(query, (username,)) # Pass parameters as a tuple
 return cursor.fetchone()
```
...
...
### 2. Hardcoded Secrets

* **Vulnerability type:** Hardcoded Secrets / Information Disclosure
* **Why it's vulnerable:** Sensitive credentials ('DATABASE_PASSWORD', 'API_KEY') are embedded directly in the source code.
* **Impact:** If the source code is compromised or accidentally exposed, these critical credentials can be easily discovered and misused by attackers.
* **Secure code fix:** 
```python
Load secrets from environment variables or a secure configuration management system
DATABASE_PASSWORD = os.getenv("DATABASE_PASSWORD")
API_KEY = os.getenv("API_KEY")

Add error handling for missing environment variables
if not DATABASE_PASSWORD:
 raise ValueError("DATABASE_PASSWORD environment variable not set.")
if not API_KEY:
 raise ValueError("API_KEY environment variable not set.")

 def connect_db():
```
You, now Ln 72, Col 30 Spaces: 4 UTF-8 CR/LF [ ] Python 3.13.5 ⟲ ⟳ Go Live ⟲ Go Live ⟲ Prettier Formatting: X
```

placeholder

The vulnerabilities Gemini detected were SQL injection through unsafe string interpolation, hardcoded API keys and database credentials stored in plain text, and weak password validation logic that only checked minimum length without enforcing complexity requirements.



Abhinave P.B

NextWork Student

nextwork.org

It also highlighted unsafe input handling patterns and recommended parameterized queries, environment variables for secret management, and stronger password policies with secure hashing algorithms like bcrypt or Argon2. The security prompt I crafted asked for explicit identification of SQL injection, hardcoded secrets, and weak password logic; a clear explanation of each vulnerability; the exact vulnerable lines of code; a severity classification (Low/Medium/High/Critical); and a secure, corrected version of the code. It required precise, technical responses rather than generic advice.

Abhinave P.B
NextWork Student

nextwork.org

Adding Severity Ratings

In this step, severity ratings are introduced to categorize identified issues by impact level. The colorama library is installed to enable colored terminal output, improving clarity and user experience when displaying scan results.

```
PS C:\Users\abhinav\OneDrive\Desktop\IronFleet\Archives\security-scanner> bash
abhinave32@DESKTOP-32UW7P9:/mnt/c/Users/abhinav/OneDrive/Desktop/IronFleet/Archives/security-scanner$ source venv/bin/activate
(venv) abhinave32@DESKTOP-32UW7P9:/mnt/c/Users/abhinav/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py
Start's an analysis of the provided code for vulnerabilities.

(venv) abhinave32@DESKTOP-32UW7P9:/mnt/c/Users/abhinav/OneDrive/Desktop/IronFleet/Archives/security-scanner$ abhinave32@DESKTOP-32UW7P9:/mnt/c/Users/abhinav/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py
Analyzing SQL Injection Example...
-----
SEVERITY: CRITICAL
TYPE: SQL Injection
DESCRIPTION: User-supplied input is directly concatenated into a SQL query string without proper sanitization or parameterization.
IMPACT: An attacker can inject malicious SQL code, leading to unauthorized data access, modification, deletion, or even full database compromise.
FDX:
```python
def get_user(username):
 query = "SELECT * FROM users WHERE username = %s"
 cursor.execute(query, (username,))
 return cursor.fetchone()
```
-----
Analyzing Hardcoded Credentials Example...
-----
SEVERITY: CRITICAL
TYPE: Hardcoded Credentials
DESCRIPTION: The database password is directly embedded within the source code.
IMPACT: Anyone gaining access to the source code can retrieve the database password, leading to potential full database compromise.
FDX:
```python
import os
DATABASE_PASSWORD = os.getenv("DATABASE_PASSWORD")
if not DATABASE_PASSWORD:
 raise ValueError("DATABASE_PASSWORD environment variable not set.")
```
-----
SEVERITY: CRITICAL
TYPE: Hardcoded Credentials
```

I built an AI-powered static security analysis tool that analyzes code using Gemini, enforces structured vulnerability reporting, and visually prioritizes issues using color-coded severity levels. This makes security review faster and more developer-friendly.

Abhinave P.B
NextWork Student

nextwork.org

Scanning Real Python Files

In this secret mission, I'm adding file reading capability and command-line argument support to my security scanner. Professional tools do this because security scanners need to integrate into real developer workflows. Tools like Bandit, Flake8, and Semgrep don't require you to paste code — they scan actual files and entire projects from the command line. Supporting file paths makes the tool:

- Practical for real-world usage
- Scriptable and automation-friendly
- CI/CD ready
- Scalable to larger codebases

```
PS C:\Users\abhinave\OneDrive\Desktop\IronFleet\Archives\security-scanner> bash
abhinave@DESKTOP-32LN7P9:/mnt/c/Users/abhinave/Desktop/IronFleet/Archives/security-scanner$ source venv/bin/activate
(venv) abhinave@DESKTOP-32LN7P9:/mnt/c/Users/abhinave/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python3 scanner.py
Here's an analysis of the provided code for vulnerabilities:
-
(venv) abhinave@DESKTOP-32LN7P9:/mnt/c/Users/abhinave/OneDrive/Desktop/IronFleet/Archives/security-scanner$ python scanner.py vulnerable.py
=====
Scanning: vulnerable.py
=====

...
TYPE: CRITICAL
DESCRIPTION: The 'hash_password' function uses SHA-256, which is too fast for password hashing, and the 'authenticate' function directly compares a cleartext password without using any hashing for verification.
IMPACT: User accounts are highly vulnerable to brute-force attacks and direct credential exposure if the database is compromised, leading to account takeover.
FIX:
```python
import bcrypt # Consider installing with 'pip install bcrypt'

def hash_password(password):
 # Use bcrypt (or Argon2/Scrypt) for strong, slow password hashing
 # bcrypt generates its own salt and includes it within the hash string
 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

def authenticate(username, password):
 conn = sqlite3.connect('users.db')
 cursor = conn.cursor()
 # Assuming the 'password_hash' column stores the bcrypt hash
 cursor.execute("SELECT password_hash FROM users WHERE username = ?", (username,))
 result = cursor.fetchone()
 conn.close() # Close connection as soon as possible
 if result:
 stored_hash = result[0]
 # Verify the provided password against the stored hash
 if bcrypt.checkpw(password.encode('utf-8'), stored_hash.encode('utf-8')):
 # Password matches, re-open connection to fetch user details (excluding password hash)
 conn = sqlite3.connect('users.db')
 cursor = conn.cursor()
 cursor.execute("SELECT id, username FROM users WHERE username = ?", (username,))
 user_data = cursor.fetchone()
 conn.close()
```
You now  Ln 33, Col 80  Spaces: 4  UTF-8  CRLF  [ ] Python  3:13.5  Go Live  Go Live  Prettier  Formatting: X
```



Abhinave P.B

NextWork Student

nextwork.org

I scanned vulnerable.py by running: python scanner.py vulnerable.py The vulnerabilities detected were CRITICAL Insecure Password Management, including use of fast SHA-256 hashing and improper password comparison, making the system vulnerable to brute-force attacks and credential exposure. The scan_file function works by reading the target file, sending its code to Gemini with a security prompt, and printing structured, color-coded vulnerability results in the terminal.



nextwork.org

The place to learn & showcase your skills

Check out nextwork.org for more projects

