Secure Software Development in Python: Hands-On Coding Workshop

Engr. Zia Ur Rehman
engrziaurrehman.kicsit@gmail.com
LinkedIn, GitHub, ResearchGate, PyPI

May 20, 2025

1. Python Security Fundamentals – Hands-on Tasks

1.1. Task 1: Dangerous Use of `eval()`

1.1.1 What is `eval()` in Python?

`eval()` is a built-in Python function that evaluates a string as a Python expression and returns the result.

Basic Usage:

```
result = eval("2 + 3")
print(result)  # Output: 5
```

It takes a string as input and interprets it as actual Python code.

1.1.2 Why is `eval()` used?

Some legitimate use cases include:

- **Dynamic Expression Evaluation:** Suppose you want to let users input mathematical expressions, like in a calculator.

  ```
      user_input = input("Enter math expression: ")
      print(eval(user_input))
  ```

- **Scripting in Applications:** Some advanced applications allow admins or power users to write small Python expressions to automate behavior.
- **Interpreting Simple DSLs (Domain-Specific Languages):** Some configuration or command files might use Python syntax to define rules or behaviors.

1.1.3 Why is `eval()` Dangerous?

Because `eval()` executes any code you give it, a malicious user can exploit it to:

- Run system commands
- Delete files
- Access unauthorized data
- Install malware

Vulnerable Code:

```
user_input = input("Enter␣a␣mathematical␣expression␣(e.g.,␣2+2):␣")
print(eval(user_input))
```

Issue: `eval()` can execute arbitrary code. For example: `__import__('os').system('rm -rf /')`

Secure Alternative:

```
import ast
import operator

def safe_eval(expr):
    allowed_ops = {ast.Add: operator.add, ast.Sub: operator.sub, ast.Mult:
        ↪  operator.mul, ast.Div: operator.truediv}

    def eval_node(node):
        if isinstance(node, ast.BinOp):
            return allowed_ops[type(node.op)](eval_node(node.left),
                ↪ eval_node(node.right))
        elif isinstance(node, ast.Num):
            return node.n
        else:
            raise ValueError("Unsupported␣operation")

    tree = ast.parse(expr, mode='eval')
    return eval_node(tree.body)

expr = input("Enter␣a␣simple␣math␣expression:␣")
print(safe_eval(expr))
```

Explanation: Parses and evaluates only safe arithmetic expressions using the AST module.

1.1.4 When NOT to Use `eval()`?

- If the input comes from untrusted users
- In web applications
- In APIs or scripts exposed publicly
- When working with files, databases, or OS-level tasks

Try this on your PC!

```
result = eval ( "__import__('os').system('dir')" )
print ( result )
```

1.2. Task 2: Insecure Subprocess/File Execution

Vulnerable Code:

```
import os
filename = input("Enter␣filename␣to␣delete:␣")
os.system(f"rm␣{filename}")
```

Issue: Shell injection vulnerability.
Secure Alternative:

```
import subprocess
from pathlib import Path

filename = input("Enter␣filename␣to␣delete:␣")

if Path(filename).exists():
    subprocess.run(["rm", filename])
else:
    print("File␣does␣not␣exist.")
```

Explanation: Avoids shell interpretation by using `subprocess.run` with a list.

1.2.1 What does it do?

It runs a shell command (like `rm important.txt`) directly using `os.system()`, which passes the string to the operating system's command line interpreter.

1.2.2 Why is it dangerous?

Dangerous Example:

```
Enter filename to delete: file.txt; rm -rf /
```

This would run:

```
rm file.txt; rm -rf /
```

If the script runs with admin rights, it could delete the entire system.

1.2.3 What kind of attacks does this allow?

- Shell injection
- Command chaining (;)
- Privilege escalation if combined with other vulnerabilities
- Remote Code Execution (RCE) in web servers

1.3. Task 3: Hardcoded Secrets

Vulnerable Code:

```
API_KEY = "abc123XYZ"
print("Using API Key:", API_KEY)
```

Secure Alternative:

```
from decouple import config

API_KEY = config('API_KEY')
print("Using API Key:", API_KEY)
```

.env File:

```
API_KEY=abc123XYZ
```

Secrets are stored outside your code and .env file can be ignored via .gitignore. **Explanation:** Avoids hardcoding secrets; uses environment variables.

1.3.1 Why is this dangerous?

Hardcoding secrets like:

- API keys
- Database passwords
- Tokens
- SSH credentials

..into your source code poses serious risks:

If your code is ever:

- Uploaded to GitHub (even accidentally)
- Shared with a teammate
- Deployed in Docker images

Your secrets are compromised.

1.3.2 Real-World Security Breaches

- Many companies have suffered data leaks because secrets were pushed to GitHub and picked up by bots scanning public repos.
- Hackers use these leaked keys to access APIs, send spam, or steal data.

1.4. Task 4: Dependency Vulnerability Audit

1.4.1 What's the issue?

Your Python app might depend on external libraries (e.g., flask, numpy, requests). These packages:

- May contain known security vulnerabilities
- Might be outdated or unpatched
- Could introduce supply chain attacks

Even if your code is safe, an insecure dependency can be an entry point for attackers.

1.4.2 Real Example

If you install a vulnerable version of requests that allows header injection, your app could be:

- Tricked into redirecting users
- Exposing internal services
- Logging sensitive data

Attack Type	**Caused by Vulnerable Package**
Remote Code Execution	Malicious code injection
Arbitrary File Write	Path traversal vulnerabilities
Credential Leak	Logging sensitive data
Denial of Service (DoS)	Poor input handling

Table 1: Common Attack Types and Their Causes

1.4.3 Secure Practice: Use Dependency Audit Tools

Python has several tools for auditing packages: **Commands:**

```
pip install bandit safety pip-audit
bandit -r .
safety check
pip-audit
```

Explanation: Use static analysis tools to detect vulnerabilities and audit dependencies.

Tool	`Command`	Purpose
bandit	`bandit -r .`	Scans your Python code for security issues and common vulnerabilities
safety	`safety check`	Scans installed packages against known CVE databases (requires `pip install safety`)
pip-audit	`pip-audit`	Audits your installed Python packages for known vulnerabilities

Table 2: Security Tools for Python Projects

Integration Tips

- Use these tools in your **Software Development** to prevent deployments with insecure dependencies
- Schedule **regular audits**, especially after `pip install` or upgrading packages
- Consider adding security scanning as both **pre-commit hooks** and **nightly jobs**
- Integrate with GitHub Actions/GitLab CI for automated vulnerability detection

Example CI integration: `bandit -r . || exit 1`

Context	Why It's Used
E-commerce app	Avoid critical vulnerabilities in `payment` or `auth` libraries that could lead to financial fraud or data breaches
Health platform	Ensure HIPAA/GDPR compliance by auditing dependencies that handle PHI (Protected Health Information)
Fintech service	Prevent financial data exposure from outdated or vulnerable libraries in transaction processing systems
Government portals	Required by security standards (OWASP Top 10, NIST SP 800-53) for public sector applications

Table 3: Security Scanning Contexts and Rationale

Bonus Tip: Pin Your Dependencies

Best Practice

Always use requirements.txt with pinned versions:

```
requests==2.28.1   # Pinned exact version
flask==2.2.2       # Avoid automatic updates
```

Then run:

```
pip install -r requirements.txt
```

This ensures:

- Consistent environments across deployments
- Auditable dependency trees
- Protection against unexpected breaking changes

1.5. Task 5: Insecure Randomness for Token Generation

1.5.1 What's the issue?

Vulnerable Code:

```
import random
token = str(random.random())
print("Token:", token)
```

Risk: Predictable token generation. This generates a predictable float between 0 and 1 (like 0.498564839).

1.5.2 Why is this dangerous?

The `random` module in Python is not cryptographically secure. It's:

- Deterministic
- Based on a seed
- Predictable by attackers

Attack Scenario: If your app uses `random.random()` to generate:

- Session tokens
- Password reset links
- API keys

...an attacker can potentially guess or brute-force valid tokens.

Secure Version:

```
import secrets

token = secrets.token_urlsafe(16)
print("Secure␣token:", token)
```

Why Secure: Uses cryptographic randomness for generating tokens. Generates a secure, URL-safe token like: 'U9$_4K9YTi35dcGfZ5GcN4w'$

1.5.3 Available Secure Methods in `secrets`

Method	Use Case
`secrets.token_bytes(n)`	Generate cryptographically secure random bytes (binary data)
`secrets.token_hex(n)`	Create secure hexadecimal tokens (2×n characters) for API keys
`secrets.token_urlsafe(n)`	Generate URL-safe base64 tokens for password reset links
`secrets.choice(seq)`	Securely select random element from sequences (no bias)

Table 4: Secure Random Generation Methods in Python's `secrets` Module

1.5.4 Real-World Use Cases

Icon	Scenario	Why Secure Randomness is Needed
🔑	Password Reset Links	Prevent unauthorized account access through predictable tokens
🔒	Session Tokens	Mitigate session hijacking and fixation attacks
▥	API Keys	Avoid key collisions and credential leakage in logs
	Temporary Access Passes	Ensure single-use, time-limited unique identifiers
💬	Invite Codes or OTPs	Prevent brute-force guessing of time-sensitive codes

Table 5: Critical Scenarios Requiring Cryptographically Secure Randomness

1.6. Task 6: Logging Sensitive Data – Avoid Credential Logging

Vulnerable Code:

```
username = input("Username:␣")
password = input("Password:␣")
print(f"User␣tried␣to␣login␣with␣{username}:{password}")
```

This logs the username and password directly to standard output or log files.

1.6.1 Why is this dangerous?

Logging sensitive data like:

- Passwords
- Tokens
- Email addresses
- Credit card numbers
- Private keys

...can expose your users if:

- Logs are stored insecurely
- Logs are accessed by attackers or unauthorized staff
- Logs are pushed to a cloud platform or GitHub

Real-World Scenarios

- **Incident Response Team** found AWS keys in debug logs.
- **DevOps team** accidentally uploaded logs to a public S3 bucket.
- **Attackers** searched logs for password patterns using scripts.

Secure Logging Practice:

```
import logging

username = input("Username:␣")
password = input("Password:␣")

logging.info(f"Login␣attempt␣by␣user:␣{username}")
# Do not log password!
```

Why Secure: Keeps passwords out of log files.

1.6.2

sectionNever Log

- `password`
- `user`$_t oken$`Authorization`$headers$
- Payment details
- Session cookies

System	What You Should Log	Security Consideration
	Login System	Username and timestamp only ⊘ Never log passwords
💳	Payment Gateway	Transaction IDs only ⊘ Mask all card/PII data
❤	Healthcare App	Patient IDs only in *encrypted* logs ✓ HIPAA compliant
☰	Web Server	Sanitized query params ⊘ Filter sensitive headers

Table 6: Secure Logging Best Practices by System Type

1.6.3 Real-World Use Cases

1.7. Task 7: Type Confusion – Validating Input Types

Vulnerable Code:

```
age = input("Enter␣your␣age:␣")
print("Next␣year␣you␣will␣be:", age + 1)
```

Risk: Throws an error or performs unexpected behavior. Here, `input()` always returns a string — even if the user enters a number.

Why is this dangerous?

If you don't validate or convert the input:

- It causes runtime errors (e.g., **TypeError: can only concatenate str (not "int"**))
- May lead to unexpected behavior or crashes
- If input is passed to other systems (e.g., DB queries, JSON, APIs), this can become a security vulnerability (e.g., injection or logic bugs)

1.7.1 Real-World Use Cases

Secure Version:

```
try:
    age = int(input("Enter␣your␣age:␣"))
    print("Next␣year␣you␣will␣be:", age + 1)
except ValueError:
    print("Invalid␣age␣input")
```

Why Secure: Prevents type confusion and handles bad input gracefully.

Scenario	Problem
Age input	String `"20"` used in math operation causes 🐞 TypeError crash
Price calculation	`"100" + "5"` concatenates to `"1005"` ⚠ instead of summing to 105
SQL query parameter	Unvalidated input `"1; DROP TABLE users"` 💀 causes SQL injection
File path handling	User input `"../../etc/passwd"` 📄 leads to path traversal
JSON parsing	Malformed input {`"admin": true`} bypasses checks 👤

Table 7: Common Input Validation Failures and Security Risks

Bonus Task: Validating Email Format

```
import re

email = input("Enter␣your␣email:␣")
pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'

if re.match(pattern, email):
    print("Valid␣email")
else:
    print("Invalid␣email␣format")
```

Why Secure: Reduces risk of malformed/unverified inputs before processing.

1.7.2 Real-World Use Cases

1.7.3 Use Input Validation Libraries

For advanced validation, use:

- `pydantic` – Enforces types, ranges, patterns
- marshmallow – Serialization validation
- `Regex` – For emails, phone numbers, etc.

Input Field	Data Type	Validation Rules
Age	`int`	Range 1–120 ✓
Product price	`float`	Positive values only $
Phone number	`string`	Digits only, length 10–15 📞
Rating	`int`	Range 1–5 ★
Quantity	`int`	≥ 0 #

Table 8: Input Validation Requirements for Common Fields

2. Secure Authentication, Input Handling, and Threat Mitigation

2.1. Task 8: Password Hashing

Vulnerable:

```
users = {"admin": "password123"}
```

This stores the password in plain text.

Why is this dangerous?

Storing raw passwords puts your users and your system at extreme risk. If your database is compromised:

- All user passwords are immediately exposed
- Users often reuse passwords → Multiple accounts compromised
- You face reputation damage, legal consequences, and data breaches

2.1.1 What is Password Hashing?

Hashing is a one-way function that converts the password into a unique, fixed-size string (hash). It cannot be reversed.

Even if a database is leaked, attackers won't get the original password. **Secure Alternative: Use `bcrypt` for hashing**

```
import bcrypt

# Hashing
password = input("Enter␣your␣password:␣").encode()
hashed = bcrypt.hashpw(password, bcrypt.gensalt())
print("Stored␣hash:", hashed)

# Verifying
if bcrypt.checkpw(password, hashed):
    print("Password␣match!")
```

```
else:
    print("Incorrect␣password")
```

Explanation: Hash passwords using salted hashing to avoid plaintext storage.

2.1.2 Why `bcrypt`?

Feature	Benefit
One-way Hashing	Cannot be reversed into original password Even with database compromise
Salting	Random data added to each password Prevents rainbow table attacks
Slow by Design	Intentional computational complexity Thwarts brute-force attempts
Widely Tested	Community-vetted algorithms ✓ Trusted by security experts

Table 9: Security Benefits of Modern Password Hashing Techniques

Real-World Password Leaks

- In 2012, LinkedIn lost 117M plaintext passwords.
- In 2021, RockYou2021 combined 8.4 billion leaked credentials.
- These breaches could've been minimized with proper hashing and salting.

2.1.3 How `bcrypt` Works (Simplified)

- Takes password + salt
- Runs slow hash algorithm multiple times (configurable)
- Produces a secure hash like:

```
$2b$12$Dyzs0Tx7BXz5vVABnm3Zt0VHgJtvMJx1CCzOmPHI9T7tTI8a7/CNi
```

2.1.4 Bonus Tip: Never Use MD5 or SHA1 for Passwords

They are:

- Fast and outdated
- Easily cracked using rainbow tables
- Not designed for password security

Use:

- `bcrypt` (recommended)
- `argon2` (next-gen)
- `scrypt` (memory-hard alternative)

2.2. Task 9: SQL Injection

Vulnerable Code:

```
import sqlite3

conn = sqlite3.connect("users.db")
cur = conn.cursor()

username = input("Username:␣")
query = f"SELECT␣*␣FROM␣users␣WHERE␣username␣=␣’{username}’"
cur.execute(query)
```

This code directly inserts user input into an SQL query string.

Why is this dangerous?

When you inject user input directly into a SQL command, attackers can manipulate the query to:

- Bypass authentication
- Steal data from other users
- Delete tables or entire databases
- Escalate privileges

2.2.1 Example of SQL Injection Attack

User Input:

```
’␣OR␣1=1␣--
```

Final query becomes:

```
SELECT * FROM users WHERE username = ’’ OR 1=1 --’
```

This always returns `True`, and the attacker can bypass login checks or dump the entire users table.
Secure Alternative:

```
cur.execute("SELECT␣*␣FROM␣users␣WHERE␣username␣=␣?", (username,))
```

Explanation: Use parameterized queries to avoid injection. This way, user input is treated as data, not code.