Here are some **secure coding practices** to follow when developing applications, particularly in **Python and Flask**:

1. Input Validation

☑ **Validate and sanitize all user inputs** to prevent injection attacks.

- Use regular expressions to enforce input constraints.
- Reject or escape special characters that could be used for exploits.

Example:

```
import re

def validate_username(username):
    if not re.match(r"^[a-zA-Z0-9_]{3,20}$", username):
        return False  # Reject invalid input
    return True
```

2. Avoid Command Injection

☑ **Never pass user inputs directly into system commands.**

- Use **whitelists** of allowed commands.
- Use `shlex.split()` when working with `subprocess` to safely parse arguments.

Example:

```
import subprocess
import shlex

ALLOWED_COMMANDS = {"ls", "whoami", "uptime"}

def run_secure_command(cmd):
    if cmd not in ALLOWED_COMMANDS:
        return "Command not allowed"
    return subprocess.check_output(shlex.split(cmd)).decode()
```

3. Secure Error Handling

☑ **Don't expose detailed error messages in production.**

- Hide stack traces from users to prevent information leaks.
- Log errors securely for debugging.

Example:

```
import logging

logging.basicConfig(filename="app.log", level=logging.ERROR)

try:
    result = 10 / 0  # Division by zero error
```

```
except Exception as e:
    logging.error(f"Unexpected error: {e}")
    print("An error occurred. Please try again later.")  # Generic message
```

4. Use Secure Authentication & Authorization

☑ **Use strong password hashing algorithms like bcrypt or Argon2.**
🔹 **Never store passwords in plain text!**
🔹 Implement **role-based access control (RBAC)** to restrict actions.

Example:

```
from werkzeug.security import generate_password_hash, check_password_hash

hashed_password = generate_password_hash("SecurePass123",
method="pbkdf2:sha256")
print(check_password_hash(hashed_password, "SecurePass123"))  # True
```

5. Secure Your Flask Application

☑ **Disable debug mode in production.**
☑ **Use HTTPS instead of HTTP.**
☑ **Set security headers** to prevent attacks like XSS and Clickjacking.

Example:

```
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
    app.run(debug=False)  # Disable debug mode
```

To enforce **security headers**, use **Flask-Talisman**:

```
pip install flask-talisman
from flask_talisman import Talisman

app = Flask(__name__)
Talisman(app)  # Enforces HTTPS and security headers
```

6. Protect Against Cross-Site Scripting (XSS)

☑ **Escape all user-generated content before rendering.**
☑ **Use Content Security Policy (CSP).**
☑ **Avoid inserting raw user input into HTML.**

Example:

```
from flask import escape

@app.route("/greet")
def greet():
    name = request.args.get("name", "Guest")
    return f"Hello, {escape(name)}!"  # Escaping prevents XSS
```

7. Prevent SQL Injection

☑ **Use parameterized queries instead of string concatenation.**

Example (BAD - Vulnerable to SQL injection):

```
query = f"SELECT * FROM users WHERE username = '{user_input}'"
```

Example (GOOD - Using parameterized queries):

```
import sqlite3

conn = sqlite3.connect("database.db")
cursor = conn.cursor()
cursor.execute("SELECT * FROM users WHERE username = ?", (user_input,))
```

8. Secure File Uploads

☑ **Check file types and restrict upload locations.**
☑ **Never store user-uploaded files in executable directories.**

Example:

```
import os
from werkzeug.utils import secure_filename

UPLOAD_FOLDER = "/safe/uploads"
ALLOWED_EXTENSIONS = {"png", "jpg", "jpeg", "gif"}

def allowed_file(filename):
    return "." in filename and filename.rsplit(".", 1)[1].lower() in
ALLOWED_EXTENSIONS

@app.route("/upload", methods=["POST"])
def upload_file():
    file = request.files["file"]
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file.save(os.path.join(UPLOAD_FOLDER, filename))
```

9. Implement Rate Limiting

☑ **Prevent brute-force attacks by limiting requests per user.**
☑ **Use Flask-Limiter to control API abuse.**

Installation:

```
pip install flask-limiter
```

Usage:

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)
limiter = Limiter(get_remote_address, app=app, default_limits=["100 per
hour"])

@app.route("/login", methods=["POST"])
@limiter.limit("5 per minute")  # Limit to 5 login attempts per minute
def login():
    return "Login endpoint"
```

10. Keep Dependencies Updated

☑ **Regularly update libraries to patch security vulnerabilities.**
☑ **Use tools like `pip-audit` or `safety` to check for vulnerabilities.**

Check outdated packages:

```
pip list --outdated
```

Audit security issues:

```
pip install pip-audit
pip-audit
```

Summary of Secure Coding Practices

☑ **Validate all inputs (whitelist approach).**
☑ **Use secure command execution methods.**
☑ **Mask errors and log securely.**
☑ **Hash passwords with strong algorithms.**
☑ **Disable debug mode and enforce HTTPS.**
☑ **Use CSP and escape user inputs to prevent XSS.**
☑ **Prevent SQL injection with parameterized queries.**
☑ **Restrict file uploads to safe locations.**
☑ **Limit API requests to prevent abuse.**
☑ **Regularly update dependencies to patch vulnerabilities.**

By following these best practices, your application will be **much more resistant** to common security threats! 🚀