Tutorial: Improving Security When Writing Computer Code

Writing secure code is one of the most important responsibilities of a software developer. Security vulnerabilities can allow attackers to steal data, gain unauthorized access, or damage systems. This tutorial covers best practices that help make your code more secure.

---

1. Validate All User Input

Never trust data entered by users, received from websites, APIs, or files.

Example (Unsafe)

```
age = int(input("Enter your age: "))
```

If the user enters text instead of a number, the program may crash.

Example (Safer)

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a valid number.")
```

Best Practices

- Check data type.
- Verify length limits.
- Allow only expected characters.
- Reject invalid input immediately.

---

2. Prevent SQL Injection

SQL Injection occurs when attackers insert malicious SQL commands into input fields.

Unsafe Example

```
query = "SELECT * FROM users WHERE username = '" + username + "'"
```

Safe Example

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

Best Practices

- Use parameterized queries.
- Never concatenate user input into SQL statements.
- Use ORM frameworks when possible.

---

3. Store Passwords Securely

Never store passwords in plain text.

Unsafe

```
password = "mypassword123"
```

Safer

Use password hashing:

```
import bcrypt

password = b"mypassword123"
hashed = bcrypt.hashpw(password, bcrypt.gensalt())
```

Best Practices

- Use bcrypt, Argon2, or PBKDF2.
- Never create your own encryption algorithms.
- Use multi-factor authentication when possible.

---

4. Keep Secrets Out of Source Code

Never place API keys, passwords, or database credentials directly in code.

Unsafe

```
API_KEY = "abc123secret"
```

Better

Use environment variables:

```
import os

API_KEY = os.getenv("API_KEY")
```

Best Practices

- Use `.env` files.
- Add secret files to `.gitignore`.
- Rotate credentials regularly.

---

5. Follow the Principle of Least Privilege

Applications should only have the permissions they actually need.

Example

If your application only reads data:

✅ Read access only

❌ Read, write, delete, and administrator access

Benefits

- Reduces damage if compromised.
- Limits attacker capabilities.

---

6. Use Secure Communication

Data should be encrypted while traveling across networks.

Best Practices

- Use HTTPS.
- Use TLS certificates.
- Avoid sending sensitive information through unencrypted channels.

Avoid

http://example.com

Prefer

https://example.com

---

7. Handle Errors Securely

Detailed error messages can reveal sensitive information.

Unsafe

print(database_password)

Better

print("An error occurred. Please contact support.")

Best Practices

- Log detailed errors privately.
- Show generic messages to users.
- Never expose stack traces in production.

---

8. Keep Software Updated

Many security breaches exploit known vulnerabilities.

Regularly Update

- Programming languages
- Libraries
- Frameworks
- Operating systems

Example

npm update

pip install --upgrade package_name

---

9. Use Code Reviews

A second set of eyes often catches security issues.

Review For

- Hardcoded credentials
- Unsafe input handling
- Missing authentication checks
- Improper permissions

Benefits

- Finds vulnerabilities early.
- Improves code quality.

---

10. Use Static Analysis Tools

Security scanners can detect common vulnerabilities automatically.

Examples

- SonarQube
- Bandit (Python)
- ESLint Security Plugins
- OWASP Dependency Check

Run scans regularly during development.

---

11. Implement Authentication and Authorization Properly

Authentication verifies identity.

Authorization determines what users can access.

Example

```
if user.is_admin:
    delete_user()
```

Always verify permissions before performing sensitive actions.

---

12. Protect Against Cross-Site Scripting (XSS)

When developing web applications, sanitize user-generated content.

Unsafe

```
<div>{{ user_input }}</div>
```

Better

Escape output before displaying it.

Best Practices

- Encode HTML output.
- Sanitize user content.
- Use modern web frameworks with built-in protections.

---

13. Use Secure Random Numbers

Predictable random numbers can weaken security.

Unsafe

```
import random
token = random.randint(1,1000)
```

Better

```
import secrets
token = secrets.token_hex(16)
```

Use cryptographically secure generators for:

- Password resets
- Session tokens
- Authentication codes

---

14. Test for Security Vulnerabilities

Perform regular testing.

Common Tests

- Penetration testing
- Vulnerability scanning
- Dependency scanning
- Authentication testing

Use the OWASP Top 10 as a checklist for common web application vulnerabilities.

Conclusion

Secure coding is not a single task but an ongoing process. The most important habits are:

1. Validate all input.
2. Use parameterized database queries.
3. Hash passwords securely.
4. Protect secrets and credentials.
5. Apply least privilege.
6. Use HTTPS and encryption.
7. Handle errors carefully.
8. Keep software updated.
9. Review code regularly.
10. Test for vulnerabilities.

By following these practices, developers can significantly reduce security risks and build safer, more reliable software.