Secure Python Basics

Python Course · Module 10

ElnurBDa

Module 10 — Security & Privacy Basics

In this module we connect your Python, database, and web skills to **security**:

- Trust boundaries & input validation
- Common pitfalls in CLI, database, and web apps
- SQL injection and **parameterized queries**
- Handling secrets and sensitive data (passwords, tokens, logs)

Goal: build a **security mindset** for all the code you write next.

Why Security Matters in Everyday Code

Even “small” scripts can:

- Delete or leak files
- Corrupt databases
- Expose personal data (names, emails, grades)

Realistic risks:

- Users type unexpected / malicious input
- Someone calls your API with weird parameters
- Logs or config files accidentally contain passwords or tokens

Security is not just about hacking tools – it’s about **writing code that behaves safely** under bad input.

Trust Boundaries — Who Do You Trust?

Every program has **trust boundaries**:

- Data you fully control (constants, internal helpers)
- Data from **users** (CLI `input()`, HTML forms, API requests)
- Data from **external systems** (files, databases, 3rd-party APIs)

Rule of thumb:

> Anything that crosses a boundary into your program is **untrusted** until checked.

We’ll see how this affects:

- Input validation
- SQL queries
- Logging and error messages

Validating User Input (CLI)

Example: simple CLI for age input.

Unsafe style (no checks, will crash on bad data):

```python
age = int(input("Enter your age: "))
print("Next year:", age + 1)
```

Safer style (validate + handle errors):

```python
def ask_age():
    text = input("Enter your age: ").strip()
    if not text.isdigit():
        print("Please enter only digits.")
        return None

    age = int(text)
    if age <= 0 or age > 130:
        print("Please enter a realistic age.")
        return None

    return age

age = ask_age()
if age is not None:
    print("Next year you will be", age + 1)
```

——————————— [finished with error] ———————————

```
Enter your age: Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermYqlMmk/snippet.py", line 14, in
<module>
    age = ask_age()
  File "/tmp/nix-shell-61594-0/.presentermYqlMmk/snippet.py", line 2, in
ask_age
    text = input("Enter your age: ").strip()
           ~~~~~^^^^^^^^^^^^^^^^^^^^
EOFError: EOF when reading a line
```

Pattern:

- **Validate** format and range
- **Fail gracefully** with a clear message

Validating Web/API Input (Concept)

In web apps (like your Module 9 To-Do or Student Manager):

- Every field from a form or JSON body is **untrusted**.
- You must check: **type**, **length**, **allowed characters**, **business rules**.

Examples of validation rules:

- `username`: 3–32 chars, letters/numbers/underscores only
- `email`: contains `@` and a domain (use a simple regex or library)
- `title` (task): non-empty, max 200 chars

Idea: keep validation **close to the boundary** (in the route handler / controller) so the rest of your code can assume data is clean.

SQL Injection – The Problem

From Module 7 you know how to build SQL queries. Now we look at a **classic attack**: SQL injection.

Dangerous pattern:

```python
def find_user_by_name(conn, username):
    # ❌ vulnerable: user can inject SQL
    sql = f"SELECT id, username FROM users WHERE username = '{username}'"
    cur = conn.cursor()
    cur.execute(sql)
    return cur.fetchall()
```

If `username` is:

```
alice' OR 1=1 --
```

the resulting SQL becomes:

```sql
SELECT id, username FROM users WHERE username = 'alice' OR 1=1 --'
```

This may return **all users** instead of just one.

SQL Injection — Safe Pattern with Parameters

Use **parameterized queries** (you started doing this in Module 7).

```python
import sqlite3

def find_user_by_name(conn, username):
    sql = "SELECT id, username FROM users WHERE username = ?"
    cur = conn.cursor()
    cur.execute(sql, (username,))
    return cur.fetchall()

conn = sqlite3.connect(":memory:")
print("This is a demo of parameterized queries, not full setup.")
```

[finished]

```
This is a demo of parameterized queries, not full setup.
```

Key rules:

- **Never** build SQL with `+` / f-strings for untrusted input
- Always use `?` placeholders (`%s` in some drivers) and pass **parameters separately**
- Let the database driver escape dangerous characters safely

This same pattern applies to INSERT, UPDATE, DELETE.

Validating Before Hitting the Database

Combine **validation** + **parameters** for defense in depth.

Example: new student for the Student Manager (Module 9 mini project):

```
def validate_student_payload(data):
    name = data.get("name", "").strip()
    email = data.get("email", "").strip()
    course = data.get("course", "").strip()

    if not (3 <= len(name) <= 60):
        raise ValueError("Name must be 3–60 characters.")
    if "@" not in email or "." not in email:
        raise ValueError("Invalid email format.")
    if not course:
        raise ValueError("Course is required.")

    return name, email, course
```

Then in your API endpoint or CLI:

1. Call `validate_student_payload()`
2. Use a **parameterized INSERT** into SQLite
3. Catch `ValueError` and return a friendly message / status code

Logging — Helpful vs Dangerous

Logging is useful for debugging and incident response, but it’s easy to leak secrets.

Common mistakes:

- Logging raw passwords or tokens
- Logging entire HTTP bodies with credentials
- Logging full database rows with sensitive columns (e.g., grades, hashes)

Guidelines:

- **Never log passwords** (even hashes are rarely needed in logs)
- Don’t log API keys, JWTs, or session cookies
- Redact or summarize sensitive fields

Bad:

```
print("Login request:", email, password)
```

Better:

```
print("Login attempt for:", email)
```

Storing Passwords — Concepts

You should almost **never** store plain-text passwords.

Concepts (high level, without full crypto details):

- Use a **hash function designed for passwords** (bcrypt, Argon2, PBKDF2)
- Store only the **salted hash**, never the raw password
- On login, hash the provided password and compare to stored hash

In this course:

- We don’t implement full auth, but you should know:
  - `hashlib.sha256` is **not ideal** for real passwords (too fast)
  - Libraries like `passlib` or frameworks handle this better

Takeaway: if you ever see code writing password strings directly to a database or CSV, that’s a **red flag**.

Config & Secrets Management (Concept)

Where do API keys, DB URLs, and secrets live?

Better options than hard-coding:

- Environment variables (e.g., `os.environ["API_KEY"]`)
- `.env` files loaded via helper libraries (in real projects)
- Separate config files with **restricted permissions**

Bad:

```
API_KEY = "super-secret-key-123"
```

Better (conceptually):

```
import os
API_KEY = os.environ.get("API_KEY")
```

For this course: avoid committing real secrets to code, and keep demo values clearly fake.

Error Messages & Information Leakage

Error messages help users, but can help attackers too.

Examples:

- Revealing exact SQL queries or stack traces in production
- Returning “User exists, but password is wrong” vs a generic message

Guidelines:

- Show **friendly, minimal** messages to end users
- Log detailed technical info to files (with **no secrets**) for developers
- Avoid exposing framework versions, internal paths, or SQL details in HTTP responses

In your CLI/web mini projects, aim for:

> “Something went wrong, please try again or contact support.”
>
> plus a more detailed log entry for yourself.
