Secure Coding Practices

Personal Banking Management System

python_owasp10_demo.zip

📋 Agenda

- 🔒 Introduction: Why Secure Coding Matters in Banking
- 🎯 Deep Dive: OWASP
  - ✓ What is the OWASP Top 10?
  - ✓ Demonstration and mitigation for each vulnerability
  - ✓ Best practices for secure coding
- 🛡️ Core Defense Mechanisms
- 💡 Best Practices & Pro-Level Tips
- ✅ Conclusion & Key Takeaways

🏛️ Why Secure Coding Matters

🤝 Trust is Everything

Customers trust us with their financial data. A single breach can destroy that trust and the bank's reputation.

💰 High Stakes

The target is money. Banking applications are prime targets for attackers.

📋 Regulatory Compliance

Financial institutions face strict regulations (PCI-DSS, GDPR) with severe penalties for non-compliance.

💻 It Starts with Code

The first line of defense against cyber threats is secure, well-written code.

OWASP Top 10: A01 Broken Access Control

Understanding and Fixing One of the Most Critical Web Vulnerabilities

FIL Fresher Training

The Core Problem - What is Broken Access Control?

HIGH RISK

🏢 Real-World Analogy

Imagine you live in an apartment building where your key can open any door, as long as you know the apartment number. You should only be able to open your own door!

Broken Access Control is exactly that. It's a security flaw where the server fails to check if the user making a request is actually authorized to access the data they are asking for.

⚠️ The Core Mistake

The application checks if the data exists but forgets to check if the current user owns that data.

The Vulnerable Code - A Step-by-Step Breakdown

HIGH RISK

Let's look at the insecure code. The logic is dangerously simple.

```
@app.get("/users/{user_id}") async def get_user(user_id: str): if user_id in users:
return users[user_id] return {"error": "User not found"}
```

What happens when a user requests /users/2:

- **Step 1:** The app receives a request for the user's data
- **Step 2:** Function takes user_id ("2") directly from URL
- **Step 3:** Code checks if user "2" exists in the database
- **Step 4:** Code sends back the data

🚨 The Fatal Flaw

Notice what's missing? The code never asks **"Who is making this request?"**. It blindly trusts that whoever is asking for user "2" is allowed to see that information.

The Attack in Action

HIGH RISK

This flaw is extremely easy to exploit:

📋 Attack Scenario:

- **Step 1:** Abinash (User ID: 1) logs in and visits `/users/1` - Gets his own data ✅
- **Step 2:** Abinash gets curious and changes the URL to `/users/2`
- **Step 3:** Server receives request, finds user "2" (Rahul), and sends back the data

💥 The Breach

Abinash can now see Rahul's name and email without permission. This is a serious data leak!

Impact: Any logged-in user can access any other user's data just by changing the URL parameter.

The Secure Code - How to Fix It

HIGH RISK

The fix involves adding one crucial step: verifying the user's identity against the requested resource.

```
# Get current user (in real app, from secure token/session) async def
get_current_user_id(): return "1" # Simulating Abinash as current user
@app.get("/users/{user_id}") def get_user(user_id: str, current_user_id:
Annotated[str, Depends(get_current_user_id)]): # 🛡 THE FIX! Security check before
accessing data if user_id != current_user_id: raise HTTPException(status_code=403,
detail="Unauthorized") if user_id in users: return users[user_id] return {"error":
"User not found"}
```

🔐 Two Key Additions:

- **User Identification:** Get the ID of the currently logged-in user
- **Authorisation Check:** Verify the requested resource belongs to the current user

The Secure Flow - Attack Prevented

HIGH RISK

Let's replay the attack against our new, secure code:

✅ Legitimate Access (Abinash requests /users/1):

- Check: user_id ("1") == current_user_id ("1")? **TRUE**
- Result: Access granted, Abinash gets his own data

🛡️ Attack Blocked (Abinash requests /users/2):

- Check: user_id ("2") == current_user_id ("1")? **FALSE**
- Result: 403 Unauthorized error returned immediately

🎉 Success!

The server has protected Rahul's data. The attack has been completely blocked!

Key Takeaways

HIGH RISK

🎯 Critical Security Principles:

- **Never Trust User Input:** Especially data coming directly from URLs. Always assume it could be malicious.
- **Authentication vs. Authorization:**
  - Authentication = Proving who you are (logging in)
  - Authorisation = Checking if you have permission to access something
- **Enforce Ownership:** For any resource, always verify the currently logged-in user owns that resource before returning it.

💡 Remember

The simple question **"Does this user own this data?"** can prevent massive security breaches!

🚀 Next Steps

Practice implementing these checks in your own code. Make authorization verification a standard part of your development process.

🔐 A01: Broken Access Control - PYTHON

HIGH RISK

What it is: This vulnerability happens when an attacker can access resources or perform actions they aren't authorized for.

Demo: A user can access another user's data by simply changing the ID in the URL.

❌ VULNERABLE

```
from fastapi import FastAPI, HTTPException # type: ignore

app = FastAPI(title="A01 Broken Access Control - Insecure")

# Dummy user data
users = {
    '1': {'name': 'Abinash', 'email': 'abinash@example.com'},
    '2': {'name': 'Rahul', 'email': 'rahul@example.com'}
}

@app.get("/user/{user_id}")
async def get_user(user_id: str):
    if user_id in users:
        return users[user_id]
    raise HTTPException(status_code=404, detail="User not found")
```

✅ SECURE

```
from typing import Annotated
from fastapi import FastAPI, HTTPException, Depends # type: ignore

app = FastAPI(title="A01 Broken Access Control - Secure")

# Dummy user data and a dependency to simulate an authenticated user
users = {
    '1': {'name': 'Abinash', 'email': 'abinash@example.com'},
    '2': {'name': 'Rahul', 'email': 'rahul@example.com'}
}

async def get_current_user_id():
    # In a real app, this would come from a token/session
    return "1"

@app.get("/user/{user_id}")
async def get_user(user_id: str, current_user_id: Annotated[str, Depends(get_current_user_id)]):
    # Secure: Ensure requested resource belongs to the current user
    if user_id != current_user_id:
        raise HTTPException(status_code=403, detail="Unauthorized")
    if user_id in users:
        return users[user_id]
    raise HTTPException(status_code=404, detail="User not found")
```

Understanding Cryptographic Failures

OWASP A02: How to Properly Protect Sensitive Data in Your Code

What's the Big Deal? 🕵️

HIGH RISK

What is a Cryptographic Failure?

A security vulnerability occurs when we fail to properly protect sensitive information like passwords, credit card numbers, or personal health records.

Why Should You Care?

Imagine your application's database is like a treasure chest. If you leave it unlocked, anyone who finds it can take everything inside.

Insecure: Storing a user's password as 'password123'

The Risk: If a hacker gains access to your database, they instantly have real passwords for every user. This leads to identity theft, financial loss, and destroys user trust.

The WRONG Way: Plain Text Passwords ❌

HIGH RISK

Let's look at a dangerously simple way to store passwords:

```
# Passwords are stored in plain text user_passwords = { 'alice': 'password123' } def
check_password(user, password): # Direct comparison of stored text return user in
user_passwords and user_passwords[user] == password print(f"Login successful:
{check_password('alice', 'password123')}")
```

What's Happening Here?

The password 'password123' is stored exactly as the user typed it. It's fully readable.

What's Happening Here?

The password 'password123' is stored exactly as the user typed it. It's fully readable.

> ***Analogy:*** *This is like writing your ATM PIN on the back of your debit card. If you lose the card, you lose your money.*

The RIGHT Way: Password Hashing ✅

HIGH RISK

Instead of storing the password, we store a "hash" of the password.

What is Hashing?

Hashing is a one-way process that transforms any data into a unique, fixed-length string of characters.

One-Way: You can't reverse it. You can turn 'password123' into a hash, but you can't turn the hash back into 'password123'.

> ***Analogy:*** *It's like turning a cow into a hamburger. You can't turn the hamburger back into the original cow.* 🐄 ➡️ 🍔

The Secret Ingredient: Salting 🧂

HIGH RISK

There's a problem with simple hashing: if 1,000 users all have the password 'password123', they will all have the same hash.

The Solution: Salting!

A "salt" is a unique, random string added to each password before hashing.

'password123' + 'random_salt_A' → Hash A

'password123' + 'random_salt_B' → Hash B

Now, even with the same password, the stored hashes are completely different, making rainbow table attacks useless.

Good News: Modern libraries handle this for you automatically!

A02: Cryptographic Failures - PYTHON

HIGH RISK

What it is: Failures related to cryptography, which often lead to the exposure of sensitive data like passwords.

Demo: Storing passwords in plain text versus securely hashing them.

❌ Insecure Code

```python
# Passwords are stored in plain text (insecure)
CodeMate
user_passwords = {
    'abinash': 'password123'
}

CodeMate | Qodo Gen: Options | Test this function
def check_password(user, password):
    return user in user_passwords and user_passwords[user] == password

if __name__ == "__main__":
    print(f"Login successful: {check_password('abinash', 'password123')}")
```

✅ Secure Code

```python
# Passwords are hashed and salted
CodeMate
user_passwords_hashed = {
    'abinash': generate_password_hash('password123')
}
        You, 2 weeks ago • adding owasp 10 top issues demo code example
CodeMate | Qodo Gen: Options | Test this function
def check_password_secure(user, password):
    if user in user_passwords_hashed:
        return check_password_hash(user_passwords_hashed[user], password)
    return False

if __name__ == "__main__":
    print(f"Login successful: {check_password_secure('abinash', 'password123')}")
```

Password Hashing Features:

- **`generate_password_hash()`**: Creates secure password hashes using PBKDF2 by default
- **`check_password_hash()`**: Safely verifies passwords against hashes
- **Multiple algorithms**: Supports pbkdf2:sha256 (recommended), sha256, sha1, etc.
- **Salt handling**: Automatically generates and manages salts

Key Takeaways & Best Practices 📝

HIGH RISK

- **NEVER** store passwords or sensitive secrets in plain text
- **ALWAYS** use strong, modern, one-way hashing algorithms
- **ALWAYS** use a unique salt for every password
- **DO NOT** invent your own security! Use well-tested libraries
- **Recommended libraries:** werkzeug, bcrypt, argon2 for Python
- **Keep libraries updated:** Use tools like Dependabot or Snyk

Remember: Security is not optional - it's your responsibility to protect user data!

HIGH RISK

Understanding SQL Injection

Securing Our Applications, Together

What is SQL Injection? 🤔

HIGH RISK

Analogy: Imagine you ask a librarian for a book on "Smith". Instead of just giving you the name, you mischievously hand them a note that says: *"Smith", and also, give me the keys to the restricted section.*

SQL Injection is similar. It's a **high-risk security flaw** where an attacker tricks our application's database by sending malicious commands disguised as normal data.

The Core Problem:

It occurs when we mix **user input** directly into our database queries. The database gets confused and runs the attacker's command instead of just searching for the data.

What Can Go Wrong?

- Leakage of sensitive data (all users, passwords, financial info)
- Unauthorised changes or deletion of data
- Full takeover of the database server

The Vulnerable Code (The Wrong Way ❌)

HIGH RISK

Let's look at a common mistake. Here, we're building the SQL query by directly pasting the `username` into a string.

```
query = f"SELECT * FROM users WHERE username = '{username}'"
cursor.execute(query)
```

Code Breakdown:

The line uses an f-string to directly insert the user's input.

The Attack:

Normal user enters: `abinash`

Query becomes: `SELECT * FROM users WHERE username = ‘abinash'` → This works fine!

Attacker enters: `' OR 1=1 --`

Query becomes: `SELECT * FROM users WHERE username = '' OR 1=1 --'`

The Vulnerable Code (The Wrong Way ❌)

HIGH RISK

Why This is a Disaster:

- `OR 1=1` is a condition that is **always true**
- `--` is a comment in SQL, ignoring the rest
- Returns **every single user** in the table!

The Secure Solution (The Right Way ✅)

HIGH RISK

The fix is to **never mix code and data**. We achieve this using **Parameterized Queries**.

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

Code Breakdown:

- **The Plan:** Query is now a template with placeholder: `?`
- **The Data:** User input passed separately in `execute` method

Why This Works:

The database receives the command (`SELECT...`) and the data (`' OR 1=1 --`) in two separate steps:

1. Prepares the "plan" for the `SELECT` query
2. Treats user input **purely as data**

It will literally search for a user named `' OR 1=1 --` - which doesn't exist, so the attack fails!

A03: Injection (SQL Injection) - PYTHON

HIGH RISK

What it is: Occurs when an attacker sends untrusted data that gets interpreted as a command or query, like SQL injection.

Demo: Using a malicious string to trick a database query into returning all users instead of just one.

❌ VULNERABLE

```python
def get_user_by_username(username: str):
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE users (id INT, username TEXT)")
    cursor.execute("INSERT INTO users VALUES (1, 'Abinash'), (2, 'Rahul')")

    # Insecure: Using user input directly in the SQL query
    query = f"SELECT * FROM users WHERE username = '{username}'"
    cursor.execute(query)
    user = cursor.fetchall()
    conn.close()
    return user

if __name__ == "__main__":
    # Attacker input: ' OR 1=1 --
    CodeMate
    attacker_input = "' OR 1=1 --"
    print(f"Users found: {get_user_by_username(attacker_input)}")
```

```sql
SELECT * FROM users WHERE username = '' OR 1=1 --'
```

- `OR 1=1` is **always true**, so it returns **all users**, ignoring security.
- `--` makes the rest of the query a comment.

👉 This means an attacker can **bypass checks** and even possibly mess with the database.

✅ SECURE

```python
def get_user_by_username_safe(username: str):
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE users (id INT, username TEXT)")
    cursor.execute("INSERT INTO users VALUES (1, 'Abinash'), (2, 'Rahul')")

    # Secure: Using parameterized queries
    query = "SELECT * FROM users WHERE username = ?"
    cursor.execute(query, (username,))
    user = cursor.fetchall()        You, 2 weeks ago • adding owasp 10 top issues demo code
    conn.close()
    return user

if __name__ == "__main__":
    # Attacker input: ' OR 1=1 --
    CodeMate
    attacker_input = "' OR 1=1 --"
    print(f"Users found: {get_user_by_username_safe(attacker_input)}")

# The secure version uses parameterized queries to prevent SQL injection attacks.
    attacker_input = "Abinash"
    print(f"Users found: {get_user_by_username_safe(attacker_input)}")
```

💉 A03: Injection (SQL Injection) - PYTHON

HIGH RISK

Why is it good?

- The database treats the input as **data**, not as part of the SQL command.
- Even if an attacker tries `' OR 1=1 --`, it will just search for a username literally equal to that string — which doesn't exist.

👉 This **prevents SQL Injection** and keeps the database safe.

📝 Summary for Beginners

- **First code:** Unsafe → directly pastes user input into the query → vulnerable to hackers.
- **Second code:** Safe → uses placeholders (`?`) and parameters → user input is treated as data only, not part of the SQL logic.

🔑 **Rule of Thumb:**

Always use **parameterized queries (placeholders like `?`)** when working with SQL databases to avoid security risks.

Key Takeaways & FIL Best Practices

HIGH RISK

Golden Rule:

Treat all user input as untrusted. Separate your SQL commands from the data you put into them.

How to Stay Safe:

1. **Always Use Parameterized Queries:** The placeholder `?` (or `%s` in some libraries) is your best friend.
2. **Avoid Manual String Building:** Never build queries using `+`, f-strings, or `.format()` with user input.
3. **Use ORMs (Object-Relational Mappers):** Frameworks like Django or SQLAlchemy handle parameterization for you.
4. **Principle of Least Privilege:** Database user should have only the minimum necessary permissions.

A04 - Insecure Design

Thinking About Security from Day One

What Exactly is "Insecure Design"? 🤔

It's not about a single coding bug; it's a flaw in the plan.

Imagine building a house:

- Forgetting to design locks for the doors
- Placing windows where anyone can easily climb in
- Not planning for a fire escape

You can have the best bricks and paint (good code), but if the blueprint (design) is flawed, the house is not secure.

In software: Insecure Design means we didn't think about potential attacks and security requirements during the initial planning and design phases. We are reacting to problems later, instead of preventing them from the start.

Let's Break Down the Example: Password Reset

A classic feature, but a huge target for attackers!

The Insecure Design 👎

- User can request password reset unlimited times
- No mechanism to stop repeated attempts
- No protection against automation

The Attacker's Goal 😈

An attacker writes a simple script to try thousands of password combinations per minute. This is called a **Brute-Force Attack**.

Result: Eventually, the script will guess the right password, and the account is compromised.

The Secure Design Approach: Building a Fortress 🏰

Fix the blueprint, not just patch a crack in the wall

1. Rate Limiting

What it is: Limit the number of attempts a user can make in a given time.

> *Example: "You can only request a password reset 5 times per hour from this IP address."*

Why it works: Makes brute-force attacks incredibly slow and impractical.

2. Use Expiring, One-Time-Use Tokens

What it is: Email a special, temporary link (token) that works only once and expires quickly (e.g., 10 minutes).

Why it works: Even if an attacker intercepts an old email, the reset link is useless.

The Secure Design Approach (Continued...)

3. Proper Logging & Monitoring

What it is: Record every failed and successful login or reset attempt. Monitor for suspicious patterns.

Example: Alert triggered if 100 failed attempts for one account come from the same IP in one minute.

Why it works: Helps detect attacks in progress so we can respond by temporarily blocking suspicious IPs.

4. Multi-Factor Authentication (MFA)

What it is: Require a second piece of proof - usually a code from phone app or SMS.

Why it works: Even if attacker steals the password, they can't log in without the second factor. It's a powerful extra layer of security.

The Golden Rule: Architectural Fix vs. Code Fix

The most important concept to remember!

Simple Code Fix

```
if (attempts > 5) { return "error"; }
```

Architectural (Design) Fix

- Implementing full rate-limiting service
- Integrating email service for secure tokens
- Connecting to centralized logging system
- Integrating with MFA provider

Key Point: Insecure Design is fixed by changing the structure and logic of the system, not just a few lines of code.

Key Takeaways for You at FIL 🎯

🛡️ Security is Not an Afterthought

Think about security during every planning meeting. Don't wait until the code is written.

🔍 Think Like an Attacker

Ask yourself: "How could I break this feature? What could go wrong?" This is the beginning of Threat Modeling.

❓ Question Everything

If a design seems too simple, it might be insecure. Ask: "What happens if someone spams this button?" or "How do we protect this sensitive data?"

🤝 Security is a Team Sport

It's everyone's responsibility, from developers to testers to project managers. Collaborate and learn from your security champions!

Rate Limit - DEMO

A05: Security Misconfiguration

One of the most common web application vulnerabilities

What is Security Misconfiguration?

(The "Open Door" Problem)

> *Imagine you get a new high-tech front door for your house. It comes with a default password of "0000".*
>
> *If you forget to change this default password, you have the best door in the world, but anyone can walk right in!* 🚪

Security Misconfiguration is exactly that:

- Forgetting to change **default settings** (like `admin/admin`)
- Leaving "developer-only" features turned on in a live application
- Showing overly detailed error messages to the public

> It's not a flaw in the code's logic, but a mistake in its **setup**.

Our Focus: The Danger of "Debug Mode"

(The "Open Door" Problem)

Developers love **Debug Mode**. When an app crashes, it gives a detailed "crash report" called a **stack trace**.

- **For Developers (Good 👍):** It shows exactly where the code broke, what the variables were, and the path it took. It's super helpful for fixing bugs!
- **In Production (Very Bad 👎):** If an attacker sees this report, they get a blueprint of your application!

This crash report can reveal:

- **File paths:** `/var/www/app/main.py`
- **Frameworks & versions:** `FastAPI v0.95.1, Python 3.10`
- **Sensitive data:** Database details, API keys, or configuration secrets

> Rule #1: Never show a detailed crash report to the end-user.

Code Example: The Vulnerable App

(The "Open Door" Problem)

Here we have a simple FastAPI application. The developer has left debug mode turned on.

```
# ❌ VULNERABLE # Insecure: Debug mode is enabled, which can expose sensitive stack traces
app = FastAPI(debug=True) @app.get("/") def read_root(): # This code will crash on purpose to
show the debug page 1 / 0 return {"Hello": "World"}
```

What Happens?

- A user (or attacker) visits the website
- The code `1 / 0` causes a `ZeroDivisionError`
- Because `debug=True`, FastAPI shows the attacker a full, detailed error page instead of a generic "Something went wrong" message
- The attacker now has valuable information to plan their next move

The Solution: Simple & Secure Code

(The "Open Door" Problem)

The fix is incredibly simple but critically important.

```
# ✅ SECURE # Secure: Debug mode is disabled for production # Configuration should be managed from environment variables app = FastAPI(debug=False) @app.get("/") def read_root(): return {"Hello": "World"}
```

How it's fixed:

- **Set `debug=False`:** This is the main fix. Now, if the app crashes, users will see a generic and safe "Internal Server Error" message, revealing nothing.
- **Best Practice:** Don't just type `False`. This setting should be controlled by **environment variables**. This lets you have `debug=True` for local development but ensures it's automatically `False` on the live production server.

Key Takeaways & Your Responsibility

(The "Open Door" Problem)

- **Always Disable Debug Mode in Production:** This is non-negotiable. It's one of the first things an attacker checks for.
- **Don't Trust Defaults:** Whether it's a database, a framework, or a cloud service, always review and change the default security settings.
- **Use Environment Variables:** Manage sensitive configurations (like debug flags, database passwords, API keys) outside of your code using environment variables. This prevents secrets from being saved in your Git history.
- **Less is More:** Error messages shown to users should be simple and give no internal details.

> Remember: Security is not just about writing good code—it's also about configuring it correctly! 🔒

Security Misconfiguration - DEMO

A06: Vulnerable & Outdated Components

The Hidden Danger in the Code We Don't Write

The House Analogy: What Are "Components"?

Imagine you're building a house. You don't make the bricks, windows, or door locks yourself. You buy them from suppliers.

In software, we do the same! We use pre-built pieces of code called **libraries**, **packages**, or **components** to save time and effort.

- **Example**: Using the `requests` library in Python to fetch data from a website.

> The problem arises when the door lock you bought has a known defect that burglars can easily exploit. That's a vulnerable component! 😱

Why Is This a Huge Deal? The Real-World Risk

Using a library with a known security flaw is like leaving your front door open for attackers. They can:

- **Steal Sensitive Data**: Access user passwords, personal information, or financial details.
- **Take Over Your System**: Gain control of your server to launch other attacks.
- **Damage Company Reputation**: A security breach erodes user trust and can have massive financial consequences.

> A famous example is the 2017 Equifax breach, where attackers stole the data of 147 million people by exploiting a vulnerability in a common web framework.

A Practical Demo: Spotting the Danger in requirements.txt

In Python, we list our project's libraries (dependencies) in a file called `requirements.txt`. This tells everyone which "components" our house is built with.

Let's look at an example:

❌ Insecure Dependency

```
# requirements.txt # This version has a known security vulnerability!
requests==2.20.0
```

> An attacker knows exactly how to break into systems using this specific version.

The Simple Fix: Updating to a Secure Version

The good news is that developers of these libraries release patches and new versions to fix security holes. Our job is to use them!

✅ Secure Dependency

```
# requirements.txt # This is the latest stable and secure version.
Phew! requests==2.31.0
```

By simply updating the version number, we've fixed the vulnerability.

How Do We Stay Safe? Our Mitigation Strategy

You can't manually check every library for vulnerabilities. It's impossible! So, we use automated tools to do the hard work for us.

1. Scan Your Dependencies Regularly

- Use tools that scan your project and compare your libraries against a database of known vulnerabilities.
- **Great Tools**:
  - **Snyk**: Integrates with your code repository (like GitHub) and alerts you automatically.
  - **GitHub Dependabot**: Automatically scans your code on GitHub and even creates pull requests to update vulnerable packages for you!
  - **pip-audit**: A simple command-line tool to run a quick check.

2. Update Promptly and Safely

- When a tool alerts you to a vulnerability, don't ignore it!
- Plan to update the library to the recommended secure version.
- **Always test** after updating to ensure nothing in your application broke.

Your Key Takeaways as a Developer

1. Be Aware: Know which third-party libraries your project depends on.

2. Automate: Set up automated scanning tools like Snyk or GitHub Dependabot from day one. **Don't rely on manual checks.**

3. Update: Make updating dependencies a regular part of your development cycle.

4. Remove: If a library is no longer used in the project, remove it completely to reduce your "attack surface."

> Security is everyone's responsibility, and it starts with the code you use.

A06: Vulnerable and Outdated Components

> **What it is:** Using libraries, frameworks, or other software components with known, unpatched security vulnerabilities.

Demo: Using an old version of a library in requirements.txt

❌ Insecure Dependency

```
# requirements.txt requests==2.20.0 # This version has a known vulnerability
```

✅ Secure Dependency

```
# requirements.txt requests==2.31.0 # Latest stable and secure version
```

Mitigation

Use tools like `pip-audit` or `Snyk` to regularly scan and update project dependencies.

DEMO - A06: Vulnerable and Outdated Components

🔐 A07: Identification & Authentication Failures

Keeping Our Digital Doors Securely Locked

🏢 What Are We Talking About?

Let's start with a simple analogy:

Your application is a secure building.

- **Identification** is telling the guard your name. *("Hi, I'm Alex.")*
- **Authentication** is proving you are who you say you are. *(Showing your ID card.)*
- **Session Management** is the key card the guard gives you to access different rooms for a limited time.

🏢 What Are We Talking About?

What is the "Failure"?

This vulnerability happens when the "guard" (our application) is careless:

- It doesn't check IDs properly
- It accepts fake or easily guessable IDs
- It doesn't take back the key card when you leave

The result? Unauthorized people can get in and access sensitive data.

⚠️ Common Example #1: The "password123" Problem

Allowing Weak or Common Passwords

Attackers don't guess passwords one by one. They use automated scripts that try millions of common passwords (like `password`, `123456`, `qwerty`) against thousands of accounts.

Why this is a huge risk:

- It's the easiest way for an attacker to walk right in
- Users often reuse the same weak password everywhere

⚠️ Common Example #1: The "password123" Problem

✅ How to Fix It:

- **Enforce Strong Password Policies:** Minimum length (e.g., 12+ characters), mix of cases, numbers, and symbols
- **Prevent Common Passwords:** Block users from setting passwords like "password123" or "company_name"
- **Check Against Breached Password Lists:** Use services to see if a user's chosen password has appeared in a known data breach

🎫 Common Example #2: The Lingering Key Card

Not Invalidating a User's Session After They Log Out

The Scenario:

1. A user logs in on a public computer and gets a session token (their "key card")
2. They click "Logout"
3. **The Failure:** The application doesn't invalidate the token. It's still active!
4. An attacker finds this token in the browser's history and uses it to access the user's account

🎫 Common Example #2: The Lingering Key Card

✅ How to Fix It:

- **Server-Side Logout:** When a user logs out, the server must immediately kill the session and blacklist the token so it can't be used again
- **Implement Session Timeouts:** Automatically log users out after a period of inactivity (e.g., 15-30 minutes)

🔨 Common Example #3: The Persistent Attacker

Not Protecting Against Brute-Force Attacks

A **brute-force attack** is when an attacker uses a script to try thousands of password combinations for a single account over and over again.

Without protection, they can keep trying until they succeed.

🔨 Common Example #3: The Persistent Attacker

✅ How to Fix It:

- **Account Lockouts:** Lock an account for a few minutes after 5-10 consecutive failed login attempts
- **Rate Limiting:** Limit how many login attempts can be made from a single IP address within a certain time frame (e.g., max 100 attempts per hour)
- **Use CAPTCHA:** Implement "I'm not a robot" tests after a few failed attempts to block automated scripts

🔐 Common Example #4: The Single Lock Problem

Not Implementing Multi-Factor Authentication (MFA)

A password is a single lock on the door. If it gets stolen, the attacker is in. **MFA adds a second, different kind of lock.**

MFA combines something you know with something you have:

- **Know:** Your password
- **Have:** A code from your phone app (Google Authenticator), an SMS message, or a physical security key

Why it's a game-changer: Even if an attacker steals your password, they can't log in without physical access to your phone or security key.

✅ Recommended Security Checklist

For Authentication:

- **DON'T** allow weak or common passwords
- **DO** invalidate sessions immediately upon user logout
- **DO** protect against brute-force attacks with account lockouts and rate limiting
- **DO** implement Multi-Factor Authentication (MFA) whenever possible

For Tokens & Sessions:

- Use **short-lived access tokens** (e.g., 5-15 minutes)
- Use **long-lived, securely stored refresh tokens**
- Use **Secure and HttpOnly flags** on cookies

A07: Identification and Authentication Failures

> **What it is:** Weaknesses in how a user's identity, authentication, and session are managed.

Common Examples

- Allowing weak or common passwords (e.g., "password123")
- Not invalidating a user's session after they log out
- Not protecting against brute-force attacks with account lockouts or rate limiting
- Not implementing multi-factor authentication (MFA)

✅Recommended security improvements & checklist

The demo intentionally shows basic behavior. For production, consider implementing the following:

From the authentication checklist (common pitfalls to avoid):

- Don't allow weak or common passwords.
- Invalidate a user's session after they log out (support token revocation/blacklist).
- Protect against brute-force: account lockouts, CAPTCHA, and rate limiting.
- Implement Multi-Factor Authentication (MFA).

Token & session best practices

- Use short-lived access tokens (e.g., 5–15 minutes) + refresh tokens.
- Keep refresh tokens highly protected and store them server-side or in secure httpOnly cookies.
- Implement a token revocation/blacklist (e.g., store revoked token jti values in Redis).
- Rotate signing keys periodically and support key IDs (`kid`) in token header.
- Log auth events (issued, refreshed, revoked) for auditability.
- Use HTTPS everywhere, set `Secure` and `HttpOnly` flags on cookies.

Hardening cryptography

- Prefer RS256/ES256 for distributed systems (private key signs, public verifies).
- When using HS256, keep the secret long and random (>= 32 bytes).

Common Ways the "Bad Guys" Get In 🕵️

These are some of the most common mistakes that can leave our digital doors wide open.

- **Weak Passwords:** Allowing passwords like `"password123"` or `"123456"`. It's like leaving the key under the doormat.
- **"Ghost" Sessions:** When a user logs out, but their session key still works. This allows an attacker to reuse the old key to get back in.
- **Letting Them Guess Forever:** Not locking an account or slowing down login attempts after too many wrong guesses. This invites automated "brute-force" attacks.
- **Single-Lock Security:** Not using **Multi-Factor Authentication (MFA)**. A password alone is just one lock on the door; MFA adds a second, like a deadbolt or a security chain.

A Quick Look at Cryptography

When we talk about "secrets" and "keys," we mean it.

- **Strong Algorithms:** We prefer modern, secure algorithms like `RS256` or `ES256`.
- **Long & Random Secrets:** When using simpler algorithms like `HS256`, the secret key must be long (at least 32 bytes) and completely random. A weak secret makes the whole system weak.

Think of it as the difference between a simple padlock and a complex bank vault lock. We always choose the vault lock. 🔐

Building secure applications is everyone's responsibility. Let's get it right from day one!

Authentication is the Front Door:

It's the first line of defense. Treat it with respect.

Passwords Are Not Enough:

Always think about implementing MFA.

Sessions Need a Clean End:

Always invalidate sessions completely on logout.

Trust No One:

Assume attackers are constantly knocking. Implement rate limiting and account lockouts.

A08 - Understanding Software & Data Integrity Failures

The Danger of Insecure Deserialization in Python

🤔 First, What is Serialization?

Think of serialization as packing a complex object into a simple package for shipping.

- You take a Python object (dictionary, list, or custom class instance) from memory
- You convert it into a flat stream of bytes or a string
- This "package" can now be easily:
  - Stored in a file or database
  - Sent over a network to another application

🧱 ***Analogy:***

It's like disassembling a Lego model, putting all the pieces and instruction manual into a box to mail to a friend.

📬 And What is Deserialization?

Deserialization is the unpacking process.

- You receive the stream of bytes or a string
- You convert it back into the original, live Python object in memory

🧱 *Analogy:*

Your friend receives the box, opens it, and uses the instruction manual to rebuild the exact same Lego model.

Key Point:

Both processes rely on "instructions" to reconstruct the original object correctly.

😈 The Vulnerability: The "Evil" Instruction Manual

What if the package wasn't from a trusted friend?

What if an attacker swapped the friendly instruction manual with a malicious one?

This is Insecure Deserialization:

- Happens when your application deserialises data from untrusted sources
- Sources include: user input, cookies, API requests
- No proper validation or checks are performed

The Risk:

If the "instruction manual" can contain commands, the attacker can make your program do anything they want!

🥒 The Dangerous Tool: Python's pickle Module

Its Power:

- Can serialize almost any Python object
- Includes functions and custom classes
- Saves complete "instructions" for reconstruction

Its Danger:

- Can serialise executable instructions
- Blindly executes instructions during deserialization
- Trusts the "instruction manual" completely

⚠️ Critical Point:

pickle doesn't distinguish between data and code - it treats everything as instructions to be executed!

❌ The Attack: pickle Deserialization (Vulnerable Code)

An attacker doesn't send you data. They send you a command disguised as data.

```
import pickle
import base64

# Attacker creates malicious payload to run command on server
malicious_payload = b"c__builtin__\nexec\n(S'import os; os.system(\"echo hacked\")'\ntR."

# Payload encoded for network transmission
encoded_payload = base64.b64encode(malicious_payload)

# INSECURE STEP: Server deserializes without validation
decoded_data = base64.b64decode(encoded_payload)

# pickle.loads() executes the malicious command!
pickle.loads(decoded_data)  # This will print "hacked"
```

What happened?

pickle.loads() didn't just rebuild data - it saw the command and EXECUTED it, leading to Remote Code Execution (RCE)!

✅ The Secure Alternative: The json Module

JSON is much safer for handling external data.

Why is JSON safe?

- Designed as a data-interchange format only
- No concept of executable code
- Cannot represent functions or commands

JSON understands only simple data types:

- Strings ("hello")
- Numbers (123)
- Booleans (true, false)
- Lists ([1, 2, 3])
- Objects ({"key": "value"})

Key Advantage:

JSON treats everything as data, never as executable code!

✅ The Defense: json Deserialization (Secure Code)

Let's see what happens when the attacker tries the same trick with JSON:

```
import json

# Attacker provides string that LOOKS like a command
malicious_payload = '{"command": "import os; os.system(\\"echo hacked\\")"}'

# SECURE STEP: Server uses json.loads() to parse payload
data = json.loads(malicious_payload)

# NO EXECUTION: Program loads string into dictionary
# Malicious string treated as plain text, not command
print(f"Data parsed safely: {data}")

# Output: Data parsed safely: {'command': 'import os; os.system("echo hacked")'}
```

What happened?

json.loads() correctly identified the payload as a dictionary. The malicious code was treated as simple text - no code was executed!

🏆 Key Takeaways & Best Practices

❌ Never Use pickle for Untrusted Data

Never deserialize with pickle if data comes from external sources you don't control 100% (user input, API clients, cookies).

✅ Prefer Data-Only Formats

Always use strict data-interchange formats like JSON for service communication and data storage. It's the industry standard for a reason!

🎯 The Core Principle:

Your program should treat incoming data as just data, not as executable code.

JSON enforces this • pickle does not

A09: Security Logging and Monitoring Failures

Why Flying Blind is a Terrible Idea in Security

A Tale of Two Login Functions

Let's look at a code example. An attacker is trying to guess a user's password.

❌ VULNERABLE CODE

```
def login(username, password):
    # ... authentication logic ...
    is_successful = False

    if is_successful:
        return True
    else:
        # No log is created!!
        return False
```

Complete silence. We have no idea an attack is even happening.

✅ SECURE CODE

```
import logging
# ... config ...

def login(username, password):
    # ... authentication logic ...
    is_successful = False

    if is_successful:
        logging.info(f'Success')
        return True
    else:
        logging.warning(f'Failed login')
        return False
```

We can see! Every failed attempt is now recorded.

The Problem & The Risk

The Problem:

- The vulnerable code has **zero logging for failed login attempts**.
- An attacker can try thousands of passwords in a **"brute-force attack"**, and the activity is completely **invisible** to us.

The Risk: 🚨

- We have **no record of the attack** and no way to know it's happening.
- This lack of visibility means we **cannot respond**. We can't block the attacker, lock the account, or alert the user. It's a massive security flaw.

The Fix & The Benefit

What's Improved in the Secure Code?

- **Logging is Configured:** A simple logger is set up to record events with a timestamp, level (e.g., INFO, WARNING), and message.
- **Success is Logged:** logging.info records successful logins, which is useful for auditing user activity.
- **FAILURE is Logged:** This is the critical part! logging.warning records every failed login attempt. Multiple failures for the same user can now trigger an alert for a potential attack.

The Benefit: You Create an Audit Trail! 🏆

By logging these events, this data can be fed into monitoring systems to:

- **Detect** suspicious behavior in real-time.
- **Provide** crucial evidence for investigation after an incident.
- **Help you understand** attacker patterns to strengthen your defenses.

Key Takeaways: What Should We Log?

Think of logs as your system's security cameras. You need to point them at the important stuff.

Always log these security-critical events:

- 🔒 **Authentication:** Both successful and, more importantly, **failed login attempts**. Also, log password resets and changes.
- 🚫 **Authorization Failures:** When a user tries to access a page or resource they don't have permission for.
- ⚠️ **Input Validation Failures:** When your system detects and blocks potentially malicious input from a user (like a suspected SQL Injection attack).
- 🔑 **Key System Events:** Changes to user permissions, roles, or other critical configurations.

Summary & Questions

Remember These Three Things:

1. **Log What Matters:** Especially failures, errors, and access denials.
2. **Silence is Not Golden:** In security, silence is suspicion. No logs mean no visibility.
3. **Logs Are Your First Response Tool:** You can't fight an enemy you can't see.

A09: Security Logging and Monitoring Failures - PYTHON

What it is: Not having enough logging, or any monitoring, to detect and respond to security incidents.

Demo: A login function that fails silently vs. one with proper logging.

❌ VULNERABLE

```python
def login(username, password):
    # ... authentication logic ...
    is_successful = False
    if is_successful:
        return True
    else:
        # No log is created for the failed attempt
        return False
```

✅ SECURE

```python
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(m

def login(username, password):
    # ... authentication logic ...
    is_successful = False
    if is_successful:
        logging.info(f'Successful login for user: {username}')
        return True
    else:
        # Secure: Log failed attempts for monitoring
        logging.warning(f'Failed login attempt for user: {username}')
        return False

login('testuser', 'badpass')
```

- **What's the problem?** The login function has no logging at all for failed attempts. If an attacker tries to guess passwords for a user (a "brute-force" attack), this activity is completely invisible.
- **The Risk:** You will have no record of the attack and no way to respond. This lack of visibility is a major security flaw.

- **What's improved?**
  - **Logging is configured:** logging.basicConfig sets up a simple logger that includes a timestamp, the log level (e.g., INFO, WARNING), and the message.
  - **Success is logged:** logging.info records successful logins. This is useful for auditing and tracking user activity.
  - **Failure is logged:** logging.warning records failed login attempts. This is critical for security monitoring. Multiple failed attempts for the same user could trigger an alert for a potential attack.
- **The Benefit:** By logging these events, you create an audit trail. This data can be fed into monitoring systems to detect suspicious behavior in real-time and provides crucial information for investigating incidents after they occur.

A10 - Understanding Server-Side Request Forgery (SSRF)

A Common and Critical Web Security Vulnerability

What is SSRF? The "Trusted Imposter" Attack

> **Core Concept:** A vulnerability that tricks a server into making a request to a location that the attacker controls.

🧑‍💼 Analogy: The Corporate Assistant

Imagine asking your company's front-desk assistant (the Server) to call a phone number and report back. The assistant has access to internal phone lines that you don't. Instead of giving a public number, you provide the CEO's private office number (an Internal Resource). The assistant, trusting your input, makes the call from inside the secure network, bypassing normal secu

> In SSRF, your malicious URL is the "phone number," and the server is your "trusted assistant."

Code Deep Dive: A Vulnerable Image Fetcher

Demo: An endpoint that fetches an image without validation

```
@app.get('/fetch_image') async def fetch_image(url: str): try: response =
requests.get(url) # 🚨 DANGER ZONE return Response(content=response.content) except
Exception as e: raise HTTPException(status_code=500)
```

⚠️ Analysis:

- Takes URL directly from user input
- **requests.get(url)** is the danger zone
- Server trusts user input completely - ZERO validation

Exploiting the Flaw: What Can an Attacker Do?

Attack 1: Access Internal Services

url=http://127.0.0.1:8080/admin

Goal: Scan internal network for hidden admin panels

Attack 2: Steal Cloud Credentials

url=http://169.254.169.254/latest/meta-data/

Goal: Leak AWS secret keys and access tokens

Attack 3: Read Local Files

url=file:///etc/passwd

Goal: Exfiltrate sensitive files from server's disk

Why It Works: The Root Causes

No URL Validation: Application accepts any string as URL without safety checks

Internal Network Access: Server has firewall permissions to access localhost and internal machines

Metadata Service Access: Cloud servers have default access to metadata services

Protocol Abuse: HTTP libraries support multiple URL schemes (http://, file://, ftp://) without restrictions

Building a Strong Defense: Preventing SSRF ✅

Fundamental Rule: NEVER TRUST USER INPUT

Strategy 1: Use an Allow-List

Define strict list of safe, pre-approved domains. Validate hostname is on this list.

Strategy 2: Validate & Sanitize Input

Ensure URL starts with http:// or https://. Reject private IP ranges.

Strategy 3: Least Privilege

Configure firewall to prevent connections to internal resources.

Key Takeaways & Q&A

What is it? SSRF forces your server to make requests to unintended destinations

Why is it bad? Exposes internal services, steals cloud credentials, reads local files

What's the cause? Blindly trusting user-supplied URLs without validation

How do we fix it? Strict allow-list of permitted domains. Always validate and sanitize input

Any Questions?

A10: Server-Side Request Forgery (SSRF)

> **What it is:** A vulnerability that tricks a server into making a request to a location the user controls, which can be used to scan internal networks or access cloud provider metadata.

Demo: An endpoint that fetches an image from a user-provided URL without validation.

```python
import requests
from fastapi import FastAPI, HTTPException
from fastapi.responses import Response

app = FastAPI()

@app.get('/fetch_image')
async def fetch_image(url: str):
    # Insecure: Making a request to a user-supplied URL without validation
    try:
        response = requests.get(url)
        response.raise_for_status()
        return Response(content=response.content, media_type="image/jpeg")
    except requests.exceptions.RequestException as e:
        raise HTTPException(status_code=500, detail=f"Error fetching image: {e}")
```

❌ VULNERABLE

Attack Examples:

```
# Access internal services
curl "http://localhost:8000/fetch_image?url=http://127.0.0.1:22"

# Access cloud metadata (AWS)
curl "http://localhost:8000/fetch_image?url=http://169.254.169.254/latest/meta-data/"

# File system access (if supported)
curl "http://localhost:8000/fetch_image?url=file:///etc/passwd"
```

Vulnerabilities in Insecure Code:

1. **No URL Validation**: The application accepts any URL from the user without validation
2. **Internal Network Access**: Attackers can access internal services (e.g., `http://localhost:8080/admin`)
3. **Metadata Service Access**: Cloud metadata services can be accessed (e.g., `http://169.254.169.254/`)
4. **Port Scanning**: Attackers can scan internal network ports
5. **Protocol Abuse**: Can use different protocols like `file://`, `ftp://`, etc.

A10: Server-Side Request Forgery (SSRF) - PYTHON

```python
import requests
from fastapi import FastAPI, HTTPException
from fastapi.responses import Response
from urllib.parse import urlparse

app = FastAPI()

# Whitelist of allowed domains
ALLOWED_DOMAINS = {'images.example.com', 'static.example.com'}

@app.get('/fetch_image')
async def fetch_image(url: str):
    parsed_url = urlparse(url)

    # Secure: Validate the domain against a whitelist
    if parsed_url.hostname in ALLOWED_DOMAINS:
        try:
            response = requests.get(url)
            response.raise_for_status()
            return Response(content=response.content, media_type="image/jpeg")
        except requests.exceptions.RequestException as e:
            raise HTTPException(status_code=500, detail=f"Error fetching image: {e}
    else:
        raise HTTPException(status_code=403, detail="Forbidden domain")
```

✅ SECURE

Security Improvements:

1. **Domain Whitelisting**: Only allows requests to predefined trusted domains
2. **URL Parsing**: Uses `urlparse()` to extract and validate URL components
3. **Explicit Validation**: Checks hostname against `ALLOWED_DOMAINS` set
4. **Error Handling**: Returns appropriate HTTP 403 status for forbidden domains

Security Best Practices

1. **Use Allowlists**: Define trusted domains/IPs explicitly
2. **Network Segmentation**: Isolate application servers from internal resources
3. **Input Validation**: Validate and sanitize all user inputs
4. **Regular Updates**: Keep dependencies updated using tools like `pip-audit`
5. **Monitoring**: Log and monitor outbound requests
6. **Least Privilege**: Run applications with minimal required permissions

Core Defence Mechanisms

Input Validation
Validate all inputs at boundaries

Parameterized Queries
Separate code from data

Output Encoding
Encode data for safe rendering

Defense in Depth
Multiple security layers

Least Privilege
Minimum necessary permissions

Security Frameworks
Use built-in protections

Framework Security Reference

Security Control		Python (FastAPI)
Input Validation		Pydantic Models with validators
Parameterized Queries		SQLAlchemy ORM with text() binding
Authentication		FastAPI security utilities (OAuth2, JWT)
XSS Prevention		Starlette response escaping

💡 Best Practices & Pro-Level Tips

🏰 Defense in Depth

Use multiple security layers: validation, parameterized queries, encoding, WAF, etc.

👤 Least Privilege

Application database accounts should have minimum necessary permissions.

⚡ Use Security Frameworks

Leverage built-in security features. Don't reinvent the wheel.

🔄 Security as Process

Integrate security throughout the entire development lifecycle.

🎯 Key Takeaways

🚫 NEVER Trust User Input

Validate, sanitize, and encode all external data

📋 Use Parameterized Queries

Best defense against SQL Injection. No exceptions.

🔒 Encode Data on Output

Prevent XSS by encoding before browser rendering

🛡️ Strong Authentication

Secure login, session handling, and logout features

🎯 Remember

You are the First Line of Defense

Secure coding practices are the foundation of a secure banking application

Mini Project - Building a Secure Transaction Search API

Your Mission (30-45 Minutes)

Goal: Build a secure REST API endpoint that allows a user to search their transaction history.

Project Scope

Personal Banking Management System

```
GET /transactions/search
```

search-api-python.zip

Core Functionality

The API should allow searching transactions based on:

- ✓ Optional keyword parameter
- ✓ Optional date range parameters

Key Security Requirements

- Safe from SQL Injection attacks
- All API inputs must be validated for data integrity

Step 1: Basic API Structure

PYTHON

Set up the basic API endpoint structure. We'll address security vulnerabilities in subsequent steps.

Python (FastAPI)

```
@app.post(
    "/v1/search",
    response_model=list[schemas.SearchResult],
    status_code=status.HTTP_200_OK,
    tags=["search"],
    summary="Search by customer name (case-insensitive partial match, SQLi-safe)",
)
def search_by_name(payload: schemas.SearchByNameRequest, db: Session = Depends(get_db)):
    svc = services.SearchService(db=db)
    return svc.search_by_name(payload)
```

Step 2: Securing Against SQL Injection

PYTHON

Fix the most common security vulnerability: string concatenation in SQL queries.

Python Team (SQLAlchemy)

SQL Injection Protection Mechanisms

1. Parameterized Queries via ORM:

```python
.filter(Customer.name.ilike(like, escape='\\'))
```

The code uses an ORM (appears to be SQLAlchemy) which automatically uses parameterized queries. The `like` variable is passed as a parameter, not concatenated into the SQL string, preventing injection.

2. Input Sanitization:

```python
term = term.replace('\\', r'\\').replace('%', r'\%').replace('_', r'\_')
```

This sanitizes LIKE wildcards by escaping:

- `%` (matches any sequence of characters)
- `_` (matches any single character)
- `\` (the escape character itself)

3. Escape Character Usage:

```python
.filter(Customer.name.ilike(like, escape='\\'))
```

The `escape='\\'` parameter tells the database to treat `\` as an escape character, so escaped wildcards are treated as literal characters rather than pattern matching operators.

```python
def search_by_name(self, payload: schemas.SearchByNameRequest) -> list[schemas.SearchResult]:
    Customer = models.Customer
    Account = models.BankAccount

    term = payload.name.strip()
    term = term.replace('\\', r'\\').replace('%', r'\%').replace('_', r'\_')
    like = f"%{term}%"

    q = (
        self.db.query(
            Account.account_number.label('account'),
            Customer.name.label('customer_name'),
            Account.balance.label('balance'),
            Account.account_type.label('account_type'),
        )
        .join(Customer, Customer.id == Account.customer_id)
        .filter(Customer.name.ilike(like, escape='\\'))
        .order_by(Customer.id.asc(), Account.id.asc())
        .limit(payload.limit)
    )
    rows = q.all()

    return [schemas.SearchResult(        You, 26 minutes ago • search api
        account=r.account, customer_name=r.customer_name,
        balance=r.balance, account_type=r.account_type
    ) for r in rows]
```

The Vulnerable Code

Sept 26, 2025

Let's examine a function meant to search for customers. Despite apparent security measures, a critical flaw exists:

```
# User input from the payload
term = payload.name.strip()

# Developer attempts to manually "sanitize" the input
term = term.replace('\\', r'\\\\').replace('%', r'\%').replace('_', r'\_')

# User input is directly formatted into the query string
like = f"%{term}%"  # ⚠️ DANGER: Manual string formatting

# The formatted string is used in the database query
q = (
    self.db.query(...)
    .join(...)
    .filter(Customer.name.ilike(like, escape='\\'))  # ⚠️ VULNERABLE
    ...
)
```

Why This Approach is Dangerous

Two Critical Mistakes:

1. Manual String Formatting: Using f-strings to build SQL patterns directly mixes SQL logic with untrusted user input.

2. Flawed Manual Sanitization: Attempting to clean input manually is error-prone and incomplete.

Potential Attack Scenarios:

- ✓ **Data Leakage:** Bypass sanitization logic to access unauthorized data
- ✓ **Denial of Service:** Craft complex queries that overload the database
- ✓ **Logic Bypass:** Manipulate query conditions to access restricted information

The Secure Solution

Trust the framework! Let SQLAlchemy handle security through proper parameterised queries:

```
# Get the raw search term from the user
search_term = payload.name.strip()

# Let the ORM handle the rest securely!
q = (
    self.db.query(...)
    .join(...)
    .filter(Customer.name.contains(search_term))  # ✅ SECURE
    .order_by(...)
    .limit(...)
)
```

How This Works:

SQLAlchemy's `.contains()` method automatically creates parameterized queries (e.g., `WHERE name LIKE %s`) and safely handles user input as separate parameters, preventing any malicious code execution.

Security Best Practices

❌ Never Do This

- Manual string concatenation
- f-string formatting for SQL
- DIY input sanitization
- Trusting user input

✅ Always Do This

- Use ORM methods (.contains(), .filter())
- Let frameworks handle parameterization
- Validate input format and type
- Follow security-first principles

Golden Rules for Secure Database Operations

✓ Never trust user input

✓ Always use parameterized queries

✓ Let your ORM do the heavy lifting

Step 3: Adding Input Validation

PYTHON

Prevent attackers from sending malicious or invalid data by validating inputs at the entry point.

Python Team (Pydantic & FastAPI)

Field Validator Breakdown

`@field_validator('name')`

This decorator tells Pydantic to apply custom validation specifically to the `name` field.

`@classmethod`

The validator must be a class method that receives:

- `cls`: The model class
- `v`: The field value to validate

Security Benefits

1. **Input Sanitization**: Automatically strips whitespace
2. **Length Limits**: Prevents excessively long inputs that could cause DoS
3. **Type Safety**: Ensures `name` is always a string
4. **Required Fields**: Prevents empty/null values
5. **Early Validation**: Catches invalid data before it reaches your business logic

This validation happens **before** the data reaches your `search_by_name` method, providing an additional layer of security on top of the SQL injection protections you already have.

```python
class SearchByNameRequest(BaseModel):
    name: str
    limit: int = 50

    Qodo Gen: Options | Test this method
    @field_validator('name')
    @classmethod
    def _name_nonempty(cls, v: str) -> str:
        v = v.strip()
        if not v:
            raise ValueError("name must not be empty")
        if len(v) > 64:
            raise ValueError("name too long (max 64).")
        return v
```

What is Input Validation?

Think of it as a bouncer at the door of our application 🚪

Before we let any data in to be processed, we check it to make sure it's safe, valid, and in the format we expect.

We can't assume data coming from users or other systems is safe. We must always verify it first.

Analogy:

A vending machine only accepts specific coins. If you try to use a button or foreign currency, it rejects it. That's input validation!

Why is This So Important?

The Dangers of Unchecked Input

If we don't validate input, we open the door to serious problems:

- **Application Crashes:** Sending a text message where a number is expected can cause the program to fail.
- **Data Corruption:** Invalid data can lead to incorrect calculations, broken user profiles, and a messy database.
- **Major Security Holes:** Attackers can send malicious data to:
    - Steal sensitive information (SQL Injection)
    - Run unauthorized code
    - Crash the entire system by sending huge amounts of data (DoS)

Our Solution: Pydantic & FastAPI

At FIL, our Python team uses a powerful combination of tools to handle this automatically and efficiently:

FastAPI

The web framework that receives the incoming requests

Pydantic

A data validation library that acts as our "bouncer"

How it works: We create a "blueprint" (called a Model) that defines exactly what the incoming data should look like. Pydantic then automatically checks every request against this blueprint.

Our Solution: Pydantic & FastAPI

Custom Validator

Sometimes, just checking the type isn't enough. We need custom rules.

```
@field_validator('name') @classmethod def _name_nonempty(cls, v: str) -> str: v =
v.strip() if not v: raise ValueError("name must not be empty") if len(v) > 64: raise
ValueError("name too long (max 64).") return v
```

- **@field_validator('name'):** "For the name field, run these special checks"
- **v = v.strip():** Input Sanitisation - removes accidental spaces
- **if not v:** Required Field Check - ensures name isn't empty
- **if len(v) > 64:** Length Limit - prevents excessively long names

Security Benefits Summary ✅

This simple piece of code provides multiple layers of protection:

Input Sanitization

Automatically cleans the data

Length Limits

Prevents attacks using excessively long data

Type Safety

Ensures a name is always a string

Required Fields

Stops empty values from causing errors

Early Validation: All these checks happen before the data even reaches our main business logic. We stop bad data right at the front door!

Key Takeaways

- **Golden Rule:** Never, ever trust user input. Treat all incoming data as potentially unsafe.
- **Validate at Entry Point:** Perform checks as soon as data enters the application.
- **Be Specific:** Clearly define the data you expect (type, length, format).
- **Use Your Tools:** Leverage frameworks like FastAPI and Pydantic to make validation robust and easy.

Thank You!

Any Questions?

Conclusion & Key Learnings

Congratulations! You've successfully built a secure API endpoint.

Congratulations! You've successfully built a secure API endpoint.

SQL Injection Defense

Never trust user input. Parameterized queries treat user data as data, not executable code, neutralizing injection threats.

Proactive Input Validation

Fail-fast approach validates size, format, and type at the entry point, protecting against invalid data and adding security layers.

Framework Power

Modern frameworks like Spring Boot and FastAPI provide powerful, built-in security tools. Use them correctly to save time and improve security.

Final Takeaway

Security isn't an afterthought. It's built in, layer by layer, starting with how you handle the very first byte of data a user sends you.