# Week 2 Python Security Exercises

## Application Security Engineering Track

December 2025

| | |
|---|---|
| **Exercise 3** | Pig Latin SQL Sanitizer |
| **Exercise 8** | URL Path Traversal Detector |
| **Focus Areas** | String transformation, Injection attacks |
| **Test Cases** | ~60 per exercise (LeetCode-style) |

# Exercise 3: Pig Latin SQL Sanitizer

## Security Concept

String transformation and demonstrating why input sanitization fails

## Inspired By

**Python Workout** Ch 3, Exercise 5-6 (Pig Latin translator, pages 29-33)
**Full Stack Python Security** page 218 (why input sanitization is dangerous)

## Description

Write a function `pig_latin_sanitize(sql_query)` that transforms a SQL query string into Pig Latin while preserving SQL keywords (SELECT, FROM, WHERE, etc.). This demonstrates why simple string transformations DON'T work for security. The function should also detect if the "sanitized" query still contains exploitable patterns.

## ■■ Critical Learning Point

**This exercise teaches WHY input sanitization fails.** As stated in *Full Stack Python Security* page 218: "Input sanitization is always a bad idea because it is too difficult to implement." You'll discover that even after transforming SQL injection payloads to Pig Latin, they can still be exploitable when interpreted by the database.

## Function Signature

```
def pig_latin_sanitize(sql_query: str) -> dict:
    """
    Transform SQL query to Pig Latin (demonstrating failed sanitization).

    Args:
        sql_query: SQL query string to transform

    Returns:
        Dictionary with 'transformed_query' and 'still_exploitable' boolean

    Example:
        >>> pig_latin_sanitize("SELECT * FROM users WHERE name='admin'")
        {
            'transformed_query': "SELECT * FROM usersway WHERE
amenay='adminway'",
            'still_exploitable': False
        }

        >>> pig_latin_sanitize("SELECT * FROM users WHERE id=1 OR 1=1")
        {
            'transformed_query': "SELECT * FROM usersway WHERE idway=1way ORway
1way=1way",
```

```
            'still_exploitable': True
        }
    """
    pass
```

## Pig Latin Transformation Rules

- **Words starting with vowels (a, e, i, o, u):** Add "way" to the end
- **Words starting with consonants:** Move first letter to end and add "ay"
- **SQL keywords (SELECT, FROM, WHERE, etc.):** Preserve exactly as-is (no transformation)
- **Numbers:** Add "way" to the end
- **Special characters (', ", =, etc.):** Preserve as-is

## SQL Keywords to Preserve (Case-Insensitive)

SELECT, FROM, WHERE, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, TABLE, INTO, VALUES, SET, AND, OR, NOT, JOIN, INNER, OUTER, LEFT, RIGHT, ON, GROUP, BY, ORDER, HAVING, UNION, ALL, DISTINCT, AS, LIKE, IN, BETWEEN, IS, NULL, LIMIT, OFFSET

## Test Cases (60 total)

- 20 tests: Valid SQL queries with various complexity
- 15 tests: SQL injection attempts (verify transformation doesn't remove threat)
- 10 tests: Edge cases (empty queries, single-word queries, keywords only)
- 10 tests: Mixed-case SQL keywords (SELECT vs select vs SeLeCt)
- 5 tests: Unicode and special characters in SQL strings

## Example Transformations

| Original Query | Pig Latin Transformation | Still Exploitable? |
|---|---|---|
| `SELECT * FROM users` | SELECT * FROM usersway | No |
| `admin` | adminway | No |
| `password` | asswordpay | No |
| `1 OR 1=1` | 1way ORway 1way=1way | Yes (logic preserved) |
| `UNION SELECT` | UNION SELECT | Yes (keywords preserved) |
| `'; DROP TABLE--` | ';way OPDRay ABLETay-- | Depends on SQL parser |

## Detecting Exploitability After Transformation

After transforming the query, check if these patterns still exist:

- **Boolean logic preserved:** "OR", "AND" keywords with numeric comparisons
- **UNION attacks:** UNION SELECT keywords still present
- **Comment injection:** -- or /* */ comment markers
- **Stacked queries:** Semicolons separating multiple statements
- **String manipulation:** Quote characters that could break context

## Implementation Notes

- Use regular expressions to identify SQL keywords (case-insensitive)
- Split query into tokens (words, numbers, operators, keywords)
- Transform only non-keyword tokens according to Pig Latin rules
- After transformation, scan for preserved attack patterns
- Return both the transformed query AND exploitability assessment
- Use tabs for indentation (not spaces)

## Learning Objective

This exercise teaches a critical security principle: **sanitization is inadequate defense**. Even sophisticated string transformations can fail because:

- The sanitizer must understand *all* ways an interpreter could parse input
- Databases, browsers, and other systems have complex parsing rules
- Attackers can encode payloads in ways sanitizers don't anticipate
- Proper defense: **parameterized queries**, **output escaping**, and **input validation**

# Exercise 8: URL Path Traversal Detector

## Security Concept

Detecting path traversal attacks in URLs

## Inspired By

**Python Workout** Ch 3 (String manipulation, pages 25-38)
**Hacking APIs** fuzzing chapter
**Secure by Design** input validation patterns

## Description

Write a function `detect_path_traversal(url_path)` that analyzes a URL path for directory traversal attempts (`../`, `..\`, URL-encoded variants) and normalizes safe paths.

## Function Signature

```
def detect_path_traversal(url_path: str) -> dict:
    """
    Detect path traversal attacks in URL paths.

    Args:
        url_path: URL path component to analyze

    Returns:
        Dictionary with is_malicious, attack_patterns, and normalized_path

    Example:
        >>> detect_path_traversal("/api/../../../etc/passwd")
        {
            'is_malicious': True,
            'attack_patterns': ['PARENT_DIRECTORY_TRAVERSAL'],
            'normalized_path': None,
            'severity': 'CRITICAL'
        }
    """
    pass
```

## Attack Patterns to Detect

| Pattern | Description | Severity |
|---|---|---|
| `../` | Relative path traversal (Unix) | CRITICAL |
| `..\` | Windows-style traversal | CRITICAL |

| | | |
|---|---|---|
| `%2e%2e%2f` | URL-encoded ../ | CRITICAL |
| `..%252f` | Double URL-encoded | CRITICAL |
| `/etc/passwd` | Absolute path to sensitive file | CRITICAL |
| `/etc/shadow` | Shadow password file | CRITICAL |
| `C:\Windows\System32` | Windows system directory | CRITICAL |
| `....//....//etc` | Obfuscated traversal | HIGH |

## Test Cases (60 total)

- 20 tests: Known traversal patterns
- 15 tests: URL-encoded traversal attempts
- 10 tests: Safe paths that should normalize correctly
- 10 tests: Mixed attack patterns
- 5 tests: Unicode and multi-byte encoding tricks

## Expected Behavior

- Detect both forward slash (/) and backslash (\) traversal attempts
- Identify URL-encoded patterns (%2e, %2f, etc.)
- Recognize double-encoded attacks
- Flag absolute paths to sensitive system files
- For safe paths, return normalized version (e.g., '/api/users' remains '/api/users')
- For malicious paths, return None for normalized_path

## Common Sensitive File Paths to Detect

- **Unix/Linux:** /etc/passwd, /etc/shadow, /etc/hosts, ~/.ssh/id_rsa, /root/.bash_history
- **Windows:** C:\Windows\System32\config\SAM, C:\Windows\System32\drivers\etc\hosts
- **Web servers:** /var/www/.htpasswd, /var/log/apache2/access.log
- **Application configs:** /app/config/database.yml, .env files

## Implementation Notes

- Use urllib.parse.unquote() to decode URL-encoded strings

- Check for multiple encoding attempts (double/triple encoding)
- Use os.path.normpath() carefully—it can be bypassed
- Maintain a blacklist of sensitive file paths
- Count consecutive '../' patterns (excessive traversal)
- Use tabs for indentation (not spaces)

## Example Test Cases

```
# Test Case 1: Basic traversal
>>> detect_path_traversal("/api/../../../etc/passwd")
{'is_malicious': True, 'attack_patterns': ['PARENT_DIRECTORY_TRAVERSAL',
'SENSITIVE_FILE_ACCESS'], 'normalized_path': None, 'severity': 'CRITICAL'}

# Test Case 2: URL-encoded traversal
>>> detect_path_traversal("/api/%2e%2e%2f%2e%2e%2fetc%2fpasswd")
{'is_malicious': True, 'attack_patterns': ['URL_ENCODED_TRAVERSAL',
'SENSITIVE_FILE_ACCESS'], 'normalized_path': None, 'severity': 'CRITICAL'}

# Test Case 3: Safe path
>>> detect_path_traversal("/api/v1/users/123")
{'is_malicious': False, 'attack_patterns': [],
'normalized_path': '/api/v1/users/123', 'severity': 'SAFE'}

# Test Case 4: Windows-style traversal
>>> detect_path_traversal("..\..\..\Windows\System32\config\SAM")
{'is_malicious': True, 'attack_patterns': ['WINDOWS_TRAVERSAL',
'SENSITIVE_FILE_ACCESS'], 'normalized_path': None, 'severity': 'CRITICAL'}

# Test Case 5: Double URL encoding
>>> detect_path_traversal("/api/..%252f..%252fetc%252fpasswd")
{'is_malicious': True, 'attack_patterns': ['DOUBLE_ENCODED_TRAVERSAL',
'SENSITIVE_FILE_ACCESS'], 'normalized_path': None, 'severity': 'CRITICAL'}
```

# General Implementation Notes

- **Use tabs for indentation** (as requested)
- **Each exercise should have ~60 test cases** following LeetCode-style format
- **Create companion Dev.to blog posts** for each exercise explaining the security concepts
- **No reference implementations** in the challenge files—only function signatures and docstrings
- **Test files separate** from challenge files (use pytest format)
- **Focus on real-world applicability**—these patterns appear in actual security assessments

## Learning Objectives

- Understand why input sanitization fails as a security control
- Practice advanced string manipulation with Python (Pig Latin transformation)
- Learn to detect injection attack patterns (path traversal)
- Develop security mindset: think like an attacker to defend better
- Build reusable security functions for real AppSec work
- Prepare for AppSec interview questions on input validation

## Connection to Your Career Goals

These exercises directly support your transition to Application Security Engineering by:

- **Building portfolio projects** demonstrating security expertise
- **Preparing for technical interviews** at Trail of Bits, NCC Group, GitLab, and Stripe
- **Developing code for your P2P open source project** (secure code dataset curation)
- **Demonstrating offensive security skills** (understanding attack patterns)
- **Complementing your enterprise threat modeling experience** at Intel with practical exploitation knowledge
- **Creating blog content** (Dev.to posts showcase your expertise to employers)

## Why Exercise 3 and Exercise 8 Work Well Together

These exercises complement each other by covering two fundamental AppSec concepts:

- **Exercise 3 (Pig Latin Sanitizer):** Teaches what NOT to do—why sanitization fails
- **Exercise 8 (Path Traversal Detector):** Teaches what TO do—proper input validation

- Both involve string manipulation and pattern recognition
- Both prepare you for real penetration testing scenarios
- Combined, they demonstrate depth in understanding secure coding principles

## Complete Source Citations

| Resource | Relevant Content | Pages/Sections |
|---|---|---|
| Python Workout (2nd Ed.) | String manipulation, Pig Latin | Ch 3 (pp. 25-38), Ex 5-6 (pp. 29-33) |
| Full Stack Python Security | Input sanitization failures | p. 218 |
| Secure by Design | Input validation patterns | pp. 246-248 |
| Hacking APIs | Fuzzing techniques | Chapter 9 |
| 48-Week Curriculum | Week 2: SQL injection fundamentals | pp. 5-8 |
| OWASP | Path traversal attacks | Testing Guide v4.0 |