LAB 6: Fuzzing for Input Validation Bugs

Objective:

To explore fuzz testing as a security and quality assurance technique by identifying and fixing bugs in input processing functions using Python and the Hypothesis fuzzing library.

Duration:

90 minutes

Software Requirements:

- Python 3.8+

Folder Structure:

PES1UG23CSXXX/

├ processor.py → Given: The buggy code that you'll test and fix

├ test_processor.py → **Complete the code in this using hypthesis**

Learning Outcomes

By the end of this lab, students will:

- Understand the purpose and methodology of fuzz testing.
- Apply property-based fuzzing using Hypothesis.
- Detect and analyse bugs caused by edge-case inputs.
- Improve code robustness through defensive programming.

Note: Students may be randomly called for a presentation after completing the lab. Please be prepared to discuss your code, bugs you found, and how you fixed them.

Introduction:

What is Fuzzing?

Fuzzing (or fuzz testing) is a **dynamic testing technique** where you feed **random, unexpected, or malformed data** to a program in order to:

- Discover bugs or crashes
- Uncover security vulnerabilities
- Ensure robustness against edge cases

Why Use Fuzzing?

- Manual tests often miss **rare edge cases**.
- Unit tests test what you think might go wrong.
- Fuzzers find what you **never imagined** could go wrong.

Example: What Can Go Wrong?

Consider this function:

```
def parse_int_list(s):
    return [int(x) for x in s.split(',')]
```

Works fine for:

```
parse_int_list("1,2,3")
```

But what about:

```
parse_int_list("1,,3")      # empty element
parse_int_list("one,two")   # ValueError
parse_int_list("")          # []
parse_int_list(None)        # TypeError
```

Fuzzers can generate such inputs automatically and expose hidden bugs like:

- Crashes (exceptions)
- Incorrect behavior
- Security issues (e.g., injection, DoS)

What is Hypothesis?

Hypothesis is a **property-based fuzz testing tool for Python**.

Instead of writing individual test cases, you describe **properties** of your function (e.g., it should not crash) and let Hypothesis generate dozens or hundreds of **random inputs** to verify those properties.

Example: Normal Unit Test vs Hypothesis

Traditional test:

```
def test_sanitize():
    assert sanitize_string("!hello!") == "hello"
```

Hypothesis Test:

```
from hypothesis import given, strategies as st
from utils import is_palindrome

@given(st.text() | st.none())
def test_is_palindrome_no_crash(s):
    try:
```

```
        is_palindrome(s)
    except Exception as e:
        assert False, f"is_palindrome crashed with: {e}"
```

You write the behaviour; Hypothesis tries to break it.

How Hypothesis Works

1. You write a test function using the @given decorator.
2. You specify **input types** (strategies) like st.text() or st.integers().
3. Hypothesis:
    - Automatically generates thousands of test cases.
    - Minimizes inputs to find the smallest failing case.
    - Logs failing inputs for debugging and replay.

Common Strategies in Hypothesis

Strategy	What it does
st.text()	Generates random Unicode strings
st.integers()	Random integers
st.lists(st.integers())	Lists of integers
st.booleans()	True/False values
st.dictionaries(keys, values)	Random dicts

You are now ready to execute this lab!

GOAL: Use fuzzing to test and fix real input validation bugs, and reflect on what you found.

NOTE: The screenshots must be pasted into a document and sent in PDF format.

Naming convention: SRN_NAME_LAB6

Deliverables:

1. PDF Document with the screenshots and Reflections
2. FIXED processor.py code and completed test_processor.py code

PUT THE FILES(fixed processor.py, test_processor.py and pdf) IN A ZIP FOLDER (SRN_NAME_LAB6.zip) and submit.

STEPS:

Step 1: Installing dependencies:

a. *pip install hypothesis* AND
b. pip install pytest

Step 2: Run the buggy processor.py:

Command: python processor.py

Provide the screenshot of the output (SS1)

Step 3: Complete the code in test_processor.py using hypothesis:

Goal: Use Hypothesis to uncover more edge cases automatically for all three functions.

- Use @given(st.text() | st.none()) to generate a wide range of inputs, including None.
- Import and test the following functions:
    - sanitize_string
    - parse_int_list
    - reverse_words

Step 4: Run the test_processor.py :

Command: pytest test_processor.py **Or**
python -m pytest test_processor.py (if pytest is installed but still showing not recognised)

Provide the screenshot of test cases failing (SS2).

Step 5: Fix the buggy processor.py code:

Goal: Harden the functions against all unexpected or invalid inputs, especially those discovered through fuzz testing.

Run the fixed processor.py – Command : python processor.py

Provide the screenshot of the output (SS3).

Step 6: Re-Run the test_processor.py wrt to fixed processor and take the ss of the output:

Command: pytest test_processor.py **Or**
python -m pytest test_processor.py (if pytest is installed but still showing not recognised)

Provide the screenshot of the output (SS4).

Reflection:

1. How did Hypothesis help?
2. What would you use Fuzzing in CI/CD Pipelines?
3. What do you observe from the screenshots SS2a and SS2b? Justify your answer.