Lab 6: Fuzzing for Input Validation Bugs

C Kaustubh — SRN: PES1UG23CS154 — Section : C

Execution Screenshots

Step 2: Initial Run (Valid Inputs)

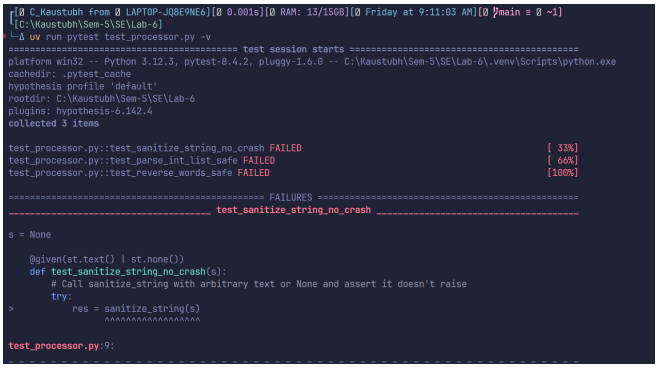
```
┌[Ø C_Kaustubh from Ø LAPTOP-JQ8E9NE6][Ø 0.229s][Ø RAM: 13/15GB][Ø Friday at 9:09:25 AM][Ø  main ≡]
└[C:\Kaustubh\Sem-5\SE\Lab-6]
└─Δ uv run .\processor.py
==== Running sanitize_string ====
Enter a string with special characters (!,@,#,$,%): Hello@World!
Sanitized String: HelloWorld

==== Running parse_int_list ====
Enter a CSV of integers (e.g. 1,2,3,4): 1,2,3,4
Parsed Integer List: [1, 2, 3, 4]

==== Running reverse_words ====
Enter a sentence without punctuation: hello world
Reversed Words Sentence: olleh dlrow
```

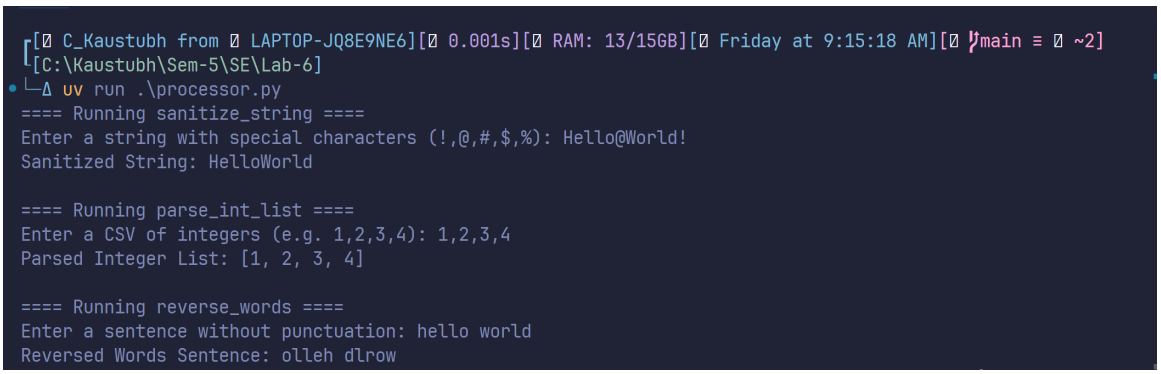
Observation: The buggy code works correctly with valid inputs (Hello@World!, 1,2,3,4, hello world).

Step 4: Tests Failing on Buggy Code



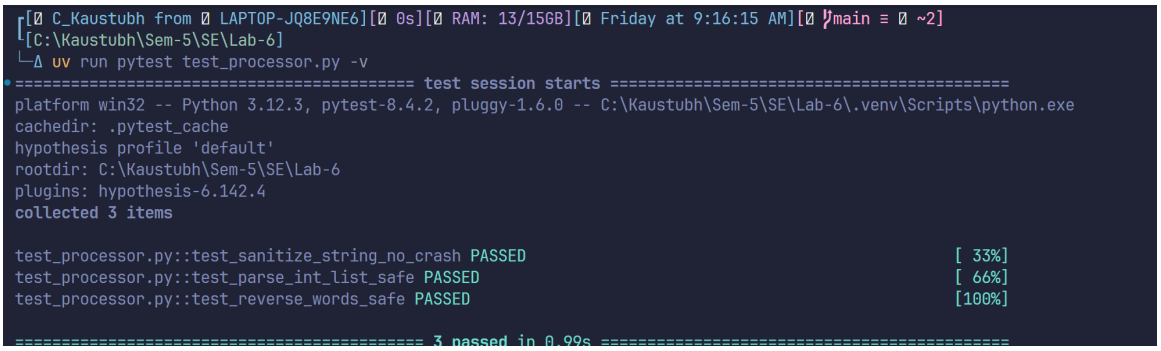
Bugs Found: All 3 tests failed. Hypothesis discovered that functions crash with `None` inputs, empty strings, and invalid data types. Errors: `AttributeError`, `ValueError`.

Step 5: Fixed Code Running



Fix Applied: Added None checks, type validation, and exception handling to all three functions.

Step 6: All Tests Passing



Result: All 3 tests passed after fixes. Code now handles hundreds of edge case inputs gracefully.

Reflections

1. How did Hypothesis help?

Hypothesis automatically generated thousands of test cases including edge cases (`None`, empty strings, Unicode). It found bugs in seconds that manual testing missed, minimized failing inputs for easy debugging, and forced defensive programming practices.

2. What would you use Fuzzing in CI/CD Pipelines?

Use fuzzing for: (a) Automated regression testing on every commit, (b) Security vulnerability detection (SQL injection, DoS), (c) API endpoint testing with malformed requests, (d) Pre-deployment quality gates to block buggy releases, (e) Data validation for parsers and input sanitizers, (f) Continuous security monitoring with nightly fuzz campaigns.

3. What do you observe from screenshots (Before vs After)?

Before (Tests Failing): All 3 tests failed. Hypothesis exposed critical bugs with `None` and empty string inputs causing `AttributeError` and `ValueError`. The code made unsafe assumptions and only worked for "happy path" inputs.

After (Tests Passing): All 3 tests passed. The fixed code handles edge cases gracefully with proper validation, type checking, and error handling. Hundreds of random inputs tested successfully without crashes.

Justification: The transformation proves that (1) Original code was fragile and production-unsafe, (2) Manual testing alone is insufficient as it misses edge cases, (3) Fuzz testing reveals hidden vulnerabilities automatically, (4) Defensive programming is essential for robustness. This demonstrates why fuzzing must be integrated into development workflows to build secure, reliable software.