Project: Find and Fix Bugs with Fuzzing

Secure System Engineering and Management

Due date: Monday, April 6, 11:59pm
Format: Code, video, report
Weight: Part of the Projects component (40% total across five projects)

Overview

This project has two parts.

- **Part 1:** Use coverage-guided fuzzing to discover, reproduce, diagnose, and fix memory-safety bugs in a buggy PNG-processing target.
- **Part 2:** Design two small libFuzzer-based property-testing targets of your own and demonstrate them running.

The Part 1 target is based on `libpng` and is packaged in a containerized local OSS-Fuzz workflow so that everyone uses the same build and runtime environment. The goal is to practice the core workflow used in modern software security engineering:

- run a fuzzer against a target,
- interpret sanitizer crash reports,
- trace the bug to its root cause in source code,
- patch the bug correctly, and
- verify that the crashing input no longer crashes without breaking valid behavior.

The goal of Part 2 is to practice fuzzing for *properties*, not just crashes: defining an invariant, turning raw bytes into structured values, and using libFuzzer creatively to find violations.

What You Are Given

You are provided with a project bundle containing:

- a buggy course-owned fuzz target based on `libpng`,
- an OSS-Fuzz project configuration for local Docker-based fuzzing,
- a small PNG seed corpus to help the fuzzer reach valid parsing paths,
- a fuzz harness and intentionally seeded memory-safety bugs,
- helper scripts for building, fuzzing, reproducing crashes, and post-fix regression testing, and
- a `student_targets/` directory with starter templates for Part 2.

You should not replace the provided Part 1 target with a different target. Your work should focus on understanding, fixing, and validating the bugs in the provided code.

Learning Objectives

By completing this project, you will:

- run a containerized fuzzing workflow with a real parser target,
- interpret AddressSanitizer reports for memory-safety bugs,
- reproduce crashes from specific inputs,
- diagnose and patch the required use-after-free bug, with an optional second bug available for extra credit,
- understand how bug discovery can be sequential,
- design small property-based fuzzing targets using libFuzzer, and
- explain how target structure affects what a fuzzer can and cannot discover efficiently.

Part 1

Task: Setup

You must use the provided Docker/OSS-Fuzz workflow for Part 1. Native local builds outside the containerized environment are **not supported** for Part 1.

Step 1: Get the Code

Use the project package provided on Canvas. Extract it and enter the repository root, `ssem-project4-fuzzing`. This project root should contain at least:

- `oss-fuzz/`
- `seed_corpus/`
- `scripts/`
- `student_targets/`

Step 2: Install Prerequisites

You need:

- Git (only if using the Git repository distribution)
- Docker Desktop (Mac/Windows) or Docker Engine (Linux)
- Python 3

Confirm the following work in a terminal:

```
git --version
docker --version
python3 --version
```

Step 3: Build the Project

From the project root, run:

```
./scripts/build.sh
```

This script:

- builds the local OSS-Fuzz project named `libpng-course`,
- builds the fuzz target, and
- installs the provided PNG seed corpus in the correct location.

If the build script asks whether to use a saved image or cached base images, always select **N**. Use the fresh rebuild path.

The helper scripts explicitly use `python3`. If your system also has a `python` command, ignore it and use the provided scripts exactly as written.

Step 4: Run the Fuzzer

From the project root, run:

```
./scripts/fuzz.sh 30
```

This runs the fuzzer for 30 seconds. You may increase the time if needed. With the provided seed corpus, the fuzzer should usually expose the required bug quickly and print an AddressSanitizer report.

Finding a bug quickly is **expected behavior** in this assignment. In many cases, the required bug appears within a few seconds. If you see an AddressSanitizer crash early, that usually means your setup is working correctly.

Step 5: Reproduce an Input

To reproduce a crash on a specific input, run:

```
./scripts/reproduce.sh seed_corpus/pngnow.png
```

The provided seed inputs (for example, `pngnow.png` and `pngtest.png`) are valid PNG files that help the fuzzer reach deeper parsing logic. In some cases, one of these seed inputs may already trigger a bug immediately. In other cases, the fuzzer may mutate a seed input and produce a new crashing input. For Part 1, use *whichever file reliably reproduces the bug you are analyzing*:

- a provided seed input, or
- a new crash file produced during fuzzing.

Step 6: Post-Fix Regression Check

After fixing the required bugs, run:

```
./scripts/test_valid.sh
```

This is the provided regression test for valid PNG behavior. It verifies that the provided valid PNG inputs still execute successfully after your fixes. You should run this before submission.

Debugging Common Issues

- **Docker not running:** Start Docker Desktop or the Docker daemon, then rerun the commands.
- **Permission denied on scripts:** Run `chmod +x scripts/*.sh`.
- **Seed corpus not used:** Re-run `./scripts/build.sh`. The build script installs the seed corpus automatically.
- **Build succeeds but fuzzing finds nothing:** Confirm that `fuzz.sh` prints `Using seed corpus: course_png_fuzzer_seed_corpus.zip`.
- **Docker build fails with "not enough free space" or apt errors:** Your Docker environment may be out of disk space. Try `docker system prune -a -f` and `docker builder prune -a -f`, then rerun `./scripts/build.sh`.

Task: Find and Fix the Required Seeded Bug

This is a debugging and code-repair task.

You must use fuzzing to identify and fix the **required seeded bug** in the provided target. A second seeded bug is also present and may become visible after the first is fixed; that second bug is **optional extra credit**.

A complete Part 1 solution requires more than simply running the fuzzer: you must understand the crash, locate the buggy code, implement a correct fix, and verify that your fix preserves expected behavior.

You should implement your fixes in:

```
oss-fuzz/projects/libpng-course/libpng/course/course_vuln.c
```

Do not replace the provided target with a different target, and do not substantially rewrite the provided fuzzing infrastructure.

What Counts as a Correct Fix?

A correct fix:

- removes the memory-safety error,
- modifies the buggy target logic rather than simply deleting large parts of the code,
- does not reject all inputs indiscriminately, and
- preserves normal behavior on the provided valid PNG inputs, as checked by `./scripts/test_valid.sh`.

Task: Short Writeup

Keep the Part 1 writeup concise.

For the **required bug**, answer the following in **2–4 sentences total**:

1. What input reproduced the bug?
2. What sanitizer error and source location were reported?
3. What was the root cause, and how did you fix it?
4. How did you verify the fix?

Then write **one short paragraph total** addressing:

1. What role did the seed corpus play in helping the fuzzer reach interesting behavior?
2. If you attempted the extra-credit second bug, what changed after the first bug was fixed?

Optional Extra Credit for Part 1

A second seeded bug is present in `course_vuln.c`. If you discover, reproduce, diagnose, patch, and validate that second bug as well, you may earn extra credit.

If you attempt the extra-credit bug, include a short additional note answering:

1. What input reproduced the extra-credit bug?
2. What sanitizer error and source location were reported?
3. What was the root cause, and how did you fix it?
4. How did you verify the fix?

Part 2: Build Two Property-Based Fuzz Targets

In Part 2, you will build **two small libFuzzer targets of your own**. This part is intentionally more open-ended and is meant to push you to think creatively about fuzzing *properties*, not just memory crashes. Each target should:

- consume raw bytes from libFuzzer,
- convert those bytes into some structured input,
- call code you wrote, and
- check a property that should always hold.

Your job is to design two targets with different fuzzing behavior:

- **Target A:** a property that is relatively easy for the fuzzer to violate or exercise,
- **Target B:** a property that is meaningfully harder for the fuzzer to violate or exercise.

For both targets, the property should be framed as an **invariant** about your program. A good mental model is a statement of the form

$$\forall x,\ P(x) \to Q(x),$$

meaning: for all inputs x satisfying some precondition $P(x)$, your program's behavior $Q(x)$ should hold. For example, if your target library implemented binary search trees, a reasonable property might be: "for all trees T and inserted values i, if T is a binary search tree, then `insert(T, i)` should also be a binary search tree." A bug in insertion would violate that invariant.

In other words, the goal is not merely to write code that crashes. The goal is to express a correctness property that should always hold and then use fuzzing to find inputs that violate it if your implementation is wrong.

Important Restrictions

- You must create **two distinct** target programs.
- You may **not** reuse the examples shown in lecture, including the exact "`bad!`" examples or the arithmetic property example shown on the slides.
- The targets should be your own work and should not be copied from QuickCheck/libFuzzer tutorials.
- The goal is not to create the biggest target possible. Small, focused, well-explained targets are better.

Suggested Subject Domains

You do **not** have to use these, but these are good kinds of programs to build targets around:

- small data structure implementations,
- language parsers, formatters, or pretty-printers,
- string normalization or cleanup routines,
- encode/decode or serialize/deserialize helpers,
- small arithmetic or numeric helper libraries,
- tiny standalone open-source utilities or packages.

Suggested Property Ideas

You do **not** have to use these exact properties, but these are the right level of ambition:

- round-trip property: encode then decode should recover the original value,
- normalization/idempotence: applying a cleanup/normalize function twice should equal applying it once,
- parser consistency: two equivalent interpretations should agree,
- data structure invariants: a transformation should preserve length, ordering, or membership,
- semantic consistency: two implementations of the same intended behavior should match.

Where to Put Your Part 2 Targets

Place your Part 2 code in:

```
student_targets/
```

Starter templates are provided there. You may duplicate and modify them to create your two targets.

Part 2 Environment and Installation Notes

Part 2 is intended to be lighter-weight than Part 1, and the default expectation is that you will compile your Part 2 targets locally with a libFuzzer-capable `clang++`.

If your local toolchain does not support libFuzzer, then you will need a compatible LLVM/Clang installation that includes:

- a working `clang++` with libFuzzer support, and
- access to the header `<fuzzer/FuzzedDataProvider.h>`.

If your local compiler does not work, consult the course staff early. If you use a non-default setup, document it clearly in your writeup.

Using `FuzzedDataProvider`

You are encouraged to use `FuzzedDataProvider` to turn raw fuzzer bytes into structured values such as integers, booleans, strings, or small arrays. This is often much cleaner than manually slicing the input buffer by hand.

To use it, include:

```
#include <fuzzer/FuzzedDataProvider.h>
```

Two useful references are:

- https://github.com/google/fuzzing/blob/master/docs/split-inputs.md#fuzzed-data-provider
- https://google.github.io/oss-fuzz/getting-started/new-project-guide/

Note that using `FuzzedDataProvider` often makes it much easier to define structured properties, but it also means the raw fuzz inputs are no longer meant to correspond to an externally meaningful file format. That is fine for this part of the assignment.

Starter Template

You may copy this into a file such as `student_targets/target1_property_fuzzer.cc` and adapt it.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <string>
#include <fuzzer/FuzzedDataProvider.h>

// TODO: replace this with your own buggy logic.
static std::string normalize_word(const std::string& s, bool lowercase) {
  std::string out = s;
  if (lowercase) {
    for (char& c : out) {
      if (c >= 'A' && c <= 'Z') c = c - 'A' + 'a';
    }
  }
  return out;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  FuzzedDataProvider fdp(data, size);

  bool lowercase = fdp.ConsumeBool();
  std::string s = fdp.ConsumeRandomLengthString(32);

  std::string once = normalize_word(s, lowercase);
```

```
  std::string twice = normalize_word(once, lowercase);

  // TODO: replace with your own property.
  // Example shape only:
  // normalization should be idempotent
  // assert(once == twice);

  (void)once;
  (void)twice;
  return 0;
}
```

Compile and Run Instructions for Part 2

Assuming your target is in a file named `student_targets/target1_property_fuzzer.cc`, compile it with:

```
clang++ -g -O1 -fno-omit-frame-pointer \
  -fsanitize=fuzzer,address \
  student_targets/target1_property_fuzzer.cc -o target1_fuzzer
```

Then run it for 30 seconds:

```
./target1_fuzzer -max_total_time=30
```

Do the same for your second target:

```
clang++ -g -O1 -fno-omit-frame-pointer \
  -fsanitize=fuzzer,address \
  student_targets/target2_property_fuzzer.cc -o target2_fuzzer

./target2_fuzzer -max_total_time=30
```

If your environment requires an equivalent libFuzzer-enabled C++ compiler, document exactly what you used in your writeup.

Note on local compilation On some systems (especially macOS), the default `clang++` may not include a working libFuzzer runtime. If you see errors mentioning missing `libclang_rt.fuzzer`, you may need to install a version of LLVM/Clang with libFuzzer support.

Constraints for Part 2

- Each target must be in its own source file.
- Each target must define a `LLVMFuzzerTestOneInput` harness.
- Each target must derive structured values from raw fuzzer bytes.
- Each target must check a nontrivial property using `assert`, `abort`, or another clear failure mechanism.
- Each target must be deterministic and should terminate quickly.
- Each target should be small: as a guideline, keep each target to roughly **100 lines or fewer**, excluding comments and blank lines.

- At least one of your two targets should actually expose a property violation within the provided time budget.
- The second target should be meaningfully different in difficulty or structure from the first.

Part 2 Writeup

For Part 2, write **one short section per target** answering:

1. What does the target do?
2. What invariant/property are you checking? State it clearly, ideally in the form "if $P(x)$, then $Q(x)$."
3. How do raw bytes get turned into structured values?
4. Why did you expect this target to be easy or hard for the fuzzer?
5. What happened in practice?

Also include a small table with **three runs per target** (30 seconds each is fine). For each run, report:

- whether a failure was found,
- approximate time to first failure, if any,
- and one short sentence comparing Target A and Target B.

Part 2 Video Demonstration

You must submit a short video demonstrating both targets.

Requirements:

- Length: **3–6 minutes** total.
- Show your code for both targets.
- Show compilation commands.
- Show at least one fuzzing run for each target.
- The video must include **at least one live demonstration of a harness finding a bug/property violation**.
- Briefly explain the property being tested and what happened.
- Upload the video to YouTube and submit a **publicly accessible YouTube link**. The link must work without requiring course staff to request access.

A simple screen recording with narration is sufficient. The video does not need to be polished.

Submission Instructions

Submit the following on Canvas as one zipfile:

Part 1 Deliverables

In a subdirectory called `Part1`:

- your patched `course_vuln.c` file from `oss-fuzz/projects/libpng-course/libpng/course/course_vuln.c` **or** a unified patch/diff against the original file,
- your short writeup (PDF), and
- a directory named `reproducers/` containing the file that reproduces the required bug in the original code, stored as `reproducers/bug1_input`.

The reproducer file is a raw binary input. It is **not** a screenshot or pasted text. It should be the exact file you use with `./scripts/reproduce.sh`. After your fix is applied, this same reproducer file should no longer trigger a sanitizer failure.
If you attempt the extra-credit second bug, you may additionally include `reproducers/bug2_input`.

Part 2 Deliverables

In a subdirectory called `Part2`:

- `target1_property_fuzzer.cc` (which was in the `student_targets` directory in your setup)
- `target2_property_fuzzer.cc` (which was in the `student_targets` directory in your setup)
- your short writeup (PDF)
- a **public YouTube link** to your video demonstration, given in a file called `video_link.txt`

Do not submit Docker images, corpora generated during fuzzing, or unrelated build artifacts.

Grading

This project is graded using a combination of automated checks and manual review.

Component	Weight
Part 1 required bug: working reproducer, diagnosis, and fix	25%
Part 1 valid PNGs still work	10%
Part 1 short writeup quality	10%
Part 2 Target A design and implementation	15%
Part 2 Target B design and implementation	15%
Part 2 analysis/writeup quality	10%
Part 2 video demonstration (clarity and completeness)	10%
Optional extra credit: second Part 1 bug reproduced, diagnosed, and fixed	+5%

Correct code is necessary but not sufficient. Clear reasoning, validation, and explanation matter.

Academic Integrity

You may discuss high-level ideas with classmates, but all code, debugging, writing, and video explanation must be your own. Do not share completed solutions, crash inputs with explanations, patched source files, or finished Part 2 targets.

Use of Generative AI Tools

Students are permitted to use generative AI tools (e.g., large language models) to assist with aspects of this project, including:

- understanding sanitizer reports, compiler errors, or unfamiliar C/C++ syntax,
- debugging code or exploring alternative bug fixes,
- clarifying how fuzzing, libFuzzer, or toolchain setup operates,
- brainstorming candidate properties for Part 2, and
- improving the clarity of written explanations.

However, all submitted work must reflect your own understanding and reasoning. In particular:

- You may not submit code, patches, targets, or explanations that you do not understand.
- You may not outsource the core debugging, repair, target design, or interpretation of results to an AI system.
- You should be able to explain what the required Part 1 bug was, why your fix is correct, how your Part 2 targets work, and why your properties make sense. If you attempted the extra-credit bug, you should be able to explain that as well.

You are responsible for the correctness of any code, reproducer inputs, writeups, and video explanations you submit, regardless of whether an AI tool was used to help generate them. We may ask questions about your fixes, reproducer inputs, Part 2 targets, or debugging process to assess understanding.

This project emphasizes debugging, reasoning, validation, creativity, and explanation. Over-reliance on generative tools without understanding may negatively affect grading, especially in categories related to correctness, target design, and written or verbal explanation.

Notes on Fuzzing Engine Choice

Part 1 uses `libFuzzer` in the provided workflow because it integrates smoothly with the local OSS-Fuzz setup and is easier to run consistently across different laptop environments through Docker. Part 2 also uses `libFuzzer`, but in a much smaller and more open-ended way. AFL++ is an important modern standalone fuzzer and may be discussed in class, but all required grading for this project uses libFuzzer-based workflows.

Appendix: Expected Output Examples

Successful build

A successful build should end with output that looks broadly like:

```
[course] Done. Output:
...
course_png_fuzzer
course_png_fuzzer_seed_corpus.zip
llvm-symbolizer
```

Expected fuzzing behavior

A correct fuzzing run may quickly print an AddressSanitizer report. For example, it may contain lines like:

```
ERROR: AddressSanitizer: ...
SUMMARY: AddressSanitizer: ...
```

Seeing a sanitizer failure early in Part 1 is normal and usually indicates that your setup is working correctly.