CS-412 Software Security Lab 2

Fuzzing Lab Report
Spring Semester 2025

Project: Tmux

Luca Di Bello (SCIPER 367552) Federico Villa (SCIPER 386986)
Noah El Hassanie (SCIPER 404885) Cristina Morad (SCIPER 405241)

Submitted: May 8, 2025

Abstract

In this lab, we integrate and evaluate a fuzzing harness for the `tmux` terminal multiplexer using Google's OSS-Fuzz infrastructure. We first establish a baseline by measuring line coverage of the existing `tmux_fuzzer` both with and without the provided seed corpus, observing similar results in both cases.

We then identify two major code regions not exercised by the baseline harness, implemented targeted harness improvements to cover those regions, and finally triage a crash uncovered in the session-detach logic, proposing a patch and assessing its exploitability.

1 Introduction

Fuzzing is a proven technique for uncovering memory-safety and logic bugs in C/C++ code. In this assignment we chose `tmux`, an open-source terminal multiplexer for Unix-like systems, as our target because it (1) has a relatively small OSS-Fuzz integration (only one harness), (2) shows very low runtime coverage, and (3) is widely used daily by developers and DevOps engineers across countless systems. By improving its fuzzing harness, we aim both to increase `tmux`'s coverage under OSS-Fuzz and to demonstrate how targeted harness modifications can uncover real bugs in a piece of critical infrastructure.

2 Methodology

All experiments were performed on Ubuntu 22.04 LTS with Docker 20.10 and Python 3.10. We maintain a local fork of OSS-Fuzz under `forks/oss-fuzz`. Two helper scripts (`run_w_corpus.sh` and `run_wo_corpus.sh`) each run a 4 h fuzzing campaign (seeded vs. unseeded) and then produce coverage reports. Each script follows the same high-level steps:

1. **Configuration.** At the top of the script we set:
   - `PROJECT=tmux`, `HARNESS=input-fuzzer`, `ENGINE=libfuzzer`
   - `SANITIZER=address` (or `undefined` for uninstrumented runs)
   - `REBUILD=true` controls whether to wipe previous build artifacts
   - `RUNTIME=14400` (fuzz for 4 h) and `FLAGS="-max_total_time=$RUNTIME -timeout=25 -print_final_stats=1 -artifact_prefix=./crashes"`.
2. **Clean build directory.** If `$REBUILD` is true, we remove the entire `forks/oss-fuzz/build` directory:

   ```
   rm -rf "$OSS_FUZZ_DIR/build" || true
   ```

   This ensures no stale build artifacts remain.
3. **Apply patch (unseeded only).** In `run_wo_corpus.sh` we apply `submission/part_1/remove_seed_corpus.patch` so that the OSS-Fuzz build scripts no longer package any initial seeds:

```
git apply submission/part_1/remove_seed_corpus.patch
```

4. **Build OSS-Fuzz image and fuzzers.**

```
cd "$OSS_FUZZ_DIR"
python3 infra/helper.py build_image "$PROJECT" --pull
python3 infra/helper.py build_fuzzers --sanitizer "$SANITIZER" "$PROJECT"
```

5. **Prepare corpus directory.** Both scripts use `build/work/$PROJECT/fuzzing_corpus` as the input corpus:
   - *Seeded run:* leave whatever files OSS-Fuzz has placed there.
   - *Unseeded run:* delete and recreate it empty:

```
rm -rf "$CORPUS_DIR" || true
mkdir -p "$CORPUS_DIR/crashes"
```

6. **Run the fuzzer (4 h).**

```
python3 infra/helper.py run_fuzzer \
  --engine "$ENGINE" \
  --corpus-dir "build/work/$PROJECT/fuzzing_corpus" \
  "$PROJECT" "$HARNESS" -- $FLAGS
```

7. **Stop Docker.** After fuzzing we clean up any running containers:

```
docker stop "$(docker ps -q)" || true
```

8. **Export the corpus.** Timestamp and zip the resulting corpus into `experiments/⟨ts⟩_w_corpus.zip` (or `_wo_corpus.zip`) for later analysis.
9. **Generate coverage report.** We rebuild the fuzzers with coverage instrumentation:

```
python3 infra/helper.py build_fuzzers --sanitizer coverage "$PROJECT"
```

Then invoke OSS-Fuzz's coverage tool:

```
python3 infra/helper.py coverage \
  --corpus-dir "build/work/$PROJECT/fuzzing_corpus" \
  --fuzz-target "$HARNESS" \
  "$PROJECT" &
```

We poll `build/out/$PROJECT/report` (up to 5 min) until the HTML is ready.

10. **Export coverage.** Finally, we stop any remaining containers again and copy the fully generated coverage report from `build/out/$PROJECT/report` into `submission/part_1/⟨ts⟩_coverage_{w,wo}_corpus`.

3 Part 1: Baseline Evaluation

3.1 With Seed Corpus

List the exact build/run commands and point to `run_w_corpus.sh`.

3.2 Without Seed Corpus

List the exact build/run commands and point to `run_wo_corpus.sh`.

3.3 Coverage Comparison

Discuss coverage percentages and key observations.

4 Part 2: Coverage Gaps

By analyzing the OSS-Fuzz introspector, we observed that several regions are not covered by the current fuzzer. Two significant uncovered regions are `client.c` and `server.c`. In the following subsections, we justify their relevance and explain the shared limitations of the current fuzzing harness that prevent them from being covered.

4.1 Region A: `client.c` – Justification of Significance

The `client.c` file implements the logic for launching a tmux client, connecting to the tmux server via a UNIX socket, and sending user commands for execution. This file is crucial because any malformed command-line input or unexpected interaction with the server could lead to vulnerabilities or instability. Testing this area is essential to ensure the robustness of the client-side logic, particularly because it processes user input.

4.2 Region B: `server.c` – Justification of Significance

The `server.c` file contains the core logic for accepting client connections, managing sessions,

and dispatching commands. It includes the entry point for the server loop and handles critical functionality such as authentication, command execution, and process management. Bugs or vulnerabilities in this region could be exploited by malicious clients to crash the server.

Shared Explanation of Coverage Shortcomings

The existing fuzzing harness, `input-fuzzer`, operates within a simulated tmux environment by directly creating a mock window and pane, and parsing raw input as if it were typed into an active terminal. However, this harness does not instantiate a real tmux client or server, nor does it set up socket-based communication between the two.

As a result, any code related to the actual startup of the client process (`client.c`) or the server's handling of connections and command dispatching (`server.c`) lies entirely outside the execution scope of the current harness. These components are only triggered in a full client-server lifecycle, which is not simulated or exercised by the current fuzzing setup. Therefore, the input-fuzzer is fundamentally limited to user-interface-level input processing, leaving the network and process-management layers untested.

5 Part 3: Fuzzer Improvements

5.1 Improvement 1 (Region A)

Describe changes, refer to `improve1/run_improve1.sh`, and summarize coverage delta.

5.2 Improvement 2 (Region B)

Describe changes, refer to `improve2/run_improve2.sh`, and summarize coverage delta.

6 Part 4: Crash Analysis

Detail crash reproduction (`run_poc.sh`), ASAN log snippet, root cause, proposed patch, and exploitability.

7 Conclusion and Future Work

Summarize achievements and outline possible next steps.