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 conducted on a Debian system running Linux kernel 6.1 (LTS), using Docker 28.1.1 and Python 3.11. 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 hour fuzzing campaign (seeded vs. unseeded fuzzing) and then produce coverage reports. The scripts are located in `submission/part_1` and are invoked as follows:

Configuration Both scripts are highly configurable and can be run with different parameters. The following variables are set at the top of each script:

- `PROJECT=tmux` - OSS-Fuzz project name
- `HARNESS=input-fuzzer` - name of the project's fuzzing harness to run
- `ENGINE=libfuzzer` - engine to use (tmux supports `libfuzzer`, `afl++` and `honggfuzz`)
- `SANITIZER=address` - Sanitizer to use. Available options: `address` (ASan), `undefined` (UBSan), `none` (disabled). [1]
- `REBUILD=true` - controls whether to rebuild the OSS-Fuzz image from scratch.
- `RUNTIME=14400` - fuzzing time in seconds (by default 4 hours).
- `FLAGS="-max_total_time=$RUNTIME -timeout=25 -print_final_stats=1 -artifact_prefix=./crashes"` - fuzzer engine flags (refer to the documentation of the engine for more details).
- `OSS_FUZZ_DIR="forks/oss-fuzz/build"`

(*optional*) – this variable is set by default to match the current repository structure. However, it is still explicitly defined to allow advanced users to override it if they wish to customize the script for their own needs (e.g. use a different fork of OSS-Fuzz).

Clean build directory If `$REBUILD` is true, we remove the entire `forks/oss-fuzz/build` directory as follows:

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

This ensures no stale build artifacts remain and that the build process starts from a clean slate.

Apply patch (unseeded only) In `run_wo_corpus.sh` we remove the seed corpus from the Docker image and the build script to ensure the fuzzer starts with no initial input files, thus avoiding any bias introduced by pre-seeding the fuzzer with a starting corpus.

To achieve this, we apply a patch `remove_seed_corpus.patch` using the `git apply` command, which modifies the project's `Dockerfile` and `build.sh` to execlude the seed corpus during the build process:

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

Build OSS-Fuzz image and fuzzers To prepare the fuzzing environment, we use the OSS-Fuzz helper script `helper.py` to build the Docker image and compile the fuzzers with the specified sanitizer (e.g., ASan, UBSan, none).

Before building the Docker image, the script checks whether rebuilding is required by evaluating the `$REBUILD` variable (refer to section 2 for more details on the script configuration). If it is set to `true`, the following command is executed to build the Docker image for the configured project:

```
cd "$OSS_FUZZ_DIR"
python3 infra/helper.py build_image \
  "$PROJECT" --pull
```

The `--pull` flag ensures that the latest version of the OSS-Fuzz base Docker image is used. This is essential to ensure compatibility with the latest updates, bug fixes, and dependency changes.

Regardless of the `$REBUILD` variable, the script always rebuilds the fuzzers with the specified sanitizer using the following command:

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

This command compiles the fuzzers for the specified project, applying the selected sanitizer to instrument the code for better error detection and reporting. This generates the fuzzing binaries under `build/out/$PROJECT`, which are later needed to run the fuzzing campaigns.

Prepare corpus directory By default, both scripts rely on `build/work/$PROJECT/fuzzing_corpus` as the input corpus directory (refer to section 2 for more details). For seeded runs, this directory is automatically populated by the unpatched `tmux`'s build scripts with files from the official `tmux-fuzzing-corpus` repository. This corpus provides a comprehensive set of input samples to simulate real-world usage scenarios and edge cases across different terminal emulators (currently only iTerm and Alacritty), including various terminal escape sequences and control characters. [2]

On the other hand, during unseeded runs we ensure that the directory containing the seed corpus is empty. Even if the build script and Dockerfile are patched to exclude the seed corpus, the directory could still contain files from previous seeded runs. To ensure a clean state, we delete the `fuzzing_corpus` directory and recreate it empty. In summary:

- *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"
  ```

By default, we configured LibFuzzer to ensure that all crash-inducing inputs are stored within the designated `crashes` subdirectory. This is done by setting the `--artifact_prefix` flag to `./crashes` in the `$FLAGS` variable (refer to section 2 for more details). This allows us to easily access and analyze any inputs that caused

the target program to crash during the fuzzing process.

Run the fuzzer (4 h) To start the fuzzing campaign with the configured fuzzer, we use the `run_fuzzer` command provided by the OSS-Fuzz helper script. We explicitly set the fuzzing engine (e.g., `libfuzzer`), the input corpus directory, the target project, and the specific harness to be used. By default the fuzzer runs for 4 hours (14400 seconds), as specified by the `$RUNTIME` variable.

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

Stop Docker To ensure that all Docker containers are stopped after the fuzzing campaign, we use the following command.

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

This is important to avoid leaving any running containers that may consume system resources or interfere with subsequent runs.

Export the corpus After each fuzzing campaign, we export the generated corpus from the `build/work/$PROJECT/fuzzing_corpus` directory to a zip file in the `experiments` directory, including in the filename the timestamp and the corpus type (seeded or unseeded). This step allows us to keep track of the corpus used in each run and facilitates further analysis or sharing of the corpus within the team.

Generate and export coverage report To generate the coverage report, we first rebuild the fuzzers with the `coverage` sanitizer enabled:

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

Then, we use the OSS-Fuzz's coverage analysis tool, specifying the corpus directory and the target fuzzing harness:

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

By default, the coverage tool starts a local web server to serve the HTML report. To ensure our script remains fully automated and headless, we run the coverage command in the background and poll the output directory (`build/out/$PROJECT/report`) for up to 5 minutes, waiting until the report becomes available. Once generated, the HTML report directory is saved as `<timestamp>_coverage_{w,wo}_corpus` and stored in the `submission` directory.

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 direct 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.

References

[1] Nicholas Marriott. *OSS-Fuzz Configuration for tmux*. https://github.com/google/oss-fuzz/blob/master/projects/tmux/project.yaml. Accessed: 2025-04-25. 2025.

[2] Nicholas Marriott. *tmux-fuzzing-corpus*. https://github.com/tmux/tmux-fuzzing-corpus. Accessed: 2025-04-25. 2021.