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 a Debian system with Linux kernel 6.1 (LTS), Docker 28.1.1, and Python 3.11. We used a local fork of OSS-Fuzz located at `forks/oss-fuzz`. To streamline execution, we developed two scripts—`run_w_corpus.sh` and `run_wo_corpus.sh`—that autonomously perform 4-hour fuzzing campaigns (with and without the seed corpus, respectively) and generate corresponding coverage reports. Both scripts are located in `submission/part_1`. Their internal logic is modularized to share a common sequence of operations, which are detailed in the following subsections.

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`). [**oss-fuzz:tmux_project_yaml**]
- `SANITIZER=address` – Sanitizer to use. Available options: `address` (ASan, *default*), `undefined` (UBSan) or `none` (disabled). [**oss-fuzz:tmux_project_yaml**]
- `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 -ignore_crashes -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. [**tmux:tmux-fuzzing-corpus**]

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
```

To continue fuzzing even after crashes, we pass the `-ignore_crashes` flag. This ensures that libfuzzer keeps running and exploring new paths even after finding a failure. Crashing inputs are still reported and saved in the `crashes` subdirectory, as defined by the fuzzer configuration.

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

We ran a 4-hour fuzzing campaign using the predefined `input-fuzzer` harness and the official tmux seed corpus, with `libfuzzer` as the fuzzing engine and `AddressSanitizer` enabled. This configuration corresponds to the default parameters outlined in section 2.

The entire process, from Docker image preparation to fuzzer execution and coverage report generation, is fully automated by the script `run_w_corpus.sh`, located in `submission/part_1`. This script is intended to be run from the project root and performs automatically all required steps. For a detailed breakdown of its internal logic, refer to section 2.

```
submission/part_1/run_w_corpus.sh
```

The seeded fuzzing campaign completed successfully and produced the following coverage results, summarized in Table 1.

Metric	Coverage
Line Coverage	14.00% (7281/51997)
Function Coverage	24.44% (558/2283)
Region Coverage	13.41% (5481/40874)

Table 1. Coverage with seed corpus after a 4-hour fuzzing run.

3.2 Without Seed Corpus

To ensure comparable results between the two runs, we used the same configuration, including the fuzzing engine, harness, and sanitizer. The only difference between the two campaigns is that we removed the seed corpus from build script by applying the patch `remove_seed_corpus.patch` using the `git apply` utility,and ran the fuzzing with an empty seed corpus directory. For more information on how the seed corpus was excluded, refer to section 2.

Similarly as the seeded run script, the unseeded run script is designed to be run from the root of the project, and it takes care of all necessary steps, including building the patched Docker image, compiling the fuzzers, running the fuzzing campaign, and generating the coverage report:

```
submission/part_1/run_wo_corpus.sh
```

The unseeded fuzzing campaign completed successfully and produced the following coverage results, summarized in Table 2.

Metric	Coverage
Line Coverage	13.94% (7248/51997)
Function Coverage	24.31% (555/2283)
Region Coverage	13.35% (5457/40874)

Table 2. Coverage without seed corpus after a 4-hour fuzzing run.

The results are very similar to those obtained with the seed corpus, with only minor differences across all metrics. This suggests that the fuzzer is able to reach most of the currently covered code paths even without initial inputs. A detailed comparison of both runs is provided in subsection 3.3.

3.3 Coverage Comparison

Table 3 summarizes the per-file differences in coverage between the two fuzzing runs, with and without the seed corpus. The table shows the percentage of lines covered by the fuzzer in each file, along with the difference in coverage between the two runs. Since the coverage is very sparse, we only show the files with at least 0.5% coverage in either run.

From the summary tables in Table 1 and Table 2 and the detailed comparison in Table 3, we can observe the following key points:

- The overall coverage is very similar between the two runs, with only a few files showing differences in coverage.
- Introducing the seed corpus adds just a few dozen lines (≈40 lines; ≈0.13% change in regions/functions coverage).
- Across the entire codebase, the only difference in coverage are observed in the following files. All of them showcased a small increase in coverage with the seed corpus.

  `input.c`: $91.64\% \to 92.31\%$ $\Delta = 0.67$

  `colour.c`: $70.38\% \to 70.78\%$ $\Delta = 0.40$

  `paste.c`: $36.67\% \to 38.33\%$ $\Delta = 1.66$

  `window.c`: $14.40\% \to 15.46\%$ $\Delta = 1.06$

  The overall increase in coverage is minimal, but still provides some insight into the fuzzer's behavior.
- These small deltas confirm that the fuzzer is already able to reach most of the code paths in the `input.c` module, which is responsible for parsing and processing user input. The small increase in coverage in `colour.c`, `paste.c`, and `window.c` suggests that the seed corpus may help explore some additional code paths which are difficult to reach with random inputs alone (e.g. color combinations, paste buffer handling, window management).

Path	$\mathrm{Cov}_{\mathrm{wo}}$ (%)	$\mathrm{Cov}_{\mathrm{w}}$ (%)	Δ (%)
src/tmux/alerts.c	15.18	15.18	0.00
src/tmux/arguments.c	5.82	5.82	0.00
src/tmux/cmd-find.c	6.54	6.54	0.00
src/tmux/cmd-parse.c	26.95	26.95	0.00
src/tmux/cmd-queue.c	21.25	21.25	0.00
src/tmux/cmd.c	26.70	26.70	0.00
src/tmux/colour.c	70.38	70.78	0.40
src/tmux/compat/recallocarray.c	13.89	13.89	0.00
src/tmux/compat/strlcat.c	90.48	90.48	0.00
src/tmux/compat/strtonum.c	96.97	96.97	0.00
src/tmux/compat/tree.h	93.86	93.86	0.00
src/tmux/compat/vis.c	15.69	15.69	0.00
src/tmux/control-notify.c	8.28	8.28	0.00
src/tmux/environ.c	3.68	3.68	0.00
src/tmux/format.c	28.37	28.37	0.00
src/tmux/fuzz/input-fuzzer.c	93.88	93.88	0.00
src/tmux/grid-view.c	69.64	69.64	0.00
src/tmux/grid.c	33.52	33.52	0.00
src/tmux/hyperlinks.c	81.25	81.25	0.00
src/tmux/input.c	91.64	92.31	0.67
src/tmux/layout.c	1.52	1.52	0.00
src/tmux/log.c	8.54	8.54	0.00
src/tmux/notify.c	49.30	49.30	0.00
src/tmux/options.c	25.85	25.85	0.00
src/tmux/osdep-linux.c	12.73	12.73	0.00
src/tmux/paste.c	36.67	38.33	1.66
src/tmux/screen-write.c	64.14	64.14	0.00
src/tmux/screen.c	50.23	50.23	0.00
src/tmux/server-fn.c	2.45	2.45	0.00
src/tmux/style.c	27.27	27.27	0.00
src/tmux/tmux.c	0.79	0.79	0.00
src/tmux/tty.c	1.87	1.87	0.00
src/tmux/utf8-combined.c	38.18	38.18	0.00
src/tmux/utf8.c	26.85	26.85	0.00
src/tmux/window.c	14.40	15.46	1.06
src/tmux/xmalloc.c	80.46	80.46	0.00

Table 3. Per-file line coverage comparison between two `libfuzzer` runs—one with and one without a seed corpus—using the `input-fuzzer` harness compiled with `AddressSanitizer`. Highlighted rows indicate files with different coverage in the two runs.

As `tmux` is a hotkey-based terminal multiplexer, we expected to get high coverage results in both runs as, as soon as the fuzzer identifies a valid hotkey sequence, it can trigger a large number of code paths with minimal effort.

- Overall coverage remains low (~14%), indicating large untested areas. By inspecting the full coverage reports, was discovered that `client.c`, `server.c`, and most `cmd-*.c` modules are not covered using the `input-fuzzer`

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.