

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. The `tmux` project supports only `libfuzzer`, `afl++`, and `honggfuzz`. [1]
- `SANITIZER=address` – Sanitizer to use. The `tmux` project supports the following sanitizers: `address` (`ASan`, *default*), `undefined` (`UBSan`) or `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
-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
```

Listing 1. Bash script to clean the OSS-Fuzz build directory with error handling

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 exclude the seed corpus during the build process:

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

Listing 2. Bash script to apply the patch to remove the seed corpus from the Docker image and build script

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
```

Listing 3. Bash script to build the Docker image for the configured project

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"
```

Listing 4. OSS-Fuzz helper script command to build the project fuzzers with the specified 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"
```

Listing 5. Bash commands used to setup an empty corpus directory for unseeded runs

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
```

Listing 6. Bash command to run a single fuzzing campaign leveraging OSS-Fuzz helper script

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
```

Listing 7. Bash command to stop all running Docker containers

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"
```

Listing 8. Bash command to rebuild fuzzers with coverage sanitizer using OSS-Fuzz helper script

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" &
```

Listing 9. Bash command to generate a coverage report after fuzzing

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
```

Listing 10. Bash command to run an automated fuzzing campaign with seed corpus

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
```

Listing 11. Bash command to run an automated fuzzing campaign without seed corpus

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; $\approx 0.13\%$ 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% \rightarrow 92.31%
 $\Delta = 0.67$

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

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

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

The overall increase in coverage is minimal, but still provides some insight into the fuzzer’s behavior.

Path	Cov_wo (%)	Cov_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/reallocarray.c	13.89	13.89	0.00
src/tmux/compat/strcat.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 (with and without a seed corpus) using the `input-fuzzer` harness compiled with `AddressSanitizer`. Highlighted rows show differing coverage.

- 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).
- Overall coverage remains low ($\sim 14\%$), 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`

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.

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.

4 Part 2: Coverage Gaps

As cited in subsection 3.3, the default fuzzing harness `input-fuzzer` is limited to the user-interface-level input parser, leaving two essential `tmux` components completely untested (0% coverage, refer to Table 3):

- Client-server communication (`client.c` and `server.c`)
- Window and pane management (`cmd*.c`)

modules)

This is expected, as the harness never initializes a real tmux client or server, nor does open any socket-based communication. Instead, `input-fuzzer` simply creates a mock window and pane in-memory, and feeds them raw bytes sequences generated by the fuzzing engine, as shown in Listing 12.

```
// Initialize the window and pane
w = window_create(PANE_WIDTH,
    PANE_HEIGHT, 0, 0
);
wp = window_add_pane(w, NULL, 0, 0);

// ...

// Process input and handle any error
event
input_parse_buffer(wp, (u_char *)data,
    size);
while (cmdq_next(NULL) != 0);
error = event_base_loop(libevent,
    EVLOOP_NONBLOCK);
if (error == -1) errx(1, "
    event_base_loop failed");
```

Listing 12. Core of the ‘input-fuzzer’ fuzzer code, including in-memory window and pane creation

Because no real client or server binary ever runs, the initialization, connection-handling, and command-dispatch code in both `client.c` and `server.c` is never reached. Likewise, none of the `cmd-*.c` handlers (e.g. `cmd-new-window.c`, `cmd-split-window.c`) ever see fuzzed input via the normal client-server path as they are not executed in this environment

4.1 Region A: `client.c` – Why it matters

The `client.c` file implements the tmux client’s startup sequence, socket connection logic, and command-forwarding routines [3]. As the client is the primary interface for users, any malformed command-line flags or unexpected socket errors could crash the client or corrupt its state. Since this module parses untrusted user input and orchestrates external processes (e.g. sends commands to clients connected to the tmux server), fuzzing it is critical to catch edge-case bugs that may lead to instability or vulnerabilities.

4.2 Region B: `server.c` – Why it matters

On the other hand, `server.c` drives the tmux server’s main loop: it accepts and authenticates client connections, manages sessions, and dispatches commands to the appropriate `cmd-*.c` handlers [4].

Vulnerabilities in this layer (e.g. buffer overflows in command parsing, logic errors in session management, etc.) could allow a malicious client to crash or compromise the server completely. Improving the code coverage of this module is crucial to ensure the security and stability of the entire tmux system.

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

This section details the analysis of vulnerabilities within tmux.

The first vulnerability triaged is CVE-2020-27347, the second is CVE-TO-CHANGE.

6.1 Vulnerability Triage: CVE-2020-27347 - Stack Buffer Overflow in SGR Parsing

6.1.1 Vulnerability Description (CVE-2020-27347)

CVE-2020-27347 is a stack-based buffer overflow in tmux versions prior to 3.1c (specifically affecting 2.9 through 3.1b). The vulnerability resides in the `input_csi_dispatch_sgr_colon()` function within `input.c`, which parses colon-separated SGR (Select Graphic Rendition) escape sequences. An attacker capable of writing to a tmux pane can send a specially crafted SGR sequence with numerous or malformed

empty parameters, causing a buffer overflow. This can crash the `tmux` server (Denial of Service) and potentially allow arbitrary code execution.

6.1.2 Proof of Concept (PoC) and Reproduction

A reproducible test was implemented using shell scripts (`run_poc.sh`, `test_vulnerable.sh`, `test_fixed.sh`, and `run_tmux_cve_test.sh`) within a Dockerized `ubuntu:22.04` environment.

Test Approach: The `run_poc.sh` script automates testing by:

1. Building a Docker image with `tmux` build dependencies and the `tmux` source code cloned from GitHub.
2. Running tests for a vulnerable and a fixed version of `tmux` in isolated container instances.

Per-Version Test Steps:

1. **Version Checkout:** The specific `tmux` versions are checked out using `git reset -hard <commit_hash>`:

Vulnerable: version 3.1b
(commit 6a33a12)

Fixed: version 3.1c
(commit 25cae5d)

2. **Compilation:** `tmux` is compiled from source using the same process as in the `tmux` Dockerfile, but without any fuzzer-related flags. The compilation process is as follows:

```
sh autogen.sh && ./configure &&
make -j"${nproc}"
```

Listing 13. Bash to compile `tmux` from source without fuzzer support

3. **tmux Launch:** A detached `tmux` session with a unique name is started using the bash command in Listing 14. This is done to ensure reliable startup in the scripted environment.

```
tmux new-session -d -s
cve_test_session_<PID>
```

Listing 14. Bash to create a new detached `tmux` session

4. **Target Identification:** The script waits for the session to be ready and identifies the target pane's TTY path using the `tmux list-panes` utility.
5. **Payload Delivery:** The SGR escape sequence reported in Listing 15 is written directly to the identified pane TTY to trigger the vulnerability. This simulates a user typing the sequence into the `tmux` pane.

```
\033[:::7::1:2:3::5:6:7:m
```

Listing 15. Payload delivery to the `tmux` pane

6. **Observation & Verification:** After an observation period (e.g., 10 seconds where the script sleeps waiting for a result), the script checks if the `tmux` session and server are still responsive (`tmux has-session`, `tmux ls`). The outcome is compared against the expected behavior (*crash* in version 3.1b, *no crash* in version 3.1c).
7. **Cleanup:** Both the `tmux` test session created earlier, and the `tmux` and server are terminated.

This setup consistently reproduces the crash on the vulnerable version and confirms the fix on the patched version.

6.1.3 Root Cause Analysis

The vulnerability is due to a stack-based buffer overflow in `input_csi_dispatch_sgr_colon()` in `input.c`. This function processes SGR escape sequences that use colons as sub-parameter separators (e.g., for 24-bit color). The function fails to adequately validate the number of these sub-parameters before copying them into a fixed-size stack buffer. An escape sequence with many empty or numerous sub-parameters, such as the PoC `\033[:::7::1:2:3::5:6:7:m`, overflows this buffer.

6.1.4 Fix Discussion

`tmux` version 3.1c fixed this by adding bounds checking in `input_csi_dispatch_sgr_colon()`. This ensures the number of parsed SGR sub-parameters does not exceed the buffer's capacity, preventing the overflow.

6.1.5 Security Implication and Severity

CVE-2020-27347 has a high severity (CVSS v3.x scores typically 7.8-8.8).

- **Denial of Service (DoS):** Most commonly, it crashes the `tmux` server.
- **Potential Arbitrary Code Execution (ACE):** As a stack overflow, it could allow ACE, though modern mitigations (ASLR/PIE) make this more challenging.
- **Attack Vector:** The attack is local, requiring the ability to write to a `tmux` pane's input.

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.
- [3] Nicholas Marriott. *tmux: client.c*. <https://github.com/tmux/tmux/blob/master/client.c>. GitHub repository, accessed 2025-05-05. 2025.
- [4] Nicholas Marriott. *tmux: server.c*. <https://github.com/tmux/tmux/blob/master/server.c>. GitHub repository, accessed 2025-05-05. 2025.