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 15, 2025

Abstract

In this lab, we improve the fuzzing efforts of the `tmux` terminal multiplexer within Google's OSS-Fuzz infrastructure. We first establish a baseline by evaluating the line coverage of the existing `input-fuzzer` harness, both with and without its provided seed corpus, obtaining comparable results.

Following this, we identify two significant code regions in `tmux` uncovered by the baseline fuzzer. To address these coverage gaps, we develop and evaluate two new targeted fuzzing harnesses named `cmd-fuzzer` and `argument-fuzzer`, demonstrating their ability to improve coverage in these previously undertested areas. As these specific fuzzing improvements did not uncover new critical vulnerabilities within the project's timeframe, our crash analysis focuses on a known historical vulnerability. We develop a proof-of-concept for CVE-2020-27347, a stack-based buffer overflow, proceed to analyze its root cause, discuss the implemented fix, and assess its overall security implications.

Introduction

Fuzzing is a proven technique to uncover memory security 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.

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 manage the various fuzzing campaigns for this project we implemented a unified and configurable scripting framework All individual run scripts – baseline evaluations for Part 1, and improved fuzzer runs for Part 3 – share the same core script that handles the common fuzzing workflow. This core script's operational sequence, including environment preparation, building, fuzzing, and reporting, is detailed in the subsections below.

Configuration The script is highly configurable via environment variables set at the top. Key configuration variables include:

- `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 `libfuzzer`, `afl++`, and `honggfuzz` [1].
- `SANITIZER=address` – Sanitizer to use. The tmux project supports `address` (ASan), `undefined` (UBSan), or `none` (dis-

abled) [1].
- `RUNTIME=14400` – fuzzing time in seconds (by default 4 hours).
- `LABEL` – A unique label for the run (e.g., `w_corpus`, `wo_corpus`) used for naming output files.
- `REBUILD_IMAGE=true` – controls whether to rebuild the OSS-Fuzz Docker image from scratch.
- `PATCH_FILE` – Optional path to a patch file to apply.
- `EXPORT_RESULTS=true` – Controls whether to export the generated corpus and coverage report after the fuzzing run.
- `LIBFUZZER_FLAGS` – Fuzzer engine flags (e.g., `-max_total_time=$RUNTIME -ignore_crashes=1`). This flag is used to override the default harness-specific options (e.g. `.option` files)

The script derives several other internal path variables based on these configurations.

Setup and Patching Before performing any builds or fuzzing, the script ensures a clean and correctly configured OSS-Fuzz environment.
- If the local OSS-Fuzz clone does not exist at `$OSS_FUZZ_DIR`, it is cloned from github.com/google/oss-fuzz.
- The script navigates into the `$OSS_FUZZ_DIR` and performs a hard reset and clean (`git reset -hard HEAD && git clean -fdx`) to discard any previous modifications and ensure a clean state.
- A mandatory patch, `oss-fuzz.diff`, located alongside the script in `$SCRIPT_DIR`, is applied to the root of the OSS-Fuzz repository. This patch contains baseline modifications required for the experiments, such as potential adjustments to the project's `Dockerfile` and `build.sh`.
- If the `PATCH_FILE` variable is set and the file exists, this optional patch is applied.

This flexible patching mechanism allows the single script to handle configurations like unseeded runs, where a patch is needed to modify the build process to exclude the seed corpus.

Clean build directory If `$REBUILD_IMAGE` is true, the script removes the entire `$OSS_FUZZ_DIR/build` directory:

```
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 from previous runs and that the Docker image build process starts from a clean slate. Additionally, the script ensures the fuzzer artifact directory exists by creating it if necessary (located at `$OSS_FUZZ_DIR/build/out/$PROJECT/crashes`).

```
git apply 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, the script uses the OSS-Fuzz helper script `infra/helper.py`.

Before building the Docker image, the script checks the `$REBUILD_IMAGE` variable. If set to `true`, the following command is executed within `$OSS_FUZZ_DIR` to build the Docker image for the configured project:

```
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 the latest OSS-Fuzz base image is used.

Regardless of the `$REBUILD_IMAGE` 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 compiles the fuzzers and instruments them with the selected sanitizer. The resulting binaries are placed in `$OSS_FUZZ_DIR/build/out/$PROJECT`.

Corpus Preparation for Build The tmux build process populates the directory `$OSS_FUZZ_DIR/build/work/$PROJECT/fuzzing_corpus` with the seed corpus, if available.

To ensure that unseeded runs start without initial data, the script checks the `REMOVE_CORPUS_BEFORE_BUILD` variable. If true (as it would be for an unseeded configuration), the script removes this directory before the fuzzer build step to ensure a clean state. This is done using the following command:

```
rm -rf "$BUILD_WORK_CORPUS_DIR"
```

Listing 5. Bash command to remove the build-time corpus directory for unseeded runs

For seeded runs, this variable would be set to `false`, allowing the build process to populate the directory with the seed corpus.

Run the fuzzer To start the fuzzing campaign, the script uses the `run_fuzzer` command from the OSS-Fuzz helper script. The input corpus directory used by the fuzzer is `$BUILD_WORK_CORPUS_DIR`, which was computed in the previous steps. The fuzzer runs for the time specified by `$RUNTIME`.

```
python3 infra/helper.py run_fuzzer \
  --engine "$ENGINE" \
  --corpus-dir "$BUILD_WORK_CORPUS_DIR"
      \
  "$PROJECT" "$HARNESS" -- \
  "$LIBFUZZER_FLAGS"
```

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

The configured flags from `$LIBFUZZER_FLAGS`, such as `-ignore_crashes=1` to continue fuzzing after finding crashes and `-artifact_prefix=./crashes/` to store crashes in the designated subdirectory (relative to the fuzzer's working directory, which is handled by `helper.py`), are passed to the fuzzer engine.

Export Results (Corpus and Coverage) If the `EXPORT_RESULTS` variable is set to `true`, the script exports both the generated corpus and the coverage report. A timestamp (`$TS`) and the run's label (`$LABEL`) are included in the output filenames for identification.

Corpus Export: The contents of the fuzzer's working corpus directory (`$BUILD_WORK_CORPUS_DIR`) are copied to a timestamped and labeled directory within `$CORPUS_EXPORT_DIR_BASE`. This directory is then compressed into a zip archive, and the original copied directory is removed.

```
cp -r "${BUILD_WORK_CORPUS_DIR}" "${
    FINAL_CORPUS_EXPORT_PATH}"
(cd "$(dirname "${
    FINAL_CORPUS_EXPORT_PATH}")" && \
 zip -qr "${FINAL_CORPUS_EXPORT_PATH}.
     zip" \
 "$(basename "${
     FINAL_CORPUS_EXPORT_PATH}")" && \
 rm -rf "${FINAL_CORPUS_EXPORT_PATH}")
```

Listing 7. Bash commands to export and zip the generated corpus

Coverage Report Generation and Export: To generate the coverage report, the fuzzers are first rebuilt with the `coverage` sanitizer enabled using `helper.py build_fuzzers`. Then, the `helper.py coverage` command is executed within `$OSS_FUZZ_DIR`, specifying the generated corpus directory (`$BUILD_WORK_CORPUS_DIR`) and the target harness.

```
python3 infra/helper.py build_fuzzers
    --sanitizer coverage "$PROJECT"
python3 infra/helper.py coverage \
  --corpus-dir "$BUILD_WORK_CORPUS_DIR"
      \
  --fuzz-target "$HARNESS" \
  "$PROJECT" &
```

Listing 8. Bash commands to rebuild with coverage and generate report

The coverage command runs in the background. The script then polls the expected output directory (`$OSS_FUZZ_DIR/build/out/$PROJECT/report`) for up to 5 minutes, waiting for the report files to appear. Once generated or the timeout is reached, any background processes and Docker containers associated with the coverage generation are explicitly stopped. Finally, the generated HTML report directory is copied to a timestamped and labeled directory within `$COVERAGE_EXPORT_DIR_BASE`, compressed into a zip archive, and the copied directory is removed (same technique as with the corpus export). After completing these steps, the script changes back to the original experiment root directory.

Part 1: Baseline Evaluation

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 the *Methodology* section.

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 submission folder root and performs automatically all required steps.

```
part_1/run.w_corpus.sh
```

Listing 9. 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.

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 .

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:

```
part_1/run.wo_corpus.sh
```

Listing 10. 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 the next section.

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.

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/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/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/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/style.c	27.27	27.27	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.

`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.

- 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).
  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`, in fact their coverage is 0% in the OSS-Fuzz generaed report.

Part 2: Coverage Gaps

As cited in the previous section, the default fuzzing harness, `input-fuzzer`, is limited to

the user-interface-level input parser, leaving multiple essential tmux components completely untested.

The current harness has a very narrow scope: it only executes low-level input parsing functions with most of the codebase remaining untested. Based on coverage reports from both with-corpus (14.00% line coverage, 24.44% function coverage, 13.41% region coverage) and without-corpus (13.94% line coverage, 24.31% function coverage, 13.35% region coverage) runs, we can see that only input parsing, color formatting code and command parsing areas are covered in the current harness.

Multiple components of the tmux codebase are quite complex and difficult to test under a fuzzer; for example all the code areas that require multiple processes or threads or the use of sockets. A prominent example of such complex regions is the `client` and `server` subsystem: when tmux is launched, it either connects to an existing server or spawns one. The server, as defined in `server.c`, is a background process that handles all active sessions, panes and windows. Each user terminal runs a client process, defined in `client.c`, that communicates with the server using a UNIX-domain socket. Other code areas that are similarly complex might require a multiplexer runtime environment or execution of multiple commands in a command line environment, therefore they might be harder to test at a high level.

Fuzzing these complex code areas would demand creating real sockets, forking processes, and simulating terminal inputs, all of which pushes typical fuzzers beyond practical limits. Mocking a socket or using other fake implementations was not considered, as it would have required testing the functions in isolation and maintaining a separate and complex implementation just to simulate each component. This approach deviates from realistic tmux execution and would therefore lack testing integration bugs that might happen during the communications between client and server. Therefore the original authors preferred to focus on low level code areas that are simpler, more deterministic and do not require complex environments.

Several code areas remain uncovered by the current fuzzing harness, such as:

- Screen and grid management code responsible for handling drawing operations, scrolling, and clipping on each terminal update
- Layout computation code that calculates pane sizes and positions for various tmux layout strategies (main-horizontal, even-horizontal, tiled, etc.)
- Window management code (creation, destruction, and layout of tmux windows and their buffers)
- Terminal capabilities handling
- Style formatting
- Event handling mechanisms

Most of these code regions are areas where users are less likely to supply malformed inputs directly. Therefore, we decided to focus on areas that are primary gatekeepers for user-supplied inputs:

- Command-line argument handling (`arguments.c`)
- Command execution, command parsing logic and individual command modules (`cmd-parse.c` and `cmd-*.c` modules)

This lack of coverage is expected, as the harness never initializes a real tmux client or server, nor does it establish any socket-based communication so it does not test the application end to end. Instead, `input-fuzzer` simply creates a mock window and pane in-memory, and feeds them raw byte sequences generated by the fuzzing engine, as shown in Listing 11.

```
// 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 11. 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. The code in `arguments.c` that handles command-line arguments is also bypassed entirely, representing a significant vulnerable code area that remains untested in the current implementation.

Region A: *Arguments*

The `arguments.c` file implements tmux's command-line argument parsing and handling system. This component plays a critical role in tmux's functionality by:

- Processing all command-line parameters passed when tmux is launched
- Managing socket paths, configuration files, and control mode settings
- Providing utilities to validate and convert argument values
- Creating structured argument collections used throughout the program

This module is particularly important for security testing because it directly processes untrusted user input. Command-line arguments represent a classic attack vector for exploitation, with potential for path traversal, buffer overflows, or format string vulnerabilities if input validation is insufficient.

Our analysis of the coverage reports confirms that the existing fuzzing harness achieves only about 5.8% line coverage of `arguments.c`. Examining the harness code in `input-fuzzer.c` reveals several fundamental reasons for this limited coverage:

- **Initialization architecture**: The fuzzer initializes a minimal tmux environment with just a window and pane object, bypassing the normal program entry point. In the `LLVMFuzzerInitialize` function, it creates basic global structures but never invokes the argument parser.
- **Input delivery method**: The fuzzer sends data directly to `input_parse_buffer()` rather than through the main command-line argument system. While some command parsing may indirectly invoke `args_parse()`, the full structured command-line handling is not properly exercised.
- **Process lifecycle**: LibFuzzer's design maintains a persistent process, calling the `LLVMFuzzerTestOneInput` function repeatedly on the same process. However, command-line arguments are typically processed once during program initialization, making this model ineffective for testing argument handling.
- **Environment isolation**: The fuzzer runs in a controlled container environment without access to the filesystem or socket resources that many argument-handling functions interact with, leaving code paths like `args_check_socket()` and `args_load_cfg()` untested.

Looking at specific uncovered functions in the coverage reports, we observe that critical sections like `args_parse()`, `args_find()`, and `args_escape()` show zero execution counts. Additionally, complex features such as the percentage-based size calculations in `args_percentage()` remain completely unexplored by the fuzzer.

Region B: *Command Parsing*

The `cmd-parse.c` file implements tmux's command parser, which is responsible for converting textual commands into executable structures. This component:

- Processes commands from interactive user input, configuration files, and scripts
- Handles complex language features including quoting, variable expansion, and command chaining
- Validates command syntax and semantics before execution
- Resolves targets for commands (session, window, and pane specifiers)

The command parser is a critical security boundary as it processes input from multiple sources including configuration files that may contain untrusted content. Vulnerabilities in this component could potentially lead to command injection or denial of service issues.

The command parse file is implemented in the code as a *Yacc* (*Yet Another Compiler Compiler*) parser file, so as a `cmd-parse.y` file that will be compiled into a `.c` file. The `cmd-parse.c` has some high level functions that interacts with and dispatches functionality to

various `cmd-*.c` files, each of which implements a specific tmux command. Most of such specific command files are not touched by the current harness even though they represent some of the core operational logic of individual tmux commands and therefore could constitute a prime fuzzing target.

Our coverage analysis shows that despite `cmd-parse.c` having better coverage than most tmux components at approximately 27% line coverage, significant portions remain untested. Examining the harness structure explains these gaps:

- **Missing server context**: The existing harness establishes a basic pane and window environment but lacks the full server infrastructure. The coverage report reveals that code paths requiring interaction with server data structures (like session and client management) show zero execution counts.
- **Limited tokenization**: While the harness does invoke `input_parse_buffer()`, which eventually leads to some command parsing, the randomized fuzzing input rarely satisfies the tokenizer's grammar requirements. Examining uncovered regions in the report shows most advanced parsing features like variable expansion (`${...}`), escape sequences (line 3400-3404), and quoted strings (lines 3419-3438) are never executed.
- **Command structure limitations**: The harness creates a minimal command environment, but many commands in tmux expect to interact with other subsystems. For example, conditional command execution (`%if` blocks) shows zero coverage in the report.
- **Tokenization barriers**: The parser's design expects structured input with properly quoted strings, escape sequences, and command chaining. From the coverage report, we can see that error paths like the ones at lines 3454-3457 are never reached because random inputs fail at earlier validation stages.

Even in more heavily covered sections of the file, the coverage is often limited to simple path validation without exploring deeper conditional branches. For instance, while the basic token reading loop (lines 3290-3448) shows some coverage, most of the specialized command handling branches within it remain untested.

Part 3: Fuzzer Improvements

In this section we describe the modifications and additions made to the fuzzing infrastructure to improve the two (currently) uncovered regions identified in part 2.

Overview

Improvements to the code coverage were achieved by targeting the two specific previously uncovered code regions with two new harnesses.

Improvement of Region A: *Arguments*

This new harness tries to cover the `arguments.c` file which, as already explained in Part 2, is responsible for processing the command-line flags that are passed to the tmux executable. Given its importance and the fact that the old oss-fuzz harness has only a 5.82% coverage on this file we decided to create a harness that specifically target this file.

To fuzz this file we created a completely new harness that tries to call as many functions as possible from this file. The first thing we do in this harness is to parse the bytes given by libfuzzer into 2 variables: `char **argv` and `int args`, which are the command-line arguments that are normally passed to a program. With these 2 variables we are able to call the function `args_from_vector` which will create a `struct args_value *`. Subsequently we used this struct with a `struct args_parse` (which we can create with the bytes given from libfuzzer) to call the `args_parse` function which will return a `struct args *`. This struct is needed for many functions in the `arguments.c` file, e.g. we can use this struct together with bytes from libfuzzer to call `args_print`, `args_has`, `args_get`, `args_first`, `args_count`, `args_value`, `args_first_value`, `args_next_value`, `args_string`, `args_copy`, `args_percentage`, `args_strtonum`. Many of these functions allocate some space on the heap calling `malloc`, so if a function fails or we don't have enough bytes from libfuzzer to call the subsequent

functions then we perform a cleanup calling `free` (or `args_free`) to avoid the LeakSanitizer to give us some errors.

To build the docker image, the harness and run the harness you should execute the script `run_improve1.sh` from the root of this project.

This new harness reach a coverage of 66.62% which is 60% more than the previous harness. The previous harness didn't focus on working with the arguments so the only called functions were `args_parse, args_create, args_free and args_print`. Since our harness focuses on calling as more functions as possible from this file it's trivial to understand the reason why our harness covers much more region.

Further improvements for this file is to target the function `args_make_commands_now` which, as the name suggests, is responsible for the execution of commands. This function could increase the coverage of this file by a lot, since it also calls other 3 functions in `arguments.c` (`args_make_commands_prepare`, `args_make_commands` and `args_make_commands_free`) that were not covered by our harness. We didn't target these functions because unlike the other functions that we targeted, these don't work with `struct args *` but instead with `struct cmd *` and `struct cmdq_item *`.

Improvement of Region B: *Command Parsing*

The improvements of *Region B* targets the parsing of user-provided arguments, command execution and command parsing logic in general. The target is a code area called when commands from the user input, configuration file or command-line arguments are translated into structured commands for execution, therefore it's an area directly in contact with the user input.

As the code inside `cmd-parse.c` is only covered partially in the current harness, and almost all of the other command specific files are not targeted (such as `cmd.c`, `key-bindings.c`, `cmd-bind-key.c`, `key-string.c`, ...), since their functions are never invoked, we decided to improve coverage in these areas. To achieve this, we added calls to several `cmd-parse.c` functions that are normally invoked during client creation, specifically in `client.c`. These functions are directly called by the user input, meaning that any input generated by the fuzzer can be easily and reliably reproduced through a corresponding direct invocation of tmux. This gives the fuzzing test cases of improved area B strong reproducibility.

The harness implementation consists of two main parts:

- Transforming the fuzzer input into a valid input format for the command parser target function, by parsing it into an argument count `argc` and an argument vector `argv`.
- Crafting argument parsing tmux structures, creating commands and parsing them using `cmd_parse_from_arguments`. To ensure reproducibility this parts is created taking inspiration from the command line parsing that happens inside the `client_main` function, which uses directly the user-provided command line arguments.

The call to `cmd_parse_from_arguments` is a high level function that has lots of underlying new function calls. Therefore the fuzzer execution may branch into multiple `cmd-*.c` files, depending on the input provided. This because the `cmd_parse_from_arguments` is the funciton responsible to parsing user-supplied command line arguments and therefore based on the specific command it handles control to the corresponding implementation file, so with that function call we can call a wide range of command-specific code paths in various `cmd-*.c` files.

In addition, the harness must initialize correctly global variables and free up all dynamically allocated data and struct to ensure the lack of errors or unexpected memory leaks.

To run the new harness two flags were added in the `cmd-fuzzer.options` file, the file containing the flags for the fuzzer (*libFuzzer*). One of the flag is `detect_leaks=0` and it pauses the leakage detection. The leakage detection was not taken into account, even if this causes missing memory leaks, because when running the new harness due to `cmd-parse.y` a memory leakage happens only after multiple runs due to the improper cleanup of the `static struct cmd_parse_state parse_state` between successive parsing runs that triggers false positives in leak detection (as the leak detected in the Yacc parser file. These variables are static variables that retain state between parser invoca-

tions. This issue could be solved with either the new flag or by adding a Bison `%destructor` rule. Therefore leak checking was disabled to prioritize identifying critical crashes or bugs. The other flag is an increase of the standard memory usage limit, `rss_limit_mb`, and it is added to avoid that an excessive use of memory (more than the standard `2048 MB` limit) causes the failure of libFuzzer.

To improve the fuzzing efficiency and guide libFuzzer input generation toward meaningful tmux command-line strings a dictionary file, `cmd-fuzzer.dict`, file was created. It contains tmux commands, options, aliases and flags commonly used (for example `"new-window", "split-window", "-h", ...`).

To build and run the new harness the script `run_improve2.sh` should be executed; it clone and applies the adequate changes to the oss-fuzz repository such that the new harness is executed out of the box.

After fuzzing the project with the new harness three times for 4 hours, the line-coverage and function coverage of the key parsing and dispatch code increased in the areas we targeted: `cmd-parse.c` jumped by 13.89% in function coverage to 77.78%(28/36) and by 15.63% in line coverage to 42.58%, reflecting the new parsing paths covered in out harness; `arguments.c`, the file responsible of interpreting command-line arguments, was increased in this harness too from 5.82%(45/773) to 45.54%(352/773) and 47.62% in function coverage; `cmd.c` rose to 39.14% of line coverage and 45.45% of function coverage. Key-handling routines were now exercised in the new harness as:

- `key-string.c` line coverage rose to 30.00%
- `key-bindings` line coverage rose to 56.05%(278/496)

Among individual command modules that were not touched by the previous harness but now are covered by the fuzzer tests:

- `cmd-bind-key.c` and `cmd-set-options.c` achieved a function coverage of 50.00%;
- `cmd-command-prompt.c`, `cmd-confirm-before.c`, `cmd-if-shell.c` and `cmd-confirm-before.c` each reached a function coverage of 25.00%;
- `cmd-display-panes.c` and `cmd-run-shell.c` attained a function coverage of 16.67%.

These modules were triggered by inputs passed to the `cmd_parse_from_arguments` function, corresponding to specific commands that users can issue in tmux to invoke different functionalities.

A further possible improvement for *region B* is to target a broader range of `cmd-*.c` or other specific command modules files by extending the new harness with additional function calls. This would enable coverage of command-releated operations involving windows, layouts, or panes for example. While such files are not invoked automatically by the new harness, executing the fuzzer for an extended period of time (as common done in research and industry scenarios) could further enhance the overall project coverage and include such code area too.

Part 4: Crash Analysis

During the course of this project, efforts were made to uncover new vulnerabilities in `tmux` using the improved fuzzing harnesses developed in Part 3. Unfortunately, within the allocated fuzzing timeframe, no new crashes indicative of distinct bugs were discovered by our updated fuzzers.

As per the assignment guidelines, in the absence of a newly found bug, this section will focus on the reproduction and detailed analysis of a known, pre-existing vulnerability in `tmux`. For this purpose, we have chosen to create a proof of concept for and triage CVE-2020-27347. The following subsections detail our approach to reproducing this vulnerability and an analysis of its root cause, the implemented fix, and its security implications.

Vulnerability Overview: CVE-2020-27347

CVE-2020-27347 is a high-severity (CVSS v3.x score 7.8) stack-based buffer overflow vulnerability in `tmux` versions prior to 3.1c (specifically affecting 2.9 through 3.1b) [2]. The flaw exists in the `input_csi_dispatch_sgr_colon()` function within `input.c`, which parses colon-separated SGR escape sequences. An attacker with local access, capable of writing to a `tmux` pane, can trigger the overflow by sending a crafted SGR sequence with excessive or mal-

formed empty parameters and can crash the `tmux` server (causing *Denial of Service*).

Proof of Concept (PoC) and Reproduction

A reproducible PoC was developed using the bash script `run_poc.sh`. This script leverages `test_vulnerable.sh`, `test_fixed.sh`, and `run_tmux_cve_test.sh` to execute tests within a Dockerized `ubuntu:22.04` environment.

Test Approach: The `run_poc.sh` script automates the 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 patched 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>`:

   version `3.1b`: commit `6a33a12` (vulnerable)
   patch: commit `a868bac`

   Version `3.1b` (commit `6a33a12`) is used to demonstrate the vulnerability. The fix is verified using commit `a868bac`, which introduced the patch and is included in version `3.1c` and later.
2. **Compilation:** `tmux` is compiled from source using:

```
sh autogen.sh && ./configure &&
    make -j"$(nproc)"
```

Listing 12. Bash to compile tmux from source

3. **`tmux` Launch:** A detached `tmux` session with a unique name is started using the bash command in Listing 13. This is done to ensure reliable startup in the scripted environment.

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

Listing 13. Bash to create a new detached tmux sesssion

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 14 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 14. Payload sent 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* after the patch).
7. **Cleanup:** Both the tmux test session created earlier and the tmux server are terminated.

This setup consistently reproduces the crash on the vulnerable version and confirms its absence after applying the patch.

Root Cause Analysis

The vulnerability is due to a stack-based buffer overflow within the `input_csi_dispatch_sgr_colon()` function in `input.c`. This function parses SGR escape sequences that can contain colon-separated numerical parameters (e.g., for specifying 24-bit colors). The PoC payload (refer to Listing 14) provides a series of parameters, many of which are empty. The function correctly identifies empty parameters but, prior to the fix, would increment its parameter counter for each one without sufficiently checking if this count exceeded the allocated size of a local integer array (`p[8]`) used to store these parsed parameters. With enough empty parameters, the counter `n` would exceed the array's bounds. Subsequent attempts to write to or read from `p[n-1]` would then occur out-of-bounds on the stack, leading to the overflow.

```
static void input_csi_dispatch_sgr_colon(...)
{
    /* ... variable declarations and setup ... */
    int p[8];           // Fixed-size stack buffer
    u_int n = 0;        // Parameter count/index

    /* consume escape sequence column by column */
    char *ptr;
    ptr = xstrdup(s);
    while ((out = strsep(&ptr, ":")) != NULL) {
        if (*out != '\0') { // Handle non-empty parameter
            p[n++] = strtonum(out, 0, INT_MAX, &errstr);
            ...
        } else { // Handle empty parameter string ("")
            n++; // VULNERABLE: Increment parameter count for an empty parameter

            // !! MISSING BOUNDS CHECK HERE !!
            // not checking if 'n' exceeds the size of 'p'
        }

        // If 'n' was incremented past nitems(p) in the 'else' above,
        // accessing p[n-1] here (or other uses of p/n later) causes an out-of-
            bounds access.

        /* read buffer 'p' at index 'n-1' */
        log_debug("%s: %u = %d", __func__, n - 1, p[n - 1]);

        ... other code processing parameters using 'p' and 'n' ...
    }
    ... rest of function and cleanup ...
}
```

Listing 15. Vulnerable logic simplified: Stack buffer overflow in empty SGR parameter handling

Fix Discussion

The vulnerability was fixed in `tmux 3.1c` by commit a868bac. The patch modifies the `input_csi_dispatch_sgr_colon()` function within `input.c` by introducing an explicit bounds check – effectively `if (n == nitems(p))` – to ensure that the number of parsed SGR sub-parameters `n` does not exceed the capacity of the buffer `p`. If the limit is reached, the function stops processing further parameters from that sequence, preventing the overflow.

```
  ...
  } else {
    n++;
    if (n == nitems(p)) {
        free(copy);
        return;
    }
  }
  ...
```

Listing 16. Fixed logic: Bounds check added to prevent stack buffer overflow in else statement

Security Implication and Severity

The high-severity `tmux` vulnerability [2] (CVSS v3.x score 7.8) is a stack-based buffer overflow. The memory corruption crash can lead to *Denial of Service*: a crash of the `tmux` server process causes the termination of all user sessions managed by the server. The vulnerability requires local access, specifically the ability of an attacker to write in a `tmux` pane.

References

[1] *OSS-Fuzz Configuration for tmux*. GitHub repository. Google / OSS-Fuzz Project. URL: https://github.com/google/oss-fuzz/blob/master/projects/tmux/project.yaml (visited on 04/25/2025).

[2] *CVE-2020-27347 Detail*. CVE Record. The MITRE Corporation. 2020. URL: https://www.cve.org/CVERecord?id=CVE-2020-27347 (visited on 05/13/2025).